Experimental Performance Analysis of a Self-Driving Vehicle Using High-Definition Maps

Onur Toker
Electrical and Computer Engineering
Florida Polytechnic University
Lakeland, FL 33805
otoker@floridapoly.edu

SoutheastCon 2023 | 978-1-6654-7611-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SoutheastCon51012.2023.10115093

***Abstract*—In this paper, we present our first experimental results on self-driving vehicles using high-definition maps. We will start with the CAN bus based Drive-By-Wire subsystem, on board computers, and the sensors used in this study. Then, we discuss the sensor fusion and vehicle guidance algorithms, and finally present our experimental results. We also have a video recording of our experiment, and its YouTube link is shared in the paper. All of the experimental results and plots presented in the paper, and the video link refer to the same self-driving experiment that we did on the Florida Polytechnic University campus. The autonomous vehicle (AV) methodology adopted in this work has some similarities with the Cruise AV's approach and the use of high definition (HD) maps. The research vehicle used in this work is equipped with radar, lidar, camera, GPS, and IMU sensors, but in this work we use only the GPS, wheel rotation and camera sensors. After presenting our first experimental AV results, we comment on sensor fusion related issues, and possible future steps for improvement.**

***Index Terms*—Self-driving vehicles, Sensor Fusion.**

I. Introduction

In 2019, Florida Polytechnic University received an NSF grant about autonomous vehicles research, and in this paper, we present our first successful self-driving vehicle experimental results, see Fig. 1. We will first present the system architecture, then discuss algorithm selection, and finally comment on possible future steps. The research vehicle used in this work is a Ford Fusion plug-in hybrid, and it has both sensors and actuators at the steering wheel, at the throttle pedal, and at the break pedal. These sensors and actuators are all on a CAN network, and they enable the vehicle's electronic control by using the on-board computer(s).

The research vehicle used in this work has three separate computers, two of which are GPU equipped and have the lambda AI stack, and one SpeedGoat Real-time target machine. There is a CAN bus network connecting all sensors, actuators, and the computers. There is also a high-speed ethernet bus connecting all computers to each other. The GPS, radar, lidar and camera sensors are treated differently, as they are on different high-speed buses. The GPS data volume is not high, and it is currently using a direct USB connection to one of the Intel x86-64 computers. The radar has CAN bus interface, and is it connected to one of the CAN ports of the PCIe add-on of the computer inside the trunk, see Fig. 2 for details. Lidars and cameras have Gigabit ethernet interfaces, and are connected to the Gigabit Ethernet ports of a different PCIe add-on of the same computer inside the trunk. Use of separate PCIe add-on cards for both CAN and Gigabit ethernet provides a much larger bandwidth and lower delay compared to USB based wiring. Basically, the research vehicle's speed is mainly limited by how fast the sensor data can be transferred to the on-board computers, how fast the sensor data can be processed, sensor fusion can be completed and finally how fast these decisions can be sent to the actuators. Therefore, USB based sensors are kept at a minimum.

This work is supported by NSF grant 1919855, and Florida Polytechnic University.

Figure 1. Our first self-driving vehicle experiment demo video, https://www.youtube.com/watch?v=2kEqh4AuMGo

The AV methodology adopted in this work is based on high-definition mapping. The literature on high definition maps is really immense, and it is not possible to provide a comprehensive review in a single paragraph. But, we would like to cite [1] where a lidar based high-definition mapping technique is presented using scan-matching. High-definition map creation is a demanding task, and its automation or semi-automation is of extreme importance. In [2], a machine learning based HD-map creation approach is introduced to streamline this complex process. Use of GPS/GNSS together with HD-maps can be quite effective for self-driving vehicles, see [3]. In [4], a crowd sourcing approach is used for high-definition map creation, again a practical and quite useful idea for streamlining the creating and maintenance of high-

definition maps. We also would like to cite [5] as a related work for use of HD-maps for autonomous robotic systems.

This paper is organized as follows: In Section II, we start with System Hardware. System software is presented in Section III, Sensor Fusion approach is presented in Section IV, and the experimental results are presented in Section V. Finally, we make some concluding remarks in Section VI.

II. System Hardware

Our research vehicle is a Ford Fusion plugin hybrid, and it has total 3 separate on-board computers, see Fig. 2. The first one, is a dual Linux/Windows laptop with Intel i7-11850H, 32GB RAM, 1TB SSD (Linux) + 1TB SSD (Windows), NVIDIA GeForce RTX 3050 Ti. This computer is located inside the vehicle, and is connected to a docking station. The CAN bus access is through a Kvaser USB cable with two seperate CAN bus endpoints. The remaining two computers are inside the trunk, and they are all automotive grade vibration resistant, and passively cooled devices. One of them is a SpeedGoat Mobile Real-Time target machine with Intel i7 CPU and four separate CAN bus interfaces. This device allows control models built in Simulink to be executed in real-time, and its is used mainly for running non-AI algorithms. The second computer inside the trunk is a desktop with Intel i9-9900 hexa-core, 64GB RAM, 1TB SSD (Linux) + 1TB SSD (Backup/Data), Quadro RTX-A4000 16GB GDDR6 140W GPU, two Gigabit Ethernet ports, two software programmable RS232/422/485 ports, a PCIe add-on providing four additional Gigabit Ethernet ports for high speed cameras, and lidars, another PCIe add-on providing four low-latency CAN bus end points. Both Linux computers have Ubuntu 20 LTS distribution of Linux, and have the lambda AI stack installed for CPU accelerated AI processing. There is also an on-board router connecting all of these computers to each other over a dedicated Gigabit ethernet.

Figure 2. On the right, the Intel x86-64 laptop with NVIDIA GPU located inside the vehicle, and on the left a more powerful Intel x86-64 desktop with NVIDIA GPU, a SpeedGoat Mobile Real-Time target machine (Intel x86-64 based), and an a Samlex inverter located inside the trunk.

A. CAN bus network, sensors and actuators

Our research vehicle is equipped with several sensors and actuators. These include a torque sensor and an angular position sensor connected to the steering wheel, a motor connected to the steering wheel; position sensors connected to the throttle pedal, and the break pedal, and electromechanical actuators connected to the throttle pedal, and the break pedal. All of these sensors and actuators are on a CAN bus, and there is a breakout box, see Fig 3, that allows multiple DB9 CAN connectors to be connected to the system. In Fig. 3, there are two CAN cables: (1) A Kvaser USB CAN cable with a single end-point, and can be used by a laptop, and (2) A regular CAN bus cable (purple color) connected to an STM32 ARM Cortex-M4F microcontroller board operating in bare-metal mode, i.e. no operating system just an embedded C/C++ code running on the controller.

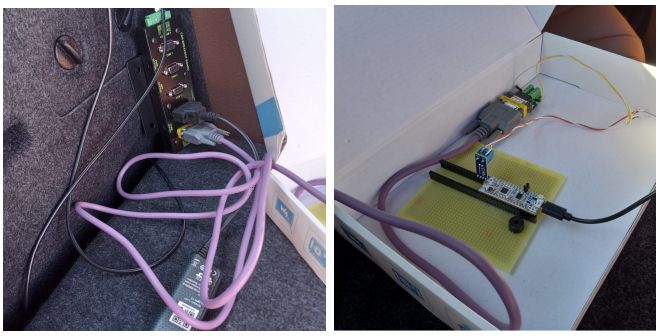

Figure 3. On the right, CAN breakout box that provides five DB9 CAN connectors, a Kvaser USB CAN cable with a single end point (black color), and a regular CAN bus cable (purple color). On the left, a STM32 board running an embedded C/C++ code for low-speed longitudinal control.

III. System Software

In this section, we will summarize our system software and the development workflow adopted by our research team. To explore different design ideas, we do use different simulators that are based on MATLAB/Simulink or Python. Ideas proven to be effective in a simulation study are considered for prototyping experiments. During the prototyping phase, we mostly use Python to implement different ideas and only when tested and proven to be really effective, then implement in C/C++ and execute either on the on-board computers as standalone apps, or on the STM32 ARM Cortex-M4F boards in bare-metal mode, see Fig. 3. Since we are still at the relatively at early stages of our research, majority of our code is still in Python. SpeedGoat Real-time target machine, being Simulink based, is also used for rapid prototyping but it is mainly for non-AI related control loops. All AI related decision making currently use GPU acceleration with the help of lambda AI stack.

A. Software for CAN network access

At the lowest level of abstraction, we need the ability to send a CAN message, and to read a CAN message with the possibility of hardware filtering. Our research vehicle's CAN network can be used in the following three different ways:

1) An STM32 ARM Cortex-M4F microcontroller operating in bare-metal form, and using `HAL_CAN_AddTxMessage` and `HAL_CAN_GetRxMessage` functions which are

simply wrappers for the CAN peripheral access using special function registers. Basically, below this level we have assembly instructions. Full details are available in the STM32Cube documentation.

2) Linux kernel has built-in support for CAN devices using the so called SocketCAN framework. SocketCAN is similar to the Berkeley socket API, and implements the CAN device drivers as network interfaces. Therefore, SocketCAN devices can be used, read or written similar to a network socket. This greatly simplifies the software development, but it limits us to computers running Linux only.
3) There are also third-party companies like Kvaser, and they have their own API for reading or writing CAN messages, and configuring the CAN network. These third party companies provide the same API for both Linux and Windows platforms, which makes AV research and prototyping easier.

In this research, we used all of these CAN network access techniques at various different stages. The first one is probably the best for final deployment, but the last one is more practical during the early stages of system development. For this last alternative, our experience suggests the use of PCIe over USB, and this is mainly to minimize latency, maximize control loop sampling frequency and maximize the upper limit imposed on the vehicle speed because of computational and communication related delays.

In our research vehicle, CAN messages generated by the sensors are called CAN reports, and CAN messages sent to the actuators are called CAN commands. In the following, we present two representative samples.

B. CAN steering command

For a self-driving vehicle application, there will be a control loop running on the on-board computers. It will start with sensor data acquisition, then continue with processing and decision making. But at the end, these decisions should be ”sent” to something/somewhere to physically change the steering angle. This last step is done by generating a CAN packet, and writing it to the CAN network. Such packets are called CAN commands, because they contain actuator related ”commands” which will be interpreted and executed by the actuator hardware.

The CAN message sent to the motor controlling the steering wheel is called the CAN steering command. This message has the message ID `0x064`, receive rate of 20ms, and timeout of 100ms. The payload of the CAN packet is 8-bytes, the first two bytes are interpreted as a 16-bit signed integer in 2's-complement format and the steering angle floating point value is obtained by multiplying this signed number by 0.1 degrees. The steering angle is physically limited to ± 470 degrees. If a CAN message is not received within timeout period, then the Drive-By-Wire system disengages as a safety feature and the system returns back to full manual-control mode.

Basically, at the end of our control-loop, we write a CAN steering command type CAN packet to the CAN bus to readjust the steering wheel. With careful coding, we were able to achieve about 10+Hz control-loop frequency.

C. CAN wheel-speed report

For a self-driving vehicle, the control loop will start with sensor data acquisition. For cameras, we do use dedicated high-speed Gigabit ethernet cables between the camera and the computer that will process the camera data. Information from vehicle sensors like wheel-speed sensors, steering wheel torque sensor, etc are all transmitted to the control-loop hardware using the CAN bus network. Once all sensor data is acquired, then the control-loop will start processing and decision making, and finally transmit control decisions to the actuators. A CAN packet generated by the sensor hardware and carrying sensor measurement data is called a CAN report.

The CAN message sent by the wheel sensors has the message ID `0x06A`, and transmit rate of 10ms or 100 Hz. The payload of the CAN packet is 8-bytes, and the first two bytes correspond to the front-left wheel speed, the second two bytes are front-right wheel speed. The rest of the four bytes in the CAN packet payload is used for the rear wheel speed. For each wheel, the 16-bit quantity is interpreted as a 16-bit signed integer in 2's-complement format and the actual floating point value of the wheel speed is obtained by multiplying this signed number by 0.01 rad/s.

IV. Sensors and Sensor Fusion

In this section, we will go over the basics of our sensors, and the sensor fusion approach adopted in this work. Literature on sensor fusion is really immense and it has applications in almost all engineering disciplines, [6]. Kalman filtering approach to sensor fusion [7] is highly popular because the cases addressed by Kalman filtering are quite common, equations are easy to use in a real application, and objectives are also quite relevant for a good number of real applications. For our self-driving vehicle experiment, we do have multiple sensors with different update rates, therefore an ”extended” version of the Kalman filtering approach is really needed [8].

A. GPS and Wheel rotation-speed sensors

Our GPS sensor is U-blox F9 high precision GNSS module operating at 115,200 bps baud rate. It provides both position, and heading information at a rate of 4Hz or 250ms update period. Whereas the wheel rotation-speed sensors are providing sensor data every 10ms or at 100Hz refresh rate. These sensors do not have synchronized clocks, which makes the problem even more complicated. In other words, we do know that the update rate of the wheel sensor is approximately 25 times faster than the GPS, but since clocks are not synchronized and may run at slightly different speeds, a rigorous analysis is more complicated compared to a standard Kalman filtering problem.

For the kinematic model of the vehicle, we assume Ackerman steering to be able to use the simplified bicycle model [9], [10]. We also assume that between two GPS readings, there will be no wheel slip. A four wheeled vehicle's equations

of motion are not simple linear state-space equations of the form $\dot{x} = Ax + Bu$, indeed it is highly nonlinear with complicated constraints [11]. But, when considered in a small time-window, they are usually linearized to simplify the sensor fusion problem.

Our initial approach was to use a semi-heuristic complimentary filtering approach [12]. In summary, if we have two sensors with one being more accurate/reliable but with a slower response, and another one being less accurate/reliable but with faster response, a good compromise solution is to use what is known as the complimentary filtering approach. Note that, our GPS sensor can be viewed as 100 Hz refresh rate position sensor when "missing" points are calculated via interpolation. We also intentionally delay GPS sensor data by 0.25 seconds to be able to get a smooth interpolated sensor output without using any extrapolation technique which may lead to possible jumps or discontinuities. Furthermore, we also make the simplifying assumption that clock rate between two sensors are exactly 25, and ignore the small phase difference between two sensor clocks. These are mild assumptions for a self-driving vehicle experiment at low-speeds and simplifies the math considerably. However, at higher speeds these assumptions will not be justifiable, and may lead to larger tracking errors and/or catastrophic results during a self-driving experiment.

If x_1 is the sensor with a more accurate/reliable output but with a relatively slower response (Our interpolated GPS sensor), and x_2 is the wheel-speed sensor, with less accurate/reliable output but much faster response, then the equation

$$x_e = H_1(z)x_1 + B(x_a)H_2(z)x_2$$

can be used for sensor fusion. Here we have chose $H_1(z)$ as a discrete-time low-pass filter with cut-off frequency of 5 Hz, and $H_2(z)$ as the complimentary high-pass filter. The term B is a matrix computed by using all of that state variable of the vehicle, including the position, heading, steering angle, and may include complex tire-models as well. The augmented state variable x_a represents the augmented state which contains both position/heading, and all other state variables used in the vehicles kinematic model. Finally, x_e represents the result of the sensor fusion process.

B. Camera and Semantic Segmentation

Our self-driving vehicle also has a 50 Hz update rate hi-resolution machine vision camera, and we run AI models to find the center of the drivable area. Use of AI models for steering control has a long history going back to 1990s [13]. More recent results, [14], are based on the use of convolutional neural networks for steering angle control. These end-to-end control approaches are different than our proposed approach based on semantic segmentation. One can argue that the AI problem is possibly simpler in our approach, because the AI model need not learn the vehicle's dynamical model. But compared to the complexity of semantic segmentation, the vehicle's dynamical model will be a simpler complexity thing to learn. Therefore, a rigorous comparison is not easy, but we would like to re-iterate that we are not using an end-to-end steering control approach.

In Fig. 4, a sample camera image captured during a self-driving experiment is presented.

Figure 4. Front camera view while driving on the campus loop.

The image obtained from this camera is fed to a semantic segmentation model to generate the segmented image shown in Fig. 5. In this particular case, we are using the PSP (Pyramid

Figure 5. Semantic segmentation of the camera view shown in Fig. 4. The purple color represents the pixels associated with the drivable road.

Scene Parsing Network) Resnet101 trained with the cityscapes dataset. Based on our experience, we recommend the use of the Apache MXNet library and models available in its model zoo. Otherwise, one may need to resolve issues related to version differences linked to tensorflow and all other additional tools. Although this may limit us to the models available in the MXNet model zoo, it greatly streamlines the AI related software.

After semantic segmentation is completed, we have a controlled erosion stage, then connected components analysis, and finally a moments based approach to find the center of the main drivable area. A limited smoothing is also applied to generate a smoother curve. The final automatically generated lane center is shown in Fig. 6. Note that this is obtained first by using the PSP-Resnet101 AI model, followed by a couple of openCV operations.

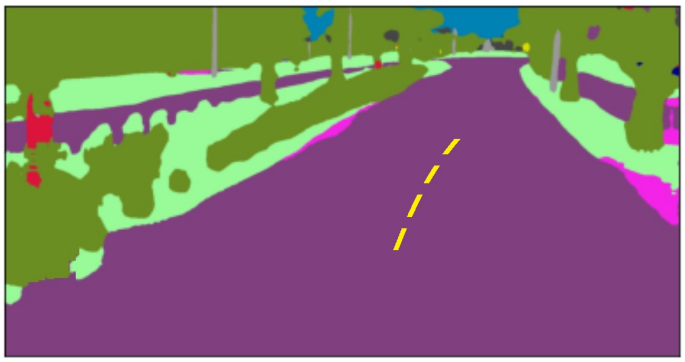

Figure 6. The yellow colored line is the automatically generated lane center. This is obtained by using the PSP-Resnet101 AI model, followed by a couple of more analytical openCV steps.

V. Experimental Results

In this section, we summarize our experimental results. Note that, the video recording of the experiment is available at https://www.youtube.com/watch?v=2kEqh4AuMGo, see Fig. 1.

In Fig. 7, we see a map of the campus obtained from OpenStreetMaps. Our self-driving vehicle experiment starts at the I-4 gate of the IST building, and ends at the Wellness Center parking lot. Unfortunately, OpenStreetMaps does not provide a high definition map, but since the Florida Polytechnic University Campus has a simple map consisting of a typical campus loop with some additional side roads, it is relatively easy to construct a simple HD-map of the campus. Our current HD-map consists of only lane information, and no other details like stop sign locations, speed-limit region definitions, etc. These are considered as future work.

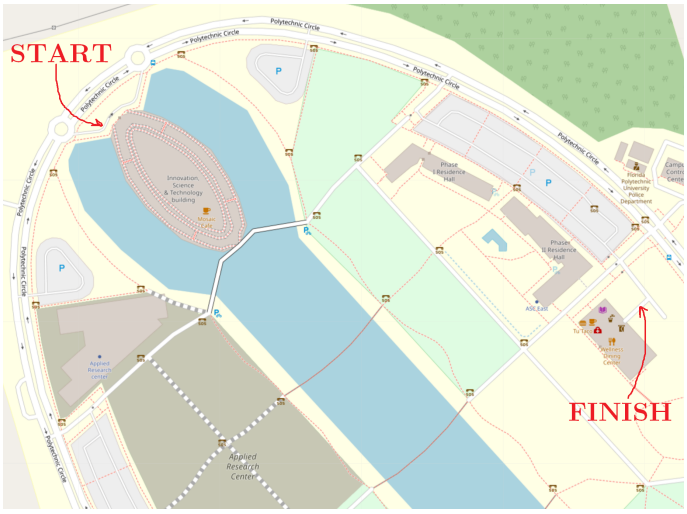

Figure 7. Florida Polytechnic University campus map shown on OpenStreetMaps. The starting point is the I-4 gate of the IST building, and destination is the Wellness Center parking lot.

In other words, our campus HD-map is stored as a directed graph with only direction/heading info about each node. Note that, nodes at intersections have multiple direction/heading possibilities. Once a destination is selected, a maze solver is executed to extract a feasible route from the HD-map of the campus. In the future, nodes are expected to have more features, like proximity to a stop sign, or being a member of a specific region. Currently, our HD-map data structure does not support regions, but again this is considered as future work. In Fig. 8, we have a route extracted from the campus HD-map. Note that, in Fig. 8, we use a simple locally linear transformation to map GPS coordinates to cartesian coordinates in cm, and the route starts from the upper-left end and ends at the lower-right end of the curve.

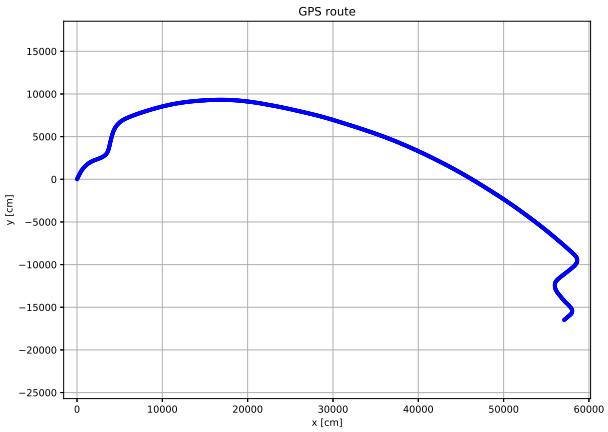

Figure 8. Route extracted from the HD-map of the campus. GPS coordinates are converted to cm by using a simple locally linear transformation.

In Fig. 9, we see the deviation of the vehicle trajectory compared to the extracted route. Blue dots, and red arrows define the route extracted from the HD map, and green dots are vehicle's GPS coordinates recording during the self-driving experiment. This figure is the detailed view of a 90° degrees turn while entering the parking lot.

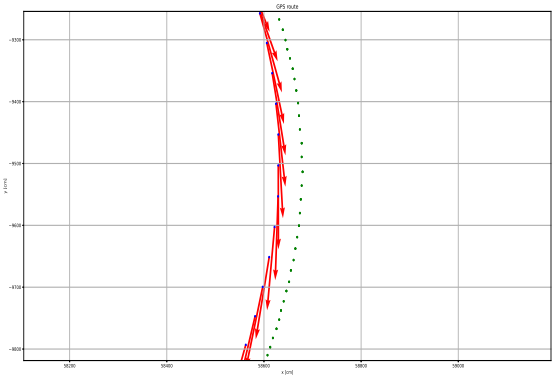

Figure 9. HD-map of a portion of the campus. Blue dots, and red arrows are part of the HD map, and green dots are vehicle's GPS coordinates.

Finally, in Fig. 10, we see the overall tracking error of the self-driving vehicle relative to the route extracted from the HD-map. All of the peaks in the tracking error occur when the vehicle is making a turn. This is possibly because of the simple guidance logic used in this preliminary work, and a model predictive control like approach is expected to reduce this error significantly. The initial large error is mainly because

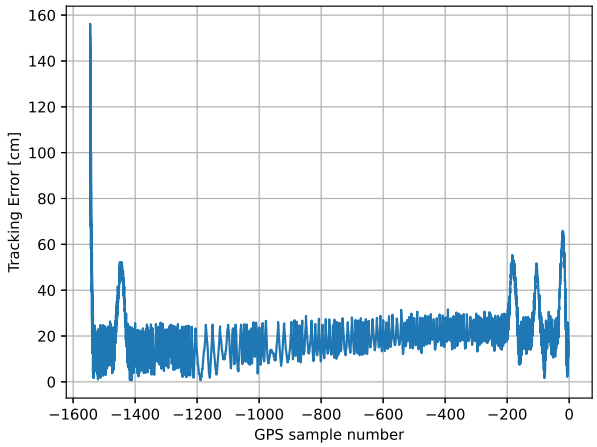

Figure 10. Tracking error of the self-driving vehicle relative to the HD-map.

of vehicle's initial location not being on the road defined by the HD-map. In other words, we started the experiment while the vehicle was parked near but not on the road defined by the HD-map.

VI. Conclusion

In this paper, we presented out first successful self-driving vehicle experimental results and shared its video recording. Compared to our earlier work [15], we now able to accomplish several more complex objectives with significantly improved autonomy. This is based on the use of high-definition maps, and sensor fusion. Our main focus was on lateral control, and the longitudinal dynamics was controlled separately by an independent control loop running on an STM32 microcontroller device. For lateral control, we used GPS, wheel rotation speed sensors, and a forward looking camera. Part of the sensor fusion is based on what is known as the complimentary filtering, and the AI processing was based on mainly the PSP-Resnet101 trained with the cityscapes dataset. Our initial results seem quite promising, with errors mainly occurring at sharper turns. We plan to address this issue by considering better sensor fusion and guidance logic, as well as more accurate/detailed vehicle kinematic models.

Aknowledgements

Funding is provided by NSF grant 1919855, and Florida Polytechnic University. We would like to thank our collaborators Arman Sargolzaei, Jorge Vargas, Suleiman Alsweiss, Navid Khoshavi, Ala Alnaser, Reza Khalghani, and our students Matthew DeCicco, and Gabriel Tavares, and finally last but not least our fabrication expert Mike Kalman.

References

[1] K.-W. Chiang, S. Srinara, S. Tsai, C.-X. Lin, and M.-L. Tsai, "High-definition-map-based lidar localization through dynamic time-synchronized normal distribution transform scan matching," *IEEE Transactions on Vehicular Technology*, pp. 1–13, 2023.

[2] J. Jiao, "Machine learning assisted high-definition map creation," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 01, 2018, pp. 367–373.

[3] S. Zheng and J. Wang, "High definition map-based vehicle localization for highly automated driving: Geometric analysis," in *2017 International Conference on Localization and GNSS (ICL-GNSS)*, 2017, pp. 1–8.

[4] C. Kim, S. Cho, M. Sunwoo, P. Resende, B. Bradaï, and K. Jo, "Updating point cloud layer of high definition (hd) map based on crowd-sourcing of multiple vehicles installed lidar," *IEEE Access*, vol. 9, pp. 8028–8046, 2021.

[5] S.-J. Han, J. Kang, Y. Jo, D. Lee, and J. Choi, "Robust ego-motion estimation and map matching technique for autonomous vehicle localization with high definition digital map," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, 2018, pp. 630–635.

[6] D. Hall and J. Llinas, "An introduction to multisensor data fusion," *Proceedings of the IEEE*, vol. 85, no. 1, pp. 6–23, 1997.

[7] A. Assa and F. Janabi-Sharifi, "A kalman filter-based framework for enhanced sensor fusion," *IEEE Sensors Journal*, vol. 15, no. 6, pp. 3281–3292, 2015.

[8] J. Wang, Y. Alipouri, and B. Huang, "Multirate sensor fusion in the presence of irregular measurements and time-varying time delays using synchronized, neural, extended kalman filters," *IEEE Transactions on Instrumentation and Measurement*, vol. 71, pp. 1–9, 2022.

[9] G. Bellegarda and Q. Nguyen, "Dynamic vehicle drifting with nonlinear mpc and a fused kinematic-dynamic bicycle model," *IEEE Control Systems Letters*, vol. 6, pp. 1958–1963, 2022.

[10] Y. Guan, Z. Zeng, D. Chen, T. Zhang, H. Zhu, and L. He, "Kinematic modeling, analysis, and verification of an essboard-like robot," *IEEE/ASME Transactions on Mechatronics*, vol. 26, no. 2, pp. 864–875, 2021.

[11] J. Ostrowski, "Steering for a class of dynamic nonholonomic systems," *IEEE Transactions on Automatic Control*, vol. 45, no. 8, pp. 1492–1498, 2000.

[12] G. J. Aparna, C. Kamal, and R. N. Motta, "Imu based attitude estimation using adaptive complimentary filter," in *2021 International Conference on Communication information and Computing Technology (ICCICT)*, 2021, pp. 1–5.

[13] D. Pomerleau, "Progress in neural network-based vision for autonomous robot driving," in *Proceedings of the Intelligent Vehicles '92 Symposium*, 1992, pp. 391–396.

[14] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," 2016. [Online]. Available: https://arxiv.org/abs/1604.07316

[15] A. Dubs, V. C. Andrade, M. Ellis, B. Karaman, D. Demirel, A. J. Alnaser, and O. Toker, "Drive a vehicle by head movements: An advanced driver assistance system using facial landmarks and pose," in *HCI International 2022 Posters - 24th International Conference on Human-Computer Interaction, HCII 2022, Virtual Event, June 26 - July 1, 2022, Proceedings, Part I*, ser. Communications in Computer and Information Science, C. Stephanidis, M. Antona, and S. Ntoa, Eds., vol. 1580. Springer, 2022, pp. 502–505. [Online]. Available: https://doi.org/10.1007/978-3-031-06417-3_67