Autonomous Design Report
Revolve NTNU Driverless, Car 463

Authors:
Revolve NTNU Driverless Team 2019
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway

Contact Person:
Toralf Frich
Email: toralf.frich@revolve.no
Phone: +47 46822523

***Abstract*—The main goal of Revolve NTNU Driverless is to achieve a top 5 position at the Formula Student Driverless (FSD) competitions. From this, a set of performance goals were derived as a basis for designing the autonomous software and hardware. The autonomous systems presented were validated with simulations and HIL tests against the targeted performance goals.**

I. Introduction

At FS Germany (FSG) in 2018, Revolve NTNU Driverless placed seventh as a debutant, having no prior experience of building autonomous race cars. In the 2019 competitions, a new dynamic event, autocross, is added, raising the achievable points in dynamic events by 25. This highlights that dynamic performance is essential to reach the main goal of Revolve NTNU Driverless this year: Achieve top 5 positions in FSD competitions. From a performance analysis of the top driverless teams from FSG 2018, concrete dynamic performance goals have been derived; drive at velocities of $5\,\mathrm{m/s}$ and $10\,\mathrm{m/s}$ on average on autocross and trackdrive, and obtain $a_x = 0.5g$ in acceleration and $a_y = 1.2g$ in skidpad.

This paper describes the main design decisions made to create an autonomous driving pipeline achieving the above-mentioned goals. Section II, driving pipeline, describes the architecture, design and testing of the autonomous systems. In section III, implementation, the main features of the software development, use of version control and validation will be highlighted. Finally, the paper is summarized in section IV.

II. Driving Pipeline

To derive the sub-goals for each autonomous system, an analysis, summarized in this paragraph, has been conducted to reach the performance goals outlined in section I. An average speed of $10\,\mathrm{m/s}$ imply a maximum speed of $12\,\mathrm{m/s}$ during straights. According to the FSD rules, the sharpest hairpin has a curvature of $\kappa = 0.22\,\mathrm{m}^{-1}$, which the car can drive at $6\,\mathrm{m/s}$ (with a safety margin). Consequently, $11\,\mathrm{m}$ of track should be planned at all times. Finally, as the maximum distance between cones along the sides are $5\,\mathrm{m}$, cones $15\,\mathrm{m}$ ahead should be visible (to capture the track boundaries).

A. Architecture

The autonomous architecture consists of all software and hardware that make the car driverless, see Figure 1. The computing devices used are the main processing unit (PU), the autonomous control unit (ACU) and the vehicle control unit (VCU). The PU manages all high-level software systems, see subsection II-C for specifications. On the low-level, the ACU is responsible for managing the autonomous state machine, steering actuator and the emergency brake system (EBS), while the VCU communicates with the inverters.

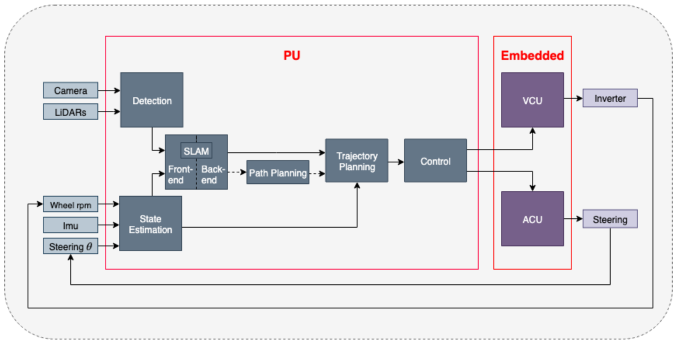


Fig. 1: The architecture of the driving pipeline.

The camera and two LiDARs supply the detection module, while the steering wheel encoder, wheel encoders and IMU supply the state estimation module with perception data. With three sources of detection data, asynchronous data processing is used. Using Probabilistic Data Association (PDA) in the Simultaneous Localization and Mapping (SLAM) front-end makes this possible without resorting to hardware synchronization. The SLAM front-end and back-end are tightly coupled to sustain a global map in the back-end with verified cones from a local map in the front-end. These algorithms are further described in sections II-D, II-E, II-F and II-G.

The path planning algorithm estimates the track boundaries and extrapolates how they evolve, see subsection II-H. A center line is interpolated between the track boundaries, and the control algorithm ensures convergence towards this path, see subsection II-I.

B. Sensors

The sensor layout is designed for position and color estimation of cones as well as estimation of the dynamic states of the car. LiDARs provide accurate position estimates, regardless of illumination, whereas camera images can be used for color classification. Two LiDARs, a Velodyne VLP-16 Puck and a 64-channel Ouster OS-1, are chosen for redundancy, providing cone estimates from different perspectives. The chosen camera is a global shutter color camera, Basler ace acA1300-200uc. The OS-1 LiDAR and the camera are mounted in the main hoop. This gives the OS-1 a near full 360° Field of View (FoV) around the car, utilizing its high vertical resolution, and a 77.32° FoV for the camera. Due to its sparse vertical FoV, the VLP-16 is placed underneath the impact attenuator to scan for cones along the ground.

For estimating the velocity, acceleration, wheel slip and wheel forces, wheel encoders are supplying the RPM of the wheels, an absolute magnetic rotary encoder in the steering wheel provides the steering angle and a Vectornav VN-300 INS supply inertial measurements.

C. *Processing Unit*

The autonomous software requires a PU which can handle consistent input streams of perception data and multitasking of the autonomous software systems, while being constrained in size to fit on the car. Hence, a compact custom built PU, running on Ubuntu 16.04, was chosen. It is composed by a Gigabyte B360n WiFi motherboard, Intel i7-8700 processor (CPU) and NVIDIA GTX 1070 graphics processor (GPU). The CPU has a x86 architecture and six cores promoting multitasking to reduce context switch overhead. The ARM-based architecture was disparaged because experience from last year showed it greatly hampered the development process. As the camera detection uses a Convolutional Neural Network, see subsection II-F, an appropriate GPU was included. The build altogether maximizes the computing power employing the smallest commercially available parts. The drawback of the build is that its power consumption generates a lot of heat. To avoid overheating with a maximum effect of 250W, the CPU and GPU are implemented into the water cooling system of the powertrain.

D. *State Estimation*

Accurate estimation of linear and angular velocity, is of great importance for autonomous driving. The solution employed last year, which only used the RPMs of the wheels, gave significant errors, due to wheel slip in the estimated velocities at speeds above $2\,\mathrm{m/s}$. To accurately maneuver at speeds consistent with the performance goals, an approach with low errors at high velocities in the presence of wheel slip is required.

The chosen approach is a combination of two nonlinear observers, developed in [1] and [2]. To estimate the linear and angular velocities, the observer estimates the bank and inclination angle of the ground relative to the car. It also estimates the friction coefficient between the wheels and the ground, which is approximated equal for all four wheels. A nonlinear observer is chosen over the more common Extended Kalman Filter (EKF) [3], because the observer has fewer parameters to tune and has better convergence properties.

The performance of the state estimator is illustrated in Figure 2. The implementation is verified on sensor data from the Endurance event at FSG 2018, driving at a mean speed of $14.9\,\mathrm{m/s}$, well above the performance goals. These experiments show that driving according to the goals, the drift should be reduced to a maintainable level for the SLAM system to correct.

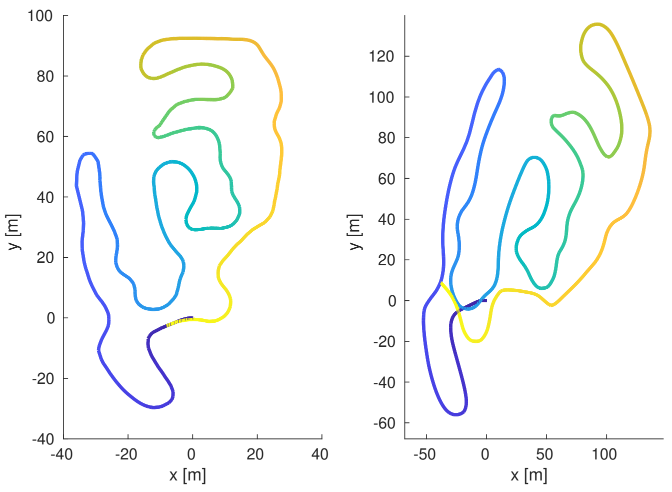


Fig. 2: Position during FSG Endurance 2018. Left plot is Dual-GPS data, while the right plot is the result of the state estimator. The color gradients visualize the progress along the track.

E. *LiDAR Detection*

The LiDAR detection algorithms utilize point clouds to detect and estimate cone positions, using the Point Cloud Library [4]. In FSD racetracks, the surrounding environment mainly consists of reasonably flat ground, cones and walls. Hence, feature selection is applied to extract the points representing cones. The algorithms were designed to achieve high recall and precision. The two LiDAR data streams are processed asynchronously for redundancy and to maintain a low individual system complexity. An overview of the algorithm can be found in Figure 3.

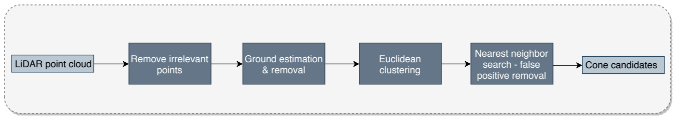


Fig. 3: The high-level approach applied to detect cones using LiDAR sensors.

Feature selection is applied on the input clouds, filtering points outside the region of interest and ground points. For the OS-1, ground points are estimated and removed by using adaptive ground line fitting, as described in [5]. A more simplistic approach is used for the VLP-16, as ground fitting removes desired cone points on sparse point clouds. Utilizing the low placement of the LiDAR, a majority of the ground points are removed by simply discarding all points from channels angled below $-3°$ horizontal. Conditional Euclidean clustering are performed to remove remaining ground lines from the channel angled at $-1°$. This removes the majority of ground points while no crucial cone information is lost.

After ground removal, the problem is reduced to identifying point clusters representing the desired objects: small and large cones. Consequently, spatial clusters are located using Euclidean clustering and verified using standardized sizes of the cones used in FSD. Nearest neighbour search is utilized to remove false positives. Finally, cone positions are corrected by their approximate radial offsets and output to SLAM, accompanied by an empirically derived covariance in the angular and radial dimensions.

The implementation is verified using sensor data from a self-designed sensor test rig[1] with a reliable detection range of $15\,\mathrm{m}$ and a runtime of ~$20\mathrm{ms}$. This shows initial results that the $15\,\mathrm{m}$ target range is attainable.

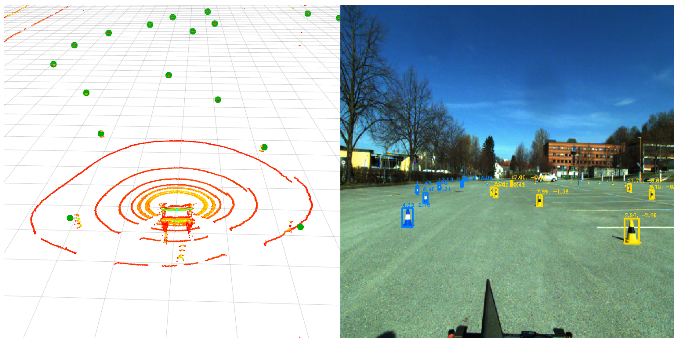


Fig. 4: Example visualization of detection using the VLP-16 LiDAR (left) and camera (right). The data was captured using a self-designed sensor test rig. The large green points in the LiDAR visualization represent detected cones.

F. Camera Detection

To localize the cones in a camera image frame, a two-step system is implemented. The first step aims to localize (with bounding boxes) and classify the cones in the 2D image frame, and the second step maps these bounding boxes to 3D position estimates.

While there exists many image object detection algorithms, YOLOv3 outperforms the others in terms of speed of inference while maintaining top-level accuracy [6]. To reach the performance goals, detections must be performed at a high frequency. This motivates the choice of YOLOv3 as the image detection algorithm as it is verified to perform bounding box localization at more than 20 FPS on the GPU. YOLOv3 is retrained on a dataset comprised of more than 45000 images, labelled with bounding boxes and classes of cones. With such a large dataset and narrow domain, the implementation produce precise bounding boxes, see Figure 4.

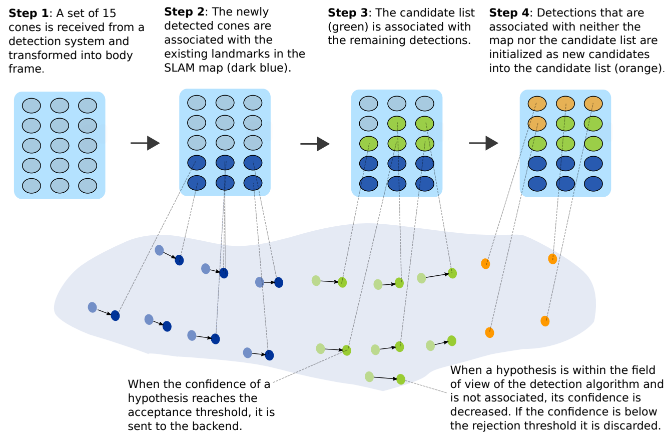
The second step of the system is inspired by [7]. Using the ratio of the real-world sizes of the cones and the height of the cones in the image projection (given by the bounding box), a scaling factor between the camera projection and the real-world position of the cones is found. This scaling factor, along with the intrinsic camera parameters, allows for determining the real-world position of the cones.

This method assumes that the YOLOv3 bounding box output is pixel-accurate, which is not always the case, especially at greater distances. To approximate a function that corrects for the systematic error caused by this, a 3-layer neural network is implemented. The network is trained to map both the bounding boxes and geometric position estimates to new position estimates, where positions obtained from associating the geometric estimates with the LiDAR detections are used as ground-truth. With this method, the error of localization is reduced by $58\,\%$. SLAM assumes the localization error to have expectation zero, which is found to hold empirically for distances $\leq 16\,\mathrm{m}$. Thus, the camera detection system can perform accurate cone localization and classification at at least 20 FPS $\leq 16\,\mathrm{m}$, achieving the $15\,\mathrm{m}$ target range.

G. SLAM

The SLAM system solves the task of building a global map of the track and estimating the location of the car in it. It does so by using the estimated pose from the state estimator and the set of local cone detections from the detection systems.

SLAM is split in two; the front-end and the back-end. The front-end, illustrated in Figure 5, associates detections with landmarks in the existing map, creates hypotheses and updates belief of existing hypotheses. Hypotheses are discarded or accepted when their belief get below or above a threshold, respectively. The back-end adds new measurements to a factor graph, optimizes the relevant parts of the graph and outputs a corrected estimate of both the map and the pose of the car.


Step 1: A set of 15 cones is received from a detection system and transformed into body frame.
Step 2: The newly detected cones are associated with the existing landmarks in the SLAM map (dark blue).
Step 3: The candidate list (green) is associated with the remaining detections.
Step 4: Detections that are associated with neither the map nor the candidate list are initialized as new candidates into the candidate list (orange).
When the confidence of a hypothesis reaches the acceptance threshold, it is sent to the backend.
When a hypothesis is within the field of view of the detection algorithm and is not associated, its confidence is decreased. If the confidence is below the rejection threshold it is discarded.
Fig. 5: The SLAM front-end illustrated and explained in 4 steps.

The front-end is built on ideas from PDA [8]. When new detections arrive, it transforms them to a global frame using the estimated pose of the car. It then uses a threshold on the Mahalanobis distance between two Gaussians to associate measurements with existing hypotheses and cones in the map.

[1]Figure 4 provide a visualization of VLP-16 data from the test rig.

If a new detection cannot be associated with an existing cone in the map or a hypothesis, and it is far enough away from all cones in the map, a new hypothesis gets spawned. Every time a detection is associated with a hypothesis, the belief in the hypothesis is increased. If a hypothesis is not seen within the FoV of a given sensor, its belief is decreased.

The back-end is built on GTSAM [9]. It uses incremental Smoothing and Mapping 2 (iSAM2) [10], an algorithm that iteratively builds a factor graph of the car poses and the observed landmarks.

ISam2 is chosen to maintain low run-time while avoiding excessive map error. Additionally, it can be tuned to determine how much of an impact on the overall error a pose or landmark must have to be included in the optimization. This has allowed the algorithm to achieve an overall RMS error of $0.1267\,\mathrm{m}$ while keeping the mean run-time to $3.7\,\mathrm{ms}$. This result was found by comparing all the cones in the SLAM map using the track layout of FSG Trackdrive 2018 as ground truth, on 10 simulations with zero mean Gaussian noise added to all detections with a standard deviation of $0.1\,\mathrm{m}$.

H. *Path planning*

The path planning algorithm is responsible for estimating the track boundaries and interpolate a center line given the map provided by SLAM. To achieve the performance goals, the center line should project at least $11\,\mathrm{m}$ ahead of the car while being robust against missing or false detections.

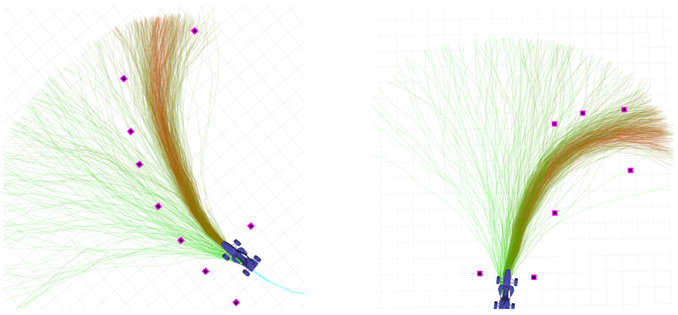


Fig. 6: Visualization of path planning algorithm in two different simulated racetrack scenarios with 500 particles. The red lines represent the best particles.

The implemented solution is a particle filter where each particle is a linear spline representing an estimated center line. All particles are of equal length and consist of 10 to 15 similar line segments.

Initially, the particles are drawn from a uniform distribution from which each line segment gets a curvature. For each iteration of the filter the particles are evaluated by a probability function. The probability function assumes that each cone is distributed normally at a slowly changing distance from the center line. Color information is a binary weight, which either validates or refutes the sign of the relative cross-track deviation from the particle to the cone measurement.

The particles with the best scores from the probability function are averaged to give the current prediction of the track. After the prediction, the particles are uniformly resampled based on their weights. They are subsequently perturbed by changing their curvature for exploration of the search space. New particles are also introduced to avoid false convergence and aid exploration of the search space. The particles are then re-evaluated, and the process repeats.

The implementation is verified on recorded data from FSG 2018 to predict the track more than $11\,\mathrm{m}$ ahead, despite the presence of a large false negatives rate, see Figure 6.

I. *Control*

It is assumed that the center line of the track is known from path planning. Furthermore, the friction and state of the vehicle is estimated by the state estimation module, see subsection II-D. Using the forward-backward consistency algorithm [11], a feasible velocity and acceleration is computed along this center line. The acceleration is additionally constrained in accordance with the performance goals, section I, $-1.2g \leq a_{\mathrm{y}} \leq 1.2g$ and $-1.0g \leq a_{\mathrm{x}} \leq 0.5g$.

The control law used by Team 2018 is based on the line-of-sight principle and simple PID speed and heading controllers. To account for the non-linearities arising during cornering at the target speed of $10\,\mathrm{m/s}$, the heading controller is extended this year. The controller is based on a cascade of a kinematic path following feedback linearization controller [12] and a feed-forward feedback controller inspired by [13]. The former ensures geometric path convergence[2], see e.g. [15], and the latter ensures sufficiently robust cancellation of the nonlinear cornering dynamics to follow the geometric set-points[3].

Firstly, the implementation is verified in a dynamics simulator[4] developed by Revolve NTNU, with added noise and time delays. Its robustness is demonstrated at $v_{\mathrm{max}} = 12\,\mathrm{m/s}$, in figure 7 with actuator time delay $\Delta_{\mathrm{delay}} = 0.1s$ and significant noise introduced to the state estimates.

Finally, the implementation is verified in Hardware In the Loop (HIL) tests with the ACU, VCU and steering actuator to verify that the system handles the variable time delays introduced by CAN and that the steering actuator responds fast enough. The average time delay of the steering actuator response was identified to be $\bar{\Delta}_{\mathrm{act}} < 10\,\mathrm{ms}$, which is within what the control system can handle.

III. Implementation

A. *Software development*

C and C++ were appointed as the primary programming languages to decrease language complexity and ensure memory and speed performant code. The embedded software is written in C using the FreeRTOS framework, and the autonomous software systems on the PU are written in C++ using the Robot Operating System (ROS). ROS is a distributed software

[2]Effectively solving the geometric task from [14]

[3]Effectively solving the speed assignment task from [14]

[4]Including a Pacejka tire model, lateral/longitudinal load transfer and a simplified aerodynamics model.

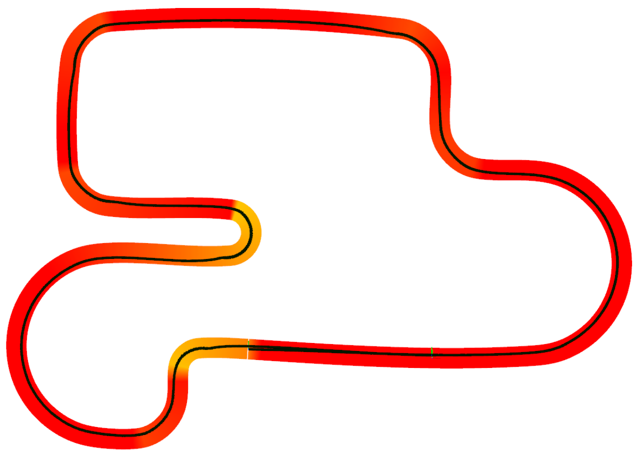


Fig. 7: The black line shows the car position relative to a $2\,\mathrm{m}$ wide track. The colors represent the desired velocity, red being above $10\,\mathrm{m/s}$. Maximum center line offset measured in this run is $1.2\,\mathrm{m}$, although this can be controlled by tuning the lookahead distance ($\Delta_{\text{Lookahead}} = 4.0\,\mathrm{m}$ in this experiment).

framework. Consequently, the integrity and functionality of the driving pipeline are volatile against minor implementation changes. Thus, the team demanded an agile development methodology to maintain collaboration and communication to mitigate the risk of individual errors and misapprehensions. Hence, a combination of features from multiple methodologies has been used. The combination exploits the meeting structure from Scrum, the task board system from Kanban and the pair programming aspect from Extreme Programming.

Rviz, the standard visualization tool for ROS, has been used for debugging, to compare algorithmic design alternatives and validate software performance with both simulated and real data.

System tests have been implemented to verify performance in run-time and accuracy. Integration tests were developed to check for malfunctions in functionality or data-flow between the systems. The HIL test mentioned in subsection II-I is an example of such an integration test.

The car possesses a diagnostic system that acquires information about the algorithmic and computational performance as well as operative status of the autonomous systems and sensors. This information is utilized in a validation algorithm to decide whether the car retains an operational and a safe autonomous driving state.

B. Version Control

Git has been used for version control, using git submodules as a tool to organize the dependencies and self-developed repositories. Despite introducing overhead and complexity, submodules separate the individual systems conceptually and allow the execution of the driving pipeline with hand-picked versions of each system. This enables the opportunity to easily test different configurations or backtrack to earlier versions of each system in the driving pipeline.

A three step git-flow with pull requests have been used within the git submodules to verify the quality of the code. The *master* branch contains the stable version of the system. A *dev* branch has been diverged out from the master to store all new proposed features developed in individual *feature* branches.

C. Validation

Continuous integration runs on a Jenkins build server to validate that the driving pipeline builds successfully from scratch, catching integration and dependency issues early.

The software systems have been tested using recorded sensor data, *rosbags*, to validate the driving pipeline. The rosbags have been recorded with a self-designed test rig which has the same sensor layout as the car, see subsection II-B and Figure 4. The test rig makes the accumulation of sensor data more accessible and practical without the need of the car. Additionally, real race data from the EV endurance and the DV trackdrive events in 2018 have been used to develop the autonomous systems.

IV. Conclusion

This paper presents an autonomous driving pipeline designed for performance goals predicated on the assumption that Revolve NTNU Driverless reach a top 5 position in FSD competitions. The subsequent implications of these goals are validated through simulations and HIL testing. Further work include full performance testing of all autonomous systems on the race car.

References

[1] T. A. J. T. I. F. J. C. K. A. S. Håvard Fjær Gripa, Lars Imsland, “Nonlinear vehicle side-slip estimation with friction adaptation,” 2008.
[2] T. A. J. J. C. K. A. S. Håvard Fjær Grip, Lars Imsland, “Vehicle sideslip estimation,” 2009.
[3] L. A. M. Gerald L. Smith, Stanley F. Schmidt, “Application of statistical filter theory to the optimal estimation of position and velocity on board a circumlunar vehicle,” 1962.
[4] “Point cloud library,” http://pointclouds.org/about/.
[5] M. Himmelsbach, F. v. Hundelshausen, and H.-J. Wuensche, “Fast segmentation of 3d point clouds for ground vehicles,” June 2010. [Online]. Available: "https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5548059"
[6] J. Redmon and A. Farhadi, “Yolov3: An incremental improvement,” 2018.
[7] B. Přibyl and P. Zemcík, “Simple single view scene calibration,” vol. 6915, 08 2011, pp. 748–759.
[8] E. T. Yaakov Bar-Shalom, “Tracking in a cluttered environment with probabilistic data association,” 1975.
[9] F. Dellaert, “Factor graphs and gtsam: A hands-on introduction,” 2012.
[10] R. R. V. I. J. L. F. D. Michael Kaess, Hordur Johannsson, “isam2: Incremental smoothing and mapping using the bayes tree,” 2012.
[11] C. Sprunk, “Planning motion trajectories for mobile robots using splines,” Student project, University of Freiburg, Germany, 2008.
[12] M. Breivik and T. I. Fossen, “Principles of guidance-based path following in 2d and 3d,” in *Proceedings of the 44th IEEE Conference on Decision and Control*, Dec 2005, pp. 627–634.
[13] B.-O. H. Eriksen and M. Breivik, *Modeling, Identification and Control of High-Speed ASVs: Theory and Experiments*. Cham: Springer International Publishing, 2017, pp. 407–431.
[14] R. Skjetne, T. I. Fossen, and P. KokotoviÄ‡, “Output maneuvering for a class of nonlinear systems,” *IFAC Proceedings Volumes*, vol. 35, no. 1, pp. 501 – 506, 2002, 15th IFAC World Congress.
[15] M. Breivik and T. I. Fossen, “Path following for marine surface vessels,” in *Oceans '04 MTS/IEEE Techno-Ocean '04 (IEEE Cat. No.04CH37600)*, vol. 4, Nov 2004, pp. 2282–2289 Vol.4.