DOI: 10.58895/ksp/1000150865 -15 erschienen in:
Forum Bildverarbeitung 2022 | Image Pocessing Forum 2022
DOI: 10.5445/KSP/1000150865 | https://www.ksp.kit.edu/site/books/m/10.58895/ksp/1000150865/

Signal processing pipeline for an autonomous electrical race car

Martina Scheffler[1], Ole Kettern[1], Oliver Zbaranski[1,2], Finn Schäfer[1,2], Kevin Schmidt[1], Bjarne Eberhardt[1], and Stefan Werling[2]

[1] CURE Mannheim, Autonomous System Team,
Handelsstraße 13, 69214 Eppelheim
[2] Duale Hochschule Baden-Württemberg Mannheim,
Coblitzallee 1-9, 68163 Mannheim

Abstract This work presents a signal processing pipeline for an autonomous race car in the context of Formula Student. The software used for each step from the detection of objects in camera images or lidar point clouds to the calculation of control outputs for the actuators is described in detail. The sensors and actuators are covered and the system output is visualized. The computational times of the pipeline are analyzed and it is derived that the complex algorithms used for motion planning and SLAM take up the most of the computation times, leaving the most room for improvements.

Keywords Autonomous driving, signal processing, Formula Student, YOLO, object detection, SLAM, MPC

1 Motivation

The future lies in autonomous driving, at least in the Formula Student (FS), an international design competition between student teams. In this work, the signal processing pipeline of the 2022 electrical and autonomous race car of the team CURE (Cooperative University Racecar Engineering) is presented. While the Formula Student poses a rather narrow challenge for autonomous vehicles due to a controlled environment and clearly specified tracks and track boundaries, it is a good development and testing ground for algorithms which are also used in agricultural, industrial or real-life traffic situations.

2 Problem description

During the FS events, the autonomous race car competes in four different types of competitions, all posing different challenges to the car and the Autonomous System (AS): Acceleration (1), Skidpad (2), Autocross (3) and Trackdrive (4). The disciplines test the car's ability to (i) drive straight lines (1), (ii) handle high acceleration and deceleration forces (1), (iii) withstand high lateral forces (2), (iv) choose the correct direction at a known intersection (2), (v) navigate unknown tracks (3) and (vi) reliably generate global maps and locate itself in them (3, 4). During all events, the track boundaries are marked by cones of known sizes [1, Tab. 3]. Small blue and yellow cones mark the left and right sides and orange cones signal finish lines and the exit areas in which the car has to come to a standstill. The challenge the cars face is to detect the cones correctly, align the detections with previous knowledge about the tracks - either from the competition rules or from internally built maps - generate a path to follow and send control signals to the actuators accordingly.

3 System overview

This section gives a brief overview of the hardware and software used to run the pipeline. In it, the processing unit, the sensors and the actuators of the race car are described.

To handle the challenges regarding the computational power and the needs of the image processing software, a custom-built Autonomous Compute Unit (ACU) consisting of an AMD Ryzen 5 5600G hexa-core CPU, a NVIDIA Tesla T4 data center GPU and 32GB of memory is used. On it, Ubuntu 20.04 LTS is installed. To implement the various functionalities of the AS, the Robot Operating System (ROS) Noetic is used. This provides the means for inter-process communication, threading, debugging as well as visualization tools. In order to simplify development, deployment and maintenance, the complete AS is containerized using Docker.

To interact with the rest of the electrical system in the race car, multiple CAN buses are used. To send / receive messages, a CAN to ROS interface is used. The sensors connected to the CAN bus include steer-

ing wheel angle sensors, wheel speed sensors and an IMU (Inertial Measurement Unit). All of them are used as inputs to the AS in order to track the car's position and generate control outputs accordingly. These are then used to control the actuators which include the motors, the motor for the steering actuation and the electrical valves for the brake system. Additionally to the sensors connected via CAN, other sensors are directly connected to the ACU via either USB or Ethernet. These include a dual-antenna GPS for position and heading information, a stereo camera and a lidar.

4 Signal processing pipeline

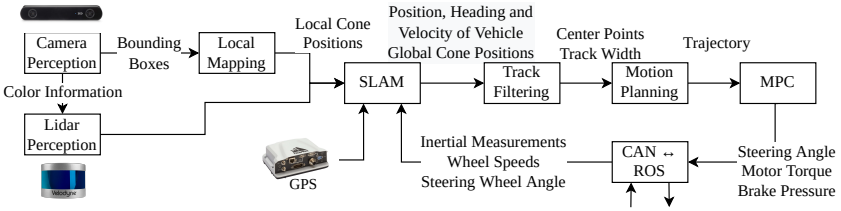

Figure 1: Overview of the modules of the signal processing pipeline.

This section gives a detailed description of the signal processing pipeline as a whole and each module in it as shown in Figure 1.

4.1 Camera perception

This section focuses on the generation of local maps from images taken with a Stereolabs ZED2i stereo camera.

Camera Calibration Since the camera images are currently used as the main way to determine the positions of the cones, the camera needs to be calibrated as precisely as possible. The local mapping process is closely related to this as it requires both an intrinsic and an extrinsic camera matrix to describe the transformation of the cone positions from image to world coordinates. With the currently used camera model,

the intrinsic matrix is supplied by the manufacturer and not subject to change. As of right now, the extrinsic calibration process is done by mapping image points to world points. In this case, our world points are represented in a 3×3 marker pattern whose position we obtain by measuring the distance to the camera itself. After capturing an image, coordinates of the markers in the image are collected by picking out the respective pixels. Using the image points, the intrinsic matrix and the world-coordinates of the points, the extrinsic matrix can be computed using OpenCV's function *solvePnP()* [2]. This method has the benefit of only using one image but the measurements of the world coordinates by hand and the determination of image points are error-prone and add a certain error to the calibration as a whole. Replacing the manual steps by automated library functions would bring a huge improvement to the accuracy of the resulting calibration.

Inference An integral part of the camera-based perception is the detection of differently colored cones in the images the camera provides. As these detections are used to calculate the position of the cones relative to the vehicle, the task of inference needs to be done both quickly and accurately.

In order to reach this goal, a neural-network-based approach for object detection was chosen. The core element is a YOLOv5 convolutional neural network [3], completely based on PyTorch, which makes it easier to work with. YOLO networks gained a lot of popularity in the last years as they achieve similar, if not better, accuracy than Single-Shot Detectors while being significantly faster [4]. Using the repository code, a network is trained using both images that were captured and labeled by ourselves, as well as additional training data from the Formula Student Objects in Context (FSOCO) repository [5]. To further improve the process, pre-trained weights are used which reduces the need for a big data set and, consequently, also the time needed for training.

The actual logic for the task of detecting cones is based on an open-source inference implementation of YOLOv5 that leverages the capabilities of NVIDIAs TensorRT library to further optimize performance [6]. Using this camera-based perception pipeline, the vehicle is able to detect cones in a distance of up to 15 meters on images with a resolution

of just 1280×720 pixels while achieving inference speeds of around 50ms on average. While the detections proved reliable under varying weather conditions, the neural network struggles in detecting cones when there is a strong backlight present, as the camera automatically lowers the image brightness as a consequence. Additionally, because of the the Python implementation, the processing of 1920×1080 pixel images that are provided by the camera is not possible without significantly sacrificing inference speed. Consequently, the migration of the code to C++ would be beneficial in the future.

Local Mapping In order to generate a map with reference to the car's current position, a translation of cone positions from image to world coordinates is necessary. The intrinsic and extrinsic matrices are used to project the top middle point of the bounding boxes around the detected cones from image to world. This projection results in a ray as the distance can not be calculated with only the pixel coordinates. To get the accurate position, the ray is intersected with a plane at the known height of the cones. This is done for all bounding boxes in the image resulting in a list of local cone positions to pass on to the rest of the pipeline.

While the calculation of the local maps itself has proven reliable during testing and competitions, its accuracy is highly dependent on the accuracy of the camera calibration, so an improved calibration process as mentioned above could significantly improve the quality of the local maps.

4.2 Lidar perception

To increase the robustness of the system as a whole, a Velodyne VLP-16 Puck Hi-Res lidar is used to generate local maps of the environment as well. For reasons of time, the lidar perception module has not actually been used during this year's competitions, However, development and tests with a test data set have been done.

First, the amount of data in the captured point cloud is reduced significantly by cropping the field of view in order to increase the computational performance. Second, the ground plane is filtered out using the Himmelsbach algorithm [7]. Once the point cloud only contains

points which are not in the ground plane, Euclidean Clustering is used to group the points. Then, the shapes of these clusters are checked to keep any cone-shaped clusters and remove erroneous detections like people, walls and other structures. The coordinates of the detected cones are then passed on to the SLAM and track filtering modules. Additionally, 3D to image translation is used to add color information to the detected cones using information from the camera images. As a side note, it has to be added that while the approach works well, the performance of the pipeline is limited by the low number of channels of the Puck Hi-Res. Cones which are 6m away from the lidar already consist of less than 10 points and the number of points decreases further with increasing distance.

4.3 Simultaneous localization and mapping

The main goal of Simultaneous Localization and Mapping (SLAM) is enabling motion planning to generate global trajectories and thus, increase the vehicle performance. The SLAM algorithm is implemented as an Unscented Kalman Filter (UKF) in Python. This type of filter was chosen as it is able to handle highly non-linear problems like polar cone positions more sufficiently than an Extended Kalman Filter (EKF). Also, it outperforms Particle Filters or Graph-based SLAM approaches due to their higher complexity. The underlying architecture and mathematics are based on the open-source library *FilterPy* [8], however adapted to increase speed and compatibility to our system. The tracked state vector $\vec{x}$ of the UKF consists of the tracked landmarks x_1, y_1 to x_n, y_n and the vehicle pose containing the vehicle position x, y, longitudinal vehicle velocity v_x and global vehicle heading ψ:

During the prediction, the system propagates through a simplified bicycle model disregarding any lateral forces and slip angles. The current steering wheel angle is used to calculate the travelled distance of the current cornering, while the current longitudinal acceleration a_x and yaw rate $\dot{\psi}$ measured by the IMU are used to calculate the new vehicle velocity and global vehicle heading. To continuously update the values, the output of the local mapping and lidar perception as well as the measurements of all four wheel speed sensors and the GPS are used.

To counter the disadvantage of the $\mathcal{O}(n^3)$ complexity of the UKF

algorithm with n being the number of state variables, the predicted state variables are limited to the vehicle pose states and the updated state variables are limited to the necessary ones, for instance, only the vehicle pose if no lidar perception or local mapping output is available and otherwise the vehicle pose and the observed landmarks. As a result, the complexity is nearly constant since the number of observed landmarks is naturally limited.

4.4 Track filtering

The track filtering module calculates the center point line of the track and the track width using the position and color of the cones. The general functionality of this module is split up into local and global filtering, based on the information passed on by the SLAM algorithm.

The local track filtering follows three steps. First, it finds the midpoints of the track using different approaches based on the number and color of cones available from SLAM. For only cone or one color available either the Dynamic Window or Border Shift approach is used. If more cones of each color are passed on, then the midpoints are calculated with the Delaunay Triangulation. With the variety of possible approaches, the reliability of this module can be enhanced. The second step is to interpolate and approximate the center line from the found midpoints. The third and last step is the definition of the legal track width for each point and the calculation of the left and right borderline. This information is then passed on to the motion planning module.

The global track filtering works very similar to its local counterpart, except that it uses only the Delaunay Triangulation for finding the midpoints, since all global cone positions are known. They are sorted with a tree algorithm and used to calculate the track width and border lines.

4.5 Motion planning

The goal of the motion planning module is to generate a trajectory to enable dynamic racing maneuvers. Therefore, it is separated into two parts: local and global. The local one is used when no closed global track is passed on by the SLAM algorithm. It is also used while the global optimization is still calculating the optimal race line for the closed and global track.

Local Motion Planning The local motion planning uses a directed geometric graph-based approach fully written in Python and based on [9]. The current vehicle position is used as origin. In regards to the centerline of the track, normals are calculated at regular intervals. The layers of the graph are made up of nodes which are evenly spaced on the normals. From one node, an edge to every node on the next layer exists. To generate a curvature-optimized race line, a cost is calculated for each edge. The cost takes into account the average and maximum of the squared curvature of the edge and its length. Using the known costs of all edges, the cheapest path can be found. The least-cost path represents the most curvature-optimal path, for which a velocity profile is then calculated. This velocity profile is calculated based on the hypothesis that the lateral velocity of the car at the apex point of a curve is 0 $\frac{m}{s}$. Due to this hypothesis, the maximum accelerating and decelerating velocity profiles are calculated from a ggv-map - it delivers the maximal acceleration forces - these two profiles are then superimposed.

Global Motion Planning The global approach is also based on curvature optimization and inspired by [10]. For generating the global race line, the problem is set up as a quadratic programming problem. The global algorithm tries to minimize the sum of the curvature for a given reference line. In this specific use case it is the closed global center line that is passed by the SLAM algorithm and used as reference line. In the following the quadratic solver tries to minimize the curvature via moving the way points on their normal vectors. The output path is then shifted into a trajectory using the same velocity profile calculation as the local approach. The global approach uses more computing power and takes more time to be calculated. Therefore, as mentioned above, the local optimization algorithm continues until a global trajectory is determined.

4.6 Model predictive control

The control module uses the trajectory from the planning module and the vehicle states from SLAM as input to control the vehicle dynamics. More precisely, the goal is to control the vehicle movement along the planned path. This *Path Tracking* problem [11] aims to minimize

the delta between the vehicle and the path points as well as to assure progress along the race track. A nonlinear model predicitve controller (NMPC) was developed to reach this goal. In general, a NMPC consists of three key components: A nonlinear vehicle model, an optimization objective and a reliable numerical solver. Based on the vehicle model, the solver calculates the optimal control values in real time in order to minimize a cost function, which serves as the optimization objective. In comparison to classic control theory approaches, the NMPC is able to predict and control the future behaviour of vehicle states inside of the prediction horizon. Hence, model predictive controllers are very popular for autonomous vehicles. The vehicle model is described as a nonlinear state space, that outlines the vehicle dynamics. We use a kinematic bicycle model, that neglects tire forces, similar to [12] and to the one used inside the SLAM algorithm. The model is implemented in Python as a system of time-continuous differential equations with the vehicle acceleration a_x and the tire angle rate $\dot{\delta}$ as model input. To discretize the model, a 4^{th} order Runge-Kutta integrator is used. In every time step, the NMPC calculates the optimal input vector to solve the optimization objective. Based on the sign of the input acceleration a_x, this value is transformed to either a pneumatic brake pressure or a motor torque value. These control values are published to ROS and then transmitted via CAN to the low-level control devices. Furthermore, these control values are filtered with an IIR-Filter to counter noises and outliers from the whole pipeline. The optimization objective is mathematically described as a quadratic cost function, where the squared difference between the predicted vehicle positions and the reference positions are summed up over the prediction horizon. The reference positions for every time step along the prediction horizon are derived by preprocessing the trajectory similar to [12]. Path points and the velocity profile are used to calculate a time profile, which then is used to extract the exactly-timed reference positions inside the prediction horizon. To solve the optimization objective in real-time, the FORCESPRO NLP solver by Embotech is used [13]. This solver predicts input values, which minimize the cost function. With solving times below 5 milliseconds this solver is very reliable for application inside the ACU. Due to the prediction horizon of $N = 20$ and a time step of 50 ms, the NMPC is able to predict and control the vehicle dynamics one second ahead of the current state using the kinematic bicycle model.

5 Results

Since the benefits and drawbacks of the single modules were already explained in Section 4, this section focuses more on the results and computation times of the whole pipeline.

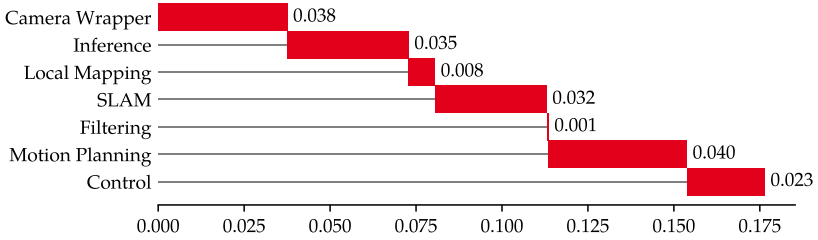

Figure 2: Median processing time of each module of the pipeline in seconds.

Figure 2 shows the median processing time of each module and subsequently of the whole signal processing timeline. It takes about 175ms from the recording of an image until it is represented in the control output. The camera wrapper includes the recording and processing of the image and the encoding into a ROS message. Computation heavy modules like the SLAM and motion planning module have a major share, which can be lowered by using a different parameter set, migrating to a more efficient language like C++ and parallelizing specific computations. The processing time of the control module is misleading, since it is executed with a fixed rate of 20 Hz and thus the median processing time includes idle time. Modules like the inference, local mapping and filtering provide little room for improvement since most of their calculations are carried out with efficient libraries like YOLOv5 or *NumPy*.

Under the assumption that the vehicle velocity is $15\frac{m}{s}$, the total processing time will lead to a loss of 2.65m effective perception range. Since the position dependent modules (filtering, motion planning and control) use separate and newer vehicle position, the control output error due to a wrongfully assumed vehicle position is limited and can be compensated by choosing a corresponding time step of the NMPC.

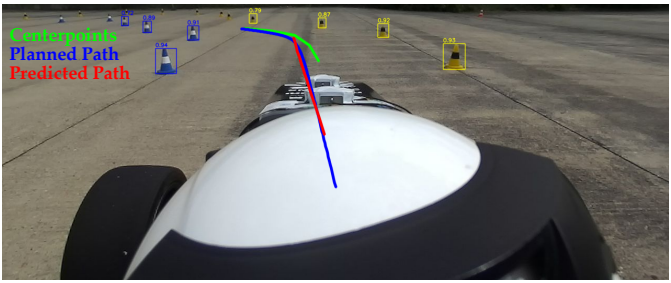

Figure 3: Output of the pipeline visualized on a camera image.

Figure 3 shows the system output visualized on a camera image captured on a testing day with a driver. The detected bounding boxes of the inference module as well as the calculated center points (green), the planned path (blue) and predicted path (purple) are shown.

6 Conclusion and outlook

In this work, the signal processing pipeline for an autonomous race car in the context of Formula Student competitions was presented. Each module was explained in detail, also focusing on its positive and negative aspects regarding computational cost and reliability. The resulting output of the system was visualized and the computational times of each module were analyzed and put into context.

To improve the system in the future, the plan is to improve the calibration method used for the camera perception to enhance the accuracy of the local maps from images. Furthermore, an investment in a lidar with more than 16 channels, Gaussian channel distribution is planned and work is done to correctly integrate it into the system. Finally, the computational times of the motion planning and SLAM modules will be reduced by migrating them to C++.

References

1. Formula Student Germany. Competition Handbook 2022. [Online]. Available: https://www.formulastudent.de/fileadmin/user_upload/all/

2022/rules/FSG22_Competition_Handbook_v1.2.pdf

2. OpenCV. solvePnP. [Online]. Available: https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#ga549c2075fac14829ff4a58bc931c033d
3. Ultralytics, "YOLOv5," Jul. 2021. [Online]. Available: https://github.com/ultralytics/yolov5
4. J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," 2018. [Online]. Available: https://arxiv.org/abs/1804.02767
5. N. Vödisch, D. Dodel, and M. Schötz, "FSOCO: The Formula Student Objects in Context Dataset," *SAE International Journal of Connected and Automated Vehicles*, vol. 5, no. 12-05-01-0003, 2022.
6. W. Xinyu, "TensorRTx," Jul. 2021, original-date: 2019-11-25T09:01:36Z. [Online]. Available: https://github.com/wang-xinyu/tensorrtx
7. M. Himmelsbach, F. Hundelshausen, and H.-J. Wuensche, "Fast segmentation of 3d point clouds for ground vehicles," 07 2010, pp. 560 – 565.
8. R. Labbe. Filterpy. [Online]. Available: https://filterpy.readthedocs.io/en/latest/
9. T. Stahl, A. Wischnewski, J. Betz, and M. Lienkamp, "Multilayer graph-based trajectory planning for race vehicles in dynamic scenarios," in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, 2019, pp. 3149–3154.
10. A. Heilmeier, A. Wischnewski, L. Hermansdorfer, J. Betz, M. Lienkamp, and B. Lohmann, "Minimum curvature trajectory planning and control for an autonomous race car," *Vehicle System Dynamics*, vol. 58, no. 10, pp. 1497–1527, 2020. [Online]. Available: https://doi.org/10.1080/00423114.2019.1631455
11. T. Faulwasser and R. Findeisen, *Nonlinear Model Predictive Path-Following Control*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 335–343. [Online]. Available: https://doi.org/10.1007/978-3-642-01094-1_28
12. A. Liniger, A. Domahidi, and M. Morari, "Optimization-based autonomous racing of 1:43 scale rc cars," *Optimal Control Applications and Methods*, vol. 36, pp. 628–647, 09 2015.
13. A. Zanelli, A. Domahidi, J. Jerez, and M. Morari, "Forces nlp: an efficient implementation of interior-point... methods for multistage nonlinear nonconvex programs," *International Journal of Control*, pp. 1–17, 2017.