


SHORT PAPER

Open Access



Competing with autonomous model vehicles: a software stack for driving in smart city environments

Julius Bächle^{1†}, Jakob Häring^{1†}, Noah Köhler^{1†}, Kadir-Kaan Özer^{1†}, Markus Enzweiler^{1*}  and Reiner Marchthaler¹

Abstract

This article introduces an open-source software stack designed for autonomous 1:10 scale model vehicles. Initially developed for the Bosch Future Mobility Challenge (BFMC) student competition, this versatile software stack is applicable to a variety of autonomous driving competitions. The stack comprises perception, planning, and control modules, each essential for precise and reliable scene understanding in complex environments such as a miniature smart city in the context of BFMC. Given the limited computing power of model vehicles and the necessity for low-latency real-time applications, the stack is implemented in C++, employs YOLO Version 5 s for environmental perception, and leverages the state-of-the-art Robot Operating System (ROS) for inter-process communication. We believe that this article and the accompanying open-source software will be a valuable resource for future teams participating in autonomous driving student competitions. Our work can serve as a foundational tool for novice teams and a reference for more experienced participants. The code and data are publicly available on GitHub.

Keywords: Autonomous model vehicle, Software architecture, Embedded real-time systems, Bosch Future Mobility Challenge, Autonomous driving

1 Introduction

Autonomous driving is one of the most significant challenges of our time, with the potential to revolutionize transportation, enhancing safety and efficiency on our roads. The development of autonomous vehicles is a complex task that requires the integration of various advanced technologies, including computer vision, machine learning, and robotics. One effective way to inspire young people to pursue careers in this field is by providing opportunities to learn and experiment with autonomous model vehicles, as depicted in Fig. 1. These models enable students to grasp the fundamental requirements of an autonomous

system, such as perception, planning, and control. Furthermore, they provide the opportunity to dive deeper into topics such as computer vision, machine learning, and control theory, fostering independent research projects and the development of new algorithms and techniques.

This paper presents a comprehensive software stack for autonomous model vehicles, utilized during the Bosch Future Mobility Challenge (BFMC) 2023 [1]. The BFMC serves as a competitive platform for students to share knowledge, build connections, and showcase their work, thereby motivating them to enhance their skills and knowledge. However, for students new to the field, developing a deep understanding of the software stack used in high-level competitions can be challenging due to the scarcity of accessible prior work. This paper and the provided code-base aim to bridge this gap.

The software stack is designed to perform a wide range of tasks while maintaining affordability by using a min-

*Correspondence: markus.enzweiler@hs-esslingen.de

¹Institute for Intelligent Systems, Faculty of Computer Science and Engineering, Esslingen University of Applied Sciences, Flandernstraße 101, Esslingen am Neckar, 73732, Germany

[†]Equal contributors



Figure 1 1:10 scale model car including sensors, compute platforms, and actuators

imal number of sensors. It is implemented in C++, utilizes YOLO Version 5 s [2] for environmental perception, and employs the state-of-the-art Robot Operating System (ROS) for inter-process communication. We provide a detailed overview of the hardware and software architecture, with a focus on the interaction between perception, behavior, and trajectory planning. The paper demonstrates how object and lane recognition approaches can be adapted to model vehicles. Additionally, we discuss the decision-making process in an autonomous vehicle and the methods for calculating actions and trajectories.

2 Architecture

In the pursuit of autonomous mobility for the BFMC 2023, each participating team developed a model vehicle with custom hardware and software architectures. The first section presents the hardware architecture, encompassing physical components such as embedded computing boards, various sensors, and critical actuators. The design prioritizes simplicity and cost-effectiveness to ensure the system functionality and accessibility. The second section details the software architecture, focusing on the efficient allocation of tasks to individual computing units to optimize resource utilization. We provide insight into the rationale behind our design decisions and highlight special features of our software architecture, offering a valuable reference point for future teams in autonomous driving student competitions.

2.1 Hardware architecture

All teams admitted to the BFMC 2023 received a 1:10 scale model car, which included a Raspberry Pi 4 Model B [3] and an STM32 Nucleo F401RE microcontroller [4], as shown in Fig. 2. To optimize performance while maintaining budget and simplicity, several components were

customized. Special wheel speed sensors were installed to measure the traveled distance more accurately. Additionally, an Nvidia Jetson TX2 [5], a high-performance and power-efficient embedded computing device, was integrated to accelerate the vehicle's perception and data processing capabilities. The primary sensor is an Intel RealSense D435 camera [6], which features an RGB color sensor and two infrared cameras for stereo vision, providing depth information about the vehicle's surroundings.

Figure 2 illustrates the overall hardware architecture. The camera is directly connected to the TX2, enabling rapid processing of the video stream. Detected objects and lane markings are then transmitted to the Raspberry Pi via User Datagram Protocol (UDP). The Raspberry Pi processes data from the TX2, wheel speed sensors, and the Inertial Measurement Unit (IMU). After processing, actuator commands are sent to the Nucleo board, which controls the longitudinal movement using the motor and motor driver, and the lateral movement using the steering servo.

2.2 Software architecture

The software architecture for the vehicle is designed to distribute tasks across the available computing units, optimizing resource utilization and improving system responsiveness. The software stack is divided into three main blocks: perception, planning, and acting. Each block is assigned to a specific computing unit to facilitate efficient data management and minimize communication overhead. Figure 3 gives an overview of the architecture, which is explained in this section.

Perception The object detection and lane detection tasks are implemented on the graphics processing unit (GPU) to enhance processing efficiency. Lane detection employs a deterministic approach, while object detection utilizes a neural network on the GPU. Data exchange between the GPU and the main processing unit employs the User Datagram Protocol (UDP) due to its lightweight nature. The real-time messages include the class, bounding box, and distance of the detected objects. The lane detection message contains information about the curvature and the distance to the center of the lane. As false detections are filtered before transmission, data transfer is minimized, allowing the behavior planning to react almost immediately to all received messages. Detailed explanations are provided in Sect. 3.

Planning The main computing unit, equipped with a Raspberry Pi, employs the Robot Operating System (ROS) Noetic [7] for robust data communication. Input data from the perception module, IMU, and wheel speed sensors are analyzed to formulate driving strategies. Target steering angles and speed signals are determined and transmitted to the Nucleo board via the Universal Asynchronous

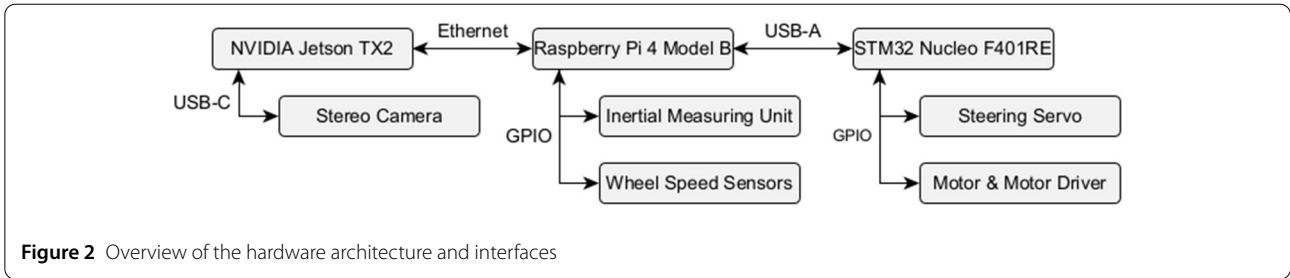


Figure 2 Overview of the hardware architecture and interfaces

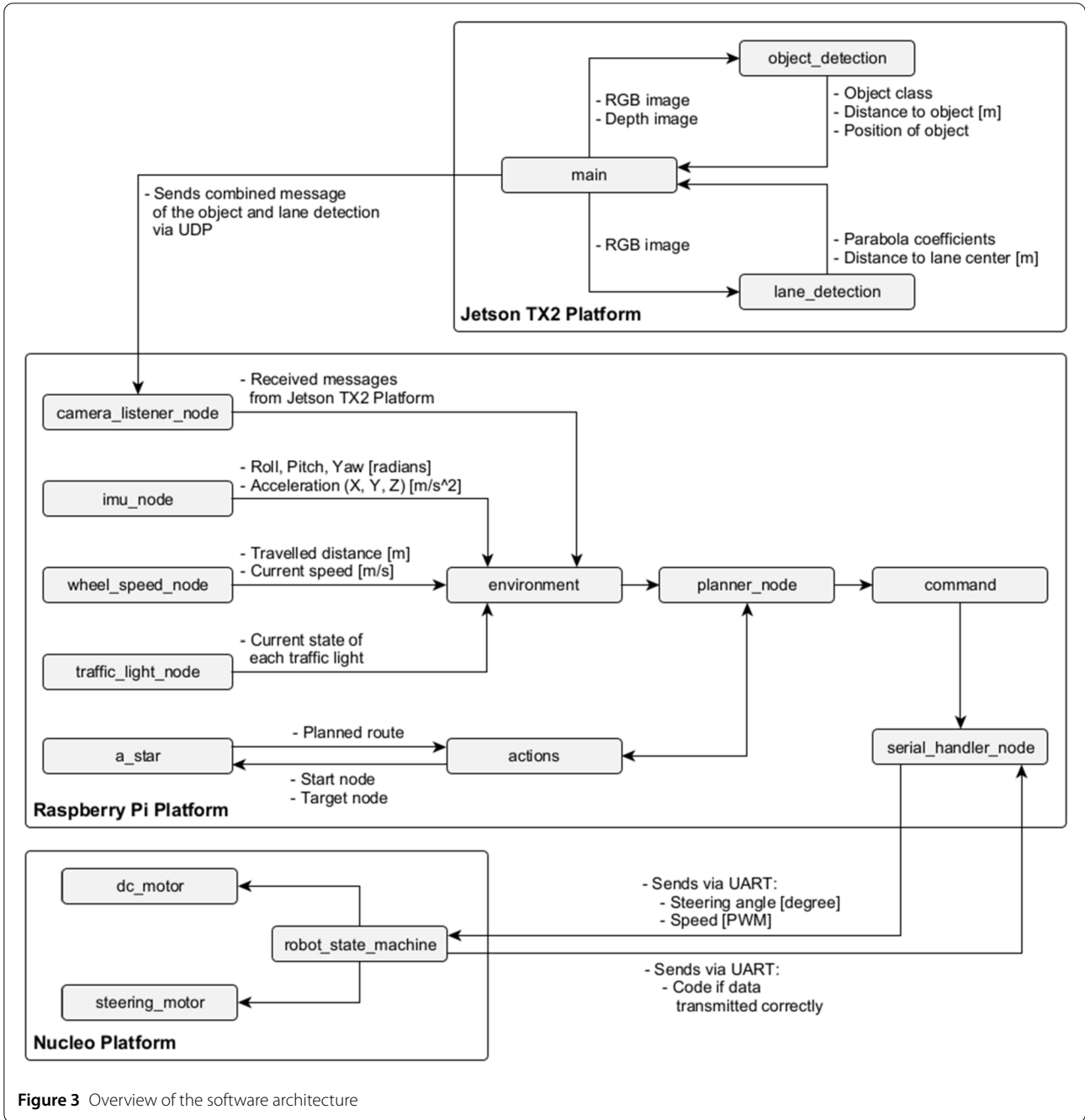


Figure 3 Overview of the software architecture

Receiver/Transmitter (UART) protocol. Detailed explanations are provided in Sect. 4.

Acting The Nucleo board receives control signals from the main computing unit and utilizes a PID controller to adjust the actual speed based on the target speed. The steering angle is set to a target value, constrained within defined boundary conditions. Details are given in Sect. 4.

3 Perception

This section examines the design of our perception system, covering camera setup, object detection, performance optimization on the Raspberry Pi, and lane detection algorithms, all tailored towards the BFMC environment. Key points include sensor choice and camera alignment, dataset creation, neural network selection, architecture and training, task parallelization, and efficiency enhancement methods. Lane detection is addressed through pre-processing steps and histogram-based techniques. This overview aims to provide a clear understanding of the systems behind the functioning of autonomous model vehicles, especially in competitive settings like the BFMC.

3.1 Camera setup

The Intel RealSense camera utilized in this system features an RGB sensor with resolutions up to 1920×1080 px [6]. To balance performance and processing speed, we operate the camera at a resolution of 960×540 pixels at a sampling rate of 30 Hz. The depth images are spatially aligned with the RGB color images. The RGB module is positioned on the left side of the camera, providing comprehensive capture on the left side. To ensure traffic signs on the right side are detected at shorter distances, the camera has been rotated accordingly.

3.2 Object detection

Dataset A test track was constructed to evaluate the software stack and capture images for training the object detection network (see Fig. 4). The test track was designed to closely follow the rules of the BFMC [1] and includes a roundabout, two intersections, and a parking lot. It is complemented by signs with 3D-printed poles and two pedestrian dummies. Although the test track is sufficient to analyze most scenarios, it is approximately four times smaller than the original competition track.

The dataset used to train the object recognition model consists of 4665 images captured while driving on the test track and during the competition. Additionally, 774 images from videos provided by Bosch were included. These images were taken from vehicles in previous competitions, using different cameras, resolutions, and aspect ratios. Despite these variations, incorporating these images improved perception in scenes that were difficult to recreate on our own test track, such as motorway exits.

Overall, the model detects 15 classes, including crosswalks, stop signs, and priority signs. Additionally, dynamic traffic participants (e.g., cars and pedestrians) and static obstacles are recognized. The model also identifies stop lines and parking spaces for junctions and parking situations.

Model selection Implementing an efficient and robust object detection system is paramount in the development of autonomous model vehicles for competitions such as the BFMC. One of the critical tasks in this domain is the identification of obstacles, paths, and other relevant environmental features. After considering various algorithms, YOLOv5s [2, 8], a variant of the YOLO family of object detection models, was selected for this purpose due to its strengths and suitability for the specific requirements of the 1:10 scale autonomous model vehicle.

YOLOv5s, as the second smallest and fastest model in the YOLOv5 series, offers a balance of speed and accuracy, making it suitable for real-time object detection in resource-constrained environments like model vehicles [9]. The model's architecture, building upon the advancements of its predecessors, incorporates several features that meet the high-performance demands of autonomous navigation while remaining computationally efficient [10]. It includes optimizations like anchor box adjustments and advanced training techniques [11], making it suitable for real-time object detection in autonomous vehicles. Its ability to detect objects of various sizes and under different conditions is crucial for the safety and reliability of autonomous driving systems [12].

In addition to these technical advantages, the widespread adoption and active development community surrounding the YOLO family of models provide resources for support and further enhancements. The availability of pre-trained models, extensive documentation, and a large user community contribute to the development process and facilitate the implementation of advanced features and improvements.

Methodology The training of YOLOv5s involved optimizing parameters such as the number of epochs, batch size, and the use of autobatching. The performance was evaluated based on four key metrics: box loss, object loss, class loss, and Mean Average Precision (MAP). These metrics collectively offer insights into the model's accuracy, reliability, and efficiency in detecting and classifying objects. Box loss emphasizes the spatial accuracy of object detection, measuring the precision of the predicted bounding boxes against the ground-truth boxes. Object loss addresses the discernment between objects and non-objects, evaluating the model's ability to detect and distinguish objects from the background. Class loss measures the accuracy of categorizing detected objects into the correct categories. A well-trained model should ideally have low scores



Figure 4 Test track at the Institute for Intelligent Systems at Esslingen University of Applied Sciences

Table 1 Parameters, validation losses, and MAP of different training / validation runs

Epochs	Batch-size	Box loss	Object loss	Class loss	MAP
75	64	0.0121	0.0040	0.0004	0.8653
75	32	0.0244	0.0064	0.0009	0.6657
100	32	0.0129	0.0039	0.0005	0.8638
100	autobatch	0.0156	0.0049	0.0008	0.8194
300	64	0.0123	0.0040	0.0005	0.8713
300	32	0.0123	0.0043	0.0005	0.8696
500	32	0.0113	0.0044	0.0009	0.8688

on all three types of losses, indicating high precision and accuracy in both detecting and classifying objects in images.

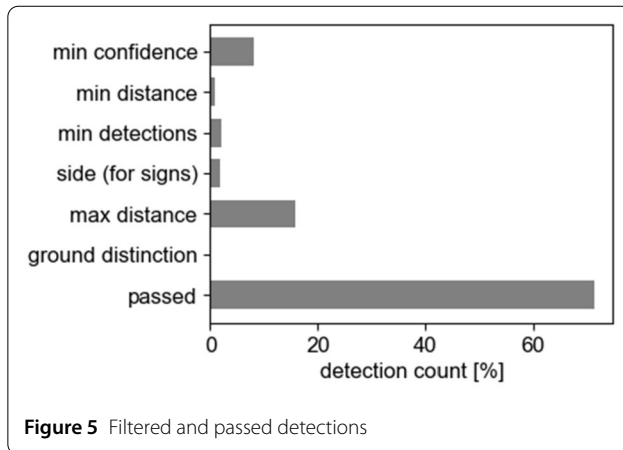
Training This section focuses on the different training parameters and the definition of various losses regarding the YOLOv5 object detection model. Table 1 provides a brief overview of the performance differences resulting from the parameter adjustments.

The analysis of various training configurations of YOLOv5s for the BFMC underscores the importance of carefully selecting training parameters. The configuration with 300 epochs and a batch size of 64 emerged as the most effective, striking an optimal balance between training duration and model performance.

This setup not only achieved the highest MAP but also maintained low loss values, making it the preferred choice for tasks requiring high precision in object detection, such

as detecting small model cars. This insight can guide future training procedures in similar applications, emphasizing the need for a balanced approach to training deep learning models.

Filtering of misdetections In addition to missing known objects (false negatives), recognizing false objects (false positives) is also a significant problem in the perception of neural networks. These misdetections can lead to incorrect reactions of the model vehicle, so detections are filtered using prior scene knowledge before being forwarded to behavior planning. To analyze the effectiveness of the filters in more detail, the number of detections removed during a drive on the test track was recorded. The applied filters and the proportion of valid detections that passed our filters are shown in Fig. 5. The filter functions are applied sequentially from top to bottom.



Before the detections are counted, a confidence threshold and a non-maximum suppression (NMS) threshold are applied, as is typical for object detection networks.

In the next step, all detections with a measured distance of zero are filtered out. The distance for a bounding box detection is determined as the average distance in the area covered by the bounding box, estimated via the available depth image. False positives typically occur inconsistently over time, so only objects recognized in several consecutive frames are considered valid.

Given the context of the BFMC, where traffic lights and most traffic signs appear on the right-hand side, those recognized on the left-hand side are ignored. Additionally, detections are filtered using distance thresholds between one and three meters, allowing the system to ignore distant objects that are not yet relevant to the vehicle. This check for maximum distance, combined with the minimum confidence threshold, accounts for approximately 24% of the filtered detections and has been empirically estimated.

The final filter ensures that detected objects match the expected spatial relationship in the scene. For example, objects are assumed to be ground-based and traffic signs are assumed to be at a certain height above the ground. For each bounding box, we estimate the distance measured via the depth image and a second distance measure obtained from the known camera geometry, punishing deviations between both estimates.

3.3 Lane detection

The lane detection algorithm follows an engineered approach rather than relying on machine learning. It is divided into two main phases: preprocessing, which includes tasks like cropping and transforming the image, and detection, in which lane markings are identified using search boxes. The result of the lane detection algorithm is the curvature of the lane markings and the offset of the vehicle from the middle of the lane.

Preprocessing The first step of the lane detection algorithm's preprocessing routine is to crop the RGB input image, focusing on a pre-selected region of interest (ROI) containing the road area. This static cropping operation eliminates the need to dynamically ascertain a ROI for each frame, thus preventing any potential performance overhead. By reducing computational complexity in subsequent processing stages, this approach enhances the efficiency and accuracy of the algorithm by focusing on the relevant portion of the image.

The cropped section of the image is then converted into a bird's eye view (BEV) format. This conversion aids in the identification of lane markings by presenting a more intuitive representation of the road layout, allowing the algorithm to interpret the spatial connections more effectively between lane markings and the location of the vehicle. The transformation to BEV is accomplished by utilizing homographies to map points from one image plane to another. The homography parameters are established through corresponding points between the initial image and a BEV reference image, ensuring precise alignment and reconstruction of the road scene (see Fig. 6a and Fig. 6b).

To enhance computational efficiency and emphasize intensity-based lane detection, the BEV image is converted to grayscale. Color information is often redundant for lane detection, as intensity-based contrast between lane markings and the surrounding road surface suffices for effective differentiation. The linear transformation method is employed for grayscale conversion, preserving intensity data and removing extraneous color information by computing a weighted average of the red, green, and blue channels.

Further refining the representation of the road layout, the grayscale BEV image is binarized in a two-step process. First, Canny edge detection [13], a technique for effectively identifying image edges, is applied. Canny edge detection is particularly well-suited for intensity-based lane detection because it robustly extracts edges while minimizing noise and preserving essential features such as lane markings. The resulting edge map, which highlights the boundaries between lane markings and the surrounding road surface, serves as a critical input for subsequent lane detection stages.

Second, a mask is generated from the pre-Canny image to eliminate unnecessary detections and additional noise. This is done by applying a threshold that binarizes each pixel into either white or black. The mask is then superimposed on the post-Canny edge image to produce a clear representation of the road layout. The result is shown in Fig. 6c.

In the final preprocessing step, a histogram is constructed by counting the number of white pixels per column of the bottom 25 pixel rows of the grayscale image. This ROI corresponds to the road surface where the origins of lane markings are typically located from the vehi-

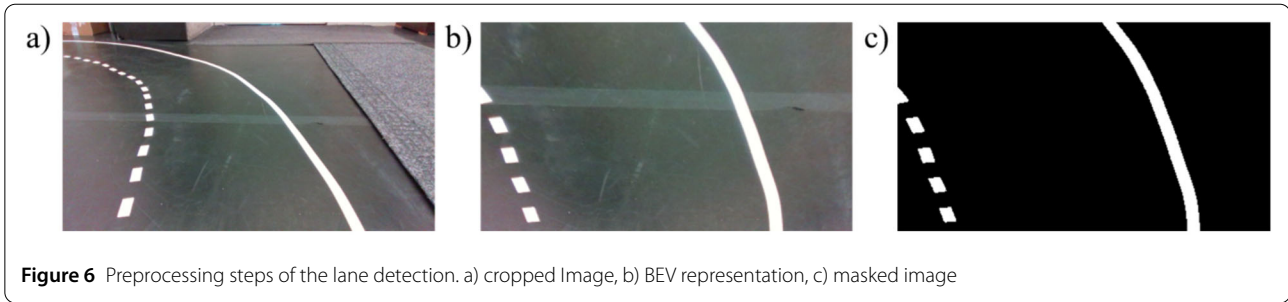


Figure 6 Preprocessing steps of the lane detection. a) cropped Image, b) BEV representation, c) masked image

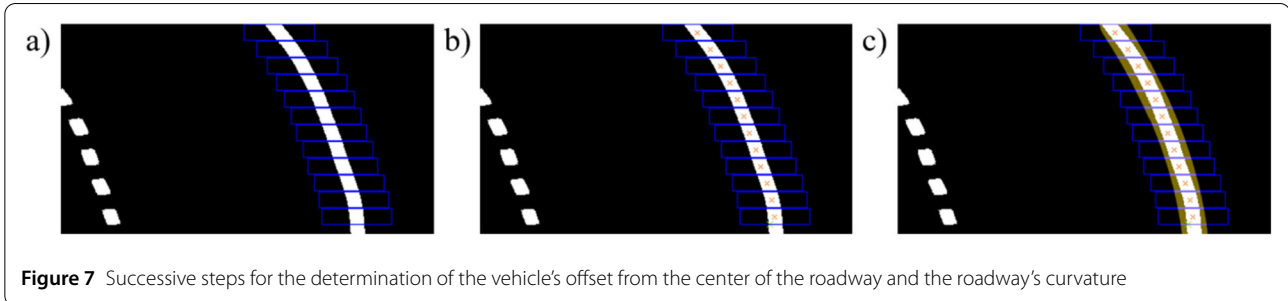


Figure 7 Successive steps for the determination of the vehicle's offset from the center of the roadway and the roadway's curvature

cle's perspective. The histogram provides a statistical representation of the intensity distribution across the image columns, allowing the identification of prominent peaks that correspond to lane markings. This information serves as crucial input to the subsequent lane detection stages (see Fig. 7).

Together, these preprocessing steps prepare the input image for the lane detection algorithm, effectively transforming the raw camera data into a format suitable for robust and accurate lane detection, similar to the approach used in [14].

Detection During the second stage, the algorithm identifies the lane markings on the road. Utilizing the origins of the lane markings at the bottom of the image, identified through the preprocessing routine, the algorithm scans the binary BEV image from bottom to top in search of lane markings. This scanning process employs search boxes (see Fig. 7a) that start at the identified origins and serve two primary purposes. First, implementing a search box reduces computational requirements by limiting the scope of pixel analysis. Second, using a search box decreases the likelihood of misinterpreting a single pixel and minimizes the effect of pixel-wise errors in the search process.

A histogram is created for the area of each search box. This histogram offers a statistical representation of the intensity distribution of white pixels in the search box across each column, facilitating the identification of significant peaks that correspond to possible lane markings. After filtering out columns lacking sufficient white pixels, the average of the remaining column numbers is used to pinpoint

the center coordinates of the next search box, as visualized in Fig. 7b.

After iterating through the complete image, the row and column values identified from the histogram peaks in each search box are preserved. To approximate the smooth trajectory of lane markings and eliminate outliers, we fit a quadratic parabola to the accumulated x-y pixel pairs using a least-squares fitting approach, similar to the method described in [15] and shown in Fig. 7c. Curve fitting provides a more reliable representation of lane boundaries by smoothing out fluctuations in the x-y-centers of the search boxes and capturing the underlying curvature of the lane markings.

The fitted parabola is then used to calculate the lane trajectory, providing necessary information about the vehicle's lane position and potential departures from the lane. This information is essential for enabling the vehicle to maintain its position within the lane and prevent lane departures, a critical safety feature for autonomous vehicles. The lane detection algorithm outputs the lane boundaries and their curvature, as well as the vehicle's offset from the center of the roadway.

3.4 Performance optimization

As discussed in Sect. 2, the TX2 runs the entire perception stack, fetching camera images, detecting lanes and objects, and communicating the results to the Raspberry Pi. While image retrieval and alignment must be carried out sequentially before processing, lane and object detection can be parallelized. With sequential execution, the entire perception module requires 64 ms from image retrieval to message transmission, as shown in Table 2. When lane and

Table 2 Perception runtime for sequential and parallel execution (average)

Module	Sequential [ms]	Parallel [ms]
Image Retrieval	13	15
Object Detection	36	36
Lane Detection	14	14
All	64	51

Table 3 Perception runtime using different inference engines (average)

Inference engine	Object detection [ms]	Perception module [ms]
ONNX	87	100
TensorRT (FP32)	55	72
TensorRT (FP16)	36	52

object detection are executed in parallel, only 51 ms are required. As lane detection is executed on the CPU and the object detection on the GPU of the TX2, the difference of 13 ms corresponds almost exactly to the execution time of the lane detection (14 ms).

In addition to parallelization, the inference engine can also be optimized to speed up perception. The effect of using an Open Neural Network Exchange (ONNX) neural network model and a TensorRT implementation [8] is investigated. When the ONNX neural network model obtained after training is integrated directly into C++ code using the OpenCV library, object detection requires 87 ms, and the perception module requires 100 ms (see Table 3). However, when the ONNX file is converted into an engine file using the TensorRT library, 55 ms are necessary for object recognition. Further reducing the internal accuracy of the model from 32-bit to 16-bit floating point numbers decreases the processing time to 36 ms, with no noticeable loss of accuracy. Overall, by improving the inference engine, the time required for perception was halved to 52 ms. Since the fastest inference engine was also used for the previous measurements, this number largely corresponds to the parallel execution time from Table 2, with a deviation of one millisecond due to measurement inaccuracies.

4 Behavior and trajectory planning

This section focuses on behavior and trajectory planning. The crucial modules used in this architecture are the “*Environment*”, “*Actions*”, and “*Command*” modules. Starting with the “*Environment*” module, it integrates sensor inputs, refines data through post-processing, and sets precise vehicle states for informed decision-making. The transition to the “*Actions*” module includes mapping vehicle states to actions and orchestrating behavior and trajectory planning. Concluding this comprehensive exploration, the focus shifts to the “*Command*” module, which executes critical commands, controls the vehicle within predetermined limits, and facilitates seamless communication with

the Nucleo board. Together, these three modules constitute the essential framework of behavior planning, enabling the vehicle to navigate, comprehend, and react to its surroundings.

Behavior and trajectory planning is performed at a frequency of 20 Hz, although this frequency could be increased as this part of the software stack involves relatively simple calculations with an overall runtime of approximately 2 ms. It is important to note that synchronization with the slowest component in the system is maintained to avoid bottlenecks; in this case the perception module.

4.1 Environment

The central pillar of the autonomous vehicle architecture is the “*Environment*” module, which combines the input from various sensors into a holistic real-time perception of the vehicle’s surroundings to determine precise vehicle states based on processed sensor data. The following paragraphs discuss the integration and post-processing of multi-sensor inputs and the use of global map features to enhance the sensor data.

Integration and post-processing of multi-sensor inputs

The “*Environment*” module fuses all sensor inputs to enable the autonomous vehicle to make informed decisions based on the current situation. This fusion includes data from the wheel speed sensor, the IMU for spatial orientation, and vehicle-to-everything communication which is available in the BFMC context [1] to obtain the status of traffic lights. Additionally, processed information from the perception module is integrated, including details on lane positions and detected objects.

A crucial aspect of navigating an environment is determining the ego pose of the vehicle. To determine the current ego pose, a dead-reckoning approach is used to dynamically update the x and y positions of the vehicle based on sensor data (distance traveled) from the wheel speed sensor and the current yaw angle of the IMU. The wheel speed sensor, with an accuracy of approximately 0.03 m/1 m, combined with the IMU, has shown sufficient precision for the tasks set within the requirements of the BFMC. The IMU, configured and used with the RTIMULib [16], which provides Kalman-filtered pose data, plays a central role in this process by significantly reducing the rotational drift and noise of the sensor values. The yaw angle exhibits minimal rotation, with a maximum difference of 0.11 degrees over a 10-minute measurement period, as indicated in Table 4, subtly impacting the vehicle’s heading information. Additionally, the rotational drift for roll and pitch angles is eliminated, and noise in the acceleration values is minimized. However, it is worth noting that dead reckoning relies on the integration of incremental changes, so errors can accumulate over time. To counteract these errors, the global map can be used to relocate the vehicle based on known map features such as stop lines.

Table 4 Static IMU measurements over a 10-minute-period

Sensor value	Start	End	Error
Roll [deg]	0.318	0.318	0.000
Pitch [deg]	1.210	1.210	0.000
Yaw [deg]	359.945	0.000	0.055
Acceleration (x) [m/s ²]	-0.216	-0.216	0.000
Acceleration (y) [m/s ²]	0.029	0.049	0.020
Acceleration (z) [m/s ²]	9.679	9.718	0.039

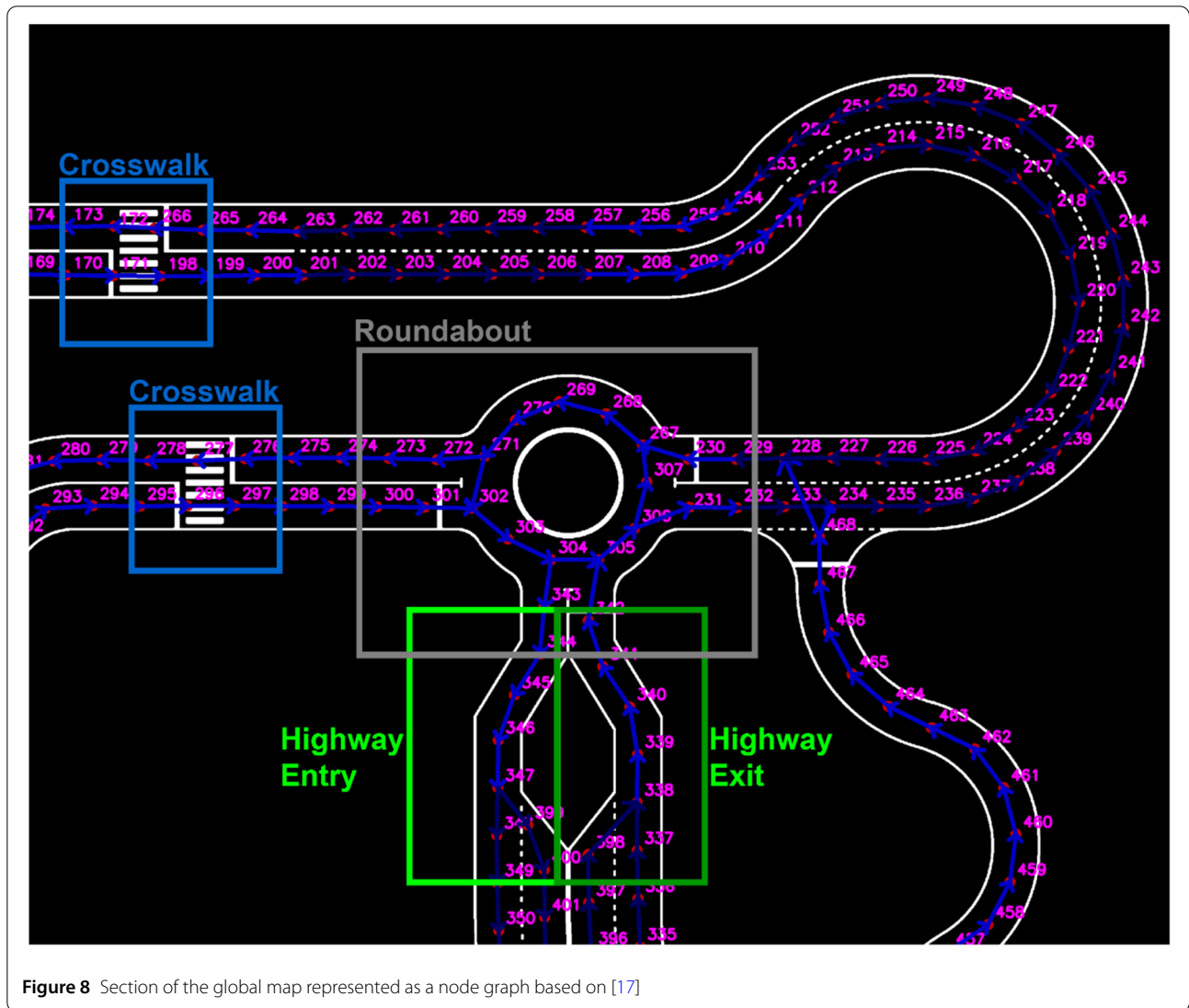


Figure 8 Section of the global map represented as a node graph based on [17]

Use of global map features The BFMC provides a detailed map of the track, as shown in Fig. 8, represented as a node graph in a GraphML file. Each node contains information about its global position, the lane type (dashed or solid), and its connections. For global route planning, an A* algorithm with a Euclidean distance heuristic is used, efficiently calculating the shortest route between two given map nodes. The output of the A* algorithm is a route

containing node information such as ID, position, and a Boolean value indicating the lane type. The algorithm also computes the summed distances between nodes in the planned route to estimate the total length of the route.

In addition, detected objects are subjected to further validation using the global map to ensure accuracy and reliability as follows. The position of detected objects is estimated based on the given distance to the object and the

current pose of the vehicle. Subsequently, this calculated position is validated against defined ROIs, as shown in Fig. 8, which are tailored to the expected location of infrastructure elements obtained from the map. It should be noted that the ROI approach only applies to static objects, such as traffic signs, as dynamic objects, such as pedestrians, can appear anywhere on the map.

4.2 Actions

The “*Actions*” module serves as a fundamental component, mapping vehicle states to specific actions and performing behavior and trajectory planning. Each vehicle state triggers certain actions, such as navigating in a lane, stopping at an intersection, or parking in a free parking space. Within each action, behavior planning and trajectory planning are integrated. Behavior planning involves decision-making processes to determine the optimal course of action in the current situation. Trajectory planning focuses on calculating a path or trajectory that the vehicle should follow to perform the desired action.

For instance, in the crosswalk state/action, behavior planning involves assessing the environment, such as detecting the presence of pedestrians on the crosswalk, and determining appropriate responses. Concurrently, trajectory planning guides the vehicle through the area around the crosswalk, considering factors such as staying in the lane and adjusting speed to ensure a safe and controlled crossing.

A basic action is to navigate in a lane, which requires the calculation of the correct steering angle. Due to the relatively low vehicle speeds of 0.3 - 0.8 m/s, a simple controller approach is sufficient. The steering angle is calculated with an adaptive P-controller using the input from the lane detection module (see Sect. 3.3). Specifically, the steering angle is determined by multiplying the offset to the center of the lane, taken at a preview distance of approximately 0.3 m, by a P-value. Based on the estimated curve coefficient, the P-value is adjusted to enable tighter curves and stabilize the general steering behavior.

Finally, the “*Actions*” module ensures the seamless execution of the planned behavior by converting high-level actions into low-level control signals.

4.3 Command

The “*Command*” module is a central element in managing commands for the autonomous vehicle, overseeing the validation and control of key parameters such as steering and speed commands. This validation prevents extreme inputs, ensuring stability and control over the vehicle’s behavior. Additionally, this module converts ROS messages into UART messages, facilitating efficient communication with the Nucleo board (see Fig. 2) and ensuring smooth integration into the software ecosystem.

5 Experimental evaluation

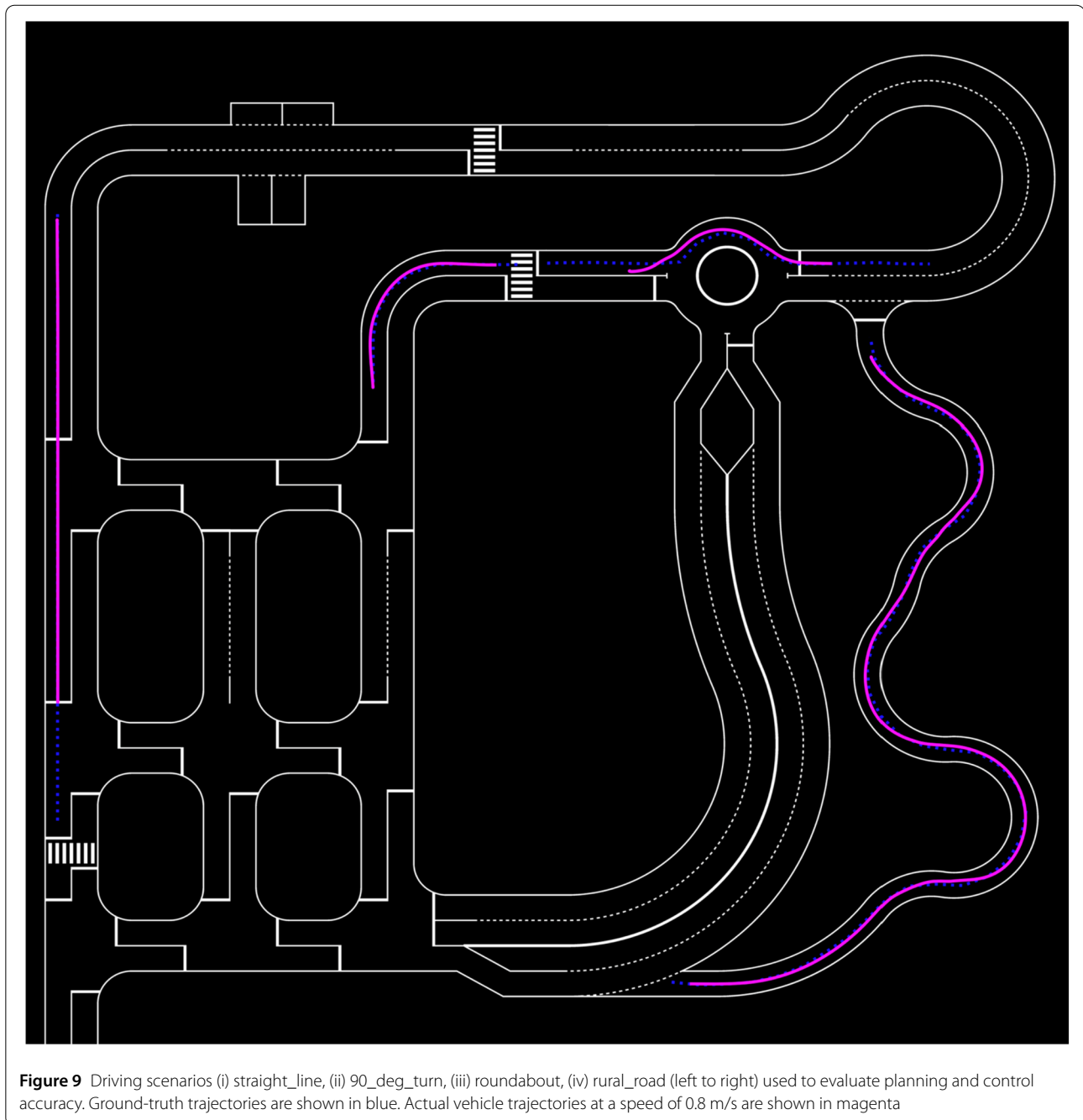
To evaluate the accuracy of our planning and controller, we compare the output of our trajectory control with the respective ground-truth trajectories in various driving scenarios. Here, the ground-truth trajectory is defined as the center of the lane in which the vehicle is driving. This evaluation is conducted within the simulator, since the exact position of our vehicle relative to the ground-truth is not obtainable in real-world driving conditions. The main metric used is the average displacement error (ADE). The ADE is sampled across the trajectories at regular intervals, providing a measure of the deviation between the planned and ground-truth paths. Specifically, at each sampling point along the trajectory, the Euclidean distance between the planned position and the corresponding ground-truth position is calculated. These positional errors are then averaged over the entire trajectory to yield the ADE.

We calculate the ADE for two different vehicle speeds in four different driving scenarios: (i) *straight_line*, (ii) *90_deg_turn*, (iii) *roundabout*, (iv) *rural_road*, as illustrated in Fig. 9. For each scenario, three separate drives are performed and the individual ADEs are averaged. Results are given in Table 5.

Note that the ADE is consistently low in the straight line scenario for both speeds, with an average ADE of 0.004 meters, indicating high accuracy of our trajectory control in maintaining a straight path. In the *90_deg_turn* scenario, the ADE increases slightly to 0.018 meters at 0.3 m/s and 0.023 meters at 0.8 m/s, suggesting that sharp turns introduce a slightly higher error margin, particularly at higher speeds. In the *roundabout* scenario, the ADE is higher, with values of 0.049 meters at 0.3 m/s and 0.064 meters at 0.8 m/s, highlighting the challenge of accurately navigating complex curved paths, especially at increased speeds. The *rural_road* scenario, which involves varied and unpredictable path deviations, shows an average ADE of 0.029 meters for both speeds, indicating that our trajectory control performs reasonably well in more dynamic and less predictable situations.

A second evaluation focuses on computational runtime and overall system delays. Table 6 lists the algorithmic execution times for the main system modules. Given that our driving scenario is relatively simple from a behavior point of view, the computation times are dominated by the perception stack, particularly the neural network.

Finally, the overall system latency, which is the time delay between an event in the scene and the initial response of our vehicle, has been estimated. This overall system latency does not only include the algorithmic execution times but also factors such as data acquisition time, inter-module communication time, network latency, and actuator response times. In our current configuration, the overall system delay is estimated to be approximately 190 ms

**Table 5** ADE in [m] for the driving scenarios under consideration

Driving scenario	Vehicle speed [m/s]	Distance driven [m]	Mean ADE [m]
straight_line	0.3	7.1	0.004
straight_line	0.8	7.1	0.004
90_deg_turn	0.3	3.0	0.018
90_deg_turn	0.8	3.1	0.023
roundabout	0.3	3.0	0.049
roundabout	0.8	3.1	0.064
rural_road	0.3	14.9	0.029
rural_road	0.8	15.0	0.029

Table 6 Average compute times of system modules (without latency considerations)

Module	Time per frame [ms]
Perception	52
Behavior and Trajectory Planning	2
Other	5
Overall	59

on average which does not impose a limit on our vehicle speeds since they are much lower than those of real-world vehicles.

6 Discussion and conclusion

This paper presented a comprehensive software stack designed for autonomous model vehicles, successfully used in the Bosch Future Mobility Challenge. It featured the implementation of a controller, advanced filters to minimize false detections, and the use of the YOLOv5s model alongside lane detection for accurate environmental perception. The coordinated approach to integrating perception, planning, and control demonstrated the system's efficiency and adaptability within the constraints of a model vehicle platform.

However, several limitations should be addressed in future work. Replacing hand-crafted filters to prevent false positives with more principled methods, such as temporal tracking or neural network uncertainty estimates, could improve detection reliability. Moving beyond the simple P-controller to advanced techniques like model predictive control would enhance trajectory tracking. Leveraging ROS capabilities for mapping, localization, and sensor fusion can boost reliability and enable more advanced autonomy features. Exploring more recent neural network models tailored for embedded devices, could provide more accurate and efficient perception.

An important aspect for future exploration is the generalizability of the results. While the software stack is designed to be transferable to other competitions, empirical evidence or case studies demonstrating its successful application beyond the BFMC are currently limited. We expect to use this stack in further competitions and will gain more experience, which we plan to report in future work. In that regard, scalability is another area that needs exploration. The scalability of the software stack for larger, more complex environments or more sophisticated models is not addressed in this paper. There could be limitations when scaling up from miniature smart cities to larger or more dynamic settings.

The modular architecture offers a versatile platform for future enhancements. While it provides a solid foundation, continued research will be crucial to push model vehicle autonomy closer to that of their full-scale counterparts. Collaborative efforts between industry and academia,

through challenges like the BFMC, provide an ideal testbed to rapidly advance this exciting field.

Acknowledgements

We would like to thank the Faculty of Computer Science and Engineering and the Institute for Intelligent Systems (IIS) at Esslingen University of Applied Sciences for providing us with resources for this project. They also provided us with financial support for the procurement of parts, and the trip to the student competition in Cluj, Romania. We would like to thank Robert Bosch GmbH for organizing the BFMC, the temporary loan of the model vehicle, the informative meetings with technical experts, and the accommodation during the competition. We acknowledge support by the state of Baden-Württemberg through bwHPC, which provided the necessary compute resources for model training.

Author contributions

JB, JH, NK and KO contributed equally to the design and conduct of the research and the writing of the manuscript. ME and RM supported and supervised the research and revised the manuscript. All authors read and approved the final manuscript.

Funding

No funding was received for conducting this study.

Data availability

The dataset used for training the object detection network is available at: <https://universe.roboflow.com/team-driverles/bfmc-6btkg>

Code availability

The source code of the presented software stack is available at: https://github.com/JakobHaeringer/BFMC_DriverIES_2023

Declarations

Competing interests

The authors declare that they have no competing interests.

Received: 21 April 2024 Revised: 5 July 2024 Accepted: 5 August 2024
Published online: 14 August 2024

References

1. Robert Bosch GmbH, Bosch Future Mobility Challenge (2023). [Online]. Available: <https://boschfuturemobility.com/>. Accessed 21 December 2023
2. G. Jocher, YOLOv5, Ultralytics Inc. (2020). [Online]. Available: <https://docs.ultralytics.com/de/models/yolov5/>. Accessed 21 December 2023
3. R. Pi, Datasheet - Raspberry Pi 4 Model B (2019). [Online]. Available: <https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf>. Accessed 21 December 2023
4. STMicroelectronics, NUCLEO-F401RE (2023). [Online]. Available: https://www.st.com/resource/en/data_brief/nucleo-f401re.pdf. Accessed 21 December 2023
5. NVIDIA Corporation Jetson TX2-Module (2023). [Online]. Available: <https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-tx2/>. Accessed 21 December 2023
6. I. Corporation, Intel RealSense Depth Camera D435 (2023). [Online]. Available: <https://www.intelrealsense.com/depth-camera-d435/>. Accessed 21 December 2023
7. S. Loretz, ROS Noetic Ninjemys, Open Source Robotics Foundation Inc. (2020). [Online]. Available: <https://wiki.ros.org/noetic>. Accessed 04 January 2024
8. N. van der Meer, YOLOv5-TensorR (2022). [Online]. Available: <https://github.com/noahmr/yolov5-tensortr>. Accessed 02 December 2023
9. G. Jocher, YOLOv5, Ultralytics Inc. (2020). [Online]. Available: <https://github.com/ultralytics/yolov5>. Accessed 21 December 2023
10. A. Bochkovskiy, C.-Y. Wang, H.-Y.M. Liao, YOLOv4: Optimal Speed and Accuracy of Object Detection (2020). <https://doi.org/10.48550/arXiv.2004.10934>
11. R.-A. Bratulescu, R.-I. Vatasoiu, G. Sucic, S.-A. Mitroi, M.-C. Vochin, M.-A. Sachian, Object detection in autonomous vehicles, in *2022 25th*

International Symposium on Wireless Personal Multimedia Communications (WP/MC), Herning, Denmark (2022). <https://doi.org/10.1109/ICSET59111.2023.10295116>

12. J. Redmon, D. Santosh, G. Ross, F. Ali, You only look once: unified, real-time object detection, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA (2016). <https://doi.org/10.48550/arXiv.1506.02640>
13. J. Canny, A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.* **PAMI-8**(6), 679–698 (1986)
14. Y. Ding, Z. Xu, Y. Zhang, K. Su, Fast lane detection based on bird's eye view and improved random sample consensus algorithm. *Multimed. Tools Appl.* **76**, 22979–22998 (2017)
15. J. Wang, F. Gu, C. Zhang, G. Zhang, Lane boundary detection based on parabola model, in *The 2010 IEEE International Conference on Information and Automation*, Harbin, China (2010). <https://doi.org/10.1109/ICINFA.2010.5512219>
16. RPi-Distro, RTIMULib (2015). [Online]. Available: <https://github.com/RPi-Distro/RTIMULib>. Accessed 19 December 2023
17. Robert Bosch GmbH, Bosch Future Mobility Challenge (2023). [Online]. Available: <https://github.com/ECC-BFMC>. Accessed 12 February 2024

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Submit your manuscript to a SpringerOpen[®] journal and benefit from:

- ▶ Convenient online submission
- ▶ Rigorous peer review
- ▶ Open access: articles freely available online
- ▶ High visibility within the field
- ▶ Retaining the copyright to your article

Submit your next manuscript at ▶ [springeropen.com](https://www.springeropen.com)
