UDC: 629.3.014.9
DOI: http://dx.doi.org/10.7708/ijtte2021.11(4).06

AUTONOMOUS DRIVING CYCLE MODELING, SIMULATION AND VALIDATION ON 1:10 SCALE VEHICLE MODEL PLATFORMS

Csanád Ferencz[1], **Máté Zöldy**[2]
[1,2] *Department of Automotive Technologies, Faculty of Transportation and Vehicle Engineering, Budapest University of Technology and Economics, Stoczek J. u. 6., 1111 Budapest, Hungary*

Received 4 May 2021; accepted 16 August 2021

Abstract: In the present research paper, the authors provided a comprehensive overview about the R&D possibilities and processes in the field of autonomous vehicle testing and validation, an exhaustive investigation concerning an autonomous vehicle driving cycle, by developing not only camera-based traffic-sign and lane markings detection and tracking algorithms, but also the implementation and simulation of these, as well as the verification and validation procedures on 1:10 scale vehicle model platform, realizing and reproducing thus a complete embedded system development life cycle.

Keywords: autonomous vehicles, traffic-sign detection, lane detection and tracking, testing, validation.

1. Introduction

Today's vehicles already have several driver assistant systems and in the near future, highly automated vehicles will also appear in road transport. Higher automation levels rely on disruptive technologies that cannot be tested and approved in the former way. To be able to guarantee future road safety also disruptive testing and validation methods are required (Szalay *et al.,* 2017).

The present paper aims to present such a method, specifically to validate and verify a digital camera-based lane and traffic-sign detection algorithm developed in virtual environment, through experimental testing in autonomous driving cycles, implemented on 1:10 scale vehicle model platform as part of an embedded system's software.

As far as the methodology is concerned, 1:10 scale vehicle model platforms were modeled, as well as a simulation was created in a virtual simulation environment, followed by the development of a digital camera-based lane tracking and traffic-sign recognition algorithm, then the experimental testing of the created system in autonomous driving cycles, concluded with the embedded system software validation and verification.

2. Virtual Simulation Environment

In this section the PreScan simulation environment and its connection to Simulink is presented, besides a simple lane detection and tracking algorithm.

Moreover, the implementation of a global state machine in Simulink is shown as

[2] Corresponding author: zoldy.mate@kjk.bme.hu

well, necessary to realize both virtual and experimental testing on the developed system functionalities.

2.1. PreScan

PreScan is a simulation, development and evaluation virtual software environment within autonomous vehicles which can actually see the surroundings in which they are driving and can subsequently respond to it. Fig. 1 below gives an impression of the different engineering tasks and phases carried out using PreScan: building a relevant scenario, adding the appropriate control systems, modeling the sensor system, running the experiment (Ferencz and Zöldy, 2020).

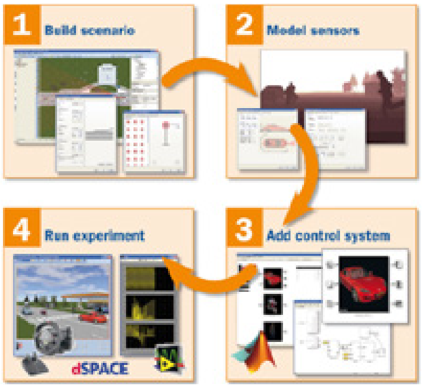

Fig. 1.
Modeling with PreScan
Source: (Tass International, 2021)

For the purposes mentioned before, sensor models containing real physical relationships can be used, in order to design scenarios based on real-life data. Once a certain concept has qualified, PreScan can also be used to check how robust it is under less ideal circumstances, making it a great tool for benchmarking various control system concepts, as it is connected to MATLAB/Simulink.

Generally speaking, PreScan truly adds value when being used for concept studies, where today's typical task is to evaluate different sensing systems or to evaluate different sensor fusion concepts (e.g., if radar with vision or radar with GPS would be the preferred combined system). Within PreScan all technologies and engineering disciplines can be seamlessly integrated since the prime interface is based on MathWorks' MATLAB/Simulink.

2.2. Control Algorithm

PreScan helps us to develop, test and debug our control algorithms prior to uploading it to the actual vehicle. It would be a more difficult and time-consuming task if we had to test and debug our Simulink code in an already compiled C++ form. Fig. 2 shows the built virtual environment, essentially the same as the one used later in the case of vehicle model platforms.

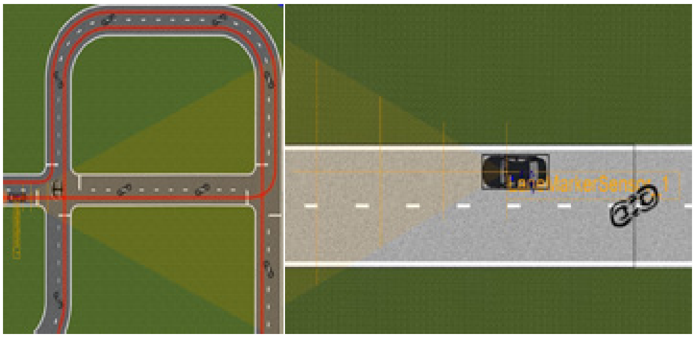

Fig. 2.
The Built PreScan Virtual Environment with Lane Marker Sensor
Source: (Own work, 2021)

A lane marker sensor is added to the model, representing the camera and image processing algorithm on the real vehicle, providing information about the lane lines present on the road as intersections between the lane lines and scan lines relative to the sensor. The Fig. 3 below shows the Simulink environment. PreScan automatically creates the *SELF*, *PathFollower*, *LandMarkerSensor* and *Dynamics_Simple* blocks. The *TechnoDriversControl* subsystem contains our custom control algorithms. So far, its input is the *LaneMarkerSensor* data and outputs the steering command, which is the input for the *Dynamics_Simple* block (Tass International, 2021).

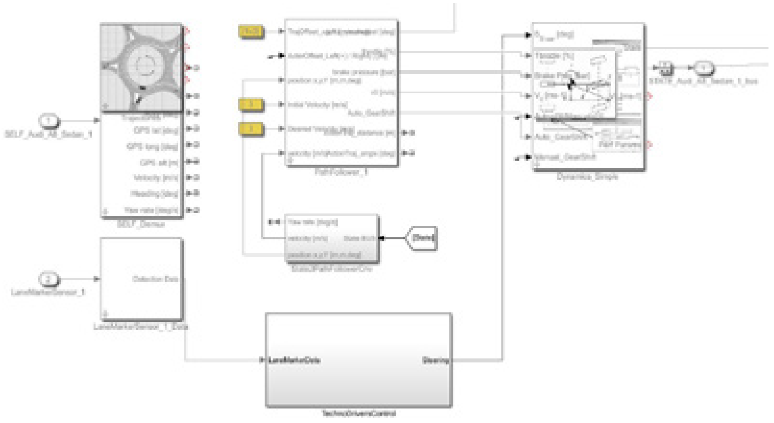

Fig. 3.
The Created Simulink Environment
Source: (Own work, 2021)

The Fig. 4 below shows the implementation of a simple lane keeping model. Its inputs are data chosen from the *LaneMakerData* bus: *DistanceFromScanCenter* and *SliceDistance* for each scan line. This data is grouped as left lane *X*, left lane *Y*, right lane *X* and right lane *Y* coordinates.

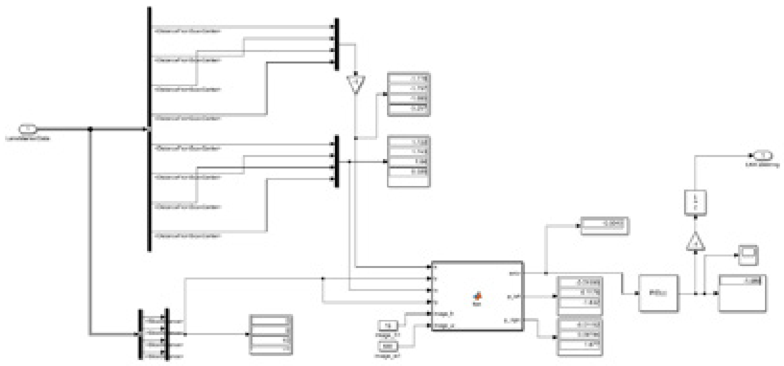

Fig. 4.
Simple Lane keeping Model Implementation
Source: (Own work, 2021)

In a MATLAB function *Eq. (1)* and *Eq. (2)* fits a second order polynomial to the *ly*, *lx*, *ry*, *rx* data points with the *polyfit()* function that will represent the left lane, and also same for the right lane:

$$p\_left = polyfit\ (ly, lx, 2) \tag{1}$$

$$p\_right = polyfit\ (ry, rx, 2) \tag{2}$$

This step was made because the Simulink model in the real vehicle will receive a vector containing the coefficients of the polynomial curve detected by the camera and image processing algorithms and then evaluates the *X* values of the defined polynomials at the given *Y* values, which are determined by the scanning lines. It calculates an average error from the left and right *X* data and outputs it to the PID controller.

One lane marker sensor can only output the date of four scanning lines (see *Fig. 5*), we can use more of these lane marker sensors if a more accurate curve is needed, but it is good enough for our purposes now.

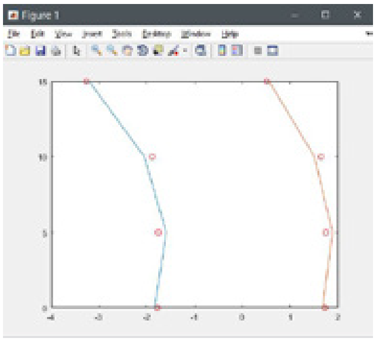

Fig. 5.
Global State Machine, MathWorks Stateflow
Source: (Own work, 2021)

Considering the global state machine, we use the Stateflow's Flow Chart state machine tool, where, based on input signals and conditions, the adequate output can be created. For example, if the camera recognizes a stop traffic sign, also seeing the line corresponding to it, then the vehicle must be stopped for 3 [s]. The logic in this sentence is the condition, the inputs are the stop sign and the line, the output is the control command (brake for 3 [s]).

3. Experimental Measurement and Testing

Besides testing our functions and algorithms in virtual environment and on publicly available datasets, we performed also experiments under real circumstances. For this purpose, we have built two measurement vehicle model platforms equipped with digital camera units, thus allowing us to test our algorithms under real circumstances in real-time. Here the calibration of sensors and that of the whole system is of key importance (BME, 2020).

Consequently, after the simulation phase we continue the exhaustive investigation with experimental testing the vehicle model platforms in different traffic situations, or with the so-called laboratory testing. Apart from the operation of components and different systems, it is also important to test reliability, durability and cyber security (Tollner *et al.,* 2019).

Laboratory tests lead to much more relevant information of these properties than simulations. In our approach laboratory types for dedicated automotive testing can be categorized as follows:

- Technology research laboratory, where the basic research, the enhancement of existing and the development of new systems for autonomous vehicles is carried out, where the basic operation principles are being tested;
- Component analysis laboratory, or so-called HIL (hardware-in-the-loop) labs, where a vehicle specific function can be tested and analyzed, like the video-based environment perception, the challenge being to simulate the missing environmental conditions;
- System integration laboratory, where the cooperation can be tested between different systems, it is the half-way between the HIL and the VIL (vehicle-in-the-loop) tests;
- VIL laboratory, where the complete vehicle is tested, necessary when the safety risk of the test is high, and the vehicle is not prepared enough for open road tests, the biggest challenge being to simulate the environment around the vehicle (Aradi *et al.,* 2014).

3.1. Measurement Vehicle Model Platform

The initial hardware model consists of two Audi RS4 1:10 scale bodies, as well as their chassis. Further electronic components are detailed below, while the final form of these measurement vehicle models is presented in Fig. 6.

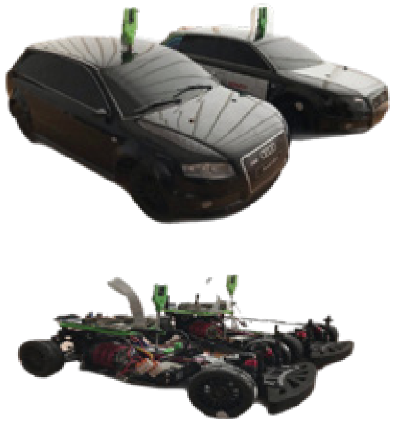

Fig. 6.
Body, Chassis and Electronic Components of the Final Vehicle Model Platforms
Source: (Own work, 2021)

Probably the most important component concerning the present study is the Raspberry Pi 4 Model B/4GB compact single-board microcomputer running on the vehicle's battery. Used basically as a development board, this will handle the image processing and the control as well, so the performance limitations must be kept in mind.

Additionally, an STM32 Nucleo-64 development board will also be used for low-level control, placed right behind, and interconnected with the Raspberry Pi board.

The Raspberry Pi Camera Module v2 will basically be the measurement vehicle's vision system, a high-quality 8-megapixel Sony IMX219 image sensor, custom designed add-on board for Raspberry Pi, featuring a fixed focus lens. It can take 3280x2464-pixel static images, also supporting 1080p30, 720p60 and 640x480p90 video with a 67 [°] view angle. It is attached to the single-board computer by the small sockets on the board's upper surface, using dedicated CSi interface, designed especially for interfacing to cameras (Raspberry, 2021).

The measurement vehicle is powered by a 7.5 [V] DC electric motor, being also equipped with a 4.8 - 6 [V] standard analogue servo for steering angle control during cornering. Alongside our DC motor we will use also a capacitive modular incremental encoder, which senses mechanical motion such as position, speed, distance, direction and it translates into electrical signals.

The necessary power to run the vehicle derives from modern LiPo (lithium-ion polymer) rechargeable battery packs (14.8 [V], 3800 [mAh]), which do not only have a clearly higher capacity than NiMH or NiCd rechargeable battery packs, but they also have a considerably lower weight.

Besides the electric motor, two further important electronic components are needed as well. One is a MOSFET H-bridge motor driver, which enables bidirectional control of the high-power DC brushed motor, supporting a wide 5.5 - 30 [V] voltage range, while delivering a continuous 15 [A]. The other is a DC-to-DC electric power converter, which enables conversion from one voltage level source DC to another, delivering it to sub-circuits, each with its own voltage level requirement different from that supplied by the battery (Conrad, 2021).

3.2. Environment Perception Algorithm

The model's core element is the control software which must control the entire driving cycle process. Through the development period traditional image processing methods were used, rather than more complex machine learning or deep learning algorithms like CNN. On one hand the implemented hardware does not really have the computational power to run such scripts, respectively; on the other hand the simplicity of the problem does not justify the implementation of such solutions. Python 3.x and OpenCV were used within the project.

The lane detection function plays a vital part in every autonomous car, but in the case of this project, it is even more important, being the only input from the environment. A robust lane detection algorithm provides reliable input for the control. There were implemented and tested a few methods explained below, in order to find the one that solves the present problem with the most success.

For the first try Hough Transform with Canny Edge Detection was used in order to detect the lane lines on publicly available test videos, and then we filtered and processed those lines, in order to determine which belongs to the left and right lane lines. Then the average slope was used, the highest point on the image, respectively the values from the previous frame to draw the lane lines on top of the original image (GitHub, 2017).

The algorithm worked fine and provided good results, however, it was only useful in straight lines, so it had to be implemented a method that was able to deal with curves as well (Chuan-En, 2018).

Thereafter, the Sliding Windows method based on Sobel operator was tried, containing six main parts:

- Using color transforms, gradients, etc. to create a threshold binary image;
- Applying a perspective transform to rectify binary image ("birds-eye view");
- Detecting lane pixels and fit to find the lane boundary;
- Determining the curvature of the lane and vehicle position with respect to center;
- Warping the detected lane boundaries back onto the original image;
- Output visual display of the lane boundaries, numerical estimation of lane curvature and vehicle position (Garv, 2020).

A window search takes a histogram of the lower half of the binary image, revealing the neighborhood where the lane lines begin (see Fig. 7).

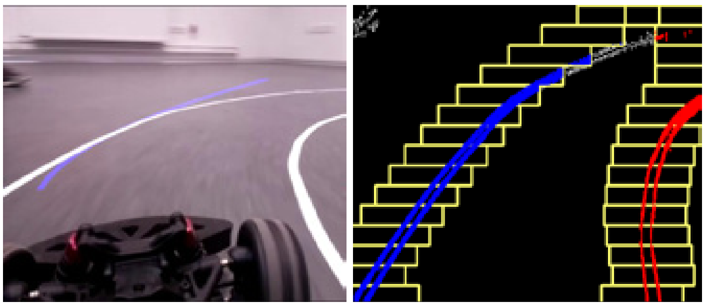

Fig. 7.
Sliding Windows Method based on Sobel Operator, Polyfit Result
Source: (Own work, 2021)

After that, the image is split into horizontal slices, sliding a search window across each slice, finding areas of highest frequency. Once this is found for both left and right lanes, a second-order polynomial was performed with the *polyfit()* function, in order to get the best fit curve on the line. After that, this information was store in a *Line()* class for later use (Wong, 2017).

The method provided great results; however it is computationally expensive, so a more sustainable algorithm had to be found.

In order to reduce the performance needs of the algorithm, the edge detection was changed to Hough Transform with simple white thresholding. While on the control side it turned out that the second-order polynomial is not necessary, only errors and orientation of the vehicle had to be provided. The Sliding Window method never achieved more the 10 fps on the Raspberry Pi, so, even though it is a robust method, it could not be used, due to the issues related to the Hough Transform and the handling of the lines in Hough Space (García *et al.,* 2019).

The different solutions, as well as their advantages and disadvantages are presented in Table 1. Based on the data collected in this table, the best solution can be easily identified, namely the Canny Edge Detection with Histogram Analysis.

Table 1
Comparison among the Tested Lane Detection Algorithms

Lane Detection Algorithm	Advantage	Disadvantage
Hough Transform with Canny Edge Detection	Appropriate working, good results	Only useful in straight lines
Sliding Windows method based on Sobel operator	Appropriate working in curves, straight lines	High computational demand
Hough Transform with white thresholding	Better filtering of the lighting	Never achieved more than 10 fps
Canny Edge Detection with Histogram Analysis	Robust, fast and simple	No issues

This implementation is the combination of the simplest and most effective parts of the previous methods. The Canny Edge Detection had the best results in the pre-processing segment, so it was decided to stick with it. It was also avoided fitting a second-degree polynomial, because it requires a lot of hardware power; however, it provides almost nothing to the control, it is better to use that performance to calculate errors.

As far as the traffic-sign detection function is concerned, the goal is that the model vehicle to successfully detect and classify four types of traffic-signs - main/priority road, stop, pedestrian crossing and parking signs (Bosch Future Mobility, 2019).

Regarding the method to achieve this goal, the task is well defined, and the traffic sign pool is small, only four signs. Deep learning is a general and popular solution to this problem, however considering the processing power limitations, we decided not to take this path, rather to base the algorithm on color threshold, shape detection and classification.

The signs have quite different colors than their typical environment - red, yellow and such blue colors are not expected in the specified area of the maps. After masking the image with the specified threshold, morphological closing is performed with different types of structural elements (ellipse and rectangular) to decrease noise.

The perspective view presented below is still an important part of the script to provide an upper view and to avoid distortions. Line edges are fitted based on the vertical white line; histogram peaks for the linear fitting are shown under the camera image in Fig. 8.

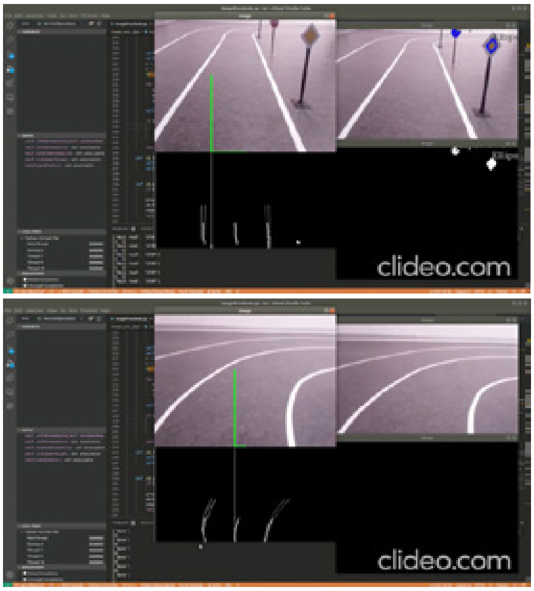

Fig. 8.
Real-time Camera Image of the Traffic Sign and Lane Detection Algorithms' Output
Source: (Own work, 2021)

The signs have different shapes, which can be found with contour searching. To avoid detecting small objects, most likely noise, a minimum arc length for the contour is set. The algorithm then classifies the found contours into typical shapes (triangle, rectangle, ellipse, circle, etc.), draws and collects them.

Based on the colors and shapes, some signs can be classified easily, the stop and main road signs are much different than the others. The two blue ones, park and pedestrian, are a bit harder to correctly classify with this method. To reduce the probability of false negative and false positive detection, bouncing counter is implemented. This helps us to provide more accurate information.

In order to overcome the issue of light-sensitivity template matching can be implemented, in order to obtain higher certainty of the detected sign. To minimize the detection of other objects (noise), we will use the so-called region of interest method, and crop the image at the beginning of the processing. This also reduces the required processing power (Ramesh *et al.,* 1995).

3.3. Vehicle Control System

Creating control models in Simulink is a very effective and simple way to handle complex equations. Simulink models are useful for the simulation of a system, however, if you want to implement your model as an embedded system, building of the model in a toolchain is needed, meaning that it has to be compiled to a corresponding language, for example C++, in order to run parallelly multiple codes and for development as well.

If the code is implemented on a single-board microcontroller, when setting the build process, the toolchain (board type) should be chosen as well as the code language, in this case GNU GCC Raspberry Pi. Moreover, having enabled the "Package code and artifact" property, the system will generate the code in a compressed ZIP file format together with other files essential for compiling, belonging to the main code. The generated code will run and build automatically on the board.

The main goal of the communication strategy's application is to send the adequate control signals from the Raspberry Pi to the STM32 Nucleo-64 board. As an overview, the Raspberry Pi is responsible for the high-level computation (e.g., image processing, positioning and control), while the task of the STM32 Nucleo-64 is to control the DC motor and steering servo, based on control commands from the Raspberry Pi.

The connection between the two above-mentioned microprocessors is a serial communication protocol, physically a cable. The messages in string format are responsible for the command transfer. There are a few types of messages and formats for different types of motion command, shown in Table 2.

These messages sent by Simulink are compiled in C++ code every 0.01 [s] (100 [Hz]), writing data to the serial device, representing the serial communication. There is a hardware support package for Simulink that has to be used for constructing blocks, in order to send messages from a given serial port of Raspberry Pi ("/dev/ttyACM0" serial write port in this case). The appropriate bit rate or baud rate (in [bits/s]) should be set on the other side (STM32 Nucleo-64) as well for a reliable communication between the two boards.

Table 2
Serial String Message Structure

Command	Message Structure	Variables	Explanation
Move()	#MCTL:%.2f;%.2f;;\r\n”	Steering angle [°] Velocity [PWM]	Responsible for simple motion of vehicle
Brake()	#BRAK:%2f;;\r\n”	Steering angle [°]	Braking command Adjust steering angle
Spline()	#SPLN:%d;%.2f;;\r\n	Logical value for direction of motion Coefficient terms of quadratic polynomial Duration time of motion	Movement of vehicle on polynomial trajectory
PID_active()	#PIDA:%d;;\r\n”	Logical value for activating PID controller Velocity values in [m/s]	Command for activating PID controller
PID_setup()	#PIDs:%.5f;%.5f;;\r\n	Coefficient terms of PID controller (k_p, ki, kd, t_f)	PID parameter setup

Regarding the lane keeping function controller, a relatively simple, but performant solution was adopted. Thus, we created PID controller only with PD terms for the distance error. After that, we considered the heading error as well with another PD controller, resulting in a more stable movement of the vehicle in corners.

The other important component of the model's control system is the global path planning and trajectory execution, a Python code that calculates a proper path for the vehicle model, based on start and desired end positions.

The given map is in graph format containing nodes with coordinates and edges that declare the connections between the nodes (see Fig. 9). Each node has a pink ID and blue edges, directed arrows. As a result, the given graph is a directed graph, meaning that there is only one option to move from a node to another connected one.

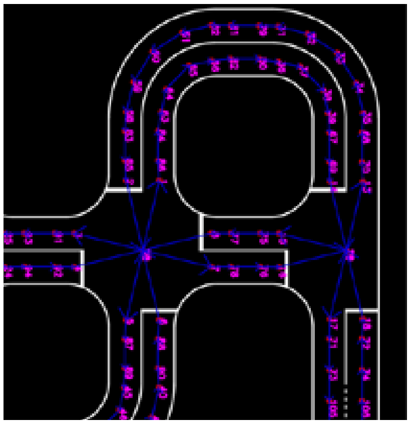

Fig. 9.
Digital Map with the Visualized Graph
Source: (Own work, 2021)

Because the IDs of the nodes with coordinates and the connection between the nodes are declared, in the code we need to make a database for the nodes and the edges, to be able to calculate a suitable path between desired positions.

The two main variable of the preprocessing part is the graph itself on which the calculation is performed (path planning) and the direction matrix, which is for the specification of the turning possibilities at cross-sections (*-1* for right turn, *1* for left turn and *0* for the straight motion).

The used path planning algorithm is Dijkstra's optimal shortest path algorithm. The output of this method is the IDs of the nodes in order of the path.

To get useful data some post-processing is needed. The coordinates of the nodes are assigned to the node IDs, the output matrix also containing the turning directions at cross-sections. At the end of the post-processing phase is the calling of the function and the inputs, moreover, it is also possible to give intermediate node IDs for the algorithm, thus different paths can be planned than the shortest.

The solution/output of the given input, the coordinates assigned to the node IDs according to the planned path, can be seen projected on the map in graph format. If the vehicle is before a cross-section, the control system will be able to know the steering commands based on the output.

3.4. Robot Operating System

The ROS, or Robot Operating System, is a flexible, open-source framework designed for writing robot software. ROS makes it easier to write transparent software, while providing libraries and tools to help developers to create robot applications. It provides, among others, device drivers, libraries, visualizers, message-passing, as well as package management (GitHub, 2017).

The basic components of the system are the nodes. A node is a process that performs computation, it can be written in C++ or Python as well. Usually, a robot is going to have different nodes for every task, so it can be developed parallelly and it will result structured software architecture (Bosch Future Mobility, 2019).

The nodes can communicate with each other via topics. A topic is a named bus with a specified message type. Every node can publish messages to a topic, and subscribers can read them. Interfaces can be defined by using topics, thus replacing modules during testing phases.

The existing algorithms were moved to the ROS framework. The image processing node needed just some simple modifications: declaring the publisher/subscriber topics, refactoring the flow of the program. Since the controller is made in Simulink, the first step was to generate a C++ code from the model. A ROS wrapper was made for the generated code, so the model behind the ROS node can be changed without any inconvenience.

This way the ROS graph illustrated below by Fig. 10 could be obtained; all the scripts can communicate with each other and provide I/O states, thus creating a complete global communication system.

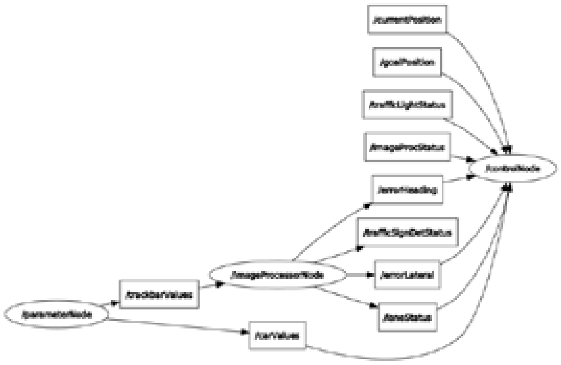

Fig. 10.
The Implemented ROS Architecture
Source: (Own work, 2021)

There are two main nodes at the moment, one is the image processing node and the other is the controller node. They communicate via the */errorLateral* topic and */laneStatus* topic, which is the error signal for the lane keeping system. The Int32 message type is used from the std_msgs library.

There is one more node, which is used for tuning the parameters of the controller and image processing. This node communicates with the dedicated debug topics and uses Float32MultiArray or Int32MultiArray.

Table 3
Parameter Value Settings

Parameter	Value, [-]
Proportional gain	49
Differential gain	14
Canny lower threshold	47
Canny upper threshold	151

The parameter node contains some trackbars; Table 3 contains the parameter settings. After a trackbar callback, the changed values are published to the different topics and the controller and image processing node change the parameters accordingly. The trackbar names, upper-lower limits, default values can be changed. The values can be saved into a configuration file.

Built-in features for debugging are also used for monitoring the error/output signals of the controller or position tracking.

4. Results and Discussion

The complexity of nowadays' systems and the stochasticity of the potential traffic situations demand new approaches with different testing levels and approval layers. As a result of new components and increased in-vehicle system complexity, vehicle testing and validation became different as earlier. Testing the vehicle, the driver-controller and the traffic situations together require new testing methods and strategies, wherewith development time and costs could be reduced drastically, while obtaining more testing data. The aim is the same as earlier, to guarantee road safety with reliable operation of the systems (Németh *et al.,* 2019).

Throughout the present study two defining automated functions, lane detection and tracking, respectively traffic-sign detection were developed, tested and verified through driving cycles implemented on vehicle model platforms. The obtained results of the applied methodologies are summarized below as well:

- *1:10 vehicle model platform*: RPi4 - image processing, positioning, control, STM32 Nucleo - DC motor and power steering control, RPi v2 8MP Sony camera;
- *Virtual simulation*: PreScan virtual environment, MATLAB polynomial curves, Simulink global state machine;
- *Control system*: Simulink PID, Dijkstra route and trajectory planning, ROS;
- *Lane detection and tracking*: Canny edge search, histogram analysis (Python, OpenCV);
- *Traffic-sign recognition*: color threshold, contour search, classification + template matching, ROI (Python, OpenCV);
- *Testing*, *validation* and *verification* through autonomous driving cycles.

In order to conclude the study, the authors hope that also new testing and validation methodologies can be based on the presented ideas and results, that further automated functions or even complex autonomous systems could be verified and validated with the help of such embedded systems.

Acknowledgements

Within the framework of the New Széchenyi Plan, the project "Development of talent management and researcher supply in the field of autonomous vehicle control technologies (EFOP-3.6.3-VEKOP-16-2017-00001)" provided funding for the study. The research was supported by the European Union and co-financed by the European Social Fund.

References

Aradi, Sz.; Bécsi, T.; Gáspár, P. 2014. Experimental Vehicle Development for Testing Autonomous Vehicle Functions, In *2014 IEEE/ASME 10th International Conference on Mechatronic and Embedded Systems and Applications (MESA)*, Senigallia, Ancona, Italy, Sep. 10-12, 2014, pp. 1-5. https://doi.org/10.1109/MESA.2014.6935534.

BME. 2021. BME Automated Drive Lab Research & Development - Creating cutting-edge solutions for tomorrow's challenges. Available from Internet: <https://www.automateddrive.bme.hu/research-development>.

Bosch Future Mobility. 2019. The Challenge Competition Regulations. Available from Internet: <https://www.boschfuturemobility.com/wpcontent/uploads/2019/01/BoschFutureMobilityChallenge_2019_Regulations.pdf>.

Chuan-En, L. D. 2018. Build a lane detector. Available from Internet: <https://towardsdatascience.com/tutorial-build-a-lane-detector-679fd8953132>.

Conrad. 2021. Conrad Electronic Education & Entwicklungskits. Available from Internet: <https://www.conrad.de>.

Ferencz, Cs.; Zöldy, M. 2020. Simulation and validation with radio-controlled (RC) autonomous vehicles in roundabout situation [In Hungarian: Körforgalmi szituáció szimulációja és validálása rádióvezérlésű (RC) autonóm járművek segítségével]. In *28th International Conference on Mechanical Engineering*, April 25, 2020, online, pp. 206-210. ISSN 2668-9685. Available from Internet: https://ojs.emt.ro/index.php/oget2020/article/view/159.

García, L. et al. 2019. Autonomous Driving in Roundabout Maneuvers Using Reinforcement Learning with Q-Learning, *Electronics* 8(12): 1536. https://doi.org/10.3390/electronics8121536.

Garv, T. 2020. Self-Driving Car: Advanced Lane Line Detection Udacity. Available from Internet: <https://medium.com/@garvtambi05/self-driving-car-advanced-lane-line-detection-udacity-p2-3745b3cc43e9>.

GitHub. 2017. The world's leading software development platform - GitHub. 2017. Lane Detection Using Hough Transform. Available from Internet: <https://github.com/karasuno7/Lane-Detection-using- Hough-Transform>.

Németh, H. et al. 2019. Proving Ground Test Scenarios in Mixed Virtual and Real Environment for Highly Automated Driving, In *Proff H. Mobilität in Zeiten der Veränderung - Springer Gabler*, Wiesbaden, Germany, pp. 198-210. https://doi.org/10.1007/978-3-658-26107-8.

Ramesh, J.; Rangachar, K.; Brian, G. S. 1995. *Machine Vision*, McGraw-Hill, Inc., USA. 549 p. ISBN 0-07-032018-7.

Raspberry. 2021. Teach, Learn and Make with Raspberry Pi. Raspberry Pi Products. Available from Internet: <https://www.raspberrypi.org/>.

Szalay, Zs. et al. 2017. Technical Specification Methodology for an Automotive Proving Ground Dedicated to Connected and Automated Vehicles, *Periodica Polytechnica Transportation Engineering*, 45(3): 168-174. https://doi.org/10.3311/PPtr.10708.

Tass International. 2021. Tass International PreScan, (8.4.0). Available from Internet: <https://tass.plm.automation.siemens.com/prescan>.

Tollner, D.; Cao, H.; Zöldy, M. 2019. Artificial Intelligence Based Decision Making of Autonomous Vehicles Before Entering Roundabout. In *2019 IEEE 19th International Symposium on Computational Intelligence and Informatics and 7th IEEE International Conference on Recent Achievements in Mechatronics, Automation, Computer Sciences and Robotics (CINTI-MACRo)*, Budapest, Hungary, Nov. 14-16, 2019, pp. 181-186, https://doi.org/10.1109/CINTI-MACRo49179.2019.9105322.

Wong, C. 2017. Advanced Lane Finding Using Sliding Window Search. Available from Internet: <https://github.com/charleswongzx/Advanced-Lane-Lines>.