ROCS: A Realtime Optimization and Control Simulator

Andreas Britzelmeier[1], Matthias Gerdts[1], Omid Moslehi Rad[1], Sonali Rani[1], Thomas Rottmann[1]

[1]Institute of Applied Mathematics and Scientific Computing, Universität der Bundeswehr München, Werner-Heisenberg-Weg 39, 85577 Neubiberg, Germany;
{*andreas.britzelmeier,matthias.gerdts,omid.moslehi,sonali.rani,thomas.rottmann*}*@unibw.de*

Abstract. The Realtime Optimization and Control Simulator (ROCS) is a software package written with Qt. It is conceived to be a versatile tool to develop, investigate, and visualize control and trajectory optimization tasks for automated vehicles, aircrafts, and robots in multi-modal scenarios. It is also conceived as a platform which allows to combine real driving data with virtual simulation using a vehicle in the loop.

1 Introduction

The task of simulating and modeling physical systems has always been an important instrument of scientific and industrial research and development. It allows to study and to evaluate the behavior of proposed models and to compare the outcome with the performance of the actual system under consideration. Therein, we can distinguish two major types of simulation tasks, that is, with and without real time requirements and visualization. Often computations are undertaken and afterwards visualized through graphs or non-immersive types of representation. However, with advancements in hardware technology and the increase in digitization in many systems, like cars, planes, and mobile robots, the requirement for virtual environmental simulation is drastically increasing. Another aspect which justifies immersive simulation tools comes along with automation of systems and the desire to create a digital twin. Developers are obliged to prove the reliability and safety of newly developed systems, as well as that the expected increase in utility is guaranteed. However, building prototypes often is very expensive as well as intensive testing to generate reproducible results. Therefore, the need of alternative methods to analyze the behavior and interaction of systems is imminent. Prominent examples of powerful simulation tools in the automotive industry are Virtual Testdrive (VTD) [17, 15], and SILAB, [11]. Both are widely used in the automobile industry, compare [1]. Apart from the automobile industry, also in aerospace engineering and flight training simulative tools have a long history and are widely used. The Flight Simulator from Microsoft, see [12], as well as X-Plane, see [18], both of which are certified by the Federal Aviation Administration, are used in pilot training and research. Hence, immersive simulation tools are not just mere tools of real-time visualization but necessary tools to develop the technology of the future. An indispensable advantage of such tools comes to play whenever new technologies which require human cooperation or interaction is necessary. Then these simulative tools provide a safe environment to test acceptance and reliability, compare [8, 13]. Despite the different types of simulation tools already available, most of them are limited to a specific use case, may it be cars or planes.

In addition, there are very popular game engines, e.g, Unity [16], which are applicable to both, gaming and simulation applications. Such game engines have numerous advantages, e.g. fast and agile development, huge asset stores, optimized graphics, physics and audio engines. However, these platforms come with some specific limitations that can hinder scientific simulation purposes. Few of those complications are licensing and costs for activation of desired features, difficulty in organizing its complex directory hierarchy, non-public source code, making it difficult to track or debug issues, increase in consumption of hardware resources due to complex environment, and finally it is convenient to use only with C Sharp as the primary scripting language. Moreover, downward compatibility issues owing to new versions may arise and can make it difficult to maintain a long term project. Finally, the addition of one's own particular models, controllers, or optimizers can be cumbersome or even impossible.

These potential drawbacks motivated us to build a

ARGESIM Report 59 (ISBN 978-3-901608-93-3), p 193-200, DOI: 10.11128/arep.59.a59026

research and development tool called Real-time Optimization and Control Simulator (ROCS). The idea was to build a versatile research tool which allows for in time visualization and testing of our online optimization algorithms and feedback controllers for automated agents in multi-model scenarios. A further goal was to include control interfaces to real systems. ROCS is build as a modular tool which allows for simple extension by further optimizers and controllers, and the integration of sensor data. Further it is able to visualize scenarios and conduct experimental validation with a variety of vehicles like cars, industrial or mobile robots, and flying platforms like drones, planes or quadrocopters. Owing to the modularity of the tool it is comparatively simple to add models for every required type of vehicle. Different modes of simulation are implemented. One can either provide precomputed data, use feedback controllers in combination with model simulation or employ an optimizer to perform online path planning tasks. ROCS already provides a set of vehicular controllers as well as different models for cars and integrators.

The outline of the paper is as follows. In Section 2 we discuss the overall conceptual design of ROCS. Simulation aspects and two selected vehicle models in ROCS are discussed in Section 3. Section 4 addresses the controller design, while Section 5 presents the 3D simulation environment. Some simulation results are presented in Section 6. Finally, a summary and an outlook with future developments conclude the paper.

2 Design

Realtime Optimization and Control Simulator (ROCS) is designed in a modular way. We decided to implement it in C++ with Qt as it is a programming language widely used in industry and academia and facilitates integration of algorithms and modules. In addition it provides convenient 3D visualization capabilities and the slots and signal mechanism is very well suited for the realtime control purposes.

The main components of ROCS are depicted in Figure 1. The core class objects are a vehicle class, a control class, an input/output class, and a visualization class. The vehicle class contains all vehicle relevant parameters, numerical integrators for motion prediction and simulation, interfaces to controllers and graphical objects describing the shape. The control class contains a collection of tracking controllers and optimization-based path planning tools as detailed in Section 4. The visualizaton class serves to display the simulation and control outputs in a 3D view or in chart plots. It is also possible to store the simulation results or measurements in a file. Likewise it is possible to use ROCS in an offline mode in oder to visualize external data from a data file. The central control unit is the User Control Interface (UCI) described in Section 2.1. An automatic world generator class is part of the concept, but not fully realized up to now.

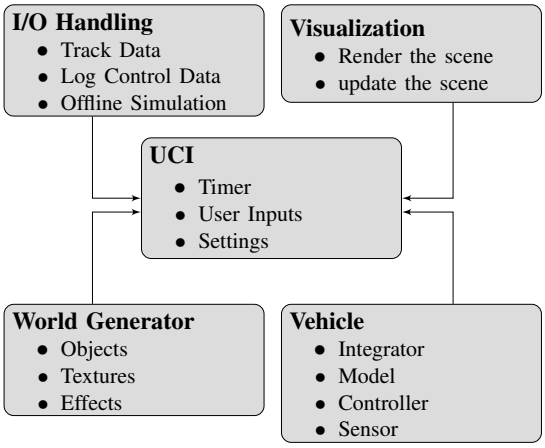

Figure 1: Information flow between simulation models and controllers.

2.1 The User Control Interface (UCI)

The central control panel of ROCS is the User Control Interface (UCI) in Figure 2. This panel allows to load reference paths, environments, and models. Moreover it provides an overview on the number and type of agents within the simulation. The properties of the agents can be edited through additional dialogs, compare Figures 4, 5. The UCI furthermore allows to select a camera perspective, to switch on or off a data logging mode, and it permits to adjust the scene timer for 3D vizualization. Finally it offers options for saving and loading in order to conveniently store or re-store complex scenarios and settings.

Figure 2: Central user control interface.

2.2 Handling of Multiple Agents

Due to the object oriented programming style the vehicle class can be sub-classed to differentiate among vehicle types. Furthermore, multiple objects of one vehicle can be created inheriting the same properties and functions, controlling their visualization. On top, each object is stored in a list such that each vehicle appearing in the scene can also be customized. Customization includes changing the objects model, linking to different controllers or integrators or changing the pipeline of data acquisition.

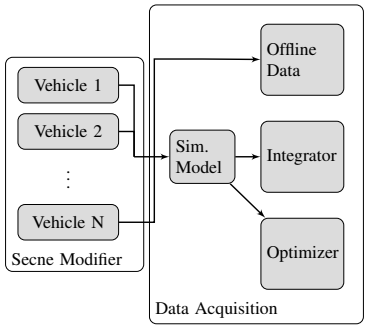

Figure 3: Customizable linking of different data structures for Simulation.

Data can be acquired three ways. The first is to load offline created data from text files which provide data required for simulation. The second method consists in online computation of simulative data through integration and feedback controllers. The computed data is then pipelined by a signal to a slot in the scene modifier class. Thereby, each individual vehicle object and the corresponding controller run in a separate thread and do no interfere with the update of the scene or other operations of the tool. Threads are managed in a synchrnous and threadsafe way. The last option includes the data generation by an optimization-based path planner, which repeatedly solves optimal control problems within a model-predictive control loop. The output data can also be directed to the scene modifier by addressing the same slot from the optimizer class. Hence, we have a uniform connection through signal and slots which can be used to adjoin further modules as well.

In summary, ROCS is centered around the feedback control loops for the agents. These control processes run at a specified frequency in their own threads and are decoupled from the visualization, which is able to run at its own frequency and merely accesses simulation data generated by the control loops. Both frequencies can be synchronized in which case visualization and control work in realtime, if the hardware permits it.

3 Simulation of Multi-Agent Systems

ROCS allows to investigate heterogeneous multi-agent systems consisting of, e.g. cars, robots, or aircrafts. These agents or vehicles, respectively, can be derived from a basic vehicle class, which inherits core functionalities for any type of agent. The derived objects allow to set particular features of the individual agents. The individual agents can be added to the simulation through a dialog window, compare Figure 4.

Figure 4: Dialog for adding agents.

The individual properties, models, and parameters of the agents can be adjusted and selected in an editing dialog, compare Figure 5.

Figure 5: Dialog for editing agents.

Once all agents have been configured (including dynamics, initial states, controller types) and added to the scenario, it remains to simulate the whole multi-agent system. To this end let $N \in \mathbb{N}$ agents be given. We assume that each agent can be controlled and for each agent $i \in \{1,\dots,N\}$ we denote the control input at time t by $u_i(t)$ and the state at time t by $x_i(t)$. The motion of the i-th agent is modelled mathematically by an initial value problem of type

$$x_i'(t) \quad = \quad f_i(t, x_i(t), u_i(t)), \quad x_i(t_{i,0}) \; = \; x_{i,0}, \quad (1)$$

with initial time $t_{i,0}$ for $i = 1,\ldots,N$. The agents can be controlled either in open-loop, i.e., by providing the control input $u_i = u_i(t)$ as a given function of time as in (1), or in closed-loop by providing a feedback law $u_i = \mu_i(t,x)$, where $x = (x_1,\ldots,x_N)^\top$ is the combined state of all agents. This leads to the closed-loop system

$$x_i'(t) \quad = \quad f(t,x_i(t),\mu_i(t,x(t))), \quad x_i(t_{i,0}) \; = \; x_{i,0}, \tag{2}$$

for $i = 1,\ldots,N$. In both cases the overall dynamic system will be solved numerically by a Runge-Kutta method. ROCS uses standard solvers with fixed step-sizes (Euler method, Heun's method, classic 4-th order Runge-Kutta method) and variable step-sizes (DOPRI5(4)), see [10].

The outcome of the simulation can be stored in a data logging file or in a chart window, compare Figure 6.

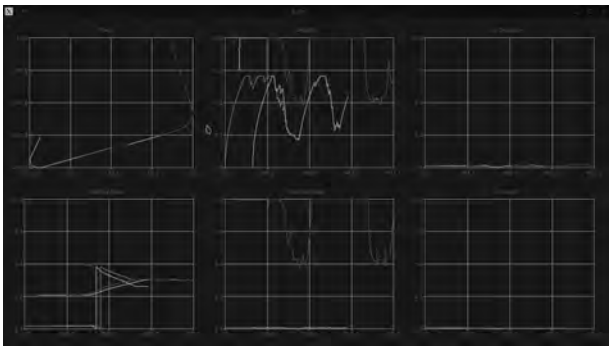

Figure 6: Chart window for detailed view of sensors, states, and controls.

The design of the feedback laws μ_i for the agents $i = 1,\ldots,N$, will be outlined in Section 4. Currently, we only have individual controllers implemented, which do not take into account the behavior of the other agents. In the future we will add controllers and optimization strategies for interacting systems as outlined in [4]. This will require to set up an agent-to-agent or agent-to-cloud communication procedure, in which, e.g., position data or driving intentions are exchanged.

3.1 Vehicle Models

At the current state of development, due to the focus on autonomous driving, two vehicle models, a single track model and a kinematic model of a two wheel driven mobile robot have been implemented. The equations of motion of the single track model read as,

$$x' = v\cos(\psi-\beta), \tag{3}$$

$$y' = v\sin(\psi-\beta), \tag{4}$$

$$v' = \frac{1}{m}\left[(F_{uh}-F_{Lx})\cos\beta + F_{uv}\cos(\delta+\beta) - (F_{sh}-F_{Ly})\sin\beta - F_{sv}\sin(\delta+\beta)\right], \tag{5}$$

$$\beta' = w_z - \frac{1}{mv}\left[(F_{uh}-F_{Lx})\sin\beta + F_{uv}\sin(\delta+\beta) - (F_{sh}-F_{Ly})\cos\beta - F_{sv}\cos(\delta+\beta)\right], \tag{6}$$

$$\psi' = w_z, \tag{7}$$

$$w_z' = \frac{1}{I_{zz}}\left[F_{sh}\ell_v\cos\delta - F_{sh}\ell_h - F_{Ly}e_{SP} + F_{uv}\ell_v\sin\delta\right], \tag{8}$$

$$\delta' = \frac{\delta_c-\delta}{T_c}. \tag{9}$$

Herein, x and y are the spacial coordinates and v denotes the velocity. The side slip angle is given by β, the yaw angle is ψ and the steering angle δ. The single track model is already a quite detailed model of a car, which is frequently used in the automotive industry for the investigation of the lateral motion of cars. The model includes various forces acting on the vehicle body. That is, the lateral tyre forces F_{sh}, F_{sv}, longitudinal forces F_{uv}, F_{uh} as well as air resistance in longitudinal F_{Lx} and lateral F_{Ly} direction. Further we have the vehicle mass m and the distance from the centre of gravity to the drag mount point e_{SP}. The distance from the centre of gravity to the front and rear wheel are described by ℓ_v and ℓ_h respectively. The control input to the model are the commanded steering angle δ_c and a combined acceleration and decelleration force, which enters the above force terms. Details of the model can be found in, e.g., [6, 7]. The constant $T_c > 0$ is used to model a delay in the adjustment of the steering angle towards the commanded steering angle.

Another model in ROCS describes a mobile robot with two driven wheels on the left and the right, respectively. Its equations of motion read as follows:

$$x' = \frac{v_L+v_R}{2}\cos\psi, \tag{10}$$

$$y' = \frac{v_L+v_R}{2}\sin\psi, \tag{11}$$

$$\psi' = \frac{v_R-v_L}{B}, \tag{12}$$

$$v_L' = \frac{v_L^c-v_L}{T_c}, \tag{13}$$

$$v_R' = \frac{v_R^c-v_R}{T_c}. \tag{14}$$

Herein, x and y denote the center of gravity of the robot, ψ the yaw angle, v_R and v_L the velocity of the right and

left wheels, respectively, and B is the width of the robot. The robot is controlled by the commanded velocities v_R^c and v_L^c of the right and left wheels. The constant $T_c > 0$ is used to model a delay in the adjustment of the velocities towards the commanded velocities.

These two models are included in order to illustrate that heterogeneous agents can be considered. Further vehicle models and models for mobile robots can be found in [5]. We like to point out that further models can be integrated into ROCS in a straightforward way. This is an important feature for our research purposes.

4 Control and Path Planning

The realtime feature is implemented through timers for the control loop of each vehicle, i.e., the control loop runs at a user-defined rate and triggers the import of sensor data and the update of controls. The computed controls are then applied to the vehicle, either for simulation purposes or to control a real vehicle. As for the control we distinguish between path tracking control and path planning control. The former aims to track a predefined (spline) path while the latter generates a path and a trajectory using mathematical vehicle models and online optimization methods in combination with model-predictive control, see, e.g., [7]. In both, path tracking and path planning, the aim is to realize the feedback law μ_i in (2). Currently, a dynamic inversion controller and a linear model-predictive controller are used for path tracking, see [5, 3] for details. These controllers can be applied to both models in Section 3.1. The model-predictive control concept is applicable to path planning as well, compare [7]. Since model-predictive control is a powerful and versatile control paradigm, especially for multi-agent systems, we outline in brief the working principle. Further details can be found in the monographs [14, 9].

To this end we consider dynamics in discrete time $t_n = t_0 + nh$, $n \in \mathbb{N}$, where $h > 0$ is the stepsize given by the control timer in ROCS. For notational convenience we restrict the discussion to $N = 1$ agent with state x, control u, and dynamics (1). Discretization of the latter using a suitable Runge-Kutta method leads to a discrete time system. A typical path tracking task requires to solve a linear-quadratic optimization problem of the following type at each t_n with measured state x_n at t_n:
Minimize the tracking error

$$\frac{1}{2}\sum_{k=n}^{n+M-1} \|x(t_k) - x_{ref}(t_k)\|^2 + \|u(t_k) - u_{ref}(t_k)\|^2$$

subject to the constraints

$$\begin{aligned} x(t_{k+1}) &= A_k x(t_k) + B_k u(t_k) && (k = n, \dots, n+M-1) \\ x(t_k) &\in X && (k = n, \dots, n+M) \\ u(t_k) &\in U && (k = n, \dots, n+M-1) \\ x(t_n) &= x_n \end{aligned}$$

Herein, the linear dynamics are obtained by linearization at the reference path (x_{ref}, u_{ref}). The number $M \in \mathbb{N}$ denotes the preview horizon, which has to be chosen appropriately. The sets X and U define state and control constraints. Likewise a typical path planning task consists in solving a nonlinear optimization problem of the following type at t_n with measured state x_n at t_n:
Minimize the objective

$$\varphi(x(t_{n+M})) + \sum_{k=n}^{n+M-1} \ell(x(t_k), u(t_k))$$

subject to the constraints

$$\begin{aligned} x(t_{k+1}) &= F(x(t_k), u(t_k)) && (k = n, \dots, n+M-1) \\ x(t_k) &\in X && (k = n, \dots, n+M) \\ u(t_k) &\in U && (k = n, \dots, n+M-1) \\ x(t_n) &= x_n \end{aligned}$$

Herein, φ and ℓ are suitable functions modelling the control objective, e.g. driving fastly or economically. Now, the standard model-predictive control (MPC) concept requires to solve one of the above optimization problems repeatedly on a shifted time horizon. Figure 7 shows the outcome of a path planning task using the single track model in Section 3.1 for a track on the campus of the Universität der Bundeswehr München. Obstacles can be avoided as well, see [3].

4.1 Sample Vehicle Controller

We outline path tracking controllers for the vehicle models in Section 3.1. For the implementation of the controllers we use slightly modified models in terms of a curvilinear coordinate system:

$$s'(t) = \frac{v(t)\cos(\psi(t) - \psi_m(t))}{1 - r(t)\kappa_m(s(t))}, \tag{15}$$

$$r'(t) = v(t)\sin(\psi(t) - \psi_m(t)), \tag{16}$$

$$\psi'(t) = v(t)\kappa(t), \tag{17}$$

$$\kappa'(t) = u(t), \tag{18}$$

$$\psi_m'(t) = v(t)\kappa_m(s(t)). \tag{19}$$

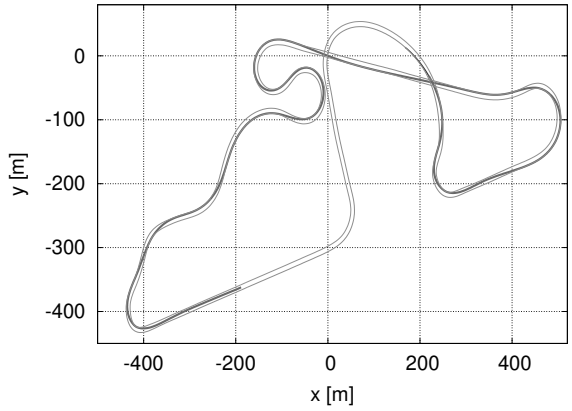

Figure 7: Nonlinear MPC result of a path planning task.

where s is the arc length along a given reference spline curve and r is the lateral offset from the reference spline. The actual heading is given by $\psi(t)$ and the corresponding reference heading is given by $\psi_m(t)$. The curvature of the driven path is denoted by κ and the curvature of the reference path is κ_m.

Both controllers are based on a simple kinematic model, Eqs. (15) to (19) and are designed to control the curvature deviation to track a given reference path. Herein, the controller class provides a control input to the vehicle models through a signal and slot connection. This allows for an easy extension with additional controllers, since the user only needs to provide an output signal. Then, the output is transformed for the respective model and eventually can be integrated employing one of the integrators, provided by the integration class, see Fig. 8.

The aforementioned transformations are model dependent. The single track model could be controlled through the steering angle δ, the commanded steering angle δ_c or the steering angle rate $\delta' = \omega_\delta$ respectively. Hence, we require a relation between the output of the controller, i.e., κ, and the control variables. For the commanded steering angle and the steering angle velocity, respectively, these relations are given by

$$\delta_c = \arctan(\ell\kappa), \omega_\delta = \ell\kappa' \cdot \cos^2(\delta). \qquad (20)$$

Herein, $\delta' = \omega_\delta = \frac{\delta_c - \delta}{T_c}$ with constant $T_c > 0$. The two wheeled robot is steered through the velocities of the left and right wheel. Exploiting physical relations yields,

$$v_L^c = v_d - \frac{1}{2}B \cdot v \cdot \kappa, \quad v_R^c = v_d + \frac{1}{2}B \cdot v \cdot \kappa, \qquad (21)$$

with B the width of the robot, v_d the desired longitudinal velocity, and $v = (v_L + v_R)/2$ the current velocity. Both controllers are discussed in detail in [4] and [3].

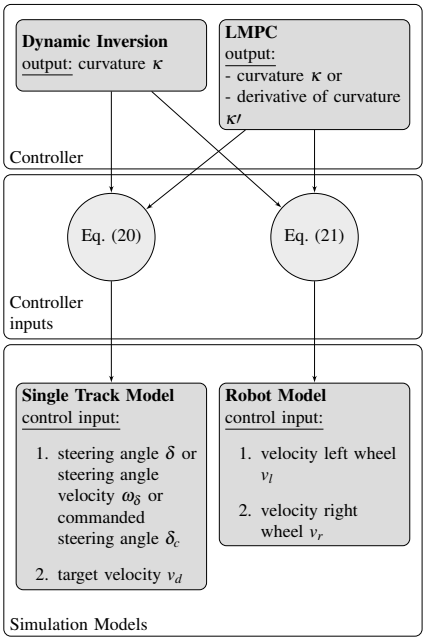

Figure 8: Information flow between simulation models and controllers.

5 Visualization

For the visualization of the control and simulation results we utilize the Qt Framework, which provides an OpenGL high level interface and allows for performant rendering in C++ applications. To visualize certain objects the data structure is based on a scene graph defined by a system of entities, where the scene graph is a tree structure made of these entities and other components. The entities to be rendered can be assigned through object files containing 3D models of, e.g., a car, an air plane, a robot, or buildings. Therefore we implemented an overloaded class of QSceneLoader addressing our requirements and managing the entities, as well as interfacing the rendering canvas. Herein, the rendering is solely a data driven process. Prebuild camera entities are provided by Qt providing viewpoints through which the scene is rendered. Multiple cameras are implemented in ROCS to capture different perspectives, e.g., the ego person's view in 9, the third person view, see Figure 10, where the camera follows in a fixed distance behind the object and a birds view in Figure 11.

Figure 9: Ego person's view of a scene.

Figure 10: Third person's view of a scene.

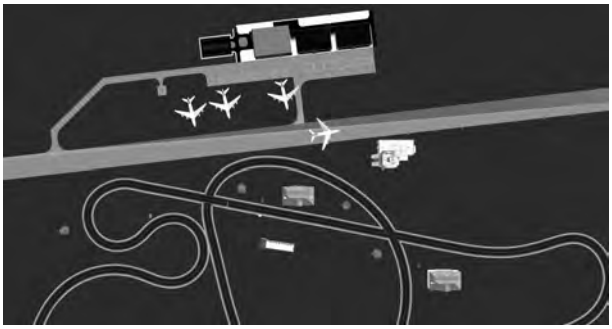

Figure 11: Bird's view of a scene.

6 Evaluation and Results

Figure 12 shows selected car data stored by the data logger function of ROCS. The virtual RAM used by ROCS for this simulation amounts to 2.37 GB and the RAM to 3.67 GB for a total of 12 simultaneously controlled cars. The GPU load amounts to 35.2 %. The computations were performed on a system with 16 GB of memory, Intel i7-8700 processor with 3.2 GHz (6 cores, 12 threads) and a Nvidia Geforce FTX 1060GB (6 GB RAM) graphic card.

The results show the output of the single track model with a linear model-predictive path tracking controller for the track depicted in Figure 7. This controller is able to track a given geometric reference path even for comparatively high velocities with a maximum deviation of 0.15 m (see r in Figure 12).

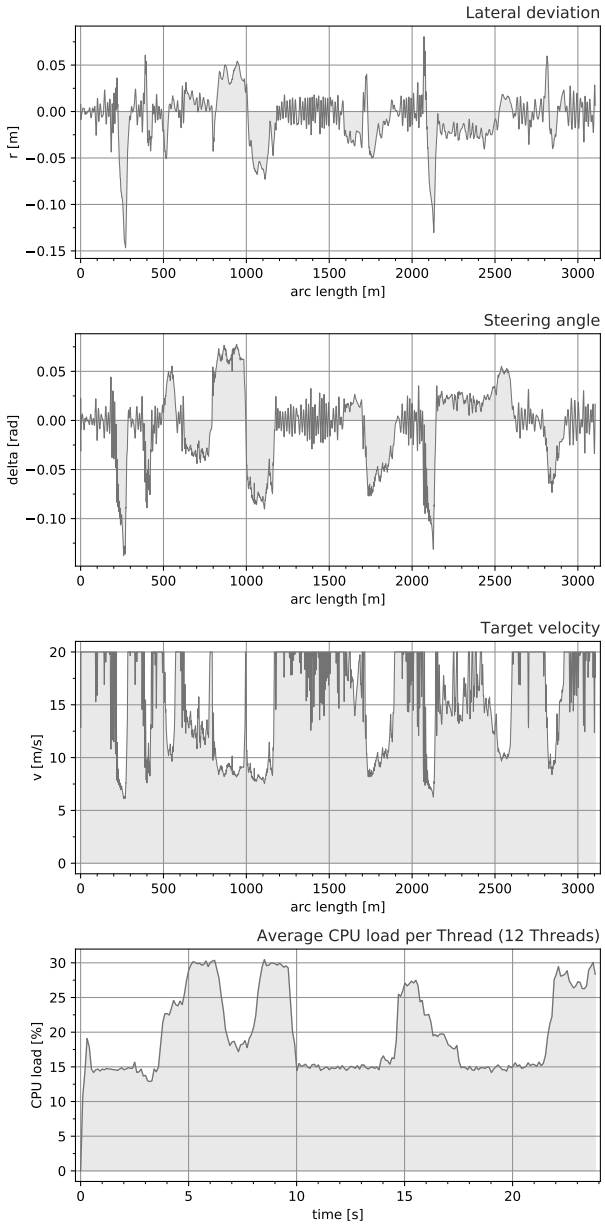

Figure 12: Data logger output (from top to bottom): lateral deviation r, steering angle δ, target velocity, and average CPU load per thread (12 threads, 12 vehicles).

7 Current Developments and Future Extensions

The development of ROCS is on-going and vehicle models from different disciplines (mobile robots, flight systems, space systems) of different complexity with appropriate controllers and path planning tools will be added step-by-step. The interfaces of ROCS will allow to directly import real sensor measurements of vehicles and to generate data to control a vehicle. This option allows to run simulation and real vehicle motion in parallel in order to overlap the two motions with the aim to design accurate digital twins. At the same time it allows to simulate a virtual world for a research vehicle at the Universität der Bundeswehr called Vehicle-in-the-loop [1, 2]. This research platform is based on a real car (Audi A6 Avant) and uses virtual environments to couple real driving experience and virtual scenarios. This concept is ideal for testing potentially dangerous scenarios in a safe way and we aim to integrate ROCS into the vehicle in the loop (VIL) for visualization, but also as an automatic control tool.

Acknowledgement

Copyright and ownership of ROCS and its derivatives solely resides with its founders Andreas Britzelmeier and Matthias Gerdts.

References

[1] Berg, G., Karl, I., Färber, B. Vehicle in the loop - validierung der virtuellen welt. In Nichtred. Ms.-dr., editor, *Der Fahrer im 21. Jahrhundert: Fahrer, Fahrerunterstützung und Bedienbarkeit*, volume 6. Verein Deutscher Ingenieure, VDI-Verl., November 2011.

[2] Berg, G., Nitsch, V., Färber, B. *Vehicle in the loop*. In: Winner H., Hakuli S., Lotz F., Singer C. (eds), Handbook of Driver Assistance Systems, Springer, 2015; pp. 199–210.

[3] Britzelmeier, A., Gerdts, M. *A Nonsmooth Newton Method for Linear Model-Predictive Control in Tracking Tasks for a Mobile Robot With Obstacle Avoidance*. in IEEE Control Systems Letters, 2020; Vol. 4(4), pp. 886–891, doi: 10.1109/LCSYS.2020.2996959.

[4] Britzelmeier, A., Gerdts, M., Rottmann, T. *Control of interacting vehicles using model-predictive control, generalized Nash equilibrium problems, and dynamic inversion*. 2020 IFAC World Congress, 2020.

[5] Burger, M., Gerdts, M. *DAE aspects in vehicle dynamics and mobile robotics*. Applications of differential-algebraic equations: examples and benchmarks, Differential-Algebraic Equations Forum, Springer, 2019; pp. 37–80.

[6] Gerdts, M. *Solving mixed-integer optimal control problems by branch&bound: a case study from automobile test-driving with gear shift*, Optimal Control Applications and Methods, Vol. 26, pp. 1–18, 2005.

[7] Gerdts, M., Karrenberg, S., Müller-Beßler, B., Stock, G. *Generating locally optimal trajectories for an automatically driven car*. Optimization and Engineering, 2009; Vol. 10, pp. 439–461.

[8] Graichen, M., Graichen, L., Rottmann, T., Nitsch, V. Using the projection-based vehicle in the loop for the investigation of in-vehicle information systems: First insights. In *Proceedings of the 4th International Conference on Vehicle Technology and Intelligent Transport Systems - Volume 1: VEHITS,*, pages 231–237. INSTICC, SciTePress, 2018.

[9] L. Grüne, J. Pannek. *Nonlinear Model Predictive Control – Theory and Algorithms*. 2nd Edition, Springer, 2017

[10] Hairer, E., Norsett, S. P., Wanner, G. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer Series in Computational Mathematics, 2nd Ed., Vol. 8, Berlin-Heidelberg-New York, 1993.

[11] Krueger, H.P., Grein, M., Kaussner, A., Mark, C. SILAB – A Task Oriented Driving Simulation. *North America*, page 9, 2005.

[12] Microsoft Corporation. Microsoft flight simulator. www.flightsimulator.com, 04 2020.

[13] Nitsch, V., Färber, B., Rüger, F. Automatic evasion seen from the opposing traffic - an investigation with the vehicle in the loop. In *IEEE 18th International Conference on Intelligent Transportation Systems*, 2015.

[14] J. B. Rawlings, D. Q. Mayne, M. Diehl. *Model Predictive Control: Theory, Computation, and Design*. 2nd Edition, Nob Hill Publishing, Madison, 2018.

[15] Roth, E., Dirndorfer, T., Knoll, A., von Neumann-Cosel, K., Ganslmeier, T., Kern, A., Fischer, M.-O.. *Analysis and validation of perception sensor models in an integrated vehicle and environment simulation.* Proceedings of the 22nd Enhanced Safety of Vehicles Conference, 2011.

[16] Unity 3D. Unity website. unity.com, 04/2020.

[17] VIRES Simulationstechnologie GmbH. Virtual test drive. vires.com/vtd-vires-virtual-test-drive, 04/2020.

[18] Laminar Research. X-plane 11. www.x-plane.com, 04/2020.