Wrocław University of Science and Technology
Faculty of Electronics, Photonics and Microsystems

Field of study: **Electronic & Computer Engineering**

ENGINEERING THESIS

Path optimisation of an autonomous Formula Student car in the BeamNG environment

Kamil Śmigielski

Supervisor
dr inż. Adam Ratajczak

WROCŁAW 2024

ABSTRACT

The goal of this documented work, has been an augmentation of the autonomous driving system, developed as part of collaboration with Racing Team, science circle of Wrocław University of Science & Technology. Especially in regards to operation of the pathing algorithms and control systems optimisations. Initial target of this augmentation, have been designed with a goal of facilitating testing for the AMU, over near and far future. This is to be achieved through study & employment of advanced, simulated environments & scenarios.
To reach this goal, a multitude of solutions for creation of input data for the robotic algorithms have been implemented & tried. With real world testing staying expensive, even after two models of Formula Student cars with autonomous operation capabilities have been finished, testing of actively developed algorithms in more and more sophisticated simulations, has become more relevant than ever.
After years of working on the Autonomous Driving project, BeamNG.tech has emerged as a major contender in the field of real driving scenarios simulation. After months of casual experimentation with the suit, it has been chosen as the final, complete solution in provision of testing scenarios & variables for the robotic system. Hence, described in this document, its implementation and first steps into exploiting functionality provided, in perfecting pathing algorithms for a 4-wheeled performance vehicle.

Keywords: Autonomous vehicle, pathing, path planning, computer vision, computer science, numerical analysis, control systems, object detection, simulation, synthetic data, ground truth data
CERCS P170/P176: Computer science, numerical analysis, systems, control/ Artificial Intelligence

CONTENTS

Preposition 3

Authors motivation 3

Goal & Overview 3

Assumptions & disclaimers 4

1. Engineering an autonomous driving system 6

1.1. The building blocks of a robotic system 6

1.1.1. Contemplation of the problem 6

1.1.2. Definition of robotic trinity 8

1.1.3. Considerations in project management 9

1.2. Processes Relevant to Formula Student competition 10

1.2.1. What is the Formula Student Car? 10

1.2.2. Important Rules & Restrictions in Roboticizing the Formula Student Car 10

1.3. Compilation of acquired knowledge into a functional autonomous driving system 12

1.3.1. Structure of an autonomously operating car 12

1.3.2. Overview of software design for autonomous driving 14

1.3.3. History of relevant solutions in the autonomous system 14

2. Simulation: a representation of reality 16

2.1. Benefits in starting with state-of-the-art physics & vehicle simulation 16

2.2. Creation of plug-and-play interfaces between simulation & physical systems is important 17

2.3. Universal interfaces & modular design improve testing workflow significantly . . 18

3. Engineering software interfaces, universally compliant with BeamNG simulation & real components 19

3.1. Designing the robotics system around concept of modularity 19

3.2. Creation of testing scenarios in BeamNG.tech 20

3.2.1. The platform 20

3.2.2. The environments 22

3.2.3. The Vehicle 23

3.3. Creation of 'virtual' interfaces in BeamNG.tech. Polling sensors & requesting controls. 26

3.3.1. Architecture 27

3.3.2. Programming 28

3.3.3. Testing 32

4. Extending path following algorithm with speed control 33

4.1. Distilling track driving rules of Formula Student competition 33

4.2. Understanding the path planning algorithm 33

4.3. Speed and acceleration optimisation in Trackdrive lap 34

4.4. Implementation 36

5. Summary 39

5.1. Validation 39

5.2. Closing words 41

Bibliography 43

List of Figures 44

Acronyms 45

Glossary 46

PREPOSITION

AUTHORS MOTIVATION

The concept of autonomous, public road transportation, once a mere dream of a handful of engineers experimenting with magnetic tracks in the 1940s, has become a tangible reality for the masses. We live in a world where ’robo-taxis’ operate, in a limited capacity, on actual city streets. And robotic cleaners are a common sight in shopping centers. Although these devices are primarily in the testing phase and their numbers are relatively small, they actively operate alongside humans and human drivers, with statistically minimal known instances of catastrophic failures or serious injuries.

Regrettably, pace of this revolution has been slow. Despite centuries with availability of technology for arbitrary transportation between two points on the globe, the roboticized solutions that have made their way into the mainstream are very limited in their scope of operation. They necessitate constant human supervision, and crucially, the pathing algorithms they employ are still far from optimal, rarely maximizing vehicle’s capabilities, increasing efficiency & improving safety by adapting to road conditions.

The most prevalent pathing and control algorithms in current times, such as those found in assist systems implemented in modern consumer cars, are often reported as operating in unpleasant or even in manners dangerous to life. It does not seem to be difficult, to guarantee stable operation of these systems, in testing & laboratory scenarios. But, despite millions of miles driven and thousands of hours of machine learning, these systems are still caught in edge cases and do not enjoy the flash of being part of any major competitions. Human drivers are still more valued on public roads, with human experience & flexibility, still king on race tracks.

I believe, that computerized control systems, being virtually free of distractions, can be enhanced way beyond human capabilities and made to operate perfectly in more scenarios than a human has the chance to experience in his lifetime. It is also plausible, as anything, through small steps. Starting with improvements of driving algorithms used in Formula Student competitions.

GOAL & OVERVIEW

This thesis and accompanying engineering project aim to familiarize the author and readers with fundamental structure of a modern autonomous driving system and the steps required for its implementation. The primary focus of this text is, to delve into details of software design of a medium-size autonomous driving system, with a particular emphasis on study of pathing algorithms and optimisation of control systems.

As arrival at the target, required many prior steps, which are no less important in function of the final product. They deserve similar recognition to the topic coming from the title & ultimate goal of this paper. To understand the whole package, which sustains optimal operation of the pathing & control systems, we are going to tour through the whole engineering process. From collective, theoretical overview, through identification of best practices, to individual parts and implementation of the movement algorithm.

In a short term, this work is going to increase performance of a Formula Student vehicle. In a long term, this work is going to provide the student with better understanding what constitutes safe and agile operation of autonomous transportation, with a sturdy platform for further development in area of vehicular motion.

In the first chapter, we are going to study, what makes an optimal collection of logic modules. How can they be divided into separate entities, granting both parallel execution & parallel development. In the second chapter, I am going to mention decisive factors behind choice of goals to pursuit for the time of work around goals of this document. In third chapter, we are going to find out how a robotics systems can be reliably coupled with a modern, advanced simulation software suit. We are going to explore the software platform, creation of environments, vehicles & scenarios. Then I am going to show, how virtualisation of physical interfaces helps reduce programming overhead, software complexity & not only speed up testing, but also make it more relevant to operation on real hardware.

After all this, we are going to conclude, with a study of the control system & its augmentation with variable speed controller. Thus, experiencing in full, how following good design practices can help us in better understanding of a problem, brainstorming probable solutions and implementing them.

ASSUMPTIONS & DISCLAIMERS

As we proceed on exploring possible augmentations in vision, computation & control models, we assume an implementation of all modules required to satisfy the threefold, robotic model already exists. If any deviations from a popular standard or presented basic implementation are required, they are to be described, or their implementation is to be presented. Otherwise, we assume their initial form or non-existence.

In the introductory chapters of this document, we mention choices in technology on both software & hardware side. With implications, their correct use bears on projects finance & time organisation. The author is not a trained professional and cannot guarantee that claims of increased awareness, gained from discussion of these solutions can and will improve operation of a robotic system unrelated to this project. These are examples of what has worked and continues to bear fruit in mentioned project & other endeavours of the author. Many details regarding implementation and functional details have been intentionally omitted, to maintain confidentiality of work provided by other contributors. Armed with this knowledge, we approach ideation, planning & implementation of possible approaches in improving confidence of pathing & control algorithms. These chapters include a proof of completeness and added value.

This paper may discuss subjects related to construction & nomenclature related to individual parts of a road going vehicle, without additional consideration whether the reader has been acquainted with such terminology. In the text, if some unpopular terms are used without additional explanation in attached glossary, reader is asked to refer to dictionary provided by Cambridge Dictionary, or definitions in Wikipedia, the free Encyclopedia.

1. ENGINEERING AN AUTONOMOUS DRIVING SYSTEM

Over the entire period of my study at Wrocław University of Technology, the Formula Student group "Racing Team" has been deeply engaged in development of an autonomously operating roboticized control system for their cars. This experience has allowed the contributing party, to traverse the entire engineering process, akin to a journey of an 'infant', maturing into adolescence. All the pre-emptive steps, from ideation, design, and engineering, through testing and hours of implementation, have led us to harvest results of this work. In the culmination of this process, we aim to clarify, how such a system is conceived and realized.

1.1. THE BUILDING BLOCKS OF A ROBOTIC SYSTEM

Fundamentals of a robotic system can be segmented into these precise steps:

- Contemplation of the problem and the objective of the eventual solution
- Evaluation of the eventual machine we intend to operate
  - Assessment of available sensor technology
  - Evaluation of available computational power
  - Assessment of available motor technology
- Examination of the environment in which we operate
- Consideration of available resources and personnel

These general steps, still cover considerable amount of complexity and would be excessively challenging to study in their entirety. This necessitates further separation of each topic, into more approachable scopes. To be precise, we are looking for topics, which can be considered individually. As long as each topic, can be divided into smaller subjects. Subjects, providing resources to the other subjects. We can steadily concentrate on satisfying specifications of each of these subjects, without being overwhelmed by the entire package. This is a basic principle of the Engineering Design Process

1.1.1. Contemplation of the problem

Embarking on the path of engineering a robotic system involves translating individual steps of a process into increasingly smaller segments. Given that the final machine requires control through minute electrical signals and movement of relatively imprecise motorics, each step of abstraction must be as descriptive and precise as possible. This has to be achieved, while accounting for noise and errors, resultant from operating on a statistically chaotic resolution of the external state and controls with margins of error. Significantly increased by the nature of loose coupling of mechanical components and their vague interactions with the environment.

In my experience, and through the experiences of other robotics engineers, a modular methodology is a great starting point for the design specification of a robotics system. Rarely can a project providing automated service, be assessed as a whole package. As per the definition of "A machine capable of performing a variety of often complex human tasks on command, by means of mechatronics, electronics, and programming solutions"[6], there is so much more to a robotics system, than what meets the eye. This has led to the recent emergence of currently leading robotics middleware suite, the "Robotic Operating System". This system has been actively developed by dozens of contributors since its inception in 2007, through the evolution of ROS2 in 2017, over 23 major releases of the package. *More about ROS in Ch.1 Engineering an autonomous driving system / Structure of the autonomous system*

To gain a more detailed understanding of how to effectively compartmentalize a robotics project into individual modules, we can draw inspiration from the most sophisticated autonomous beings known to mankind - humans. By examining how these autonomous beings operate, what they seek, how they determine their most fundamental goals, and how they achieve them, robotics pioneers have identified three key problems. When solved individually, these problems can be integrated to create an entity capable of traversing space.

— The localisation problem, encapsulated by the question,
  "Where am I?". This problem necessitates some form of sensing and reduction of spatial characteristics to establish a reference point for the present state.
— The self-governance and planning problem, represented by the question,
  "What is my goal?". This problem requires a deeper awareness of the past and present state in the environment to project future states into actionable instructions.
— The control and actuation problem, posed by the question,
  "How do I achieve set goal?". This problem focuses on the transcription of instructions into actions, through conversion of stored energy into motion.

This theory, succinctly summarized by the "See-Think-Act" maxim, forms the backbone of modern robotics and is the initial consideration in compartmentalization of robotic systems components.

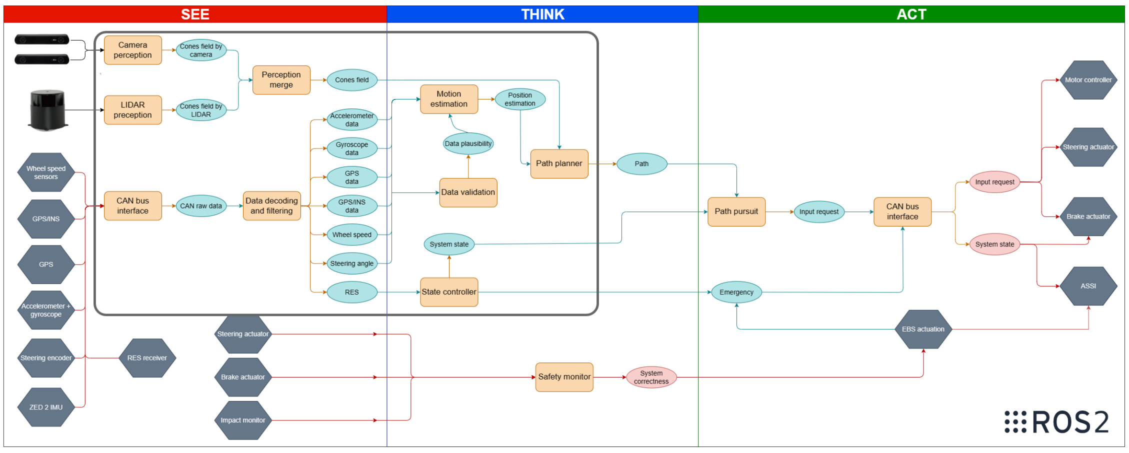


Fig. 1.1. Exemplary system design

1.1.2. Definition of robotic trinity

Having made our first division in the design of the robotics system, it is revealed that a robotic system can be analyzed like any other algorithm, which takes some form of input, transforms it, and produces output organized in a way that another system can receive as input.

1.1.2.1. Vision

Let us first consider the aspect of robotic vision. This term extends beyond definition of an input receiver. The information we acquire from sensors, such as pixels from a camera, cannot be directly utilized by the decision-making and pathing modules. Therefore, these signals need to be analyzed and transformed into a form, which discloses information, that can be acted upon. The usability of data acquired by a vision system is pivotal to accurate operation of every other module in our system, necessitating significant investment and refinement. For example, if the vision system does not include redundant components, a minor defect in one of the sensors, can halt the entire operation. If the input sensors are noisy or prone to drift, the performance will be significantly reduced despite best efforts from filtration, analysis & motion mechanisms. If the sensors do not reliably collect information about the task's environment, or if control information is not receivable in some states, the robot is unlikely to continue operation. It is important to be aware, that an automaton can use any kind of input source that helps it understand the nature of its environment. Just like humans, we are not limited to using cameras, lasers, and lidars. We can also consider limb motion monitors, magnetic line trackers/contactors, touch sensors/buttons, GPS systems, gyroscopes, etc. The vision system can be further divided into smaller modules responsible for connectivity, data acquisition of individual sensors/sources of control signals, signal filtration and error correction, obstacle/object/environment features detection, active localization, and environment mapping.

1.1.2.2. Governance & decision making

Once the vision system has completed a sequence of its planned operation and produced a frame of reference about the vehicle's state, we can pass it onto systems whose responsibility is to organize location and state data in temporal continuity and decide on the best next action. The decision-making part can be divided into smaller modules responsible for operation safety analysis and action, subsystems management/overseeing redundant operation, goal selection, and pathing.

1.1.2.3. Action

After the robot has determined its place in space and the next step in its operation has been planned, this information needs to be projected onto motors, actuators, displays, or other motion or communication modules. The action part can be divided into smaller modules responsible for transformation of absolute instructions into their relative derivations, acceptable by actuators controllers, external communications systems, protocoling and networks interfacing modules, motor controlers and electronics.

1.1.3. Considerations in project management

When general structure of a robotic system is understood, organisational aspects come into daylight. According to the "Guide to project management body of knowledge"[4] by "Project Management Institute", the preceding tasks were aimed at "envisioning the project". This envisioning of a robotic system in general, does not take into account any limitations posed for the final work and for the engineering process at hand. Before any research and fabrication can commence, it is essential to select what to research, what to prioritize and what are the constraints of budget, tooling, technology, or regulations of the environment/competition the product is intended for. How to transition from a vision, through budgeting, tooling, workspace allocation and operational goals?

For what can be considered a medium- to large- scale endeavour, organization and preparation define the whole experience. Aforementioned Guide[4] is a comprehensive 900-page resource of all the information one might need to organize one's work from the start. Materials like this are a holy grail in constituting success and failure of a robotics project, hence issues and processes outlined in the PMBOK[4] should not be overlooked. For instance, consider an exemplary project. The task is in creation of a controller for window blinds. A rotary position sensor, contactors at the edge positions, a radio receiver, and a small engine appear to be sufficient to complete the task on a weekend. But when we start dissecting the project into its individual components, analyzing how each should fit with the rest, how the final product should be assembled, how each electronic component needs to be powered and connected with the others, a complex network of wiring and individual parts emerges, each infinitely complex in their own right. Therefore, embarking on any robotic project should be undertaken with a threefold expected time reserve and only with prior study and preparation to handle each of the individual subsystems.

1.2. PROCESSES RELEVANT TO FORMULA STUDENT COMPETITION

With an understanding of workflow in a robotics project, before we delve into the structure of an autonomous system, we need to outline the machine we are going to operate and the rules of the environment/competition it is going to function in.

1.2.1. What is the Formula Student Car?

In competitive scenarios considered by this thesis, we are to operate an open-wheel passenger car, in a multitude of racing scenarios. Formula Student (FS) is the most established educational engineering tournaments series, with teams from all over the globe, and over 13 teams in Poland alone, struggle for the grand prize. The goal of Formula Student is to build a single-seated formula-style car with which they can compare against teams from all over the world not only in engineering and racing, but also in design, management and sales abilities. Formula Student challenges the team members to go the extra step in their education by incorporating intensive experience in design and manufacturing, but also in considering the economic aspects of the industry. The FS competition blueprint celebrated its 25th anniversary in 2023.[2]

For instance, the current machine built by the team from the Technical University of Wrocław is a nimble 300kg rocket, complete with a massive aerodynamic package, powered by two large 47kW electric motors, and equipped with individual suspension for each of its 4 racing tires. [5]

1.2.2. Important Rules & Restrictions in Roboticizing the Formula Student Car

1.2.2.1. General Rules of Formula Student Car

FSG cars are pieces of modern automotive architecture. With complex systems of carbon fiber, aluminum, printed plastics, and the finest components in modern propulsion. Capable of achieving acceleration in excess of 1G and deceleration over 2G, while generating a significant amount of downforce with massive aerodynamic packages. They are extremely dangerous creations for both the driver inside and anything that happens to be in their path. It is an exhilarating concept and even more thrilling when experienced in person. However, it is first and foremost part of a competition, targeted towards students with often limited abilities and foundations, who want to develop their skills from the ground up, through toughest challenges and interaction with latest technological solutions known to man. Because of these two factors, extensive sets of rules developed throughout mentioned years of Formula Student concept development are presented to candidates. Compliance with these rules is verified through rigorous pre-admission testing, conducted by Marshals, before these aspiring engineers get to test their creations on stage. First off, the vehicle size or complexity is not limited, beyond guidelines for structural rigidity, minimum safety requirements of the Monocoque form and materials. But it is required to be a 4-wheeled, Open Wheel car, which fits onto a 6m wide track, capable of carrying its own weight, including energy source, propulsion unit, and providing space for a driver, whether operated manually or fully by a computer.

Banned in professional forms of motor racing concept of a Fan Car, is acceptable in this formula & practiced by one or two teams on bigger competitions.

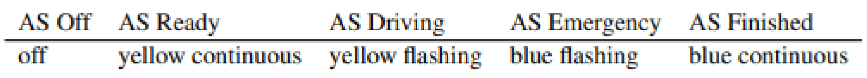
The engines, required to be either of internal combustion or electric type, have limited power output, restricted by cylinder stroke displacement or supply power in either case, to small 700cc, rarely exceeding 100bhp or 80kW of supply power.

This aspect of Formula Student creates initiative to reduce size, weight, complexity & cost, while maintaining safety precautions in pursuit of overall performance.

And with this knowledge, we have realized the gravity of the situation, and complexity of the project, without getting our hands dirty.

1.2.2.2. Rules around autonomous system & on-track goals

When it comes to augmenting the vehicle with hardware facilitating autonomous driving, there are strict guidelines revolving around acceptable states of operation which need to have procedures for remote and in-car control. And a whole 20 point paragraph about emergency braking, it's operation & required operation values. There are also many limitations emerging from general rules about vehicle structure, which impact our abilities in positioning actuators & sensors. When it comes to software, as long as it serves its function, allows for unobstructed and rapid engagement of safety systems, & supports all the expected operation modes, its structure is open for interpretation in its entirety. To aid us in complying with required safety limitations and standardising communication of machines state to marshals and operator, we are presented with a *status flow diagram* & indicator list with colors for each *status*.



AS Off	AS Ready	AS Driving	AS Emergency	AS Finished
off	yellow continuous	yellow flashing	blue flashing	blue continuous

Fig. 1.2. Autonomous System indicator list with colors for each *status*.

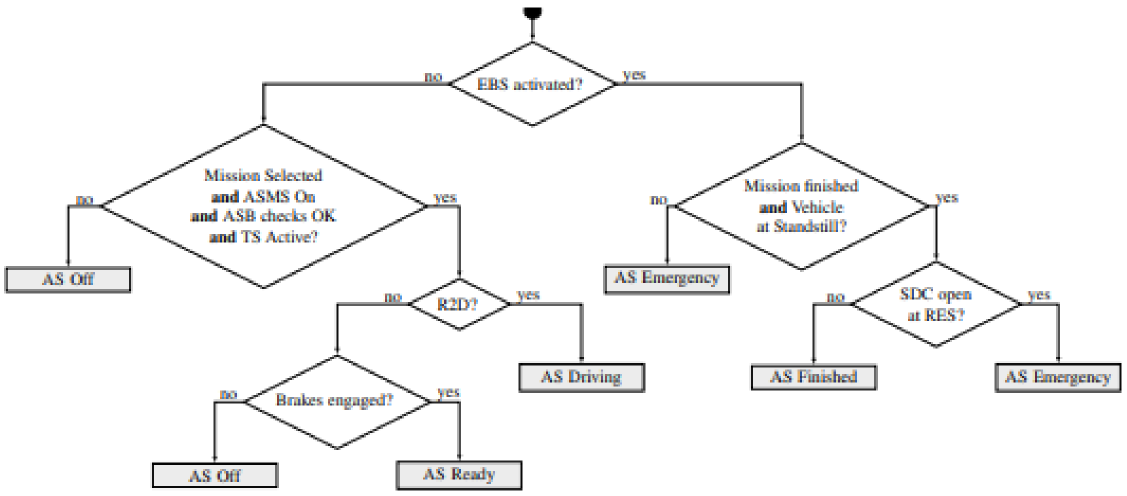


Fig. 1.3. Autonomous System *status flow diagram*

In operation, the autonomous system needs to have programmed goals & procedures to accomplish certain missions: Acceleration; Skidpad; Autocross; Trackdrive; Emergency Braking System (EBS) test; Inspection; Manual driving. Plus anything that the team might need during testing in pitlane. Remote operation is limited according to paragraphs mentioning RES device, where anything beyond "armed, start & emergency stop" signals is prohibited. All the configuration needs to be done through controls on car, or wired interfaces. No wireless steering functionality is permitted.

1.3. COMPILATION OF ACQUIRED KNOWLEDGE INTO A FUNCTIONAL AUTONOMOUS DRIVING SYSTEM

1.3.1. Structure of an autonomously operating car

In order to make development easier and more predictable during the concept phase, based on our current knowledge, we have made following assumptions:

1. **Desired sensorics**
   a) Vehicle speed (through rotation of wheels, or a Pitot Tube)
   b) Steering position
   c) Potential obstacles, track defining cones, including distance to them
   d) Position of the car relative to starting line, or absolute position on earth (gps)
   e) Status of on-board systems. Power unit, EBS, control systems, etc.
2. **Expected controls**
   a) Power unit output
   b) Steering motor position
   c) Braking force
   d) Operational status of on-board systems

3. **AMU** needs to support fast image manipulation, be a multi-core unit for improved distribution of tasks based on required latency of operation and the platform should help us in rapid development as much as possible.
   a) A data storage & exchange solution (being the Robotic Operating System 2 [ROS 2] LTS).
   b) Operating system compatible with ROS 2 (Linux kernel & Ubuntu operating system)
   c) A compute unit itself (Nvidia Jetson platform computer)
4. **Physical communication layer of the system**
   a) CAN 2.0b bus lines between sensors & control units (low weight, low latency, small throughput, high importance data transfer)
   b) Ethernet for devices like cameras, slave computation units & external control/monitoring units (high throughput, higher latency)
5. **Devices responsible for Autonomous operation**
   a) As modular as possible, allowing for prioritised engineering around the passenger/driver, improved serviceability & handling of special or emergency states
   b) Their presence should not block manual operation of the vehicle, especially in powered & engaged, manual driving state
   c) The vehicle has to permit manual operation when components of autonomous system (AMU, sensors, steering actuators) are disengaged or removed

Selection of ROS 2 as the main component in software development, allows us to focus on implementing logic of individual computational modules, instead of worrying about structure of internal communication between software components. Although, for us, its biggest selling point is not that it comes with "eProsima's Fast DataDistributionSystem", with standardised schematics for publisher/subscriber data packets exchanging definitions in both Cpp & Python. It is the fact, that it is an industry tested and approved solution, not only available for public use without royalties, but with many basic robotics operations extensively documented & presented in many examples. The community support for ROS project, also brings us solutions for almost plug-and-play interfacing, with all hardware solutions the team has come to include in the project. It is a perfect tool for both, initial engagement with robotics and professional, commercial solutions.

To physically connect all the components of our system, electronics team, has created a separate party of students, dedicated to handling the wiring harness & communication devices related to autonomy. It allows us to freely connect/disconnect systems, like brake, steering, display controllers, to/from the buses present on car, allowing us to comply with the 4th point of our assumptions.

1.3.2. Overview of software design for autonomous driving

When working with Robotic Operating System, we can see, that many elements of its structure are heavily standardised with concepts developed by industry leaders, through 15+ years of its development. These standards are not worth deviating from, so structure of the software solution relies mostly on what has been proposed by Open Robotics in many examples. Work has been divided into:

— **autonomy_interfaces:** custom messages and services, specified in ROS specific format. These definitions are used by ROS2 compiler to create internal data distribution structures.
— **autonomy_simulation:** software related to simulation of the vehicle, systems managing simulated operation & interfacing with currently adapted virtual environment provider
— **autonomy_system:** core of the autonomous system logic. Everything here is to be executed on the real vehicle.
  — **autonomy_perception**: solutions related to acquisition of physical sensor data & its translation into structures actionable by control systems
  — **mission_status_management**: software related to mission & status management
  — **autonomy_control**: solutions for operating on environment information, in both simulated & physical operation. Includes pathing & creation of instructions for actuator controllers.
— **network_interfaces:** CAN & Ethernet bus related software, like translation between ROS messages & linux kernel *socketcan*.
— **observability:** GUI modules used for system state manipulation and visualization, mostly with ROS RQT & Rviz tools.

1.3.3. History of relevant solutions in the autonomous system

Autonomy_simulation Initial attempts in finding a suitable software suit for housing a testing environment, have lost me months in experimentation. We have worked through solutions providing basic tooling for mathematical modeling & unit testing, like MatLab & its child project SimuLink. These systems work great for plugging in individual algorithms, like state management, or control system, providing some input information and verifying if their output matches our expectations. There was my personal, naive idea, incentivized by amateur experience in simulation racing games. That it cannot be difficult to use a commercial game, which already expertly models vehicle dynamics for purpose of fun and virtual competition, to be used as testing grounds for the complete package. As it turns out, getting data from closed, proprietary software and even worse, manipulating state of such software, is close to impossible without mastering software disassembly. Also, out of the question for a student project aiming for rapid iteration. Going a step further, into use of open source examples of vehicle dynamics code for use in game engines. They are a great place to learn how to create a drivable vehicle for purpose of entertainment, but these models are nowhere near to behaviour of a real vehicle, still require a lot of additional code. Most importantly, there are no examples how to effectively simulate desired sensors and no support from creators of the real hardware in creation of these representations. So, in the end, the team has gone to the trusted & endorsed by Open Robotics, GazeboSim suit. Great for simulation of robotic arms & simple navigators. Provides more or less advanced representations of real sensors, with at least some support from hardware vendors. And data synchronisation with interfacing to ROS, is a breeze to use. It has served the team well for a long time, in testing each iteration of complete autonomous operation package.

network_interfaces A package which had its conception only after many months into the project, when need & desire to work with real sensors first came to be. Initially, the software project did not go beyond the box of a desktop computer running Robotic Operating System. This resulted in programmers hardcoding interfacing with ROS nodes, where information from real hardware should be received or dispatched. This permitted us to postpone tasks like data packaging, serialization & understanding of network APIs provided by Linux Kernel, but when the need & desire to work with real world components arrived, we were scrambling to get the whole project transitioned from a single package interconnected with ROS nodes. Previous addiction to virtually unlimited power of data distribution inside a single machine, has caused us to disregard throughput limitations, latency requirements and prioritisation present when working with real interfaces. Consequently forcing the team to do a complete re-organisation of the project, with re-write of all the edge operating modules, split of low-performant modules into smaller problems and most importantly, separation of simulation through with the wall of physical interfaces in virtualised operation mode.

2. SIMULATION: A REPRESENTATION OF REALITY

As previously discussed, being a growing enthusiast, I have initially held a misconcepted image, that numerous steps can be omitted at the beginning of a robotics project, with expectation of little added overhead in later stages of development. Ensnared by previous experience in purely software sciences and assurance of simplified development made by ROS. While it might be true, that some simplification and focus on smaller steps is advantageous for educational purposes. In the end, realization of the whole engineering process first hand, from from inception to completion, has proven to be enlightening. But, when it comes to professional solutions, omitting any steps in continuous fulfilment of design objectives, incomplete blueprints, or disregard for impact of physical and computational inaccuracies, will inevitably lead to necessity for re-designs & re-writes, often in substantial portions of the codebase. The bigger the machine, the more complex coupling, the higher speeds & weight, the more inertial effects, material deformation & losses of precision. Yet, the inherently entropic nature of reality means that there are going to be imprecisions at every level of complexity. From Roomba cleaning robots, to heavy, industrial machinery.

2.1. BENEFITS IN STARTING WITH STATE-OF-THE-ART PHYSICS & VEHICLE SIMULATION

Drawing from personal experience, I would like to assert that integration of highly sophisticated simulated environments into the testing pipeline of robotics systems, particularly autonomous passenger cars, carries numerous advantages, with minimal drawbacks. Modern software suits designed for representation of real world characteristics of robots & vehicles, not facilitate comprehensive evaluation of automation systems under diverse and challenging conditions, but also aid in early detection of omitted steps in continuous fulfilment of design objectives. In my experience, incomplete design and disregard for impact of physical and computational inaccuracies, represent the greatest sink of time on debugging, redesigns & rewrites. Occurring multiple times after the product has been deemed final and scheduled for release.
Promoting such practices, in a world boiling with Agile, iterative, test driven development methodologies, is equivalent to fostering waste of time & resources on discarded prototypes and hours next to drawing boards, seeking defects on graph & projections.

By incorporating simulated environments early into the testing process, allows developers to identify and rectify these issues, before they escalate into costly problems beyond purview of a single department. High-fidelity simulations aid in this process revealing issues related not only to emloyed algorithms & procedures, but also to interactions between the robot and its environment, approximate performance of the robot's sensors, and effectiveness of robot's decision-making algorithms under challenging conditions.

Advanced simulated environments also enable testing beyond what would be feasible in the physical world. For instance, they can simulate dangerous situations that would be too risky to replicate in reality, or rare events that the robot might not encounter during its normal operation but still needs to be able to handle. Furthermore, the low-cost, no-risk nature of these simulated scenarios, not requiring physical hardware outside the computational units, permits exponentially higher volume of tests.

As mentioned in the History of relevant solutions in the autonomous system. Throughout years of experimentation, I have explored as many simulation solutions as the time permitted. From purely discreet, mathematical solutions, to those incorporated into workflow of this thesis. Both approaches have their place in the testing & validation process, like Unit Testing for discreet solutions and object detection tests for the advanced virtual environments. However, using sub-par solutions for simple tests, might be more cumbersome in development or slower in execution, using a sub-par solution for simulation of whole scenarios and real world operation is equal or even worse than not using anything.

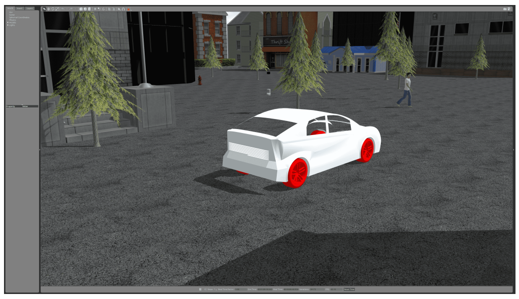


Fig. 2.1. Graphics of a popular robotics environment

Fig. 2.2. Graphics of a state-of-the-art simulation solution

2.2. CREATION OF PLUG-AND-PLAY INTERFACES BETWEEN SIMULATION & PHYSICAL SYSTEMS IS IMPORTANT

Drawing from previous discussions, where significant divergence in completeness of software & hardware parts of a robotic system is mentioned. I would like to point out that a software team, focused exclusively on working with data from virtualized systems for extended periods, is destined to offhandedly construct systems and software interfaces, which are not readily portable to work with the hardware they are intended for. Such events have led the Formula Student Autonomy project in Wrocław, to undertake a re-write of the data exchange code, reorganisation of ROS message types, and additional optimisations of code concerning throughput & priority. While stating & following limitations of target hardware, along with programming with these targets in mind, would have saved dozens of programming hours. For a student engineer, it has been difficult to make a convinced decision, which limitations are relevant to the project and need to be taken into account. However, an experienced engineer, aware of possible communication standards and what are their main uses, is not going to be faced with a problem, for which he is not going to be able to pick target components and verify, which sockets & communication standards they provide. Implementation of relevant interfacing does not pose much of an issue, as most of the details & edge cases are handled by libraries like SocketCAN (Linux Kernel), or DDS (Robotic Operating System). Their inclusion in the design is a matter of awareness and foresight, which result in not only major reduction in labor, but also their alignment with modular design, making them a valuable asset in the workflow.

2.3. UNIVERSAL INTERFACES & MODULAR DESIGN IMPROVE TESTING WORKFLOW SIGNIFICANTLY

When testing software of a robotic system in a simulation, we can choose to easily capture required memory state of the simulation & use it in our systems with addition of a few noise generators & control scripts. In no-time we get our robotic software interacting with virtual hardware. But what happens if we are tasked to adjust our software to work with real hardware? Or even just switch to a different simulation software suit? All the software working in sync with the state of current simulation, has to be adapted from ground up, to work with alternative interfaces, data types, libraries, clocks. Wouldn't a unified translation layer make things much simpler, when switching between these systems? We are already aiming to use some physical interfaces, for which some data serialization schematics are made. If we adapt our initial design to these schematics, we are no longer swapping between communication over physical interfaces and spoofing data in software. Instead we jump from virtual interfaces, made to operate indistinguishably from the real thing, straight into working with physical interfaces. If we put a little bit of effort into matching these two worlds, we are moving large portion of work destined for programming data adapters and investing it into productive work.

3. ENGINEERING SOFTWARE INTERFACES, UNIVERSALLY COMPLIANT WITH BEAMNG SIMULATION & REAL COMPONENTS

When attempting to couple a robotics system with simulated environment and the knowledge of used hardware components, especially their communication protocols is known, based on statements from ch. 2, I support the claim that maintaining communication over the same protocols, when working with real components & a simulated environment. When protocols & communication standards, for interfacing with physical components become known, it is the last moment, where development without support for these protocols, might not inflate technical debt.

The autonomous driving system we are working on for this thesis, has already been adapted from completely internal system, for data exchange between the logic and simulated environment. In following steps, I am going to describe my thinking & design process, with examples in creation of such interfaces with a machine running BeamNG.tech simulation suit.

3.1. DESIGNING THE ROBOTICS SYSTEM AROUND CONCEPT OF MODULARITY

To best understand the mode and priority of operation for each hypothetical module, which we are going to introduce into our robotics system, I have started analysis of the system based on “See-Think-Act” division trinity and began my work from desired output, through median steps, to required input.

Our Actuation third of the system, is to result in actuation of braking system, modulation of engine power & rotation of steering shaft. Final output signals, relevant to the software component, are to be instrumental for operation of a three distinct actuators, which need to function in relative harmony. In a simplified stack, these systems can be controlled from a single module, part of the Autonomy Main Unit (), but the project may include a Vehicle Performance Unit (). A VPU is to take control signals from AMU and augment these signals with high precision optimisations, at low latency and high rate. With VPU in the loop, the output signal from AMU no longer requires us to provide signals interpretable by a motor control board. In the case of our car, we are going to limit ourselves to outputting desired maximum speed of the center of mass & desired turning circle. Additional benefit of tasks separation between a lower frequency AMU and high frequency VPU, is the fact that BeamNG.tech software includes algorithms used in VPU. Systems like torque vectoring and traction control can be made part of the vehicle definition, sparing us the work in recreating these functions and their characteristics on our own.

To create these desired displacement vectors, we need to know between which points this traversal is permitted, safe & as optimal as we are able to make it. For this task, a pathing algorithm needs to be created, which is to take information about the environment around it and draw safe passage based on its immediate state. A basic pathing algorithm is enough to allow for low speed traversal. Unfortunately, basing our next step solely on information about current appearance of a few meters in front, does not leave us with much space for performance improvement.

To combat this issue, a long term pathing algorithm has to be added, parallel to the temporal pathing algorithm. This long term pathing algorithm is to operate on information from previously traversed & mapped environment. Creating this long term pathing algorithm & optimising its operation, in combination with modules providing input data, are a source of the biggest leap in on-track performance, allowing the car to reach speeds beyond few meters per second. The long term planning algorithm is to be combined with the temporal pathing algorithm in a path pursuit module, which takes on the incredibly important task of being the source of truth for the path the car is going to take in reality.

For the next steps, we consider the control hardware divided into:

— A set of motor controllers, responsible for powering control motors (part of simulation).
— A Vehicle Performance Unit, responsible for creation of precise, desired force instructions, based on physical capabilities of the car
— An Autonomy Unit, responsible for pathing and creation of motion vector for the VP unit.

3.2. CREATION OF TESTING SCENARIOS IN BEAMNG.TECH

3.2.1. The platform

BeamNg.drive is a full featured and versatile driving simulator with unique physics engine developed in-house and based on soft-body deformations powered by beams and simplified, interconnected structures. Its initial development was based on academic work in soft-body physics, turned into a multitude of video game editions. Initial creation named "rigs of rods", evolved to "beams" developed on top of Crytek CryEngine3 graphics engine and later moved to open source torque3D engine. Thus becoming the starting ground for current versions of the video game and simulation suit. Thanks to developers academic background & transparent approach to the video game as a catalyst to grow interest in the technology, an extended edition of the software has been worked on alongside the casual

experience. Called **BeamNG.tech**
Key features and capabilities of BeamNG.tech include[1]:

1. **Soft-Body Physics**: the physics model provides believably accurate simulation of real-world deformation characteristics and coupling inaccuracies of vehicle components. It enables simulation of various vehicle dynamics, including weight distribution, impact/tensile stress forces propagation & energy distribution of power train, with accurate losses. The soft-body physics engine is also applied to tire contact simulation, resulting in their incredibly believable behaviour, including different pressure levels, centrifugal force deformations at speed etc.
   The physical car model extends beyond soft-body parts. It includes impact of temperature on power and dissipation of this temperature, highly customizable aerodynamics characteristics (up to ability to create airplanes), etc.
2. **Model-in-Loop (MIL) Testing**: BeamNG.tech includes all the required tooling and control scripts for MIL testing. With straight forward integration of BeamNGpy library, the simulation can be easily launched, managed & controlled, on as many systems and scenarios as the user desires.
3. **API and Automation**: The BeamNG team provides not only a set of open-source tool for automation of services execution. They also provide support for integration of BeamNG.tech into any development framework with an extensive memory mapping API. Thus enabling testing & quality verification through reading and controlling every component of a car.
4. **Ready-made Sensors sim**: BeamNG.tech provides a set of simulated representations of data acquisition systems. Including an array of vision sensors, like cameras, Lidar, Radar & ultrasound, with access to plenty of other information, like position and speed of individual nodes and motors on the car.
5. **Customizable Vehicles & Drivetrain**: BeamNG.tech not only comes with a wide range of highly detailed and configurable vehicles with realistic driving dynamics. It also provides a complete tooling suit and a plethora of guides, for creating personalised vehicles, just as detailed as those provided with the software. This includes completely custom suspension, frame & powertrain characteristics. From CVT electric cars, to 12 gear diesel trucks. Vehicle behaviour and operation of individual components can also be extended, using Lua extensions, allowing for fast development iteration times.
6. **Traffic Simulation**: Autonomous driving training and testing extends to creation of real, public road traffic scenarios in various environments. Custom paths and behaviour for traffic can be scripted, including correct reactions to signalisation and road rules, with randomised placement of vehicles ranging from small, city cars, to long range haul trucks.
7. **Environments**: BeamNG.tech offers 12 hand-crafted, open-world environments with

hundreds of miles of roads to traverse. Spanning from city centers with complex lane layouts, through rural villages to highways in rural environments.

8. **Complete customisation**: BeamNG.tech provides editors for everything, from custom scenarios, full environment sculpting and object placement, up to on-the-fly customization/swapping of vehicle parts. The World Editor is a powerful tool, allowing for creations on par and better than what is provided by the development team.

When working on autonomous driving scenarios, we are provided with everything that one might need to test the robotic system. The only limit is our imagination and desired time investment for experimentation. For the purpose of preparing for formula student competition, we can start from simple cone track configuration on a flat plane, up to recreations of real stadiums we get to compete on.

3.2.2. The environments

To support engineering work in this thesis, we are going to create two, simple scenarios. One being an autocross circuit and one being an acceleration stage. To focus on providing the basic facilities, neither including additional obstacles, or changes in ground elevation. Thus meaning we are only to create traffic cone limited spaces, understood by the autonomous systems programming and standardised by the formula student guide books.

BeamNG provides three methodologies for creation of objects/obstacles.

1. **Static, non-interactable objects** - 3d objects or sprites on 2d planes, without defined or activated collision properties. These objects cannot be collided with or displaced in any way, other than modification in World Editor. Can be used for saving resources on objects that are not reachable by the actor, or in our case, to allow the controller to make mistakes and continue driving without requiring to restart the whole scenario.
2. **Static objects, with collisions** - 3d objects or sprites on 2d planes, with defined and activated collision meshes. These objects also cannot be displaced in simulation and collisions with them are always hard and not-deforming. Can be used to define hard limits of the track, or for unforgiving testing, where any mistake is most often going to result in halting of the car.
3. **Vehicles** - Every object in BeamNG which has weight, soft-body deformation physics and ability to be displaced, has to be defined as a vehicle. This allows us to place cones which can impede movement of the car, deform and be left on track in unexpected locations. Thanks to this, we can test how the localisation and mapping system handles rejection of such anomalies and if it stays on previously defined track.
   Unfortunately, the fact that such object inherits properties of a vehicle and has the same computational priority as our main actor (formula car), creating scenarios filled with complex, deformable environments, quickly overwhelms even the most powerful modern CPUs.

For the purpose of this thesis, we have created both autocross and acceleration scenarios, filled with traffic cones of each of the type. Non-interactable version is to be used for basic testing, deformable version used for testing of localisation and mapping and one with hard cones, is to be used for unit testing and some planned optimisation of path planner, based on machine learning.

A method for manual tracing of images of tracks, has been described in a project previously aimed to be used with Gazebo. `https://github.com/HighPriest/SvgGazeboCones` The output XML structure is easily translatable into XML structure used by BeamNG scenarios.

3.2.3. The Vehicle

Vehicles in BeamNG.drive are contained in freely movable directories, which are required to contain:

— **info.json** file - a file describing information visible in GUI, like name, author, drivetrain type
— **mesh & texture** files - files which define visual appearance of the car (e.g. .dae, .dds, .png extensions)
— **jbeam** files - files which define the beam structure of each vehicle component and their default properties. This includes information about powertrain, brakes & suspension.
— **lua** scripts - scripts which executed with instantiation of the car model. They allow for interaction with systems present on the car. Including driving & sensors configuration
— **part configuration (.pc)** files - files defining compilations of standardised components named and defined in jbeam files.

Other than the main structure definitions held in .jbeam files, contents and creation of each of these files is presumably straight forward. Contents of **.jbeam** files follow a hierarchical structure written in JSON format. Exemplary structure of car chassis is as follows:

```
"object_name": { // whatever we like
"information":{
  "authors":"Kamil Śmigielski",
  "name":"BeamNG Chassis Part", // whatever we like
},
"slotType" : "FormulaCar_Chassis", // personalized name of the component,
↪  or main for the top most definition
"slots": [
  ["type", "default", "description", "additional parameters"]
  ["FormulaCar_Engine","FormulaCar_Engine", "Engine",{"nodeOffset":"1"}],
],
"controller": [ // links to sub-modules that control various functions of
↪  a vehicle through bindings in lua scripts.
```

```
    ["fileName"],
    ["vehicleController", {}],
  ],
  "refNodes":[ // shared reference points for car orientation
    ["ref:", "back:", "left:", "up:", "leftCorner:",
    ↪  "rightCorner:"],
    ["fr3r", "fr4r", "fr3l", "fr3tr", "fr1l", "fr1r"],
  ],
  "flexbodies":[ // links to meshes and with which jbeam group they are to
  ↪  deform
    ["mesh", "[group]:", "nonFlexMaterials"],
    ["TPUF_Body", ["FormulaSpaceframe"]],
    ["TPUF_BodyBolts", ["FormulaSpaceframe"]],
    ["TPUF_Frame", ["FormulaSpaceframe"]],
    ["TPUF_FrameRails", ["FormulaSpaceframe"]],
    ["TPUF_Dash", ["FormulaSpaceframe"]],
    ["TPUF_DashBolts", ["FormulaSpaceframe"]],
    ["TPUF_DashSwitches", ["FormulaSpaceframe"]],
    ["TPUF_Windscreen", ["FormulaSpaceframe"]],
    ["TPUF_WindscreenBolt", ["FormulaSpaceframe"]]
  ],
  "nodes":[ // points of attachment for beams, these should not be edited by
  ↪  hand without aid of vehicle editor and beam visualization tool
    ["id", "posX", "posY", "posZ"],
    {"selfCollision":true},
    {"collision":true},
    {"nodeMaterial":"|NM_METAL"},
    {"frictionCoef":0.6},
    {"group":"FormulaSpaceframe"},

    {"nodeWeight":1.6},
    //Floor
    ["fr1l", 0.178, -1.311, 0.09],
    ["fr3l", 0.32, -0.175, 0.09],

    ["fr1r", -0.178, -1.311, 0.09],
    ["fr3r", -0.32, -0.175, 0.09],
    {"nodeWeight":2},
    ["fr2l", 0.29, -0.75, 0.09],
    ["fr2r", -0.29, -0.75, 0.09],
    ["fr4l", 0.32, 0.42, 0.09],
    ["fr4r", -0.32, 0.42, 0.09]
  ],
  "beams":[ // spring & damper connections between nodes
    ["id1:", "id2:"],
    {"deformLimitExpansion":1.2},
    {"beamPrecompression":1, "beamType":"|NORMAL"},
    {"beamSpring":1501000,"beamDamp":150},
    {"beamDeform":50000,"beamStrength":"FLT_MAX"},

    //floor
    ["fr1l","fr2l"],
    ["fr2l","fr3l"],
    ["fr3l","fr4l"],
    ["fr4l","fr5l"],
    ["fr1r","fr2r"],
    ["fr2r","fr3r"],
    ["fr3r","fr4r"],
    ["fr4r","fr5r"],
    {"deformGroup":"taillight_break", "deformationTriggerRatio":0.01},
    ["fr5l","fr6l"],
    ["fr5r","fr6r"],
    {"deformGroup":""}
  ],
  "variables": [
    ["name", "type", "unit", "category", "default", "min", "max", "title",
     ↪  "description"]
    ["$brakestrength", "range", "", "Brakes", 1, 0.6, 1.0, "Brake Force",
        ↪  "Scales the overall brake torque for this car", {"minDis":60,
        ↪  "maxDis":100}]
    ["$ffbstrength", "range", "", "Chassis", 1, 0.5, 1.5, "Force Feedback",
        ↪  "Scales the force feedback strength for this car", {"minDis":50,
        ↪  "maxDis":200}]
  ],
  }
```

Sensors are not part of the jbeam structure, instead they are instantiated completely dynamically. Either inside the always present, internal .lua scripts of the vehicle, or by external API. By example, an Inertial Measurement Unit can be created with

```
extensions.load('imu')
imu.addIMU(name, pos, [debug])
// or imu.addIMUAtNode(name, node, [debug])
```

then we can poll the IMU instance for its measurements. If it has been created correctly, it should start returning an array structure:

```
{ "aX": "0.00046557877210677",
  "aY": "4.8179574853672",
  "aZ": "0.0015101531196775",
  "gX": "-3.9092405963171",
  "gY": "-5.8214835947941",
  "gZ": "0.026276952910208",
  "name": "IMU",
  "pos": "vec3(0.73,0.51,0.8)" }
```

3.3. CREATION OF ‘VIRTUAL’ INTERFACES IN BEAMNG.TECH. POLLING SENSORS & REQUESTING CONTROLS.

Before we are able to start working on interfacing between the simulation and autonomous system programming, we need to go through a few preliminary steps:

1. What are the requirements for execution of these programs. Their compatibility with operating system & hardware requirements?
2. How we actually need this software to be distributed? Should everything run on a single machine, multiple machines, a high power, cloud computational service?

Software for autonomous operation has been written from ground up to function on the target Main Unit. As this main unit has been a Nvidia Jetson solution for a few years now, the programming relies heavily on its features, with a few compatibility additions which allow it to work on a compatible Linux machine equipped with a modern Nvidia GPU. This forces us to run it on either Jetson hardware, or a workstation with highly customised Linux operating system. Here is a hardware requirements form for autonomous system

```
OS: Ubuntu Linux 20 + ROS2 Foxy
CPU: Quad core 2GHz equivalent of a recent intel core cpu
RAM: 512 MB
GPU: NVIDIA GPU with CUDA Compute Capability beyond 8.X
    & compatibility with CUDA Toolkit 11.4+ (RTX 3XXX series+)
Storage: 1GB
```

On the other hand, BeamNG has been primarily written to work on popular for recreational use, Windows operating system. Although in recent releases a Linux compatible executable has been introduced, its created with help of an emulation layer for heavily relied on Windows specific APIs. This makes the Linux version much less performant and often unpredictable in execution stability. At the time of writing, developer team works on porting the graphics API to VulkanAPI, which should allow the simulated environment to be safely executed on Linux OS. Here are the published requirements for a machine running BeamNG

```
OS: Windows 10 64-Bit
CPU: AMD Ryzen 7 1700 3.0Ghz / Intel Core i7-6700 3.4Ghz (or better)
RAM: 16 GB RAM (less in our case, when running simple scenarios)
GPU: Radeon HD 7750 / NVidia GeForce GTX 550 Ti
Storage: 20 GB available space
```

It becomes apparent that to run both packages on a single machine, it needs to be a recently released, powerful workstation machine, capable of performant virtualisation of two operating systems & direct assignment of CUDA units strictly to one OS.

Or, two machines. One, a popular grade gaming desktop to support autonomy software and a cheap, second hand workstation to support the simulation.

3.3.1. Architecture

To know for which interfaces a connection, data source or data sink needs to be created, we need to take a look at the current design of the autonomous system, destined for assimilation.

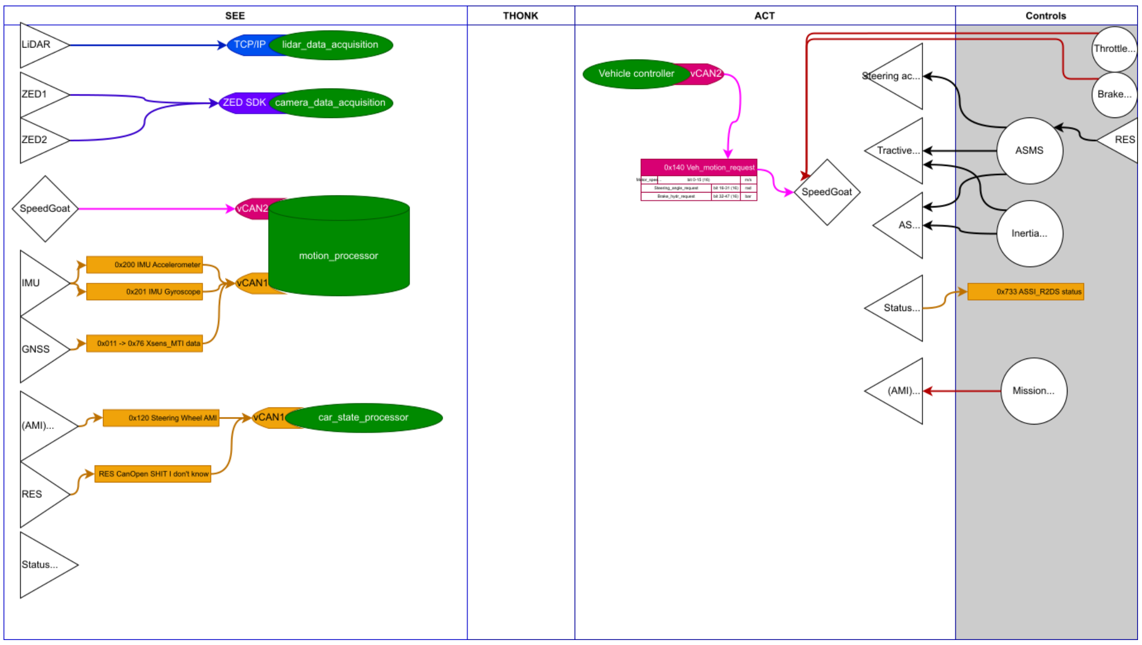


Fig. 3.1. Interfacing architecture

On the figure 3.1, we can observe, that the system ingests a deluge of information, coming in over two CAN lines and two TCP/IP connections. One IP connection coming from LIDAR and one from Camera combined system. The state changing controls, other than actual electric power engaging switches, are also coming in through CAN lines. Their representation can be added into interactive part of the simulation. As the main aim for the simulation suit is to be the external program we launch from the workstation we develop autonomous system on, it is not a priority.

Through design and implementation attempts, I have concluded that the most straightforward topology for remote interfacing is a flow similar to this diagram.

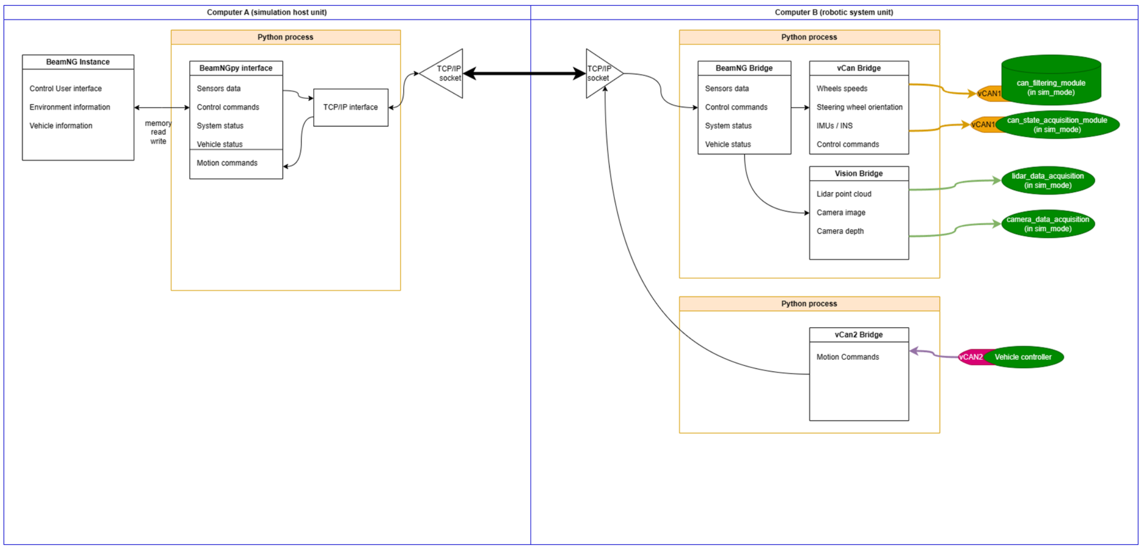


Fig. 3.2. Network & data exchange design

Here, a BeamNG Bridge process is executed on the machine housing the Robotic Operating System. This bridge creates a TCP/IP connection with a computer containing simulation software. From the autonomy machine, we choose desired scenario, requests its launch & start exchange of sensors & control information through TCP/IP and back through ROS Nodes. Received information is then pushed onto vCan transcribers, which push the information from the sim, into packets expected by the data reception & filtration modules.

An alternative, is to acquire a machine which can both run the simulation and either includes CAN ports like the Nvidia Jetson computer, or a daughter board with these ports. Then the AMU could be removed from the car and connected to an indistinguishable from real thing, virtual environment. This requires execution of BeamNG bridge wholly on the Simulation machine, including vCan transcription modules.

3.3.2. Programming

Now we can begin working on programming the desired components on both sides of the computational exchange. Starting on the side of simulation, we need to instantiate & make following parts available over the network:

— 1x Lidar - 3D point cloud, with parameters similar to Velodyne HDL-64E
— 1x Camera - Colour image - Pixel-wise depth map & object tagging
— 1x INS (GPS + Inertial unit) - Absolute vehicle positioning
— 2x IMU - Temporally relative vehicle positioning
— 4x Wheel speed
— 1x Steering column position

Motors:

— 1x 80kW electric motor
— No clutch or gear system

Additional controls:

— Start, emergency stop buttons
— Mission selection buttons

A `BeamNGClient` object is created, to manage information related to connections with the simulation machine.

```
default_scenario = Scenario('smallgrid', 'Default',
↪  '/levels/smallgrid/scenarios/Default.json')
default_vehicle = Vehicle('local_identifier',
↪  model='name_of_the_car_in_files',
↪  part_config='vehicles/name_of_the_car/PartsCollection.pc',
↪  license='AI')

def __init__(self, scenario=default_scenario, vehicle=default_vehicle):
  self.client = None
  self.scenario = scenario
  self.vehicle = vehicle

def connectClient(self):
  beamng = BeamNGpy('localhost', 64256, home=BNG_HOME, user=BNG_USER)
  try:
    self.client = beamng.open(launch=False, deploy=False)
  except:
    self.client = beamng.open(launch=True)
```

BeamNGpy middleware is written in a way, where the `BeamNGpy` class is used mainly to initiate connections, to which objects caching information about the world are attached. Then, these world objects are automatically filled with new information, whenever we interact with them. Hence we create `scenario` & `vehicle` objects. These objects are prefilled with already known definitions, but can be reassigned dynamically, with whatever world / vehicle we might desire. Scenario changes require a reload, but vehicles can be removed & added at will. Also, if the local software crashes, we allow for re-connection to the already running simulation.

Here is an example of dynamic scenario creation, where we add the vehicle into the world, by hand.

```
def loadScenario(self, scenario=None):
  if scenario is None:
    self.scenario = self.default_scenario
    if self.scenario.get_vehicle('TPUFormulaBee') is None:
      self.vehicle = self.default_vehicle
      self.scenario.add_vehicle(
        self.vehicle, pos=(-29, -39, 0), rot_quat=(0, 0, -0.7071068,
        ↪  0.7071068))
      #self.scenario.make(self.client)
    self.client.scenario.load(self.scenario,
    ↪  connect_existing_vehicles=False, connect_player_vehicle=True)
    self.client.settings.set_deterministic()
    self.client.settings.set_steps_per_second(60)
  else:
    self.client.scenario.load(self.scenario)
```

Then, in the main program loop, all we need to do is:

```
beamng = BeamNGClient()
beamng.connectClient()
beamng.enterScenario() #additional wrapper around loadScenario, loading
↪  the default one, starting execution and enabling all the additional
↪  features
```

BeamNG allows for instantiation of sensors using both internal .lua scripts and external API, with which we can interact using BeamNGpy middleware. In our example, we want to create a lidar sensor, store its readings and advertise them on ROS network as a `pointcloud`.

```
from sensor_msgs.msg import PointCloud, ChannelFloat32
from geometry_msgs.msg import Point32

self.lidar_publisher = self.create_publisher(
  msg_type=PointCloud,
  topic=self.get_parameter('topic_lidar').value,
  qos_profile=10
)
self.msg_lidar = PointCloud()

lidar = Lidar('lidar', beamng.client, beamng.vehicle, pos=(0,-1.7,0.2),
↪  dir=(0, -1, 0), horizontal_angle=180, requested_update_time=0.01,
↪  horizontal_angle=60, is_visualised=True)

def publishLidar(self):
  data = self.lidar.poll()
  self.msg_lidar.header.stamp = self.get_clock().now().to_msg()
  points = data['pointCloud']
  intensity = data['colours']
  for point in points:
    self.msg_lidar.points.append(point)
  self.msg_lidar.channels.intensity.values = intensity
  self.lidar_publisher.publish(msg_lidar)
```

We can then make the lidar readouts be published on a timer, or through querying the simulation, whether a new lidar reading is available and sending this new reading

```
if lidar.is_ad_hoc_poll_request_ready():
  self.publishLidar()
```

Because the Lidar we use is connected through internet and has its own ROS interpreter, we don't need to convert it into CAN Frames.

But, for motion sensors (IMU), on each update of data in their ROS node, a data conversion is done by a transmission emulator written in Cpp

```
frame.header.stamp = imu->header.stamp;
// frame 1
frame.id = 0x174;
frame.dlc = 8;
const double factor_gyro = 0.005;
const double offset_gyro = -163.84;
auto val =
  static_cast<uint16_t>(((imu->angular_velocity.z * kRadToDeg) -
  ↪ offset_gyro) / factor_gyro);
frame.data[0] = (val >> 0) & 0xFF;
frame.data[1] = (val >> 8) & 0xFF;
const double factor_acc = 0.000127465;
const double offset_acc = -4.1768; // simulates gravity reading this
↪ sensor doesn't account for on its own
val =
  static_cast<uint16_t>(((imu->linear_acceleration.y * k_m_s2_toG) -
  ↪ offset_acc) / factor_acc);
frame.data[4] = (val >> 0) & 0xFF;
frame.data[5] = (val >> 8) & 0xFF;
can1_transmit_pub_->publish(frame);
// frame 2
frame.id = 0x178;
frame.dlc = 8;
val = static_cast<uint16_t>(((imu->angular_velocity.x * kRadToDeg) -
↪ offset_gyro) / factor_gyro);
frame.data[0] = (val >> 0) & 0xFF;
frame.data[1] = (val >> 8) & 0xFF;
val =
  static_cast<uint16_t>(((imu->linear_acceleration.x * k_m_s2_toG) -
  ↪ offset_acc) / factor_acc);
frame.data[4] = (val >> 0) & 0xFF;
frame.data[5] = (val >> 8) & 0xFF;
can1_transmit_pub_->publish(frame);
```

```
// frame 3
frame.id = 0x17C;
frame.dlc = 8;
val =
  static_cast<uint16_t>(((imu->linear_acceleration.z * k_m_s2_toG) -
   ↪  offset_acc) / factor_acc);
frame.data[4] = (val >> 0) & 0xFF;
frame.data[5] = (val >> 8) & 0xFF;
can1_transmit_pub_->publish(frame);
```

3.3.3. Testing

Thanks to these creations, we can start with a test of our object classification model using BeamNGs pixel-wise object tagging and comparing this output with the areas our model places the object at. This, incredibly, can be used to incorporate machine learning into the process and improve precision of object tagging, by training on theoretically infinite set of data.

Until now, all the object recognition data has been created & labeled by hand by us and also other competitors, then shared in a public repository at `https://www.fsoco-dataset.com/`. This is a very good base, but we can use our newly created infinite worlds with perfect data, to enhance the dataset with images from a virtual camera. For this, we are going to run the simulation on multiple different tracks collecting images with precise position of the obstacle. Then, these images don't have to be labeled by hand using LabelImg software `https://github.com/heartexlabs/labelImg`, like it was done until now, but large dataset is going to be created automatically. In the future, generated images can be even enhanced with filters imitating dirt on lens or sun glare.

As we will be able to observe in following chapter, the simulated environment allows us to quickly test & benchmark changes to pathing & control systems aswell.

4. EXTENDING PATH FOLLOWING ALGORITHM WITH SPEED CONTROL

A simulated environment provides us with an infinite collection of testing scenarios and provides pixel-perfect methods for verification & tuning of positioning and pathing algorithms. An invaluable tool for study, experimentation & development of control systems.

In this part, we are going to extend an algorithm for pure pursuit of a temporally limited path. And study, how such a system can be extended.

4.1. DISTILLING TRACK DRIVING RULES OF FORMULA STUDENT COMPETITION

The circuit driving based contest is divided into stages of a singular Exploration lap and a set of Endurance laps, both timed, but timed separately. Exploration lap requires from us not only to pass successfully, but also to create a map of the environment for further laps. Because of this, our best bet for the Exploration lap, is to ensure reliable conquest of the track, to not compromise mapping accuracy. While this contest is also timed, we can not compromise in haste.

In Endurance contest, when the circuit is mapped, we are able to focus on full path optimisation, by creating both expected position and speed profiles for the whole circuit, using pre-calculated models.

4.2. UNDERSTANDING THE PATH PLANNING ALGORITHM

For the task of defining a path through a field of points, a Rapidly-exploring Random Tree algorithm, ideated, implemented & shared publicly, by Maxim Yastremsky [7]

The path planner operates by creation of triangular connections between 2 dimensional vectors, using Delaunay triangulation algorithm. These 2 dimensional vectors, have been defined by prior obstacle detection system. Because of potentially infinite complexity of such process, it is limited, by both certainty in object detection & tunable limitation in amount of triangles.

A stack, listing connections between centers of edges of Delanuated triangles, is created. Then, the Rapidly-exploring Random Tree algorithm draws possible paths, limited by additional control factors, defined by cost of their respectable characteristics. E.g. Blue cones on the left of the connection, yellow on the right, angle change below 60deg, or general heading direction away from the start point.

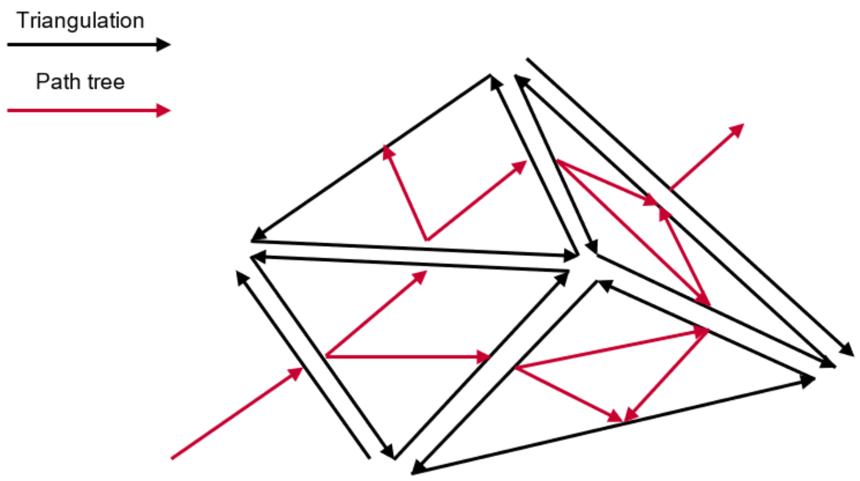


Fig. 4.1. Delaunay Triangulation

This algorithm permits us to traverse a world with only knowledge about feature immediately in front of the subject. At each update step, detected obstacles are fed back to a Simultaneous Localisation & Mapping algorithm, which stores their positions for later use. When the whole track is traversed and the SLAM algorithm detects that the subject is back at the start, vehicle is put into stopping state & the Trackdrive algorithm seizes operation.

4.3. SPEED AND ACCELERATION OPTIMISATION IN Trackdrive LAP

There exists a multitude of plausible approaches for controlling vehicle speed and acceleration rate. First, here are some clarifications of destined properties we are to work with. These are going to help us understand how to approach the calculations.

— Speed (goal): as in "The next point in our path, has the Speed goal of e.g. $50km/h$"
— Acceleration (goal): as in "To achieve $50km/h$ at the next point, we need to accelerate at rate of $5m/s^2$"
— Jerk (goal): as in "how quickly can we increase the acceleration to reach desired acceleration rate"

To calculate Speed (goal), we can seek maximised product of turning radius (angle between points connecting vectors) and lateral grip algorithm (Clamped for safety).
To calculate Acceleration (goal), we can seek maximised product of speed difference from current to goal at next point. Because we want to reach the goal speed not at the goal, but ASAP, distance is not needed.
To calculate Jerk (goal), we can seek maximized rate of change of acceleration, while

limiting calculated value, by factors preventing loss of longitudinal grip & unnecessary wear and tear on the vehicle.

Because there are multiple layers of decision making in creation of control signals, we need to study & decide, which layer is going to take care of which limitations. These layers are: Autonomy Main Unit; Vehicle Performance Unit; Electronic Control Board

An easier solution is to create a 0-100% engine power request (Figure: 4.2), calculated by a hand-tuned PID controller, straight from the AMU software, without bothering oneself with resultant torque.
To avoid twitchy motion, very high Integration(PID) values might need to be used, or alternative base states might need to be provided, based on the current state of operation.
To avoid loss of traction, a torque clamping algorithm needs to be employed. This software module is to observe change in wheel speed and react to sudden acceleration spikes, with significant reduction of output signal.
To avoid exceeding safe speed, we need both safe braking distance calculation & absolute maximum speed we feel safe to let the vehicle travel at.

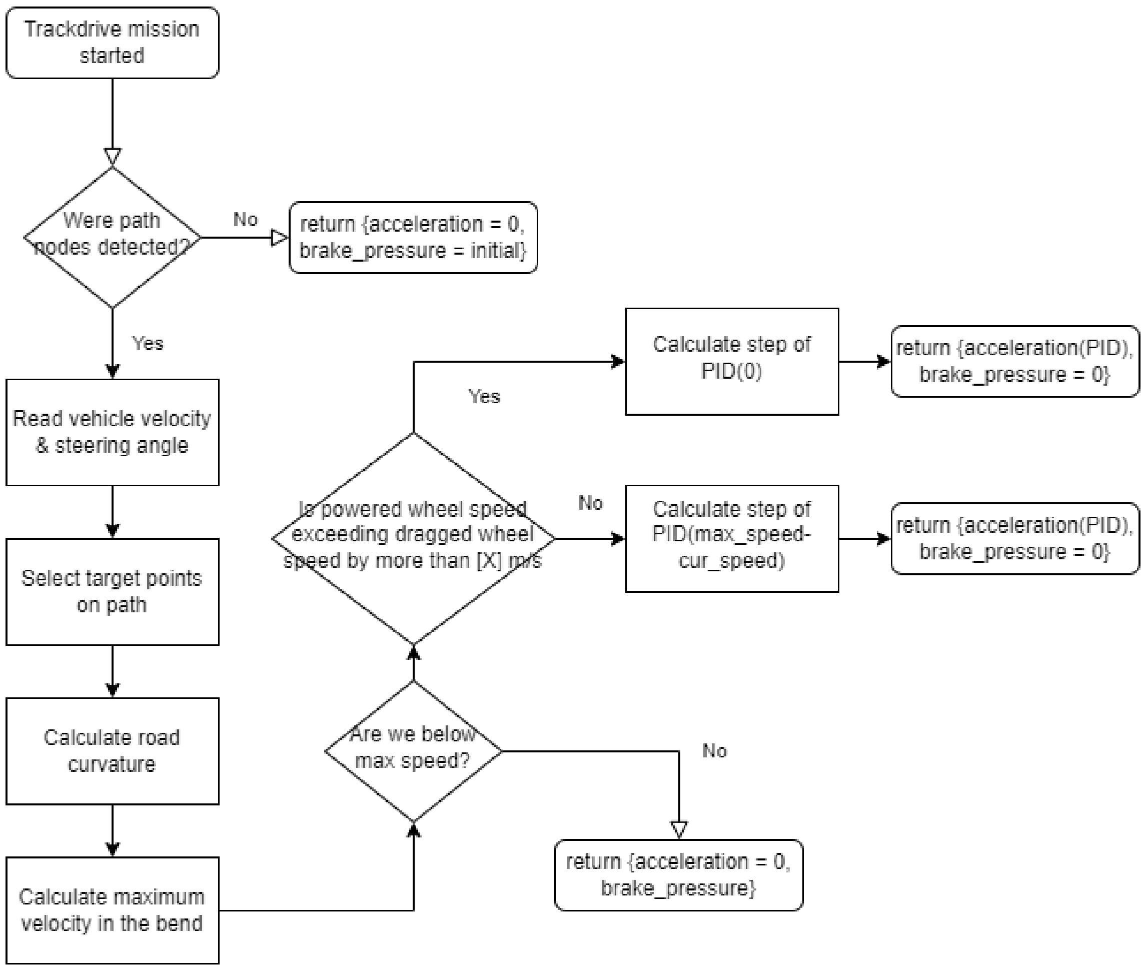

Trackdrive mission started
Were path nodes detected?
No
return {acceleration = 0, brake_pressure = initial}
Yes
Read vehicle velocity & steering angle
Select target points on path
Calculate road curvature
Calculate maximum velocity in the bend
Are we below max speed?
No
return {acceleration = 0, brake_pressure}
Is powered wheel speed exceeding dragged wheel speed by more than [X] m/s
Yes
Calculate step of PID(0)
return {acceleration(PID), brake_pressure = 0}
No
Calculate step of PID(max_speed-cur_speed)
return {acceleration(PID), brake_pressure = 0}

Fig. 4.2. FlowChart of simplified acceleration controller

Second proposition (Figure: 4.3), is to reduce influence of AMU on resultant torque.

And passing this responsibility, to modules capable of higher frequency operation, on larger set of input data. Through communication of desired speed straight from AMU, to VPU, we can exploit its features & precalculated power-to-acceleration LUT, for high fidelity governance over longitudinal acceleration.
This implementation requires coordination with the team team responsible for the powertrain. First, results of dynamo-metric measurements is required. Secondly, tire grip functions need to be created & embedded into the script.

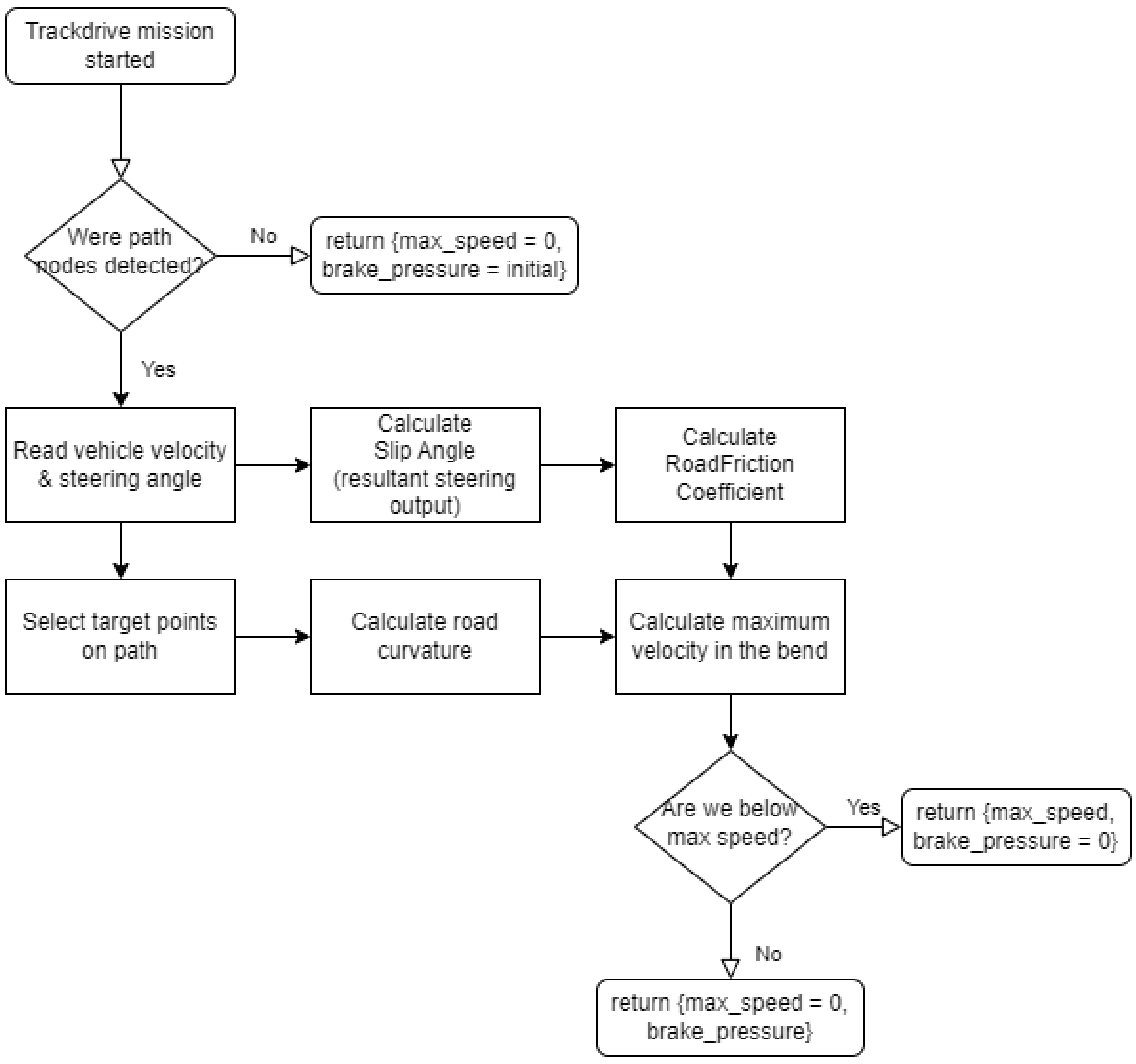

Trackdrive mission started
Were path nodes detected?
No
return {max_speed = 0, brake_pressure = initial}
Yes
Read vehicle velocity & steering angle
Calculate Slip Angle (resultant steering output)
Calculate RoadFriction Coefficient
Select target points on path
Calculate road curvature
Calculate maximum velocity in the bend
Are we below max speed?
Yes
return {max_speed, brake_pressure = 0}
No
return {max_speed = 0, brake_pressure}

Fig. 4.3. FlowChart of advanced speed controller

4.4. IMPLEMENTATION

To comply with goals of this thesis, we are going to focus on implementation of the latter solution, incorporating variability in speed into the path following script.

Variables required for calculations:

— $V_{frontwheels}$

— $V_{longitudinal}$ & $V_{lateral}$
— Pacejka Model Coefficients: $B = 10, C = 1.9, D = 1.0, E = 0.97$
— $D_{lookahead}$ minimum distance for objective seeking & steering destination

Calculations to be executed:
Current Slip Angle

$$L_{sa} = (V_{frontwheels} - V_{longitudinal})/V_{longitudinal}$$

Friction Coefficient

$$\mu(\rho) = D_{Pacejka} * sin(C_{Pacejka} * atan(B_{Pacejka} * L_{sa} - E * (B * L_{sa} - atan(B * L_{sa}))))$$

Distance (max)

$$D = D_{lookahead} + (V_{frontwheels}) * \mu(\rho)$$

Road angle

$$R_{angle} = angle(D) - angle(D_{lookahead})$$

Bend curvature

$$\frac{(2 * sin(R_{angle}))}{D_{lookahead}}$$

Speed (max)

$$V_{max} = \frac{Gravity * \mu(\rho)}{Bend_{curvature}}$$

Acceleration (goal)

$$A_g = \frac{V_{max}^2 - V_{current}^2}{2D}$$

Since loss of stable tire friction results in significant deviations from movement trajectory, we are not aiming to drive at the absolute maximum possible speed.

These calculations are based on Masters Thesis:
Longitudinal and lateral control of an autonomous racing vehicle[3]

Coefficients $B = 10, C = 1.9, D = 1.0, E = 0.97$ for the Friction Coefficient calculations, based on Pacejka Tire Model, have been sourced from proprietary specifications, provided by the manufacturer of Hoosier Tires. The resultant operation is satisfying, no further consideration is given.

For more accurate speed control, calculation of road-tire friction coefficients, specific for the whole vehicle package are required. These can be determined solely, by complicated, experimental evaluation, or approximated through simulations.

This model is thought out to work in natural/analog stages. Not through discrete modes of operation, but by mathematical functions.

— If the vehicle is traversing on a straight line, with no defined apexes, both high speed

goal and a large distance to the goal are given.
This results in always high value of desired acceleration posed by the `Speed (max)` equation.

— The target accelerator position and resultant forces are transposed onto the longitudinal velocity & result in Slip Angle reduction.
— If the vehicle is approaching a corner with defined apex distance, a lower Bend Curvature is given.
This might result in the car sustaining higher velocities than expected.
— When distance to Apex goes below safe braking distance, given by Acceleration & Longitudinal Grip functions, emergency procedures are executed

5. SUMMARY

5.1. VALIDATION

The autonomous driving system has been executed with two different speed control algorithms, on identical testing scenario, consiting of a cone defined track with multiple cornering types, created according to FSG rules. Two algorithms are being tested:

— The default, single safe speed algorithm (on the right)
— New, speed managed algorithm (on the left)

we can see significant improvements on straight sections of the track, with instabilities in braking zones and more careful behaviour in corners.

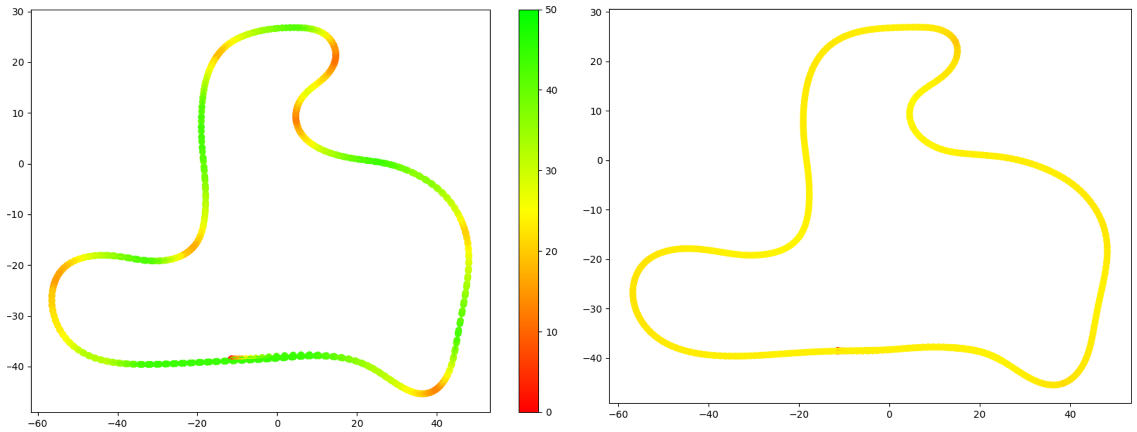


Fig. 5.1. SpeedGraph of Managed (left) and Set (right) speed algorithm

Shape comes from reading GPS positioning data, while the color represents speed of vehicles rear wheels in `km/h`.

We can observe the difference our new algorithm brings, by placing checkpoints in crucial spots on the track & recording the time at which we arrive at each of them.

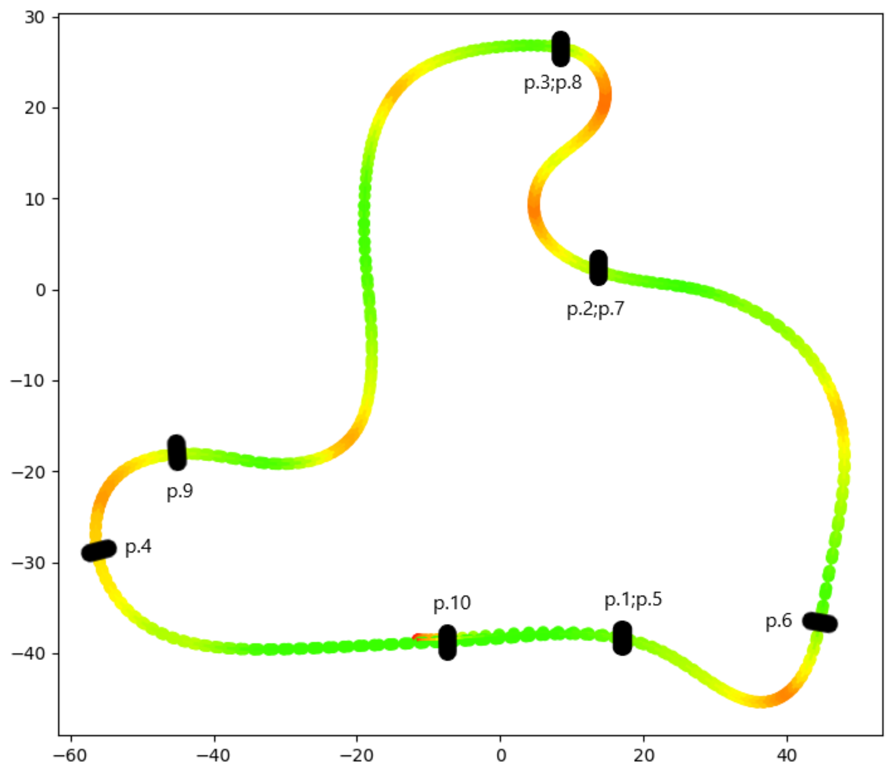


Fig. 5.2. Map of timing checkpoints

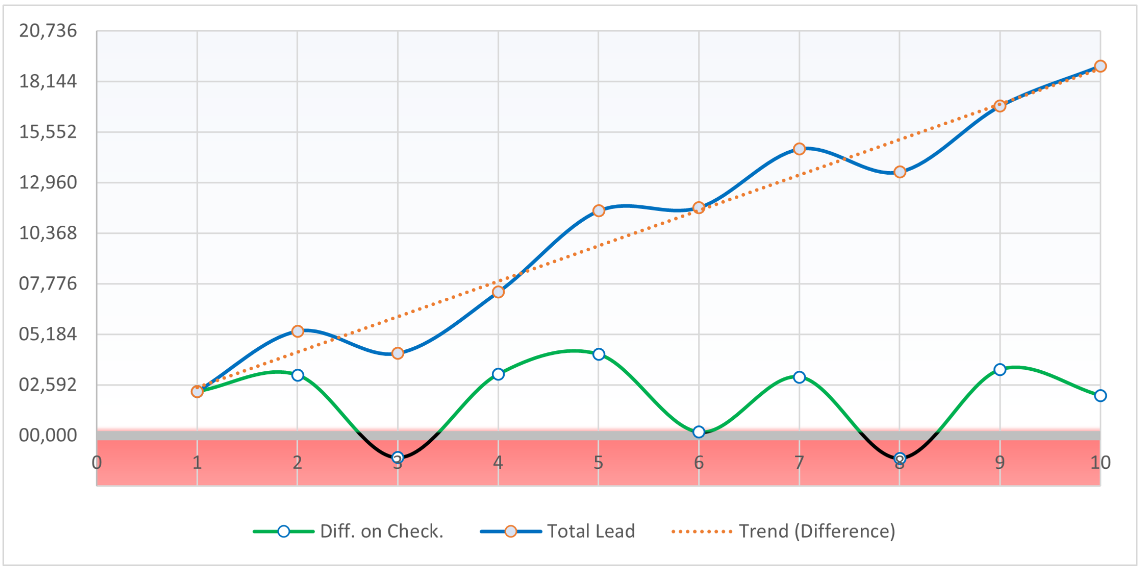


Fig. 5.3. Time difference graph

The green line represents time difference between following checkpoints, starting from starting line, at the intersection of graph axis, to checkpoint 10 at the end of the graph. The blue line represents difference in arrival at the Nth checkpoint, measured from starting point and a yellow trend line of time reduction

We can also compare readings from accelerometers placed around the front and rear axles. By comparing the reach of maximum forces on the rear and front axle, we can identify, that the set speed algorithm is abusing the front axle in tighter corners, with rear axle not seeing much action.

While the variable speed algorithm not only makes much better use of straights, but also adds lateral potential of the rear axle into cornering. The increased use of rear axles lateral grip not only improves performance of the rear axle, but also translates into front axles increased potential for higher lateral acceleration.

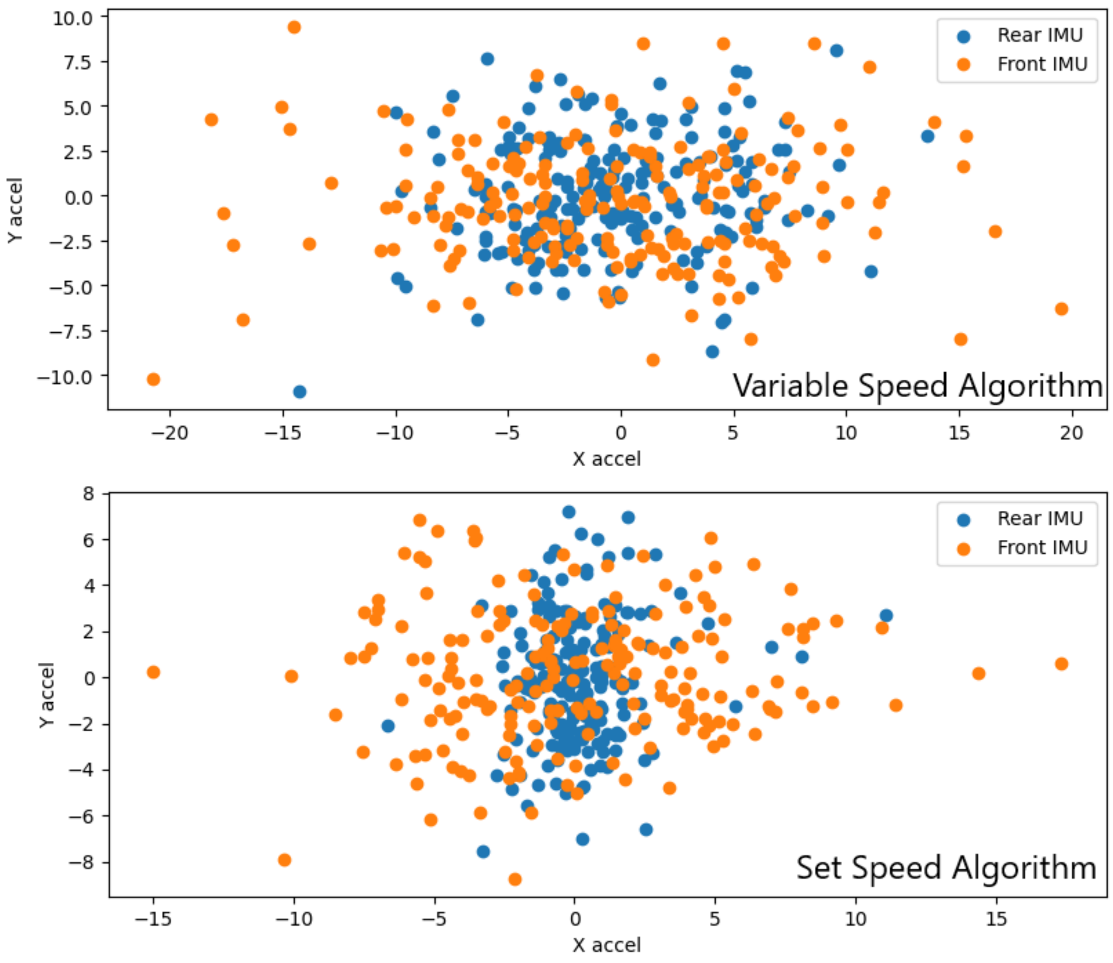


Fig. 5.4. AccelerationGraph of Managed (Top) and Set (Bottom) speed algorithms

5.2. CLOSING WORDS

This thesis closes with mark of successful completion of key objectives aimed at improving the testing processes for autonomous passenger cars & development of its components. Work containing creation of a testing scenario in BeamNG.tech simulation software, establishing an interface between the simulation suite and the autonomous operating system, adaptation of a path-following algorithm, and research of optimisation possibilities for a path planning algorithm.

The implemented testing scenario, demonstrated that use of complex simulated environments in evaluation of robotic systems, under diverse and challenging conditions, concludes with multitude of useful datasets. This approach enables further identification and resolution of design issues and performance deficits.

Successful interfacing between the simulation suite and the autonomous operating system has been successful. The simulation provides correct, real-time representation of scenarios imitating real world tasks. Adaptation of the path-following algorithm allowed the system to effectively navigate the simulated environment, providing vital insights into the systems performance and its pitfalls.

In the end, the study has identified & explored potential for optimising the path planning algorithm. These ideations are to be studied, specified in detail & implemented in further exploration on the subject.

Completion of these objectives has resulted in improvements in development & testing processes of the Formula Student car, proving authors abilities in working with autonomously operating, robotic systems.

BIBLIOGRAPHY

[1] BeamNG.gmbh, *BeamNG.Tech Main Website & Introduction*, `https://beamng.tech/`. 2023.

[2] Formula Student Germany Association, *FSG:Concept*, `https://www.formulastudent.de/about/concept?date=10.11.2023`. 2023.

[3] Kiran Kone, *Lateral and longitudinal control of an autonomous racing vehicle*, `https://webthesis.biblio.polito.it/11982/1/tesi.pdf`. 2019.

[4] Project Management Institute, *A guide to the project management body of knowledge (PMBOK guide)*, Sixth Edition wyd., PMBOK guide (Project Management Institute, Pennsylvania, USA, 2017).

[5] Racing Team of Wrocław University of Science & Technology, *PWrRT - Race Car brochure*, `https://web.archive.org/web/20230521144617/https://racing-pwr.pl/en/race-car/`. 2023.

[6] Wikipedia contributors, *Robot — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Robot&oldid=1192261043`. 2023. [Online; accessed 21-December-2023].

[7] Yastremsky, M., *Rapidly-exploring Random Tree Path Planner*, `https://github.com/MaxMagazin/ma_rrt_path_plan`.

LIST OF FIGURES

1.1 Exemplary system design 8
1.2 Autonomous System indicator list with colors for each *status*. 11
1.3 Autonomous System *status flow diagram* 12

2.1 Graphics of a popular robotics environment 17
2.2 Graphics of a state-of-the-art simulation solution 17

3.1 Interfacing architecture 27
3.2 Network & data exchange design 28

4.1 Delaunay Triangulation 34
4.2 FlowChart of simplified acceleration controller 35
4.3 FlowChart of advanced speed controller 36

5.1 SpeedGraph of Managed (left) and Set (right) speed algorithm 39
5.2 Map of timing checkpoints 40
5.3 Time difference graph 40
5.4 AccelerationGraph of Managed (Top) and Set (Bottom) speed algorithms 41

ACRONYMS

AMU Autonomy Main Unit. , 13, 19, 20, 28, 35, 36
API Application Programming Interface. 21, 25, 26, 30
ASAP As Soon As Possible. 34

EBS Emergency Braking System. 12

LUT Lookup Table. 36

PID Proportional–Integral–Derivative Controller. 35

SLAM Simultaneous Localisation & Mapping. 34

VPU Vehicle Performance Unit. 19, 20, 35, 36

GLOSSARY

Acceleration (competition category) - a straight line test, driving from standstill to an N meters away drive through point. 12

Autocross (competition category) an endurance competition, where we drive multiple laps, as fast as possible, around a track mapped during Trackdrive. 12

Autonomy Main Unit A computer of high computational power, functioning as the brain of autonomous driving system. 19, 35, 45

Fan Car a car which includes a ground suction system, in all current depictions created with help of large & loud fans. . 11

Formula Student a competition conceptualized by SAE International, with a theme, where a fictional manufacturing company contracts a student design team to develop a small Formula-style race car. The created prototype race car is to be evaluated for its potential as a production item in both on track and design, management and presentation aspects.. 3, 11

Marshal track marshals wave the racing flags and assist crashed or broken-down vehicles and their drivers, pit marshals watch over the procedures in the pits & carrying out safety testing, fire marshals are responsible for handling fire events on track or in the pit.. 11

Monocoque a chassis made of composite material.. 11

Nvidia Jetson A small packaged, integrated compute platform. Used edition sports an 8 core ARM CPU, a powerful GPU and dedicated “Jetpack” software package, for interfacing with compute units present on-board. 13, 28

Open Wheel The wheel/tire assembly is unobstructed when viewed from the side. & no parts altering aerodynamic behaviour are placed directly before or after the wheel. With details presented in rule books.. 11

Pitot Tube A pitot tube measures fluid flow velocity. It is widely used to determine the airspeed of aircraft; the water speed of boats; and the flow velocity of liquids, air, and gases in industry. A basic pitot tube consists of a tube pointing directly into the fluid flow. As this tube contains fluid, a pressure can be measured; the moving fluid is brought to rest as there is no outlet to allow flow to continue. This pressure is the stagnation pressure of the fluid, also known as total pressure or (particularly in aviation) the "pitot pressure".. 12

Skidpad (competition category) a turning ability test, driving from standstill around tight loops, to a standing finish. 12

Slip Angle The difference between set steering angle, imposed by the direction of wheel rim and the angle of tire surface in respect to the longitudinal of wheel displacement vector. This variable tells us about the actual change in movement angle.. 37, 38

Trackdrive (competition category) a mapping & unknown terrain driving test, where we drive a loop around a complex track, without prior knowledge of its configuration. 2, 12, 34, 46

Unit Testing Unit testing is a software testing method, by which individual units of source code (sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures) are tested to determine whether they are fit for use. 17

Vehicle Performance Unit A computer responsible for optimal power distribution & grip on tires, through torque vectoring & traction control.. 19, 35, 45