Universidade de Aveiro
2022

Sara Costa Pombinho

Arquitetura integrada de software no ATLASCAR2
Integrated software architecture in ATLASCAR2

Universidade de Aveiro
2022

Sara Costa Pombinho

Arquitetura integrada de software no ATLASCAR2
Integrated software architecture in ATLASCAR2

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestrado em Engenharia de Automação Industrial, realizada sob orientação científica de Miguel Armando Riem de Oliveira, Professor Auxiliar do Departamento de Engenharia Mecânica da Universidade de Aveiro e de Vítor Manuel Ferreira dos Santos, Professor Associado c/ Agregação do Departamento de Engenharia Mecânica da Universidade de Aveiro.

Agradecimentos / Acknowledgements

Agradeço em especial ao professor Miguel Riem e ao professor Vítor Santos pelo seu apoio e disponibilidade mesmo quando o tempo era escasso. A sua orientação foi fundamental para a realização desta dissertação.
Agradeço aos meus pais pelo seu apoio incondicional nestes 5 anos de Universidade, que fizeram tudo para que nada me faltasse. Às minhas irmãs, pelos desabafos e pelas conversas.
À Lara por estar presente em todas as fases da minha vida.
À Mané e à Beatriz que me apoiaram nestes 5 anos de Universidade.
Ao Rafael pela motivação e companhia demonstrada em todos os momentos.
Por fim, agradeço ao pessoal do LAR, em especial ao Manuel pela entreajuda e troca de ideias que contribuiu para esta dissertação e ao Engenheiro Rui Heitor que se mostrou sempre disponível para me ajudar quando precisei.

Palavras-chave

Odometria; Arquitetura ROS; Atlascar2; Electronic Control Units; CAN; Modelo Ackermann;

Resumo

A organização do sistema computacional de um veículo autónomo é crucial para uma melhor implementação de algoritmos na área da navegação autónoma. O primeiro objetivo desta dissertação consiste na organização e atualização da arquitetura de software do Atlascar2. Esta organização cobre a atualização e adição dos ficheiros de lançamento dos sensores, atualização de "drivers", melhoramento da unidade de processamento e atualização da infraestrutura de comunicação.

O segundo objetivo principal é o desenvolvimento de uma solução de odometria, para ser integrada na nova arquitetura de software do Atlascar2 que utiliza o barramento CAN do veículo como um meio de transmissão e receção dos dados necessários, como o ângulo das rodas do carro e a sua velocidade. Para a aquisição da velocidade, um codificador incremental foi instalado na roda esquerda traseira do Atlascar2, enquanto os valores do ângulo são obtidos pela rede do CAN.

Os resultados mostram melhorias na arquitetura de software e no desempenho obtido pelo sistema de odometria, simulado e real. Os resultados reais indicam a necessidade de ser feita a calibração da odometria, que quando elaborada num trabalho futuro, trará melhorias ao sistema de navegação do Atlascar2.

Keywords

Odometry; ROS architecture; Atlascar2; Electronic Control Units; CAN; Ackermann model

Abstract

The organization of an autonomous vehicle computational system is crucial for a better implementation of algorithms in the area of autonomous navigation. The first objective of this dissertation consists of organizing and updating the Atlascar2 software architecture. This organization covers updating and adding the sensor's launch files, updating drivers, improving the processing unit and updating the communication infrastructure.
Another objective is the development of an odometry solution, to be integrated into the new Atlascar2 software architecture. It used the vehicle's CAN bus as a means of transmitting and receiving the necessary data, such as the angle of the car's wheels and its speed. To measure the speed, an incremental encoder was installed on the Atlascar2 rear left wheel, while the angle values are provided by the CAN.
The results show improvements in the software architecture and in the performance obtained by the odometry system, in simulation and real. In real mode, the results indicate the need to calibrate the odometry, which, when elaborated in a future work, will bring improvements to the Atlascar2 navigation system.

Acronyms

AD Autonomous Driving. 1, 2, 3, 6, 7, 8, 14, 18, 63

ADAS Advanced Driver Assistance Systems. 1, 2, 7, 63

AV Autonomous Vehicles. 5, 7

CAN controller area network. 23

ECU Eletronic Control Unit. 23, 43

GNSS Global Navigation Satellite System. 26

GPS Global Positioning System. 14

HMI humanmachine interface. 8, 12

iCab intelligent Campus AutomoBile. 13

IMU Inertial Measurement Unit. 15, 26

INS Inertial Navigation System. 15

LiDAR Light Detection And Ranging. ix, 8, 11, 16, 17, 18, 23, 24, 26, 56

MEMS micro-electro-mechanical system. 15

OS Operating System. 35

ROS Robot Operating System. iii, ix, 4, 29, 30

SAE Society of Automotive Engineers. 1

UAV Unmanned Aerial Vehicle. 16

URDF Unified Robotics Description Format. 32, 37

Contents

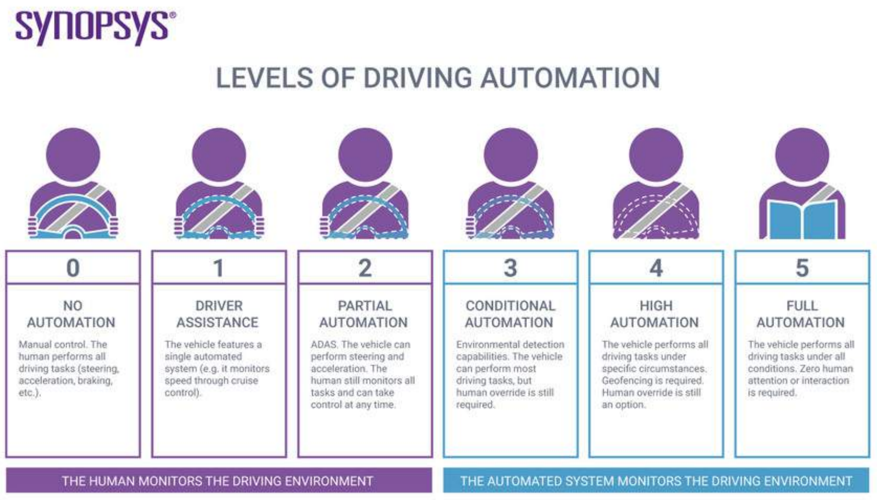
List of Tables

List of Figures

Chapter 1

Introduction

"We are at the dawn of the future of autonomous driving" [1]. With each passing day, the studies and technological advances in the area of Autonomous Driving (AD) and Advanced Driver Assistance Systems (ADAS) get one step ahead to save time and give comfort and safety to the drivers. According to the Society of Automotive Engineers (SAE), there are six levels of driving automation ranging from 0 (fully manual) to 5 (fully autonomous) [2].


SYNOPSYS
LEVELS OF DRIVING AUTOMATION
THE HUMAN MONITORS THE DRIVING ENVIRONMENT
THE AUTOMATED SYSTEM MONITORS THE DRIVING ENVIRONMENT

0	1	2	3	4	5
NO AUTOMATION	DRIVER ASSISTANCE	PARTIAL AUTOMATION	CONDITIONAL AUTOMATION	HIGH AUTOMATION	FULL AUTOMATION
Manual control. The human performs all driving tasks (steering, acceleration, braking, etc.).	The vehicle features a single automated system (e.g. it monitors speed through cruise control).	ADAS. The vehicle can perform steering and acceleration. The human still monitors all tasks and can take control at any time.	Environmental detection capabilities. The vehicle can perform most driving tasks, but human override is still required.	The vehicle performs all driving tasks under specific circumstances. Geofencing is required. Human override is still an option.	The vehicle performs all driving tasks under all conditions. Zero human attention or interaction is required.

Figure 1.1: Levels of driving automation [3].

To step up in the hierarchy of AD, more complex systems are created, which leads to more data exchange between the various modules of an autonomous car. Although an autonomous vehicle with a higher autonomy level has a more complex system, even at level 1, they require sophisticated algorithms capable of guaranteeing safety at every moment. Furthermore, the non-organization of that data can lead to errors, which may be harmless or be fatal on the road. To avoid this, software architecture was born. A good architecture is crucial to keep an efficient and versatile system in which new developers can easily manage and facilitate the addition of new modules when needed.

This thesis has two main subjects. The first is the creation of a software architecture that implements an efficient and reliable system, using ROS as the main framework of the vehicle. The second is the implementation of an odometry node to test the newly implemented software architecture and calculate the relative position of the vehicle.

1.1 ATLAS project

The Group of Automation and Robotics created the ATLAS project at the Department of Mechanical Engineering of the University of Aveiro, Portugal. The project's main focus was to develop advanced sensing and active systems and to implement them in automobiles and platforms alike. This dissertation is one of the many projects/dissertations in the context of the ATLAS project [4]. The ATLAS project started to develop mobile autonomous robots (Figure 1.2), which participated in multiple robotics competitions and won various awards.

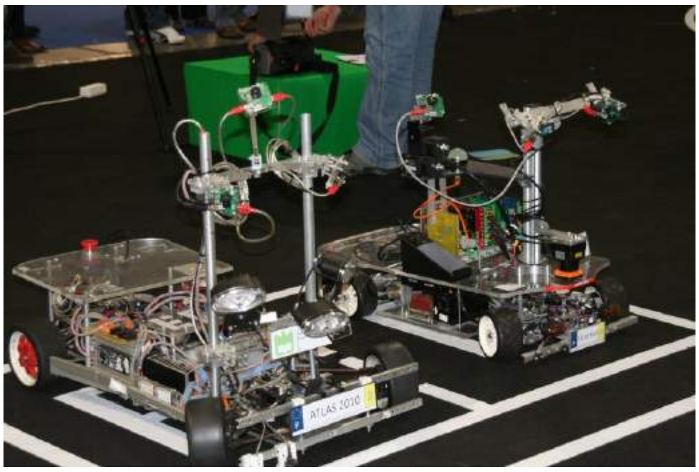


Figure 1.2: Atlas2010 and AtlasMV3.

With newfound success in competitions, the automation group expanded the work further and created the AtlasCar1. The Atlascar1 was a Ford Escort equipped with multiple sensors that enabled the vehicle to perceive the road around it [4]. This car was a prototype for research on Advanced Driver Assistance Systems.

After several years of studies, algorithms, and hardware, the Atlascar1 had fallen behind in technology and was no longer enough for the project's requirements, so the Atlascar2 was developed. The Atlascar2 (fig 1.4) is an electric Mitsubishi i-MiEV vehicle from 2015. Its most significant advantage compared to the Ford Escort is the nonexistence of gear changing, which facilitates the automation control, and the fact that the car is electric also facilitates the direct battery access for the sensors. This is the vehicle used in this dissertation.

1.2 Problem Description

With the years passing by, the Autonomous Driving projects kept expanding the scope with increased complexity and a higher level of automation. With that complexity came more sensors that lead to more data exchange, which increases the possibility of errors. The problem with this increase is that it also increases the probability of accidents we

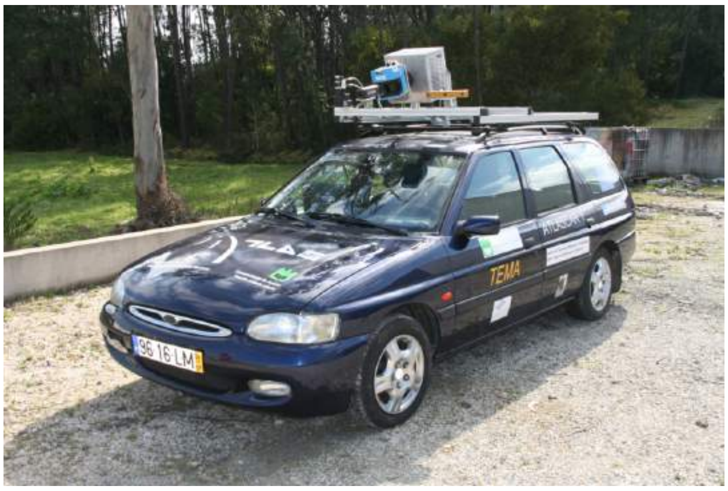


Figure 1.3: Atlascar1.

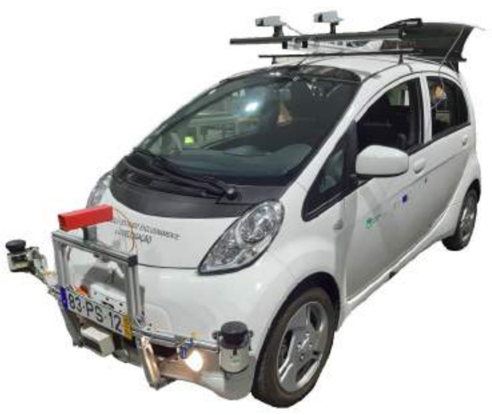


Figure 1.4: Atlascar2.

want to avoid at all costs. In that line, the importance of software architecture is to give reliability to the vehicle and organize the system to ensure the safety of the drivers and other agents. On the other hand, with the growing development of AD, certain features can be challenging but need to be implemented to develop a fully functional autonomous vehicle, such as self-localization. Therefore, the proposed objective to acquire the localization of the moving vehicle is to apply an odometry algorithm with the new software architecture already developed to facilitate its implementation.

1.3 Objectives

With the addition of multiple projects along the years, the Atlascar2 data started to become difficult to maintain organized and robust due to the excess of information, which causes multiple errors when an inexperienced user starts working with the vehicle, losing weeks, even months in an attempt to understand the state of the car. This problem occurs since there is a lack of organization and documentation about the system and its software.

After developing a software architecture, the next step is to create an odometry algorithm, which can test the improvements made in the architecture in terms of quality attributes and acquire the vehicle's relative position.

In summary, this thesis aims at the following objectives:

- Update of the Atlascar2 software architecture.
- Development, update, and integration of the AtlasCar2 software packages.
- Development and testing a node to compute the vehicle odometry.

1.4 Document structure

This document has seven parts:

- Chapter 1: Presents the introduction, which has the purpose of contextualizing the problem and background and establishing goals to achieve at the end of this thesis.
- Chapter 2: Presents the state of the art, the various works in this area in software architecture, and a study of the odometry solutions.
- Chapter 3: Explains the used infrastructure, both software and hardware, describing the various sensors in the Atlascar2 and the Robot Operating System (ROS) and its packages and other software.
- Chapter 4: Describes the methodology used to change the vehicle software architecture, the requirements needed, and the nodes.
- Chapter 5: Describes the methodology used to implement the odometry node.
- Chapter 6: Presents the qualitative and quantitative results in terms of software and odometry.
- Chapter 7: Presents the conclusion of the thesis, by evaluating the results and discussing the future work that may improve the implemented subjects.

Chapter 2

State of the Art

As stated previously, this work focuses on two subjects. The first is the software architecture. To create an appropriate architecture for the Atlascar2, there is the need to understand what is a software architecture for an autonomous vehicle and the various works in this area that tend to show what can be used to improve the vehicles' architecture.

The second subject is to implement an odometry node. This theme is studied by many authors worldwide, and there are many solutions and different methodologies to overcome the problem. Some of them are described in this chapter.

2.1 Software architecture

Although describing what is a software architecture may appear an easy task, it can be very convoluted due to its two conflicting descriptions:

Functional architecture: According to Behere et al. (2016) [5] and Serban et al. (2018) [6], functional architecture can be defined using the ISO 26262 functional safety standards definition of a functional concept, which is the specification of the intended functions and their interactions necessary to achieve the desired behavior [7]. Thus, functional architecture refers to "the logical decomposition of the system into components and sub-components and the data-flows between them" [5]. Figure 2.1 shows an example of a functional architecture in AV.

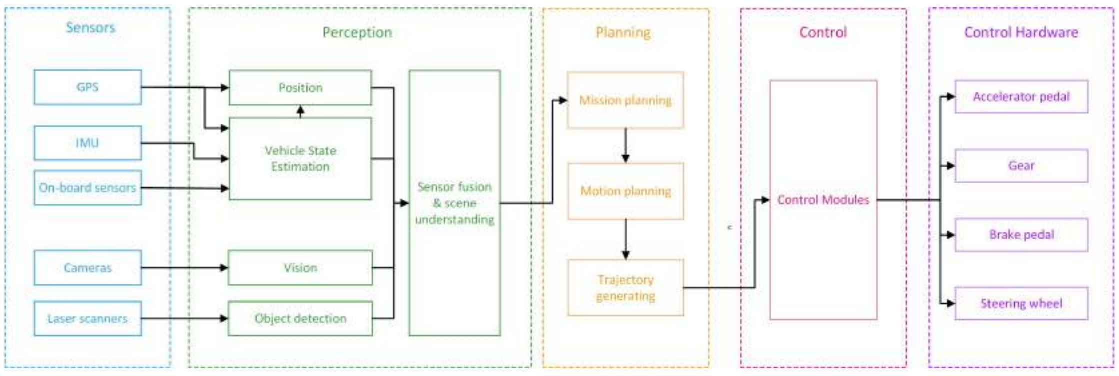


Figure 2.1: Functional architecture [8].

Software architecture: Software architecture is the organization of a system. This organization usually includes all components in AD are I/O devices, operating systems, middleware, and application modules. Figure 2.2 gives an example of a software architecture, which describes the same AV as shown in the figure above, depicting its functional architecture.

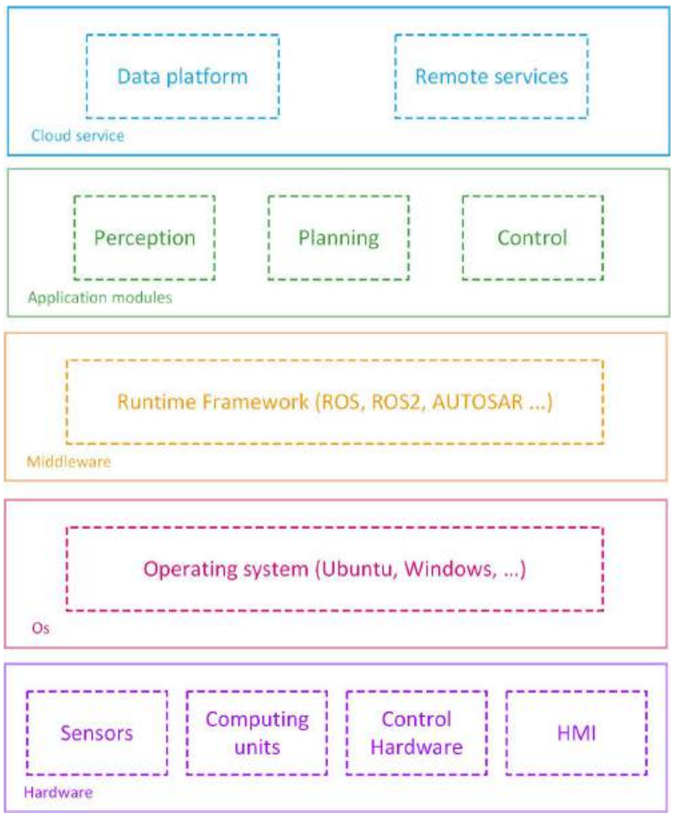


Figure 2.2: Software architecture [8].

Despite different names and descriptions, both are considered software architectures. The first one presents a detailed high-level architecture, while the second shows a low-level general approach to the architecture.

In terms of components in the software architecture, Pendletan et al. (2017) [9] and Behere et al. (2016) [5] state three primary tasks in autonomous vehicles: Perception, planning, and control. These three high-level functional components can be described as follows:

Perception: Perception refers to collecting information from the sensors and "understanding" the relevant data. The sub-modules are usually localization, which is responsible for determining the position of the ego-vehicle in the environment, and sensor fusion, which uses the multiple data gathered to construct a hypothesis about the state of the environment and can also perform object tracking and association. There also exist others, such as the world model, which holds the state of the external environment as perceived by the ego-vehicle, and semantic understanding, which implements the detection of ground planes, roads' geometries, and drivable areas as well as predict future behavior from objects with references to physical models.

Planning: Planning refers to the decision-making to achieve the intended goal. In Autonomous Vehicles that is typically the realization of a trajectory from a start location to a goal location while avoiding obstacles and optimizing complex heuristics. This component usually has sub-modules such as mission planning, behavioral planning, and motion planning, which are responsible for generating the best trajectory, following the stipulated road rules, and deciding a sequence of actions to avoid obstacles.

Control: Control is the process of converting intentions into actions, receiving the movements generated by the planning component, and providing the necessary inputs to the hardware to generate those desired motions. This functional component can also stabilize the vehicle, where unreasonable motion requests may be rejected or adapted to stay within the car's physical capabilities and safety envelope.

However, as described in section 2.1.1, many projects in the AD area change the formula to a more robust approach to increase the vehicle's safety.

When developing a software architecture, there are attributes to check the quality of the new system. The book *The Future of Software Quality Assurance* [10] has a list of topics needing consideration when testing an autonomous vehicle architecture:

Functional Suitability: "It must be checked if the functional properties of the system are implemented "complete," "correct," and "appropriate."" To notice if the higher and lower-level components are doing their functions properly.

Reliability: "The ability of the system to maintain its once achieved quality level under certain conditions over a fixed period of time." In the context of AD, the vehicle can repeat the same behavior multiple times in a row without errors.

Performance efficiency: "The time behavior of the system and its components and the consumption of resources must be checked."

Compatibility: "Interoperability between components of the system itself (sensors, controls, actuators) as well as compatibility with external systems." If the actuators are correctly controlled, the communication protocols are compatible with other components and external systems.

Security: "To check how resistant the system is against unwanted access or criminal attack on data of the system or its users or on the entire system itself."

Maintainability: "If software and hardware are modular and the respective components are reusable and easily changeable."

2.1.1 Related Work

Work in the area of AD can date back to the 1920s. This thesis focuses on some of the more essential and prominent work of the ADAS vehicles to acquire the needed information to improve the Atlascar2.

DARPA Grand Challenge

The DARPA Grand Challenge was a big step in the area of Autonomous Driving. It was a competition from 2004 to 2007 to develop an autonomous vehicle that could drive autonomously on a given route. Despite the difficulty of the challenge, in 2005 and 2007, various vehicles successfully completed the challenge.

By observing their work in software architecture, we can start with the winning vehicle of 2005, Stanley [11]. Stanford racing team based Stanley on a 2004 Volkswagen Touareg R5 TDI (Figure 2.3) and his architecture from a three-layer architecture [12].

Figure 2.3: Stanley the 2005 winning vehicle of the DARPA challenge.

This architecture was divided into six main functional groups: sensor interface, perception, planning/control, vehicle interface, user interface, and global services (Figure 2.4).

The sensor interface was in charge of receiving and time stamping all of the sensor data and containing the database of the course coordinates. Then, the perception layer mapped the sensor data into internal models, such as the ego-vehicle pose estimation, which determined the vehicles coordinates, orientation, and velocities. Next, it built the environment using three mapper modules from various sensors, cameras, radars, and LiDARs. Lastly, it required a surface assessment module to extract the parameters from the current road to determine safe vehicle speed. Knowing the surrounding environment, the planning/control module was capable of planning the vehicle's trajectory in steering- and velocity-space. This trajectory was passed to two closed-loop controllers, one for the steering control and one for brake and throttle control. Both these controllers sent the commands to the actuators that faithfully execute the trajectory created by the planner.

The vehicle interface contains the interfaces to the vehicles brakes, throttle, and steering wheel and an interface to the vehicle's server, which regulated the physical power of many of the system components.

The user interface comprised a remote E-stop button and an HMI to start the software. To communicate with all the modules, the global services are provided through CMUs Inter-Process Communication (IPC) toolkit and keeping track of the health of all systems and restarting them when needed. These modules can be observed in more detail in section 1.3.2 of Stanley's article [11].

This robust architecture gave the system quality attributes such as reliability, where

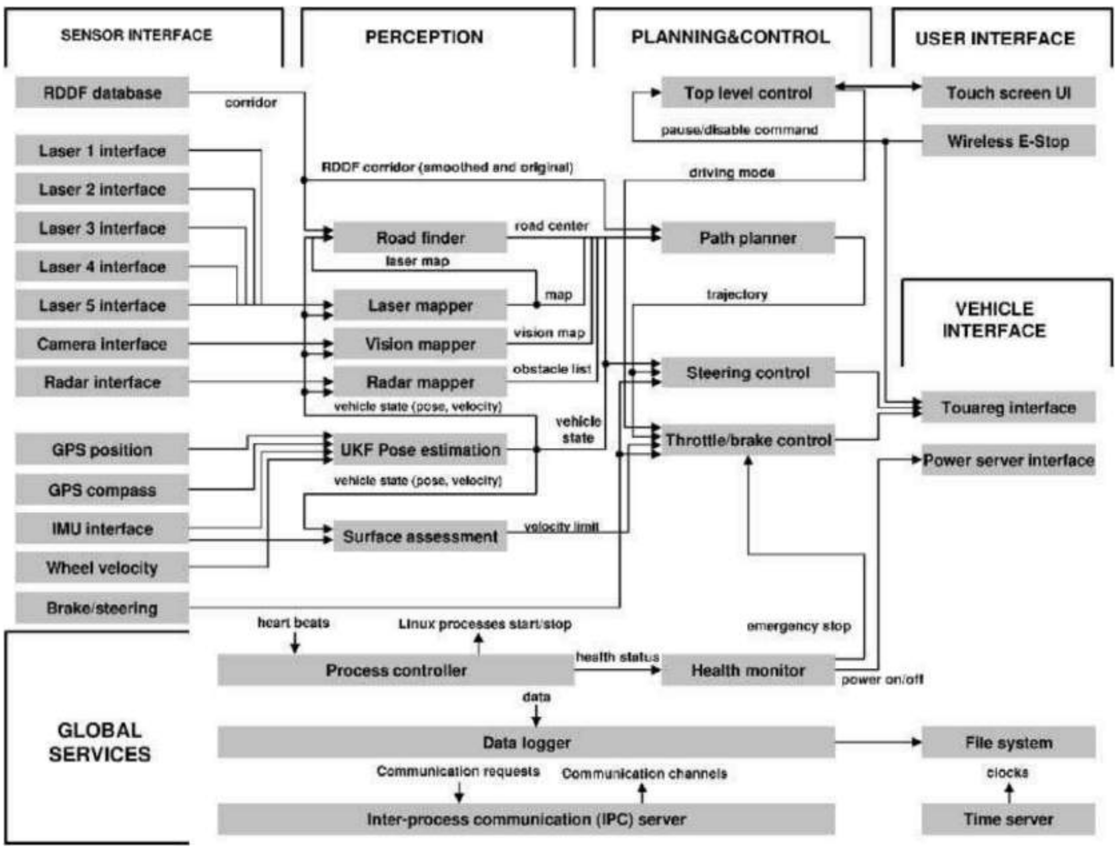


Figure 2.4: Stanley functional architecture [11].

the global services layer monitored the health of individual software and hardware components and restarted the components when a failure was observed, performance efficiency because lacking a master process reduced the risk of deadlocks and undesired processing delays, maintainability since "the developer can easily run just a sub-system of the software, and effortlessly migrate modules across different processors", and compatibility for the reason that "nearly all inter-process communication is implemented through publish-subscribe mechanisms" [11] through the IPC toolkit.

Its successor Junior had an almost identical approach and finished second in the 2007 DARPA urban challenge. Like most of the following vehicles presented in this thesis, the system lacks security since it is a research vehicle. That type of requirement would be time consumer that provided slight improvement to the vehicle itself [13] and its functional components.

Following the victory of Stanley in 2007, another vehicle took the podium, Boss [14]. Boss was developed by the Tartan Racing Team and based on a 2007 Chevrolet Tahoe (Figure 2.5).

Boss had four main functional components: Perception, Mission Planning, Behavioral Executive, and Motion Planning.

Like in most architectures, the Perception module was responsible for recreating a model of the world to the behavioral and motion planning components. That model in-

Figure 2.5: Boss the 2007 winning vehicle of the DARPA challenge.

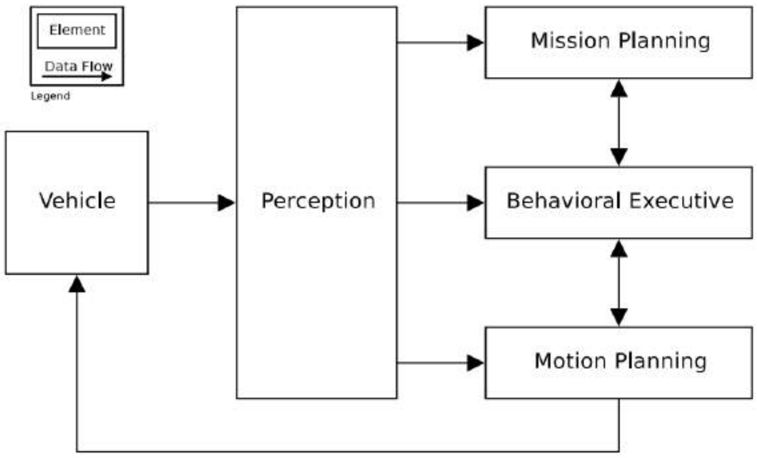


Figure 2.6: Boss software architecture [15].

cluded tracked and static objects, vehicle position estimation, and road detection. The mission planner used the course coordinates to create a graph that encodes the connectivity between the environment. This module provided the current waypoint to the goal, following the road rules and detecting blockages. Next, it was up to the behavioral module to execute the proposed policy: making lane-change and safety decisions respectively on roads, intersections, and yields, as well as recovering from abnormal situations. With that, the motion planning could execute the current motion goal issued from the behavioral module.

Specifically, there is not much data about the attributes of the software architecture of Boss. However, it had a remarkable property because its progress monitoring system was embedded inside its behavioral executive module, described in detail in section 3 [16], giving it reliability. The system was also compatible because the four main functional components in figure 2.6 "communicate via message-passing according to the anonymous publish-subscribe" [16] are maintainable since, like Stanley, it could just run a sub-system of the software.

More DARPA challenge vehicles are discussed in the section 2.3 in a more summarized way.

A1 Racecar

In 2012 a similar competition to the DARPA Grand Challenge took place in Korea. The autonomous vehicle A1 created by the Hyundai Motor Group won the 2012 Autonomous Vehicle Competition and completed all the missions, such as handling moving vehicles and pedestrians and understanding traffic lights, among others [17, 18].

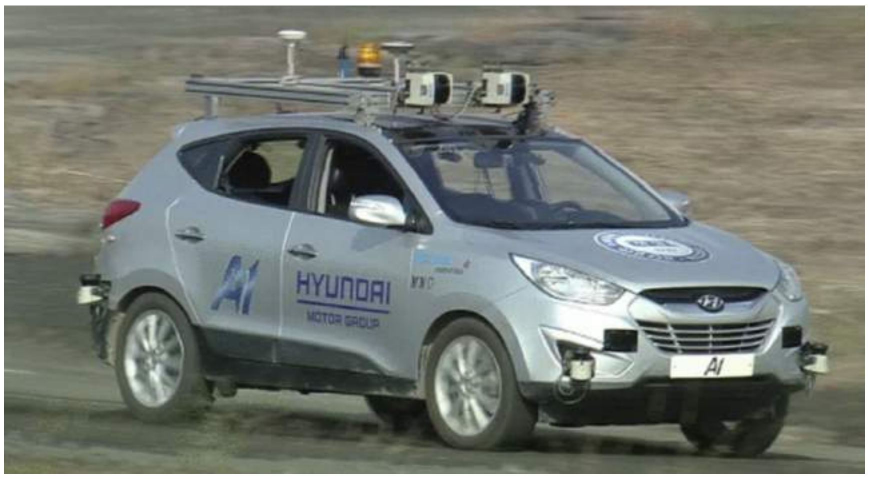


Figure 2.7: A1 vehicle [18]

Its functional architecture has five main components: Localization, Perception, Planning, Vehicle Control, and System management, since sensor fusion is not the primary function but a sub-module from perception.

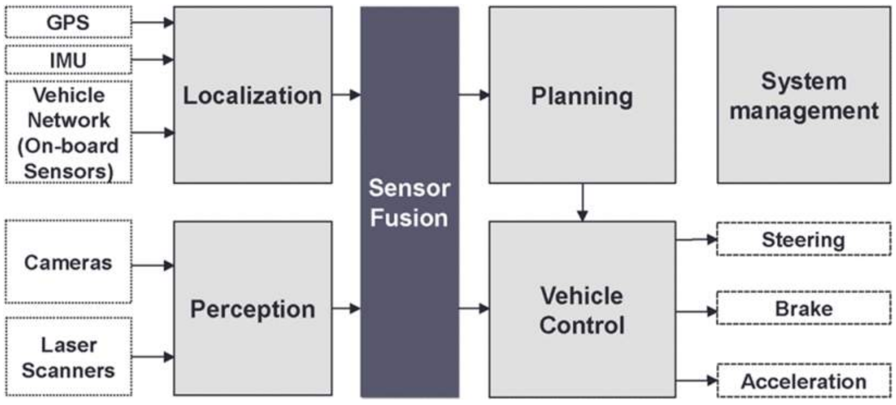


Figure 2.8: A1 functional architecture [18].

The localization component in most software architectures is considered a sub-module from perception. This component calculates the ego-vehicle coordinates, orientation, and velocity, which is later fused in the sensor fusion with the information on the surrounding environment and object detection and classification determined by the perception component using sensors such as cameras, radars, and LiDARs. The planning component determines the necessary maneuvers for the car to reach the main goal successfully. The A1 planning algorithm focused on behavioral reasoning and local motion planning. The behavioral reasoning implemented a rule-based decision process based on finite-state machines to follow the traffic regulations and accomplish various tasks (lane keeping, obeying traffic lights, and keeping under speed limits). Furthermore, to drive in various environments, the local motion planning was composed of two types of path planning algorithms. These types are road-map-based and free-form, which are used to drive in everyday situations, such as lane driving, and in complex unstructured areas, such as roads in construction, respectively.

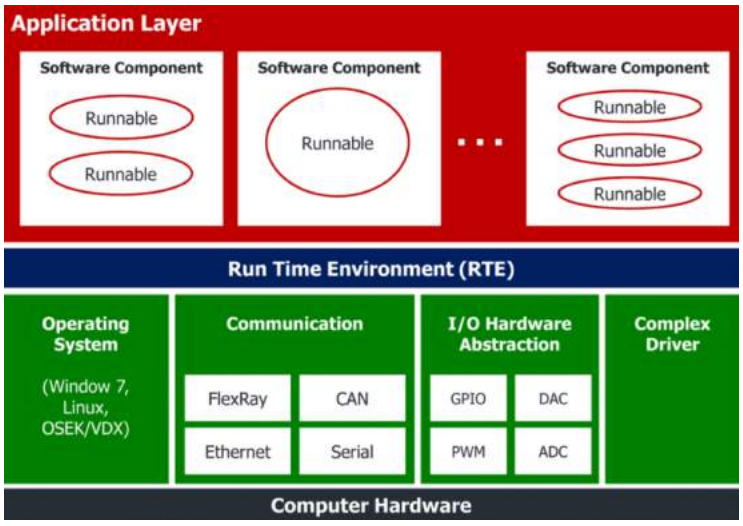


Figure 2.9: A1 software architecture [17].

The control module executes the trajectory planned by the planning component safely, controlling the vehicle to be robust in various environmental conditions such as snow, rain, and wind, and dealing with the physical limitations and vehicle dynamics [19]. To resolve these problems, the control system of A1 is divided into two algorithms. A lateral control algorithm assumes that the vehicle moves along the Ackermann steering geometry, and a longitudinal control algorithm derives the target position of the acceleration and brake pedals from reference inputs [18]. Lastly, the system management component supervises the system to detect errors in the modules described. This module implemented the following functions: A HMI, a driving mode management, and fault management. These modules monitored the vehicle's health status and changed the car mode to autonomous or manual.

The A1 applied a layered architecture-based software platform to accomplish a robust system, which originated from AUTOSAR [20]. The software architecture has three layers: the application layer, the run-time environment (RTE), and the basic software layer. The application layer implements software components, and the basic software layer consists of the OS, communication modules, input and output (I/O) hardware abstraction module, and a complex driver. In the middle lies the RTE layer to remove the dependence of the software components from the hardware and networks [17]. This architecture improved the compatibility, maintainability, and reliability of the system.

Intelligent Campus AutomoBile iCab

The iCab is an autonomous vehicle with a ROS-based architecture made at the University Carlos III of Madrid, Spain. The used platform is an electric golf cart, model E-Z-GO (Figure 2.10) modified to fulfill the autonomous navigation, path planning, and cooperation objectives [21].

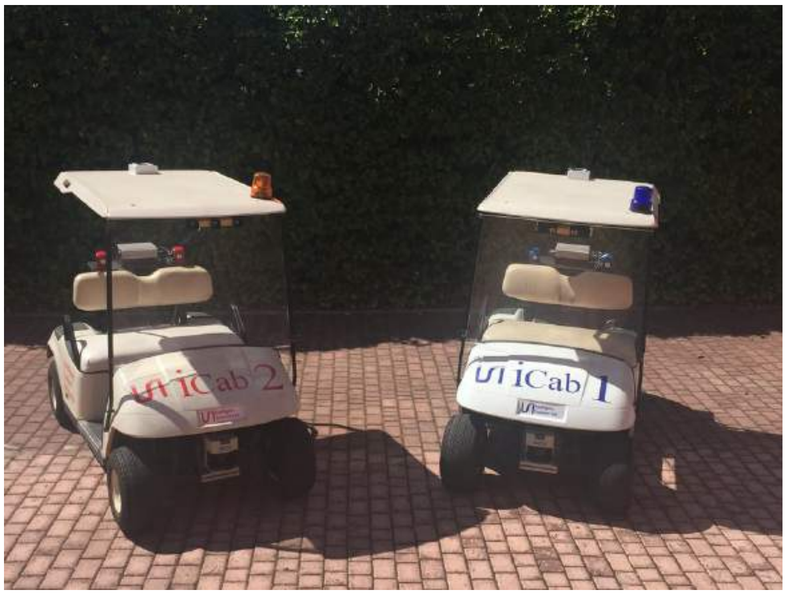


Figure 2.10: iCab 1 and iCab 2 platforms [21].

This architecture possesses various levels of complexity categorized into three layers [22]: deliberative, sequencing, and reactive skills.

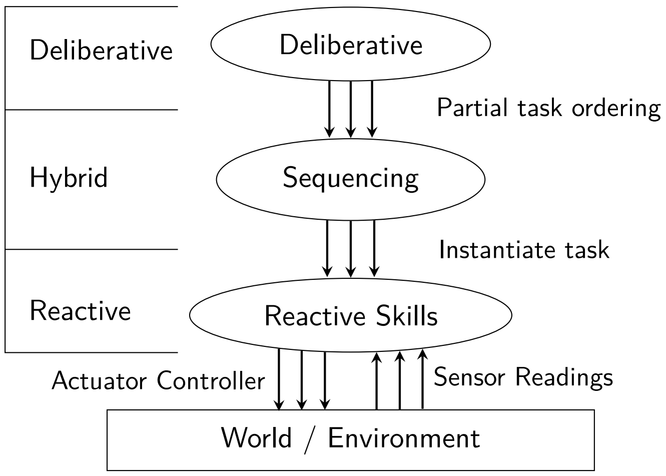


Figure 2.11: iCab three-tier architecture. Adapted from [22].

The deliberative layer manages path planning, navigation, and mapping sub-modules. The user defines the destination, and then the deliberative layer generates the output tasks for the sequencing layer to split the tasks, whose complexity resides in the accuracy of generating simple skills after splitting the mid-level tasks. These tasks are then conveyed to the reactive layer to generate movement outputs via ROS services and send them to the controller. These basic commands to move the vehicle are Move Forward, Move Backward, Turn Left, Turn Right, and Stop [22].

The advantage of this ROS-based architecture is that it gives to the vehicle system quality attributes, such as reliability, compatibility, and maintainability. Furthermore, this system enables inter-process communication in an independent and modular way and the capability to run multiple algorithms in parallel.

Other vehicles

There are numerous AD projects such as Waymo from Google, autopilot and Full Self Driving from Tesla, and Motional from Aptiv. However, these are commercial solutions, so the information regarding their technological advances is not available to the public. However, we know that all these projects have the quality attributes and requirements for a robust and safe software architecture since they need to be according to the strict ISO 26262 functional safety constraints [7].

2.1.2 Work developed at the University of Aveiro.

The work developed in this area at the University of Aveiro dates to the Atlascar2 predecessor, Atlascar1. The work in [23] shows that the Atlascar1 was based on CARMEN [24], which made it a modular architecture with various functional components, such as the other vehicles described previously. Furthermore, this architecture presented quality attributes such as maintainability since it divided every system into multiple modules to facilitate debugging and improve the robustness of the code, decreasing its complexity and compatibility considering the use of an IPC communication protocol to exchange data with the various components.

Different from the AtlasCar1, the software used in Atlascar2 is ROS, which is approached in section 3.2.1. In that line, it is essential to create a new solution that settles on the latest software.

2.2 Odometry

One of the most essential information for a robot is its self-localization in the environment. Although advanced Global Positioning System (GPS) can, at best, provide accurate positioning within a few centimeters, it is still not reliable enough for a core navigation system of autonomous platforms. In addition, various factors disturb the acquisition and tracking of GPS, such as the signal strength variation, depending on the place and environment conditions and [25, 26] multipath reception, where GPS signals arrive at a receiver from more than one satellite or via multiple reflective surfaces [27, 28].

Therefore, other methods are needed to estimate the self-localization of the autonomous system more reliably. One particular method is Odometry which uses local sensory data to determine the orientation and position of the platform relative to a given starting point. Odometry can be categorized into five types: Wheel, Radar, Inertial, Visual, and Laser (Figure 2.12), where each category is based on the type of sensor data used for the odometry [29].

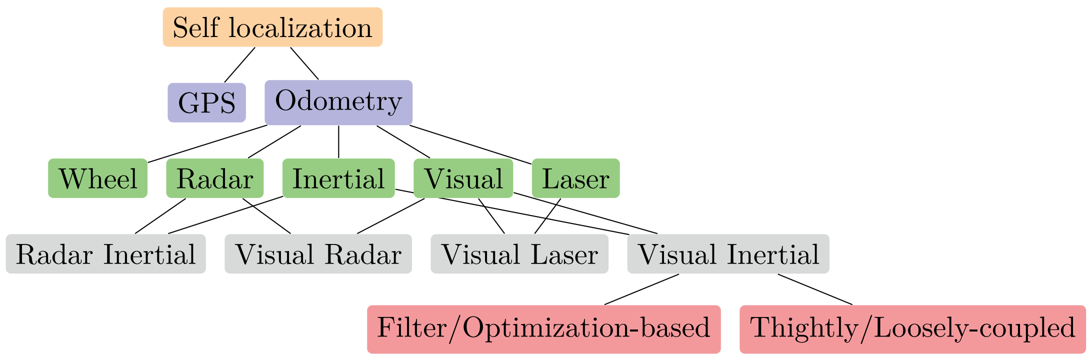


Figure 2.12: Types of Odometry. Adapted from [29].

2.2.1 Wheel Odometry

Wheel Odometry is the most straightforward technique of self-localization used in many robots. This method usually uses wheel encoders mounted on the back wheel of the robot to track the number of revolutions performed by each wheel. Then, the speed information is converted into travel distance through the wheel radius. Finally, it uses the position of the autonomous vehicle at the last obtained position to calculate the current one.

Although a very inexpensive technique, this method suffers from multiple disadvantages such as position drift, where the error in the measurements accumulates over time, poor performance in complex, uneven terrains and slippery surfaces due to wheel slippage, which makes this method not suitable for autonomous vehicles since it requires a reliable and precise localization system. Therefore, many researchers use this method combined with other procedures for a more accurate solution [30, 31] .

2.2.2 Inertial Odometry

Inertial Odometry, most commonly known as Inertial Navigation System (INS), is a localization method that uses the measurements provided by the IMU (Inertial Measurement Unit) sensor to continuously calculate by dead reckoning the position, orientation, altitude, and linear velocity of the moving vehicle without external references. An IMU sensor is a micro-electro-mechanical system (MEMS) device mainly composed of a 3-axis accelerometer and a 3-axis gyroscope and may also include a 3-axis magnetometer [32]. Although small in size, having low power consumption, and resolving some problems from the wheel odometry, these sensors still suffer from drifting issues due to constant measurement errors from the gyroscopes and accelerometers, which leads to an incremental error in the estimated velocity and position. Therefore, inertial odometry systems are inaccurate and unsuitable for an autonomous vehicle that requires localization for long periods of time.

2.2.3 Radar Odometry

Radar Odometry is a technique used to estimate the relative motion of a platform by determining the velocity, range, and angle of surrounding objects using radio waves.

The radar is a long-range active sensor immune to poor weather conditions and can operate in low-texture environments. However, it still suffers from disadvantages such as outliers and uneven terrain [33]. An outlier rejection scheme was used to improve the Radar Odometry solution to tackle these limitations. In addition, to overcome the problem of bumpy terrain, the radar measurements are fused with other sensors, such as a camera [34] and the IMU [35].

2.2.4 Laser Odometry

Laser odometry, or LiDAR odometry, is an approach for estimating the position and orientation of a platform by tracking laser speckle patterns reflected from surrounding objects [29]. This method uses the advantages of the LiDAR, such as being insensitive to ambient lighting and low-texture environments [36].

Although the LiDAR can be considered better than the radar, as it is capable of detecting small objects using a short wavelength and building an extract 3D monochromatic image of the surrounding objects, it also suffers from limitations concerning transparent objects and not being immune to poor weather conditions. Furthermore, LiDAR odometry also suffers from the amount of points received by the LiDAR, since it applies iterative optical matching among points of two sets, which requires fairly demanding computations, causing poor performance [37].

2.2.5 Visual Odometry

Visual odometry is a process used to determine the position and orientation of a platform by extracting key information from a sequence of images.

This type of odometry is much more precise than the wheel and inertial odometry. However, it still suffers from some drawbacks. Most of these problems are mainly related to computational complexity and image conditions. This last one is caused by low lighting, shadow, and low texture environments. Moreover, it also suffers from drifting issues caused by error accumulation as visual odometry is based on relative measurements such as wheel and inertial odometry.

Thus, researchers combine these methods to obtain more precise measurements and tackle the disadvantages of each one. Figure 2.12 indicates three types of visual odometry: Visual-Radar, Visual-Laser, and Visual-Inertial odometry.

Visual-Radar Odometry is one way to overcome the limitations in visual odometry in terms of environmental conditions such as rain, fog, and snow since the radar is immune to those issues. This type of method is used [34] to estimate the forward velocity of a Unmanned Aerial Vehicle (UAV).

Visual-Laser odometry overcomes the limitations of the LiDAR, such as motion distortion and non-prominent environments, and limitations of the visual odometry, such as drifting and low-texture environments.

Visual-Inertial odometry can overcome the limitations of visual and inertial odometry. This odometry can be categorized as loosely-coupled and tightly-coupled. The loosely-coupled [38] approach is the moderation of the computational load. However, the correlation between the internal measurements and vision data is disregarded, decreasing efficiency. On the other hand, the tightly-coupled [39] approach fuses the data from the IMUs and cameras in an early stage, which leads to more precise estimations, although it demands more computational power.

2.2.6 Work developed at the University of Aveiro.

The first work based on odometry in a real car at the University of Aveiro was implemented in the Atlascar2 predecessor, Atlascar1. In 2011, Tiago Rocha implemented a wheel odometry system using an incremental encoder on the back wheel of the Atlascar1 to measure its velocity [40]. This incremental encoder gave a new value of the car's velocity every 3.49 centimeters.

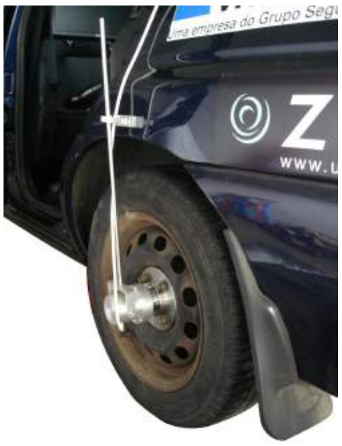


Figure 2.13: Velocity measurement system [40]

To obtain the angle of the Atlascar1, the steering wheel angle was measured with a potentiometer connected through a pulley system directly in the steering column. This was an invasive method and forced modifications on the structure and devices of the steering column (Figure 2.14).

In 2014, Ricardo Silva used visual odometry in his master's thesis to create a solution with Ackermann steering [41]. This method was tested using the ground materials fixated on a robot and the camera stationary on a tripod (Figure 2.15).

R. Silva proved the ability of this method using various types of movement such as circular, linear, diagonal, and different velocities. In conclusion, he stated that this method had a high error dealing with low velocities and a lack of precision in practical terms.

Lastly, in 2016, Jorge Almeida used Laser odometry in his Ph.D. thesis [42] to estimate the ego-motion of a vehicle using exclusively laser range data. This approach considered the local discrepancies between closely spaced laser scans to calculate the current vehicle velocity and steering angle. These measurements incorporated a non-linear model and successfully provided an accurate vehicle motion estimation. However, this algorithm using exclusively the LiDAR can fail because it is mainly dependent on the first guess and the lack of features in the range of the laser scanners.

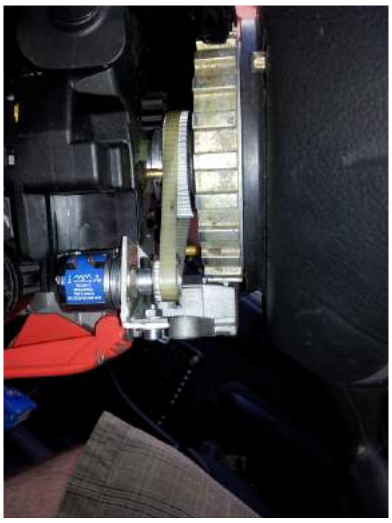


Figure 2.14: Steering wheel angle measuring system [41]

2.3 Summary

Two different subjects were addressed throughout this chapter: Software architecture and odometry. Thus, this section intends to compress the essential topics and discuss some comparisons among vehicles.

Starting with the discussion of the different vehicles, two tables were created: Tables 2.1 and 2.2, which show, respectively, the various sensors and software architecture from multiple vehicles.

Notably, excluding Tesla, all of these vehicles, commercial or not, have some type of LiDAR used in autonomous driving. For example, the commercial solutions by Waymo and Motional are both, according to SAE, a level 5 autonomous vehicle, and Tesla's autopilot and Full Self Driving are just level 2 (although it is being marketed as a fully autonomous vehicle). Furthermore, [36] it is referred that the LiDAR is necessary for level 4 and 5 autonomous vehicles.

In terms of software architecture, most of these vehicles approached the Autonomous Driving problem differently, with different communication protocols, functional components, frameworks, operating systems, and quality attributes.

These vehicles have distinct software architectures but can all function properly, proving that there does not exist only one good architecture and that each architecture is implemented depending on the attributes and necessities for the autonomous vehicle.

On the subject of odometry, the various types of odometry were summarized in 2.3:

The first metric, performance, is the capability to measure values in a strict time interval because autonomous vehicles need a strong timing constraint since a failed deadline can result in an accident. The second metric is power efficiency, which is the demand for computational resources for each method. The third metric is the accuracy of the method used to obtain the position and orientation when the system is active. The fourth metric is energy, which refers to the amount of electrical energy consumed by the sensors and processing units. The fifth metric is robustness against the lack of illumination and environmental conditions such as rain, fog, dust, and snow. The sixth and last metric is the dimensionality of the calculated pose, which can be either 2D or 3D.

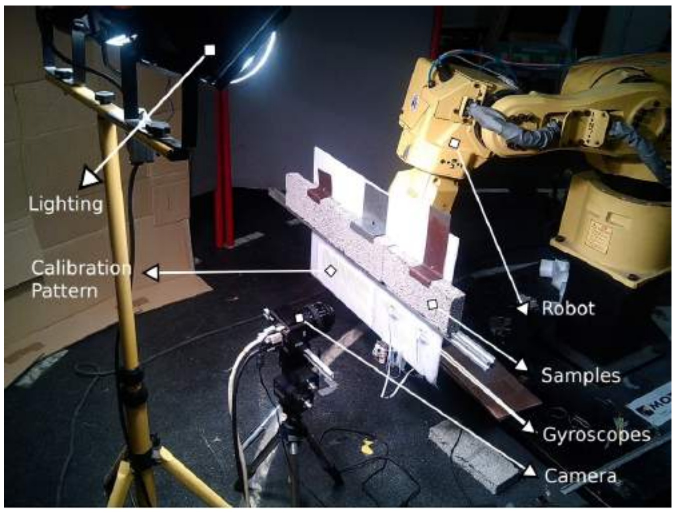


Figure 2.15: The experimental setup on the industrial robot [41]

From table 2.3, the following conclusions can be made:

- Except for radar odometry and GPS, it can be seen that accuracy and robustness have an inverse relationship. If one is high, the other is low.
- All non-visual odometry has high performance since the amount of information to be processed is not large, except GPS due to its need to connect to a satellite, causing it to be slow.
- In visual odometry, performance is medium because the vision sensor provides a considerable amount of data.

In the end, the chosen odometry should be the one that provides the capabilities that the user needs or is best suitable for the type of project.

Table 2.1: Sensors from multiple vehicles

Cars	Stationary LIDAR	360° rotating LIDAR	Camera	Radar	GPS/INS
Stanley [11]	✓	✗	✓	✓	✓
Junior [13]	✓	✓	✗	✓	✓
Boss [14]	✓	✓	✓	✓	✓
Sandstorm [43]	✓	✗	✗	✓	✓
Odin [44]	✓	✗	✓	✗	✓
Talos [45]	✓	✓	✓	✓	✓
A1 [18]	✓	✗	✓	✗	✓
iCab [46]	✓	✗	✓	✗	✗
Motional [47]	✗	✓	✓	✓	-
Waymo [48]	✓	✓	✓	✓	✓
Tesla [49]	✗	✗	✓	✓	✗
Atlascar1 [23]	✓	✗	✓	✗	✓
Atlascar2 [50]	✓	✗	✓	✗	✓

Table 2.2: Software architecture from multiple vehicles

Cars	Functional components	HMI	Communication	Framework	OS
Stanley [11]	**6**: Perception, Planning/ Control, Sensor interface, User interface, Vehicle interface, Global services	✓	IPC	-	Linux
Junior [13]	**6**: Perception, Navigation, Sensor interface, User interface, Vehicle interface, Global services	✓	IPC	-	Linux
Boss [14]	**4**: Perception, Mission planning, Behavioral executive, Motion planning	✓	TCP/IP or UDP	-	UNIX
Sandstorm [43]	-	-	IPT	-	RTOS
Odin [44]	**5**: Perception, Planning, Sensor interface, User interface, Control	-	JAUS	-	Windows/ Linux
Talos [45]	**3**: Perception, Planning/ Control	✓	LCM	-	-
A1 [18]	**5**: Localization, Perception, Planning, Control, System management	-	FlexRay	AUTOSAR	Windows/ OSEK
iCab [46]	**3**: Deliberative, Sequencing, Reactive	✓	TCPROS	ROS	Linux
Motional [47]	-	-	-	-	-
Waymo [48]	-	-	-	-	-
Tesla [49]	-	-	-	-	-
Atlascar1 [23]	-	-	TCP/IP	CARMEN	Linux
Atlascar2 [50]	-	✗	TCPROS	ROS	Linux

Table 2.3: Comparison of self-localization methods

Self-localization methods	Performance	Power eff.	Accuracy	Energy	Robustness	dimensions
GPS	Low	low-power	semi-accurate	non-efficient	high	2D
Wheel Odometry	High	low-power	non-accurate	efficient	low	2D
Inertial Odometry	High	low-power	non-accurate	efficient	high	3D
Radar Odometry	High	low-power	accurate	efficient	high	3D
Laser Odometry	High	high-power	accurate	non-efficient	medium	3D
Visual Odometry	Medium	high-power	accurate	non-efficient	low	3D

Chapter 3

Experimental Infrastructure

This chapter describes the tools used in this dissertation, both in hardware and software. The hardware includes the sensors applied in Atlascar2, which gather navigation information and obtain localization in the environment, the processing unit of the system and the component used to acquire CAN messages (Section 5.1). In terms of software, the vehicles framework serves to analyze the information gathered by the sensors, as well as other software tools used for the system to work correctly.

3.1 Hardware

The hardware described in this section is the Atlascar2's cameras, 3D and 2D LiDARs, GPS + IMU, and the processing unit. These components give Atlascar2 the necessary information about its surroundings. This section also describes the CANalyze component, which is used to receive and send the CAN messages from the vehicle's ECUs.

3.1.1 CANalyze

The CANalyze (Figure 3.1) is an open-source hardware capable to receive and transmit CAN messages using the Linux can-utils package. The CANalyze device performs on-board processing of the CAN packets to make them readable to the user, allowing the computer to become a node of the vehicle CAN bus.

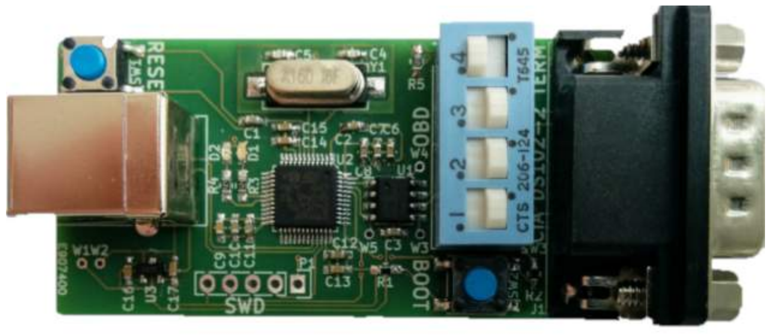


Figure 3.1: CANalyze [51].

3.1.2 Camera Point Grey FL3-GE-28S4-C

The Point Grey FL3-GE-28S4-C (Figure 3.2) is a 2.8 Megapixel color GigE Vision digital camera that uses a Sony ICX687 EXview HAD CCD II image sensor to deliver high resolution, high-quality images in a compact and low-cost package. At its highest resolution of 1928 × 1448, the camera runs at 15 FPS. However, it is possible to decrease its region of interest to obtain more frames per second [52]. Table 3.1 describes the main specifications of this sensor.

Figure 3.2: Camera Point Grey FL3-GE-28S4-C [52].

Table 3.1: Specifications from the Point Grey camera.

Specifications	
Resolution	1928 × 1448
Frame rate	15 FPS @ 1928 × 1448
Megapixels	2.8 MP
Chroma	Color
Sensor	Sony ICX687 EXview HAD CCD II
Dimensions	29 × 29 × 30 mm
Weight	38 g

3.1.3 Sick LMS151 LIDAR

The LiDAR is a sensor used to measure the distance to objects using laser delay time from the moment it is projected until it is received.

The LMS151 (Figure 3.3) is a 2D laser scanner (LiDAR) that presents reliable navigation, detection, and measurement data. Although the LiDAR described in Section 2.2.4 has limitations regarding poor weather conditions and transparent objects, the LMS151 provides a multi-echo technology and fog detection, allowing it to measure distances through glass, fog, and dust [53]. Table 3.3 describes the main features of this sensor.

Figure 3.3: LiDAR Sick LMS151 [53].

Table 3.2: Features from the LMS151 LiDAR.

	Features
Application	Outdoor
Light source	Infrared light (905 nm)
Laser class	1 (IEC 60825-1:2014, EN 60825-1:2014)
Aperture angle	270° (Horizontal)
Scanning frequency	25 Hz – 50 Hz
Angular resolution	0.25° – 0.5°
Working range	0.5 m – 50 m
Scanning range	18 m (10%) 50 m (90%)
Number of evaluated echoes	2

3.1.4 Sick LD-MRS400001 LIDAR

The LD MRS400001 (Fig. 3.6) is a 3D LiDAR that can measure up to 4 planes. Like the LMS151, it provides multi-echo technology which gives it immunity to rain, snow, and dust. This type of sensor is ideal for collision detection on automated vehicles or scanning objects [54]. Table 3.3 describes the main features of this sensor.

Figure 3.4: LiDAR Sick LD-MRS400001 [54].

Table 3.3: Features from the LD-MRS400001 LiDAR.

	Features
Application	Outdoor
Light source	Infrared light (905 nm, ś 10 nm)
Laser class	1 (IEC 60825-1:2014, EN 60825-1:2014)
Aperture angle	85° (Horizontal) 3.2° (Vertical)
Scanning frequency	12.5Hz - 50Hz
Angular resolution	0.125°,0.25°, 0.5°
Working range	0.5 m - 230 m
Scanning range	50 m
Amount of evaluated echoes	3

3.1.5 Novatel SPAN-IGM-A1 and Novatel GPS-702-GG

The Novatel SPAN-IGM-A1 (Figure 3.5a) and Novatel GPS-702-GG (Figure 3.5b) are the inertial navigation system and GPS of the Atlascar2. Combining these sensors can offer an ideal positioning solution based on Synchronous Position, Attitude, and Navigation (SPAN) technology. This method takes advantage of both the accuracy of the Global Navigation Satellite System (GNSS) and the stability of the Inertial Measurement Unit (IMU), resulting in a stable solution, allowing for exact global positioning, even when the satellite signal is blocked. The Tables 3.4 and 3.5 show some of these components' specifications.

(a) Novatel SPAN-IGM-A1 [55].

(b) Novatel GPS-702-GG [56].

Figure 3.5: GNSS and Inertial Navigation.

Table 3.4: Novatel SPAN-IGM-A1 specifications [55].

Features	
Input voltage	10-30 VDC
Time Accuracy	20 ns RMS
Max Velocity	515 m/s
Single point L1/L2 accuracy	1.2 m
IMU measurement data rate	200 Hz
INSS solution data rate	Up to 200 Hz
Dimensions	152 × 142 × 51 mm
Weight	515 g

Table 3.5: Novatel GPS-702-GG specifications [56].

Features	
Input voltage	4.5-18.0 VDC
Current (typical)	35 mA
3 dB pass band (typical)	L1: 1588.5 ±23.0 MHz L2: 1236.0 ±18.3 MHz
Noise figure (typical)	2.5 dB
L1-L2 differential propagation delay	5 ns
Diameter	185 mm
Weight	500 g

3.1.6 Nexus P-2308H4/HR4

The Nexus P-2308H4/HR4 is the processing unit of the Atlascar2. It has a 2Tb HHD storage capacity to store data and a 120Gb SSD where the OS is located for faster processing which is described in Section 3.2. More details can be seen in Table 3.6.

Figure 3.6: Nexus P-2308H4/HR4 [57].

Table 3.6: Nexus P-2308H4/HR4 technical specifications.

Specifications	
Processor	2× Intel XEON E5
RAM	32 GB
Storage	2 TB HHD + 120 GB SSD
Network	2× Gigabit onboard
Graphics	Nvidia Quadro
Dimensions (mm)	178W × 437H × 648L

3.1.7 Atlascar2

The Atlascar2 is equipped with all the sensors and equipment described above. There are also other sensors like the four Sick DT20 Hi optoelectronic sensors that assist in the inclinometer module. Figure 3.7 shows how the sensors are placed in the vehicle and Figure 3.8 shows the CANalyze connection with the Atlascar2's OBD-II port.

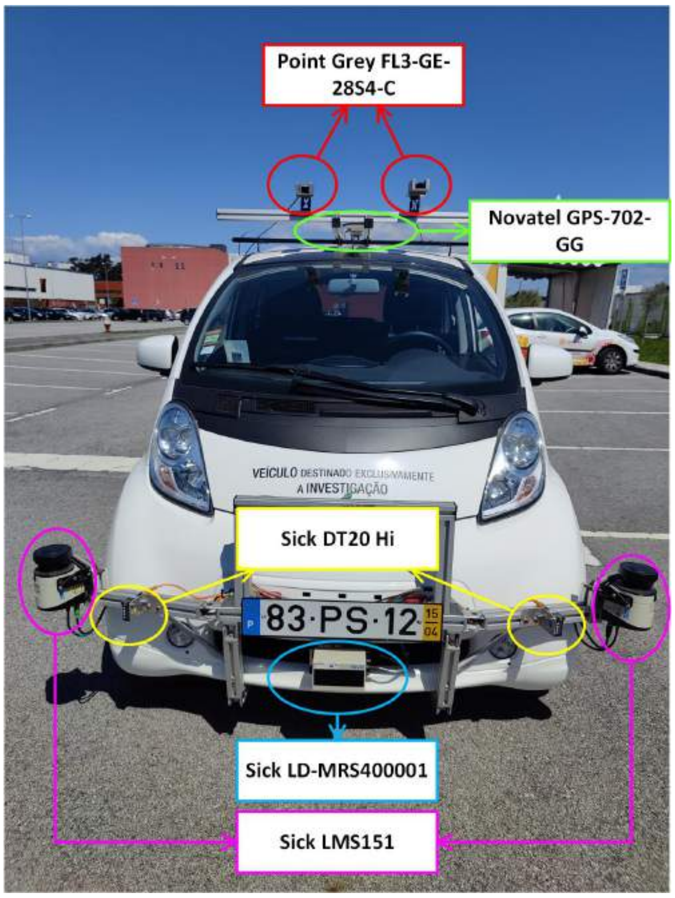


Figure 3.7: Position of sensors on Atlascar2.

3.2 Software

The software described in this Section is the main framework, ROS and its used packages for this project as well as other software tools. This software gives Atlascar2 the capability to gather the information from the sensors and perceive its surroundings.

3.2.1 Robot Operating System (ROS)

As described in Section 2.1, we need certain attributes for this project, such as independence of hardware and software, a communication protocol that is compatible between both components and software to provide a robust architecture. To deal with this complexity, ROS was used. ROS is an open-source framework for robot software development that provides the functionality of an operating system on a heterogeneous computer cluster. It enables communication between hardware and software using a well-planned communication protocol based on publisher-subscriber logic.

The way ROS works is simple; each task is hosted in a ROS `node`. These nodes communicate with each other via `messages` which can be published and subscribed through `topics` and `services`, linking the nodes. The communication process control is done by the master, who keeps a registry of all the nodes. Another important ROS feature is a set of tools to record `topics`, `rosbags`. The importance of this tool is to

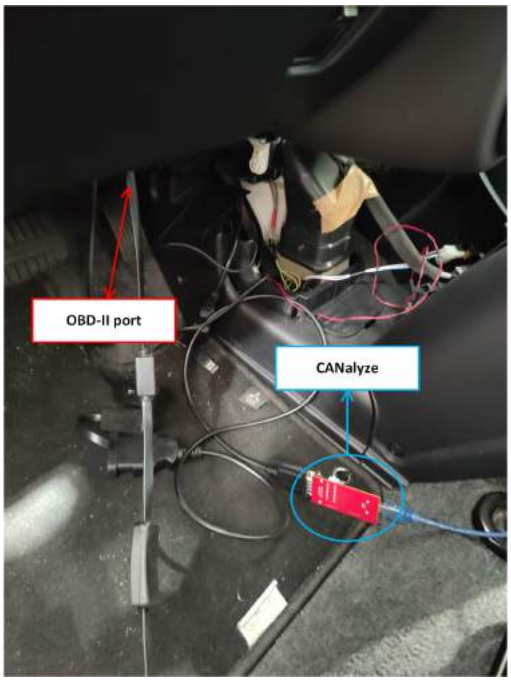


Figure 3.8: OBD-II port and CANalyze connection.

provide the capability to test the system with pre-recorded ROS `topics` which contain real data from the sensors. This `bag` file enhances work progress since we can change the developed tools and use them with that data, without always having to perform real testing, thereby improving debugging.

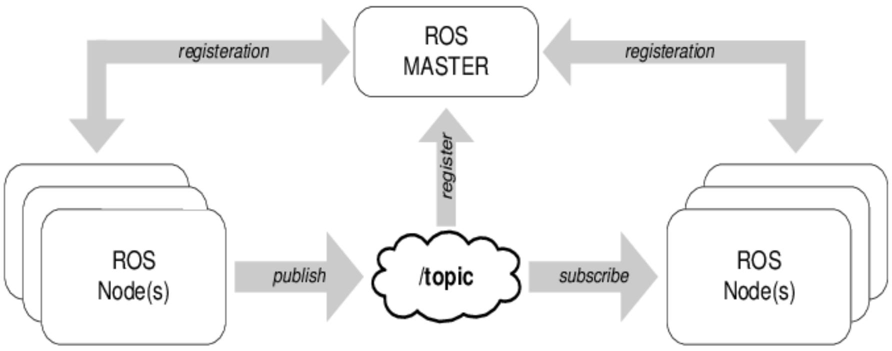
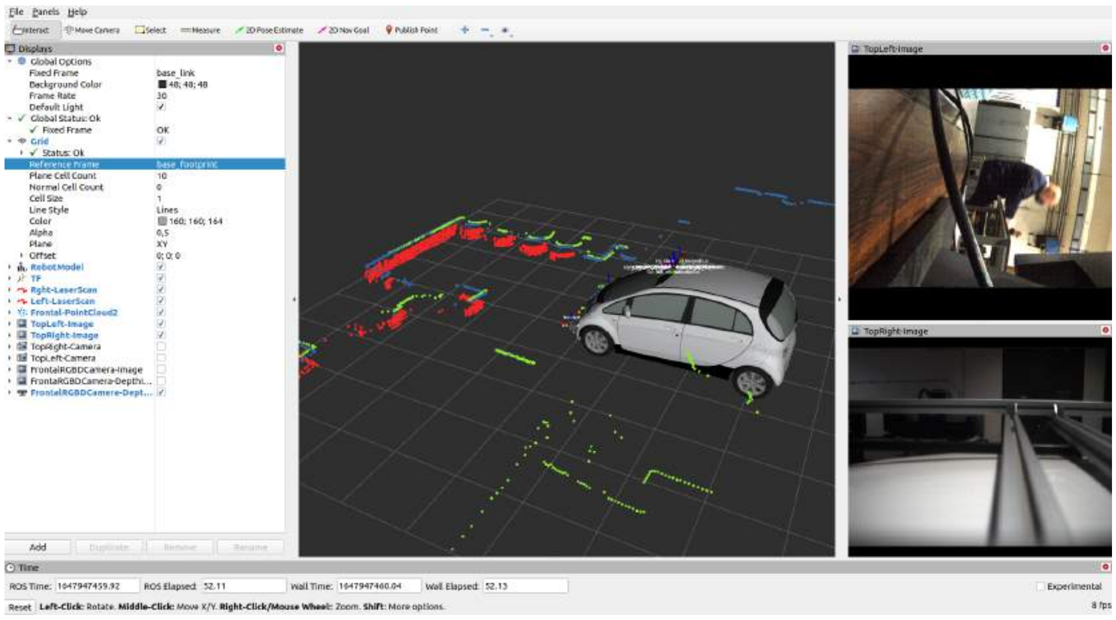


Figure 3.9: Diagram of ROS infrastructure [58]

3.2.2 Rviz

Rviz is an abbreviation for "ROS visualization" which works as a 3D visualization tool for the ROS framework [59]. This tool can show the visual information of a ROS project, such as the point cloud and laser scanners generated by the data that comes from a sensor, captured images from a camera, the robot model, odometry values, and more[1].

[1]http://wiki.ros.org/rviz/UserGuide



```
Global Options
  Fixed Frame              base_link
  Background Color         48; 48; 48
  Frame Rate               30
  Default Light            ✓
✓ Global Status: Ok
  ✓ Fixed Frame            OK
Grid                       ✓
  ✓ Status: Ok
  Reference Frame          base_footprint
  Plane Cell Count         10
  Normal Cell Count        0
  Cell Size                1
  Line Style               Lines
  Color                    160; 160; 164
  Alpha                    0,5
  Plane                    XY
  Offset                   0; 0; 0
RobotModel                 ✓
TF                         ✓
Rght-LaserScan             ✓
Left-LaserScan             ✓
Frontal-PointCloud2        ✓
TopLeft-Image              ✓
TopRight-Image             ✓
TopRight-Camera
TopLeft-Camera
FrontalRGBDCamera-Image
FrontaRGBDCamera-Depthi...
FrontalRGBDCamere-Dept...  ✓

ROS Time: 1647947459.92  ROS Elapsed: 52.11  Wall Time: 1647947460.04  Wall Elapsed: 52.13  Experimental
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.  8 fps
```

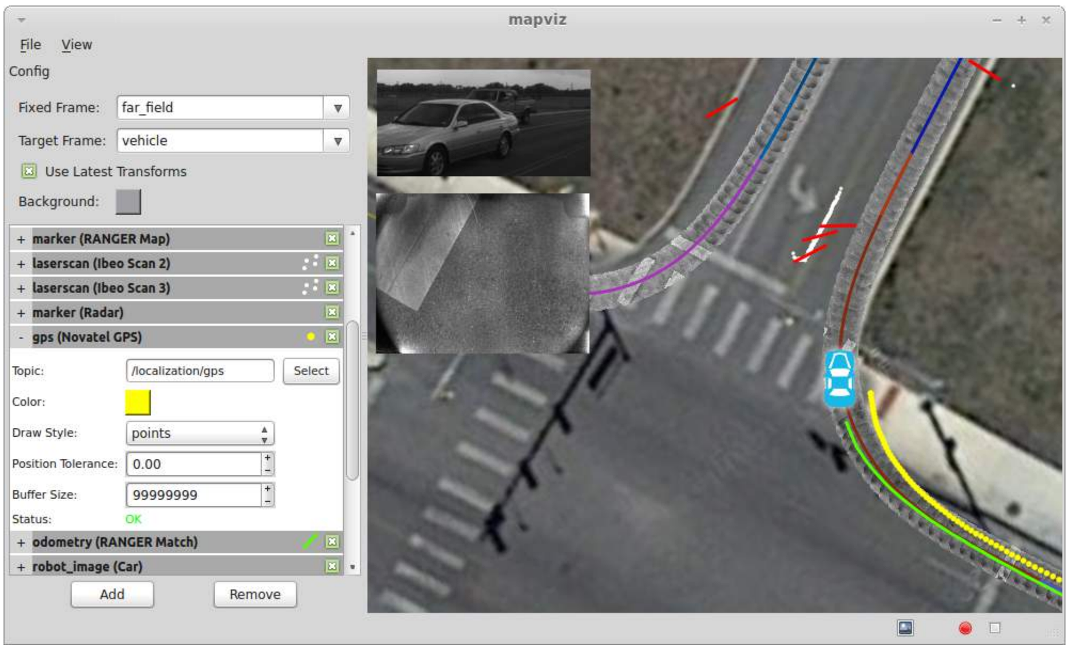
Figure 3.10: Rviz tool with Atlascar2 sensors.

Figure 3.10 shows the Atlascar2's sensors, the Atlascar2's model and its transformations, where the red pointcloud is the 3D LiDAR, the blue is the right 2D LiDAR and the green is the left 2D LiDAR. Camera's images are shown on the right.

3.2.3 Mapviz

Mapviz is a highly customizable ROS-based visualization tool focused on large-scale 2D data, with a plugin system for extreme extensibility [2]. The advantage of this tool is the capability of adding maps to improve the data visualization of moving objects such as vehicles. Figure 3.11 shows the data the mapviz tool can visualize, such as the GPS, the odometry, the LiDARs and more, everything in the environment it was taken, thanks to the map addition.

[2]https://swri-robotics.github.io/mapviz/



Fixed Frame: far_field
Target Frame: vehicle
Use Latest Transforms
Background:

+ marker (RANGER Map)
+ laserscan (Ibeo Scan 2)
+ laserscan (Ibeo Scan 3)
+ marker (Radar)
- gps (Novatel GPS)

Topic: /localization/gps
Color:
Draw Style: points
Position Tolerance: 0.00
Buffer Size: 99999999
Status: OK

+ odometry (RANGER Match)
+ robot_image (Car)

Figure 3.11: Mapviz tool.

3.2.4 Rqt

The rqt is a software framework of ROS that implements the various GUI tools in the form of plugins[3]. Some of the tools used in this project is the `rqt_graph` which helps visualize the topics and nodes, and the `robot steering`, which publishes a `Twist` message in a `topic` of the user's choice.

3.2.5 ROS packages used in this project

Follow the packages used for each sensor:

Sick LMS151 - `LMS1XX`

The `LMS1XX` ROS package supports every Sick LMS1xx laser scanner. This package includes the LiDARs' URDF (Unified Robotics Description Format).

Sick LD MRS - `sick_ldmrs_laser`

The `sick_ldmrs_laser` ROS package supports every Sick LD-MRS laser scanner. This package includes features such as the LiDARs' URDF and the capability to change specific properties like the start and end angle, the angular resolution, and the ability to ignore scan points up to 15m.

Point Grey Camera - `pointgrey_camera_driver`

The `pointgrey_camera_driver` ROS package is specially designed for Point Grey cameras and includes many interesting features such as the camera urdf and a config-

[3]http://wiki.ros.org/rqt

uration file that allows the camera details (e.g. video mode) to be changed without accessing to the Flycap application, the application normally used by this cameras.

Novatel GPS + IMU - `novatel_gps_driver`

The `novatel_gps_driver` ROS package is designed for Novatel GPSs and includes features such as choosing the device type (Ethernet or USB) and the messages the user wants to publish. Using USB, the user is able to choose the baudrate, the sample rate and the device port of the sensor, without needing to use the "Novatel connect" application.

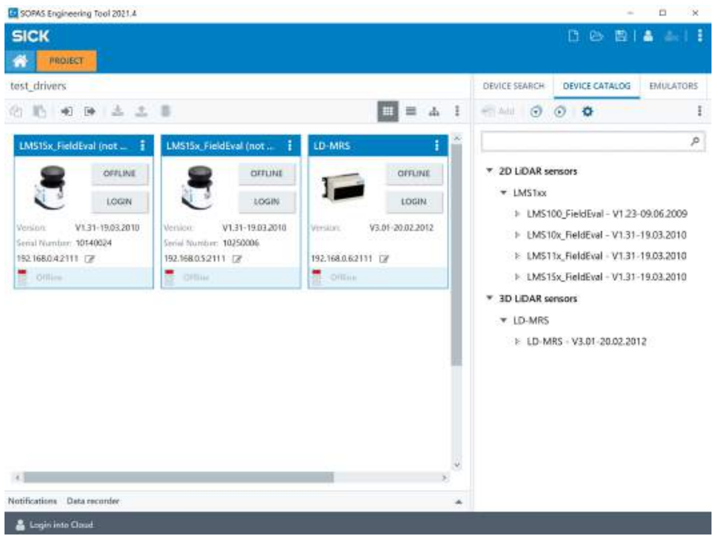
These packages provide ROS with the ability to receive the sensor data from the ROS topic when the components `nodes` are launched.

3.2.6 Other software tools

The software tools described in this Section are the SOPAS Enginnering tool and the flycapture application. Their purpose is changing settings that cannot be done in ROS, such as the IP address.

SOPAS Engineering tool

The SOPAS Engineering tool (Figure 3.12) is an application developed by SICK to display the detected SICK sensors, and choose its appropriate drivers [60]. With it, the user can see the data from the sensors and change the settings, such as the IP address, which cannot be done using the ROS packages.



LMS15x_FieldEval (not ...
Version: V1.31-19.03.2010
Serial Number: 10140024
192.168.0.4:2111

LMS15x_FieldEval (not ...
Version: V1.31-19.03.2010
Serial Number: 10250006
192.168.0.5:2111

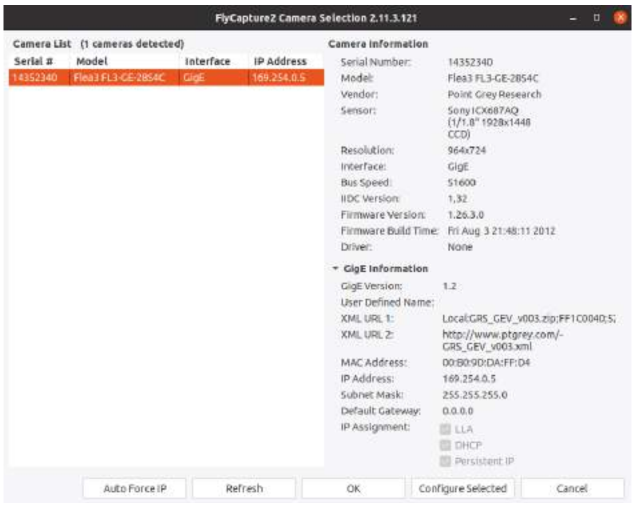
LD-MRS
Version: V3.01-20.02.2012
192.168.0.6:2111

- 2D LiDAR sensors
  - LMS1xx
    - LMS100_FieldEval - V1.23-09.06.2009
    - LMS10x_FieldEval - V1.31-19.03.2010
    - LMS11x_FieldEval - V1.31-19.03.2010
    - LMS15x_FieldEval - V1.31-19.03.2010
- 3D LiDAR sensors
  - LD-MRS
    - LD-MRS - V3.01-20.02.2012

Figure 3.12: SOPAS Engineering Tool

FlyCap

The FlyCap application (Figure 3.13) is a generic, easy-to-use streaming image viewer included with the FlyCapture SDK, [61] which can be used to test many of the camera capabilities. It enables a live video stream from the camera, to save individual images, to adjust the various video formats, frame rates, properties, and settings of the camera, as well as to access camera registers directly. As in the case of the SOPAS Engineering tool, the IP address can only be changed in this application.



Camera List (1 cameras detected)

Serial #	Model	Interface	IP Address
14352340	Flea3 FL3-GE-28S4C	GigE	169.254.0.5

Camera Information

Serial Number:	14352340
Model:	Flea3 FL3-GE-28S4C
Vendor:	Point Grey Research
Sensor:	Sony ICX687AQ (1/1.8" 1928x1448 CCD)
Resolution:	964x724
Interface:	GigE
Bus Speed:	S1600
IIDC Version:	1.32
Firmware Version:	1.26.3.0
Firmware Build Time:	Fri Aug 3 21:48:11 2012
Driver:	None

GigE Information

GigE Version:	1.2
User Defined Name:	
XML URL 1:	Local:GRS_GEV_v003.zip;FF1C0040;5;
XML URL 2:	http://www.ptgrey.com/-GRS_GEV_v003.xml
MAC Address:	00:B0:9D:DA:FF:D4
IP Address:	169.254.0.5
Subnet Mask:	255.255.255.0
Default Gateway:	0.0.0.0
IP Assignment:	LLA, DHCP, Persistent IP

Figure 3.13: FlyCap application

Chapter 4

Software Architecture

This chapter describes the advances and updates made to the software architecture of the Atlascar2. Although there are many requirements like the ones described in Section 2.1, this project focuses on reliability, compatibility, scalability and maintainability since these are more revelant in autonomous vehicles.

4.1 Processing unit performance

As mentioned in Section 2.1, compatibility is the proper communication between components, external systems, and the system itself. To begin with, the main component to be improved is the one that connects all the others, the processing unit.

The Atlascar2's processing unit described in Chapter 3, while very powerful in computational resources, is very slow in launching programs and even the system itself. Hence, for the system to work correctly, we decided to wipe all the data from the PC, to update the Ubuntu version 16.04 to the most recent one, 20.04 and to add a 120 Gb SSD card to separate the Operating System (OS) and data that can take up a large amount of space and processing capability. To achieve that separation, manual partitions were made on the processing unit, as shown in Table 4.1.

Table 4.1: Processing unit partitions.

Disk	Total space	Partitions	Space
HDD	2 TB	/data	2 TB
SDD	120 GB	/	50 GB
		swap	16 GB
		/home	54 GB

The boot time needs to be improved with the new system, which means that the configuration files and services from the PC must be changed. These are going to be described further in detail in Chapter 6.1.

4.2 Communication infrastructure

In Section 3.2.2 Diogo Correia showed in his work [50] how the components communicate with the processing unit. With the time passing, some components were removed and others added for the students' needs, which made this infrastructure outdated, requiring it to be redone. Figure 4.1 shows a newly added switch to separate the networks of the bumper and roof components, as well as a GPS+IMU that uses USB to communicate with the processing unit.

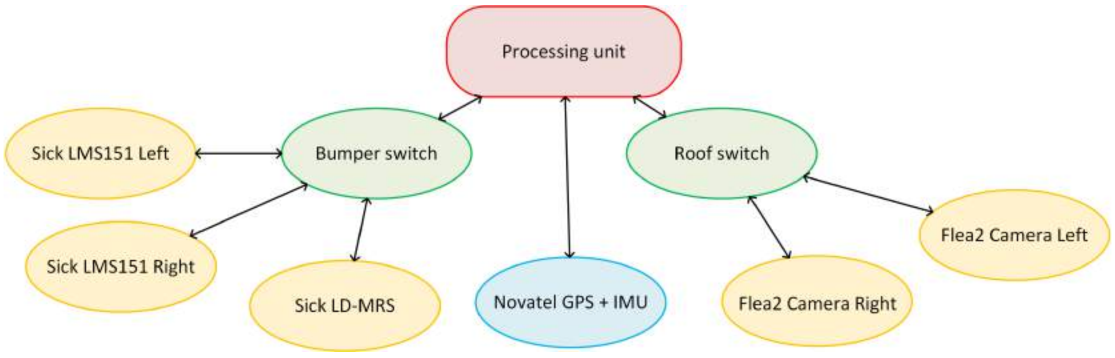


Figure 4.1: Communications diagram.

Static addresses were created for each processing unit's port, as well as to change the IP addresses to a more straightforward approach since the previous works in this area were confusing and not well dated. To change the IP addresses of the lasers and the cameras, the SOPAS software and the flycapture program described in Section 3.2.6 were used respectively.

Table 4.2: Atlascar2's ports addresses.

Port name	Usage	IP address
ens6f1	Front Bumper Switch	192.168.0.3
ens6f0	Roof Switch	169.254.0.3
enp5s0f1	UA Ethernet	automatic

Table 4.3: IP addresses of sensors.

Sensors	Old IP address	New IP address
Sick LMS151 Right	192.168.0.231	192.168.0.4
Sick LMS151 Left	192.168.0.134	192.168.0.5
Sick LD-MRS	192.168.0.244	192.168.0.6
Flea2 Camera Right	169.254.0.102	169.254.0.4
Flea2 Camera Left	169.254.0.101	169.254.0.5
Novatel GPS + IMU	-	-

The 192.168.0.X and 169.254.0.X networks were chosen bearing in mind that they are

private IP addresses, being non-routable. That means they cannot be reached outside their network, which helps with data protection and privacy and is more secure thanks to the lack of access by the other networks [62].

4.3 Launching Nodes

The launch files and some of the driver's code had to be enhanced to improve the specifications this project focuses on. The launch files provide a convenient way to start up multiple nodes and a master and other initialization requirements such as setting parameters and loading YAML files.

4.3.1 Top level launch file

The main file for the Atlascar2, `bringup.launch` launches the sensors previously described in Chapter 3.1.7 with the `drivers_bringup.launch`, Rviz with the `visualize.launch` file and the Atlascar2's URDF with the `model.launch` file. This launch file has arguments for the sensors installed on the car and the ones that were already removed, in case they need to be installed again. Users may choose if they want to launch the sensors or not by setting the respective variable to true or false. The same can be done for the `visualize` argument.

Listing 4.1: Main launch file

```xml
<?xml version="1.0"?>
<launch>
   <arg name="visualize" default="true" />
   <arg name="2DLidar_left_bringup" default="true"/>
   <arg name="2DLidar_right_bringup" default="true"/>
   <arg name="3DLidar_bringup" default="true"/>
   <arg name="top_camera_right_bringup" default="true"/>
   <arg name="top_camera_left_bringup" default="true"/>
   <arg name="front_camera_bringup" default="false"/>
   <arg name="RGBD_camera_bringup" default="false"/>
   <arg name="novatel_bringup" default="false"/>
   <arg name="vehicle_name" default="atlascar2"/>


   <include file="$(find atlascar2_bringup)/launch/drivers_bringup.launch" >
      <arg name="2DLidar_left_bringup" value="$(arg 2DLidar_left_bringup)"/>
      <arg name="2DLidar_right_bringup" value="$(arg 2DLidar_right_bringup)"/
          >
      <arg name="3DLidar_bringup" value="$(arg 3DLidar_bringup)"/>
      <arg name="top_camera_right_bringup" value="$(arg
          top_camera_right_bringup)"/>
      <arg name="top_camera_left_bringup" value="$(arg
          top_camera_left_bringup)"/>
      <arg name="front_camera_bringup" value="$(arg front_camera_bringup)"/>
      <arg name="RGBD_camera_bringup" value="$(arg RGBD_camera_bringup)"/>
      <arg name="novatel_bringup" value="$(arg novatel_bringup)"/>
   </include>

   <include file="$(find atlascar2_bringup)/launch/model.launch" />

   <group if="$(arg visualize)">
```

```
      <include file="$(find atlascar2_bringup)/launch/visualize.launch"/>
   </group>

</launch>
```

This launch file makes the users prior knowledge about the files unnecessary since they only need to change the variable value depending on the sensors they want to use. Figure 4.2 shows the `rqt_graph` when the `bringup.launch` file is launched.

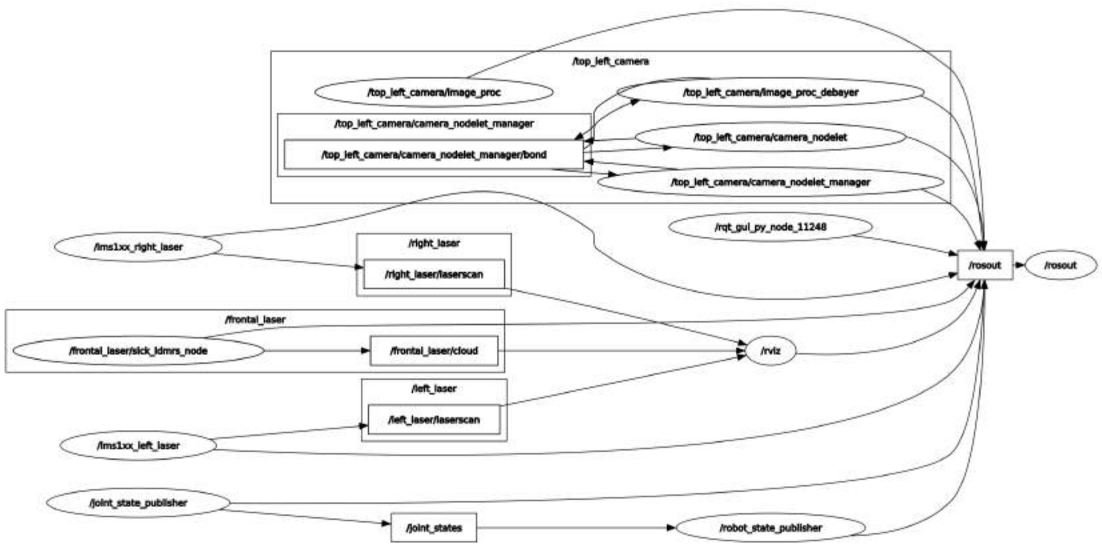


Figure 4.2: `rqt_graph` of the `bringup.launch`.

4.3.2 Launch file architecture

As previously mentioned, there are three major launch files in the architecture, the `drivers_ bringup.launch`, the `visualize.launch` and the `model.launch`.

The `drivers_bringup.launch` is the one that includes the sensors launch files, which are included in a different directory to maintain the system organized. Those files are usually the launch files the drivers' ROS packages provide. Listing 4.2 shows the usage of the sensors' arguments.

Listing 4.2: `drivers_bringup` launch file

```
<?xml version="1.0"?>
<launch>
<!-- args to specify what sensor to launch -->
   <arg name="2DLidar_left_bringup" default="true"/>
   <arg name="2DLidar_right_bringup" default="true"/>
   <arg name="3DLidar_bringup" default="true"/>
   <arg name="top_camera_right_bringup" default="true"/>
   <arg name="top_camera_left_bringup" default="true"/>
   <arg name="front_camera_bringup" default="false"/>
   <arg name="RGBD_camera_bringup" default="false"/>
   <arg name="novatel_bringup" default="false"/>

<!-- left 2D laser -->
```

```
  <group if="$(arg 2DLidar_left_bringup)">
      <include file="$(find atlascar2_bringup)/launch/include/laser2d_bringup.launch">
         <arg name="name" value="left" />
      </include>
  </group>

<!-- right 2D laser -->
  <group if="$(arg 2DLidar_right_bringup)">
      <include file="$(find atlascar2_bringup)/launch/include/laser2d_bringup.launch">
         <arg name="name" value="right" />
      </include>
  </group>

<!-- front 3D laser-->
  <group if="$(arg 3DLidar_bringup)">
      <group ns="frontal_laser">
         <include file="$(find atlascar2_bringup)/launch/include/sick_ldmrs_node.launch">
         </include>
      </group>
  </group>

<!-- top left camera -->
  <group if="$(arg top_camera_left_bringup)">
      <include file="$(find atlascar2_bringup)/launch/include/top_cameras_bringup.launch">
         <arg name="name" value="left" />
      </include>
  </group>

<!-- top right camera -->
  <group if="$(arg top_camera_right_bringup)">
      <include file="$(find atlascar2_bringup)/launch/include/top_cameras_bringup.launch">
         <arg name="name" value="right" />
      </include>
  </group>

<!-- front camera -->
  <group if="$(arg front_camera_bringup)">
      <include file="$(find atlascar2_bringup)/launch/include/pointgrey_zebra2.launch">
         <arg name="camera_name" value="frontal_camera" />
         <arg name="camera_serial" default="14233704" />
         <arg name="frame_id" value="frontal_camera" />
      </include>
     <node pkg="free_space_detection" type="device_frame_publisher_node" name="device_frame_publisher" required="true" output="screen"/>
  </group>

<!-- RGBD camera -->
  <group if="$(arg RGBD_camera_bringup)">
      <include file="$(find atlascar2_bringup)/launch/include/asus_xtion.launch">
         <arg name="camera" value="top_center_rgbd_camera" />
         <arg name="camera_serial" default="" />
```

```
        <arg name="rgb_frame_id" default="/$(arg cammera_name)
            _rgb_optical_frame" />
        <arg name="depth_frame_id" default="/$(arg cammera_name)
            _depth_optical_frame" />
     </include>
  </group>

   <group if="$(arg novatel_bringup)">
      <include file="$(find atlascar2_bringup)/launch/include/
          novatel_simple_bringup.launch">
      </include>
   </group>

</launch>
```

The other two files the `visualize.launch` in Listing 4.3, which launches the saved rviz file that appears in Figure 3.10 and the `model.launch` in Listing 4.4, which launches the urdf from the Atlascar2 as well as the robot joints and the robot state publisher, which calculates the forward kinematics and publishes it via tf.

Figure 4.3 shows the system hierarchy when launching the `bringup.launch` file.

Listing 4.3: `visualize` launch file

```
<?xml version="1.0"?>
<launch>
   <arg name="model_name" default=""/>
   <node pkg="rviz" type="rviz" name="rviz" required="false" args="-d $(find
       atlascar2_bringup)/config/rviz$(arg model_name).rviz"/>
</launch>
```

Listing 4.4: `model` launch file

```
<?xml version="1.0"?>
<launch>
   <arg name="robot_description" default="robot_description"/>
   <param name="$(arg robot_description)" command="$(find xacro)/xacro '$(
       find atlascar2_description)/urdf/atlascar2.urdf.xacro'" />
   <node pkg="joint_state_publisher" type="joint_state_publisher" name="
       joint_state_publisher"></node>
   <node pkg="robot_state_publisher" type="robot_state_publisher" name="
       robot_state_publisher" />
</launch>
```

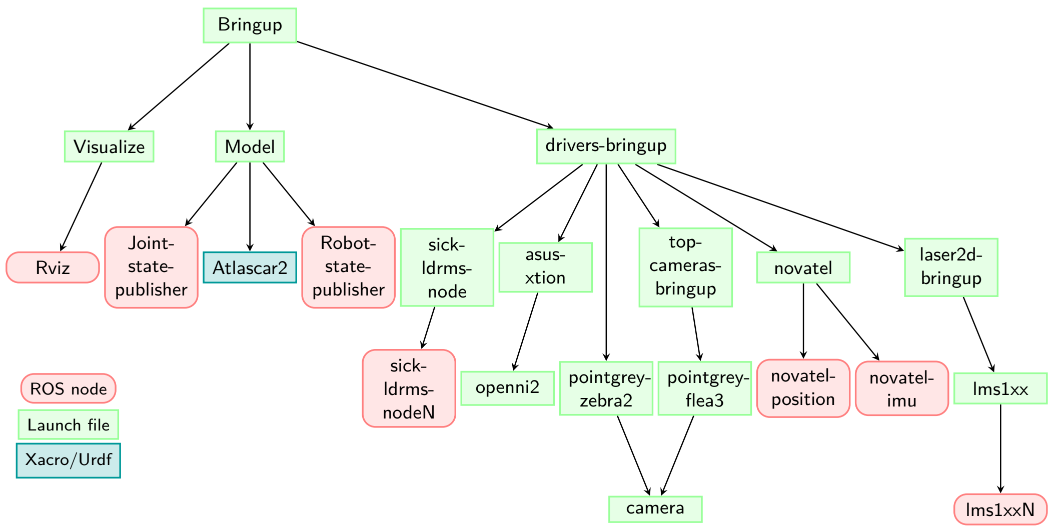


Figure 4.3: System hierarchy when launching the `bringup` file

4.4 Documentation

For a system to maintain its robustness, future users need to properly understand how to use it. For that, documentation was created in the project's README on github [63] for the following projects to come, allowing the students to run the system efficiently. The documentation includes:

- The core packages the project needs for it to work;
- How to turn on the vehicle and its components;
- How to configure the IP addresses;
- How to connect to the Atlascar2 with remote work;
- Testing the sensors;
- Launching the system;
- How to simulate the Atlascar2 on the PC.

Chapter 5

Development of an Odometry Solution

The second main objective of this dissertation is to develop an odometry solution for the Atlascar2. To accomplish that, it is necessary to understand the components and protocols needed and how to compute the odometry.

5.1 Approach of the Odometry Solution

The first thing to establish is the required variables to compute the odometry and how we can collect them. Since the odometry is the position estimation over time, the values needed to obtain are the vehicle's pose (x,y,θ) shown in Figure 5.1. Since we do not know these values, we need to calculate them, which is possible if we know the speed and wheel angle of the vehicle.

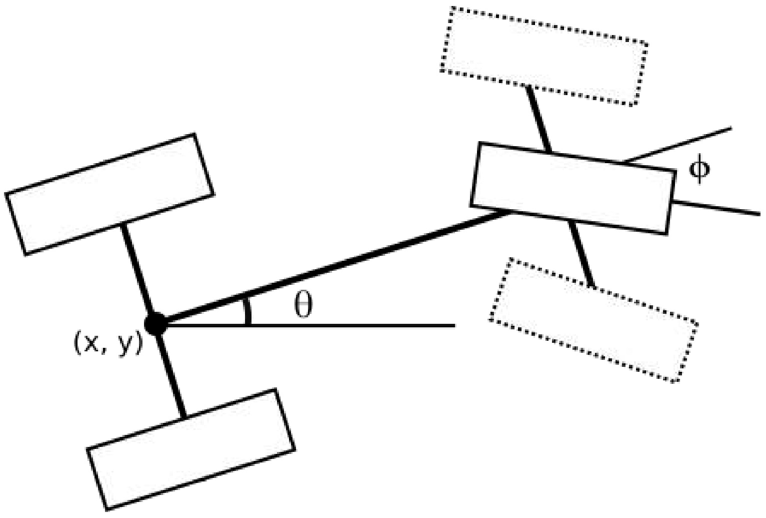


Figure 5.1: Ackermann kinematic model [64]

In a car such as the Atlascar2, there are ECUs (Eletronic Control Units), which gather most of the vehicle's information, including the vehicle's speed and steering angle. We can receive the ECUs information using the vehicle's CAN bus, a standardized serial communication protocol widely used in automobile internal control systems [65]. Some of the most common vehicle's ECUs are shown in Figure 5.2.

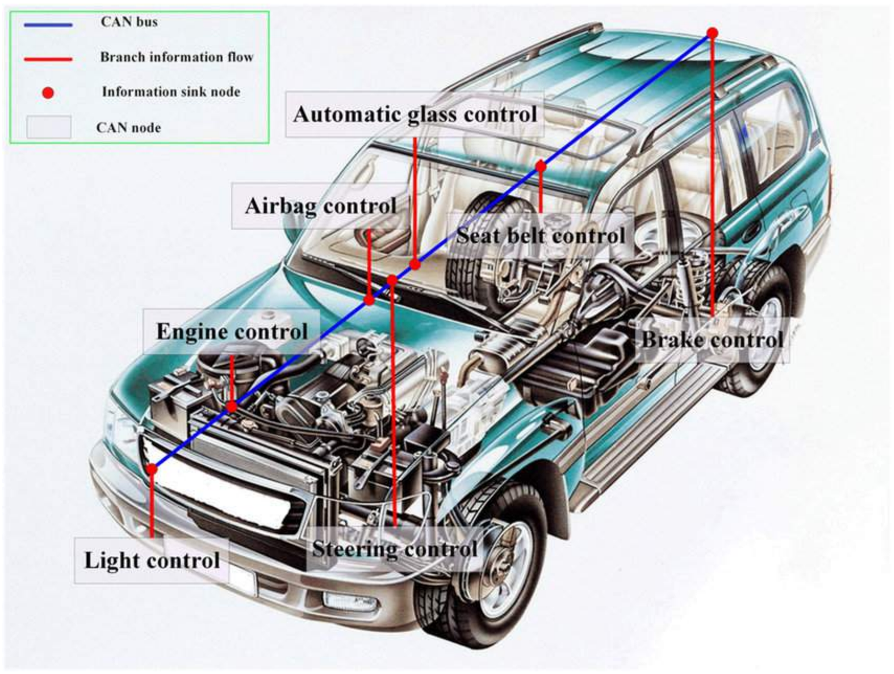


Figure 5.2: Some of the ECUs and CAN bus in a vehicle [66]

This approach looked very promising since the CAN bus also provides various advantages for this project [67, 68]:

- CAN speed can go up to 1Mb/s, allowing a considerable amount of data to be exchanged.
- It is a simple wired structure, reducing errors, weight, wiring, and costs;
- It is robust system against electromagnetic interference and electric disturbances;
- It has message prioritization, avoiding data interruption and CAN errors.

After collecting the necessary variable values, we must choose a software infrastrcuture to compute the odometry. ROS was the obvious choice since it can be integrated with the Atlascar2's software architecture, facilitating its implementation, which is one of the objectives of this thesis.

5.2 CAN communication

The CAN communication is done using CAN frames. The structure of these messages is presented in Figure 5.3, and explained below.

- SOF (Start of Frame): logic 0 to indicate the other nodes (ECUs) the beginning of a CAN frame;

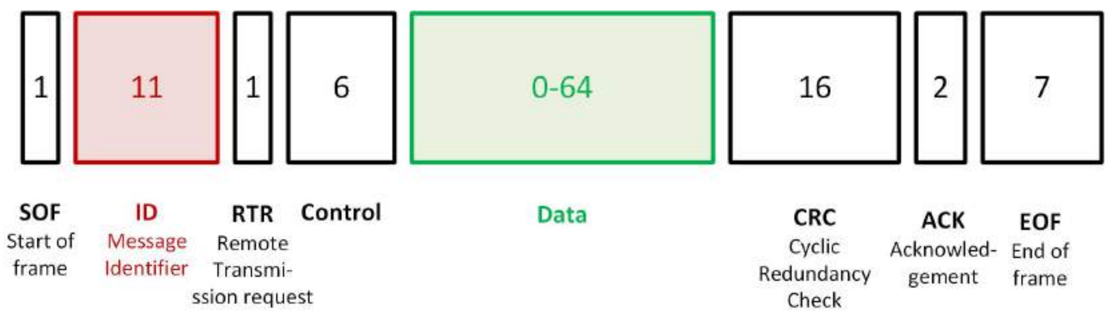


Figure 5.3: Standard CAN frame structure [67].

- ID (Identifier): identifies the data content;
- RTR (Remote Transmission Request): specifies whether the frames sends data or requests data from another node;
- Control: indicates the frame type and data length;
- Data: contains up to 8 bytes of data;
- CRC (Cyclic Redundancy Check): ensures data integrity checking for errors;
- ACK (Acknowledge): field used to specify if the node has received the data correctly;
- EOF (End of Frame): indicates the end of the CAN frame.

In this thesis, the two fields that require greater attention are Message Identifier and Data. The ID field identifies the nature of the Data and is the first field to look for and procress.

5.3 Mitsubishi i-MiEV CAN values

The works in [69, 70, 71, 72] indicate the most common informations in the CAN bus of the Mitsubishi i-MiEV, such as their ID and respective function. For this project, the more important values are presented in Table 5.1.

Table 5.1: CAN bus values.

Variable	ID	Bytes used	Formula
Speed (V)	0x412	B1	$V = B1$ m/s
Motor rpm (RPM)	0x298	B6,B7	$RPM = (B6 \times 256 + B7) - 10000$ rpm
Steering angle (ψ)	0x236	B0,B1	$\psi = \frac{(B0 \times 256 + B1) - 4096}{2}\ ^{\circ}$

5.3.1 Development of the First Solution

The first solution used the speed and steering angle from the CAN bus. A problem with this method was discovered right at the beginning. As seen in the equations above, the steering angle uses 2 bytes, which gives 0.5 degrees of resolution. Although this value is suitable for this work, the speed value is not because it only has 1 km/h resolution (0.33 m/s). Since the goal is also to calibrate the Atlascar2 at low speeds, this resolution was insufficient, so another solution had to be found.

The second solution used the vehicle's electric motor revolutions with the vehicle's wheel radius (r) and gear ratio from the Mitsubishi specifications to calculate the velocity (V) with the expression (5.1). This method provided a larger resolution which improved the speed's resolution.

$$V = \frac{2\pi r \times RPM}{60 \times Gear_{Ratio}} \text{ m/s} \tag{5.1}$$

The problem with this method comes in practice. When the vehicle brakes or stops, the motor still takes time to stop, which produces noise values. Comparing both solutions, we get the result in Figure 5.4.

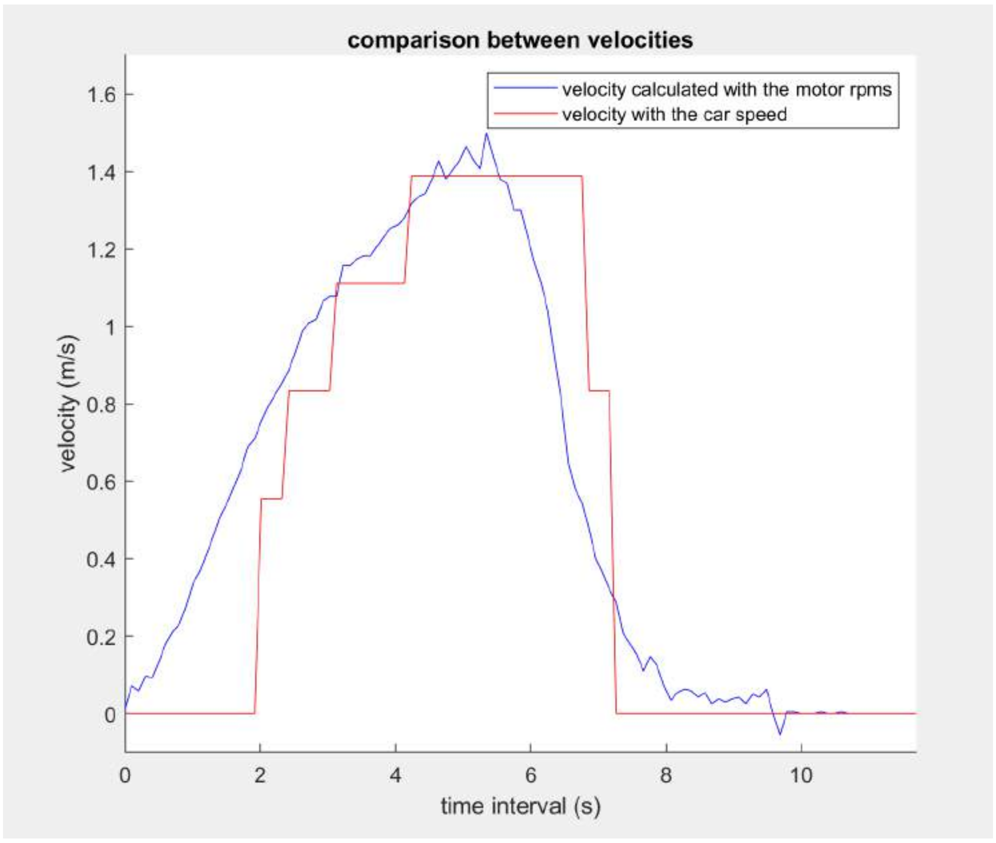


Figure 5.4: Comparison of the first and second solutions' velocities

Figure 5.4 shows the imprecision of the first solution, using the speed, and the braking noises in the beginning and ending from the second one, which uses the RPMs. So, a new approach was chosen to increase the chance of a more precise result.

Another method is needed and that involves adding a wheel encoder to the Atlascar2's back wheel, calculating the speed with the encoder's pulses, and sending them using a custom message identifier in the CAN bus.

5.4 Wheel Encoder Installation

The installation of the encoder on the Atlascar2 was based on the 2011 work of Tiago Rocha [40], mentioned in Section 2.2.6. Most of the parts of that implementation were used. The changes made were in the part that connects to the wheel's rim and the encoder. Also, it was decided to use an encoder with a higher pulse per revolution, explained next.

5.4.1 Selection of a new Encoder

A new encoder was chosen due to the need of a precise odometry solution at low speeds. The encoder is the RI32-0/1000ER.14KB (Figure 5.5), which has similar measures to the one in Tiago's thesis. Table 5.2 shows the encoder specifications.

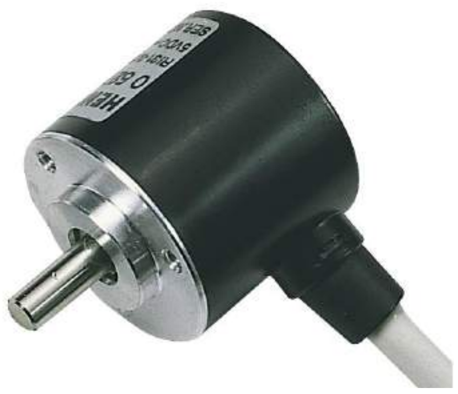


Figure 5.5: Encoder RI32-0/1000ER.14KB [73].

Table 5.2: RI32-0/1000ER.14KB specifications [73].

Specifications	
Type	Incremental
IP rating	IP40, IP50
Diameter	30 mm
Shaft Length	10 mm
Shaft Diameter	5 mm
Resolution	1000PPR
Max rotational speed	6000 rpm
No. of Channels	3 channels

For this encoder to operate properly, the vehicle's RPMs cannot pass the encoder's max rotational speed. Knowing that the Atlascar2 wheel diameter (D) is 0.57 m and the top speed is 130 km/h (2166.6 m/min), we get the following revolutions per minute:

$$RPM = \frac{V}{\pi \times D} = \frac{2166.6}{\pi \times 0.57} = 1210\text{rpm} \tag{5.2}$$

This value is significantly below the encoder max value. Another critical factor is the encoder's resolution, which can be calculated using the wheel perimeter and the pulses per revolution.

$$distance_{traveled} = \frac{P}{PPR} = \frac{\pi \times 0.57}{1000} = 0.0017 \text{ m} \tag{5.3}$$

This value means that, for every 0.0017m that the car moves, the velocity value is updated, giving us a precise value of the traveled distance each instance.

These two factors and having an allowed input voltage of 10-30VDC, which can be connected to the 12V Atlascar2's electric panel, make this encoder an adequate component for the project.

5.4.2 Assembly of Encoder

This section focuses on the assembly of the encoder. Most of the manufactured parts are presented in Tiago Rocha's thesis [40].

Besides the different part that connects to the rim, there is also a minor difference from Tiago Rocha's work. The new encoder has a shaft diameter of 5mm, which is a millimeter smaller than the previous one. The shaft was inserted in a plastic tube heated to stick to the encoder and secure the part to accommodate that difference without creating a new piece. The piece's stud was also added for extra security. This connection is shown in Figure 5.6.

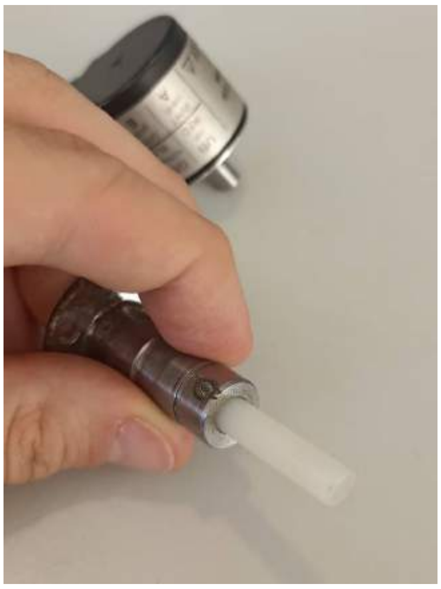


Figure 5.6: The tube that connects the encoder to the manufactured parts.

The assembly is presented in Figures 5.7 and 5.8 and finally, the final product is shown in Figure 5.9. The encoder's cable connects using the same pathway the Sick DT20 Hi optoelectronic sensors' back cables, shown in Figure 5.10.

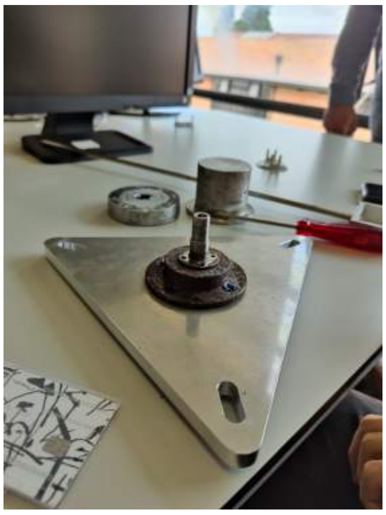


Figure 5.7: Encoder assembly beginning.

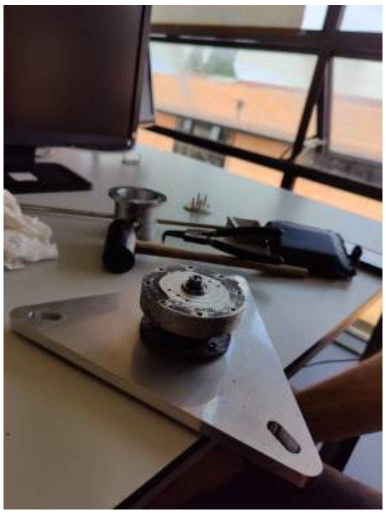


Figure 5.8: Encoder assembly intermediate.

5.4.3 Development of the Final Solution

Finally, the last solution consists of using the already acquired steering angle from the CAN bus of the Atlascar2 and the calculated speed from the newly installed encoder's pulses. As already mentioned, the encoder pulses need to be acquired and sent to the CAN bus with a custom message identifier. To count the encoder's pulses, it was decided to use an Arduino UNO wifi rev2 (Figure 5.11a) with a CAN-BUS shield (Figure 5.11b). The Arduino specifications are shown in Table 5.3.

Table 5.3: Arduino UNO wifi rev2 specifications [75].

Specifications	
Operating voltage	5V
Input voltage (recommended)	7 - 12V
Digital I/O pins	14 (5 Provide PWM Output)
Analog input pins	6
Clock speed	16 MHz
Dimensions	68.6×53.4 mm
Weight	25 g

Since the encoder has a high resolution, some calculations are necessary to check if the Arduino is powerful enough to handle the encoder's data flow. As mentioned before, the Atlascar2's top speed is 130 km/h (36.1 m/s), and its wheel diameter is 0.57 m. The number of rotations the wheel gives in one second (RPS) at top speed is

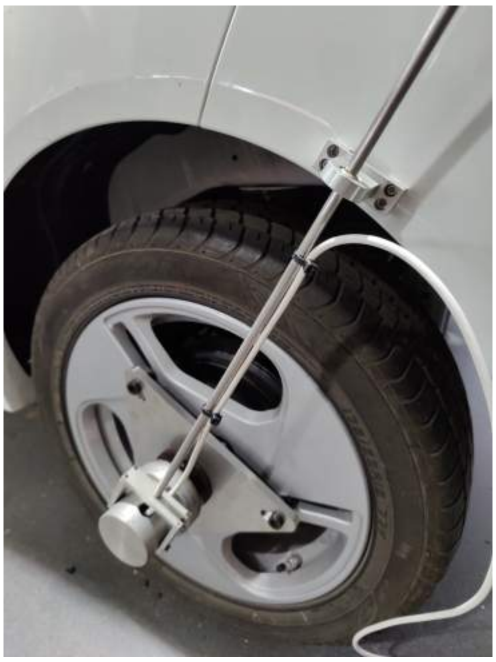


Figure 5.9: Encoder assembly in the Atlascar2.

$$RPS = \frac{V}{P} = \frac{36.1}{\pi \times 0.57} = 20.16 \text{ rps} \tag{5.4}$$

Knowing the RPS and the number of pulses per rotation, which are specified in Table 5.2, it is possible to know the number of pulses per second (PPS).

$$PPS = PPR \times RPS = 1000 \times 20.16 = 20160 \text{ pps} \tag{5.5}$$

For an encoder with two channels the maximum number of interrupts is 4, so this encoder needs 80640 interrupts per second at its top speed, meaning that the Arduino needs to run at more than 80 kHz to read all the encoder pulses.

Checking the Arduino's specifications in Table 5.3, we can see that its clock speed is 16 MHz. However, each instruction and interruption takes time, decreasing the frequency the Arduino is capable of running. In [77, 78] there is some information about this topic that concludes that with a simple program this frequency should work.

Furthermore, the next step consists of connecting the Arduino to the encoder. Since the 12V from the electric panel powers the encoder, its channels vary between 0V and 12V, which is too high because the Arduino operates at 5V. A voltage divider was added in both channels, as shown in Figure 5.12.

The values of resistors R1 (R1=R2) and R3 (R3=R4) are 1k8Ω and 1k3Ω, respectively. The Arduino code was then tested, by moving the car 1.80 m, which is the wheel's perimeter, meaning the encoder should give 1000 pulses. The acquired values were 1030 and -6, which is accurate enough since the measuring method is limited since the encoder is very precise and we used a measuring tape. It was then modified to have triggers on both channels in change mode, which gives four times more pulses, for better precision in case the Arduino misses some pulses. The results were 4046 and 2 which dividing by four show values closer to expected.

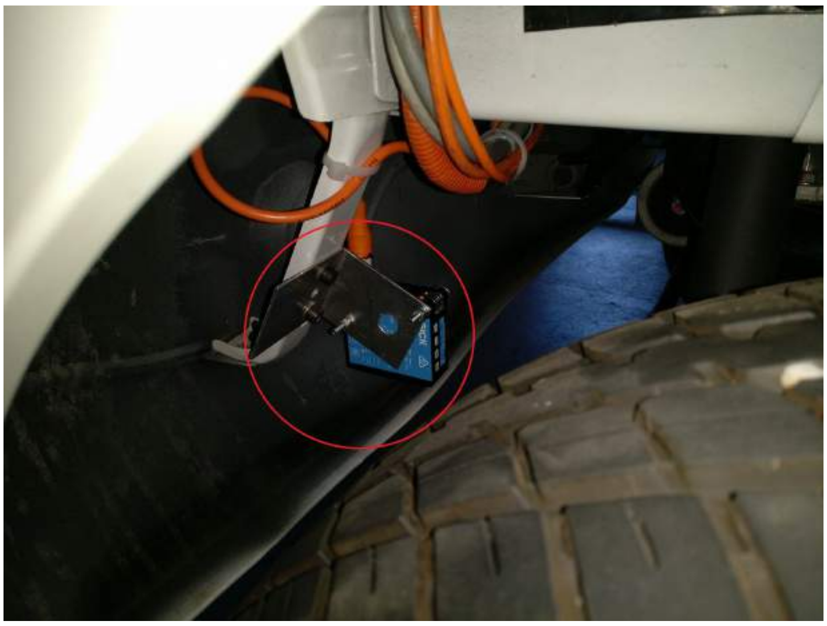


Figure 5.10: Sick DT20 Hi optoelectronic back path [74].

(a) Arduino UNO wifi rev2 [75]

(b) CAN-BUS SHIELD V2.0 [76]

Figure 5.11: Solution to send the CAN bus data.

5.5 Computation of the odometry

To use the speed and angle of the vehicle described above, a ROS node needs to subscribe to those messages, converting them to odometry values, and publishing them. To accomplish that, two programs were created. The first one obtains the CAN messages, converts the encoder pulses and steering angle to the necessary values, and publishes them as an `AckermannDriveStamped` message. The second one receives those values and calculates the odometry, publishing it as an `Odometry` message. Both of these messages are standard in ROS.

This implementation of the odometry solution is presented in Figure 5.13, which shows the necessary data, the Python codes used, and the published and subscribed topics for this project.

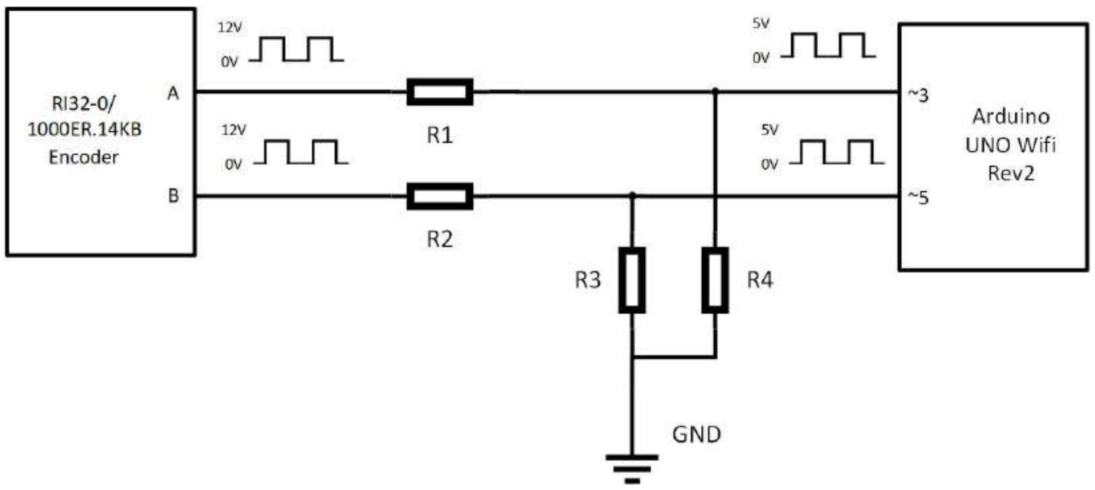


Figure 5.12: Arduino and encoder connection.

5.5.1 Collecting the Ackermann values

The speed and angular velocity are necessary to calculate the odometry, and both can be obtained using the encoder pulses and steering angle from the CAN-Bus of the vehicle.

To calculate the speed, we need to know the number of pulses of the encoder per second (PPS), acquired by the difference of pulses dividing by the time, which dividing by the max PPR of the encoder, converts into the number of rotations per second. Then, that number can be converted into meters per second by multiplying the wheel's perimeter, finding the speed.

$$PPS = \frac{\Delta Pulses}{\Delta t} \text{ pps} \tag{5.6}$$

$$RPS = \frac{PPS}{PPR} \text{ rps} \tag{5.7}$$

$$V = RPS \times P \text{ m/s} \tag{5.8}$$

Also, the speed and the steering angle are necessary to calculate the angular velocity. After receiving the steering angle (ψ) from the CAN using the formula in Table 5.1 and converting it to radians, we can use the formula (5.9) to obtain the wheel angle (ϕ), which divides the steering angle by the steering ratio 16.06, the Mitsubishi's difference of the wheel and steering angle that can be obtained in the Mitsubishi i-Miev specifications [79]. Finally, the angular velocity (ω) is calculated with formula (5.10), where W is the wheelbase, the distance between the front and back wheels and V is the vehicle's speed [64].

$$\phi = \frac{\psi}{16.06} \text{ rad} \tag{5.9}$$

$$\omega = \frac{V}{W} \times \tan(\phi) \text{ rad/s} \tag{5.10}$$

These values are sent in the `AckermannDriveStamped` message to use in the odometry calculation.

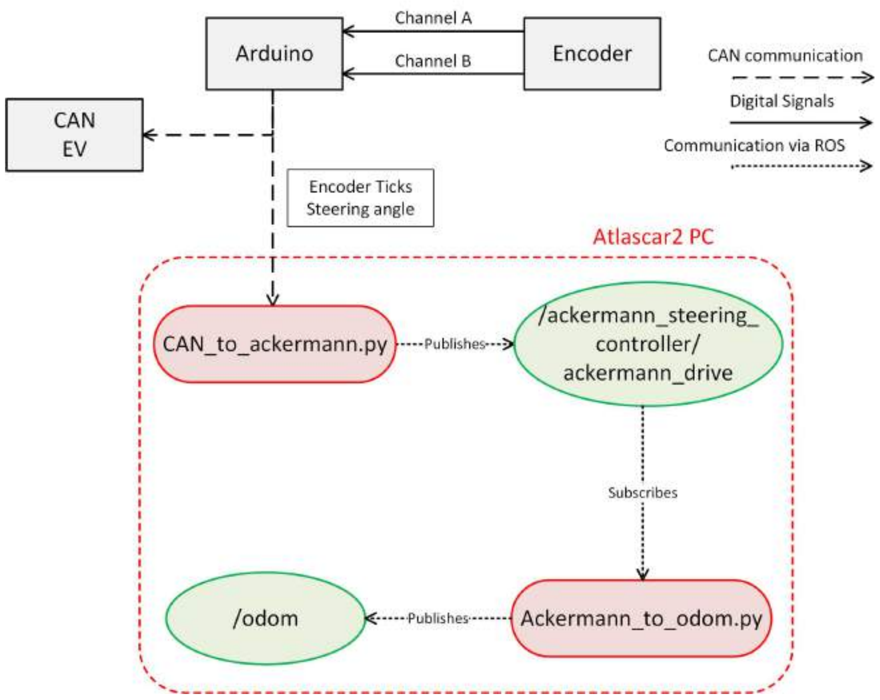


Figure 5.13: Diagram of the implemented odometry solution

5.5.2 Computing the odometry

Computing the odometry is rather straightforward, thanks to the ROS tutorials and ROS community[1]. With the speed, wheel angle, and the previous positions, we can get the current (x,y,θ) values with the expression (5.11).

$$\begin{bmatrix} x_i \\ y_i \\ \theta_i \end{bmatrix} = \begin{bmatrix} x_{i-1} + V \times \sin\theta_{i-1}\Delta t \\ y_{i-1} + V \times \cos\theta_{i-1} \times \Delta t \\ \theta_{i-1} + \omega \times \Delta t \end{bmatrix} \tag{5.11}$$

Expression (5.11) is approximated, since it approximates an arc to a straight line and disregards errors and noises, which have consequences in Section 6.2.2.

To integrate these scripts in the software architecture, a launch file was created to launch the `bringup.launch` odometry code with the name `bringup_odom.launch`. Figure 5.14 shows the rqt_graph of the `bringup_odom.launch`.

With the software architecture more organized, integrating these two nodes was easier, which with a system more unnecessarily complex could take a lot more time. The `bringup_odom.launch` shows that the `bringup` file can have a top-level file where the user can choose which algorithm is to be launched.

[1]http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom

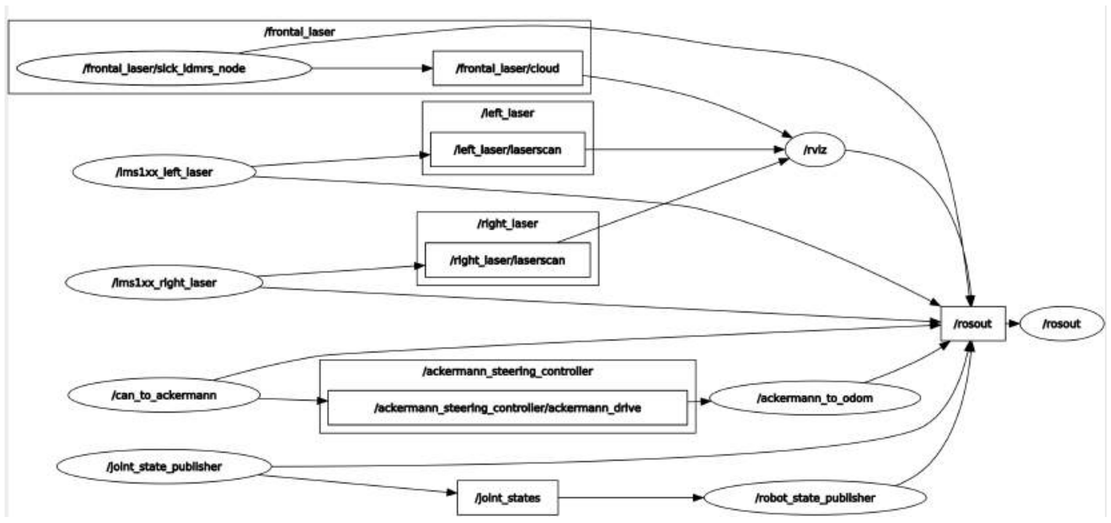


Figure 5.14: `Rqt_graph` of the `bringup_odom.launch`.

Chapter 6

Tests and Results

This chapter has two types of results: From the software architecture results and from the odometry system. The software architecture was tested following the Atlascar2's documentation especially prepared where the results can be seen in video. The odometry solution is tested using simulated data to compare the algorithm results with the ackermann controller, and with real data taken from the CAN bus, which is compared with the estimated trajectory made by the vehicle.

6.1 Software Architecture

The software architecture results cover the processing unit and the architecture performance. The processing unit is evaluated by comparing the Atlascar2's computer booting time before and after the intervention. Concerning the software architecture, although there isn't prior information about the system before, videos of the system working were taken to show its performance and ease to use.

6.1.1 Processing Unit Evaluation

At the start of this project, the boot time of the Atlascar2 was checked using the `system-analize` command. This gave the following output:

```
Startup finished in 2.079s (kernel) + 5min 17.273s (userspace) = 5min
    19.353s graphical.target reached after 5min 17.236s in userspace
```

This value is a very big time for a system to boot. The main cause for this was that the Ubuntu was trying to raise the network interfaces during those 5 minutes. To prevent that, the `networking.service` was edited, changing the `TimeoutStartSec` from 5 minutes to 5 seconds. After this step, another `system-analize` was made, which gave the output:

```
Startup finished in 2.131s (kernel) + 1min 45.568s (userspace) = 1min
    47.699s graphical.target reached after 1min 41.196s in userspace
```

Being almost 4 minutes faster. Although it is already a good value, the timeout from the start and stop jobs was also configurated in the `system.conf` file, changing the `DefaultTimeoutStartSec` from 90s to 5s. This gave the final output:

```
Startup finished in 2.124s (kernel) + 18.435s (userspace) = 20.560s
    graphical.target reached after 18.417s in userspace
```

Comparing the initial and final values, the booting time is now approximately 5 minutes faster than before. The advantage of this change is giving the user the best possible conditions to work without losing time when booting the Atlascar2's main computer.

6.1.2 Performance of the Software Architecture

To show the Software Architecture's performance, two videos were made following the Atlascar2's README file, which is included in Appendix A.

Launching the System

The improvement with the system's launch is the capability of choosing which sensor the user wants to see, not needing to comment code or change lower-level files by hand. Launching the system can be seen in the following link: `https://www.youtube.com/watch?v=dkipOMLDfOE`

Testing the Sensors

To test the sensors one of a time, although the user could only get one argument to true when launching the system, the documentation gives the possibility to use the lower level files, which gives more versatility to the system. A file was created for both the cameras and 2D LiDARs, where the users only need to change a variable to choose if they want the left or right sensor. Testing the sensors can be seen in the following link: `https://www.youtube.com/watch?v=Vudxz4I5coU`

6.2 Odometry

The odometry results can be divided into two sections: the performance in the simulation and the performance in the real vehicle. The first shows the accuracy of the odometry values using simulated data, by modifying the `ackermann_steering _controller` from the `ros_controllers` package. The second shows the algorithm performance using the Atlascar2's CAN bus data.

6.2.1 Performance with the Simulation Values

To acquire data to compare the values between the algorithm and the simulation, the user publishes a `Twist` message[1] in the `ackermann_steering_controller/cmd_vel` topic, using the robot steering from the rqt ROS tool. Receiving those values, the modified controller publishes two topics:
The `/ackermann_steering_controller/ackermann_drive` and the `/Ackermann _steering_controller/odom`. The python script subscribes to the `/ackermann_drive`

[1]https://docs.ros.org/en/diamondback/api/geometry_msgs/html/msg/Twist.html

which publishes the calculated odometry in the /odom topic, with a different transformation to avoid overlapping data. Figures 6.1 and 6.2 show the created simulation and a diagram showing the subscribed/published topics for this comparison, respectively.

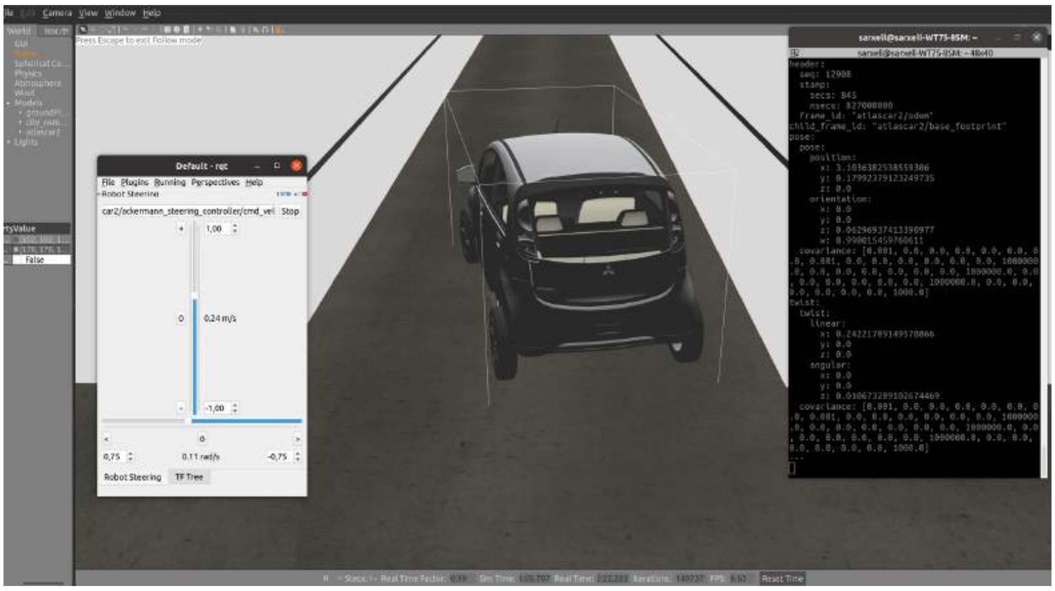

```
header:
  seq: 12908
  stamp:
    secs: 845
    nsecs: 827000000
  frame_id: "atlascar2/odom"
child_frame_id: "atlascar2/base_footprint"
pose:
  pose:
    position:
      x: 3.1036382538559386
      y: 0.17992379123249735
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.06296937413390977
      w: 0.9980154597060611
  covariance: [0.001, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.001, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1000000
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1000000.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 1000000.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 1000.0]
twist:
  twist:
    linear:
      x: 0.24221789149570866
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: 0.010673289102674469
  covariance: [0.001, 0.0, 0.0, 0.0, 0.0, 0.0, 0
.0, 0.001, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1000000
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1000000.0, 0.0
, 0.0, 0.0, 0.0, 0.0, 0.0, 1000000.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 1000.0]
---
```

Figure 6.1: Atlascar2 odometry simulation.

Both odometry topics were recorded simultaneously and performed in the same trajectory, with the car moving 30, 50 and 100 meters straight, moving in a circle and doing a random trajectory. Figure 6.3 shows the traveled route and Figures 6.4 and 6.5 show the difference between the values of the controller code and the created python script ackermann_to_odom, developed in Section 5.5.2 and included in Appendix D, during the course. The rest of the Figures from the tests can be seen in Appendix F. Rviz shows the comparison side-by-side (Figure 6.6).

To summarize, Table 6.1 shows the biggest difference between the tests' x and y values. Overall, the values do not much: the longest error is 0.023 m, and looking at the difference pictures we can notice that, although odometry usually has a cumulative error, this does not happen in the simulation. The difference is primarily due to the fact that the Ackermann controller uses integration methods such as the Runge Kutta and the exact integration method to improve the accuracy of the values, which produces the observed difference [80].

Table 6.1: Comparison between odometry values in six experiments.

30m forward		100m forward		200m forward	
x (m)	y (m)	x (m)	y (m)	x (m)	y (m)
0.0099	8.63×10^{-5}	0.0107	1.03×10^{-4}	0.0096	1.07×10^{-4}
0.20 rad/s turn		**0.75 rad/s turn**		**Random course**	
x (m)	y (m)	x (m)	y (m)	x (m)	y (m)
0.0193	0.0144	0.0229	0.0174	0.0146	0.0130

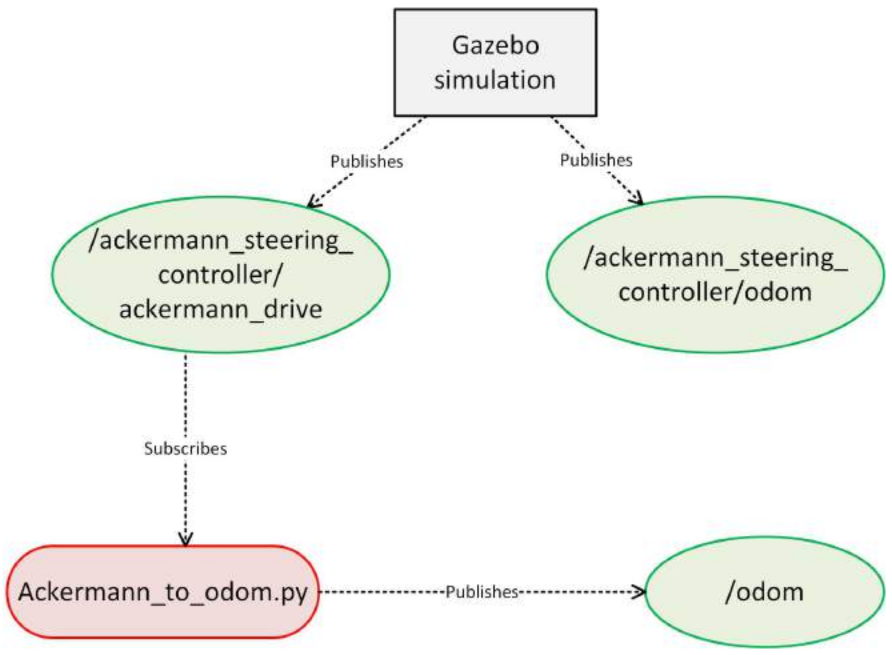

Gazebo simulation
Publishes
Publishes
/ackermann_steering_ controller/ ackermann_drive
/ackermann_steering_ controller/odom
Subscribes
Ackermann_to_odom.py
Publishes
/odom

Figure 6.2: Comparison of the odometry topics using the simulation.

6.2.2 Performance with Real Vehicle Data

The created algorithm was tested in the Crasto's parking lot (Figure 6.7), where with types of tests were performed. The first test varied velocities from 5, 10 and 15 km/h, with the same starting point and travel the same distance. The second test was travelling two paths with the vehicle: a simple path and a more complex course in the parking lot and stopping in the same spot, to check the performance of the received values.

For the first test, the distance was measured using a measuring tape on one of the parking spots. Knowing the parking space width we counted the number of spots the car travelled and got the distance. Since one parking spot measures 2.44 m and the car travelled 15 parking spots, the final distance the Atlascar2 moved was 36.6 m.

In Table 6.2, we can see that this value is inaccurate since the 2 m difference is a considerable distance in such a short course. However, these values are fairly precise because they are all approximately 38 m, which shows that varying the velocity does not influence the odometry values, at least at low speeds.

Table 6.2: Comparing travelled distances using different velocities.

Measured Distance	Vehicle Velocity	Travelled Distance
36.6 m	5 km/h 10 km/h 15 km/h	38.01 m 37.98 m 38.06 m

The second test involved travelling a path which started and ended in the same stop, to observe if the odometry values received would stop close or far away from the

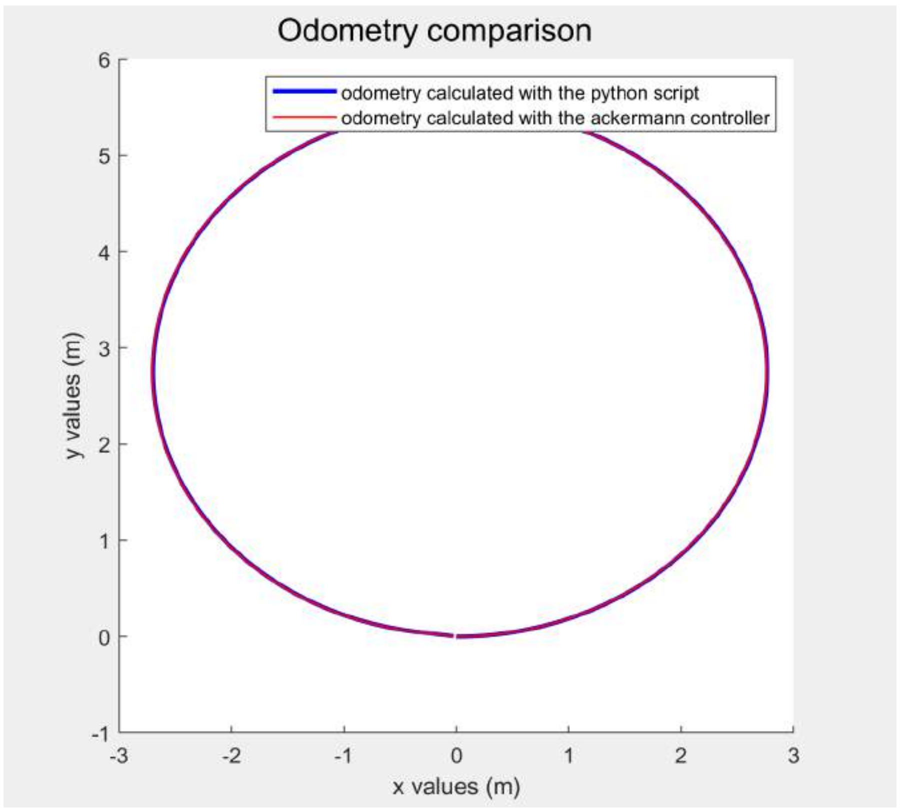


Figure 6.3: 0.75 rad/s trajectory with the ackermann controller and python script.

starting point. Figures 6.8 and 6.9 show the results from both courses. Although these values are weak, it was expected since the odometry solution is not calibrated, which consists of the identification of a set of kinematic parameters that allow reconstructing the vehicle's absolute position and orientation [81]. The green line in Figures 6.8 and 6.9 show a very simple calibration, which was possible by changing the wheel's radius and the vehicle's wheelbase from the previous values without the calibration, which are considered systematic errors. With a simple route, like the one in Figure 6.8, this calibration looks moderately good, since the vehicle gets close to the starting point. However, with more complex routes, this simple calibration no longer works. This topic is going to be discussed in Section 7, since it is a vital matter to obtain a fully functional odometry algorithm.

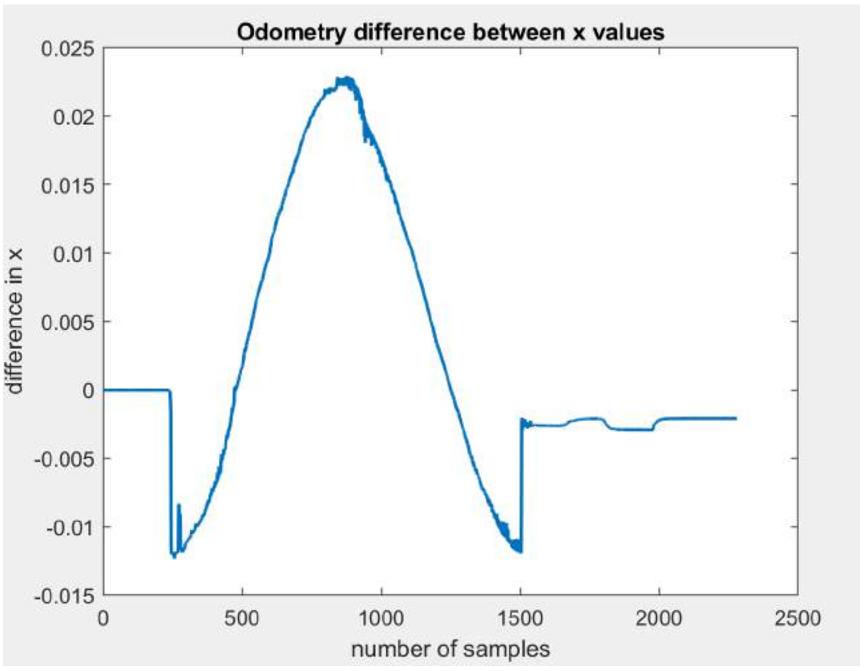


Figure 6.4: 0.75 rad/s trajectory difference of x values.

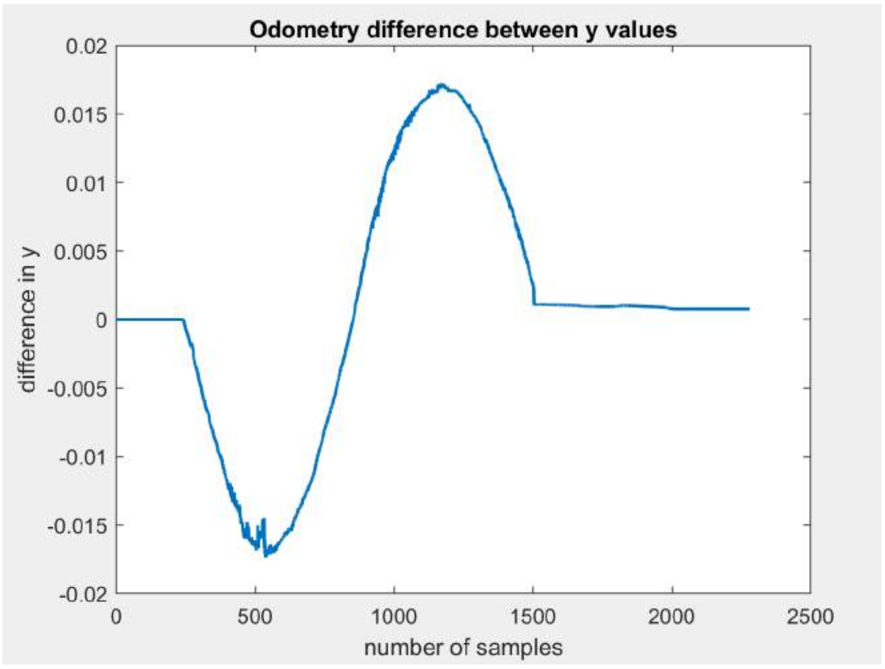
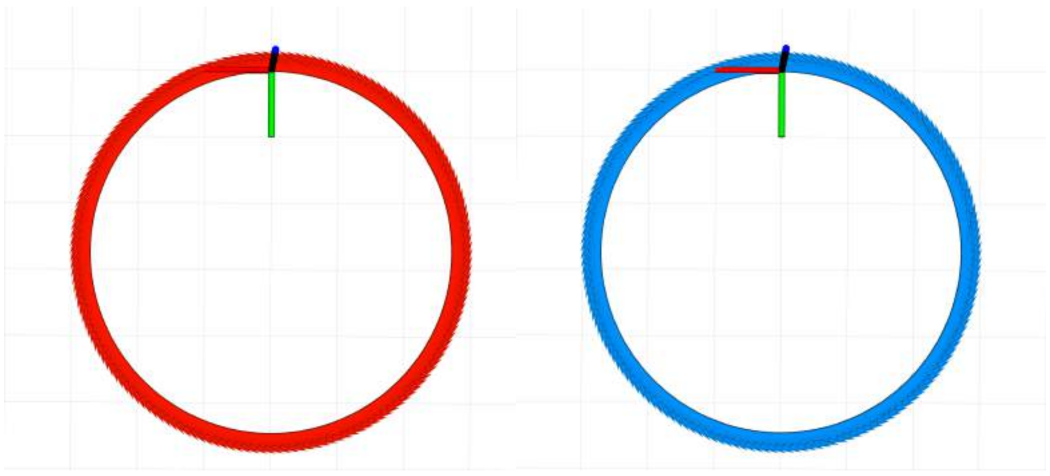


Figure 6.5: 0.75 rad/s trajectory difference of y values.



(a) Odometry calculated with the python script

(b) Odometry calculated with the ackermann controller

Figure 6.6: Comparison side-by-side of the odometry values from the controller and the python script

Figure 6.7: Atlascar2 in the Crastro's parking lot.

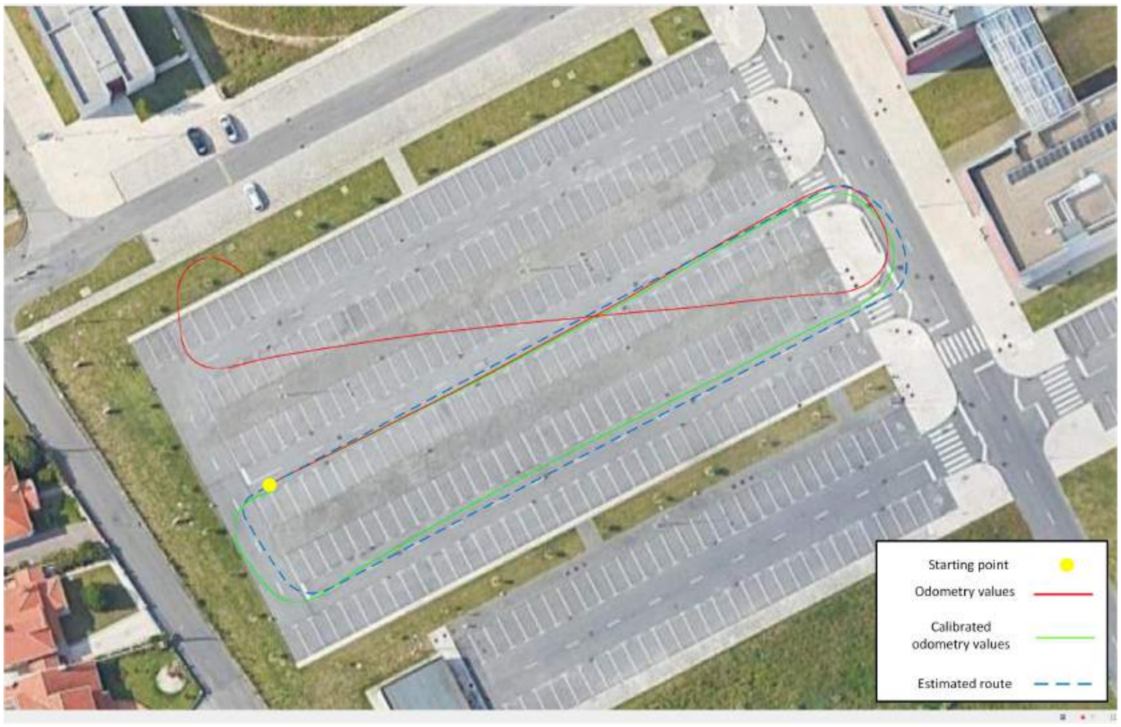


Figure 6.8: Odometry values from the Atlascar2 with the first route. Results presented using the Mapviz application.

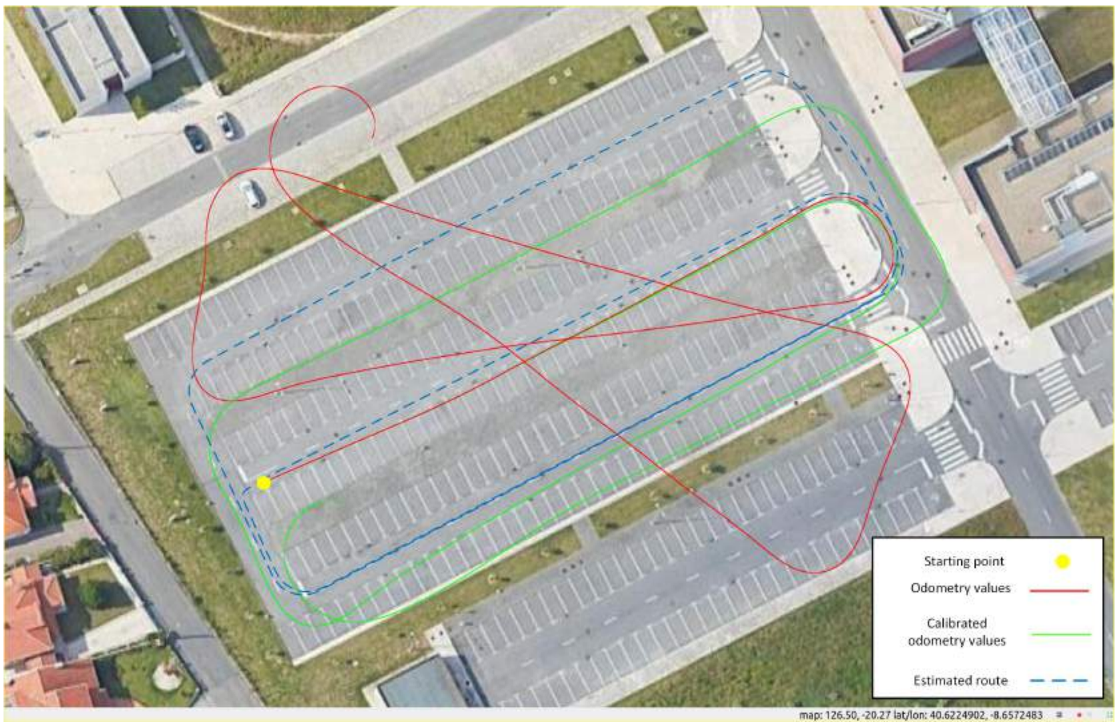

Starting point
Odometry values
Calibrated odometry values
Estimated route
map: 126.50, -20.27 lat/lon: 40.6224902, -8.6572483

Figure 6.9: Odometry values from the Atlascar2 with the second route. Results presented using the Mapviz application.

Chapter 7

Conclusions and Future Work

This chapter is a summary of the work developed in the matter of Atlascar2's software architecture and the development of an odometry solution. Based on these conclusions a few proposals for future works using Atlascar2 are presented.

7.1 Conclusions

The two primary purposes of this project were the improvement of the Atlascar2's software architecture and the development of an odometry solution.

Regarding the software architecture, this work presents an update to the previous architecture. The files necessary to launch the system are more efficient, the processing unit is faster and updated, and the required documentation was created to understand the system. The vehicle's current status is crucial for the better development of AD and ADAS projects.

In terms of developing the odometry solution, this work presents a solution that sends the needed values to the CAN Bus, which is a practical and efficient way to send the required data. The odometer installation was completed, and the solution already sends the values and can create an odometry trajectory, which can be very useful for other AD projects when adequately calibrated. The integration of the odometry was also accomplished since using ROS gave a compact solution without needing extra software and offered new ideas for future work.

Broadly, the initially defined goals have been met. This work led to an update on the Atlascar2's architecture which functions well, without errors that used to occur often. The development of the odometry solution was completed and tested, which can provide essential information for the Atlascar2 navigation module in the future.

The code and the Atlascar2's documentation can be seen in the Atlascar2 repository in `https://github.com/lardemua/atlascar2` and most of the developed work is documented in the opened and closed repository issues `https://github.com/lardemua/atlascar2/issues`.

7.2 Future Work

A wide variety of work can be done following this thesis. The camera calibration was not performed in the software architecture, which is necessary for works involving perception and navigation algorithms. Regarding the launch files, with the development of more algorithms, a new launch file or even changing the `bringup.launch` could be interesting, providing arguments where the user could choose which node to launch.

Regarding future work in the odometry solution, the essential is its calibration. The implemented odometry has two types of errors: systematic and non-systematic errors [82]. The odometry calibration can improve on the systematic errors, which are correlated with incorrect odometry parameters such as wheel misalignment, unequal wheel diameter, effective wheelbase and length between the front and rear axle [83]. For the non-systematic, also called random errors, calibration can't tackle, and so, to improve the navigation method, the odometry can be integrated with the GPS and IMU.

Bibliography

[1] Shaoshan Liu, Liyun Li, Jie Tang, Shuang Wu, and Jean-Luc Gaudiot. Creating autonomous vehicle systems. *Synthesis Lectures on Computer Science*, 6(1):i–186, 2017.

[2] SAE. Sae levels of driving automation refined for clarity and international audience. May 2021. Last accessed 23/01/2022.

[3] Synopsis. The 6 levels of vehicle autonomy explained. `https://www.synopsys.com/automotive/autonomous-driving-levels.html`, 2020. Last accessed 21/01/2022.

[4] Group of Automation and Robotics. Atlascar project. `http://atlas.web.ua.pt/index.html`, 2012.

[5] Sagar Behere and Martin Törngren. A functional reference architecture for autonomous driving. *Information and Software Technology*, 73:136–150, 2016.

[6] Alexandru Constantin Serban, Erik Poll, and Joost Visser. A standard driven software architecture for fully autonomous vehicles. In *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 120–127, April 2018.

[7] ISO 26262-1:2018. Road vehicles functional safety. `https://www.iso.org/standard/68383.html`, 12 2018.

[8] Van Chan Ngo. `https://channgo2203.github.io/av_software/`, 07 2020.

[9] Scott Pendleton, Hans Andersen, Xinxin Du, Xiaotong Shen, Malika Meghjani, You Eng, Daniela Rus, and Marcelo Jr. Perception, planning, control, and coordination for autonomous vehicles. *Machines*, 5:6, 02 2017.

[10] Tilo Linz. *Testing Autonomous Systems*, pages 61–75. Springer International Publishing, Cham, 2020.

[11] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascal Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney. *Stanley: The Robot That Won the DARPA Grand Challenge*, pages 1–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[12] Erann Gat, R. Peter Bonnasso, Robin Murphy, and Aaai Press. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press, 1997.

[13] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. *Junior: The Stanford Entry in the Urban Challenge*, pages 91–123. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[14] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, M. N. Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M. Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matt McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, William "Red" Whittaker, Ziv Wolkowicki, Jason Ziglar, Hong Bae, Thomas Brown, Daniel Demitrish, Bakhtiar Litkouhi, Jim Nickolaou, Varsha Sadekar, Wende Zhang, Joshua Struble, Michael Taylor, Michael Darms, and Dave Ferguson. *Autonomous Driving in Urban Environments: Boss and the Urban Challenge*, pages 1–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[15] Dave Ferguson, Thomas M. Howard, and Maxim Likhachev. *Motion Planning in Urban Environments*, pages 61–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[16] Christopher R. Baker, David Ferguson, and John M. Dolan. Robust mission execution for autonomous urban driving. In *Proceedings of 10th International Conference on Intelligent Autonomous Systems (IAS '08)*, pages 155 – 163, July 2008.

[17] Kichun Jo, Junsoo Kim, Dongchul Kim, Chulhoon Jang, and Myoungho Sunwoo. Development of autonomous carpart i: Distributed system architecture and development process. *IEEE Transactions on Industrial Electronics*, 61(12):7131–7140, 2014.

[18] Kichun Jo, Junsoo Kim, Dongchul Kim, Chulhoon Jang, and Myoungho Sunwoo. Development of autonomous carpart ii: A case study on the implementation of an autonomous driving system based on distributed architecture. *IEEE Transactions on Industrial Electronics*, 62(8):5119–5132, 2015.

[19] Wuwei Chen, Hansong Xiao, Qidong Wang, Linfeng Zhao, and Maofei Zhu. *Integrated vehicle dynamics and control.* John Wiley & Sons, 2016.

[20] AUTOSAR. General information about autosar. `https://www.autosar.org/about/`, 2022. Last accessed 8/02/2022.

[21] Andras Kokuti, Ahmed Hussein, Arturo de la Escalera, and Fernando Garcia. Market-based approach for cooperation and coordination among multiple autonomous vehicles. pages 534–539, 10 2017.

[22] David Martín Gómez, Pablo Marín, Ahmed Hussein, Arturo de la Escalera, and J.M. Armingol. *ROS-based Architecture for Autonomous Intelligent Campus Automobile (iCab)*, pages 257–272. 01 2016.

[23] Vitor Santos, Jorge Almeida, E. Ávila, D. Gameiro, Miguel Oliveira, R. Pascoal, R. Sabino, and Procópio Stein. Atlascar - technologies for a computer assisted driving system on board a common automobile. pages 1421 – 1427, 10 2010.

[24] M. Montemerlo, N. Roy, and S. Thrun. Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (carmen) toolkit. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, volume 3, pages 2436–2441 vol.3, 2003.

[25] Mohammad Mozaffari, Ali Broumandan, Kyle O'Keefe, and Gérard Lachapelle. Weak gps signal acquisition using antenna diversity. *NAVIGATION*, 62(3):205–218, 2015.

[26] Daniele Borio, Laura Camoriano, and Letizia Lo Presti. Impact of gps acquisition strategy on decision probabilities. *IEEE Transactions on Aerospace and Electronic Systems*, 44(3):996–1011, 2008.

[27] Tomislav Kos, Ivan Markezic, and Josip Pokrajcic. Effects of multipath reception on gps positioning performance. In *Proceedings ELMAR-2010*, pages 399–402, 2010.

[28] Shunsuke Miura, Shoma Hisaka, and Shunsuke Kamijo. Gps multipath detection and rectification using 3d maps. In *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, pages 1528–1534, 2013.

[29] Sherif A. S. Mohamed, Mohammad-Hashem Haghbayan, Tomi Westerlund, Jukka Heikkonen, Hannu Tenhunen, and Juha Plosila. A survey on odometry for autonomous navigation systems. *IEEE Access*, 7:97466–97486, 2019.

[30] Shaojiang Zhang, Yanning Guo, Qiang Zhu, and Zhiyuan Liu. Lidar-imu and wheel odometer based autonomous vehicle localization system. In *2019 Chinese Control And Decision Conference (CCDC)*, pages 4950–4955, 2019.

[31] Martin BROSSARD and Silvère BONNABEL. Learning wheel odometry and imu errors for localization. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 291–297, 2019.

[32] CHARLES PAO. What is an imu sensor? `https://www.ceva-dsp.com/ourblog/what-is-an-imu-sensor/`, 2018. Last accessed 11/02/2022.

[33] Eric B. Quist, Peter C. Niedfeldt, and Randal W. Beard. Radar odometry with recursive-ransac. *IEEE Transactions on Aerospace and Electronic Systems*, 52(4):1618–1630, 2016.

[34] Mostafa Mostafa, Shady Zahran, Adel Moussa, Naser El-Sheimy, and Abu Sesay. Radar and visual odometry integrated system aided navigation for uavs in gnss denied environment. *Sensors*, 18(9), 2018.

[35] Antonio Scannapieco, Alfredo Renga, Giancarmine Fasano, and Antonio Moccia. Experimental analysis of radar odometry by commercial ultralight radar sensor for miniaturized uas. *Journal of Intelligent Robotic Systems*, 90, 06 2018.

[36] LeddarTech. Why lidar. `https://leddartech.com/why-lidar/`, 2022. Last accessed 13/02/2022.

[37] Mohammad OA Aqel, Mohammad H Marhaban, M Iqbal Saripan, and Napsiah Bt Ismail. Review of visual odometry: types, approaches, challenges, and applications. *SpringerPlus*, 5(1):1–26, 2016.

[38] Salim Sirtkaya, Burak Seymen, and A. Aydin Alatan. Loosely coupled kalman filtering for fusion of visual odometry and inertial navigation. In *Proceedings of the 16th International Conference on Information Fusion*, pages 219–226, 2013.

[39] Giovanni Cioffi and Davide Scaramuzza. Tightly-coupled fusion of global positional measurements in optimization-based visual-inertial odometry. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5089–5095, 2020.

[40] Tiago Nunes da Rocha. Piloto automático para controlo e manobras de navegação do atlascar. Master's thesis, Universidade de Aveiro, 2011.

[41] Ricardo Luís da Mota Silva. Removable odometry unit for vehicles with ackerman steering. Master's thesis, Universidade de Aveiro, 2014.

[42] Jorge Manuel Soares de Almeida. *Active Tracking of Dynamic Multivariate Agents using Vectorial Range Data.* PhD thesis, Universidade de Aveiro, 2016.

[43] Chris Urmson, Joshua Anhalt, Daniel Bartz, Michael Clark, Tugrul Galatali, Alexander Gutierrez, Sam Harbaugh, Josh Johnston, Hiroki "Yu" Kato, Phillip Koon, William Messner, Nick Miller, Aaron Mosher, Kevin Peterson, Charlie Ragusa, David Ray, Bryon Smith, Jarrod Snider, Spencer Spiker, Josh Struble, Jason Ziglar, and William "Red" Whittaker. *A Robust Approach to High-Speed Navigation for Unrehearsed Desert Terrain*, pages 45–102. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[44] Andrew Bacha, Cheryl Bauman, Ruel Faruque, Michael Fleming, Chris Terwelp, Charles Reinholtz, Dennis Hong, Alfred Wicks, Thomas Alberi, David Anderson, Stephen Cacciola, Patrick Currier, Aaron Dalton, Jesse Farmer, Jesse Hurdus, Shawn Kimmel, Peter King, Andrew Taylor, David Covern, and Mike Webster. Odin: Team victortango's entry in the darpa urban challenge. *J. Field Robotics*, 25:467–492, 01 2008.

[45] John Leonard, Jonathan How, Seth Teller, Mitch Berger, Stefan Campbell, Gaston Fiore, Luke Fletcher, Emilio Frazzoli, Albert Huang, Sertac Karaman, Olivier Koch, Yoshiaki Kuwata, David Moore, Edwin Olson, Steve Peters, Justin Teo, Robert Truax, Matthew Walter, David Barrett, Alexander Epstein, Keoni Maheloni, Katy Moyer, Troy Jones, Ryan Buckley, Matthew Antone, Robert Galejs, Siddhartha Krishnamurthy, and Jonathan Williams. *A Perception-Driven Autonomous Urban Vehicle*, pages 163–230. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[46] Pablo Marín, Ahmed Hussein, David Martín Gómez, Fernando Garcia, Arturo de la Escalera, and J.M. Armingol. Ros-based architecture for autonomous vehicles. 11 2016.

[47] Motional. Motional home page. `https://motional.com/`, 2021. Last accessed 29/01/2022.

[48] Waymo. Waymo driver. `https://waymo.com/waymo-driver/`, 2019. Last accessed 29/01/2022.

[49] Tesla. About autopilot. `https://www.tesla.com/ownersmanual/model3/en_eu/GUID-EDA77281-42DC-4618-98A9-CC62378E0EC2.html`, 2022. Last accessed 29/01/2022.

[50] José Diogo Madureira Correia. Visual and depth perception unit for atlascar2. Master's thesis, Universidade de Aveiro, 2017.

[51] Kenny Kuchera. Canalyze - native can interface for linux. `https://kkuchera.github.io/canalyze/`, 06 2017.

[52] Automation. Point grey announces flea3 fl3-ge-28s4 2.8 megapixel digital cameras. 2020. Last accessed 20/02/2022.

[53] SICK. 2d lidar sensors lms1xx. `https://www.sick.com/ag/en/detection-and-ranging-solutions/2d-lidar-sensors/lms1xx/c/g91901`, 2022. Last accessed 19/02/2022.

[54] SICK. 3d lidar sensors ld-mrs. `https://www.sick.com/fi/en/detection-and-ranging-solutions/3d-lidar-sensors/ld-mrs/c/g91913`, 2022. Last accessed 19/02/2022.

[55] Novatel. Span-igm user manual. `https://portal.hexagon.com/public/Novatel/assets/Documents/Manuals/OM-20000141`, 2 2020.

[56] Novatel. Gps-702-gg/gps-701-gg/gps-702-gg-n antenna. `https://portal.hexagon.com/public/Novatel/assets/Documents/Manuals/om-20000095`, 5 2020.

[57] Nexus. P-1308h3/hr3. `https://nexus-solutions.pt/sistemas/servidores/servidor-nexus-p-1308h3hr3/`, 2014.

[58] M.S. Achmad, Gigih Priyandoko, R. Roali, and Mohd Daud. Tele-operated mobile robot for 3d visual inspection utilizing distributed operating system platform. *International Journal of Vehicle Structures and Systems*, 9, 09 2017.

[59] The construct. [ros in 5 mins] 025 what is rviz? `https://www.theconstructsim.com/ros-5-mins-025-rviz/`, 9 2019.

[60] Sick. Sopas engineering tool. `https://www.sick.com/cn/en/sopas-engineering-tool/p/p367244`, 2022.

[61] Teledyne Flir. Flycapture sdk. `https://www.flir.com/products/flycapture-sdk/`, 2022.

[62] Frontier Business. What is a private ip address and how can it benefit your business? https://enterprise.frontier.com/blog/What-Is-a-Private-IP-Address-and-How-Can-It-Benefit-Your-Business, 6 2018.

[63] Group of Automation and Robotics. Atlascar2 manual. https://github.com/lardemua/atlascar2, 2022.

[64] Robert Eisele. Ackerman steering. https://www.xarg.org/book/kinematics/ackerman-steering/, 2022.

[65] Michael Muter and Naim Asaj. Entropy-based anomaly detection for in-vehicle networks. *IEEE Intelligent Vehicles Symposium, Proceedings*, pages 1110–1115, 06 2011.

[66] Zhenwang Li and Shen. Anomaly detection of can bus messages using a deep neural network for autonomous vehicles. *Applied Sciences*, 9:3174, 08 2019.

[67] Martin Falch. Can bus explained - a simple intro [2022]. https://www.csselectronics.com/pages/can-bus-simple-intro-tutorial, 4 2022.

[68] Staff Writer. What is can bus protocol? https://www.totalphase.com/blog/2019/08/5-advantages-of-can-bus-protocol/, 9 2020.

[69] Pritt Laes. Mitsubishi i-miev obd-ii pid documentation. https://github.com/plaes/i-miev-obd2, 03 2013.

[70] Mitsubishi I-Miev Forum. Decyphering imiev and ion car-can message data. http://myimiev.com/forum/viewtopic.php?f=25&t=763&hilit=send+messages+to+can/, 05 2013.

[71] Diogo Augusto Rodrigues de Figueiredo. Remote control for operation and driving of atlascar2. Master's thesis, Universidade de Aveiro, 2020.

[72] Luís Cristovão. Interface obd para o atlascar2 e monitorização do seu estado. Technical report, Universidade de Aveiro, 2018.

[73] Farnell. Ri32-0/1000er.14kb. https://pt.farnell.com/hengstler/ri32-0-1000er-14kb/encoder-rotary/dp/615985, 2022.

[74] Armindo Silva. Inclinómetro planar de precisão para o atlascar-2. Technical report, Universidade de Aveiro, 6 2016.

[75] ARDUINO. Arduino uno wifi rev2. https://store.arduino.cc/products/arduino-uno-wifi-rev2, 2021.

[76] botnroll. Can-bus shield v2.0. https://www.botnroll.com/pt/arduinos/2470-can-bus-shield-v20.html, 2022.

[77] Arduino Community. Max interrupts / second. https://forum.arduino.cc/t/max-interrupts-second/357256/8, 01 2016.

[78] Majenko. Arduino uno's timer maximum frequency using timer compare interrupt, not timer output pins (e.g. oc0a). `https://arduino.stackexchange.com/questions/83426/arduino-unos-timer-maximum-frequency-using-timer-compare-interrupt-not-tim` 04 2021.

[79] Car and Driver. 2012 mitsubishi i-miev se 4dr hb features and specs. `https://www.caranddriver.com/mitsubishi/i-miev/specs/2012/mitsubishi_i-miev_mitsubishi-i-miev_2012`, 2016.

[80] GIANNI A. DI CARO. Lecture 8: Kinematics equations odometry, dead reckoning. `https://web2.qatar.cmu.edu/~gdicaro/16311-Fall17/slides/16311-8-Kinematics-DeadReckoning.pdf`.

[81] G. Antonelli, S. Chiaverini, and G. Fusco. A calibration method for odometry of mobile robots based on the least-squares technique: theory and experimental validation. *IEEE Transactions on Robotics*, 21(5):994–1004, 2005.

[82] Johann Borenstein, Hobart R. Everett, and Liqiang Feng. Where am i?" sensors and methods for mobile robot positioning. 1996.

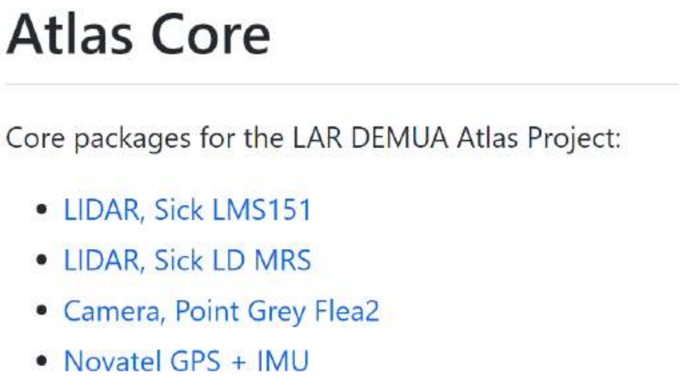
[83] Kooktae Lee and Woojin Chung. Calibration of kinematic parameters of a car-like mobile robot to improve odometry accuracy. In *2008 IEEE International Conference on Robotics and Automation*, pages 2546–2551, 2008.

Appendix A

Project's README information

A.1 Core packages

The core packages presents the essential links for the drivers of this project.



Atlas Core

Core packages for the LAR DEMUA Atlas Project:

- LIDAR, Sick LMS151
- LIDAR, Sick LD MRS
- Camera, Point Grey Flea2
- Novatel GPS + IMU

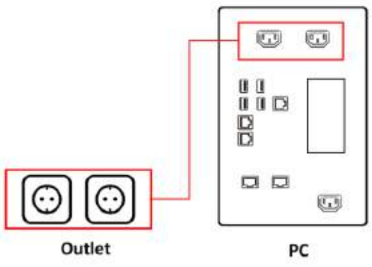
Figure A.1: Core packages for the Atlascar2

A.2 Turning on the vehicle and its components

Shows the necessary steps to turn on the vehicle and its components.

1: Turning ON everything you need

- **Step 1**: Turn on the car.
- **Case 1**: When the car is stopped:
  - **Step 2**: Connect the atlas machine to an outlet near the car.



  - **Step 3**: Turn on the atlas computer.
  - **Step 4**: Plug the ethernet cable (the cable is outside the atlascar2) to the atlas computer (on the figure port enp5s0f1).

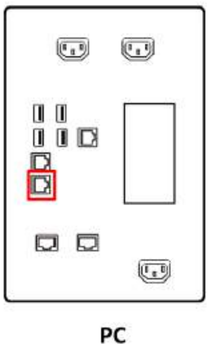
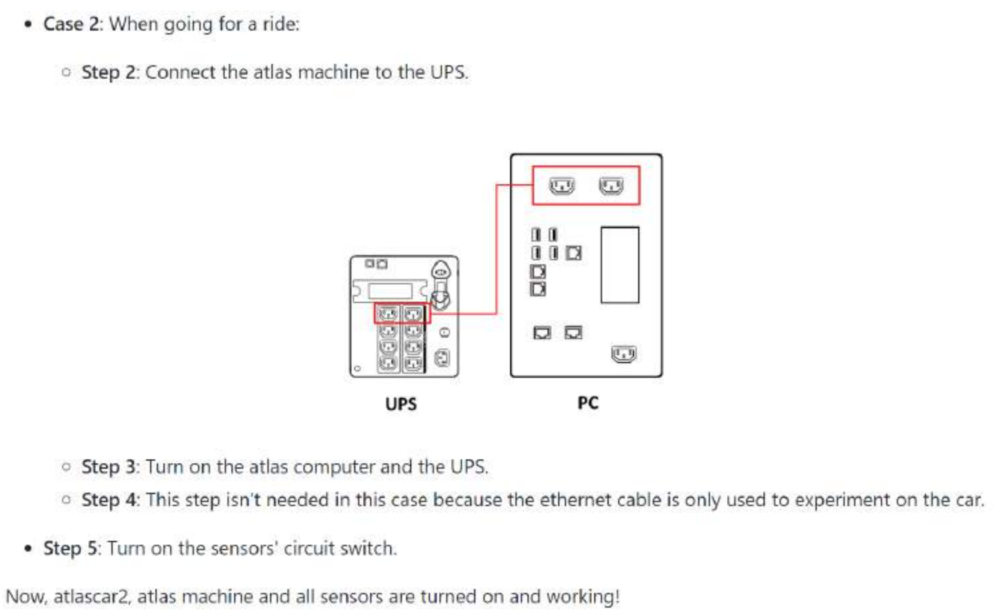


Figure A.2: Steps to turn on the vehicle



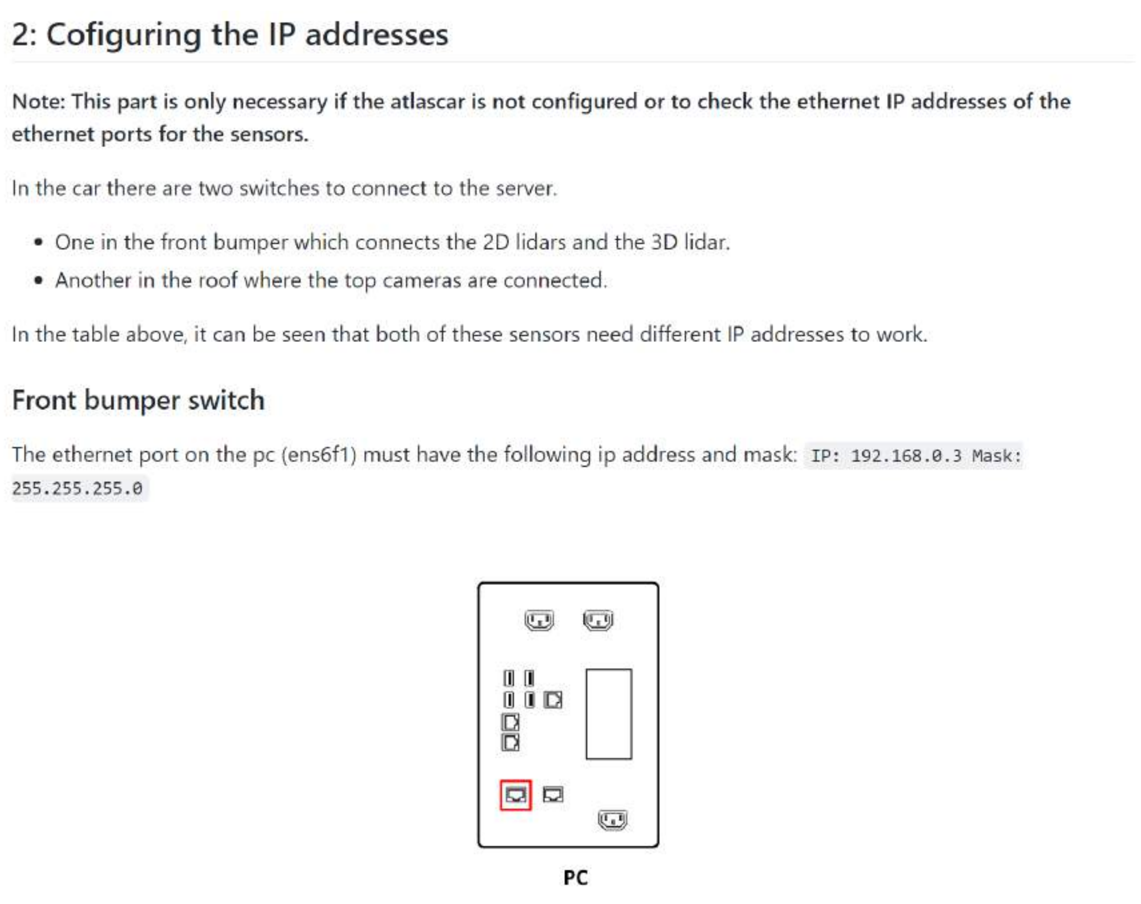
- **Case 2**: When going for a ride:
  - **Step 2**: Connect the atlas machine to the UPS.
  - **Step 3**: Turn on the atlas computer and the UPS.
  - **Step 4**: This step isn't needed in this case because the ethernet cable is only used to experiment on the car.
- **Step 5**: Turn on the sensors' circuit switch.

Now, atlascar2, atlas machine and all sensors are turned on and working!

Figure A.3: Steps to turn on the vehicle

A.3 Configuring the IP addresses

Check if the static addresses and the ethernet cables are connected.



2: Cofiguring the IP addresses

Note: This part is only necessary if the atlascar is not configured or to check the ethernet IP addresses of the ethernet ports for the sensors.

In the car there are two switches to connect to the server.

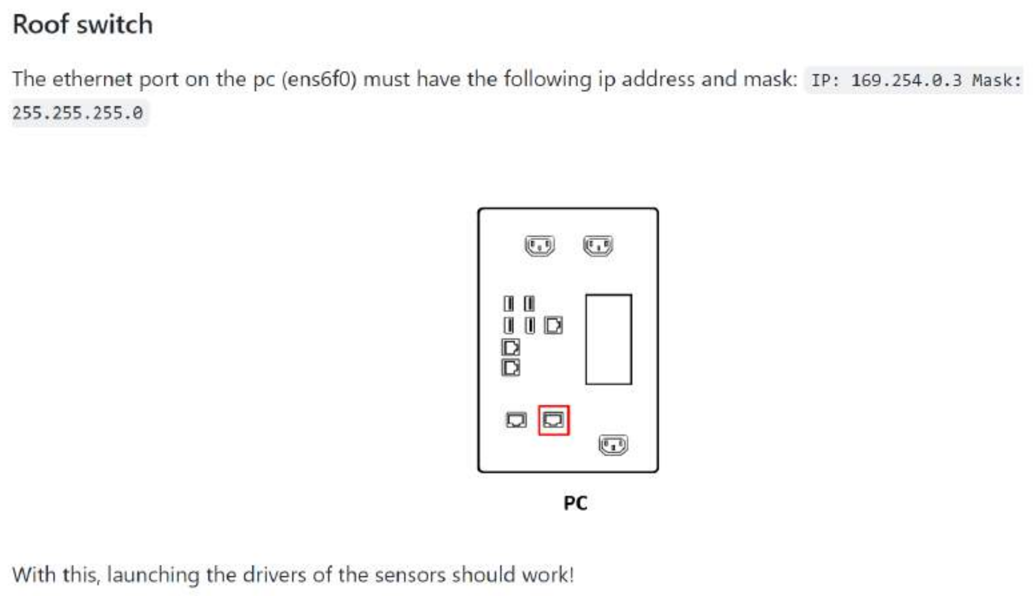
- One in the front bumper which connects the 2D lidars and the 3D lidar.
- Another in the roof where the top cameras are connected.

In the table above, it can be seen that both of these sensors need different IP addresses to work.

Front bumper switch

The ethernet port on the pc (ens6f1) must have the following ip address and mask: `IP: 192.168.0.3 Mask: 255.255.255.0`

Figure A.4: Configuring IP addresses of the switches



Roof switch

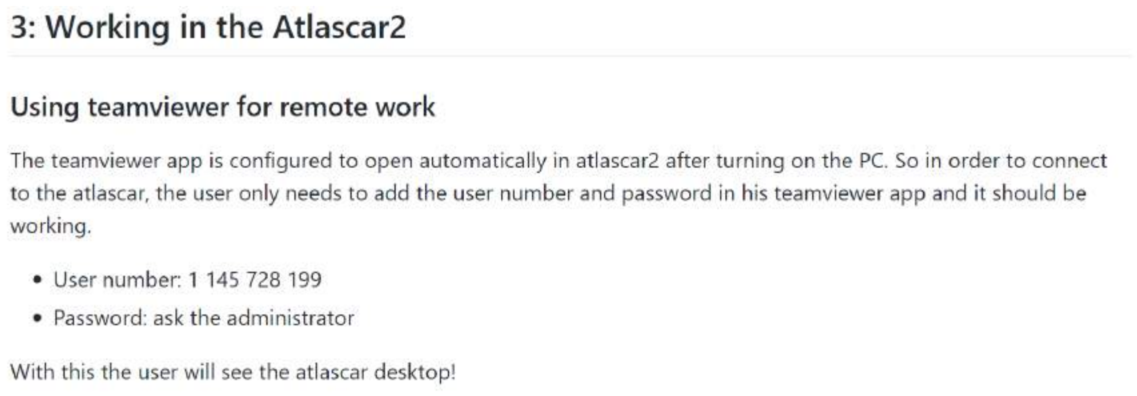
The ethernet port on the pc (ens6f0) must have the following ip address and mask: `IP: 169.254.0.3 Mask: 255.255.255.0`

With this, launching the drivers of the sensors should work!

Figure A.5: Configuring IP addresses of the switches

A.4 Connecting to the Atlascar2 with remote work

Using the teamwork viewer to use the atlascar2 with remote work.



3: Working in the Atlascar2

Using teamviewer for remote work

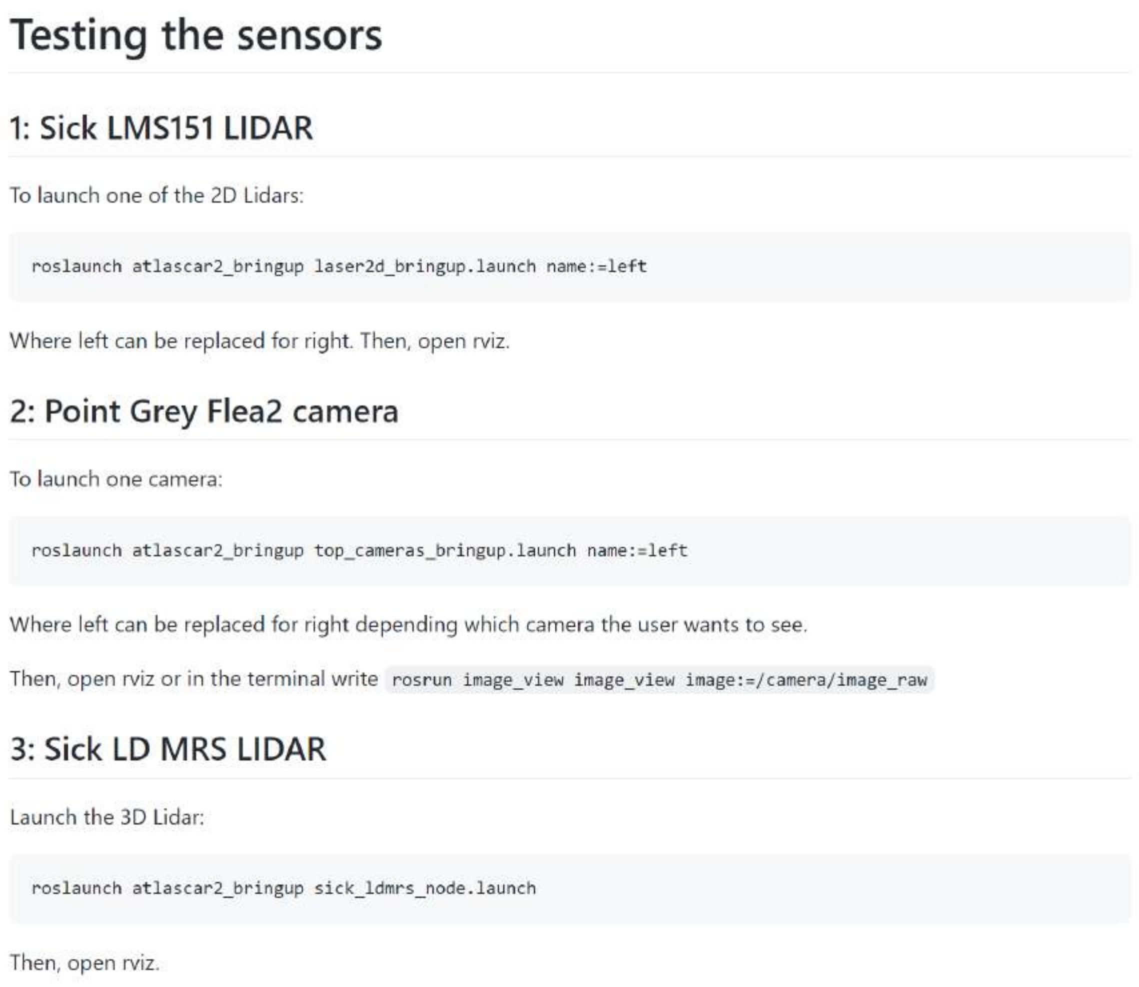
The teamviewer app is configured to open automatically in atlascar2 after turning on the PC. So in order to connect to the atlascar, the user only needs to add the user number and password in his teamviewer app and it should be working.

- User number: 1 145 728 199
- Password: ask the administrator

With this the user will see the atlascar desktop!

Figure A.6: Remote work configuration

A.5 Testing the sensors



Testing the sensors

1: Sick LMS151 LIDAR

To launch one of the 2D Lidars:

```
roslaunch atlascar2_bringup laser2d_bringup.launch name:=left
```

Where left can be replaced for right. Then, open rviz.

2: Point Grey Flea2 camera

To launch one camera:

```
roslaunch atlascar2_bringup top_cameras_bringup.launch name:=left
```

Where left can be replaced for right depending which camera the user wants to see.

Then, open rviz or in the terminal write `rosrun image_view image_view image:=/camera/image_raw`

3: Sick LD MRS LIDAR

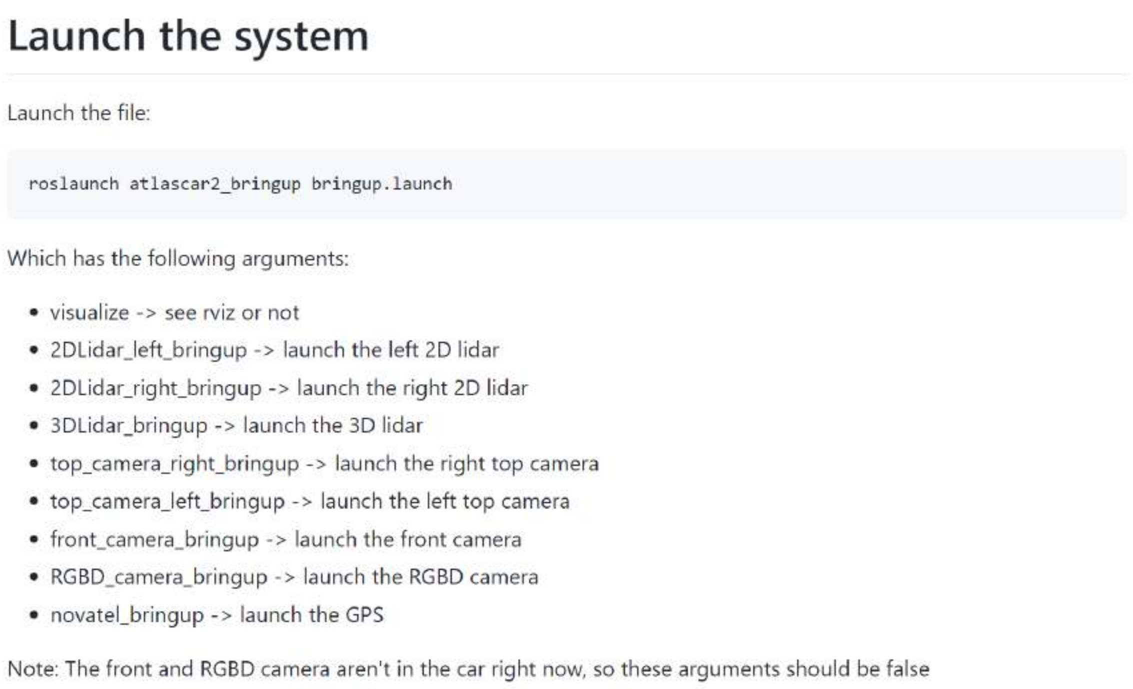
Launch the 3D Lidar:

```
roslaunch atlascar2_bringup sick_ldmrs_node.launch
```

Then, open rviz.

Figure A.7: Testing the sensors

A.6 Launching the system



Launch the system

Launch the file:

```
roslaunch atlascar2_bringup bringup.launch
```

Which has the following arguments:

- visualize -> see rviz or not
- 2DLidar_left_bringup -> launch the left 2D lidar
- 2DLidar_right_bringup -> launch the right 2D lidar
- 3DLidar_bringup -> launch the 3D lidar
- top_camera_right_bringup -> launch the right top camera
- top_camera_left_bringup -> launch the left top camera
- front_camera_bringup -> launch the front camera
- RGBD_camera_bringup -> launch the RGBD camera
- novatel_bringup -> launch the GPS

Note: The front and RGBD camera aren't in the car right now, so these arguments should be false

Figure A.8: Launching the system

Appendix B

Arduino IDE code

Listing B.1: Arduino code

```
#include <SPI.h>

#define encoder0PinA 5
#define encoder0PinB 3
#define PI 3.1415926535897932384626433832795
#define CAN_2515
// #define CAN_2518FD

// Set SPI CS Pin according to your hardware
// For Arduino MCP2515 Hat:
// the cs pin of the version after v1.1 is default to D9
// v0.9b and v1.0 is default D10
const int SPI_CS_PIN = 9;
const int CAN_INT_PIN = 2;

#ifdef CAN_2515
#include "mcp2515_can.h"
mcp2515_can CAN(SPI_CS_PIN); // Set CS pin
#endif


volatile long encoder0Pos=0;
volatile long newposition;
volatile long oldposition = 0;
long newtime;
long oldtime = 0;


void setup() { //Setup runs once//

 pinMode(encoder0PinA, INPUT);
 pinMode(encoder0PinB, INPUT);
```

```
  digitalWrite(encoder0PinA, HIGH);
  digitalWrite(encoder0PinB, HIGH);
  // checking the four pulses from the encoder
  attachInterrupt(encoder0PinA, doEncoderA, CHANGE); //Interrupt
      trigger mode: RISING
  attachInterrupt(encoder0PinB, doEncoderB, CHANGE); //Interrupt
      trigger mode: RISING

// connection to the CAN bus
  SERIAL_PORT_MONITOR.begin(115200);
  while(!Serial){};
//
  while (CAN_OK != CAN.begin(CAN_500KBPS)) { // init can bus :
     baudrate = 500k
     SERIAL_PORT_MONITOR.println("CAN init fail, retry...");
     delay(100);
  }
  SERIAL_PORT_MONITOR.println("CAN init ok!");
}


byte signed stmp[8] = {0, 0, 0, 0, 0, 0, 0, 0};
void loop() { //Loop runs forever//

   newposition = encoder0Pos;
   newtime = millis();
   if (newtime - oldtime >= 10) {
     // encoder ticks
     SERIAL_PORT_MONITOR.print ("position = ");
     SERIAL_PORT_MONITOR.println (newposition);
     oldposition = newposition;
     oldtime = newtime;

     // Encoder ticks to bytes
     stmp[0] = (newposition >> 56);
     stmp[1] = (newposition >> 48);
     stmp[2] = (newposition >> 40);
     stmp[3] = (newposition >> 32);
     stmp[4] = (newposition >> 24);
     stmp[5] = (newposition >> 16);
     stmp[6] = (newposition >> 8);
     stmp[7] = newposition;

     // prints values
     long newLong = (stmp[4] << 24) | (stmp[5] << 16) | (stmp[6]
         << 8) | (stmp[7]);
     Serial.println(newLong);
```

```
    Serial.print(stmp[4],HEX);
    Serial.print(" ");
    Serial.print(stmp[5],HEX);
    Serial.print(" ");
    Serial.print(stmp[6],HEX);
    Serial.print(" ");
    Serial.println(stmp[7],HEX);
    // sends value to the CAN bus
    CAN.sendMsgBuf(0x500, 0, 8,stmp);
    SERIAL_PORT_MONITOR.println("CAN BUS sendMsgBuf ok!");
   }
}

void doEncoderA()
{
  if (digitalRead(encoder0PinA) != digitalRead(encoder0PinB)) {
    encoder0Pos++;
  } else {
    encoder0Pos--;
  }
}

void doEncoderB()
{
  if (digitalRead(encoder0PinA) == digitalRead(encoder0PinB)) {
    encoder0Pos++;
  } else {
    encoder0Pos--;
  }
}0
```

Appendix C

CAN messages to ackermann values program

Listing C.1: `CANmsgs_to_ackermann` script

```python

#!/usr/bin/env python

import rospy
import can
from can.bus import BusState
from ackermann_msgs.msg import AckermannDriveStamped
import math

ack_pub = rospy.Publisher('ackermann_steering_controller/ackermann_drive',
    AckermannDriveStamped, queue_size=10)


def receive_all():
   """Receives the steering angle and encoder tick messages"""
   global ack_pub
   steering_angle = 0
   steer_velocity = 0
   speed = 0
   ackMsg = AckermannDriveStamped()
   oldposition = 0
   maxPPR = 1000
   wheelbase = rospy.get_param('~wheelbase', 2.55)
   wheel_radius = 0.285
   oldtime_pos = rospy.Time.now()

   with can.interface.Bus(bustype="socketcan", channel="can0", bitrate
       =500000) as bus:
      # filter all the messages and only let's the 0x500 and 0x236 message ID
      bus.set_filters([{"can_id": 0x500, "can_mask": 0x530}, {"can_id": 0x236
          , "can_mask": 0x237}])
      while not rospy.is_shutdown():
         # receives the message
         msg = bus.recv(1)
         # print(msg)
         if msg is not None:
            if msg.arbitration_id == 0x500:
               # gets the encoder ticks
               newposition = int.from_bytes(msg.data, "big", signed=True)
               newtime_pos = rospy.Time.now()
               if newtime_pos.to_sec() - oldtime_pos.to_sec() < 0.1:
                  continue
               # calculates the speed in m/s
               frequency = (newposition - oldposition) / (newtime_pos.to_sec() - oldtime_pos.to_sec())
               rps = frequency / maxPPR
               speed = (rps * math.pi * wheel_radius * 2)
               ackMsg.header.stamp = rospy.Time.now()
               ackMsg.header.frame_id = "atlascar2/ackermann_msgs"
               ackMsg.drive.speed = speed
               ackMsg.drive.steering_angle_velocity = steer_velocity
               ackMsg.drive.steering_angle = steering_angle
               # print(f"newposition: {newposition} , oldposition: {oldposition}")
               print(speed, steer_velocity, steering_angle)
               oldposition = newposition
               oldtime_pos = newtime_pos
               ack_pub.publish(ackMsg)

            if msg.arbitration_id == 0x236:
               # to get the steering angle its the following formula:
               # ((B0*256 + B1) -4096)/2
               # to get the wheel angle : divide the formula by 16.06
               steering_angle = ((msg.data[0] * 256 + msg.data[1]) - 4096) / (2 * 16.06)
               steering_angle = (math.pi * steering_angle) / 180
               # calculate the angular velocity
               steer_velocity = math.tan(steering_angle)*(speed/wheelbase)
               ackMsg.header.stamp = rospy.Time.now()
               ackMsg.header.frame_id = "atlascar2/ackermann_msgs"
               ackMsg.drive.speed = speed
               ackMsg.drive.steering_angle_velocity = steer_velocity
               ackMsg.drive.steering_angle = steering_angle
               print(speed, steer_velocity, steering_angle)
               ack_pub.publish(ackMsg)


def main():
   rospy.init_node('ackermann_publisher')
   receive_all()
   rospy.spin()


if __name__ == '__main__':
   main()
```

Appendix D

Ackermann to Odometry script

Listing D.1: `ackermann_to_odom` script

```

#!/usr/bin/env python

import math
from math import sin, cos, pi

import rospy
import tf
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Point, Pose, Quaternion, Twist, Vector3
from ackermann_msgs.msg import AckermannDriveStamped


def odom_callback(data):
   global last_time, current_time, x, y, th, vx, vy, vth, wheelbase,
       covariance_twist, covariance_pose
   global odom_pub, odom_broadcaster
   current_time = rospy.Time.now()

   # compute odometry in a typical way given the velocities of the robot
   dt = (current_time - last_time).to_sec()
   delta_x = vx * cos(th) * dt
   delta_y = vx * sin(th) * dt
   delta_th = vth * dt
   x += delta_x
   y += delta_y
   th += delta_th


   # vx of the vehicle is the speed and vth is the angular velocity
   vx = data.drive.speed
   vy = 0.0
   vth = data.drive.steering_angle_velocity

   odomMsg = Odometry()
   odomMsg.header.stamp = rospy.Time.now()
   odomMsg.twist.twist.linear.x = vx
   odomMsg.twist.twist.linear.y = vy
   odomMsg.twist.twist.angular.z = vth
```

```
    odomMsg.twist.covariance = covariance_twist

    odomMsg.header.frame_id = 'atlascar2/odom'
    odomMsg.child_frame_id = 'atlascar2/base_footprint'

    # since all odometry is 6DOF we'll need a quaternion created from yaw
    odom_quat = tf.transformations.quaternion_from_euler(0, 0, th)

    # first, we'll publish the transform over tf
    odom_broadcaster.sendTransform(
        (x, y, 0.),
        odom_quat,
        current_time,
        "atlascar2/base_footprint",
        "atlascar2/odom"
    )

    # set the position
    odomMsg.pose.pose = Pose(Point(x, y, 0.), Quaternion(*odom_quat))
    odomMsg.pose.covariance = covariance_pose

    # publish the message
    odom_pub.publish(odomMsg)
    last_time = current_time


def main():
    global last_time, current_time, x, y, th, vx, vy, vth, wheelbase,
        covariance_twist, covariance_pose
    global odom_pub, odom_broadcaster, twist_pub

    rospy.init_node('odometry_publisher')
    odom_pub = rospy.Publisher("atlascar2/odom", Odometry, queue_size=10)
    odom_broadcaster = tf.TransformBroadcaster()
    wheelbase = rospy.get_param('~wheelbase', 2.55)
    current_time = rospy.Time.now()
    last_time = rospy.Time.now()

    covariance_twist = [0.001, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.001, 0.0, 0.0,
        0.0, 0.0, 0.0, 0.0, 1000000.0, 0.0, 0.0,
                    0.0, 0.0, 0.0, 0.0, 1000000.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                        0.0, 1000000.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                    0.0, 1000.0]
    covariance_pose = [0.001, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.001, 0.0, 0.0,
        0.0, 0.0, 0.0, 0.0, 1000000.0, 0.0, 0.0, 0.0,
                   0.0, 0.0, 0.0, 1000000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                       1000000.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                   1000.0]

    # subscribe to the ackermann messages
    rospy.Subscriber('ackermann_steering_controller/ackermann_drive',
        AckermannDriveStamped, odom_callback, queue_size=10)
    x = 0.0
    y = 0.0
    th = 0.0

    vx = 0.0
    vy = 0.0
    vth = 0.0

    rospy.spin()


if __name__ == '__main__':
    main()
```

Appendix E

Modified Ackermann controller

Listing E.1: `ackermann_steering_controller` script

```

/*********************************************************************
 * Software License Agreement (BSD License)
 *
 * Copyright (c) 2013, PAL Robotics, S.L.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * * Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above
 * copyright notice, this list of conditions and the following
 * disclaimer in the documentation and/or other materials provided
 * with the distribution.
 * * Neither the name of the PAL Robotics nor the names of its
 * contributors may be used to endorse or promote products derived
 * from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 * FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
 * COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
 * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *********************************************************************/

/*
 * Author: Masaru Morita, Bence Magyar, Enrique Fernández
 */

```

```
#include <cmath>
#include <pluginlib/class_list_macros.h>
#include <tf/transform_datatypes.h>
#include <urdf_parser/urdf_parser.h>

#include <ackermann_steering_controller/ackermann_steering_controller.h>

static double euclideanOfVectors(const urdf::Vector3& vec1, const urdf::
    Vector3& vec2)
{
  return std::sqrt(std::pow(vec1.x-vec2.x,2) +
                   std::pow(vec1.y-vec2.y,2) +
                   std::pow(vec1.z-vec2.z,2));
}

/*
 * \brief Check if the link is modeled as a cylinder
 * \param link Link
 * \return true if the link is modeled as a Cylinder; false otherwise
 */
static bool isCylinder(const urdf::LinkConstSharedPtr& link)
{
  if (!link)
  {
    ROS_ERROR("Link pointer is null.");
    return false;
  }

  if (!link->collision)
  {
    ROS_ERROR_STREAM("Link " << link->name << " does not have collision
        description. Add collision description for link to urdf.");
    return false;
  }

  if (!link->collision->geometry)
  {
    ROS_ERROR_STREAM("Link " << link->name << " does not have collision
        geometry description. Add collision geometry description for link to
        urdf.");
    return false;
  }

  if (link->collision->geometry->type != urdf::Geometry::CYLINDER)
  {
    ROS_ERROR_STREAM("Link " << link->name << " does not have cylinder
        geometry");
    return false;
  }

  return true;
}

/*
 * \brief Get the wheel radius
 * \param [in] wheel_link Wheel link
 * \param [out] wheel_radius Wheel radius [m]
 * \return true if the wheel radius was found; false other
```

```
wise
 */
static bool getWheelRadius(const urdf::LinkConstSharedPtr& wheel_link, double
    & wheel_radius)
{
  if (!isCylinder(wheel_link))
  {
    ROS_ERROR_STREAM("Wheel link " << wheel_link->name << " is NOT modeled as
        a cylinder!");
    return false;
  }

  wheel_radius = (static_cast<urdf::Cylinder*>(wheel_link->collision->
      geometry.get()))->radius;
  return true;
}

namespace ackermann_steering_controller{

  AckermannSteeringController::AckermannSteeringController()
    : open_loop_(false)
    , command_struct_()
    , wheel_separation_h_(0.0)
    , wheel_radius_(0.0)
    , wheel_separation_h_multiplier_(1.0)
    , wheel_radius_multiplier_(1.0)
    , steer_pos_multiplier_(1.0)
    , cmd_vel_timeout_(0.5)
    , allow_multiple_cmd_vel_publishers_(true)
    , base_frame_id_("base_link")
    , odom_frame_id_("odom")
    , enable_odom_tf_(true)
    , wheel_joints_size_(0)
    , publish_ackermann_drive_(false)
  {
  }

  bool AckermannSteeringController::init(hardware_interface::RobotHW*
      robot_hw,
                           ros::NodeHandle& root_nh,
                           ros::NodeHandle& controller_nh)
  {
    typedef hardware_interface::VelocityJointInterface VelIface;
    typedef hardware_interface::PositionJointInterface PosIface;
    typedef hardware_interface::JointStateInterface StateIface;

    // get multiple types of hardware_interface
    VelIface *vel_joint_if = robot_hw->get<VelIface>(); // vel for wheels
    PosIface *pos_joint_if = robot_hw->get<PosIface>(); // pos for steers

    const std::string complete_ns = controller_nh.getNamespace();

    std::size_t id = complete_ns.find_last_of("/");
    name_ = complete_ns.substr(id + 1);

    //-- single rear wheel joint
    std::string rear_wheel_name = "rear_wheel_joint";
    controller_nh.param("rear_wheel", rear_wheel_name, rear_wheel_name);

    //-- single front steer joint
    std::string front_steer_name = "front_steer_joint";
    controller_nh.param("front_steer", front_steer_name, front_steer_name);

    // Publish ackermannDrive message (speed and steering angle of the robot
    controller_nh.param("publish_ackermann_drive", publish_ackermann_drive_,
        publish_ackermann_drive_);

    // resets the ackermannDrive message
    if (publish_ackermann_drive_)
    {
      cmd_ackermann_drive_pub_.reset(new realtime_tools::RealtimePublisher<
          ackermann_msgs::AckermannDriveStamped>(controller_nh, "
          ackermann_drive", 100));
    }

    // Odometry related:
    double publish_rate;
    controller_nh.param("publish_rate", publish_rate, 50.0);
    ROS_INFO_STREAM_NAMED(name_, "Controller state will be published at "
                          << publish_rate << "Hz.");
    publish_period_ = ros::Duration(1.0 / publish_rate);

    controller_nh.param("open_loop", open_loop_, open_loop_);

    controller_nh.param("wheel_separation_h_multiplier",
        wheel_separation_h_multiplier_, wheel_separation_h_multiplier_);
    ROS_INFO_STREAM_NAMED(name_, "Wheel separation height will be multiplied
        by "
                          << wheel_separation_h_multiplier_ << ".");

    controller_nh.param("wheel_radius_multiplier", wheel_radius_multiplier_,
        wheel_radius_multiplier_);
    ROS_INFO_STREAM_NAMED(name_, "Wheel radius will be multiplied by "
                          << wheel_radius_multiplier_ << ".");

    controller_nh.param("steer_pos_multiplier", steer_pos_multiplier_,
        steer_pos_multiplier_);
    ROS_INFO_STREAM_NAMED(name_, "Steer pos will be multiplied by "
                          << steer_pos_multiplier_ << ".");

    int velocity_rolling_window_size = 10;
    controller_nh.param("velocity_rolling_window_size",
        velocity_rolling_window_size, velocity_rolling_window_size);
    ROS_INFO_STREAM_NAMED(name_, "Velocity rolling window size of "
                          << velocity_rolling_window_size << ".");

    odometry_.setVelocityRollingWindowSize(velocity_rolling_window_size);

    // Twist command related:
    controller_nh.param("cmd_vel_timeout", cmd_vel_timeout_, cmd_vel_timeout_)
        ;
    ROS_INFO_STREAM_NAMED(name_, "Velocity commands will be considered old if
        they are older than "
                          << cmd_vel_timeout_ << "s.");

    controller_nh.param("allow_multiple_cmd_vel_publishers",
    allow_multiple_cmd_vel_publishers_, allow_multiple_cmd_vel_publishers_);
    ROS_INFO_STREAM_NAMED(name_, "Allow mutiple cmd_vel publishers is "
                    << (allow_multiple_cmd_vel_publishers_?"enabled":"disabled"));

    controller_nh.param("base_frame_id", base_frame_id_, base_frame_id_);
    ROS_INFO_STREAM_NAMED(name_, "Base frame_id set to " << base_frame_id_);

    controller_nh.param("odom_frame_id", odom_frame_id_, odom_frame_id_);
    ROS_INFO_STREAM_NAMED(name_, "Odometry frame_id set to " << odom_frame_id_);

    controller_nh.param("enable_odom_tf", enable_odom_tf_, enable_odom_tf_);
    ROS_INFO_STREAM_NAMED(name_, "Publishing to tf is " << (enable_odom_tf_?"enabled":"disabled"));

    // Velocity and acceleration limits:
    controller_nh.param("linear/x/has_velocity_limits" , limiter_lin_.has_velocity_limits , limiter_lin_.has_velocity_limits );
    controller_nh.param("linear/x/has_acceleration_limits", limiter_lin_.has_acceleration_limits, limiter_lin_.has_acceleration_limits);
    controller_nh.param("linear/x/has_jerk_limits" , limiter_lin_.has_jerk_limits , limiter_lin_.has_jerk_limits );
    controller_nh.param("linear/x/max_velocity" , limiter_lin_.max_velocity , limiter_lin_.max_velocity );
    controller_nh.param("linear/x/min_velocity" , limiter_lin_.min_velocity , -limiter_lin_.max_velocity );
    controller_nh.param("linear/x/max_acceleration" , limiter_lin_.max_acceleration , limiter_lin_.max_acceleration );
    controller_nh.param("linear/x/min_acceleration" , limiter_lin_.min_acceleration , -limiter_lin_.max_acceleration );
    controller_nh.param("linear/x/max_jerk" , limiter_lin_.max_jerk , limiter_lin_.max_jerk );
    controller_nh.param("linear/x/min_jerk" , limiter_lin_.min_jerk , -limiter_lin_.max_jerk );

    controller_nh.param("angular/z/has_velocity_limits" , limiter_ang_.has_velocity_limits , limiter_ang_.has_velocity_limits );
    controller_nh.param("angular/z/has_acceleration_limits", limiter_ang_.has_acceleration_limits, limiter_ang_.has_acceleration_limits);
    controller_nh.param("angular/z/has_jerk_limits" , limiter_ang_.has_jerk_limits , limiter_ang_.has_jerk_limits );
    controller_nh.param("angular/z/max_velocity" , limiter_ang_.max_velocity , limiter_ang_.max_velocity );
    controller_nh.param("angular/z/min_velocity" , limiter_ang_.min_velocity , -limiter_ang_.max_velocity );
    controller_nh.param("angular/z/max_acceleration" , limiter_ang_.max_acceleration , limiter_ang_.max_acceleration );
    controller_nh.param("angular/z/min_acceleration" , limiter_ang_.min_acceleration , -limiter_ang_.max_acceleration );
    controller_nh.param("angular/z/max_jerk" , limiter_ang_.max_jerk , limiter_ang_.max_jerk );
    controller_nh.param("angular/z/min_jerk" , limiter_ang_.min_jerk , -limiter_ang_.max_jerk );

    // If either parameter is not available, we need to look up the value in the URDF
```

```
  bool lookup_wheel_separation_h = !controller_nh.getParam("
      wheel_separation_h", wheel_separation_h_);
  bool lookup_wheel_radius = !controller_nh.getParam("wheel_radius",
      wheel_radius_);

  if (!setOdomParamsFromUrdf(root_nh,
                            rear_wheel_name,
                            front_steer_name,
                            lookup_wheel_separation_h,
                            lookup_wheel_radius))
  {
    return false;
  }

  // Regardless of how we got the separation and radius, use them
  // to set the odometry parameters
  const double ws_h = wheel_separation_h_multiplier_ * wheel_separation_h_;
  const double wr = wheel_radius_multiplier_ * wheel_radius_;
  odometry_.setWheelParams(ws_h, wr);
  ROS_INFO_STREAM_NAMED(name_,
                        "Odometry params : wheel separation height " << ws_h
                        << ", wheel radius " << wr);

  setOdomPubFields(root_nh, controller_nh);

  //-- rear wheel
  //---- handles need to be previously registerd in ackermann_steering_test.
      cpp
  ROS_INFO_STREAM_NAMED(name_,
                        "Adding the rear wheel with joint name: " <<
                            rear_wheel_name);
  rear_wheel_joint_ = vel_joint_if->getHandle(rear_wheel_name); // throws on
       failure
  //-- front steer
  ROS_INFO_STREAM_NAMED(name_,
                        "Adding the front steer with joint name: " <<
                            front_steer_name);
  front_steer_joint_ = pos_joint_if->getHandle(front_steer_name); // throws
      on failure
  ROS_INFO_STREAM_NAMED(name_,
                        "Adding the subscriber: cmd_vel");
  sub_command_ = controller_nh.subscribe("cmd_vel", 1, &
      AckermannSteeringController::cmdVelCallback, this);
  ROS_INFO_STREAM_NAMED(name_, "Finished controller initialization");

  return true;
}

void AckermannSteeringController::update(const ros::Time& time, const ros::
    Duration& period)
{
  // COMPUTE AND PUBLISH ODOMETRY
  if (open_loop_)
  {
    odometry_.updateOpenLoop(last0_cmd_.lin, last0_cmd_.ang, time);
  }
  else
  {
    double wheel_pos = rear_wheel_joint_.getPosition();
    double steer_pos = front_steer_joint_.getPosition();

    if (std::isnan(wheel_pos) || std::isnan(steer_pos))
      return;

    // Estimate linear and angular velocity using joint information
    steer_pos = steer_pos * steer_pos_multiplier_;
    odometry_.update(wheel_pos, steer_pos, time);
  }


  // MOVE ROBOT
  // Retreive current velocity command and time step:
  Commands curr_cmd = *(command_.readFromRT());
  const double dt = (time - curr_cmd.stamp).toSec();

  // Brake if cmd_vel has timeout:
  if (dt > cmd_vel_timeout_)
  {
    curr_cmd.lin = 0.0;
    curr_cmd.ang = 0.0;
  }

  // Limit velocities and accelerations:
  const double cmd_dt(period.toSec());

  limiter_lin_.limit(curr_cmd.lin, last0_cmd_.lin, last1_cmd_.lin, cmd_dt);
  limiter_ang_.limit(curr_cmd.ang, last0_cmd_.ang, last1_cmd_.ang, cmd_dt);

  last1_cmd_ = last0_cmd_;
  last0_cmd_ = curr_cmd;

  // Set Command
  const double wheel_vel = curr_cmd.lin/wheel_radius_; // omega = linear_vel
       / radius
  rear_wheel_joint_.setCommand(wheel_vel);
  front_steer_joint_.setCommand(curr_cmd.ang);

  // Publish odometry message
  if (last_state_publish_time_ + publish_period_ < time)
  {
    last_state_publish_time_ += publish_period_;
    // Compute and store orientation info
    const geometry_msgs::Quaternion orientation(
          tf::createQuaternionMsgFromYaw(odometry_.getHeading()));

    // Populate odom message and publish
    if (odom_pub_->trylock())
    {
      odom_pub_->msg_.header.stamp = time;
      odom_pub_->msg_.pose.pose.position.x = odometry_.getX();
      odom_pub_->msg_.pose.pose.position.y = odometry_.getY();
      odom_pub_->msg_.pose.pose.orientation = orientation;
      odom_pub_->msg_.twist.twist.linear.x = odometry_.getLinear();
      odom_pub_->msg_.twist.twist.angular.z = odometry_.getAngular();
      odom_pub_->unlockAndPublish();
    }

    // Publish tf /odom frame
    if (enable_odom_tf_ && tf_odom_pub_->trylock())
    {
     geometry_msgs::TransformStamped& odom_frame = tf_odom_pub_->msg_.
         transforms[0];
     odom_frame.header.stamp = time;
     odom_frame.transform.translation.x = odometry_.getX();
     odom_frame.transform.translation.y = odometry_.getY();
     odom_frame.transform.rotation = orientation;
     tf_odom_pub_->unlockAndPublish();
    }

    double steer_angle = odometry_.getAngular();
    double speed = odometry_.getLinear();
// double steer_angle = curr_cmd.ang;
// double speed = curr_cmd.lin;

    // Publish robot speed and steering angle
    if (publish_ackermann_drive_ && cmd_ackermann_drive_pub_ &&
        cmd_ackermann_drive_pub_->trylock())
    {
     cmd_ackermann_drive_pub_->msg_.header.stamp = time;
     cmd_ackermann_drive_pub_->msg_.header.frame_id = base_frame_id_;
     cmd_ackermann_drive_pub_->msg_.drive.speed = speed;
     cmd_ackermann_drive_pub_->msg_.drive.steering_angle_velocity = 0.0;
     cmd_ackermann_drive_pub_->msg_.drive.steering_angle = steer_angle;
     cmd_ackermann_drive_pub_->msg_.drive.acceleration = 0;
     cmd_ackermann_drive_pub_->msg_.drive.jerk = 0;
     cmd_ackermann_drive_pub_->unlockAndPublish();
    }
   }

  }

  void AckermannSteeringController::starting(const ros::Time& time)
  {
    brake();

    // Register starting time used to keep fixed rate
    last_state_publish_time_ = time;

    odometry_.init(time);
  }

  void AckermannSteeringController::stopping(const ros::Time& /*time*/)
  {
    brake();
  }

  void AckermannSteeringController::brake()
  {
    const double steer_pos = 0.0;
    const double wheel_vel = 0.0;

    rear_wheel_joint_.setCommand(steer_pos);
    front_steer_joint_.setCommand(wheel_vel);
  }
```

```

void AckermannSteeringController::cmdVelCallback(const geometry_msgs::Twist
    & command)
{
  if (isRunning())
  {
    // check that we don't have multiple publishers on the command topic
    if (!allow_multiple_cmd_vel_publishers_ && sub_command_.getNumPublishers
        () > 1)
    {
      ROS_ERROR_STREAM_THROTTLE_NAMED(1.0, name_, "Detected " << sub_command_
          .getNumPublishers()
          << " publishers. Only 1 publisher is allowed. Going to brake.");
      brake();
      return;
    }

    command_struct_.ang = command.angular.z;
    command_struct_.lin = command.linear.x;
    command_struct_.stamp = ros::Time::now();
    command_.writeFromNonRT (command_struct_);
    ROS_DEBUG_STREAM_NAMED(name_,
                           "Added values to command. "
                           << "Ang: " << command_struct_.ang << ", "
                           << "Lin: " << command_struct_.lin << ", "
                           << "Stamp: " << command_struct_.stamp);
  }
  else
  {
    ROS_ERROR_NAMED(name_, "Can't accept new commands. Controller is not
        running.");
  }
}


bool AckermannSteeringController::setOdomParamsFromUrdf(ros::NodeHandle&
    root_nh,
                           const std::string rear_wheel_name,
                           const std::string front_steer_name,
                           bool lookup_wheel_separation_h,
                           bool lookup_wheel_radius)
{
  if (!(lookup_wheel_separation_h || lookup_wheel_radius))
  {
    // Short-circuit in case we don't need to look up anything, so we don't
        have to parse the URDF
    return true;
  }

  // Parse robot description
  const std::string model_param_name = "robot_description";
  bool res = root_nh.hasParam(model_param_name);
  std::string robot_model_str="";
  if (!res || !root_nh.getParam(model_param_name,robot_model_str))
  {
    ROS_ERROR_NAMED(name_, "Robot descripion couldn't be retrieved from
        param server.");
    return false;
  }

  urdf::ModelInterfaceSharedPtr model(urdf::parseURDF(robot_model_str));

  urdf::JointConstSharedPtr rear_wheel_joint(model->getJoint(rear_wheel_name
      ));
  urdf::JointConstSharedPtr front_steer_joint(model->getJoint(
      front_steer_name));

  if (lookup_wheel_separation_h)
  {
    // Get wheel separation
    if (!rear_wheel_joint)
    {
      ROS_ERROR_STREAM_NAMED(name_, rear_wheel_name
                             << " couldn't be retrieved from model description");
      return false;
    }

    if (!front_steer_joint)
    {
      ROS_ERROR_STREAM_NAMED(name_, front_steer_name
                             << " couldn't be retrieved from model description");
      return false;
    }

    ROS_INFO_STREAM("rear wheel to origin: "
                    << rear_wheel_joint->parent_to_joint_origin_transform.
                        position.x << ","
                    << rear_wheel_joint->parent_to_joint_origin_transform.
                        position.y << ", "
                    << rear_wheel_joint->parent_to_joint_origin_transform.
                        position.z);

    ROS_INFO_STREAM("front steer to origin: "
                    << front_steer_joint->parent_to_joint_origin_transform.
                        position.x << ","
                    << front_steer_joint->parent_to_joint_origin_transform.
                        position.y << ", "
                    << front_steer_joint->parent_to_joint_origin_transform.
                        position.z);

    wheel_separation_h_ = fabs(
            rear_wheel_joint->parent_to_joint_origin_transform.position.x
            - front_steer_joint->parent_to_joint_origin_transform.position.
                x);

    ROS_INFO_STREAM("Calculated wheel_separation_h: " << wheel_separation_h_
        );
  }

  if (lookup_wheel_radius)
  {
    // Get wheel radius
    if (!getWheelRadius(model->getLink(rear_wheel_joint->child_link_name),
        wheel_radius_))
    {
      ROS_ERROR_STREAM_NAMED(name_, "Couldn't retrieve " << rear_wheel_name
          << " wheel radius");
      return false;
    }
    ROS_INFO_STREAM("Retrieved wheel_radius: " << wheel_radius_);
  }

  return true;
}

void AckermannSteeringController::setOdomPubFields(ros::NodeHandle& root_nh
    , ros::NodeHandle& controller_nh)
{
  // Get and check params for covariances
  XmlRpc::XmlRpcValue pose_cov_list;
  controller_nh.getParam("pose_covariance_diagonal", pose_cov_list);
  ROS_ASSERT(pose_cov_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
  ROS_ASSERT(pose_cov_list.size() == 6);
  for (int i = 0; i < pose_cov_list.size(); ++i)
    ROS_ASSERT(pose_cov_list[i].getType() == XmlRpc::XmlRpcValue::TypeDouble
        );

  XmlRpc::XmlRpcValue twist_cov_list;
  controller_nh.getParam("twist_covariance_diagonal", twist_cov_list);
  ROS_ASSERT(twist_cov_list.getType() == XmlRpc::XmlRpcValue::TypeArray);
  ROS_ASSERT(twist_cov_list.size() == 6);
  for (int i = 0; i < twist_cov_list.size(); ++i)
    ROS_ASSERT(twist_cov_list[i].getType() == XmlRpc::XmlRpcValue::
        TypeDouble);

  // Setup odometry realtime publisher + odom message constant fields
  odom_pub_.reset(new realtime_tools::RealtimePublisher<nav_msgs::Odometry>(
      controller_nh, "odom", 100));
  odom_pub_->msg_.header.frame_id = odom_frame_id_;
  odom_pub_->msg_.child_frame_id = base_frame_id_;
  odom_pub_->msg_.pose.pose.position.z = 0;
  odom_pub_->msg_.pose.covariance = {
      static_cast<double>(pose_cov_list[0]), 0., 0., 0., 0., 0.,
      0., static_cast<double>(pose_cov_list[1]), 0., 0., 0., 0.,
      0., 0., static_cast<double>(pose_cov_list[2]), 0., 0., 0.,
      0., 0., 0., static_cast<double>(pose_cov_list[3]), 0., 0.,
      0., 0., 0., 0., static_cast<double>(pose_cov_list[4]), 0.,
      0., 0., 0., 0., 0., static_cast<double>(pose_cov_list[5]) };
  odom_pub_->msg_.twist.twist.linear.y = 0;
  odom_pub_->msg_.twist.twist.linear.z = 0;
  odom_pub_->msg_.twist.twist.angular.x = 0;
  odom_pub_->msg_.twist.twist.angular.y = 0;
  odom_pub_->msg_.twist.covariance = {
      static_cast<double>(twist_cov_list[0]), 0., 0., 0., 0., 0.,
      0., static_cast<double>(twist_cov_list[1]), 0., 0., 0., 0.,
      0., 0., static_cast<double>(twist_cov_list[2]), 0., 0., 0.,
      0., 0., 0., static_cast<double>(twist_cov_list[3]), 0., 0.,
      0., 0., 0., 0., static_cast<double>(twist_cov_list[4]), 0.,
      0., 0., 0., 0., 0., static_cast<double>(twist_cov_list[5]) };
  tf_odom_pub_.reset(new realtime_tools::RealtimePublisher<tf::tfMessage>(
      root_nh, "/tf", 100));
  tf_odom_pub_->msg_.transforms.resize(1);
  tf_odom_pub_->msg_.transforms[0].transform.translation.z = 0.0;
  tf_odom_pub_->msg_.transforms[0].child_frame_id = base_frame_id_;
```

```
    tf_odom_pub_->msg_.transforms[0].header.frame_id = odom_frame_id_;
  }

  PLUGINLIB_EXPORT_CLASS(ackermann_steering_controller::
      AckermannSteeringController, controller_interface::ControllerBase)
} // namespace ackermann_steering_controller
```

Appendix F

Odometry simulation results

The various tests made to compare the results of the ackermann controller and the python script.

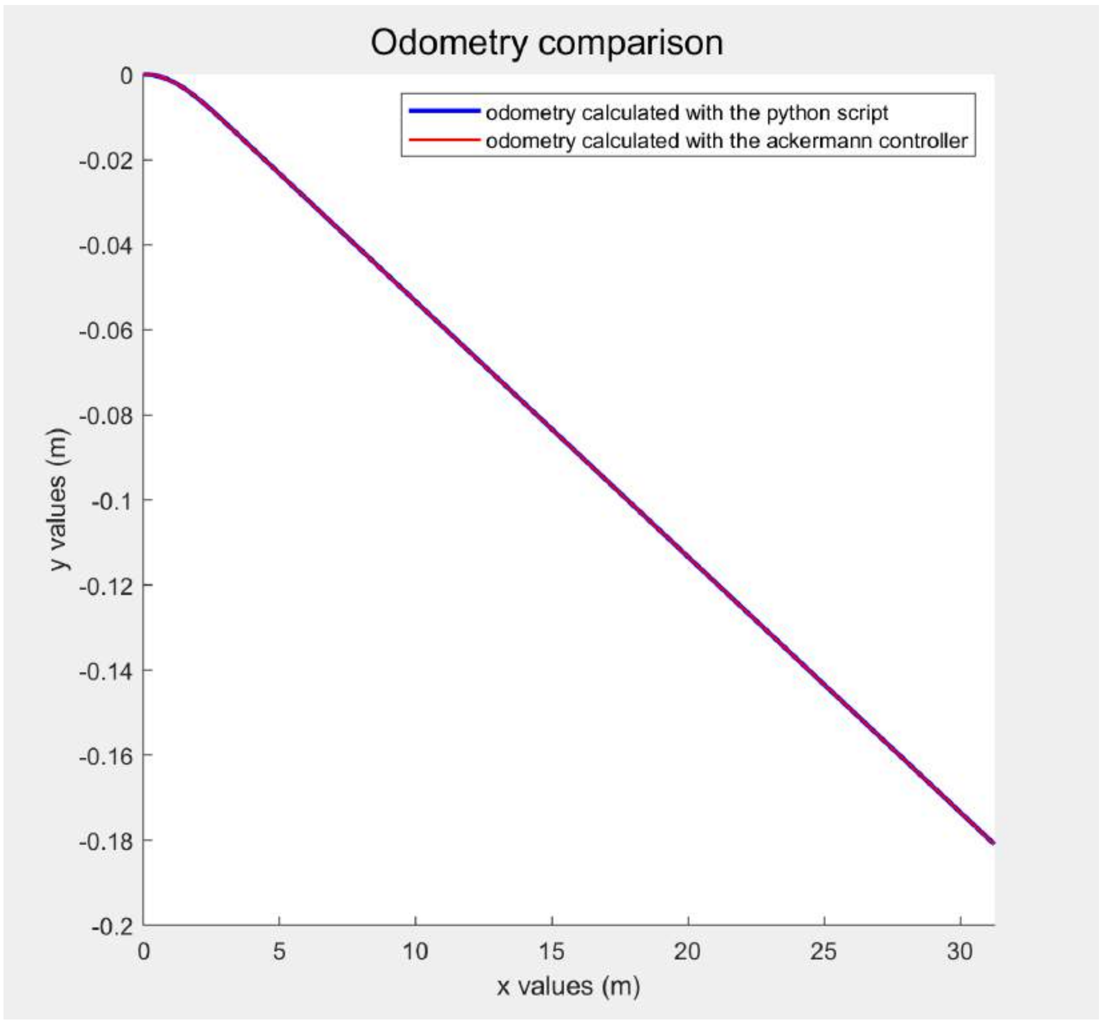


Figure F.1: 30m forward odometry simulation.

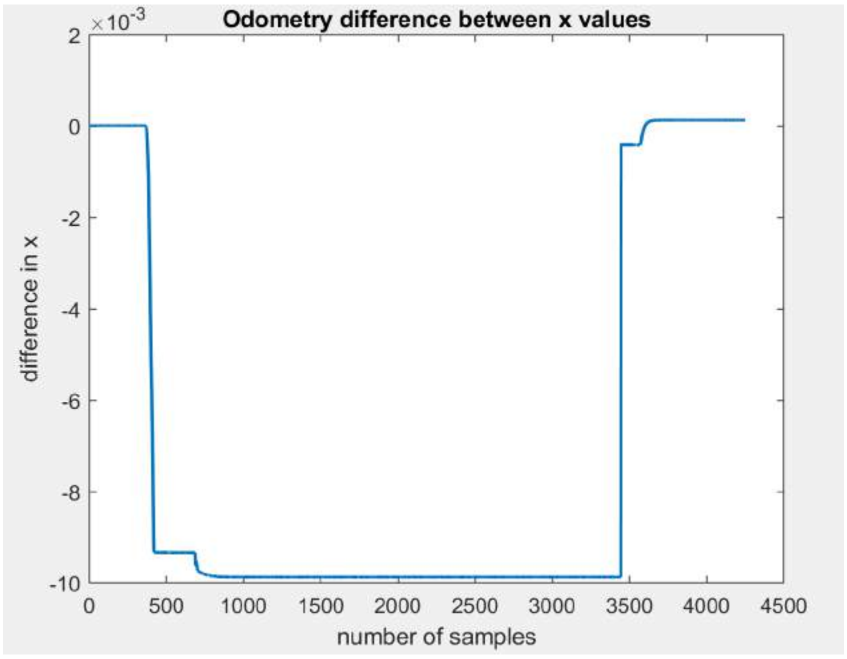


Figure F.2: 30m forward x values difference.

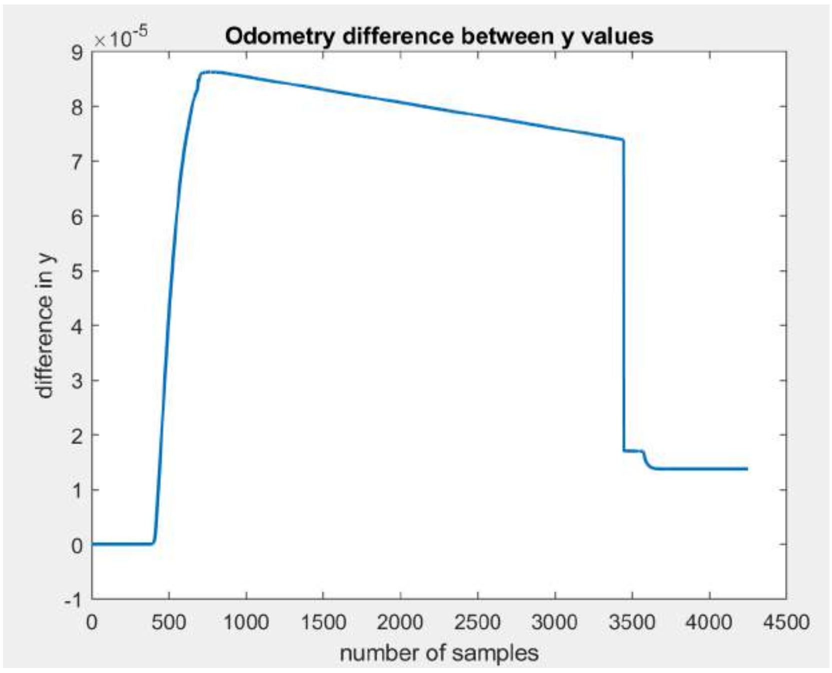


Figure F.3: 30m forward y values difference.

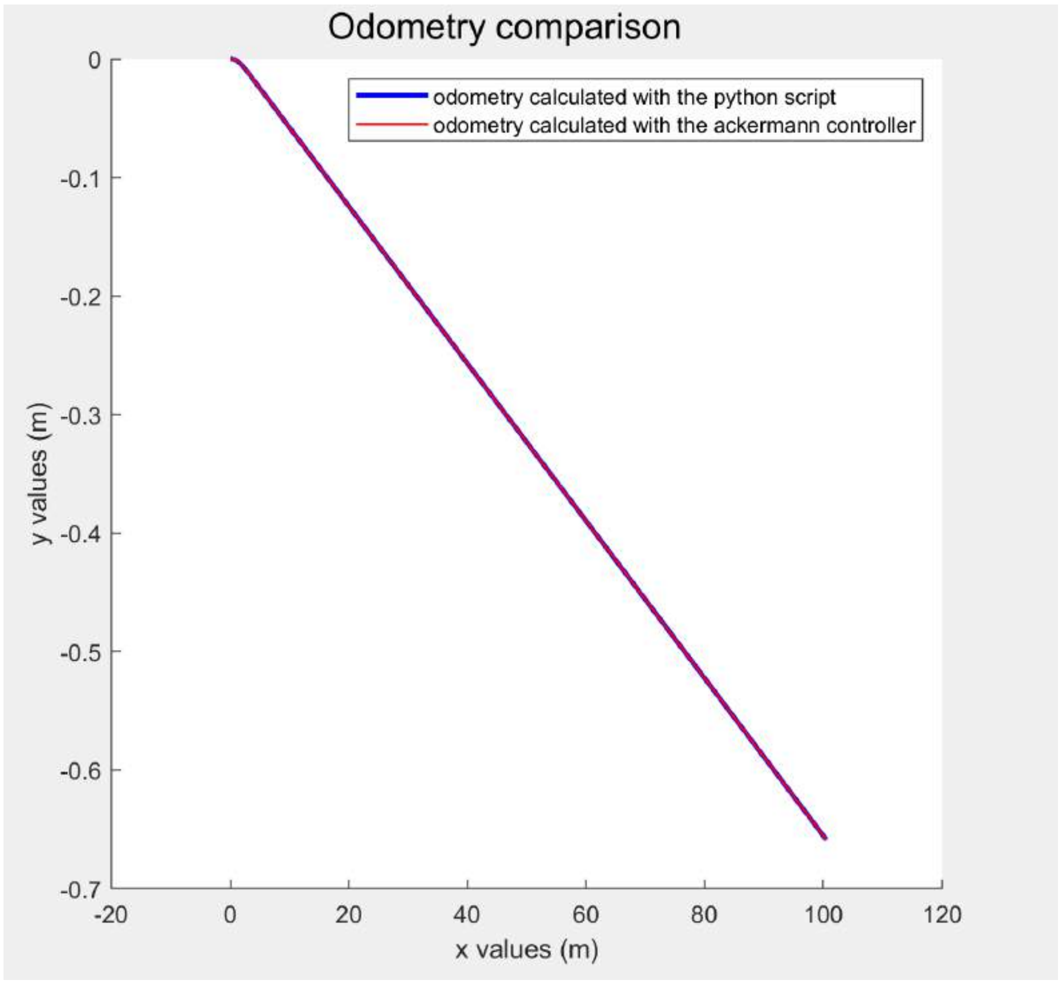


Figure F.4: 100m forward odometry simulation.

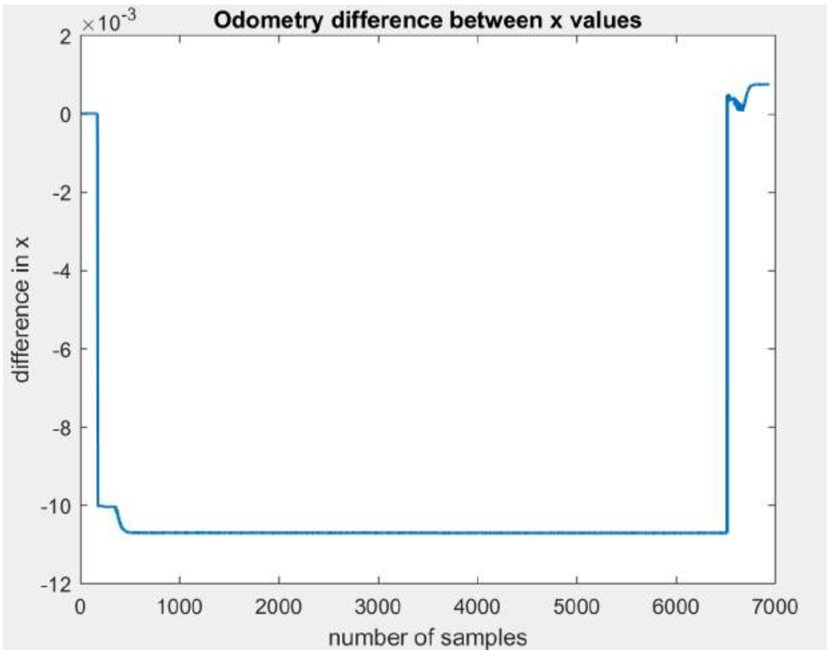


Figure F.5: 100m forward x values difference.

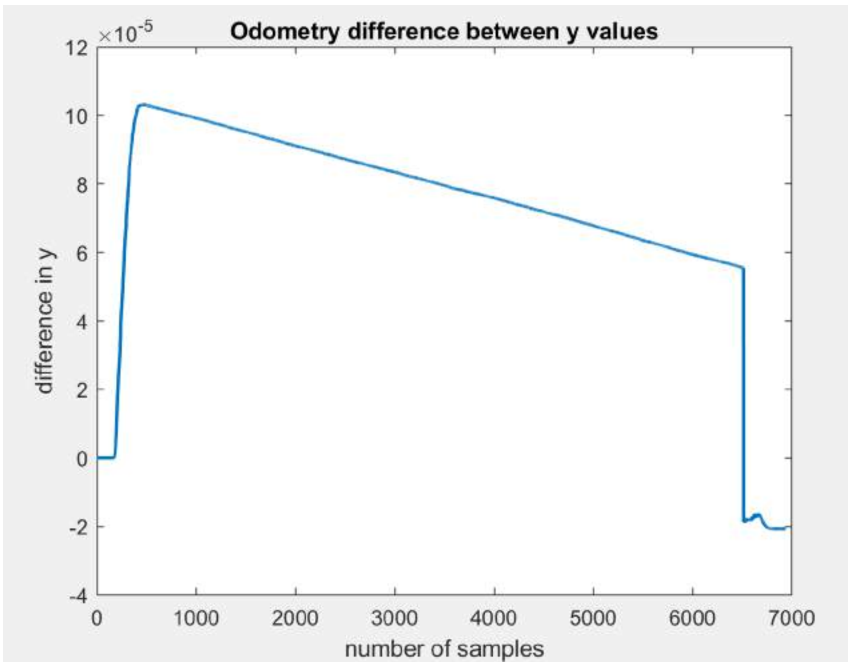


Figure F.6: 100m forward y values difference.

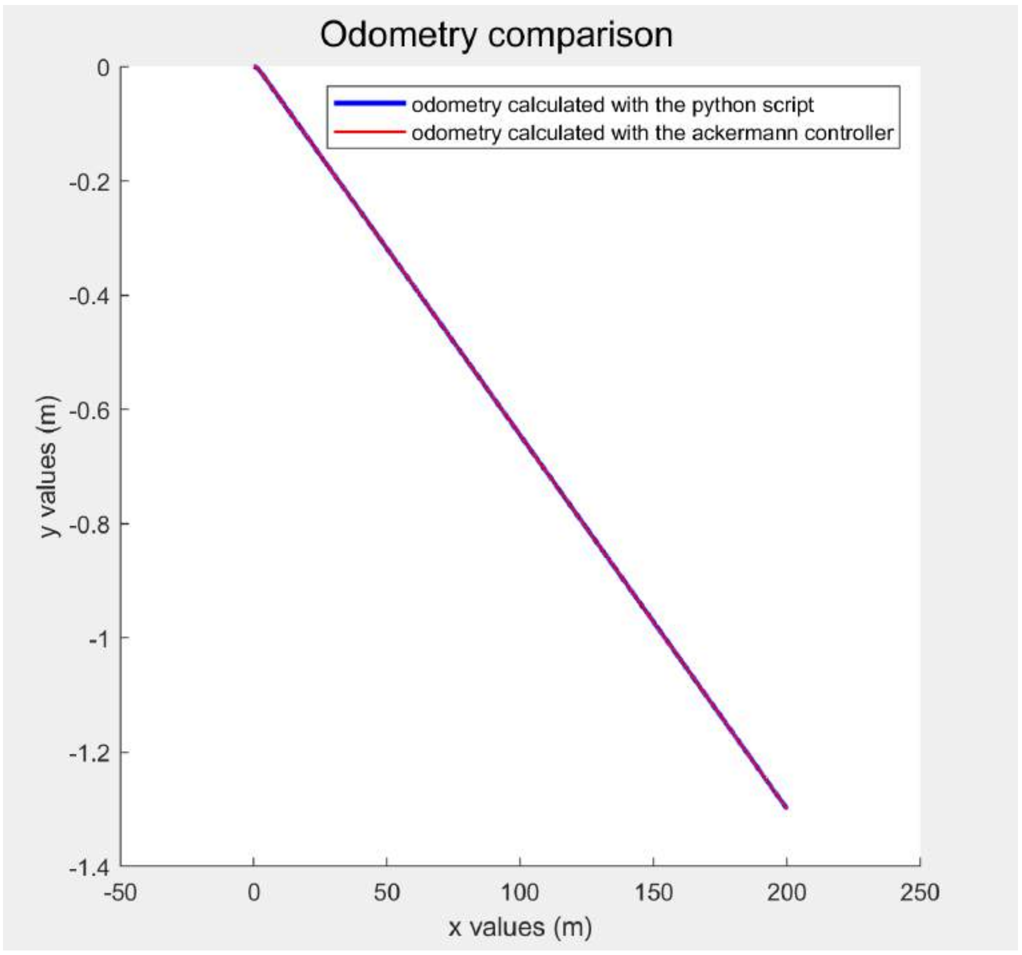


Figure F.7: 200m forward odometry simulation.

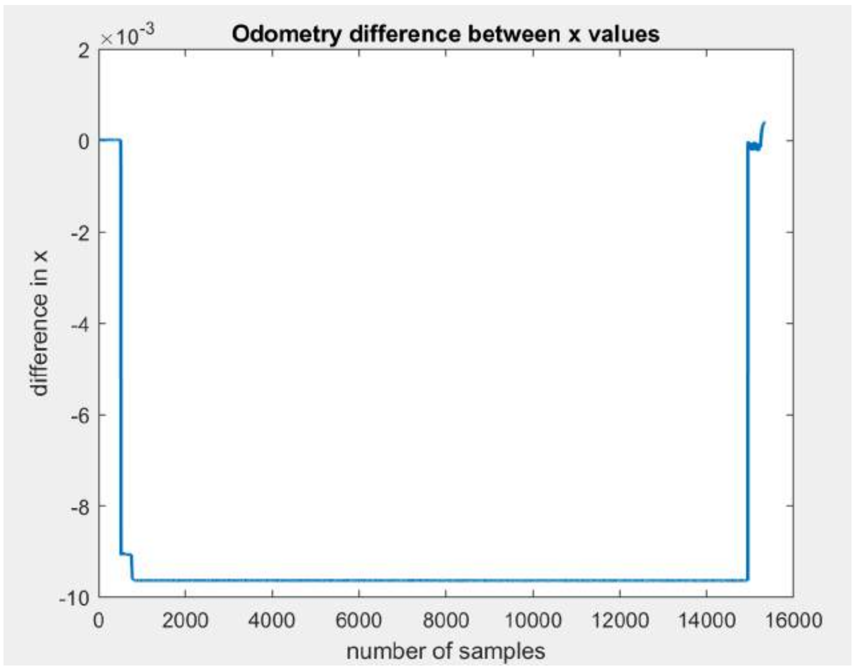


Figure F.8: 200m forward x values difference.

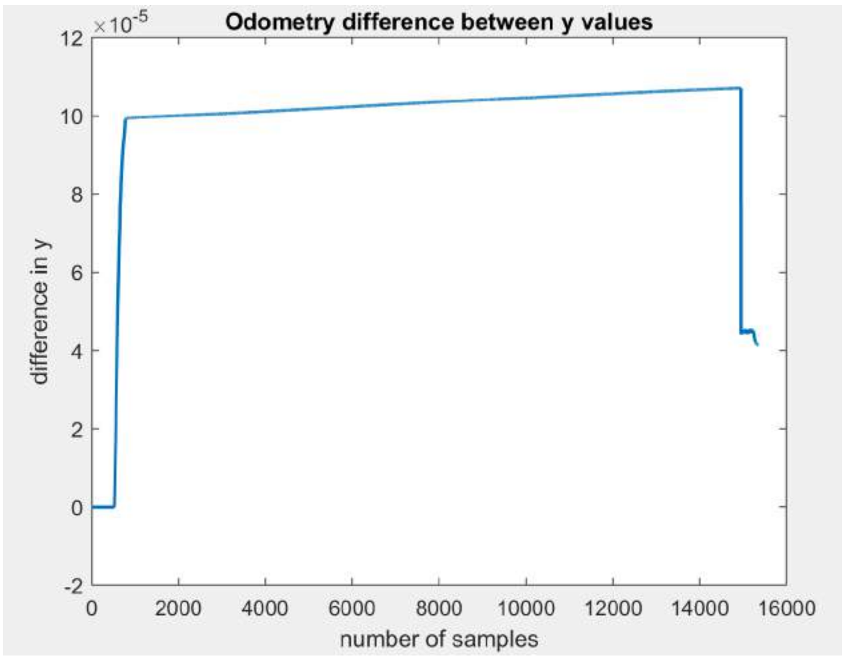


Figure F.9: 200m forward y values difference.

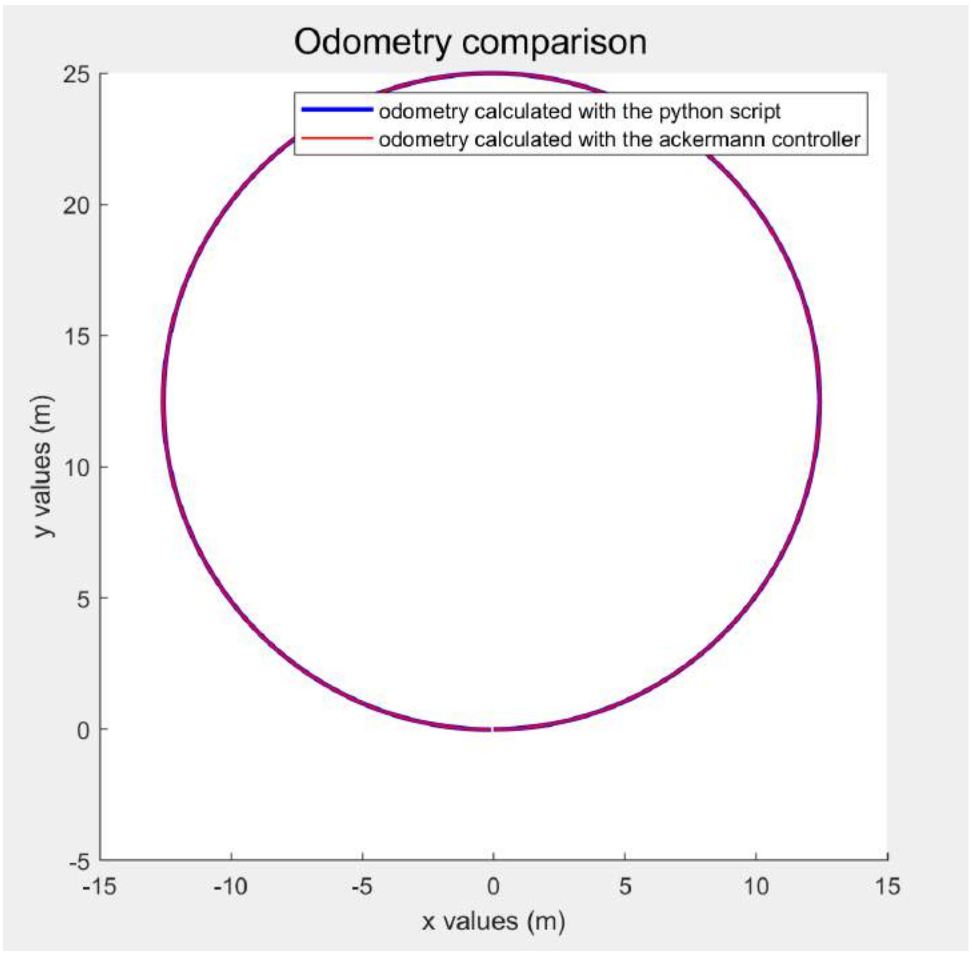


Figure F.10: 20rad/s turn odometry simulation.

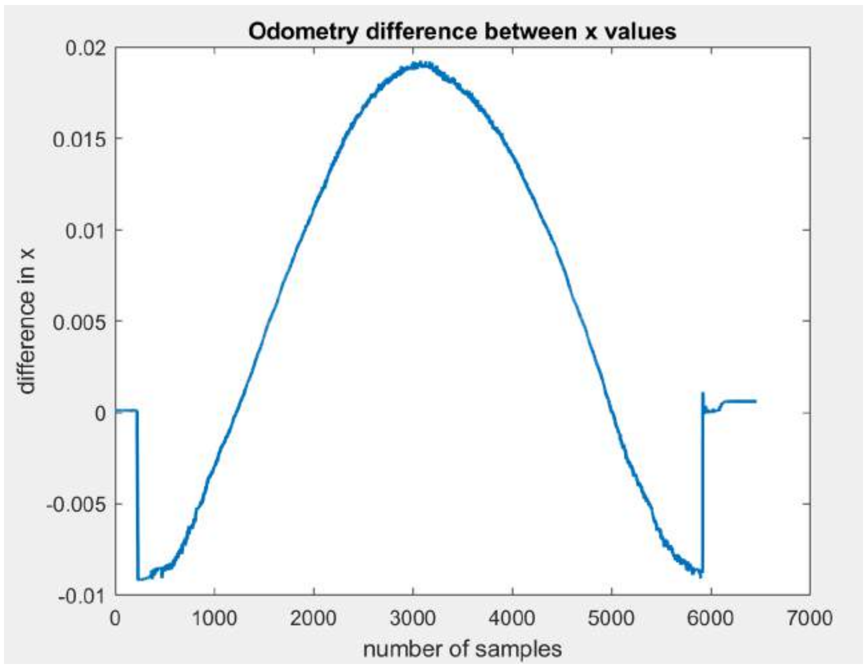


Figure F.11: 20rad/s turn x values difference.

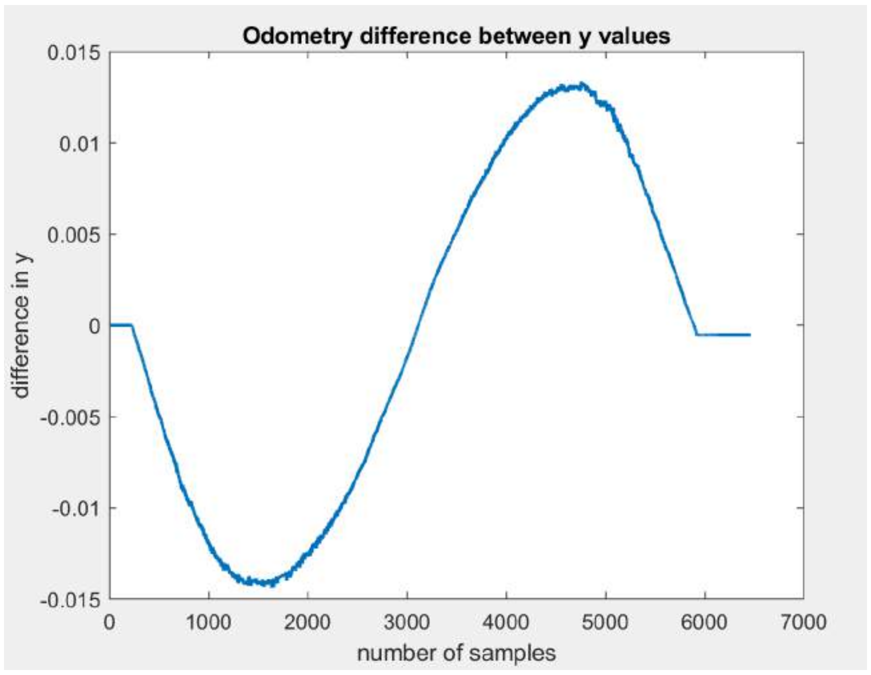


Figure F.12: 20rad/s turn y values difference.

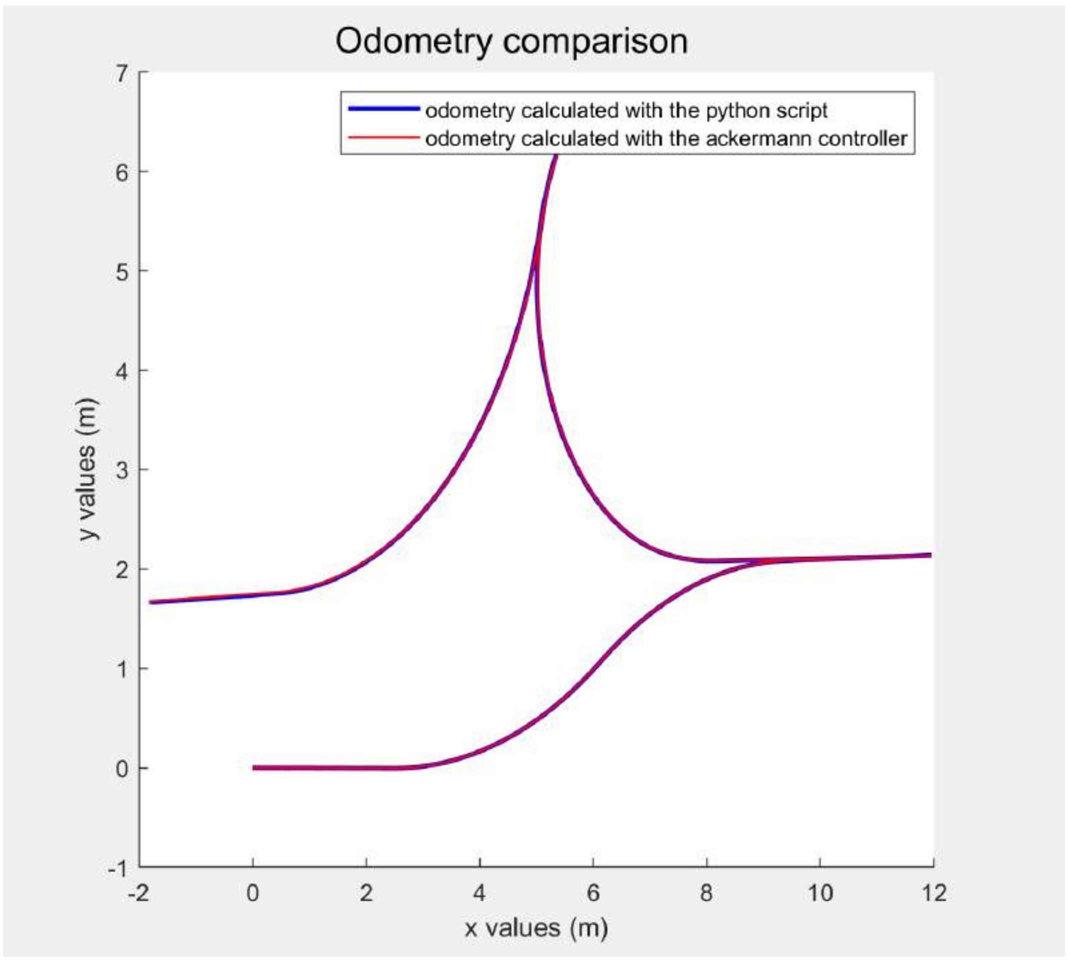


Figure F.13: Random course odometry simulation.

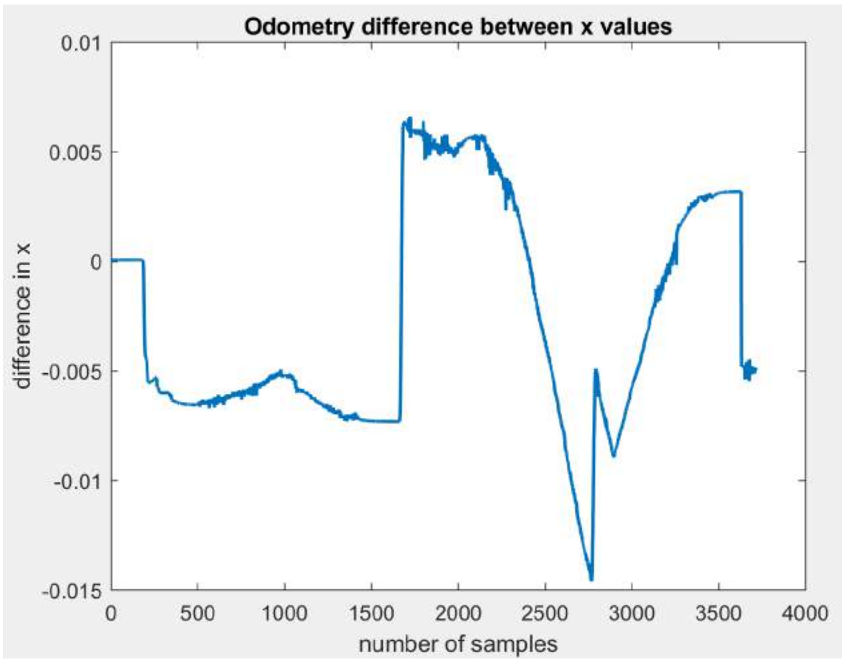


Figure F.14: Random course x values difference.

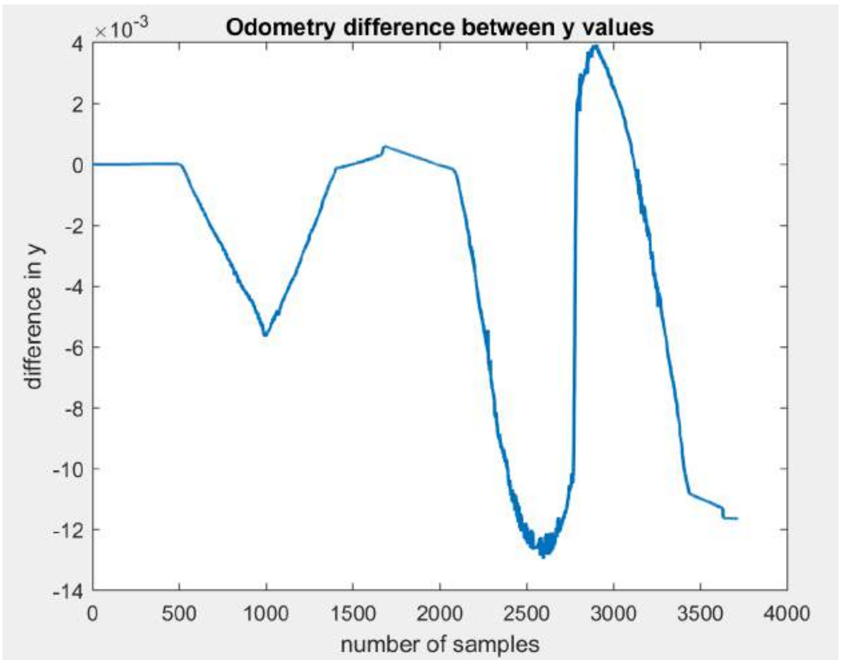


Figure F.15: Random course y values difference.