POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

High Level Control Architecture and Dynamic Simulation for an Autonomous All Terrain Robot

AI & R Lab
Laboratorio di Intelligenza Artificiale
e Robotica del Politecnico di Milano

Relatore: Ing. Matteo Matteucci
Correlatore: Ing. Davide Cucci

Tesi di Laurea di:
Gianluca Bardaro, matricola 787012

Anno Accademico 2013-2014

Abstract

In this thesis, the control architecture of an autonomous all terrain vehicle has been designed and developed together with the dynamic simulation used for its validation and for the rapid development/improvement of autonomous navigations strategies. The development of the control architecture has been done using a well-known and widely used framework for robotics: ROS (Robot Operating System). It is characterized by a high modularity, implemented with a publish-subscribe pattern, and the availability of several off-the-shelf modules. Taking advantage of the flexibility of the ROS framework, we integrated ROAMFREE (Robust Odometry Applying Multisensor Fusion to Reduce Estimation Errors) in our architecture, which is a library for multisensor fusion and pose estimation. We use this library to solve the localization problem, always present in the case of autonomous vehicles, and we couple it with a module that integrates odometry data to achieve a faster local estimate based on the absolute one provided by ROAMFREE. The simulation tools have been realized using V-REP (Virtual Robot Experimentation Platform), a simulator that allows to model complex scenarios that includes robot models with multiple sensor, physical object, and terrains; moreover, V-Rep has a native integration with ROS. While realizing the dynamic simulation great attention was given to integrate it seamlessly with the control architecture. The overall design of the control architecture and the dynamic simulation have been validated on the real physical platform through an extensive experimental activity with experiments on sensor fusion, localization, trajectory planning, and trajectory following.

Sommario

In questa tesi, l'architettura di controllo di un veicolo autonomo all-terrain è stata progettata e sviluppata insieme alla simulazione dinamica usata per la sua validazione e per il rapido sviluppo/miglioramento delle strategie di navigazione autonoma. Lo sviluppo dell'architettura di controllo è stato fatto usando un noto e largamente usato framework per la robotica: ROS (Robot Operating System). È caratterizzato da un'elevata modularità, implementata con un publish-subscribe pattern, e dalla disponibilità di svariati moduli già pronti. Sfruttando la flessibilità del framework ROS, abbiamo integrato ROAMFREE (Robust Odometry Applying Multisensor Fusion to Reduce Estimation Errors) nella nostra architettura, che è una libreria per la fusione multisensore e per la stima della posizione. Usiamo questa libreria per risolvere il problema della localizzazione, sempre presente quando si tratta di veicoli autonomi, ed è stata associata ad un modulo che integra i dati dell'odometria per ottenere una stima locale veloce basata su quella assoluta fornita da ROAMFREE. Gli strumenti di simulazione sono stati realizzati usando V-Rep (Virtual Robot Experimentation Platform), un simulatore che permette di creare modelli di scenari complessi che includono modelli di robot con multipli sensori, oggetti fisici e terreni; inoltre, V-Rep ha un'integrazione nativa con ROS. Nel realizzare la simulazione dinamica particolare attenzione è stata fatta nell'integrarla senza soluzione di continuità con l'architettura di controllo. L'intera struttura dell'architettura di controllo e della simulazione dinamica sono state convalidate sulla piattaforma fisica reale attraverso un'estesa attività sperimentale con esperimenti in fusione di sensori, localizzazione, pianificazione della traiettoria e inseguimento della traiettoria.

Ringraziamenti

Innanzi tutto voglio ringraziare il professor Matteo Matteucci, che mi ha dato l'opportunità di svolgere questa tesi, lavorando con un oggetto particolare com'è il quad. In secondo luogo voglio ringraziare Davide Cucci, che mi ha seguito più da vicino in questo lavoro e che ha dovuto sopportare, suo malgrado, la mia scarsa propensione per i test e per la documentazione. Un grazie generale a tutte quelle persone che hanno partecipato alle "giornate al quad", che hanno dovuto sopportare il freddo, o il caldo, a seconda della stagione.

Inoltre, se sono arrivato fino a questo punto, è solo grazie ai miei genitori, Giuseppe e Roberta, che mi hanno permesso di intraprendere e completare questo cammino. Con loro voglio ringraziare tutta la mia famiglia, mia nonna Luigia, che verso la fine era ben più in ansia di me per i miei esami, e mia sorella Serena, che non ha mai smesso di ricordami quanto impegno e dedizione siano necessari per affrontare l'Università. Ringrazio anche tutti i parenti vicini e lontani che si sono sempre interessati ai progressi del mio percorso in tutti questi anni.

Un ringraziamento speciale a tutti i miei amici, quelli che conosco da anni, quelli che nella mia vita sono apparsi da poco, perché senza di loro probabilmente la conclusione di questa tesi sarebbe arrivata prima, ma l'avrei barattata con un bel po' della mia sanità mentale.

Non posso escludere da questi ringraziamenti tutti i ragazzi e le ragazze che ho incontrato durante le mie ore di volontariato a Nature. Qualunque fossero i miei impegni in questi ultimi anni, ho sempre cercato di ritagliare un pomeriggio alla settimana da dedicargli. L'ho fatto per loro, per aiutarli con i loro studi e i loro compiti, ma l'ho fatto anche per me, perché sono sicuro di aver imparato più io da loro di quanta matematica sia mai riuscito ad insegnargli.
Grazie a tutti.

Gianluca

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, the high-level control architecture of an autonomous all terrain vehicle has been designed and developed together with the dynamic simulation used for its validation and for the rapid development/improvement of autonomous navigations strategies. The robot used is an unmanned ground vehicle (UGV); this category contains various type of vehicles that share the characteristic of being able to operate without an on-board human presence. UGVs can be used for applications where it may be inconvenient, dangerous, or impossible to have an operator present; for instance, they can be used to explore hazardous environments or difficult places to be reached for humans. Generally, they can be divided in two categories, those built on a custom platform and those based on a modified vehicle. Our robot is part of the second one, since it is based on a commercial ATV modified to support autonomous driving and equipped with various sensors used for perception and localization.

The work of this thesis starts from the robot already sporting a high-level control architecture developed using OROCOS. Although the original architecture implemented most of the core features required to an autonomous vehicle (i.e., manual drive, localization, and autonomous drive), those functionalities required significant improvements. It was also necessary to include the path following algorithm and the path planning module, both realized by other thesis, and this motivated the decision of developing a completely new architecture.

The development of the control architecture has been done using a well-known and widely used framework for robotics: ROS (Robot Operating System). It is characterized by a high modularity, implemented with a publish-subscribe pattern, and the availability of several off-the-shelf modules. This allowed us to develop a flexible architecture suitable for a prototype, which could undergo various modification during its development, while remaining a sufficiently robust and viable system for long-term use. Moreover, ROS simplify the integration of modules developed separately, e.g., the path following algorithm, and the planner.

Among the modules we are interested in integrating we have ROAMFREE (Robust Odometry Applying Multisensor Fusion to Reduce Estimation Errors), which is a library for multisensor fusion and pose estimation. We use this library to solve the localization problem, always present in the case of autonomous vehicles, and we couple it with a module that integrates odometry data to achieve a faster local estimate based on the absolute one provided by ROAMFREE.

Working with an autonomous all terrain vehicle, the role of the simulator is fundamental. The characteristics of this kind of robots, for example the physical dimensions and the typical operating environment, make it challenging to develop and test all the software components on the field. Therefore, a simulation that mimics the robot and the environment is required. The simulation we have developed is realized using V-REP (Virtual Robot Experimentation Platform), a simulator that allows to model complex scenarios that includes robot models with multiple sensor, physical objects, and terrains; moreover, it has a native integration with ROS. While realizing the dynamic simulation great attention was given to integrate it seamlessly with the control architecture. The result is a simulation environment that can substitute completely the real vehicle and it can be used to test and validate the system, and to perform complex experiments (i.e., high-speed trajectory following, rough terrain navigation, etc.). The overall design of the control architecture and the dynamic simulation have been validated on the real physical platform through an extensive experimental activity in sensor fusion, localization, trajectory planning, and trajectory following.

The contents of this thesis can be divided in two main contributions: the high-level control architecture and the simulation. First, we describe the architecture, starting from the communication with the low-level control system and with the sensors. Following how we solved the localization problem combining the absolute position estimated by the ROAMFREE library with a predictor based on the odometry data. Then we describe how we integrated in the architecture the path following and the path planning modules. Regarding the simulation, we provide a general description of the simulated environment, and then a detailed description on how we simulated each sensor to match the real behavior. Finally, we describe how the simulation interacts with the high-level architecture. Experiments done with both the robot and the simulation complete the thesis.

Starting from the contents of this thesis, in particular the interaction between the architecture and the simulation, a paper titled “A Simulation Based Architecture for the Development of an Autonomous All Terrain Vehicle” has been submitted and accepted for oral presentation at the 2014 International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2014).

The structure of the thesis is the following:

- In Chapter 2 some examples of autonomous vehicles proposed in literature are illustrated, followed by a description of the state of the robot before this thesis. The chapter contains also an overview of existing simulators and a detailed description of V-Rep. Also, the ROS framework and the ROAMFREE library are discussed.

- In Chapter 3 the current hardware configuration of the robot is illustrated. Starting from the vehicle characteristics, followed by a description of the sensors and the systems used to control the robot.

- In Chapter 4 an overview of the high-level control architecture is described, followed by a detailed description of each module developed or integrated.

- In Chapter 5 the simulated environment is illustrated, then the robot model is presented in details. The chapter contains also a description on how the simulation interact with the architecture.

- In Chapter 6 experimental results of experiments on localization, trajectory planning, and trajectory following are analysed and discussed.

- In Chapter 7 conclusions are made, and some possible extensions and improvement of this work are presented.

- Appendix A contains the paper derived from the contents of this thesis and accepted for publication in the 2014 International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2014).

Chapter 2

State of the art

2.1 Unmanned vehicles

Vehicular automation is as old as the vehicles themselves; today numerous automatic system are included in aircraft, boats, industrial machinery and agricultural vehicles. However, fully autonomous ground vehicles able to navigate on rough terrain or complex environment remain an ambitious goal. This kind of vehicles has various applications, for instance they can be used to explore hazardous environments or difficult places to be reached for humans. They can also improve everyday life, for example reliable driverless car could reduce the risk of accidents.

Unmanned ground vehicles (UGV) comes in very different sizes and shapes depending on the task they have been designed to accomplish. Generally, they can be divided in two categories, those built on a custom platform and those based on a modified vehicle. Over the years various vehicles were used as base to develop autonomous robots, like military vehicles, trucks, fuel-powered or electric automobiles, four-wheelers, and buses. Even if the world of UGV is quite diverse, they share some common characteristics:

- they are equipped with sensors to perceive the environment;
- it is possible to remotely control the vehicle;
- they are able to perform some autonomous tasks.

Moreover when designing an autonomous vehicle some common problems have to be faced, namely how to control the actuators of the vehicle, how to fuse the information from the sensors to determinate its position and how to drive autonomously. During the years, various approaches have been adopted in order to build and manage an UGV. In the following we present a list of some UGVs developed during the years, starting from the first examples of autonomous mobile robots [20].

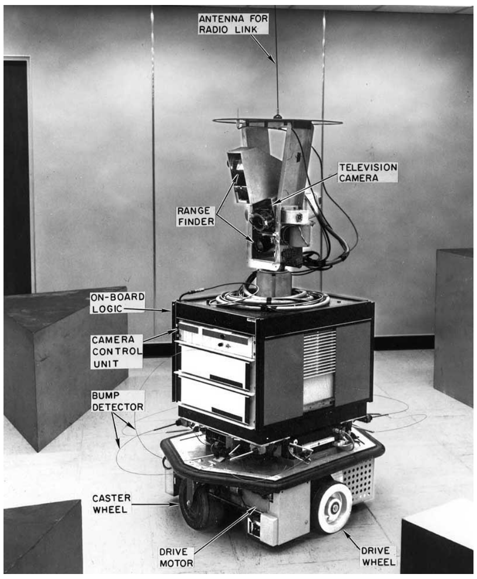
Shakey [36] (SRI International, United States, 1966-1972, Figure 2.1a) is considered the first mobile robot capable of autonomous behavior. It was a wheeled platform equipped with steerable TV camera, ultrasonic range finder, and touch sensors. An SDS-940 mainframe computer performed navigation and exploration tasks, a RF link connected the robot to it. While the robot autonomous capabilities were simple, it established the functional baselines for mobile robots of its era.

Stanford Cart [34] [35] (Stanford University AI Lab, United States, 1973-1981, Figure 2.1b) was a remotely controlled TV-equipped mobile robot. A computer program drove the Cart through cluttered spaces, gaining its knowledge of the world entirely from images broadcast by an on-board TV system. It used a sophisticated stereo vision system, where the single TV camera was moved to each of nine different positions on the top of its simple mobility base.

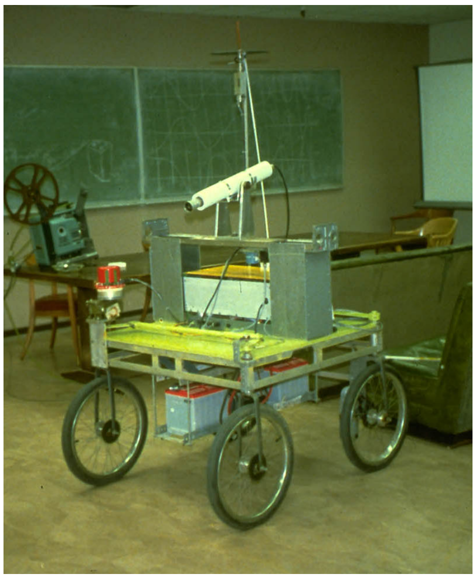
DARPA Autonomous Land Vehicle [30] [42] (DARPA's Strategic Computing, United States, 1985-1988, Figure 2.1c) was built on a Standard Manufacturing eight-wheel hydrostatically-driven all-terrain vehicle capable of speeds of up to 45 mph on the highway and up to 18 mph on rough terrain. The sensor suite consisted of a color video camera and a laser scanner. Video and range data processing modules produced road-edge information that was used to generate a model of the scene ahead.

Ground Surveillance Robot [23] [22] (Naval Ocean Systems Center, United States, 1985-1986, Figure 2.1d) project explored the development of a modular, flexible distributed architecture for the integration and control of complex robotic systems, using a fully actuated 7-ton M-114 armored personnel carrier as the host vehicle. With an array of fixed and steerable ultrasonic sensors and a distributed blackboard architecture implemented on multiple PCs, the vehicle successfully demonstrated autonomous following of both a lead vehicle and a walking human.

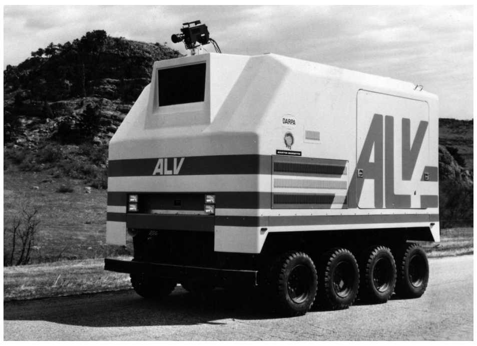
VaMP [16] [32] [31] (Bundeswehr University of Munich, Germany, 1993-1995, Figure 2.2a) is considered the first truly autonomous car, it was able to drive in heavy traffic for long distances without human intervention, using computer vision to recognize rapidly moving obstacles such as other cars, and automatically avoid and pass them. It was a 500 SEL Mercedes modified such that it was possible to control steering wheel, throttle, and brakes



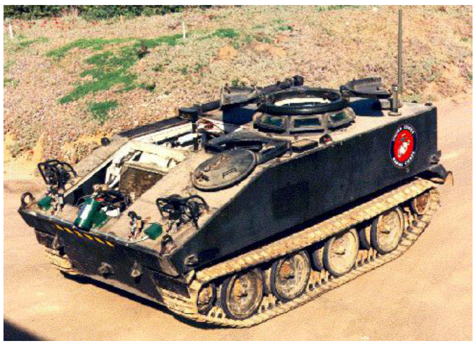
(a)



(b)



(c)



(d)

Figure 2.1: Some of the first prototypes: Shakey (a), Stanford Cart (b), DARPA ALV (c), and Ground Surveillance Robot (d)

(a)

(b)

Figure 2.2: Two prototypes of autonomous car: VaMP (a) and ARGO (b)

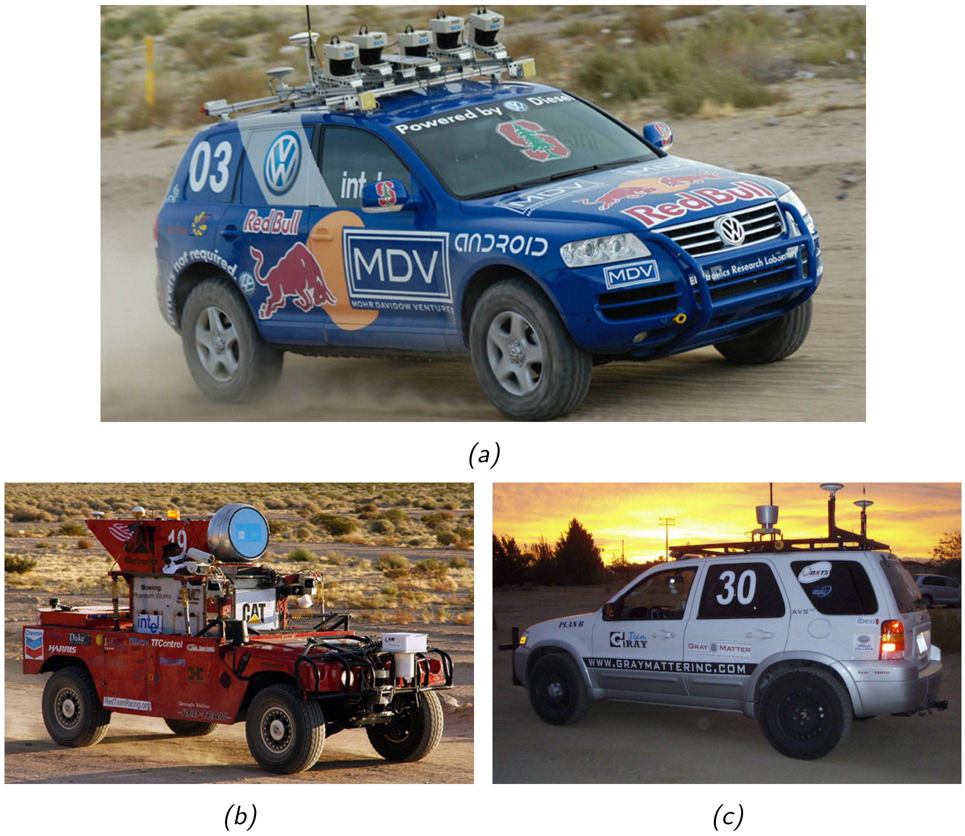


Figure 2.3: Three participants in the DARPA Grand Challenge: Stanley (a), Sandstorm (b), and Kat-5 (c)

through computer commands, and equipped with four cameras. In 1995, the vehicle was experimented on a long-distance test from Munich (Germany) to Odense (Denmark), and it was able to cover more than 1600 km, 95% of which with no human intervention.

ARGO [4] (University of Parma, Italy, 1998, Figure 2.2b) was a Lancia Thema passenger car equipped with a stereoscopic vision system consisting of two synchronized cameras able to acquire pairs of grey level images, which allowed to extract road and environmental information for the automatic driving of the vehicle. The ARGO vehicle had autonomous steering capabilities and human-triggered lane change maneuvers could be performed automatically. In June 1998, the vehicle was able to carry out a 2000 km journey through the Italian highway [7], 94% of the total trip was performed autonomously.

Stanley [44] (Stanford University, United States, 2005, Figure 2.3a) is an autonomous car that participated and won the second edition of the DARPA Grand Challenge in 2005 [10]. Stanley is based on a diesel-powered Volkswagen Touareg R5 with a custom interface that enables direct electronic actuation of both the throttle and brakes. A DC motor attached to the steering column provides electronic steering control. It is equipped with five SICK laser range finders, a color camera for long-range road perception, two RADAR sensors, and a GPS positioning system. It was able to complete the 212 Km off-road course in 6 hours and 54 minutes.

Sandstorm [47] (Carnegie Mellon University, United States, 2004-2005, Figure 2.3b) is an autonomous vehicle that participated at both editions of the DARPA Grand Challenge, the first in 2004, the second in 2005. Sandstorm is based on a heavily modified 1986 M998 HMMWV and drive-by-wire modifications control the acceleration, braking and shifting. The sensors used in 2004 included three fixed LIDAR laser-ranging units, one steerable LIDAR, a radar unit, a pair of cameras for stereo vision, and a GPS. In 2005, three additional fixed LIDAR were added, while the stereo cameras were removed. In 2004, Sandstorm obtained the best result but covered only 11.9 Km, in 2005, finished the race in 7 hours and 5 minutes, placing second.

Kat-5 [46] (GrayMatter, Inc., United States, 2005, Figure 2.3c) is an autonomous car developed by a team comprising employees from The Gray Insurance Company and students from Tulane University. It participated to the 2005 DARPA Grand Challenge and finished with a time of 7 hours and 30 minutes, only 37 minutes behind Stanley. Kat-5 is a 2005 Ford Escape Hybrid modified with the sensors and actuators needed for autonomous operation. It uses oscillating LIDARs and information from the INS/GPS unit to create a picture of the surrounding environment and drive-by-wire systems to control the vehicle.

RAVON [40] [39] (University of Kaiserslautern, Germany, 2006-2013, Figure 2.4a) is an autonomous robot based on the RobuCar TT platform by Robosoft. The vehicle's battery-driven 4WD utilizes four independent motors. It is equipped with two 2D SICK laser scanner, a custom-built stereo camera, a GPS, an IMU, and a magnetic field sensor. In 2007 and 2008, RAVON participated to the ELROB [41] competition obtaining good results in non-urban scenarios.

RTS-HANNA [28] [24] (University of Hannover, Germany, 2008-2013, Figure 2.4b) is based on a Kawasaki Mule 3010 side-by-side vehicle. Equipped with a drive-by-wire retrofit kit, this vehicle is fully controllable via computer. The robot is equipped with various sensors for tele-operation, semi-autonomous operation and fully autonomous operation. The main sensor is a pair of 3D laser range scanner; in addition, multiple cameras, Differential-GPS and inertial sensors are used.

Mörri [45] (University of Oulo, Finland, 2008-2010, Figure 2.4c) is a six wheel solid base robot with very strong motors. The robot has been designed in order to operate in extreme conditions and weathers and with heavy load to carry. It is equipped with two laser scanners, an omnidirectional camera, a thermal camera, inclinometer, and a GPS. Several autonomous features are availble, including collision avoidance, route execution, obstacle analysis, and data fusion.

Mana [5] (LAAS/CNRS, France, 2011-2013, Figure 2.4d) is a SegWay RMP400 that has been equipped with a stereovision bench, a LIDAR, a solid-state inertial measurement unit and a fiber-optic gyro. To perform autonomous navigation, Mana is assisted by the drone Mentor, which embeds a Hokuyo laser scanner and a camera. Mentor can assist Mana by providing him global information on the terrain ahead and by localizing him.

ARTOR [27] (ETH of Zurich, Swizerland, 2012-2014, Figure 2.4e) is a robotic vehicle capable of driving in rough terrain and at relatively high speed. It is based on a LandShark from Black-I Robotics, a 6-wheeled, skid-steered electric vehicle. The equipment includes a rotating 3D laser scanner, two 2D laser scanners, a stereo camera, a GPS receiver and an inertial measurement unit. Furthermore, a pan-tilt-zoom unit containing both a visual and a thermal camera is installed. The robot won the Creative Solution Award at the seventh European Land Robot Trials.

Looking at the prototypes listed above it is possible to note that originally mobile robots were simple prototype (i.e., Shakey, Standford Cart) or products resulting from substantial investments (i.e., DARPA Autonomous Land Vehicle, Vamp). While today big competition, like the DARPA Grand Challenge that offers a prize in millions of dollars, still exist, more low cost prototype with complex functionalities are developed thanks to the increased computational power of computers and advances in robotics. Moreover, it is possible to see a trend in the sensors used. Originally, vision sensors were preferred and used extensively, today, most of the prototypes relies on GPS, laser scanner and IMU to determinate their position.

2.2 Quadrivio project

In 2008, the AIRLab (Artificial Intelligence and Robotics Laboratory) and the MERLIN (Mechatronics and Robotics Laboratory for Innovation) of Politecnico di Milano, in collaboration with Aero Sekur S.p.A., developed

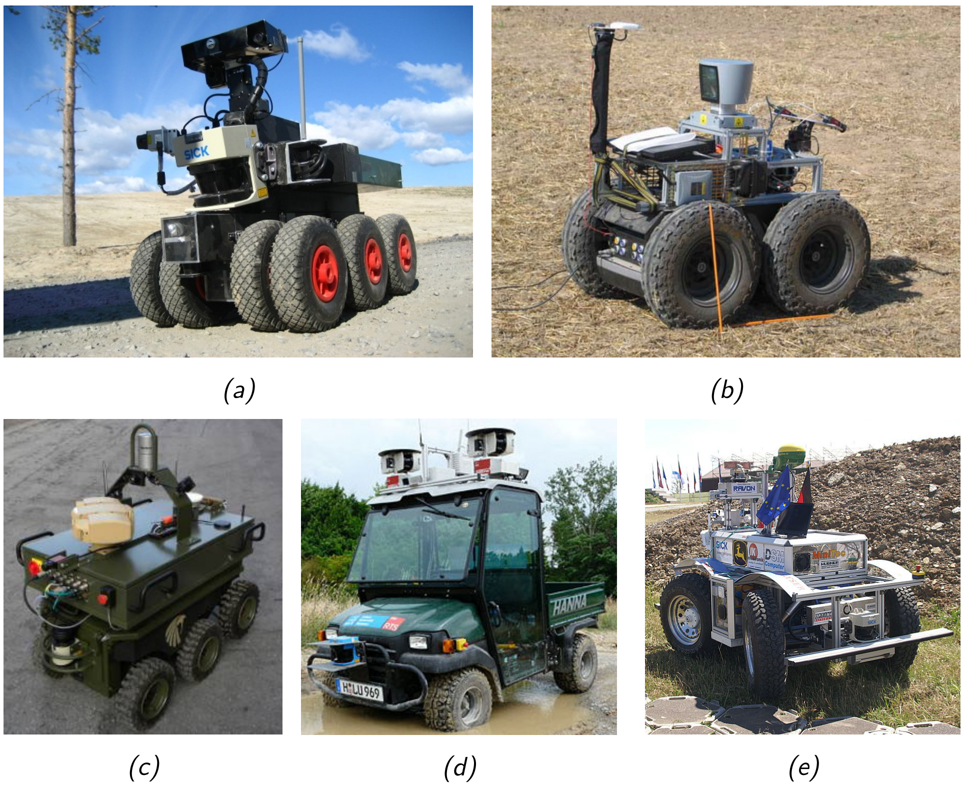


Figure 2.4: Some recent examples of UGV: Mörri (a), Mana (b), ARTOR (c), RTS-HANNA (d), and RAVON (e)

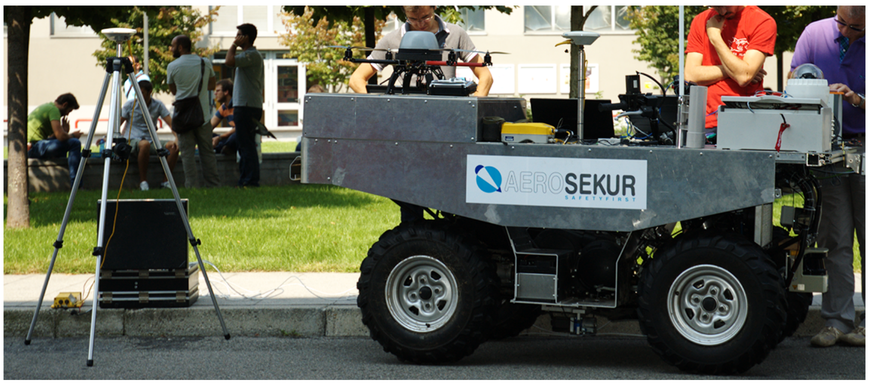


Figure 2.5: The robot, before the work of this thesis, with the drone parked on its platform. On the left it is possible to see the ground station for the RTK correction

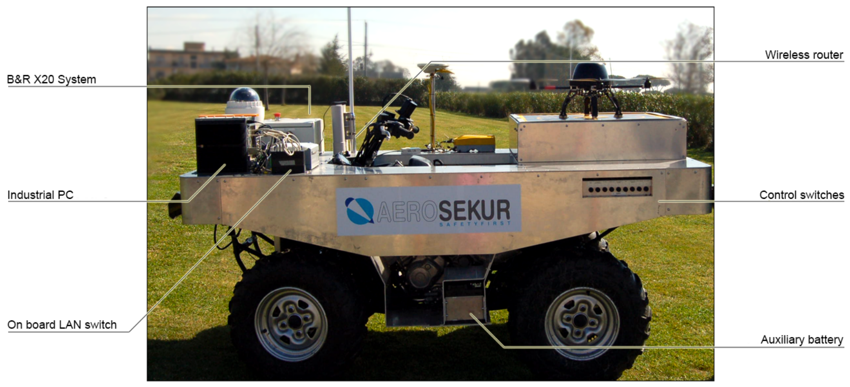


Figure 2.6: The position on the robot of it main components

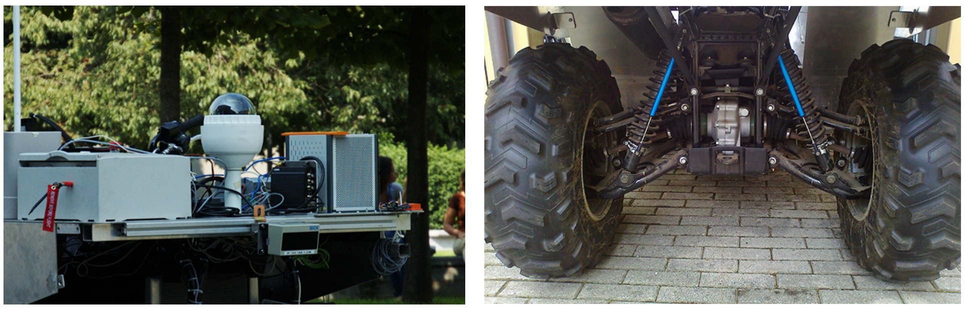


Figure 2.7: Two of the sensors that are still installed on the robot but not currently used: the omnidirectional camera (left) and the linear potentiometers (right)

a mobile robot based on a commercial ATV [8] [11] [49]. The final aim of the Quadrivio project was to develop an autonomous vehicle equipped with a manipulator arm in order to interact with object in the environment. While the arm was never installed, the robot reached its goal of autonomous driving.

2.2.1 Main vehicle characteristics

The robot is equipped with:

- an aluminum platform mounted over the chassis
- an industrial pc
- the actuators to control the vehicle
- a low-level control system that controls the actuators and receives the odometry
- a GPS with a ground station for the RTK correction
- an IMU with a magnetometer
- four linear potentiometers to measure suspensions stroke

The vehicle had also a platform where a drone could land and an omnidirectional camera associated with it. A second camera was mounted in front of the robot, and was used when teleoperating the vehicle. Finally, a laser scanner was mounted between the front wheels, but it was not connected to the system. The control of the robot was distributed; part on a teleoperating station and part on the robot itself. The robot was able retrieve its own speed and steer position, received from the GPS a fairly accurate global position, thanks to the RTK correction, and determinate its orientation using the IMU. The linear potentiometer were used to derive a rollover index.

An operator could control the steer and the speed of the robot remotely using a joypad (a Microsoft Xbox 360 Wireless Controller). The teleoperation system was robust and it has been tested outdoor on numerous occasions. Moreover, the robot could follow a trajectory defined with a list of waypoints by the user. No fully autonomous behavior with the use of a planner was available.

Shortly before the beginning of this thesis, the robot was revised, all the original equipment is still on the vehicle, but some of them now are disabled, and not used anymore. The industrial pc was replaced with a

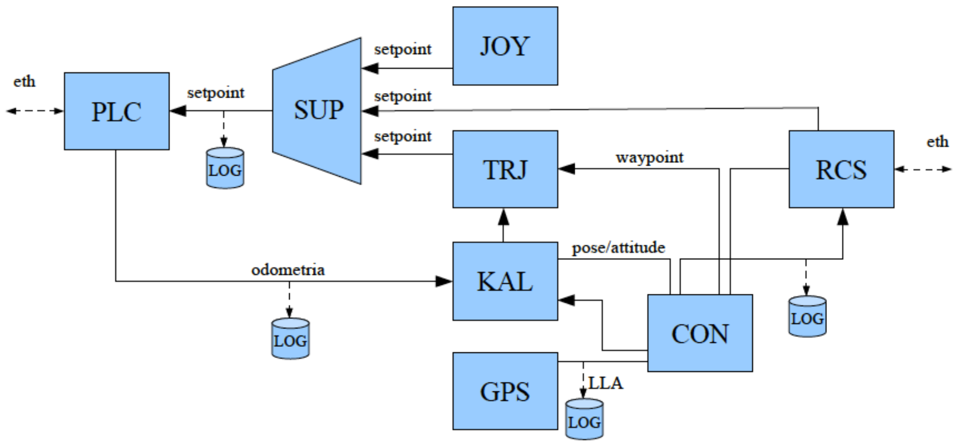


Figure 2.8: Schema of the original high-level control system

commercial laptop with superior computational power to accommodate the new software architecture. The omnidirectional camera is no longer used, since the aerial drone no longer works in conjunction with the robot (it was an external component from the sponsoring company). The GPS no longer use the RTK correction, since there is no terrestrial control station.

Additional changes have been made during this thesis, the camera used for teleoperation is now replaced by two stereo cameras with superior characteristics. Lastly, the laser scanner is now connected to the system and works properly. A more detailed description of the current robot hardware is given in Chapter 3.

2.2.2 Original software architecture

The original high-level control architecture was implemented in C/C++ using a framework known as OROCOS (Open Robot Control Software) [9]. The system was organized in modules implemented separately and integrated with OROCOS. This framework provides support for efficient and reliable management of real-time application, concurrent execution and communication between modules. Figure 2.8 shows a complete scheme of the architecture and following a brief description of the modules.

Remote Control Station (RCS) interface: This module communicates through a Wi-Fi connection with the remote graphical interface, which allows the manual drive of the vehicle, the definition of the GPS waypoints for autonomous drive and the monitoring of the vehicle status via telemetry data. This module periodically sends telemetry data at a frequency of 10 Hz, and it is activated when receiving new messages on the Wi-Fi connection.

Low-level control system (PLC) interface: This module communicates using an Ethernet socket with the low-level control system that implements the low-level control loops and collects measurements coming from the sensors. Therefore, this module sends to the low-level system the speed, steer and brake setpoints, and receives odometry data (i.e., speed, and steer). The control system sends odometry measurements at a frequency of 20 Hz.

Converter (CON): This module converts measurements in the NMEA format coming from the GPS and the Remote Control Station to the ENU format used on the vehicle for planning and navigation. Moreover, it is used to process the waypoints received from the remote control system and the state of the vehicle received from the Kalman filter.

Driver GPS (GPS): The GPS module interfaces with the GPS unit mounted on board of the vehicle, reads the GPS record in the NMEA 0813 format (at a frequency of 5 Hz), extracts the useful information (i.e., latitude, longitude, altitude and the estimated errors) and makes them available to the Kalman filter.

Kalman filter (KAL): This module implements the sensor fusion between the odometry coming from the low-level control system and the GPS data, in order to obtain an accurate estimate of position, speed and orientation of the vehicle. The Kalmann filter integrates, using a six degrees of freedom model, the previous state of the vehicle with the GPS and the odometry, providing an estimate of the current robot position. The initial position of the robot comes from the first available GPS data, while, for the orientation, it is necessary to drive the vehicle briefly.

Trajectory planner (TRJ): This module implements the planner algorithm and the trajectory following algorithm. It receives from the Kalmann filter the estimate of the current state of the robot, and from the Converter a list of waypoints. From these, the module derives the current setpoint of speed, steer and brake to be sent to the low-level control system.

Supervisor (SUP): This module manages the different operating modes of the vehicle: autonomous drive, manual drive from the Remote Control Station and manual drive using the joypad. Moreover, it implements security policy that stops the vehicle in the case of faulty behaviours.

Although the original architecture implemented most of the core features required to an autonomous vehicle (i.e., manual drive, localization, and autonomous drive), those functionalities required significant improvements. Therefore, we decided to develop a new architecture, described in details in Chapter 4, based on ROS that is characterized by its flexibility and modularity. Moreover, this architecture permits to integrate easily the new path planning and path following modules developed for the vehicle. The only module reused from the original architecture is the one that communicated with the low-level control system, reimplemented as a ROS node is now called `PLCClient`.

2.3 Physical simulation of vehicles

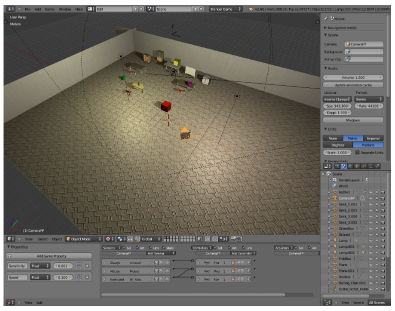
When working with an autonomous all terrain vehicle, the role of the simulator is fundamental. The characteristics of this kind of robots, for example the physical dimensions, the typical operating environment and the complexity of the system architecture, make it challenging to develop and test all the software components. Therefore, a simulation that mimic the robot and the environment is necessary.

Simulators are commonly used in various areas of science and engineering, robotics is no exception. Each step of the development of a robot may benefit from the use of a simulator. Even before building the real robot is it possible to use the simulator to create a prototype and verify the feasibility of the project. After that, it can be used to assist the design and development of the robot. Lastly, when doing experiments with the real robot they can be validated by similar ones in the simulation.

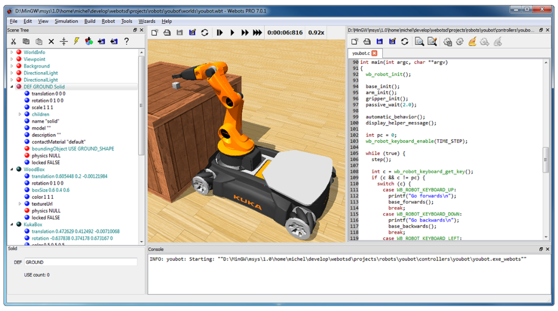
There are various types of simulators available. Some of them are specific for robotic applications, some focus on simplicity and fast prototyping, while others aim at flexibility. Moreover, there are simulators designed for a specific category of robots, e.g., humanoids, manipulators or mobile robots. On the other side, there are software for simulating vehicles that focus on a precise physical simulation and on the interaction of multiple systems. In the following we provide a list of simulators with a brief description for each one:

V-Rep [19] [38] is a general purpose robot simulator with integrated development environment. It is based on a distributed control architecture; each object can be individually controlled via scripts, remote APIs or ROS nodes. This makes V-REP versatile and ideal for multi-robot applications. It can be used for fast algorithm development, fast prototyping and verification.

Webots [33] (Figure 2.9b) is a development environment used to model, program and simulate mobile robots. The simulator has various built-in models of robots, sensors and actuators. The robot behavior can be tested in physically realistic worlds, simulated using the Open Dynamic Engine.



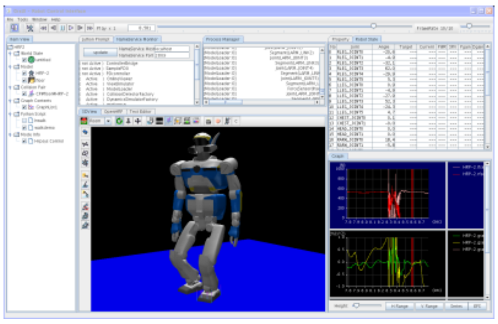
(a)



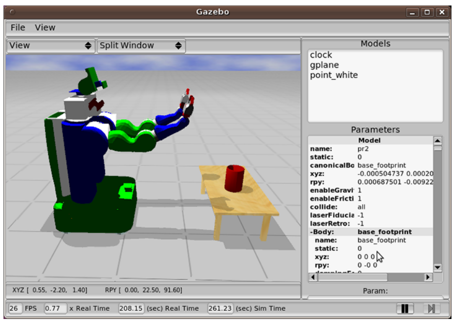
```
int main(int argc, char **argv)
{
  wb_robot_init();

  base_init();
  arm_init();
  gripper_init();
  passive_wait(2.0);

  automatic_behavior();
  display_helper_message();

  int pc = 0;
  wb_robot_keyboard_enable(TIME_STEP);

  while (true) {
    step();

    int c = wb_robot_keyboard_get_key();
    if (c && c != pc) {
      switch (c) {
        case WB_ROBOT_KEYBOARD_UP:
          printf("Go forwards\n");
          base_forwards();
          break;
        case WB_ROBOT_KEYBOARD_DOWN:
          printf("Go backwards\n");
          base_backwards();
          break;
        case WB_ROBOT_KEYBOARD_LEFT:
```

```
INFO: youbot: Starting: ""D:\MinGW\msys\1.0\home\michel\develop\webotsd\projects\robots\youbot\controllers\youbot\youbot.exe_webots""
```

(b)



(c)

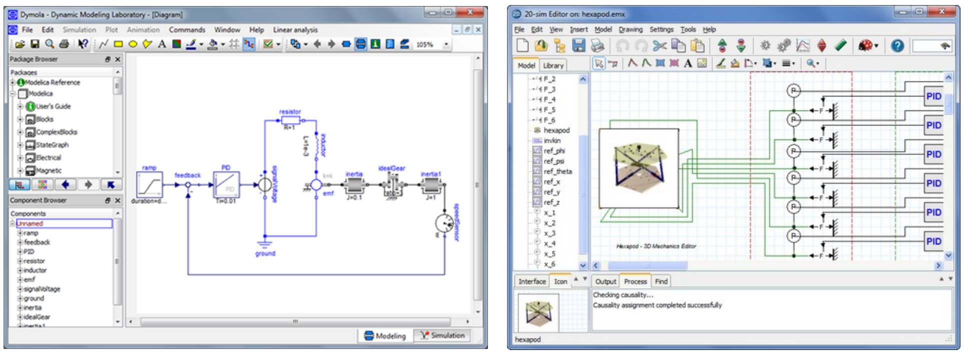


	Model
name:	pr2
static:	0
canonicalBo	base_footprint
xyz:	-0.000504737 0.00020
rpy:	0.000687501 -0.00922
enableGravi	1
enableFricti	1
collide:	all
laserFiducia	-1
laserRetro:	-1
-Body:	base_footprint
name:	base_footprint
static:	0
xyz:	0 0 0
rpy:	0 -0 0

```
XYZ [ 0.55, -2.20, 1.40]    RPY [ 0.00, 22.50, 91.60]
26 FPS  0.77 x Real Time  208.15 (sec) Real Time  261.23 (sec) Sim Time
```

(d)

Figure 2.9: Some examples of simulators for robotic application: MORSE (a), Webots (b), OpenHRP3 (c), Gazebo (d)



```
Checking causality...
Causality assignment completed successfully
```

(a) (b)

Figure 2.10: Some examples of multi-domain simulators: Dymola (a) and 20-sim (b)

Moreover, it offers an integrated IDE to develop controllers, which can be then directly transferred to commercially available real robots.

Gazebo [26] (Figure 2.9d) is an open source simulator. It was originally integrated with ROS, but now it is an independent project. Gazebo can simulate with accuracy and efficiency populations of robots in complex indoor and outdoor environments. It offers various models for sensors and a rich choice of physics engines, i.e., ODE, Bullet, Simbody, and DART.

OpenHRP3 [25] (Figure 2.9c) is an integrated software platform for robot simulations and software developments. It allows the users to inspect an original robot model and control program by dynamics simulation. In addition, OpenHRP3 provides various software components and libraries that can be used for robotics related software developments.

MORSE [29] (Figure 2.9a) is a generic simulator for academic robotics. It focuses on realistic 3D simulation of small to large environments, indoor or outdoor, with one to tenths of autonomous robots. It comes with a set of standard sensors (e.g. cameras, laser scanner, GPS), actuators and robotic bases. MORSE bases the rendering on the Blender Game Engine. The MORSE OpenGL-based Engine supports shaders, advanced lightnings, and uses the Bullet library for physics simulation.

Dymola [17] (Figure 2.10a) is a commercial modeling and simulation environment based on the open Modelica modeling language. It offers unique multi-engineering capabilities, which mean that it is possible to simulate the dynamic behavior and complex interactions between systems of many engineering fields, such as mechanical, electrical, thermodynamic, hydraulic, pneumatic and control systems. It lacks built-in sensors simulation.

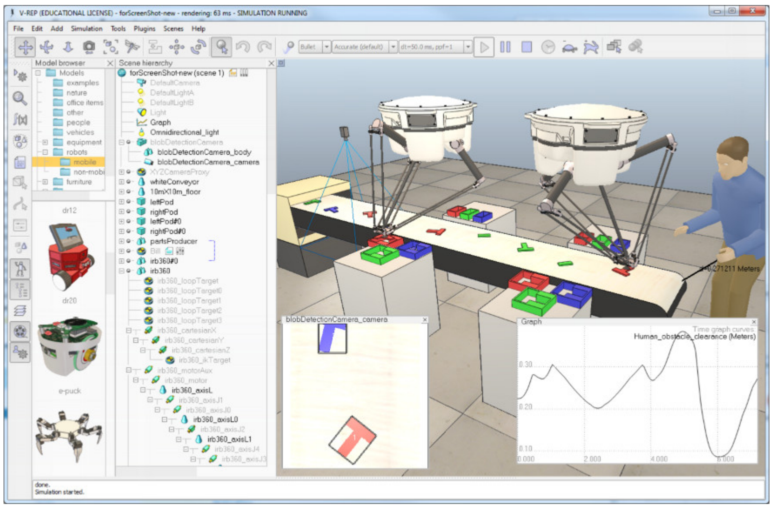


Figure 2.11: Main window of V-Rep

MapleSim [1] is a multi-domain modeling and simulation tool. It generates model equations, runs simulations, and performs analyses using the symbolic and numeric mathematical engine of Maple. Models are created using a drag-and-drop interface selecting modules that model object from various areas of science and engineering.

20-Sim [6] (Figure 2.10b) is a modeling and simulation program for mechatronic systems. Models are created graphically, similar to an engineering scheme, and they can be used to simulate and analyze the behavior of multi-domain dynamic systems and create control systems.

For our work, we chose to adopt V-Rep. What driven our choice was it simplicity and ease of use, contrary to Gazebo, which is notoriously difficult to install. Moreover, its flexibility makes it suitable to simulate different types of robots, differently from OpenHRP3 focused on a specific type. Lastly, V-Rep has a native integration with ROS, which simplify the interaction with our robot architecture. Simulators like Dymola are not suitable for our needs; while they have a precise physical simulation, they lack sensor models, which are fundamental to create a complete robot simulation.

2.4 Virtual Robot Experimental Platform

Virtual Robot Experimental Platform (V-Rep) [19] [38] is a general purpose robot simulator developed by Coppelia Robotics. Its main characteristics are the integrated development environment, with a customizable user interface, and a modular structure, both for the objects that compose the simulation and for the control methods. The development environment, available inside the simulator, simplifies the creation of robot models and simulations allowing for fast prototyping, fast algorithm development and verification. Moreover, when the simulation is active, this area act as a 3D rendering of the simulation, giving a real time feedback of the behavior of the models. Regarding the modularity, it takes form in the three main functionalities of the simulator: the objects that compose the scene, the control mechanisms and the computing modules. A V-Rep simulation scene, or simulation model, contains several objects that are assembled in a tree-like hierarchy. The following elements can compose a scene:

- Shapes: triangular meshes used for rigid body simulation and visualization. Other scene objects or calculation modules heavily rely on shapes for their calculations.
- Joints: elements that link two or more objects together with one to three degrees of freedom. There are four different types: prismatic, revolute, screw-like and spherical. Joints can be controlled using a direct force or torque, simulating electrical motors, or left in a passive mode.
- Proximity sensors: they calculate the exact distance to a shape in a configurable volume, instead to simply performing detection based on rays. This results in a more continuous operation and allows for a simulation that is more realistic.
- Vision sensors: They extract complex image information from a simulation scene (colors, object sizes, depth maps, etc.). A built-in filtering and image processing function enables the composition of blocks of filter elements.
- Force sensors: they are rigid links between shapes that can record applied forces and torques, and that can conditionally break apart when reaching a given threshold.
- Graphs: they can record predefined or custom data streams. Data streams can then be displayed directly in function of time, combined in two-dimensional graphs or 3D curves.
- Cameras: they allow scene visualization when associated with a viewport.
- Lights: they provide illumination to the scene and directly influence cameras or vision sensors.

Figure 2.12: Some of the built-in models that V-Rep offers

- Paths: they allow complex movement definitions in space (succession of translations and rotations), and they could be used for guiding a robot along a predefined trajectory.
- Dummies: a dummy is a reference frame, which is useful for various tasks, and it is mainly used in conjunction with other scene objects.
- Mills: they are customizable convex volumes used to simulate surface cutting operations on shapes.

The combination of these simpler elements permit to create complex models raging from manipulators to wheeled robots, and to simulate different types of sensors, like cameras, laser scanner, GPS, gyroscopes, etc. Moreover, V-Rep offers multiple build-in model, both for robot and for sensors, which are fully customizable. This makes the creation of fully functional scenes even simpler and faster.

Various control mechanism are available to manage the behavior of each element of the simulation, some integrated inside the simulator, while others permit to control the object from outside the simulated environment. Among internal mechanisms, the main one is the use of child scripts, which can be associated with any element in the scene. They handle a particular part of the simulation and they are an integral part of their associated object, being duplicated and serialized, together with them. Therefore, they are a portable and scalable control element: a single package containing the model definition together with its control or functionality.

A main script, which handles general functionalities is contained in each scene, and it calls the child scripts in a cascaded way with respect to the

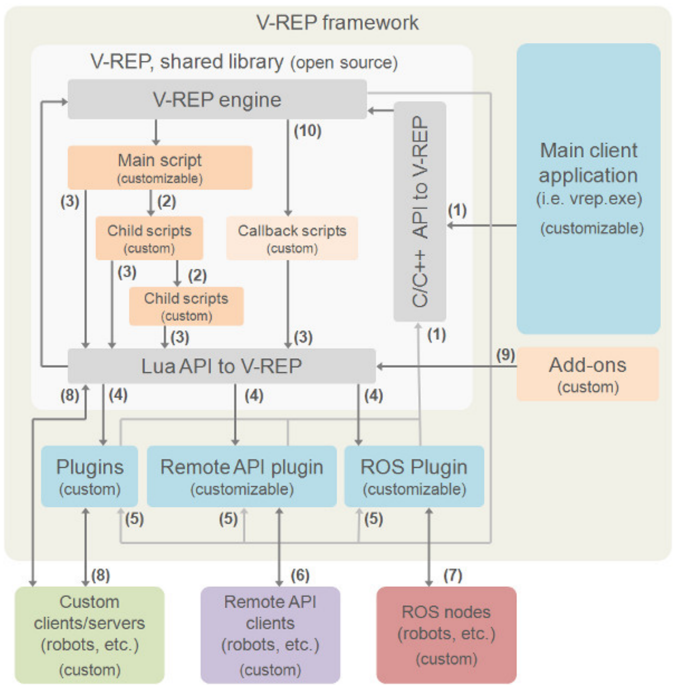

V-REP framework
V-REP, shared library (open source)
V-REP engine
Main script (customizable)
Child scripts (custom)
Callback scripts (custom)
Child scripts (custom)
Lua API to V-REP
C/C++ API to V-REP
Main client application (i.e. vrep.exe) (customizable)
Add-ons (custom)
Plugins (custom)
Remote API plugin (customizable)
ROS Plugin (customizable)
Custom clients/servers (robots, etc.) (custom)
Remote API clients (robots, etc.) (custom)
ROS nodes (robots, etc.) (custom)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)
(10)

Figure 2.13: V-REP framework. Colored areas can be customized

scene hierarchy. Child scripts have two execution modes: non-threaded and threaded. In the first case, the script is called at each main loop execution, synchronized with the simulation. For threaded script, V-Rep's thread scheduler makes them behave and appear as coroutines, which allow controlling the time at which the thread is executed, allowing for a synchronization with the main script or other child scripts. Moreover, each threaded script can programmatically request behaving like a real thread, for example when managing I/O operations. Embedded scripts also fill the gaps between various interfaces offered by the simulator: they can register ROS publishers and subscribers, open and handle communication lines, launch executables, load and unload plug-ins, or start remote API server services.

In order to customize and extend the functionalities offered by the embedded scripts it is possible to use plug-ins, written in C/C++. They can register custom Lua commands, allowing the execution of fast callback functions from within an embedded script. They can also extend the functionality of a particular simulation element, and they can be used to implement an interface to a specific hardware. In a similar way, add-ons are lightweight simulator customization methods built inside V-Rep using Lua scripts; they can implement stand-alone functions or regularly executed code.

V-Rep offers also a method to control the simulation from outside the simulator, this is useful for distributed architectures or simulator-in-the-loop configurations. The remote API interface in V-REP allows interacting with the simulation using a socket communication. It is composed by remote API server services and remote API clients. The client side, written in C/C++, Python, Java, Matlab or Urbi, can be embedded in any software running on remote control hardware or real robots, and it allows remote function calling, as well as fast data streaming. Functions support two calling methods to adapt to any configuration: blocking, waiting until the server replies, or non-blocking, reading streamed commands from a buffer. Plug-ins implement the API server inside V-Rep, therefore, other than those built-in in the simulator, it is possible to develop custom remote APIs. Moreover, the simulator offers a native integration with ROS. It implements a ROS node with a plug-in, which allows a node to call V-REP commands via ROS. Embedded scripts can enable publishers and subscribers using custom functions to stream data for the simulator to ROS and vice versa. V-Rep supports various standard ROS messages that are directly associated with the characteristics of the objects composing the simulation, like geometrical and dynamic properties and sensors readings.

To manage the interaction between the objects in the simulation, V-Rep contains various calculation modules. It has a kinematic module to calculate forward and invers kinematic of any mechanism (e.g., branched, closed, redundant, containing nested loops). The simulator includes a dynamic module that handles rigid body dynamic calculation and interaction via the Bullet Physics Library [13], the Open Dynamics Engine [43] or Vortex Dynamics [2]. It is possible to choose which physics engine to use before starting the simulation. Independent from the dynamic module, there is a collision detection module, which allows fast interference checking between any shape or collection of shapes. It uses data structures based on a binary tree of oriented bounding boxes for accelerations. The mesh-mesh distance calculation module uses the same structure to calculate the minimum distance between any shape or collection oh shapes. For versatility all these modules are implemented in a general way, without any assumptions on the underlying simulation scenes or models.

2.5 Robot operating system

In robotics, writing software is difficult because different types of robot can have extremely diverse hardware, making code reuse non-trivial. Moreover, the modules developed must implement a deep stack starting from driver-

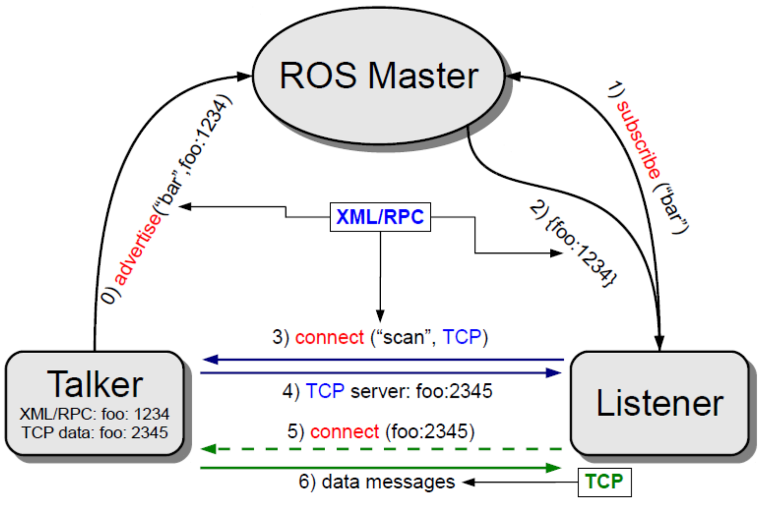


Figure 2.14: Main elements of the ROS framework

level software up to high-level functionalities, like autonomous driving, reasoning or localization. In order to resolve these issues, during the years, various frameworks have been developed often aiming at a very specific purpose. This caused a fragmentation in the robotic software systems used in industry and academia. ROS is an attempt to create a general framework that promotes modularity and code reuse, and it became the de facto standard for robot software.

The Robot operating system (ROS) [37] is a flexible framework, developed by the Stanford Artificial Intelligence Laboratory and by Willow Garage. It is an open-source, meta-operating system, providing typical services of operating systems, including hardware abstraction, low-level device control, message-passing between processes and package management. The main points around which ROS is designed are the portability and thinness of the software, a distributed architecture, the support of multiple programming languages and the availability of tools to extend its functionalities.

A typical ROS system consists of a number of processes, called nodes, potentially on a number of different hosts, connected at runtime in a peer-to-peer topology. Each node is an independent unit that performs computation, usually associated with a specific functionality or hardware component.

Nodes are organized in packages, which are directories that contain an XML file describing the package and stating any dependency. In order to increase the flexibility and portability of the system, it is possible to implement nodes using four different languages: C++, Python, Octave and LISP. Modules implemented with different languages can coexist in the same system, therefore it possible to use different tools for specific needs, e.g., fast prototyping and implementation of simpler node using Python with core functionalities implemented with C++. This is possible because the specification of ROS is at the messaging layer.

Messages defined with a simple, language-neutral Interface Definition Language (IDL) allow the communication between nodes. The IDL uses short text files to describe fields of each message, and allows composition of messages. Code generators for each supported language then generate native implementations, which are automatically serialized and deserialized by ROS as messages are sent and received. The ROS-based codebase contains hundreds of types of messages, which transport data ranging from sensor feeds to objects and maps, moreover it is possible to define custom messages for any specific need.

A node sends a message by publishing it to a given topic, which is identified by its name. A node that is interested in a certain kind of data subscribes to the appropriate topic. Multiple concurrent node can publish or subscribe on a single topic, and each node can interact with multiple topic. In general, publishers and subscribers are not aware of each other existence.

In order to complement the asynchronous communication system realized by the topic-based publish-subscribe model, ROS provides a synchronous system, called *services*. Each service is defined by a string name and a pair of strictly typed messages: one for the request and one for the response. Unlike topics, only one node can advertise a service of any particular name.

The peer-to-peer topology requires some sort of lookup mechanism to allow processes to find each other at runtime. The *master* has this role, it enables individual ROS nodes to locate each other. Once these nodes have located each other, they communicate with using peer-to-peer channels. Moreover, the master provides a *parameter server*, which is a shared, multi-variate dictionary that is accessible via network APIs. Nodes use this server to store and retrieve parameters at runtime.

Another strength of ROS is the availability of various existing nodes and packages, currently there are several hundreds of ROS packages available on publicly-viewable repositories. Most of them implements specific functionalities, e.g., sensors drivers, others more general ones, e.g., a path planner. For instance, `tf` [18] is a package that lets the user keep track of multiple coordinate frames over time. It maintains the relationship between coordinate frames in a tree structure buffered in time, and it lets the user transform points between any two coordinate frames at any desired point in time. The package offers two interfaces to access its functionalities: a listener and a broadcaster. The broadcaster notifies to the system about a transformation between two coordinate frames at a specific time, even in the future, while the listener provides the transformation at any time.

Along with the meta-operating system, the ROS environment provides various tools. These tools perform various tasks: for example navigate the source code tree, get and set configuration parameters, visualize the peer-to-peer connection topology, run collection of nodes, monitor the behavior of topics, graphically plot message data and more. Some of these tools have simple functionalities, e.g., showing all the messages published on a topic, while others are more complex. For example, `rviz` [21] is a complete visualization tool that shows in real time, in a 3D environment, data streamed on the topics. Another example is `rosbag`, which records and plays back to ROS topics.

2.6 ROAMFREE

Pose tracking is one of the most important issue in autonomous mobile robotics, because the performance of high-level control systems and navigation modules are related to the localization accuracy. Usually, in order to estimate the position of the robot, multiple sensor are used, which need to be calibrated and their measurements combined in one single estimate. ROAMFREE (Robust Odometry Applying Multisensor Fusion to Reduce Estimation Errors) [14] is a framework developed by Politecnico di Milano that offers:

- a library of sensor families handled directly by the framework
- an on-line tracking module based on Gauss-Newton minimization
- an off-line calibration suite which allows to estimate unknown sensor parameters

The framework is designed to fuse measurements coming from an arbitrary number of sensors. In order to maintain a general approach, it abstracts from the nature of the information sources, and it works with logical sensors, which are characterized only by the type of measurements provided. Therefore, the association between physical and logical sensors is not unique, since a single device can correspond to multiple logical sensors (e.g., IMU) or multiple physical sensors can cooperate to obtain a single measurement (e.g., stereo cameras). ROAMFREE divides the sensors in categories according to the type of measurements they provide: absolute position and/or velocity, angular and linear speed, acceleration and vector field (e.g, magnetic field, gravitational acceleration). For each of these categories an error

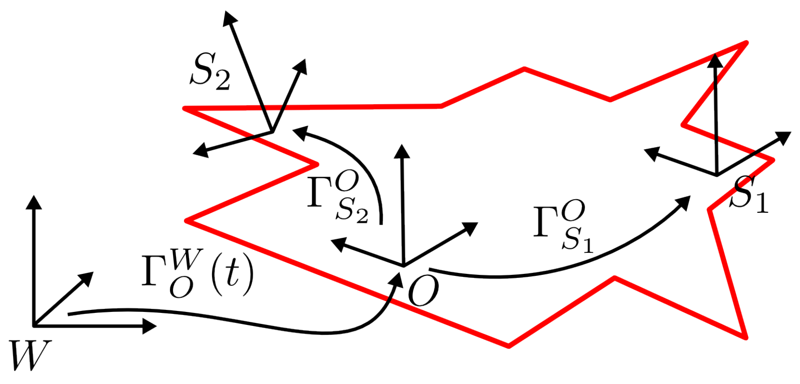


Figure 2.15: Reference frames and coordinate transformations in ROAMFREE

model exists, which relates the state estimate with the measurement data, taking into account all the common sources of distortion, bias and noise. Moreover, it is possible to define a set of the predefined calibration parameters using specific values or by letting the framework estimate them with an off-line formulation of the tracking problem.

ROAMFREE uses three reference frames: W, the fixed world frame, O, the moving reference frame, placed at the odometric center of the robot, and the i-th sensor frame, S_i, whose origin and orientation are defined with respect to O. The tracking module estimates the position and orientation of O with respect to W, i.e., Γ^W_O (Figure 2.15).

The tracking problem is formulated as a maximum likelihood optimization on a hyper-graph in which the nodes represent poses and sensor parameters and hyper-edges correspond to measurement constraints. An error function is associated to each edge in order to measure how well the values of the nodes connected to edge fit the sensor observations. The goal is to find a configuration for the poses and sensor parameters that minimizes the negative loglikelihood of the graph given all the measurements.

Let $e_i(x_i, \eta)$ be the error function associated to the i-th edge in the hyper-graph, where x_i is a vector containing the variables appearing in any of the connected nodes and η is a zero-mean Gaussian noise. Thus $e_i(x_i, \eta)$ is a random vector and its expected value is computed as $\bar{e}_i(x_i) = e_i(x_i, \eta)|_{\eta=0}$. Since e_i can involve non-linear dependencies, the covariance of the error is computed through linearization.

$$\Sigma_{e_i} = J_i \Sigma_\eta J_i^T|_{\eta=0} \tag{2.1}$$

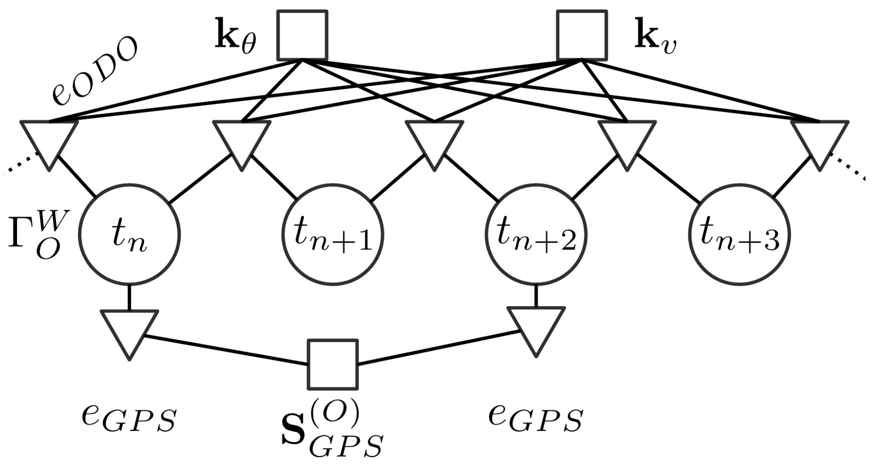


Figure 2.16: An instance of the hypergraph, with four pose vertices $\Gamma_O^W(t)$ in circles, odometry edges e_{ODO} (triangles), two shared calibration parameters k_v and k_θ(squares), two GPS edges e_{GPS} and the GPS displacement $\mathbf{S}_{GPS}^{(O)}$

where Σ_η is the covariance matrix of η and J_i is the Jacobian of e_i with respect to η. The optimization problem is defined as follows:

$$\mathcal{P} : \arg\min_x \sum_{i=1}^{N} \bar{e}_i(x_i)^T \Omega_{e_i} \bar{e}_i(x_i) \tag{2.2}$$

where $\Omega_{e_i} = \Sigma_{e_i}^{-1}$ is the i-th edge information matrix and N is the total number of edges. If a reasonable initial guess for x is known, a numerical solution for the problem can be found by means of Gauss-Newton or Levenberg-Marquardt algorithms.

In order to build the graph it is necessary to define a master sensor, with a high frequency, and for which it is possible to predict $\Gamma_O^W(t+\Delta t)$ given the last pose estimate available, $\breve{\Gamma}_O^W(t)$, and its measurement $z(t)$. Each time a new reading for this sensor is available, we instantiate a new node $\Gamma_O^W(t+\Delta t)$ using the last pose estimate available, $\Gamma_O^W(t)$, and $z(t)$ to compute an initial guess for this node. $z(t)$ is also employed to initialize an odometry edge between poses at time t and $t + \Delta t$. Each time a new measurements is available, their corresponding edge is inserted into the graph between the two nodes with the nearest timestamp. The graph optimization approach can be used to solve both the on-line position tracking problem, in which the requirements are related to precision and robustness, and the off-line calibration problem, in which the goal is to determine the sensor parameters directly form data.

A general framework for graph optimization, called g^{2}o, solves the optimization problem, and it is reported to solve graph problems with thousands of nodes in fractions of a second. Anyhow, for real time online tracking, it is necessary to define a finite time window and discard the older poses to avoid an excessive increase of computational load. Conversely, during off-line calibration, a set of the parameter nodes is chosen for estimation and the graph containing all available measurements is considered. The ROAMFREE library provides a simple interface that allows adapting the environment to the specific needs of each robot. It is possible to add logical sensors, choose the master, define the time window and the execution frequency, and more. Moreover, a ROS wrapper is available that subscribe to the sensors topics and periodically broadcast the estimated position using `tf`.

Chapter 3

Robot platform

In this chapter we provide a description of the robot platform, consisting in the vehicle, all the sensors and all the high-level and low-level control systems.

3.1 Vehicle

The vehicle used as a base for the development of the system is the Grizzly 700, a commercial all-terrain vehicle (ATV) produced by Yamaha. The Grizzly 700 is a utility ATV and it is specifically designed for agriculture work. As a result it is equipped with knobby tires and it has a total load capacity of 130 Kg. It is powered by a single cylinder 686cc 4-stroke engine with electronic fuel injection and it has an output of 46 HP (33.8 kW). The ATV features a fully automatic CTV type drive (Constantly Variable Transmission) which uses a low maintenance belt to transfer the power from the engine to the gearbox. It is possible to drive the vehicle with a two-wheel drive (2WD) or four-wheel drive (4WD) configuration. The braking system is hydraulic with two dual disc brake on front and rear wheels. A 12 V battery, with a capacity of 18 Ah, powers the electrical system and it is recharged by a 35 A alternator. Moreover, there are two auxiliary 12 V batteries, with a capacity of 38 Ah, used to power all the additional instrumentation placed on the vehicle. The tires used are Dunlop/KT421 (standard AT25 x 8-12) on front and Dunlop/KT425 (standard AT25 x 10-12) on rear. For the purpose of the project, the original chassis of the Grizzly 700 was removed and replaced with an aluminum platform used as a base for all the electronic equipment.

The steering system of the vehicle has been modified to allow the auto-

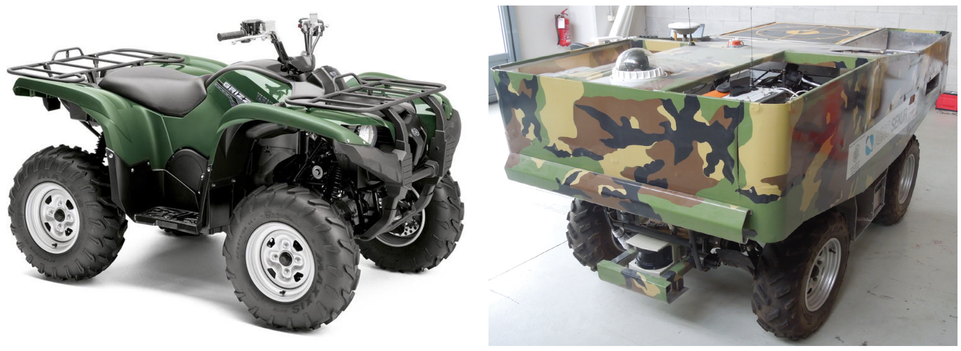


Figure 3.1: The original Grizzly 700 (left) and how the vehicle appears after the customizations (right)

matic driving. The EPS system was removed, interrupting the connection between the handlebar and wheels, then the steering column was replaced with a DC-motor (Intecno ND180.240 motor and Intecno NDP180/813 reducer) which controls the pitman arm and, consequently, the wheels. The actuation of two stepper motor, B&R 80MDP4.300S000-01, controls the accelerator and the braking system. Regarding the braking system, since the motor pulls the brake lever near the right foot-board, only the rear brake is controlled.

3.2 Sensor

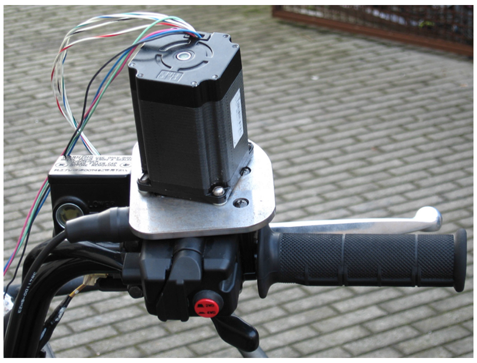
The robot uses the sensors to perceive the environment. The measurements are used to estimate the position and to perform other autonomous tasks. Here we provide a description of each sensor.

3.2.1 Global positioning system

In order to have an absolute position used as a ground truth in simultaneous localization and mapping experiments and for global localization, the robot is equipped with a GPS; the model is a Trimble 5700. It is a rugged device designed to withstand the conditions that typically occurs in field operations, it is coupled with an external antenna to maximize the quality of the measurement.

The GPS alone provides sub-meter accuracy real-time positioning using pseudorange corrections with less than 20 ms latency, these measurements can be enhanced using the RTK correction, which achieve an accuracy in

(a) The Intecno ND180.240 motor that controls the steer


(b) The B&R 80MDP4.300S000-01 stepper motor that controls the accelerator

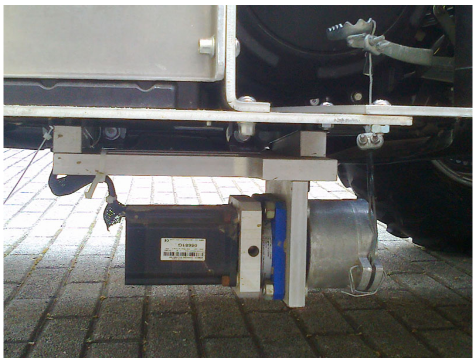

(c) The B&R 80MDP4.300S000-01 stepper motor that controls the brake

Figure 3.2: The three motors that controls the accelerator, the brake and the steer

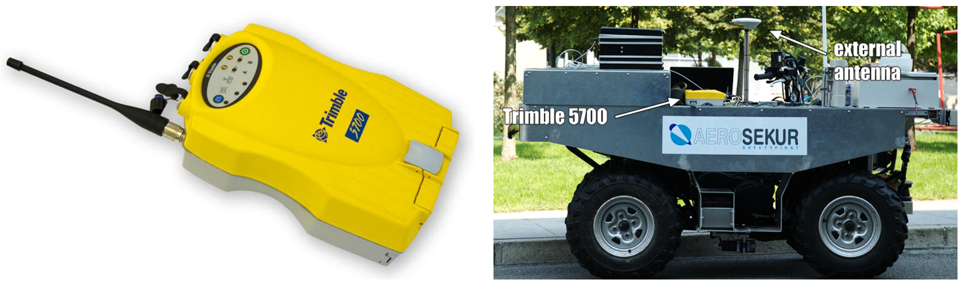


Figure 3.3: The Trimble 5700 (left) and its position on the robot with the external antenna (right)

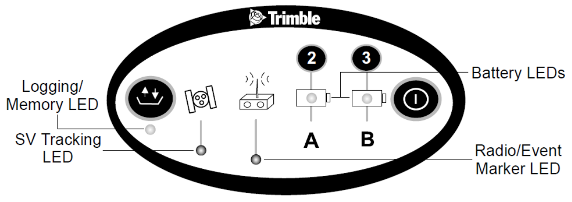
 Logging/ Memory LED SV Tracking LED 2 3 Battery LEDs A B Radio/Event Marker LED		
Logging/memory	On Off	Data is being logged Data is no being logged
SV tracking	Slow flash Fast flash Off	Tracking four or more satellites Tracking three or more satellites Not tracking any satellites
Radio	Slow flash	A CMR packet marker has been received
Battery	On Fast flash Off	Healthy Low power No power source

Table 3.1: LEDs behaviour of the Trimble 5700.

the order of centimetres. In our setup, the GPS alone is used and no RTK accuracy is obtained.

The connection between the GPS receiver and computer is serial, using the RS-232 standard. Since the computer is not equipped with a serial port, a serial-to-USB adapter is used. The Trimble 5700 steams its measurements using the NMEA standard at 5 Hz. In this standard each message is an ASCII string with a specific format. Each sentence's starting character is a dollar sign, all the data fields are comma-separated and a newline terminates the message. The NMEA standard includes various messages with a fixed number of fields, the first one is a code to identify them. Messages coming from a GPS, like in our case, have the GP prefix, the following is an example of a fix message:

```
$GPGGA,172814.0,37.46,N,12.26,W,2,6,1.2,18.893,M,-25.669,M,
2.0,0031*4F
```

In this case the fields are:

- Timestamp (UTC) of the position fix (`172814.0`)
- Latitude with its direction (N for North, S for South) (`37.46,N`)
- Longitude with its direction (E for East, W for West) (`12.26,W`)
- GPS quality indicator (from 0, no fix, to 5, RTK correction) (`2`)
- Number of satellites tracked (`6`)
- Horizontal dilution of precision (`1.2`)
- Height with its unit of measurement (M for meters)(`18.893,M`)
- Geoidal separation with its unit of measurement (M for meters) (`-25.669,M`)
- Age of differential GPS data record (`2.0`)
- Reference station ID (`0031`)

The hexadecimal number after the asterisk is the checksum. As it is possible to see from the structure of the message, the position is expressed in the geodetic coordinate system, which uses latitude, longitude and altitude.

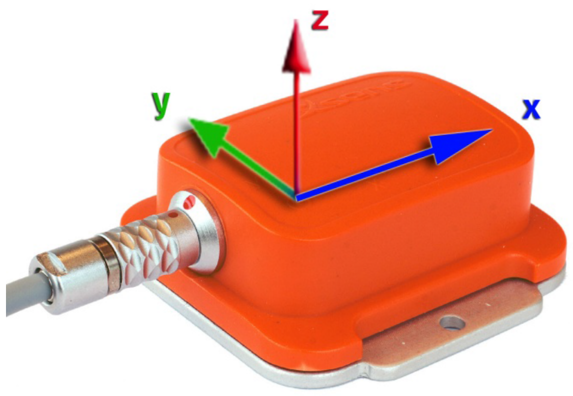


Figure 3.4: MTi with sensor-fixed coordinate system overlaid

3.2.2 Inertial measurement unit and magnetometer

The robot is equipped with an xsens MTi, which is a miniature inertial measurement unit (IMU) with integrated 3D magnetometers (3D compass). The IMU is composed by accelerometers and gyroscopes, it can calculate roll, pitch and yaw in real time, as well as outputting calibrated 3D linear acceleration and rate of turn. The magnetometer provides 3D earth-magnetic field data. The sensor provides measurements at a frequency up to 100 Hz.

All calibrated sensor readings (accelerations, rate of turn, earth magnetic field) are in the right handed coordinate system as defined in Figure 3.4. This coordinate system is body-fixed to the device. The Earth-fixed coordinate system used as a reference to calculate the orientation is defined as a right handed coordinate system with:

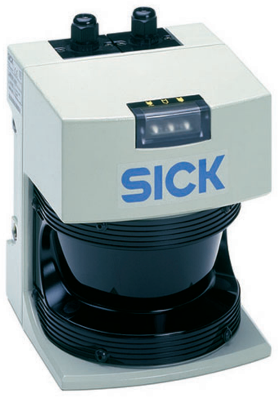
- x positive when pointing to the local magnetic North.
- y according to right handed coordinates (West).
- z positive when pointing up.

In this work the MTi is used to acquire raw magnetic field measurements and raw rate of turn. The specification associated with those measurements are listed in 3.2.

The xsens is placed on the robot in a central position, between the vehicle and the sensor there is a piece of foam used as a mechanical filter to remove vibration. The connection between the MTi and the computer is done by a USB interface, which is used for both data transfer and power.

	rate of turn	acceleration	magnetic field
Unit	[deg/s]	[m/s^2]	[mGauss]
Dimensions	3 axes	3 axes	3 axes
Full Scale [units]	+/-300	+/-50	+/-750
Linearity [% of FS]	0.1	0.2	0.2
Bias stability [units 1σ]	1	0.02	0.1
Scale factor stability [% 1σ]	-	0.03	0.5
Noise density [units /$\sqrt{\text{Hz}}$]	0.0513	0.002	0.5 (1σ)
Alignment error [deg]	0.1	0.1	0.1
Bandwidth [Hz]	40	30	10
A/D resolution [bits]	16	16	16

Table 3.2: Calibrated data performance specification.



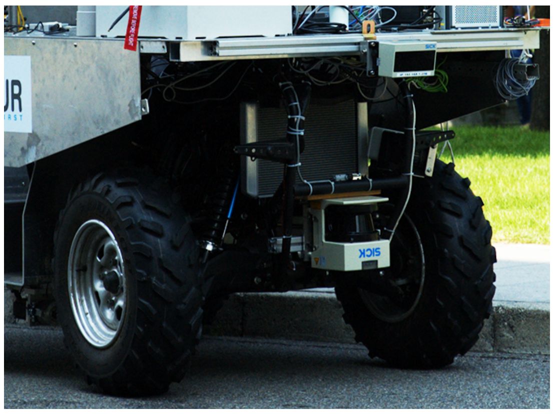


Figure 3.5: The SICK LMS291 laser scanner (left) and where it is mounted on the robot (right)

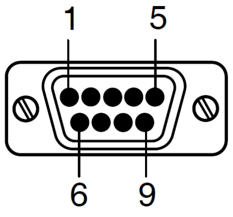
	Pin	Signal	Data interface
	1	RD-	RS 422
	2	RD+/RxD	RS 422/RS 232
	3	TD+/TxD	RS 422/RS 232
	4	TD-	RS 422
	5	GND	Ground
	6	n.c.	
	7	Brigde to pin 8	Enables the RS 422
	8	Brigde to pin 7	
	9	n.c.	

Table 3.3: Pin assignment of the 9-pin serial inferface.

3.2.3 Laser scanner

In order to perceive the environment near the vehicle and to detect unpredictable obstacles, the robot is equipped with a laser scanner; the model is Sick LMS291. The sensor can measure the distance from an object on a plane up to 80 meters, in a range of 180°with a resolution of 1°, 0.5°or 0.25°depending on the configuration. The laser can transmit measurements to the computer at various rates, starting from 9600 Bd up to 500 kBd, to achieve the highest transmission speed and receive real time measurements it is necessary to connect the sensor using a RS-422 serial cable with a custom wiring. Since the computer used on the robot has no serial connection, we use a custom-built RS-422-to-USB adapter with the wiring described in Table 3.3

The LMS291 requires an operating voltage of 24 V with a power consumption of less than 20 W, power is supplied using another custom-wired serial cable, connected directly to the electric system of the robot. In this cable only two pins are used: number 1 for ground and number 3 for power.

The laser is mounted on the robot upturned between the front wheels. Given its orientation, the laser beam turns towards the right and its coordinate frame has the z-axis pointing down, the x-axis pointing ahead and the y-axis pointing right.

3.2.4 Cameras

Mounted on the front of the robot there are two cameras, both pointing forward and parallel to each other. The model is Prosilica GC750C. The cameras use an Ethernet connection to communicate with the system, they

Figure 3.6: The Prosilica GC750C camera

support a Gigabit connection to stream uncompressed images at high frame rate. They are powered at 12 V with a custom wiring that connects them to the electric system of the robot, a DC-DC converter decouples the two parts for increased security. The cameras are connected to each other, this is necessary to have synchronous images in the stereo configuration. The master camera sends a signal to the slave camera using this connection, while the master streams frames continuously, the signal trigger the capture of each frame in the slave. Since the connection is symmetrical, both cameras can be master, which one has the master role is defined via configuration. In our system the right camera streams images continuously and act as a master, while the left camera receives the trigger signal.

Each Prosilica mounts a 1/3" Apatina MT9V022 CMOS sensor and it can stream color images up to 67 frames per second with a resolution of 752 x 480 pixels. We set our cameras to stream greyscale raw images, with a full resolution, at 5 frames per second, when connected using Fast Ethernet, or 20 frames per second, when using Gigabit Ethernet.

The coordinate frame of the cameras it the one commonly used for this type of sensor, which means x is pointing right, y downwards and z from the camera into the scene.

As stated before, an Ethernet connection connects the cameras to the system, it is not necessary to define static IPs to identify the devices, since each one has a unique ID that can be used to establish a connection. In the current setup, the ID of the right camera is 45032 and for the left one is 45031.

3.3 Low-level control system

An X20 system produced by the Austrian company B&R realizes the low-level control system. It is a programmable logic controller (PLC) used in

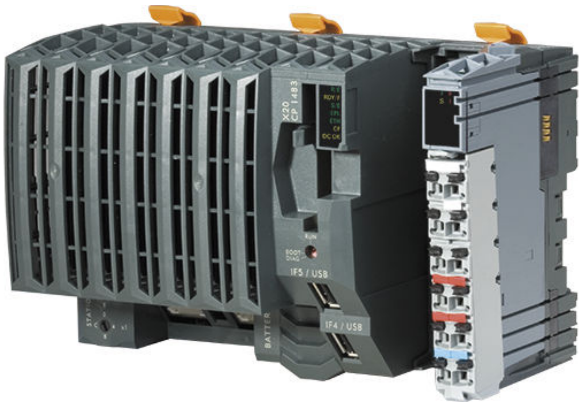


Figure 3.7: The X20 system

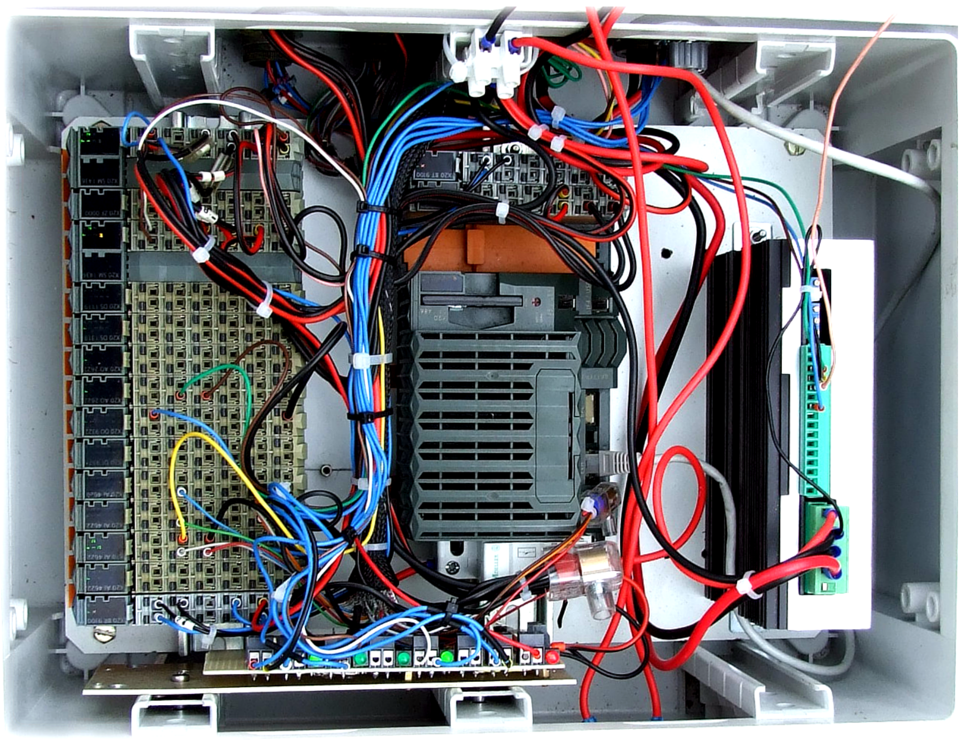


Figure 3.8: The low-level control system of the vehicle, with the X20 in the center and the modules on the left

industrial applications to monitor and control various devices (e.g., sensors, actuators, industrial PC, etc.). The system, powered by 24 V, is equipped with an Intel Celeron processor at 650 MHz, 64 MB of RAM, 1 MB of SRAM and a backup battery to maintain global variables. A CompactFlash memory card (up to 8 GB) acts as mass storage to contain the operating system and the working data. The system is equipped with the following connection ports: Ethernet, two USB ports and an RS232 port. The X20 is modular, it is possible to assemble various units, and each one can manages I/O signals (analog or digital), controls stepper motor or acquires data from various sensors. Here a brief description of the modules used:

- **Analog output module AO2622**: the module is equipped with two outputs with 12-bit digital converter resolution. It is possible to select between a current (0-20 mA) or voltage (+/- 10 V) signal. One of these modules is used to send the current setpoint to the motor controlling the steer.

- **Digital output module DO9322**: the module is equipped with twelve outputs for 1-wire connections, each one can have a value of 24 V (logic level high) or 0 V (logic level low). The outputs are controlled independently with Boolean values or in groups with strings. This module is used to send to the steer motor the Enable signal.

- **Stepper motor module SM1436**: the stepper motor module is used to control stepper motors with a rated voltage of 24 to 39 V at a motor current up to 3 A. Additionally, this module has four digital inputs that can be used as limit switches or as encoder inputs. In the low-level control system there are two of these modules, one is used to controls the stepper motor which moves the brake pedal, the other one controls the motor associated with the accelerator lever. One of the two modules also receives the signal coming from the phonic wheel (speed sensor) of the vehicle.

The connection between the low-level control system and the rest of the hardware is Ethernet; the X20 system has a static IP, which is 192.168.0.1. The same channel is used to transmit setpoints and receive odometry messages. The messages are strings with a specific format. A valid sentence is a list of values separated by commas between angular brackets. The next chapter contains a more detail description of these messages.

Figure 3.9: The Acer aspire 5742G

3.4 Computer

The computer receives all sensor readings, implements localization algorithms, act as an interface between human and vehicle during manual drive and sends command to the low-level control system during autonomous drive. Originally, the robot was equipped with an industrial pc mounted directly on the vehicle, but to simplify the development of the new software it has been replaced with a commercial laptop. The model is an Acer aspire 5742G, it is equipped with an Intel Core i5-450M processor running at 2.6 GHz, 4 GB of RAM and a ATi HD5470 video card. Concerning the storage, we replaced the original mechanical hard disk with a solid-state drive to avoid damage caused by the vibration of the engine; it has a capacity of 60 GB.

The on-board operating system is Ubuntu-Linux. This choice is driven by the OS full compatibility with ROS and by the availability in the Ubuntu repository of most of the ROS packages and add-ons. The version is Ubuntu 13.10 because it was the newest version available compatible with the ROS distribution used, ROS Hydro Medusa, when we stared developing this project. The computer has a static IP, which is 192.168.0.100.

3.5 Teleoperation

The robot can be teleoperated using a wireless joypad. The joypad used is a Microsoft Xbox 360 Wireless Controller, but any device supporting XInput API is compatible with the system. The controller is connected to the computer with an USB wireless receiver with a range of 9 meters. The joypad has two analog sticks with an angular range of 360°, a 4-way digital

Figure 3.10: Microsoft Xbox 360 Wireless Controller

pad, two analog triggers, ten digital buttons, two of which activated by pressing the analog sticks. The device is used to access various functionality: manual drive, changing the internal state of the system and activate specific behavior in some software modules. When driving manually the operator controls the speed of the vehicle using the left analog, the speed increase by tilting it up, to a maximum of 5 m/s. Tilting the left analog backward activates the brake. The right analog controls the steer, tilting it left to make a left turn and vice versa, in a range from -20°and +20°.

In addition to teleoperation, the controller also is used to control some of the software modules, a detailed description about this is provided in Chapter 4.

Chapter 4

Software architecture

This chapter describes the new software architecture designed and developed for the robot within this thesis. First, a general description of the architecture is given, then we describe how the communication with the low-level control system is implemented. A summarization of the modules used to communicate with the sensors follows. The last part is about the autonomous drive and explains how the robot is localized, the path is generated, and the actuators are controlled.

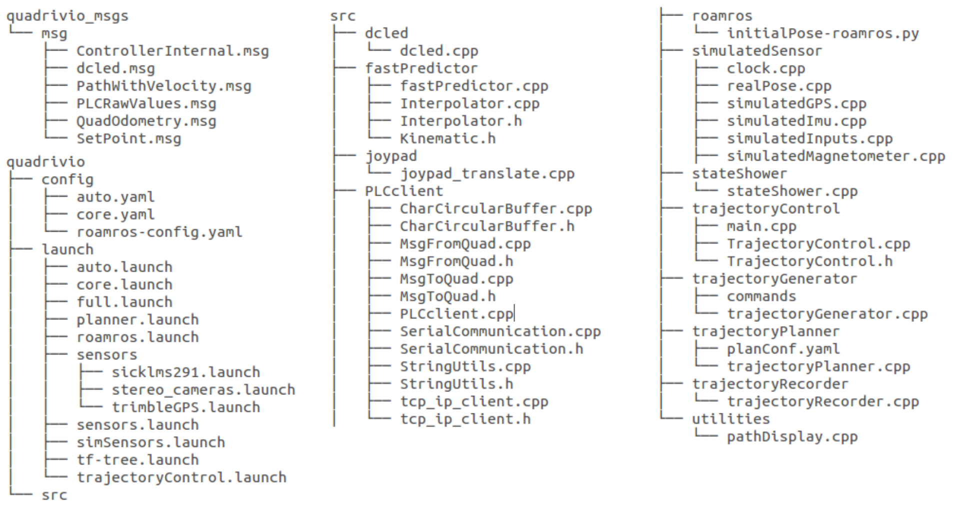
4.1 Architecture overview

4.1.1 General structure

In order to develop the architecture of our robot we used ROS, an open source framework which as recently become the de facto standard in many robotic applications. One of the main characteristics of ROS is its modularity, implemented through a structure based on packages, nodes, messages and services. We used these features to design a highly modular architecture.

The mains reasons behind this choice are, fundamentally, two; first, when working with a prototype, it is important to have the possibility to add or remove parts (e.g., sensors) with ease and, consequently, to enable or disable the associated software modules without compromising the entire system. This is realized in our architecture by the use of independent nodes for each functionality and by a hierarchical structure of ROS launch files.

The second reason that drove our decision is the integration with a simulated environment. Given the characteristics of the robot, a realistic simulation is fundamental, even more important is to have a seamless interchangeability between the real vehicle and the simulated environment.



```
quadrivio_msgs
└── msg
    ├── ControllerInternal.msg
    ├── dcled.msg
    ├── PathWithVelocity.msg
    ├── PLCRawValues.msg
    ├── QuadOdometry.msg
    └── SetPoint.msg
quadrivio
├── config
│   ├── auto.yaml
│   ├── core.yaml
│   └── roamros-config.yaml
├── launch
│   ├── auto.launch
│   ├── core.launch
│   ├── full.launch
│   ├── planner.launch
│   ├── roamros.launch
│   ├── sensors
│   │   ├── sicklms291.launch
│   │   ├── stereo_cameras.launch
│   │   └── trimbleGPS.launch
│   ├── sensors.launch
│   ├── simSensors.launch
│   ├── tf-tree.launch
│   └── trajectoryControl.launch
└── src
```

```
src
├── dcled
│   └── dcled.cpp
├── fastPredictor
│   ├── fastPredictor.cpp
│   ├── Interpolator.cpp
│   ├── Interpolator.h
│   └── Kinematic.h
├── joypad
│   └── joypad_translate.cpp
├── PLCclient
│   ├── CharCircularBuffer.cpp
│   ├── CharCircularBuffer.h
│   ├── MsgFromQuad.cpp
│   ├── MsgFromQuad.h
│   ├── MsgToQuad.cpp
│   ├── MsgToQuad.h
│   ├── PLCclient.cpp
│   ├── SerialCommunication.cpp
│   ├── SerialCommunication.h
│   ├── StringUtils.cpp
│   ├── StringUtils.h
│   ├── tcp_ip_client.cpp
│   └── tcp_ip_client.h
```

```
├── roamros
│   └── initialPose-roamros.py
├── simulatedSensor
│   ├── clock.cpp
│   ├── realPose.cpp
│   ├── simulatedGPS.cpp
│   ├── simulatedImu.cpp
│   ├── simulatedInputs.cpp
│   └── simulatedMagnetometer.cpp
├── stateShower
│   └── stateShower.cpp
├── trajectoryControl
│   ├── main.cpp
│   ├── TrajectoryControl.cpp
│   └── TrajectoryControl.h
├── trajectoryGenerator
│   ├── commands
│   └── trajectoryGenerator.cpp
├── trajectoryPlanner
│   ├── planConf.yaml
│   └── trajectoryPlanner.cpp
├── trajectoryRecorder
│   └── trajectoryRecorder.cpp
└── utilities
    └── pathDisplay.cpp
```

Figure 4.1: The structure of the packages `quadrivio` and `quadrivio_msgs`

The software architecture has been designed introducing a decoupling layer between the real robot and the high-level perception and control software. This layer can be removed and substituted with the simulation; a detailed description on how the architecture interact with the simulation is in Chapter 5. Briefly, we created two packages for the core system, `quadrivio` and `quadrivio_msgs`, the first one contains all the nodes, while the other one contains the messages. Figure 4.1 shows a simplified structure of the packages, it shows only the source, launch and configuration files. Splitting packages in nodes and messages, or services, is a standard ROS practice. Inside the `quadrivio` package each node, or each group of nodes, has a separate folder, this structure creates a well-organized workspace where it is simple to disable unused node, for example when switching between the real robot and the simulated environment.

Other nodes developed during this project but not explicitly for it, for example sensors nodes or the internal state machine, have their own packages.

4.1.2 Main modules

Figure 4.2 shows the main modules of the high-level software architecture and their relations with the external sensors and the vehicle servomechanisms. The core part of the perception architecture consists in the localization node (Section 4.5.2), which is based on ROAMFREE. This framework provides pose tracking fusing the information coming from an arbitrary number of sensors. In the current configuration the localization module estimates

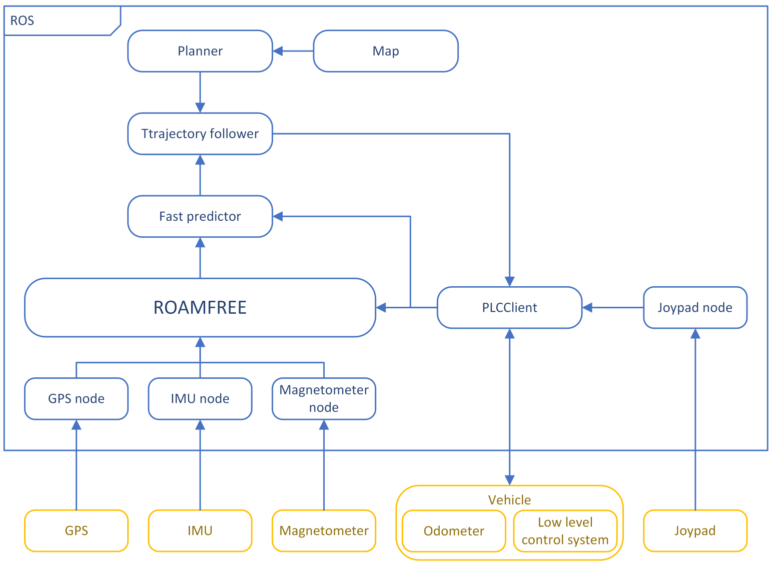


Figure 4.2: The main module composing the architecture

the robot poses exploiting vehicle kinematic data (i.e., the handlebar position and the rear wheel speed), GPS, magnetometer, and the gyroscopes in the inertial measurement unit. These measurements are provided to the localization module by the nodes that act as drivers (Section 4.4) for the sensors and communicate directly with the physical device.

The pose estimate is generated by the localization node at a frequency of 10 Hz. However, due to the latencies introduced by the ROS network, delays in the trajectory control loop which affect the system stability can occasionally arise. In order to prevent the detrimental effects of these delays, we introduced a predictor node (Section 4.5.3). This node computes a prediction of the future robot pose at a frequency of 50 Hz; this prediction is based on the latest available global pose estimate and on the integration of the Ackermann kinematic model with the kinematic readings from the vehicle.

Given a map and a goal, a planner node (Section 4.6.3), based on the SBPL library , produces a global path, which is then fed to a lower level trajectory following module (Section 4.7). This module computes setpoints for the vehicle speed and handlebar angle, based on the current pose and velocity estimates, and on the planned trajectory. These setpoints are sent to the low-level regulators by a ROS node (Section 4.3) communicating with the PLC, which additionally acts as a multiplexer between the autonomous drive and the manual setpoints, depending on the current operating mode.

4.2 State machine

In order to manage the functionalities of the robot, the software architecture is design around an internal finite state machine. The possible states are MANUAL, ASSISTED, AUTO, SAFE and HALT. Each state corresponds to one of the possible modes in which the robot can operate:

- MANUAL: in this state, the robot is driven manually controlled by the joypad. `PLCClient` discards all the setpoints coming from any source other than JOYPAD, these are sent directly to the low-level control system.
- ASSISTED: currently this state behave exactly like MANUAL. It exist because in the future the robot will have collision detection features. In this state, these features will be active to assist the manual drive.
- AUTO: The follower is active and the robot is driven autonomously, any trajectory published is executed immediately. PLCClient forwards to the low-level control system only setpoints having the source AUTO.
- SAFE: The robot stands still, a neutral setpoint is sent to the low-level control system. The vehicle cannot be operated until a different state is selected.
- HALT: This is the starting state of the robot and its emergency state. Reachable from each other state, it stops the vehicle and put the low-level control system in its own halt state.

The transition between two states is done using the controller by pressing one or more buttons simultaneously. The state-transition function is defined in the following way:

- from HALT to SAFE by pressing BACK and START at the same time;
- from SAFE to:
  - MANUAL by pressing B;
  - ASSISTED by pressing RB;
  - AUTO by pressing LT and RT at the same time;
  - HALT by pressing BACK and START at the same time;
- from MANUAL, ASSISTED or AUTO to:
  - SAFE by pressing LB

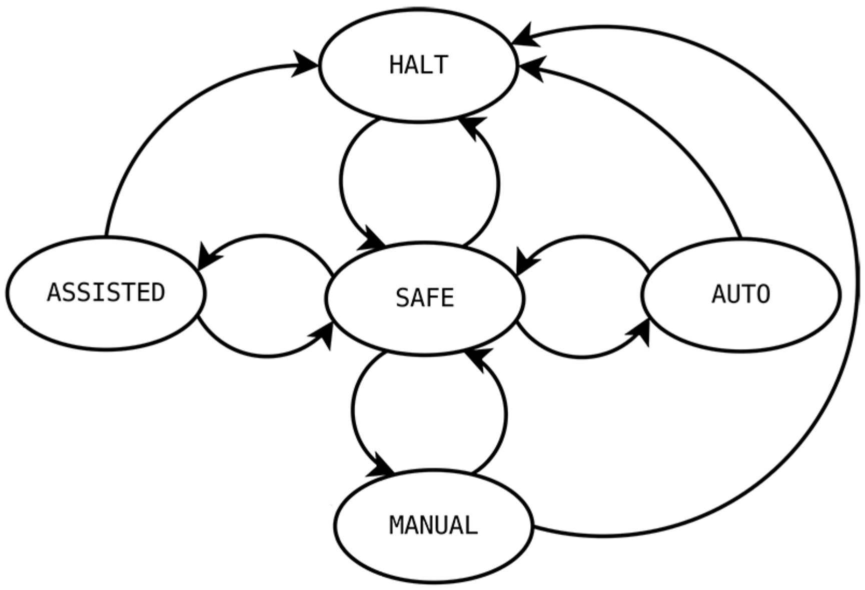


Figure 4.3: The internal state machine

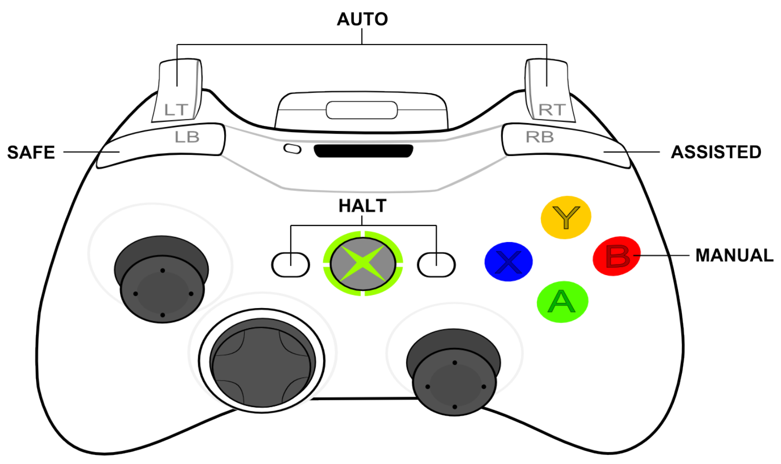


Figure 4.4: Button used to change the state of the internal state machine

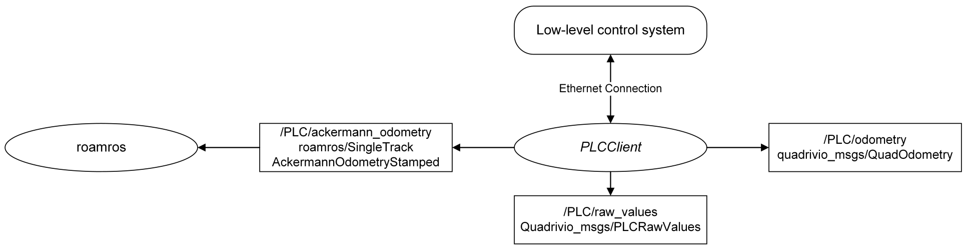


Figure 4.5: `PLCClient` node and its communication channels

– HALT by pressing BACK and START at the same time;

In order to implement this state machine in our architecture we have adopted a package developed by Politecnico di Milano, called `heartbeat`. This package offers two main functionalities, first to manage the internal state machine, second to keep updated all the nodes on the current state of the system. This is done using a client-server paradigm, a node act as a server and manage the state machine, other nodes defines a client object that can retrieves the current state or change it. In order to start the server it is necessary to launch the `server_node`, found inside the `heartbeat` package. This node manages the state machine and the requests coming from the clients. Each node that want to know or change the current state of the system have to instantiate a HeartbeatClient object. It is possible to initialize the object in two ways, with and without a timeout. In the first case the node can retrieve the current state using the method `getState()` and modifying it using `setState(newState)`. In the second, other than these functionalities, the node must periodically call a method called `alive()`. If more than the time defined in the initialization passes between two calls, the node is considered malfunctioning, this is notified with an error message and the system is shut down. Currently only one node use this functionality, `PLCClient` (Section 4.3).

4.3 Low-level interface

As described in Chapter 3 the low-level control system is connected to the computer via Ethernet. A node called `PLCClient` act as an interface between the high-level and low-level system.

The `PLCClient` receives messages form the X20 system; these messages contain information about the status of the vehicle. They are strings with a specific structure: each one starts with a $<$ symbol and ends with a $>$ symbol, the fields are comma-separated. Every message has a fixed amount of fields with a specific order: time, steer SP, steer PV, steerCurrent SP, steerCurrent PV, speed SP, speed PV, throttle SP, throttle PV, throttle CS, brake SP, brake PV, brake CS, stateMachine. After processing the message, the values are published directly on a topic called `/PLC/raw_values`, this is to expose the internal values of the low-level control system on a higher level and simplify tests and validations.

From the content of these messages, `PLCClient` derives the odometry of the vehicle, which is published on two different topic. One is `/PLC-/ackermann_odometry`, the message published is `roamros_msgs/Single-TrackAckermannOdometryStamped`, which contains the raw value of speed and steer of the vehicle. Moreover, the message has a header, a field including a sequence number, a timestamp, and a frame id, which is a string identifying the reference frame. The other one is `/PLC/odometry`, this message (`quadrivio_msgs/QuadOdometry`) contains speed, steer and brake, differently from the previous one, the values are expressed in m/s, for speed, and radians, for steer. The conversion from the raw measurements received from the X20 system to values in standard units of measurement is done using three constants defined as parameters in the parameter server: `ksteer`, estimated as -0.5622, `psisteer`, estimated as -0.0061, and `kspeed`, estimated as 0.0584. These values are estimated with the off-line calibration suite provided by the ROAMFREE library (see Section 2.6) using data recorded from the vehicle during manual drive. The conversion formulae are:

$$speed_{m/s} = K_{speed}\, speed_{odo} \tag{4.1}$$

$$steer_{rad} = K_{steer}\, steer_{odo} + \psi_{steer}. \tag{4.2}$$

The node `PLCClient` also has the task to send to the low-level control system setpoints generated by the high-level system, whether they are from manual or autonomous drive. Setpoints are published by other nodes on a topic called `/setpoint`, the message used is `quadrivio_msgs/SetPoint`, which contains four fields: steer, speed, brake and source. The last field identifies the source of the setpoint, if it comes from the manual drive (JOYPAD) or the follower (AUTO). `PLCClient` reads these messages and, depending on the internal state of the system, forwards them to the low-level control system. The messages sent to the X20 system are strings with the following structure: starts with a $<$ symbol, ends with a $>$ symbol and the fields are comma-separated. Each string contains the following values in this order: brake, speed, steer angle, steer speed and command. The last one is used to change the internal state of the low-level control system. The states are identified by integers numbers, and the node can set two different values:

Name	Value	Description
quadPLCIpAddress quadPLCPort frequency	127.0.0.1 20001 20 Hz	Parameters of the connection to the low-level control system. Address, port and frequency of the messages.
heratbeatTimeout	0.5 s	Maximum time is it possible to wait for a message from the low-level control system.
kspeed ksteer psisteer	0.0584 -0.5622 -0.0061	Parameters used to convert the odometry to to standard units of measurement.

Table 4.1: Paramteres used to configure the `PLCClient` node.

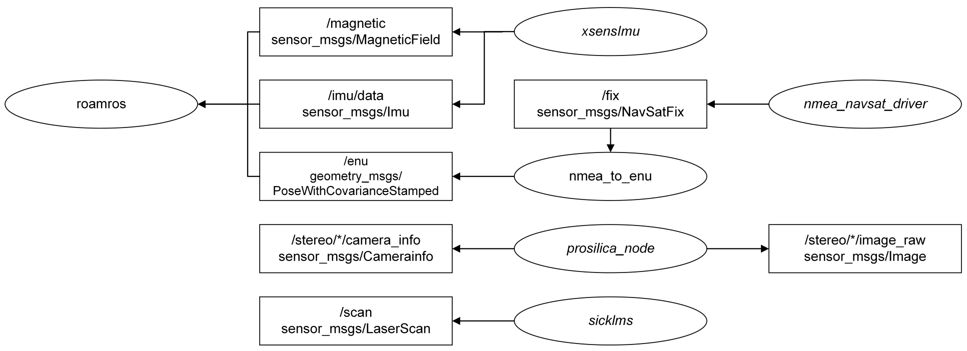


Figure 4.6: Sensors nodes and their topics

20, a safe state where no setpoint is forwarded to the actuators, and 40, the normal operating state of the system.

Given the crucial role of this node in the architecture, it has a thigh coupling with the internal state machine. Each time a new odometry message is received `PLCClient` notify to the state machine that the connection to the low-level control system is still active. A missing notification for a prolonged time cause the system to shut down, in order to prevent unexpected behaviours. As stated before, the node is configurable via parameters, which are specified in a configuration file called `core.yaml`. Various parameters, described in Table 4.1, are used to configure the node.

4.4 Sensors

This robot, like most of the robots, is equipped with various sensor. In this section, we describe the ROS nodes used to retrieve data from these devices.

Name	Value	Description
STATUS_NO_FIX	-1	Fix unavailable
STATUS_FIX	0	Unaugmented fix available
STATUS_SBAS_FIX	1	Fix with satellite-based augmentation available
STATUS_GBAS_FIX	2	Fix with ground-based augmentation available

Table 4.2: Possible values of the status field in the `sensor_msgs/NavSatFix` message.

4.4.1 GPS

As described in Chapter 3 the output of the Trimble 5700 is in NMEA format, which is based on strings with a well-defined structure. ROS already has in its repository a node able to read these messages and convert them in standard ROS messages. It is contained in a packaged called `nmea_navsat_driver`. The node, called `nmea_serial_driver`, reads NMEA sentences directly from the GPS, converts them in a `sensor_msgs/NavSatFix` message, and publish them on a topic called `/fix`. The message contains a standard ROS header, the coordinates, expressed as latitude, longitude (both in degrees), and altitude (in meters), a status field and the covariance matrix. The status field contains information about the quality of the fix and the number of satellites used. Since the message is always sent, even when there is no fix, this field is used also to notify that the values are not valid. The last field is a 3x3 matrix filled with covariance values derived directly from the NMEA sentences.

Two parameters have to be set to have this driver working properly: the logical port corresponding to the device and its baud rate. The baud rate is a value defined by the device, and in our case it is set at 57600 baud. Concerning the port, instead, since various devices are connected to the computer via USB, we cannot predict which virtual port will be assigned to the GPS. We solve this problem by writing an udev rule. It allows you to identify devices based on their properties, like vendor ID and device ID, dynamically, and to specify a name, which is given to it regardless of which port is plugged into. In our case, we built a symbolic link for the logical port of the GPS and use that link as a port name for the node. The name used to identify the device is `/dev/trimble`, and the rule is:

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="0557", SYMLINK+="trimble",
ATTRS{manufacturer}=="Prolific Technology Inc."
```

Since the `/fix` is given in geodetic coordinates (i.e., latitude, longitude and altitude) and ROAMFREE works with ENU coordinates (i.e. East, North and Up), node `nmea_serial_driver` is coupled with another one that implements this conversion. The node `nmea_to_enu` reads the message published by `nmea_serial_driver`, converts the coordinates and publish it on the topic `/enu`, using a `geometry_msgs/PoseWithCovarianceStamped` message. This message has a header, filled with the same values of the original one, information about the position (using three coordinates), information about the orientation (using a quaternion), and a covariance matrix, which is copied from the original message. Since this is a message about GPS coordinates, the orientation field is not used and the covariance matrix, which is 6x6, is only partially filled:

$$\begin{bmatrix} V(x) & C(x,y) & C(x,z) & 0 & 0 & 0 \\ C(y,x) & V(y) & C(y,z) & 0 & 0 & 0 \\ C(z,x) & C(z,y) & V(z) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.3}$$

where $C(x,y)$ is the covariance between x and y, defined as $E[(x-\mu_x)(y-\mu_y)]$, μ_x being the expected value of x. $V(x)$ is the variance of x, defined as $C(x,x)$. There is no information about the fix quality in this message, therefore when the status field notify that there is no fix, no message is published.

The conversion from the geodetic to ENU has two steps, first a conversion from geodetic to Earth-centered earth-fixed (ECEF), then from ECEF to ENU. Geodetic coordinates (latitude ϕ, longitude λ, height h) can be converted into ECEF coordinates using the following formulae:

$$X = (N(\phi) + h)\cos\phi\cos\lambda \tag{4.4}$$

$$Y = (N(\phi) + h)\cos\phi\sin\lambda \tag{4.5}$$

$$Z = \big(N(\phi)(1-e^2) + h\big)\sin\phi \tag{4.6}$$

Where:

$$N(\phi) = \frac{a}{\sqrt{1 - e^2\sin^2\phi}} \tag{4.7}$$

$$e^2 = f(2-f) \tag{4.8}$$

a is the major equatorial radius, and f is the flattening, both values are chosen by reference to the WGS84 datum, which is the standard reference ellipsoid used to model Earth by the Global Positioning System.

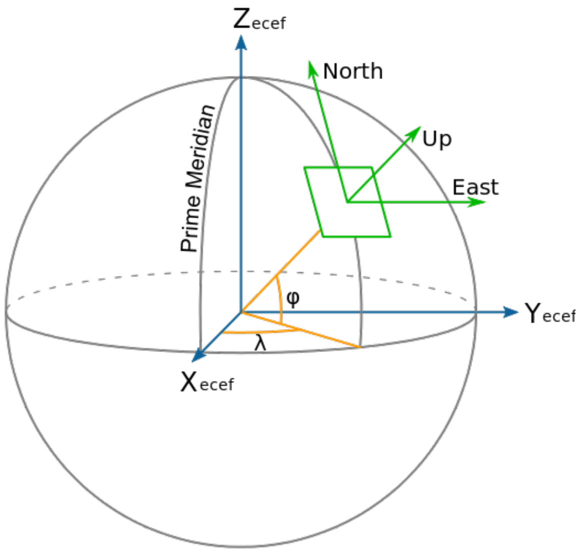


Figure 4.7: Geodetic (yellow), ECEF (blue) and ENU (green) coordinates

To transform from ECEF coordinates to the local coordinates a local reference point is necessary. In our node is defined using geodetic coordinates via parameters, this is the best solution because it can be set on the field using the coordinates from the GPS. Given the reference point in ECEF coordinate as X_r, Y_r, Z_r and the GPS as X_p, Y_p, Z_p then the vector pointing from the reference point to the GPS in the ENU frame is:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -\sin\lambda_r & \cos\lambda_r & 0 \\ -\sin\phi_r\cos\lambda_r & -\sin\phi_r\sin\lambda_r & \cos\phi_r \\ \cos\phi_r\cos\lambda_r & \cos\phi_r\sin\lambda_r & \sin\phi_r \end{bmatrix} \begin{bmatrix} X_p - X_r \\ Y_p - Y_r \\ Z_p - Z_r \end{bmatrix}. \tag{4.9}$$

4.4.2 IMU

The ROS repository already has a driver node for the xsens MTi, but two problems forced us to develop a new one. First, there was a connection problem, where the node could not always find the device, second, the message type used for magnetic field was incorrect, `geometry_msgs/Vector3Stamped` instead of `sensor_msgs/MagneticField`.

The developed node is based on a C++ library provided by the vendor, this library manages the connection with the device and the retrieval of the measurements. We developed a node that act as a wrapper for this library, publishing the raw data on the ROS topics: `/imu/data` and `/magnetic`. The first one contains data about the accelerometers and the gyroscopes. The message type is `sensor_msgs/Imu`, which has a header, a quaternion for the orientation and two vector of three elements for linear acceleration and angular velocity. The header is filled with a timestamp taken from the system (using `ros::Time::now()`), an increasing sequence number and "/imu" as the reference frame. Velocity and acceleration are expressed in rad/s and m/s^2, respectively.

The other message contains the Earth magnetic field. The message type is `sensor_msgs/MagneticField`, which has a header, filled with the same values of the other message, a vector of three elements for the magnetic field, expressed in mG, and a covariance matrix. The sensor provides no information about the covariance, so it is assumed to be the identity matrix. It is possible to define the frequency at which the sensor provides its measurements, currently it is set at 20 Hz, and this frequency must match the frequency of the node to avoid a delay in the messages.

As for the GPS node, it is necessary to define the logical port in the parameters; being impossible to predict which port the system will assign, so we used another udev rule. The name used to identify the device is `/dev/xsens`, and the rule is:

```
SUBSYSTEM=="tty", ATTRS{serial}=="XSV8BTGS", SYMLINK+="xsens",
ATTRS{idProduct}=="d38b"
```

4.4.3 Laser scanner

Natively the Sick LMS291 provides drivers only for Windows environments, but an open-source software package that implements stable C++ drivers and configuration tools is available. Moreover, the ROS repository supplies a wrapper for these drivers, resulting in a complete integration with our architecture. The package is called `sicktoolbox_wrapper`, while the driver node is called `sicklms`.

The node streams the measurements received by the sensors on a topic called `/scan`, using a `sensor_msgs/LaserScan` message. This message contains a standard ROS header, the string "/laser" used to identify the frame, a set of information regarding the configuration of the laser (e.g., scanning range and resolution), and an array of distances in meters measured by the sensor in each angular section, in our case the array contains 361 elements. The order of the values is counter-clockwise around the z-axis, in accordance to the convention used to measure angles. As stated in Chapter 3, the laser is mounted upturned, therefore the first values in the array contains the distance of a point at the extreme left of the robot. In order to complete the configuration of the node we have to set some parameters: the baud rate, set at 500000 Bd to obtain real time measurements, and the device path,

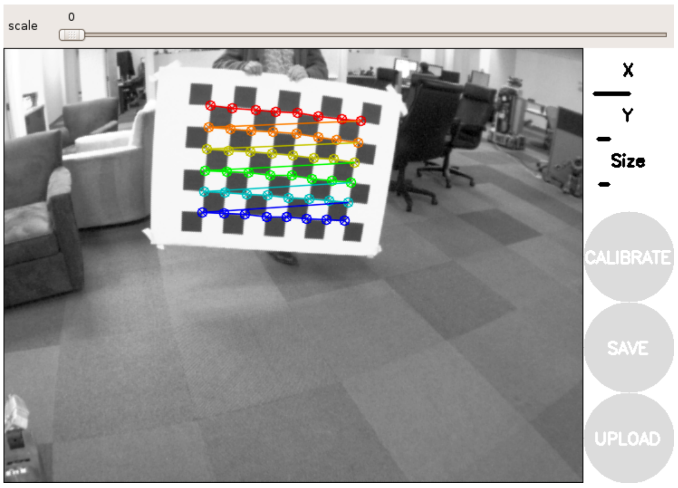


Figure 4.8: Example of the calibration procedure

defined as `/dev/lms291`. As with all other sensors connected via USB, we defined the following udev rule to identify univocally the laser, regardless of the port it is connected to:

```
SUBSYSTEM=="tty", ATTRS{manufacturer}=="FTDI", SYMLINK+="lms291",
ATTRS{serial}=="FTUUYD2A"
```

4.4.4 Cameras

ROS provides various nodes to work with cameras: drivers to retrieve images, calibration tools, and utilities to view the images in real time. All these node are compatible with the Prosilica cameras installed on the robot.

The package containing the driver is `prosilica_camera`, while the node is called `prosilica_node`. This node retrieves images from the camera and publishes them on a topic using a `sensor_msgs/Image` message, which contains a header and the uncompressed image. Every frame is coupled with calibration information, published by the node using a `sensor_msgs/CameraInfo` messages. This message contains a header, a distortion matrix, a 3x3 intrinsic camera matrix, a 3x3 rectification matrix and a 3x4 projection matrix, these last two are used only for stereo cameras. ROS automatically determines and fills all the values after the camera is calibrated using the tools provided with the `camera_calibration` package.

In our work, we used the cameras both in stereo and mono configuration, the calibration procedure is basically the same, the only difference is that for stereo cameras only one calibration is done for both cameras simultaneously, while for two mono cameras two separate calibrations are required.

The node used for calibration is `cameracalibrator.py`, which is part of the package `camera_calibration`. The procedure to calibrate a camera is the following. First, a large chessboard with known dimension is needed, the board must be rectangular (i.e., more columns than rows), so that the calibrator can recognize its orientation. Then, execute the calibrator node and provide as parameters the characteristics of the chessboard and the topic where the images of the camera are published. For example, this is the command to execute to calibrate a mono camera using an 8x9 chessboard with squares with sides of 5 cm:

```
rosrun camera_calibration cameracalibrator.py --size 8x9
--square 0.05 image:=/camera/image_raw camera:=/camera
```

Once the node is in execution, in order to achieve a good calibration, you have to move the chessboard around in the camera frame such that:

- The chessboard is on the camera's left, right, top and bottom parts of the field of view.
- The chessboard fills the whole field of view.
- The chessboard is tilted to the left, right, top and bottom.

When enough samples have been collected, press the CALIBRATE button, after the calibration is complete use the COMMIT button to send the calibration parameters to the camera for permanent storage. The driver node retrieves these parameters during execution to fill the calibration message.

Since there are two cameras installed on the robot, two instances of node are in execution. To avoid conflicts we need to distinguish the left camera from the right one. In order to do this we give an identifying name to each instance of the node and we use the remap function of ROS to rename the topics. A unique ID identifies each camera. Therefore the left camera is managed by a driver node called `left_driver`, which publish its messages on `/stereo/left/image_raw` and on `/stereo/left/camera_info`, while the right one interacts with a node called `right_driver`, which publish on `/stereo/right/image_raw` and on `/stereo/right/camera_info`.

Some common parameters are defined for both cameras, like the byte rate, which defines the amount of images per seconds, the use of auto exposure, the gain and the reference frame id, which is "stereo_frame".

4.5 Localization

In order to drive the robot autonomously it is necessary to know its position precisely. To do this, in our current configuration, we use measurements

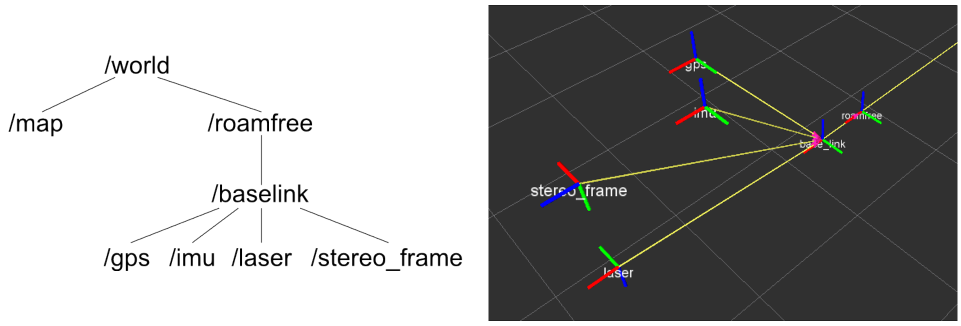


Figure 4.9: Hierarchy of the coordinate frames (left) and their position on the moving robot as seen in rviz (right)

coming from the GPS, the IMU, and the odometer, and estimate an absolute position using the ROAMFREE library. We used a node called `roamros`, a ROS wrapper for the functionalities of the library, which subscribes to all the sensor topics and provides a position using `tf`. This node is coupled with another one, called `fastPredictor`, developed within this thesis as an extension to the ROAMFREE library [14], which uses odometry and the absolute position estimated by ROAMFREE to predict local positions at a higher frequency.

4.5.1 Coordinate frames

We have to define the coordinate frames, these are necessary for various reasons: first, ROAMFREE needs to know the sensor displacement to calculate the estimate, moreover, some of the ROS tools used, `tf` to publish the pose and `rviz` for visualization, require a well-defined coordinate frames tree to work properly. Lastly, the robot planner (Section 4.5.3) uses a map, which has a different reference point with respect to the GPS, and the difference between these two origins is defined with a coordinate frame. The structure of the coordinate frames of the robot is the following (defined following the ROS convention and all of them are right-handed):

- `/world`: is the root of the tree and the global coordinate frame, it is defined with the z-axis pointing up. Its origin corresponds to the local reference point used in the ENU coordinates.
- `/map`: this is the coordinate frame of the map. Each time a map is used a static transform between its origin and the `/world` coordinate frame has to be published.

- `/roamfree`: this frame correspond to the robot position estimated by `roamros`, it is placed in the middle of the rear axle of the vehicle. It has the x-axis directed as the vehicle pointing ahead, the z-axis pointing up and the y-axis pointing to the left of the robot.
- `/base_link`: this frame is published by the predictor node, as a transformation from the `/roamfree` frame. This is the coordinate frame used as a reference by every node that needs to know the position of the robot. It has the same convention, with respect to the robot, of its parent frame.
- Sensor frames: each sensor has its own coordinate frame that represents its position on the robot, each one defined statically as a roto-translation from the base_link frame.
  - `/gps`: defines the displacement of GPS, x = 0.7 m, y = -0.46 m, z = 0.88, with no rotation.
  - `/imu`: defines the displacement of the IMU, x = 0.96 m, y = -0.03 m, z = 0.9 m, with no rotation.
  - `/laser`: defines the displacement of the laser scanner, x = 1.74 m, y = 0 m, z = 0.15 m, with a roll of 3.14 radians
  - `/stereo_frame`: defines the displacement of the stereo cameras, x = 1.77, y = 0 m, z = 0.91 m, with a yaw of -1.5707 radians and a roll of -1.5707 radians

4.5.2 Absolute positioning with ROAMFREE

Node `roamros` subscribes to sensor topics: `/PLC/ackermann_odometry`, the topic about the odometry of the vehicle, `/magnetic`, with information about the measurement of the magnetic field, `/enu`, gps readings in ENU coordinates, and `/imu/data`, for gyroscopes and accelorometers. Each sensor with its corresponding topic is defined in the configuration file of the node; in such file which sensor is the master is also specified, i.e., the odometer in our case. In the parameters, each sensor has its own static covariance matrix if not provided dynamically by the sensor itself; in our case, only the GPS has a dynamic matrix.

The sensors listed in the file are not the physical sensors of the vehicle, but the logical sensors used by the fusion engine. While in most cases the correspondence is one to one, i.e., odometer, GPS, and magnetometer, for the IMU it is necessary to define two different logical sensors: a gyroscope and an accelerometer. The odometer, which is the master, is used to build

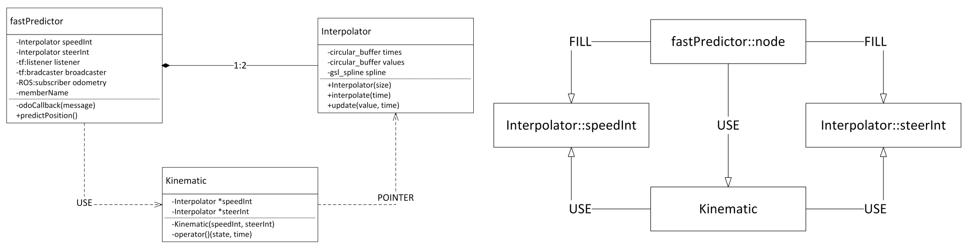


Figure 4.10: The class diagram (left) of the `fastPredictor` node, and the relationship between the instances of the classes (right)

the hyper-graph, as described in Chapter 2, since it provides enough information to predict the next robot pose. The covariance matrix is used to determinate the reliability of each sensor, and therefore their weight in calculating the estimated position.

The configuration file contains also the parameters used to configure the error models for each sensor. For example, hard and soft iron distortion that affect the magnetometer readings, or the constants used to convert from the raw odometry values to the actual steer and speed measurements.

Lastly, this file contains the parameters about the frequency of the estimated position and the size of the time window used. The time window size is the maximum number of measurements that are considered to determinate the position, and it is defined in seconds. Defining this parameter is crucial, because a larger window means that more measurements are used to determinate the position, therefore attaining more accuracy, but at the same time this increases the computation time reducing the frequency of the node. During experiments and simulations, we tried various values for these parameters, in the end the more suitable were:

- Window size: 5 seconds
- Frequency: 10 Hz

When started, `roamros` waits for the availability of the sensor displacements and retrieves the initial pose, which is published by another node on the topic `/initialpose`. This node is called `initialPose-roamros.py`, it retrieve the position using the GPS and the orientation using the magnetometer, when these two values are available, it composes them in a initial pose. Starting form this pose and after collecting enough sensor measurements, `roamros` estimates the current robot position, and broadcasts it periodically using `tf`.

4.5.3 Predicted position

In order to drive the robot autonomously it is necessary to provide the current position with high frequency and the lowest possible delay. It is possible to increase the frequency of `roamros` at the cost of a lower accuracy and increased computational load, but there will be always a delay caused by the time required to compute the position. Therefore, we developed a node, i.e., `fastPredictor`, that predicts a relative position, starting from the absolute position given by `roamros` and integrating the odometry of the vehicle.

The design of `fastPredictor` is based on the fact that position calculated from odometry is reliable if the distance travelled is short, but it accumulates errors over time. The node subscribes to the topic `/PLC/odometry` to receive information about the speed and the steer of the vehicle and implements a listener to receive the position broadcast by `roamros`. The node is characterized by few parameters: first, it is necessary to define the coordinate frame of the absolute position, in our case it is `/roamfree`, and at which frequency this position is broadcast, 10 Hz as stated before, lastly, the frequency of the odometry, 20 Hz in our robot. These two values are used to determinate the number of odometry measurements to store, since in the worst case you only need the last $f_o/f_r + 1$ measurements to calculate the relative position until the next global update is generated.

Each time a new odometry message is received the node extracts the steer and speed values and stores them in in two instance of an object called `Interpolator`, each measurement is coupled with its timestamp. Inside each `Interpolator`, values are stored in two circular buffer (one for the measurements and one for the timestamps), which size is determined by the frequencies defined in the parameters. Therefore, the node keeps in two separate containers the last $f_o/f_r + 1$ values of speed and steer.

The main purpose of these objects is to convert a discrete measurement, in our case speed and steer, into a continuous one. In order to do this, it necessary to interpolate, hence the name of the object, and because we want to do a prediction some extrapolation is required, too. To achieve a smooth interpolation we used splines, which are continuous smooth functions that are piecewise-defined by polynomial functions, and to integrate them in our software we used the GNU scientific library, which provides functions to create and use splines. For the extrapolation we used a linear extrapolation based on the two newest values, this approximation is sufficient as the extension in the future is in the order of milliseconds.

The continuous function defined by `Interpolator` is used by `Kinematic`, another object that defines the differential equations describing the kinematic of the robot. This structure permits to generalize the use of `fast-Predictor`, because it is possible to adapt the node by modifying the differential equations defined inside `Kinematic`. In the case of our robot, which is a four-wheeled vehicle with an Ackermann steer, the differential equations describing its kinematic are:

$$\begin{cases} \dot{x} & = & v(t) \\ \dot{\theta}_z & = & v(t)/L\, tan(\phi(t)) \end{cases} \tag{4.10}$$

Where $v(t)$ is the speed, $\phi(t)$ is the steering angle, both derived from the odometry, and L is the wheelbase of the vehicle. Integrating these equation gives the position of the robot in its reference frame.

In order to integrate these equations we used odeint [3], a C++ library for numerically solving ordinary differential equations. This is done directly in the main loop of the node. Each time the main loop of the node is executed a listener retrieves the timestamp of the newest absolute position broadcast by `roamros`, this is the starting time of integration, while the ending time is the current time plus a *dt*. Integrating a few milliseconds in the future gives a prediction of the future position of the robot, and grants the lowest possible delay. The result of the integration is the relative position of the robot, and it is broadcast as a transformation from the coordinate frame `/roamfree` to `/base_link` using `tf`. In order to maintain the coordinate frames tree consistent, when it is impossible to integrate because not enough odometry or no absolute position is available the node broadcasts the identity transformation between `/roamfree` and `/base_link`.

4.6 Trajectory generation

Integral part of any autonomous driving system are methods to define a trajectory to follow. In our architecture, we have different nodes that provide a path and all of them have a common interface. Trajectories are published on a topic called `/path`, using a custom message: `quadrivio_msgs/PathWith-Velocity`. The message contains a `nav_msg/Path`, which has a standard ROS header and a list of positions, and an array of speeds. The size of the array is the same of the number of positions. Another topic called `/visualize_path` is used to view the trajectory on `rviz`, on this topic only the positions are published using a `nav_msg/Path`. Currently there are three different methods to create a trajectory: drive the vehicle manually to record its path then publish it, generate a specific trajectory from a list of instruc-

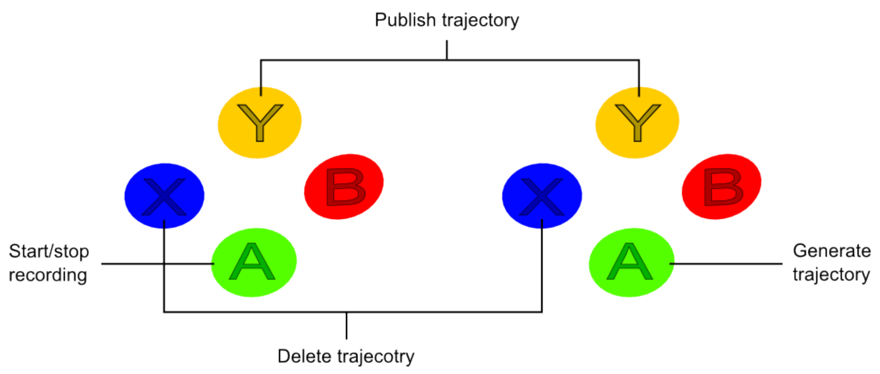


Figure 4.11: Button used to control the behaviour of the `trajectoryRecorder` node (left) and the `trajectoryGenerator` node (right)

tion or a planner that creates a plan given a goal, a map, and the starting robot position.

4.6.1 Recorded trajectory

The node that has the task to record and publish a trajectory is called `trajectoryRecord`. The functionalities of the node are accessed using the controller and are represented by three internal states. In order to receive the commands from the joypad, the node subscribes to the `/joy` topic, where a message containing the current status of the buttons of the controller is published by the joypad drive node. When a message is received the node reads the content and change the internal state accordingly. In the WAIT state, the node does nothing, pressing the A button starts the recording and change the state to RECORD. In this state, a listener retrieves the broadcasted robot position, which is used to create the path. In order to avoid duplicates and an accumulation of points when the robot stands still or when is moving slowly, a new position becomes part of the recorded trajectory only if the distance from the last one is greater than a specified value. A constant value fills the array of speeds, this allows driving the robot slowly on a well-defined path, record it, and then replay it with different speeds. Pressing again the A button while in the RECORD state ends the recording and change the state to TRAJECTORY. In this state the path is ready to be sent to the follower, by pressing the Y button a message is published on the `/path` topic, with no changes to the state, therefore pressing Y again sends again the message. Pressing the X button deletes the path and returns the node to the WAIT state.

4.6.2 Generated trajectory

To generate a specific path we developed a node called `trajectoryGenerator`, this node reads a list of command from a file and creates a well-defined trajectory, which is built using a list of poses, containing a specific position and orientation. We developed this node because while the recorded path was useful for the first experiments of autonomous navigation, the fact that is based on the estimated position made it unsuited for more advanced ones. In this case, instead, only the starting position of the robot is required to create a path with a specific shape, i.e., a circle, an oval, a sinusoid, therefore it is possible to predict the path of the robot and, consequentially, verify it.

The file containing the commands have a specific structure: each line is a different command, a line starting with a hash is a comment and each command contains three numbers separated by a comma. The three numbers in each line are, in order: a translation, in meters, a rotation, in radians, and a number of repetitions. Accordingly, the commands to create a straight line of 5 meters with a point every 10 centimetres followed by a 90°curve six meters long with a point every 30 centimetres are:

```
#straight line 5 meters
0.1,0,50
#left turn
0.3,0.07853,20
```

Combining rotations and translations it is possible to create any kind of trajectory, moreover a specific command to create sinusoid-like path is available.

In a way similar to the recording node, `trajectoryGenerator` is controller using the joypad. Therefore, this node, in the same of the previous one, subscribes to the `/joy` topic and use the content of the message published to change the its internal state. There are only two internal states: WAIT and TRAJECTORY. In the WAIT state pressing the A button generates the path and changes the state to TRAJECTORY. A listener retrieves the current position of the robot, one meter ahead from that position the trajectory is created by reading the commands in the file. This slight advancement makes the path more manageable by the follower. A value specified in the parameters fills the array of speeds. In the TRAJECTORY state the generated path is published on the topic `/path` by pressing the Y button, further pressions send more messages. In order to go back to the WAIT state and delete the trajectory it is necessary to press the X button. The node retrieves the speed and the commands again at each path generation, therefore it is possible to change them during execution without restarting.

4.6.3 Planner

The two methods to define a trajectory described above are useful when conducting laboratory experiments, but are ill-suited for real world applications. Therefore, a global planner developed by another thesis at Politecnico di Milano [12] is an integral part of the software architecture of the robot. It is an extension of a ROS compatible library, SBPL (Search Based Planning Library), inserting in it the possibility to have more vehicle features than those considered in its current version. The result is a planner usable with an ATV vehicle, which reduces the risk of overturn, and it can take into account many sources of information during planning. In order to use the planner in our architecture, a ROS node has been developed, called `trajectoryPlanner`. The node use AD* as a default planning algorithm and have the capability to publish a suboptimal solution. This path is improved over time and is published again every iteration of the main loop of the node. It is possible to change the starting point, the goal and the map between two iteration of the planning algorithm, therefore the suboptimal trajectory is improvable even if the robot is moving, i.e., it changed its starting position. The node is configurable with various parameters:

- the file of motion primitives to use, which contains a collection of the possible minimal movements of the robot
- an initial suboptimality value
- the width of the vehicle footprint
- the length of the vehicle footprint
- the type of the environment to use among the one with three state variables (x, y, θ), the one with four state variables (x, y, θ, v) and the two with five state variables (x, y, θ, v, δ) or (x, y, θ, v, ω)
- the obstacle threshold (from which cost value a cell has to be considered as an obstacle)
- the cell size (i.e., the resolution of the map)
- the number of orientations used to initialize the environment
- the maximum time allowed to find a feasible path
- the flag indicating if the node must continue search until a solution is found (if exists) ignoring the maximum time allowed to find a feasible path

- the search direction flag (true is forward, false is backward)
- some parameters needed if the environment used is the one with three variables
  - the nominal velocity
  - the time to turn in place
- some parameters if the environment used takes into account (x, y, θ, v [, . . .])
  - the number of linear velocities admitted
  - the linear velocities admitted
  - the number of steering angles if the environment take into account the steering angles
  - the number of angular velocities admitted if the environment take into account angular velocities
  - the angular velocities admitted if the environment take into account angular velocities

The node retrieves the map from a topic called `/map`, which is published by a modified version of the ROS map server, called `map_server_decimal_cost`. This version admits for each cell decimal costs, while the version available on the ROS repository only admits integer costs. A listener retrieves the starting position of the robot broadcast by the predictor. Since the planner needs to know the current speed of the robot, the node subscribes to `/PLC/odometry`, to retrieve the odometry message. The goal is a position published on the topic `/goal`, currently we use `rviz` to select a goal directly on the map.

When the node receives the map, the starting position and the goal, it begins to plan. As soon as a trajectory is available, the node publishes it and the follower starts following it. The node runs at a frequency of 15 Hz, and at each iteration of the main loop the trajectory is improved, each time, if necessary, using the updated starting position. The node publishes the improved path at the end of each iteration. The improvement continues until the optimal solution is found, or if the goal is modified, in this case a new plan starts.

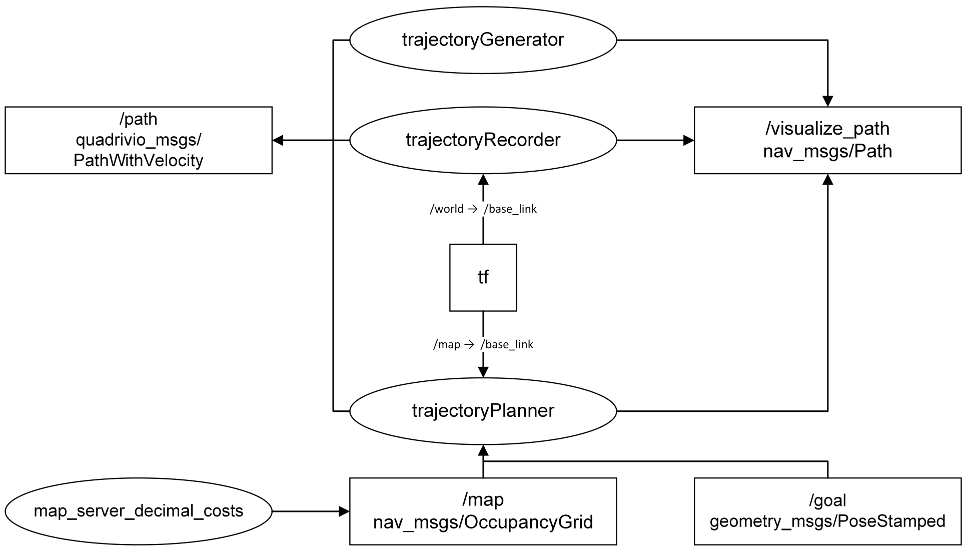


Figure 4.12: Nodes used to generate the trajectory and their topics

4.7 Path following system

The path follower used to control the robot while it is in the autonomous state is designed by another thesis at Politecnico di Milano [15] specifically for the ATV used as a base for our robot. In order to integrate it in our architecture we developed a ROS node that implements the functionalities of the follower. The node, called `TrajectoryControl`, subscribes to the topic `/path` to receive the trajectory to follow, and uses a listener to retrieve the current robot position broadcast by the predictor. The path received contains a collection of points, but the specific algorithm performing the trajectory control needs a continuous trajectory. Therefore, we used splines, implemented again using the GNU scientific library, to create a continuous function from the positions contained in the message. This new path is usable by the path follower, which, at every execution loop of the node, retrieves the current robot position, generate a setpoint compatible with the trajectory and publish it on the `/setpoint` topic with the value AUTO as a source. The node works only when the internal state machine is in the autonomous state, otherwise every path received is discarded. In any case, if the node sends setpoints when not in the autonomous state, the `PLCClient` ignores them. This ensures that the vehicle does not move when not in the autonomous state. The robot follows the trajectory until the distance to the goal is lower than a threshold, at that point the node stops the following and sends to the low-level control system setpoints that keep the vehicle braked.

Name	Value	Description
maxCurvature	0.625	maximum curvature that the vehicle can perform
lambda epsilon	0.001 0.7	Parameters of the ellipsoid, which represent the maximum distance between the followed reference frame and the robot
sogliaAlpha gamma H beta	0.001 1 3.0 2.2	Gains of the internal controller of the path following algorithm for speed and steer

Table 4.3: Paramteres used to configure the `trajectoryControl` node.

Other than the setpoints, the node publish two more topics: one contains the followed reference frame, viewable on `rviz`, it is useful to know if the robot is at the right point of the followed trajectory, while the other is a custom message that expose some of the internal values of the path following algorithm. In order to have a complete integration of the follower in our architecture we have turned its parameters into ROS parameters described in Table 4.3.

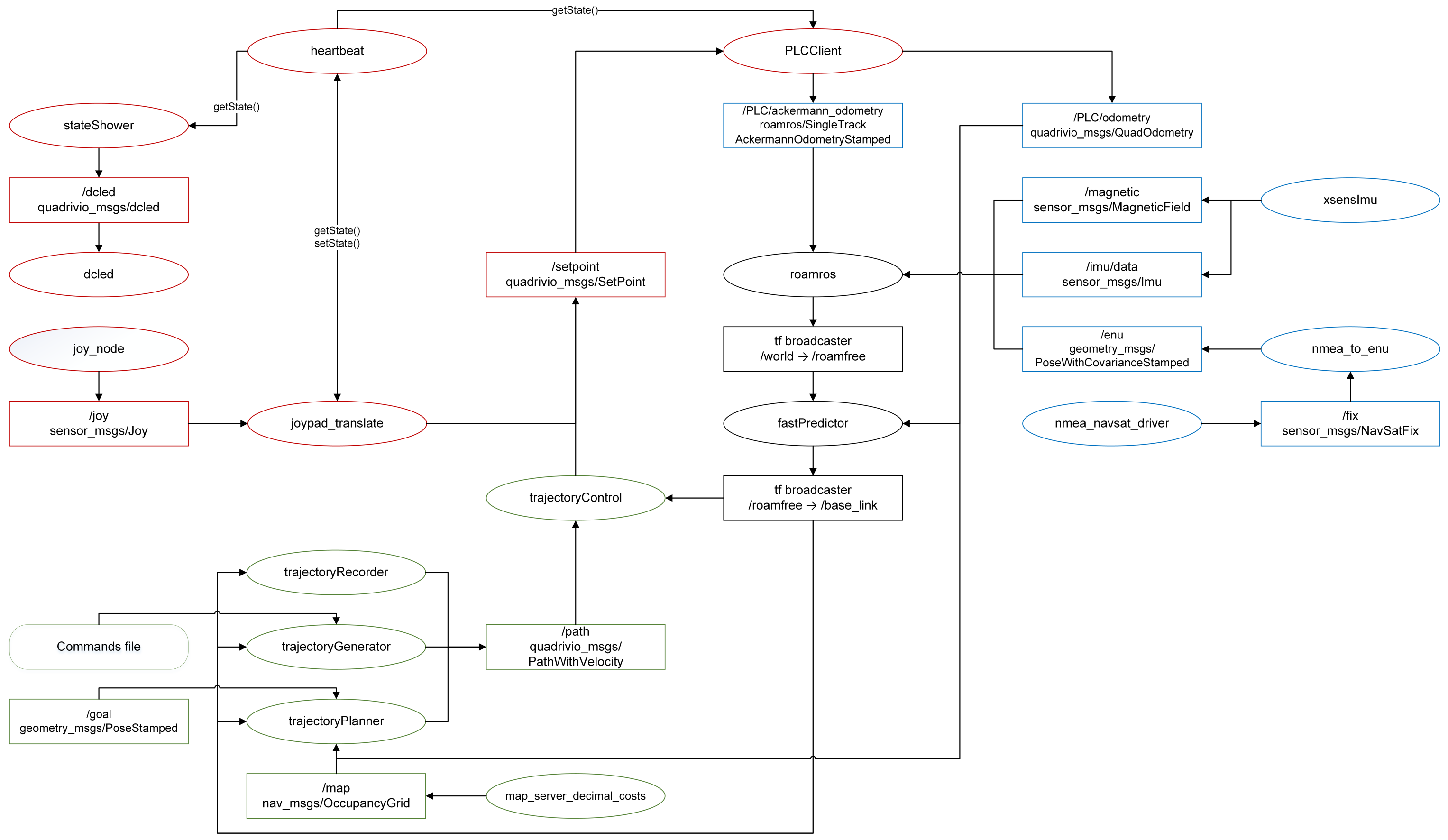


Figure 4.13: The overall architecture

Chapter 5

Robot simulation

In this chapter, we describe the simulation tools developed alongside the software modules. We used the simulation to improve and test the architecture before the deployment on the robot. We start with a general description of the simulated environment, followed by a detailed analysis of the robot model. After that, we introduce how the sensors are simulated and how we integrated them with the robot architecture.

5.1 Scene description

In V-Rep a scene is a simulated environment, it includes all the elements that compose the simulation. A scene contains a hierarchy of objects; these can be physical (e.g., the robot, obstacles or a terrain), or virtual (e.g., scripts, lights or cameras). Each scene has a main script, which contains the basic code that allows a simulation to run. While V-Rep allows you to customize models using child scripts, the main script is not supposed to be modified, although possible. Moreover every scene contains a series of cameras, not to be confused with vision sensors, that allow to view a real-time rendering of the simulated objects from different points of view while composing the scene and during the simulation.

For this work, we created two different scenes; one is a complete scene, with a rough terrain covered with a texture, 3D models of trees, to create obstacles and reference points used when computing the visual odometry. Moreover, in this scene, the robot is equipped with all its sensors, including vision sensors and laser. The other scene is simpler, it as a flat terrain, no texture, no obstacles and the robot lacks the laser and the vision sensors. The use of these two scenes allows us to simulate easily a large number of

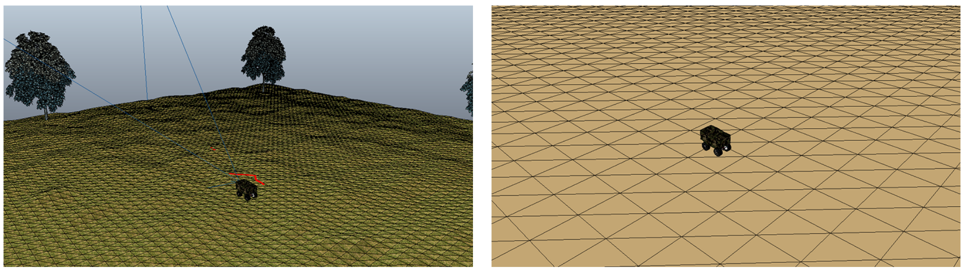


Figure 5.1: The two scenes used. On the left, `quadrivio_full.ttt`, on the right, `quadrivio_light.ttt`

different behaviors.

In both scenes, to simulate the ground, we use a heightfield shape, which represent a terrain as a regular grid, where only the heights change. This shape is optimized for dynamics collision response calculation, and V-Rep allows importing specific terrains using two types of sources:

- Images files: an image file (JPEG, PNG, TGA, BMP, TIFF or GIF file) where the various height values are taken from the red, green and blue color components of each pixel: height = (red+green+blue)/3.
- Comma-separated values files: the file should contain y rows where each has x values separated by commas.

After selecting the file to import it possible to specify the size of terrain and the real height corresponding to the maximum height value in the source file. In our scenes we use images to generates random terrains, for example a completely black image for a flat terrain or a grey-scale image for a rough terrain, and we use CSV files to recreate in the simulation terrains taken from real maps.

In early stages of development for our architecture, we designed and used the simpler scene, called `quadrivioLight.ttt`. Having a flat terrain, i.e., without any roughness, and lacking obstacles, it is well suited to test the model of the robot, its localization and the path following behavior. Since problems encountered during any of these tests can be isolated as a fault in the software and not caused by external elements. Moreover, the absence of the laser and vision sensors reduce the computational load, making the simulation running almost in real time, this permits to an operator to drive the simulated robot without difficulties.

Going on in the development of the software architecture, we had the need to test the robot in a more complex environment, therefore we designed the second scene, `quadrivioFull.ttt`. The rougher terrain is useful to test

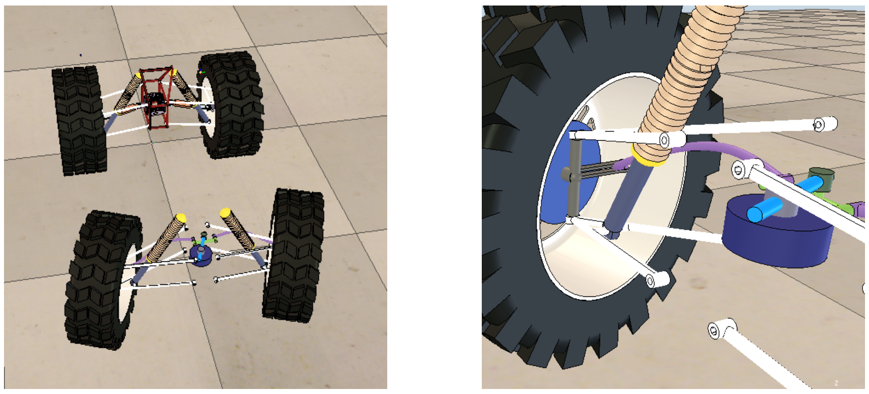


Figure 5.2: Steering system and suspensions of the vehicle model

the robustness of the path follower, the 3D models of trees are obstacles detectable by the laser and act as a reference point for the vision sensors. The simulation of this scene cannot be carried on in real time, therefore manual drive results difficult, but, thanks to the use of simulation time in ROS, there are no limitation when the robot is autonomous.

5.2 Robot model

To simplify the creation of our robot model, we used one of the built-in models that V-Rep offers as a base. We chose one that shares the Ackermann steering and the suspension geometry of our vehicle and we customized it to match the robot characteristics. In this section, we describe the characteristics of the model and the customization we have implemented.

5.2.1 Steering system and suspensions

As the structure of the suspension and the Ackermann steering system are the characteristics on which we based our choice of the starting model, we did few modifications to match the behavior of the robot.

The Ackermann steering system is untouched, except for the change of size to match the real vehicle dimensions. A rotational joint controls the position of the handlebar, its angle is the same as the steering angle of the model, and therefore no conversion is needed when applying the setpoints. The joint is part of a system connected to two rigid arms that control the position of the front wheels. As we are not interested in studying the dynamics of the steering motion control system, but only in simulating the overall vehicle dynamics, the simulator does not include an accurate model of the steering column.

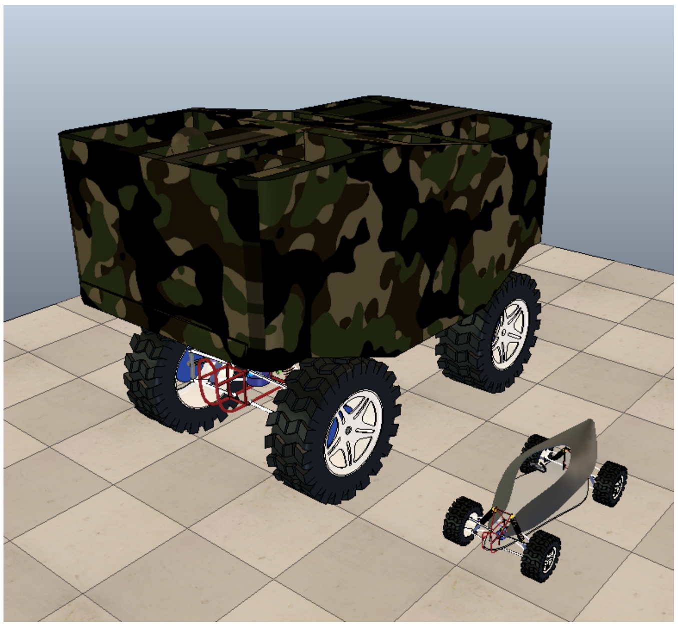


Figure 5.3: Comparison between the robot model (left) and the original model (right)

Four prismatic joints realize the suspensions, each one connected to the vehicle chassis and to the wheels by mean of rotational joints. In order to recreate the behavior of a real suspension, a PID controls the joint to achieve the damping effect of a spring. In order to tune the parameters of the PID, we started from the values used in the original model and modified them until a behavior comparable to the real vehicle was reached. In the end, the configuration of the joints is the following: maximum force of 400 N, the proportional control action is 100 and the derivative control action is 800.

5.2.2 Geometric characteristics

The most noticeable difference between the original model and our robot is the size. Despite the realism of the components, like the steer and the suspensions, the original model is comparable to a toy car in terms of dimensions. Moreover, the ratio between the wheelbase and the track is different from that of our robot.

In order to remove these differences we did some customization: first, we used a scaling function offered by V-Rep to match the size of the model with that of the robot, since the proportion are not the same, we scaled the model using the track of the vehicle as a reference. The choice of this dimension was made in order to maintain the structure of the transmission and the steering system. After that, we changed the size of the wheels, and gave

Vehicle model specifications.	
Wheelbase	1920 mm
Track	1250 mm
Weight (vehicle)	390 kg
Front wheel (WxH)	201 x 635 mm
Weight (front wheel)	7 Kg
Moment of inertia (front wheel)	0.3528 kgm^2
Rear wheel (WxH)	247 x 635 mm
Weight (rear wheel)	8 Kg
Moment of inertia (rear wheel)	0.4032 kgm^2

them a mass and a moment of inertia equal to the real ones. In addition, we reduced the wheelbase by moving back all the components on the front of the vehicle: the front wheels, the steering system and the front suspensions. Lastly, we moved up the bodywork to adapt the position of the center of mass to that of the vehicle, moreover we changed its mass to obtain a total value comparable of that of the robot. In order to match the robot visually as well as structurally, we replaced the bodywork of the original model with a mesh with the shape the robot one. Table lists the values used to size the model of the robot.

5.2.3 Step response

The next step in vehicle simulation is to ensure the real vehicle and the simulation model share the same dynamic and kinematic behavior. In particular, we required the step response for the handlebar loop and the speed loop on the real vehicle and in simulation to be similar. In order to achieve this we regulated the values of the PIDs that control the steer and the speed loops, until the simulated responses of these controlled systems were as close as possible to the experimental ones.

First, we recorded data of the handlebar step response and the speed step response of the real vehicle, then, using a ROS node to send the same input to the simulator, we recorded the response of the model. We used the comparison between these two results to tune the parameters of the simulated PID controllers.

Since the steering system is particularly similar to the vehicle one, it is possible to tune its step response using the joint built-in controller. The PID controls the position of the handlebar column by modulating its velocity. Using the values from the real vehicle as a reference, we defined the maximum

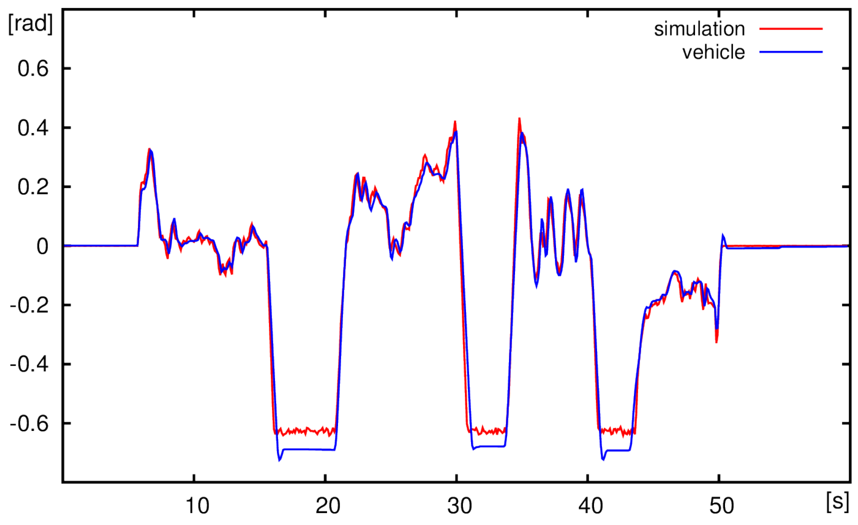


Figure 5.4: Model parameters.

speed of the joint as 70 °/s. Figure 5.4 shows a comparison between the handlebar response for the real vehicle (blue line) and the simulated one (red line). It is possible to see that the two behaviors substantially match, but the simulated steer cannot reach a value as high as the real one, because of geometric limitations in the original model. Nevertheless, we obtained a reasonable behavior in common operation ranges.

For the speed response, we could not use the built-it PID, because this controller minimizes the error on the position, which is incompatible with a continuously rotating joint like the one that act as a motor. Moreover, the motor joint has a behavior similar to that of an electric motor, while the vehicle has a fuel-powered engine with significantly different characteristics. Therefore, we developed a custom PID controller, which controls the torque applied to the motor joint minimizing the difference between the current rotating speed and the target speed. In particular, the controller has two different behaviors, one for the acceleration and one for the deceleration, in both cases only the proportional control action is used. While accelerating the K_p is 0.875, on the contrary while decelerating K_p is 0.7875. In the end, we obtain a good matching behavior in the acceleration phase, while there is a slight difference in deceleration, even with the specific proportional control action, due to the difficulties in modelling the engine braking when the throttle setpoint decreases suddenly. Figure 5.5 shows a comparison between the real vehicle and the simulator.

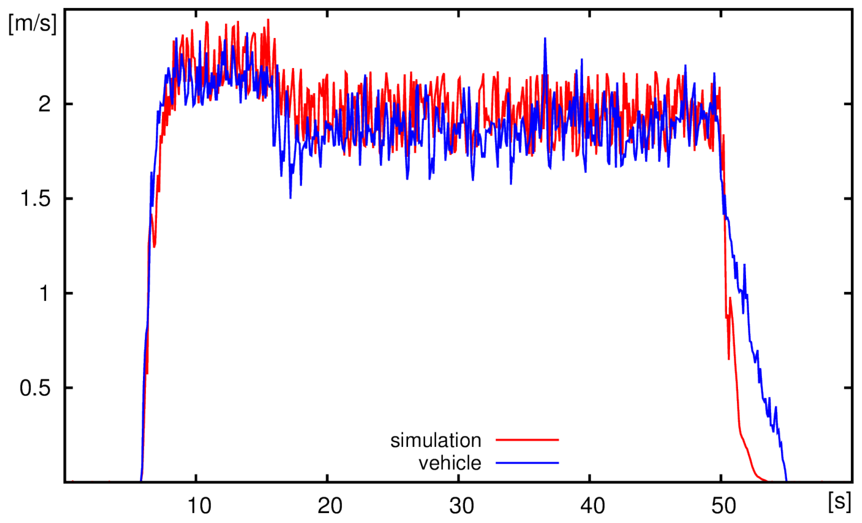


Figure 5.5: Plot of the vehicle linear speed step responses on the real vehicle and on the the simulated one.

5.3 Simulated sensors

In order to complete the model, we added a simulated version of each sensor installed on the robot. Most of them are already available in V-Rep, while others has been developed ad hoc. All the simulated sensors lacked ROS integration and they did not account for noise and faulty behavior, therefore we customized each of them to be more realistic and to match the ROS interface of the real sensor. In order to integrate these simulated sensors in our software architecture we designed their structure with a “two layer” approach. The first layer is implemented inside the simulator, it consists in the sensor itself and a script that prepares and publishes ROS messages. The second layer is outside the simulator and it consists in a ROS node that reads the messages published by the simulator and converts them into a format that matches the one produced by sensors on the real vehicle. This design aims at creating a decoupling between the simulator and the high-level perception and control architecture. As a result, from the point of view of the high-level, system there is no difference between the simulated sensors and the real ones.

In the following we provide a detailed description of each sensor, how it is implemented inside V-Rep and how we designed the ROS interface.

5.3.1 Global positioning system

V-Rep already has a built-in simulated GPS, which provides x/y/z-coordinates; while these are not the usual GPS coordinates (latitude, longitude and altitude), they are compatible with the East-North-Up (ENU) system used in our architecture, so no further conversion is needed. The script associated with the sensor introduces some uniform noise and streams the values on a ROS topic, called `/vrep/gps`, using a `geometry_msgs/Point32` message, therefore only the three coordinates are available.

The node associated with the sensor subscribes to that topic, retrieves the message coming from the simulator and converts it in a `geometry_msgs-/PoseWithCovarianceStamped`. In order to do the conversion it fills the header, using `ros::Time::now()` for the timestamp and setting "/gps" as the frame, moreover it adds a covariance matrix derived from real data. To obtain a more realistic behavior of the sensor, the node can simulate downtimes; using parameters it is possible to define their frequency, ranging from a perfect sensor to an always-off sensor. The node has an internal timer, when it fires a roll is performed, if its values exceeds the threshold defined via parameters, the sensor turn off for a variable time interval, going from 0.75 seconds to 1.25 seconds. As for the real sensor the node publishes the message on the `/enu` topic.

5.3.2 Inertial measurement unit

The simulator provides out of the box models for accelerometers and gyroscopes. A mass coupled with a force sensor implements the accelerometers; at each simulation step, the script measures the force applied to the mass and derives three accelerations. The gyroscopes consist in a dummy object, at each simulation step, the script compares its position with the one at the previous step, and the change of orientation in time determinates the angular velocity on the three axes. Inside V-Rep, the IMU is implemented as two different sensors, i.e., an accelerometer and a gyroscope, since it is necessary to have synchronized values, the measurements are collected by a third script though an internal communication system. This script receives the values from the accelerometers and the gyroscopes, combines them in a `geometry_msgs/TwistStamped` and streams them on the `/vrep/imuData` topic. Normally, this message is used to contain a linear velocity and an angular velocity, in this specific case it contains a linear velocity and an angular acceleration. The values in the message are the following:

- `twist.linear` contains the values of the linear acceleration.

- `twist.angular` contains the values of the angular speed.

The corresponding node subscribes to two topics: `/vrep/imuData` and `/vrep/realPose`. The first one to retrieve the measurements, the other one to have some orientation data to fill the `sensor_msgs/Imu` message (the localization module of the robot does not use that information). The node composes a message published on the `/imu/data` topic.

5.3.3 Magnetometer

To implement the Earth magnetic field sensor we extract the current vehicle model orientation with respect to the global fixed reference frame and we stream it to ROS using a `geometry_msgs/PoseStamped` message published on the topic `/vrep/simuMag`. The node associated with this sensor subscribes to the topic and retrieves the orientation. In order to create a `sensor_msgs/MagneticField` message, we have to derive a magnetic field value from the orientation of the model. Since on the real vehicle hard and soft iron distortion affect the magnetometer readings, according to the sensor model presented in [48], we use a set of parameters to modify the simulated magnetic field and match the behavior of the real sensor. The values of these parameters have been calibrated using the sensor self-calibration capabilities of the ROAMFREE sensor fusion framework. Moreover, it is possible to set the current Earth magnetic field via parameters to have a complete and realistic simulation of the magnetometer. The complete formula used to derive the measurement of the magnetic field is the following:

$$M = R\,q^{-1}\,h + S \tag{5.1}$$

Where q is the quaternion that represents the orientation of the robot, h is the Earth magnetic field, R and S are the distortion parameters.

5.3.4 Odometer

The position of the joint that controls the handlebar provides directly the steering angle. For the current speed of the vehicle, we measure the orientation of the wheels at each simulation step its change in time is used to calculate the angular velocity, and, consequently, the linear velocity of the vehicle.

As described in Chapter 4, the low-level control systems sends the odometry in non-standard units of measurements. In order to mimic this behavior we have to convert the values derived from the simulation using `kspeed`, `ksteer` and `psisteer`. The formulae used are:

$$speed_{odo} = speed_{m/s}/K_{speed} \tag{5.2}$$

$$steer_{odo} = \frac{steer_{rad} - \psi_{steer}}{K_{steer}}. \tag{5.3}$$

After that we build a message with the same format of the one used by the real low-level control system, and transmit it using a TCP socket to the `PLCClient` node. In order to implement the communication between the simulator and the architecture, we used a separate threaded script. This is necessary because transmissions using sockets are potentially blocking, disrupting the flow of the simulation. The threaded script that manages the socket connection receives the odometry from the script associated with the robot model through a tube, which is a specific communication channel used inside V-Rep to transmit information between two different scripts. Another tube is used to transmit setpoints in the opposite direction. At each simulation step the main script produce an odometry message, which is sent to the socket script, and transmitted to the high-level architecture as soon as possible.

5.3.5 Laser and camera

V-Rep already provides a highly customizable vision sensor that can match the characteristics of the cameras mounted on the robot. Moreover, it has a laser scanner with the same specification of the LMS291. Since both sensors already streams their measurements in a format compatible with the real ones, there is no need of external ROS nodes to interface them with the high-level architecture. In order to make the vision sensor behaves as the Prosilica cameras, we changed the internal resolution and increased the distance of the far clipping plane. For the laser, we set the scanning angle at 180°and the resolution at 0.5°.

The only difference that remains between the real sensors and the simulated ones is the frequency at which measurements are transmitted. While the camera operates at 60 fps and the laser at 50 Hz, the simulation only works at 20 Hz and this is the highest frequency at which the simulated sensors can operate. This issue remains unresolved but does not comprise the validity of the simulation, since the cameras and the laser scanner are secondary sensors not currently used for localization. Moreover, in the current configuration, the cameras operates at 20 fps on the real vehicle, too.

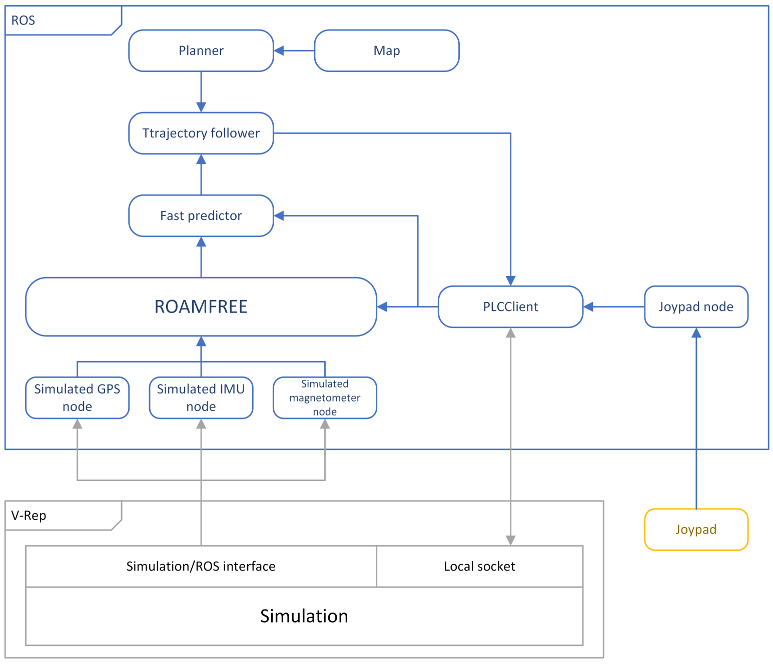


Figure 5.6: How the main modules of the architecture interacts with the simulation

5.4 Integration with the architecture

As described in Chapter 4, the design of the software architecture is highly modular, therefore we can seamlessly replace the robot with the simulation. In order to do this we remove all the ROS nodes communicating with the real sensors and replace them with ROS interfaces described in this chapter. Using a hierarchical structure of the launch files, we can do this substitution by replacing the sensors launch file with the simulated one. When every sensor is replaced by its simulated counterpart, nothing changes from the point of view of the perception and control modules, since the communication is done on the same ROS topics and using the same messages. Moreover, the PLCClient node interacts with a simulated low-level control system through a local socket connection; it receives odometry messages from the simulation and sends setpoints. In order to connect the node to the simulator, it is necessary to modify the connection parameters changing the IP from the address of the low-level control system to the IP of the machine running the simulation, usually 127.0.0.1.

One important issue to address when coupling a simulator with a control architecture is to make sure that they share a global time reference. In order to obtain this we exploit a ROS functionality that allows to use a simulated clock instead of the system clock. We developed a node that, using a ROS service provided by V-Rep, publishes the simulation time on the topic `/clock`. To make the nodes use the time published on the topic, the `use_sim_time` parameter must be set to true before the node is initialized. The use of the simulated clock become essential when running challenging simulations which involve complex terrain or vision sensors and cannot be carried out in real-time.

Chapter 6

Experimental results

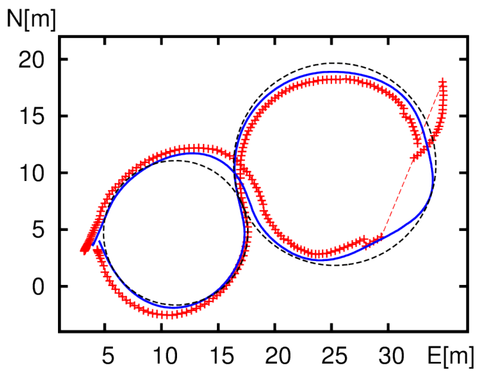
In this chapter, we provide the results obtained from experiments done both with the robot and the simulation using the proposed architecture. First, we report a combination of localization and autonomous drive on the vehicle with and without the predictor. Then, the same behavior in the simulation is presented, followed by more complex trajectories and rougher terrains to show the capabilities of the simulator.

6.1 Localization and autonomous drive

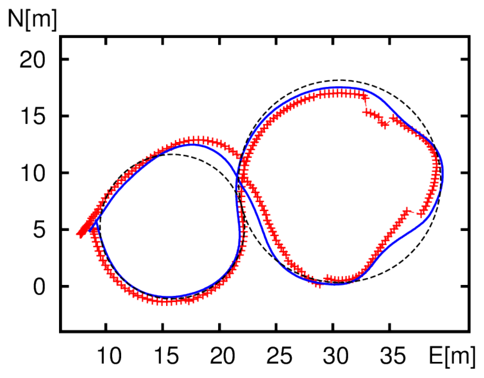
Given the characteristics of the vehicle, it is difficult to test the localization alone. In order to drive the robot wide outdoor spaces are required, therefore no external tracking system is available to derive a ground truth, neither it is possible to create a structured environment. In our experiment, we use well-defined trajectories to test localization and autonomous drive simultaneously. The more the behavior is close to the planned trajectory the better it is. The localization is the first functionality to be validated in order to validate the entire architecture, since various modules rely on the estimated position. Therefore our first experiments were aimed at determining the reliability of the localization module.

The first set of examples precedes the development and use of the predictor, and shows the first experiments of autonomous drive of the robot. We employed an eight-shaped trajectory originating 1 meter ahead with respect to the current position of the robot, the robot follows this path starting from the smaller circle and doing a left turn first. The bigger circle has a diameter of 18 meters, while the smaller one has a diameter of 12.5 meters.

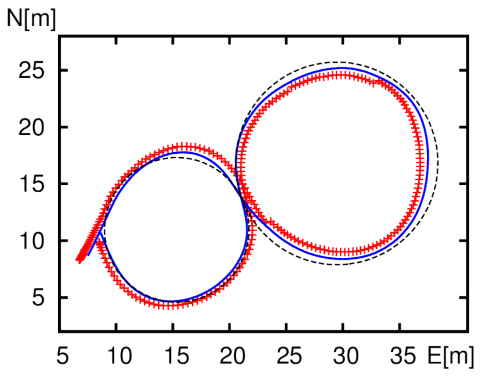
Figure 6.1 shows six runs on the real robot; in it we have plotted the



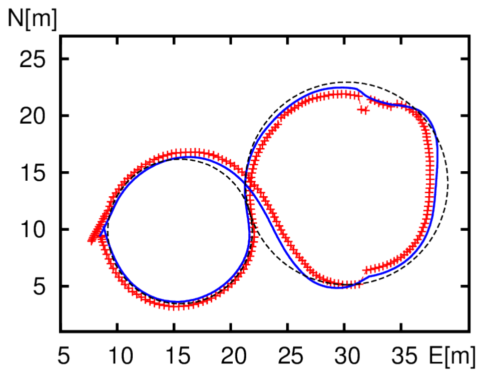
(a) 2 *m/s*



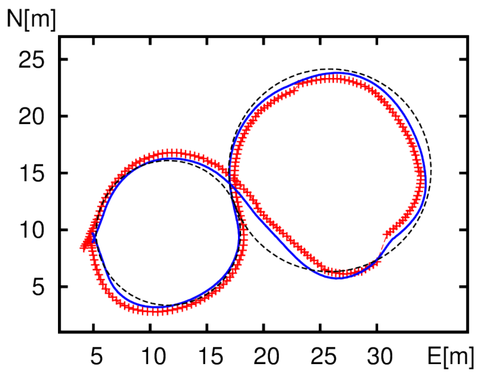
(b) 2 *m/s*



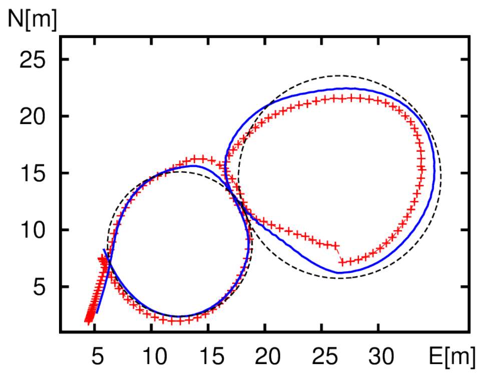
(c) 2 *m/s*



(d) 2 *m/s*



(e) 3 *m/s*

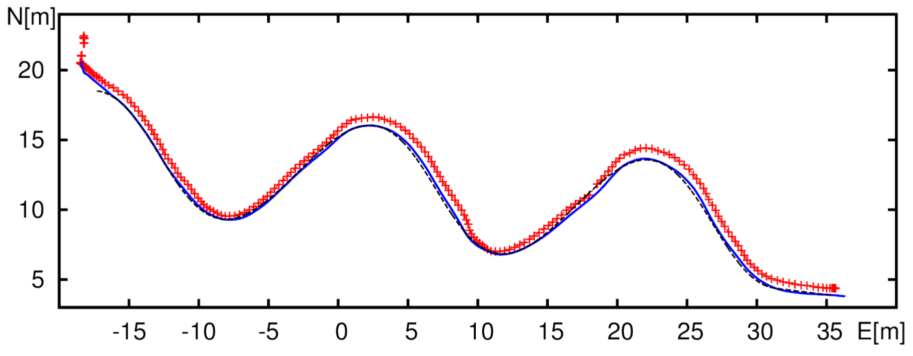


(f) 4 *m/s*

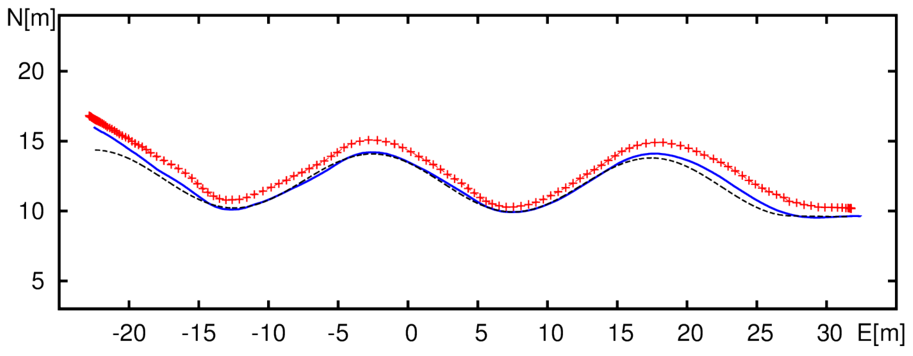
Figure 6.1: Online trajectory following results, on the real platform, without the predictor. Reference path for the trajectory follower (black dashed line), the ROAMFREE position output (blue line), and the GPS readings (red crosses).

reference path, with a black dashed line, the robot position estimated by ROAMFREE, with a blue solid line, and the raw GPS readings, with red crosses. These graphs show that the GPS, the closest information we have to a ground truth, is not always reliable. Figure 6.1a, in particular, but also the other ones, shows the multipath phenomenon where the position given by the GPS may be shifted by several meters. Despite these inaccuracies, it is possible to see that ROAMFREE is able to estimate a reasonable position, particularly visible when the GPS is correct, and to account for compromised sensor readings. Regarding the trajectory following, the graphs show that the robot follows the path with reasonable accuracy, but tends to oversteer or understeer, even more when the localization is partially compromised by a faulty GPS behavior. While the autonomous driving of the vehicle requires some improvements, these first experiments show that the overall architecture is solid, since the robot was able to generate and follow with sufficient accuracy a predefined path. In an effort to increase the accuracy of the path following, we designed and developed the predictor. This addition gives us the possibility to reduce the frequency of the pose estimated by ROAMFREE, and therefore increase its accuracy. Moreover, the predictor reduces delays caused by computation of the position and those introduced by ROS.

Figure 6.2 and Figure 6.3 show the results obtained in the experiments when using the predictor. As before, we plotted the generated path, the estimated position and the GPS readings. In these tests we use three different trajectories, the first two are sinusoids, one with an amplitude of 4 meters, while the other only 2 meters, both with a distance between two crests of 15 meters. The third one is a rectangular-shaped path with smooth turns at the corners. It contains four left turns, and four straight road, two long ones of 15 meters and two short ones of 2 meters. The robot repeats this trajectory two times to complete the path. As it is possible to see from the graphs in Figure 6.3 the robot follow the trajectory more closely, some oversteer problems persists, because the vehicle has an higher speed than expected when approaching the turn, therefore these problems are not associated with the localization or the path following algorithm. Moreover, multi-path phenomena have less effect on the position estimate, because reducing the frequency and increasing the time window in `roamros` grants more robustness.

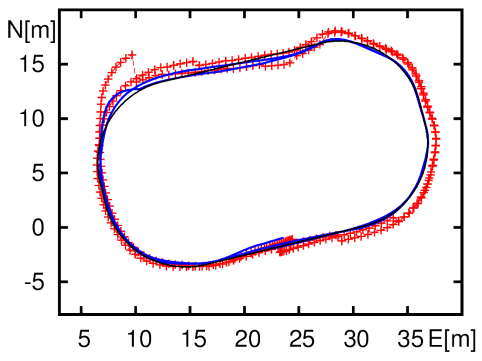


(a) 2 *m/s*

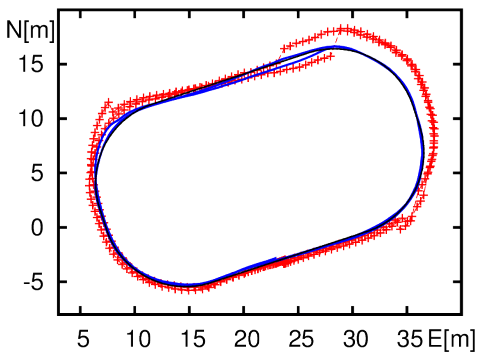


(b) 3 *m/s*

Figure 6.2: Online trajectory following results with the predictor on a sinusoidal path. Reference path for the trajectory follower (black dashed line), the predictor position output (blue line), and the GPS readings (red crosses).



(a) 3 *m/s*



(b) 3 *m/s*

Figure 6.3: Online trajectory following results with the predictor on a rectangular-shaped path. Reference path for the trajectory follower (black dashed line), the predictor position output (blue line), and the GPS readings (red crosses).

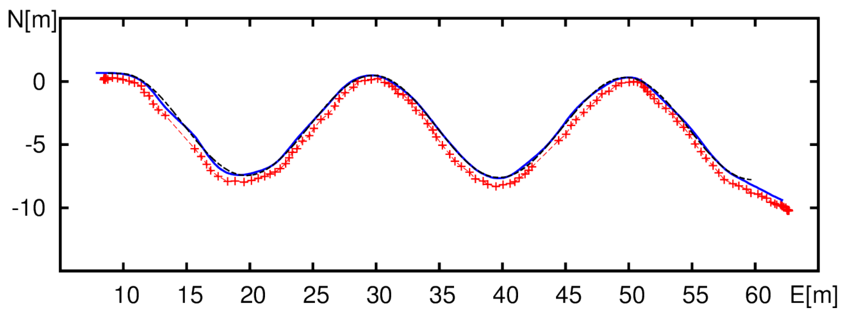
6.2 Simulation

In order to validate the simulated environment created with V-Rep, we replicated the same experiments done with the real vehicle in the simulation, using the same configuration parameters and the same sinusoidal and rectangular-shaped trajectory. The final aim is to verify that the behavior of the simulated robot is close enough to the real one in order to use the simulation to perform complex experiments (i.e., high-speed trajectory following, rough terrain navigation, etc.) before doing them with the real robot.

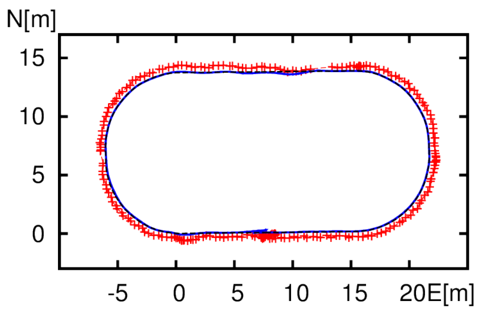
Figure 6.4 shows a series of experiments done with the simulator without the use of the predictor. With respect to the real vehicle we obtain a different behavior, the model closely follows the trajectory, but tends to oscillate around it. In some cases, for example Figure 6.4d and Figure 6.4e, these oscillations diverge, making it impossible for the simulated robot to complete the path. This significant difference between the behavior of the real vehicle and the simulation is not due structural differences between the model and the robot. It is caused by the delays introduced by ROS and the computation of the position, which cause an expected behavior in path following algorithm when the time is discrete, like in the simulator.

Figure 6.5 shows how the simulation behaves when the predictor is active. The robot model follow the trajectory closely, with no oscillations, and can complete all the paths. These graphs also show the simulated downtimes in the GPS and how the localization is robust to the temporary absence of one of the sensors. Despite the noise and the faulty behaviors added to the simulated sensors, these remain more reliable than the real ones. Therefore, the results obtained in the simulation are more precise than those obtained with the real vehicle are. Nevertheless, the two behaviors are sufficiently similar to use the simulated environment to perform experiments in condition difficult to obtain in reality, e.g., wide rough terrains, or dangerous for the robot, e.g., high-speed trajectory following. Figure 6.6 show the results of some of these experiments.

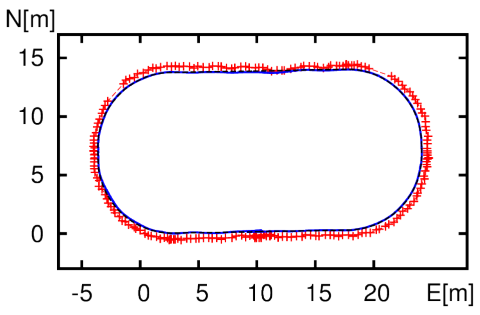
Figures from 6.6a to 6.6c show what we obtained from a high-speed trajectory following experiment done in the simulation. While not significantly different from the previous ones, these are useful to test the limits of the vehicle, which has a high rollover risk, in a simulated environment before doing them with the real robot. Moreover they show the robustness of the follower when high speed manoeuvres are required. Figure N.d shows the results obtained in trajectory tracking performed on a rough terrain with a path more complex than those seen so far, i.e., the eight-shaped trajectory



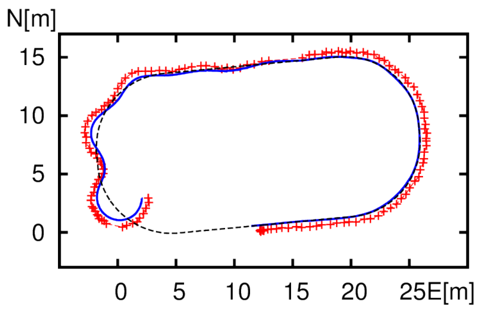
(a) 4 *m/s*



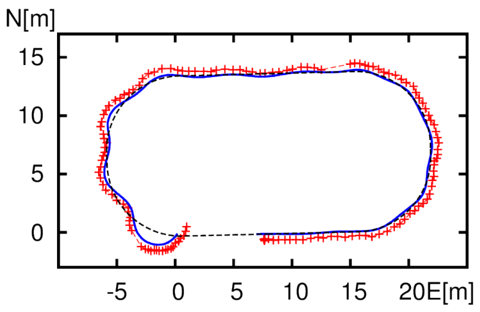
(b) 2 *m/s*



(c) 2 *m/s*

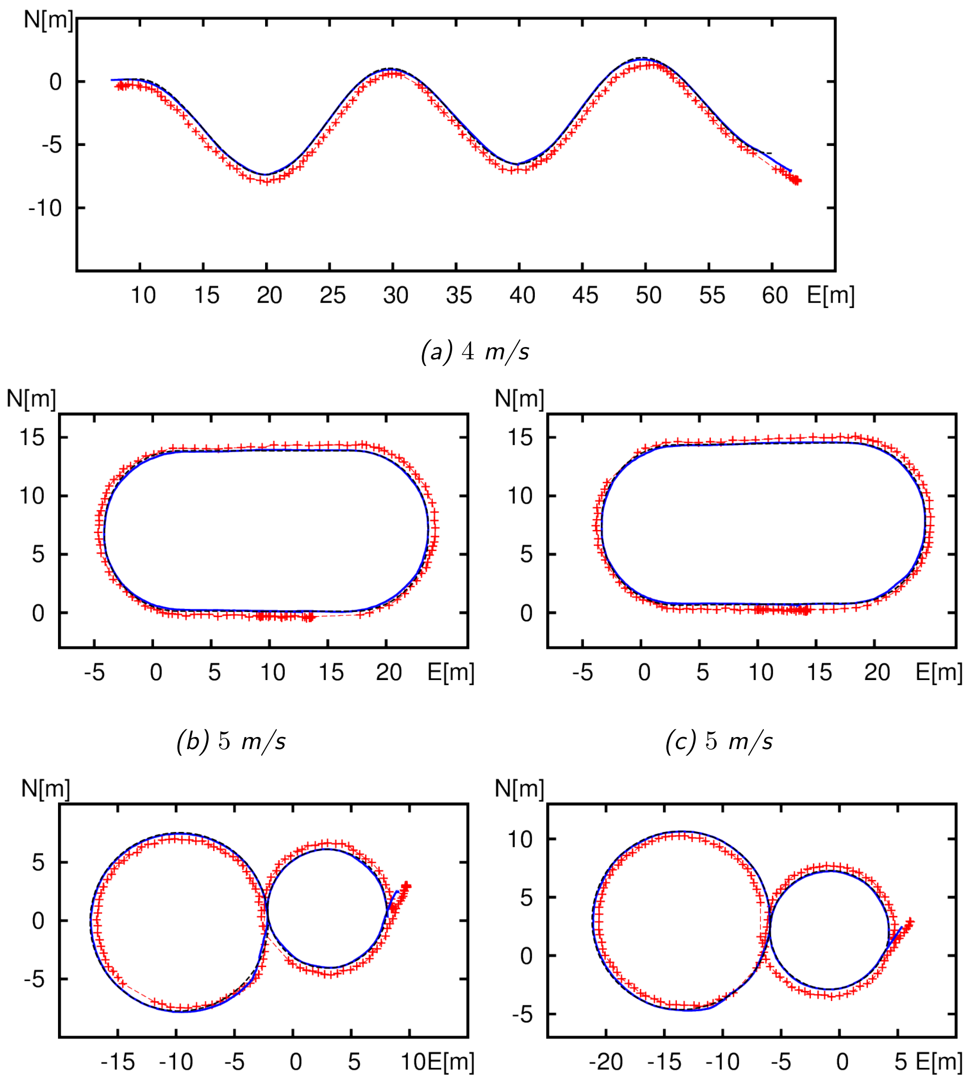


(d) 3 *m/s*



(e) 3 *m/s*

Figure 6.4: Online trajectory following results without the predictor. Reference path for the trajectory follower (black dashed line), the ROAMFREE position output (blue line), and the GPS readings (red crosses).



(a) 4 *m/s*

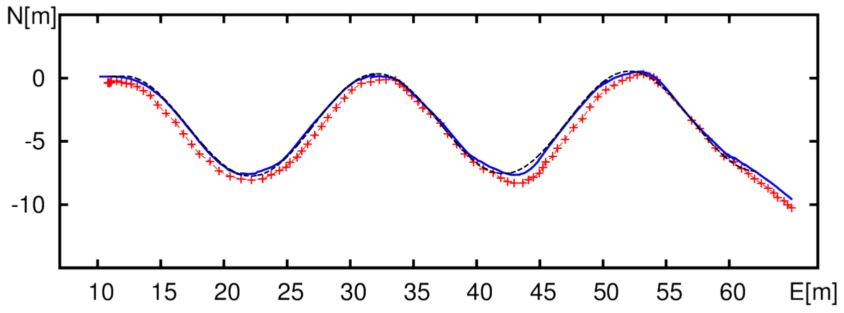
(b) 5 *m/s*

(c) 5 *m/s*

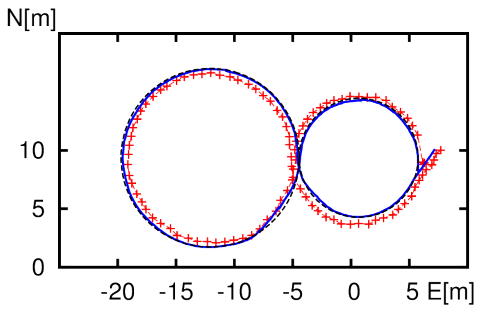
(d) 4 *m/s*

(e) 4 *m/s*

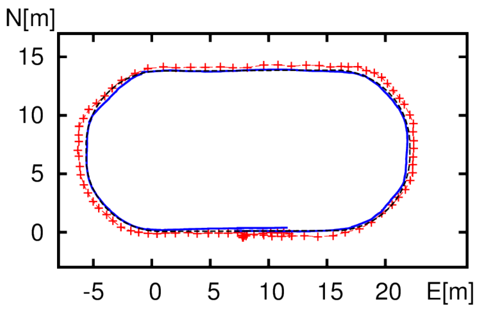
Figure 6.5: Online trajectory following results with the predictor. Reference path for the trajectory follower (black dashed line), the predictor position output (blue line), and the GPS readings (red crosses).



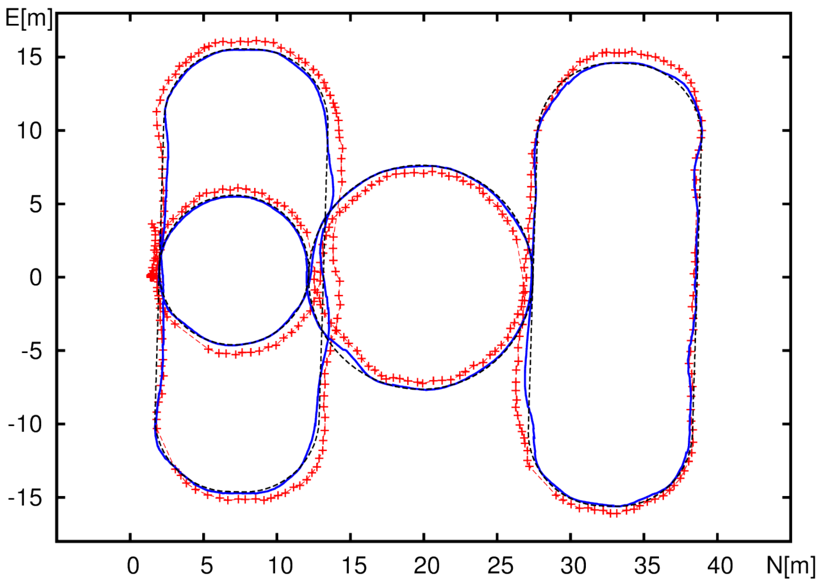
(a) $7\ m/s$



(b) $7\ m/s$



(c) $7\ m/s$



(d) $4\ m/s$

Figure 6.6: Online trajectory following results in more complex conditions: high speed and rough terrains. Reference path for the trajectory follower (black dashed line), the predictor position output (blue line), and the GPS readings (red crosses).

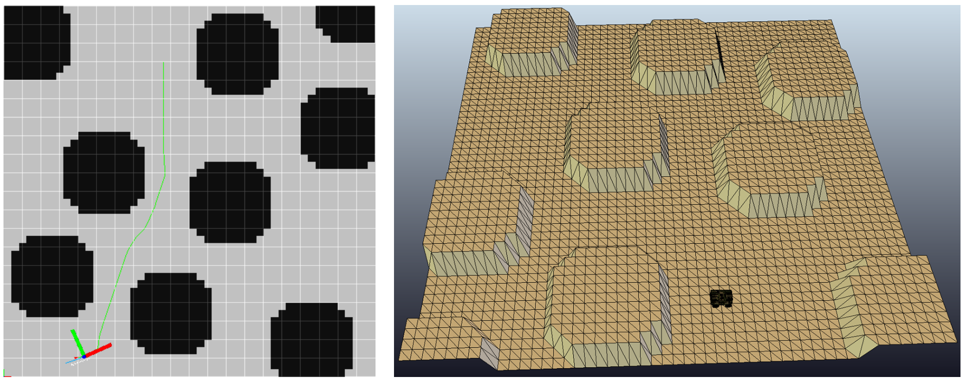
is combined with two rectangular-shaped ones. As it might be expected given the characteristics of the robot, the vehicle follows the path with some difficulties, the few deviations are caused by the irregularities in the terrain.

6.3 Path planning

The path planning module is the newest addition to the control architecture and the ROS node that integrates it in the architecture is still under development. Nevertheless, thanks to the simulation, it is possible to test its functionalities in an environment as close as possible the reality and to identify potential problems, before deploying it on the real robot. Here we provide an example of path planning and consequent trajectory following realized in the simulation. Since the development of the node is still in its earlier stages, we use a map with only two height levels: the terrain level, completely flat, and the obstacles, impossible to be overcome by the robot.

The map is represented by a black and white image (Figure 6.7a), where the obstacles are the black areas. To provide a realistic simulation, we recreated the same map inside the simulator; this is possible thanks to V-Rep, which permits to create terrains from grey-scale images. Figure 6.7b shows the map after we imported it in the scene.

For this experiment, we provide the goal using `rviz`, which allows publishing directly on a topic a point selected on the map with a mouse click. Figure 6.7a shows the path generated by the planner, i.e., the green line, and the position of the robot, i.e., the reference frame. As it is possible to see from Figure 6.8 the robot follows the path generated by the planner with no issue, except for the beginning where the robot performs an initial maneuver to align with the first point of the trajectory. This is caused by the fact that the planner discretizes the map in a finite number of cells, and it places the first point of the trajectory in the middle of the cell closer to the robot. Some solutions have already been proposed in order to solve this issue, for instance, removing from the plan all the points in the immediate proximity of the robot. Understanding this kind of problems is the exact reason why a reliable and realistic simulation is required when testing new algorithms.



(a) `rviz` *(b) V-Rep*

Figure 6.7: The map as seen in `rviz` *(a) and V-Rep (b).*

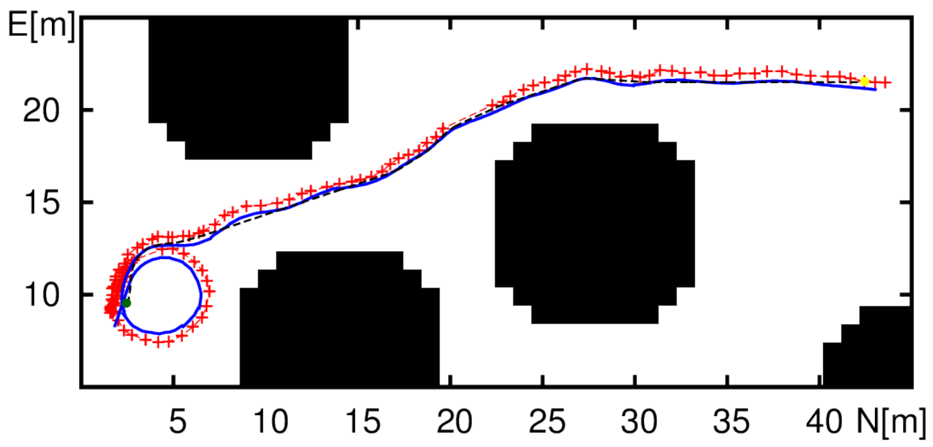


Figure 6.8: Online trajectory following results on a path generated by the planner. Reference path for the trajectory follower (black dashed line), the predictor position output (blue line), and the GPS readings (red crosses). The black areas are the obstacles, the green dot is the first point of the plan, and the yellow dot is the goal.

Chapter 7

Conclusions and future work

7.1 Conclusions

The result of this work is a high-level control architecture developed with ROS that includes localization, path following and path planning functionalities. It is characterized by a high flexibility and modularity, key features when working with a prototype that can undergo many changes in configuration. This architecture, which integrates software modules developed separately, has proven effective during the experiments, obtaining good results in both localization and path following. Moreover, we have produced a simulation environment that simulates the robot and can substitute it seamlessly from the point of view of the architecture. This permits to test and validate the software before deploying it on the robot. In addition, it is useful to perform experiments in condition difficult to obtain or potentially dangerous for the real robot, simplifying future developments of additional functionalities.

7.2 Future work

Currently the robot has all its core functionalities implemented: manual drive, localization, autonomous drive and path planning. However, it is possible to extend and enhance them. As described in Chapter 3 the robot is equipped with two cameras and a laser scanner, while these sensors are integrated in the architecture, the localization module does not use them. Adding them would increase the accuracy and the quality of the estimated position. Moreover, the laser scanner can be used to detect obstacles in front of the robot and to implement an obstacle detection system, useful during both manual and autonomous drive. The results of the experiments presented in Chapter 6 show some of the limitation of the path following algorithm. The oversteer problem highlighted in Figure 6.3 is caused by a speed higher than expected. In order to avoid this behavior the path following module can be enhanced with a system to limit the speed based on the curvature of the trajectory. Moreover, when the robot reaches a faulty state during autonomous drive, like in Figure 6.4d, there is no system that can detect it. A heuristic able to detect and potentially avoid this kind of behavior would increase the general robustness of the system.

Regarding the path planning module, currently it is still in a testing phase, but its functionalities can be extended by planning on two different levels. On a higher level a plan based on a map of the area and a goal defined by the user. While, on a lower level a local plan based on a map created using the laser scanner mounted on the robot and a goal defined by the intersection of the two maps. This system accounts for unpredictable obstacles and for errors in the current position of the robot.

The simulation has proven useful to test and validate the software architecture, but in order to use it for more advanced experiments, it is necessary to improve the robot and the sensors models, also exploiting V-Rep frequent updates. Regarding the robot model, the suspensions have to be updated in order to simulate better the behavior of the real robot, especially in the case of experiments on rough terrains. In order to improve the sensors models it is possible to add more faulty behaviors and noise to match the real ones. For example adding the multipath phenomenon to the GPS, or accounting for engine vibration in the IMU.

Bibliography

[1] Maplesim, high performance physical modeling and simulation. http://www.maplesoft.com/products/maplesim/. Accessed: 2014-09-12.

[2] Vortex dynamics. http://www.vxsim.com. Accessed: 2014-09-12.

[3] Karsten Ahnert and Mario Mulansky. Odeint-solving ordinary differential equations in c++. *arXiv preprint arXiv:1110.3397*, 2011.

[4] Massimo Bertozzi, Alberto Broggi, Gianni Conte, and Alessandra Fascioli. Vision-based automated vehicle guidance: the experience of the argo vehicle. *Tecniche di Intelligenza Artificiale e Pattern Recognition per la Visione Artificiale*, pages 35–40, 1998.

[5] R Boumghar and S Lacroix. Over the hill and far away: aerial/ground cooperation for long range navigation. World Scientific.

[6] Jan F Broenink. 20-sim software for hierarchical bond-graph/block-diagram models. *Simulation Practice and Theory*, 7(5):481–492, 1999.

[7] Alberto Broggi, Massimo Bertozzi, and Alessandra Fascioli. The 2000 km test of the argo vision-based autonomous vehicle. *IEEE Intelligent Systems*, 14(1):55–64, 1999.

[8] Brunati and Porta. Progetto e realizzazione di un servomeccanismo per il controllo dello sterzo di un veicolo atv. Master thesis, Politecnico di Milano, 2008.

[9] Herman Bruyninckx. Open robot control software: the orocos project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528. IEEE, 2001.

[10] Martin Buehler, Karl Iagnemma, and Sanjiv Singh. The 2005 darpa grand challenge. *Springer Tracts in Advanced Robotics*, 36(5):1–43, 2007.

[11] Castelli. Progetto e realizzazione di un servomeccanismo per il controllo della frenata di un veicolo atv. Master thesis, Politecnico di Milano, 2009.

[12] A Conforto. On the development of a search-based trajectory planner for an ackermann vehicle in rough terrains. Master thesis, Politecnico di Milano, 4 2014.

[13] Erwin Coumans et al. Bullet physics library. *Open source: bulletphysics. org*, 4(6), 2006.

[14] Davide Antonio Cucci and Matteo Matteucci. A flexible framework for mobile robot pose estimation and multi-sensor self-calibration. In *ICINCO (2)*, pages 361–368, 2013.

[15] E D'Amelio and F Fontanile. Sintesi legge di controllo per l'esecuzione di path following per un atv. Master thesis, Politecnico di Milano, 4 2013.

[16] Ernst Dieter Dickmanns, Reinhold Behringer, Dirk Dickmanns, Thomas Hildebrandt, Markus Maurer, Frank Thomanek, and Joachim Schiehlen. The seeing passenger car'vamors-p'. In *Intelligent Vehicles' 94 Symposium, Proceedings of the*, pages 68–73. IEEE, 1994.

[17] Hilding Elmqvist, Dag Brück, and Martin Otter. Dymola-user's manual. *Dynasim AB, Research Park Ideon, Lund, Sweden*, 1996.

[18] Tully Foote. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, pages 1–6. IEEE, 2013.

[19] Marc Freese, Surya Singh, Fumio Ozaki, and Nobuto Matsuhira. Virtual robot experimentation platform v-rep: a versatile 3d robot simulator. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 51–62. Springer, 2010.

[20] Douglas W Gage. Ugv history 101: A brief history of unmanned ground vehicle (ugv) development efforts. Technical report, DTIC Document, 1995.

[21] David Gossow, Adam Leeper, Dave Hershberger, and Matei Ciocarlie. Interactive markers: 3-d user interfaces for ros applications [ros topics]. *Robotics & Automation Magazine, IEEE*, 18(4):14–15, 2011.

[22] S Harmon. The ground surveillance robot (gsr): An autonomous vehicle designed to transit unknown terrain. *Robotics and Automation, IEEE Journal of*, 3(3):266–279, 1987.

[23] SY Harmon, G Bianchini, and B Pinz. Sensor data fusion through a distributed blackboard. In *Robotics and Automation. Proceedings. 1986 IEEE International Conference on*, volume 3, pages 1449–1454. IEEE, 1986.

[24] Matthias Hentschel and Bernardo Wagner. Autonomous robot navigation based on openstreetmap geodata. In *Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on*, pages 1645–1650. IEEE, 2010.

[25] Fumio Kanehiro, Hirohisa Hirukawa, and Shuuji Kajita. Openhrp: Open architecture humanoid robotics platform. *The International Journal of Robotics Research*, 23(2):155–165, 2004.

[26] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE, 2004.

[27] P Krüsi and P Frugale. Artor (autonomous rough terrain outdoor robot). http://www.artor.ethz.ch/doku.php?id=robots:artor. Accessed: 2014-09-12.

[28] Marco Langerwisch, Marko Reimer, Matthias Hentschel, and Bernardo Wagner. Control of a semi-autonomous ugv using lossy low-bandwidth communication. In *The Second IFAC Symposium on Telematics Applications (TA). Timisoara, Romania*, 2010.

[29] Séverin Lemaignan, Gilberto Echeverria, Michael Karg, Jim Mainprice, Alexandra Kirsch, and Rachid Alami. Human-robot interaction in the morse simulator. In *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*, pages 181–182. ACM, 2012.

[30] James W Lowrie, Mark Thomas, Keith Gremban, and Matthew Turk. The autonomous land vehicle (alv) preliminary road-following demonstration. In *1985 Cambridge Symposium*, pages 336–350. International Society for Optics and Photonics, 1985.

[31] Markus Maurer, Reinhold Behringer, Dirk Dickmanns, Thomas Hildebrandt, Frank Thomanek, Joachim Schiehlen, and Ernst D Dickmanns. Vamors-p: An advanced platform for visual autonomous road vehicle guidance. In *Photonics for Industrial Applications*, pages 239–248. International Society for Optics and Photonics, 1995.

[32] Markus Maurer, Reinhold Behringer, S Furst, F Thomanek, and ED Dickmanns. A compact vision system for road vehicle guidance. In *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, volume 3, pages 313–317. IEEE, 1996.

[33] Olvier Michel. Webots: a powerful realistic mobile robots simulator. In *Proceeding of the Second International Workshop on RoboCup*, 1998.

[34] Hans P Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. Technical report, DTIC Document, 1980.

[35] Hans P Moravec. *The Stanford cart and the CMU rover*. Springer, 1990.

[36] Nils J Nilsson. Shakey the robot. Technical report, DTIC Document, 1984.

[37] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[38] Eric Rohmer, Surya PN Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1321–1326. IEEE, 2013.

[39] Bernd-Helge Schäfer and Karsten Berns. Ravon - an autonomous vehicle for risky intervention and surveillance. In *International Workshop on Robotics for risky intervention and environmental surveillance - RISE*, 06 2006.

[40] Bernd-Helge Schäfer, Martin Proetzsch, Tim Braun, Jan Koch, Norbert Schmitz, and Karsten Berns. Ravon - a robust autonomous vehicle for off-road navigation. In *European Land Robot Trial - ELROB*, May 2006.

[41] Frank E Schneider, Dennis Wildermuth, Bernd Brüggemann, and Timo Röhling. European land robot trial (elrob) towards a realistic benchmark for outdoor robotics. 2010.

[42] Daniel G Shapiro. Three anecdotes from the darpa autonomous land vehicle project. *AI Magazine*, 29(2):40, 2008.

[43] Russell Smith et al. Open dynamics engine, 2005.

[44] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692, 2006.

[45] A Tikanmaki and J Roning. Development of mörri, a high performance and modular outdoor robot. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 1441–1446. IEEE, 2009.

[46] Paul G Trepagnier, Jorge Nagel, Powell M Kinney, Cris Koutsougeras, and Matthew Dooner. Kat-5: Robust systems for autonomous vehicle navigation in challenging and unknown terrain. *Journal of Field Robotics*, 23(8):509–526, 2006.

[47] Chris Urmson, Charlie Ragusa, David Ray, Joshua Anhalt, Daniel Bartz, Tugrul Galatali, Alexander Gutierrez, Josh Johnston, Sam Harbaugh, William Messner, et al. A robust approach to high-speed navigation for unrehearsed desert terrain. *Journal of Field Robotics*, 23(8):467–508, 2006.

[48] JF. Vasconcelos, G. Elkaim, C. Silvestre, P. Oliveira, and B. Cardeira. Geometric approach to strapdown magnetometer calibration in sensor frame. *Aerospace and Electronic Systems, IEEE Transactions on*, 47(2):1293–1306, 2011.

[49] M Zago. Modellistica e controllo del servomeccanismo di sterzo di un atv. Master thesis, Politecnico di Milano, 4 2012.

Appendix A

A Simulation Based Architecture for the Development of an Autonomous All Terrain Vehicle

A Simulation Based Architecture for the Development of an Autonomous All Terrain Vehicle

Gianluca Bardaro, Davide Antonio Cucci, Luca Bascetta, and Matteo Matteucci

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy

Abstract. In this work we describe a simulation environment for an autonomous all-terrain mobile robot. To allow for extensive test and verification of the high-level perception, planning, and trajectory control modules, the low-level control systems, the sensors, and the vehicle dynamics have been modeled and simulated by means of the V-Rep 3D simulator. We discuss the overall, i.e., high and low-level, software architecture and we present some validation experiments in which the behavior of the real system is compared with the corresponding simulations.

1 Introduction

In this work we present an Autonomous All-Terrain Robot developed starting from a commercial, fuel-powered, All-Terrain Vehicle (ATV), i.e., a Yamaha Grizzly 700. This robot is characterized by an Ackermann steering kinematic and the original vehicle commands have been replaced by servomechanisms controlling the handlebar position, the throttle, and the brake. Multiple sensors have been fitted on the robot to perform perception activities: two laser range-finders, a stereo camera rig, a GPS, an Inertial Measurement Unit (IMU), as well as, wheel and handlebar encoders.

Given the physical dimensions of the robot, the typical operating environment, and the complexity of the system architecture, it has been quite challenging to develop and test all the software components, especially in early stages of development. The main difficulties come from the intrinsic complexity in operating the robot, the little repeatability of experiments, the time consuming activity of fault detection and isolation. Moreover, meteorological and space issues further affected field evaluation, either because a suitable test area was not always available for experiments, and because of safety issues for the vehicle itself, which have a high roll-over risk, and for the people working with it.

To address these challenges we developed a simulation environment in which the vehicle and its sensors are substituted by a simulator in a way that is transparent with respect to the high-level perception and control software architecture. In contrast with respect to classical hardware-in-the-loop techniques, in which key elements of the real system, which might be difficult to model, replace their simulated counterpart, here the real system and the environment are replaced with models without changes in the robot control software.

In our work we employed the Virtual Robot Experimentation Platform (V-REP) [7], a physical simulator which relies on a distributed and modular approach and allows to model complex scenarios in which multiple sensors and actuators operate asynchronously at various rates. Other simulator were available, such as Gazebo [10], which is mainly focused on robotic applications, and Dymola [4], which instead focuses on highly accurate multi-domain, multi-body, physical, simulations. We decided to use V-REP instead of Dymola because of its capability in simulating the vehicle sensors and we preferred V-REP to Gazebo for its ease of use. The high-level perception and control architecture of the robot is implemented relying on the Robot Operating System [13], an open source framework which has recently become popular in the literature for its turn-on-and-go functionalities, easiness of deployment, large community, and support.

The use of simulators is common in robotics and several works related to the use of a simulator in the development of an autonomous all terrain robot, and autonomous robots in general, have been presented in the literature: in [9] a high fidelity model, including sensors, is developed to study the behavior of an autonomous ATV, focusing on the simulation itself, rather than the integration with the robot architecture. A simulator is also used in [8] in the actual robot architecture for real-time path planning, where the aim is to foresee potential collisions and change the plan accordingly. In [12], the SimRobot simulator is introduced and example applications in the RoboCup competition are discussed.

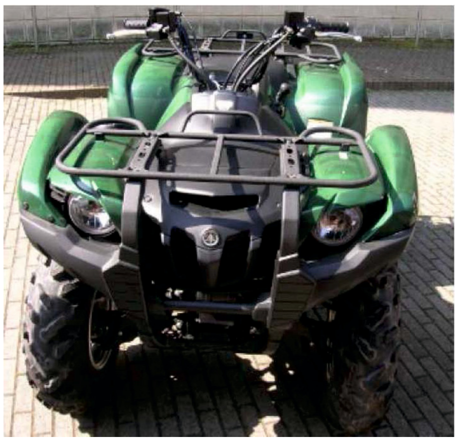
This work is organized as it follows: in Section 2 the overall robot architecture is briefly discussed. Next we move to the high-level perception and control modules implemented employing the ROS framework. In Section 3 we present the simulation environment and we discuss sensor and vehicle models. Finally, in Section 4 we validate our approach comparing the behavior of the simulated system with the real one during autonomous trajectory following experiments.

2 The Quadrivio ATV

In this section we briefly review the vehicle specifications and the developed hardware and software architectures. The original vehicle used is a Yamaha Grizzly 700 (see Figure 1a), a commercial fuel-powered utility ATV, specifically designed for agriculture work. For the purposes of the project, the vehicle cover has been removed and substituted with an aluminum one; this new cover allows to easily accommodate for control hardware and sensors. Furthermore, the vehicle has been equipped with three low-level servomechanisms, each one with its own control loop, to automatically regulate the steer position, the throttle aperture and the braking force [2][3]. Figure 1b shows the vehicle after customization. The main characteristics of the vehicle are listed in Table 1.

2.1 The Hardware Architecture

In order to allow for teleoperation and autonomous navigation, an on-board hardware/software control architecture has been developed.


(a) Original Yamaha Grizzly 700

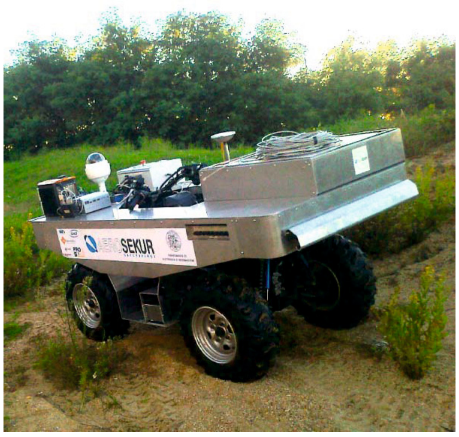

(b) Quadrivio ATV

Fig. 1. On the left the original all-terrain vehicle, on the right the vehicle after the changes to make it autonomous

The architecture is divided in two different layers: the higher level is developed using ROS and is responsible for acquiring data from external sensors, such as GPS, magnetometer, Inertial Measurement Unit (IMU), cameras and laser range-finders. Moreover, it hosts the modules for localization, path planning, high-level trajectory control and autonomous driving. The lower level acts as an interface between the vehicle servomechanisms and the ROS architecture: it receives desired setpoints from the higher level, reads the handlebar angle, throttle ratio, vehicle speed measurements and runs the low-level control loops.

To implement such an architecture that includes high-level and low-level tasks, a multi-layered and multiprocessor hardware/software architecture is required, which consists of: an industrial PLC, which allows a good compromise between the hard real-time requirements and high-level programming, and a standard i5 PC, on which runs the high-level ROS architecture (perception, localization,

Table 1. Vehicle characteristics

Main characteristics of the vehicle	
Engine type	686cc, 4-stroke, liquid-cooled, 4 valves
Drive train	2WD, 4WD, locked 4WD
Transmission	V-belt with all-wheel engine braking
Brakes	dual hydraulic disc (both f/r)
Suspensions	independent double wishbone (both f/r)
Steering System	Ackermann
Dimensions (LxWxH)	2.065 x 1.180 x 1.240 m
Weight	296 Kg (empty tank)

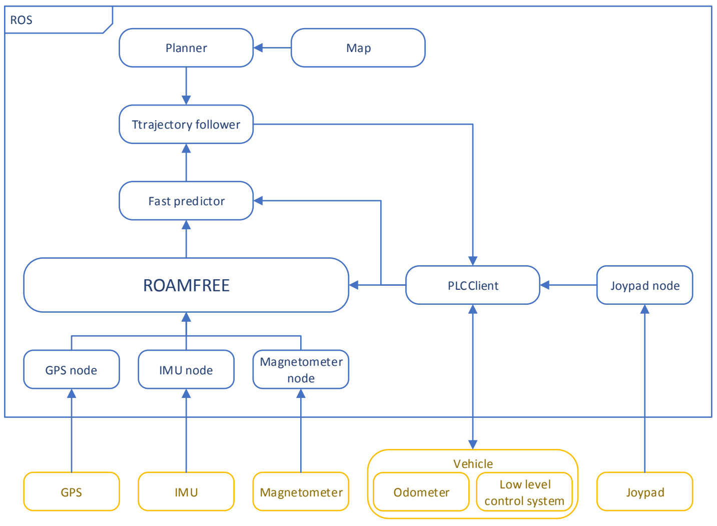


Fig. 2. The real system architecture

obstacle avoidance, medium-long range navigation, planning, etc.). Communication between the two layers is obtained through an Ethernet link.

2.2 The Software Architecture

Figure 2 shows the main modules of the high-level software architecture and their relations with the external sensors and the vehicle servomechanisms. These modules live as independent applications running on the standard PC and the communication between them is guaranteed by the ROS middleware.

The core part of the perception architecture consists in the localization node, which is based on ROAMFREE [5]. This open source framework provides out-of-the-box 6-DOF pose tracking fusing the information coming from an arbitrary number of information sources such as wheel encoders, inertial measurement units, and so on[1]. In ROAMFREE, high-level measurement models are used to handle raw sensor readings and provide calibration parameters to account for distortions, biases, misalignments between sensors and the main robot reference frame. The information fusion problem is formulated as a fixed-lag smoother and it runs in real time tanks to efficient implementations of the inference algorithms (for further details see [6], and [11]). At the present stage of development the localization module estimates the robot poses exploiting vehicle kinematic data (i.e., the handlebar position and the rear wheel speed), GPS, magnetometer, and the gyroscopes in the inertial measurement unit.

[1] `http://roamfree.dei.polimi.it`

The pose estimate is generated by the localization node at a frequency of 20 Hz. However, due to the latencies introduced by the ROS network, delays in the trajectory control loop which affect the system stability can occasionally arise. In order to prevent the detrimental effects of these delays, we introduced a predictor node. This node computes a prediction of the future robot pose at a frequency of 50 Hz; this prediction is based on the latest available global pose estimate and on the integration of the Ackermann kinematic model with the kinematic readings from the vehicle.

Given a map and a goal, a planner node, based on the SBPL library [1], produces a global path, which is then fed to a lower level trajectory following module. This module computes setpoints for the vehicle speed and handlebar angle, based on the current pose and velocity estimates, and on the planned trajectory. These setpoints are sent to the low-level regulators by a ROS node communicating with the PLC, which additionally acts as a multiplexer between the autonomous drive and the manual setpoints, depending on the current operating mode.

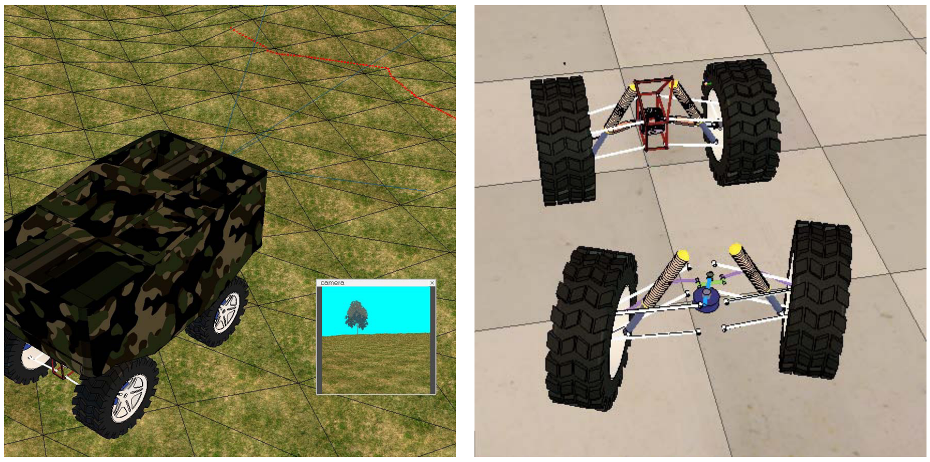
3 The Simulation Environment

The software architecture in Figure 2 has been designed introducing a decoupling layer between the real robot and the high-level perception and control software. This layer is composed by ROS nodes that respectively handle the GPS and the IMU sensors, and the communication module between the standard PC and the PLC. In this section we describe how we have replaced the real vehicle with a physical simulator and how we set up vehicle and sensor models so that they accurately mimic the real robot.

3.1 The Simulator

The vehicle simulator was developed using V-Rep [14], a software for robot modelling offering an accurate physics simulation. This software has been chosen for some of its features that fit particularly well with the requirements of our application. First of all, it is simple to set up and use with its integrated development environment, it has a library with various examples of robots already modeled, and one of them is particularly similar to our vehicle in terms of kinematics and suspension geometry. Another important feature is the possibility to control every object in the simulation with a remote API, that allows the integration with ROS, making it suitable for interacting with our software architecture.

One important issue which has to be addressed in coupling a simulator with a control architecture is to make sure that they share a global time reference. In our case, this was obtained enabling the `use_sim_time` parameter is ROS and having V-Rep publishing the current simulation time on the `clock` ROS topic. This is particularly useful when challenging simulations are run which involve complex terrain or environments and cannot be carried out in real-time.



(a) The customized model (b) Suspensions and Ackermann steering

Fig. 3. The vehicle model used in simulation

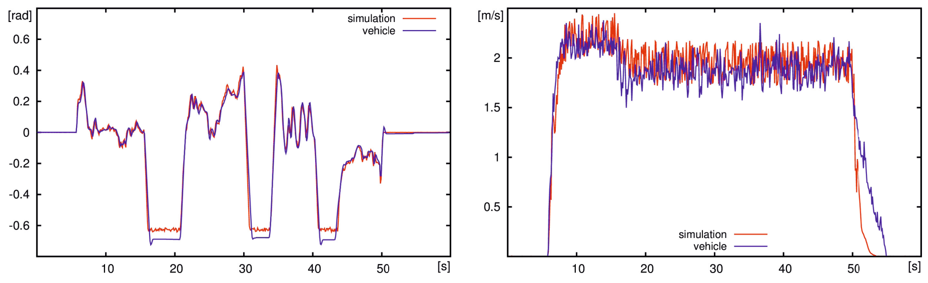
3.2 The Vehicle Model

V-Rep offers multiple built-in vehicle models. We chose one that shares the Ackermann steering and the suspension geometry with our vehicle and we customized it to match the Quadrivio ATV specifications. The vehicle characteristics required to set up the model are listed in Table 2, while Figure 3a shows the customized model in which it is possible to see the image from the camera and the trace of the laser range-finder on the terrain. Figure 3b shows details of the Ackermann steering and suspensions.

The next step in vehicle simulation was to ensure the real vehicle and the simulation model share the same dynamic and kinematic behavior. In particular, we required that the step response for the handlebar and the speed loops on the real vehicle and in simulation were similar. As we are not interested in reproducing the engine behavior or in studying the dynamics of the steering motion control system, but only in simulating the overall vehicle dynamics, the simulator does not include an accurate model of the steering column and of the engine. Instead, the steer and speed loops, both based on a PID controller, were

Table 2. Model parameters

Vehicle model specifications.	
Track	1920 mm
Wheelbase	1250 mm
Front wheel (WxH)	201 x 635 mm
Rear wheel (WxH)	247 x 635 mm
Weight (estimated)	390 kg



(a) Handlebar position response (b) Vehicle linear speed response

Fig. 4. Plot of the vehicle actuators step responses on the real vehicle and on the the simulated one

directly modelled in such a way that the simulated responses of these controlled systems were as close as possible to the experimental ones.

We recorded data for the handlebar step response of the real vehicle and then we tuned the PID regulators that control the handlebar column in the V-Rep simulator so that the simulated vehicle handlebar position step response matches the one of the real vehicle. In particular, setpoints for the handlebar position and vehicle speed were recorded while being sent to the real vehicle, then we feed the simulated vehicle with the same setpoints, allowing to tune the simulator response. Figure 4a shows a comparison between the handlebar response for the real vehicle (blue line) and the simulated one (red line). It is possible to see that the two behaviors substantially match, but the simulated steer cannot reach a value as high as the real one, because of geometric limitations in the original model. However, we obtained a reasonable behavior in common operation ranges.

For the speed step response we implemented a custom PID controller, which controls the torque applied to the motor joint minimizing the error over the target speed. It was not possible to use the one integrated in the simulator because the model uses a motorized joint, which models an electric motor, while the vehicle has a fuel-powered engine with a significantly different characteristic. After the tuning with field data of the motor PDI, we obtained a good matching behavior in the acceleration phases, while there is still a slight difference in deceleration due to the difficulties in modeling the engine braking when the throttle setpoint is suddenly decreased (see Figure 4b).

3.3 The Sensor Models

After modeling the vehicle we added the sensors: GPS, IMU, magnetometer and odometer. Most of them were already available in V-Rep, but they lacked the ROS integration and they did not account for noise and fault situations.

The sensors are realized with a “two layers” approach; the first layer is implemented directly inside the simulator, it consists in the sensors itself and a script that prepares and publishes ROS messages. The second layer is outside the simulator and it consists in a ROS node that reads the messages published by the simulator and converts them into a format which matches the one produced by sensors on the real vehicle. This double conversion has the aim of obtaining the decoupling between the simulator and the high-level perception and control architecture; indeed, from the point of view of the high-level architecture, there is no difference between the real sensors and the simulated ones.

The following are the sensor we have simulated with a brief description:

– GPS: V-Rep provides already a simulated GPS providing x/y/z-coordinates which are compatible with the East-North-Up (ENU) reference frame used in our architecture, so no further conversion was needed. The node coupled with this sensor builds the correct ROS message introducing some Gaussian noise (derived from real data) and allows to model random downtimes, to simulate for real world GPS unavailability;
– IMU: models for accelerometers and gyroscopes are provided by the simulator out of the box. The raw readings are published as ROS messages and converted in the desired format;
– Magnetometer: to implement this sensor we extract the current vehicle model orientation with respect to the global fixed reference frame; from it we can compute a simulated value for the Earth magnetic field reading in the sensor reference frame. However, on the real vehicle, hard and soft iron distortion affect the magnetometer readings. These are considered by employing the sensor model presented in [15], whose parameters have been calibrated by means of the sensor self-calibration capabilities of the ROAMFREE sensor fusion framework;
– Odometer: the vehicle rear wheel speed and the current handlebar position are extracted from the current status of the relevant joints in V-Rep, and a TCP socket is employed to communicate with the PLCClient ROS nodes, in a way that mimics the behavior of the X20 industrial PLC.

Moreover, there are two sensors that are simulated but not currently used for localization:

– Laser: the simulator provides a laser scanner out of the box. The associated script required some adjustment to publish the correct ROS message;
– Camera: V-Rep offers a highly customizable vision sensor that we used to realize a camera that matched our needs.

Figure 5 shows how the overall architecture changes when the simulator substitutes the vehicle and the real sensors. Every sensor now is substituted by its simulated counterpart, yet nothing changes from the point of view of the perception and control modules, since the communication is done on the same ROS topics. The PLCClient, in charge of communicating with the low-level control of the vehicle, interacts with a simulated low-level control system through a local socket connection. Inside the simulator a script converts setpoints from the PLCClient into setpoints of the model joints, another one collects odometry and sends it back to the ROS node. Migration from the simulated environment and the real vehicle is simple and requires only the change of few parameters.

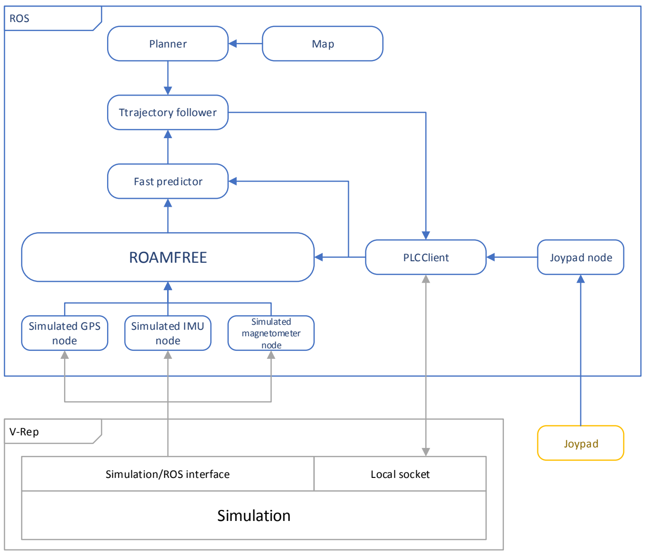


Fig. 5. Overall architecture in the simulation setup

4 Experimental Evaluation

In this section we discuss some autonomous trajectory following experiments done on the real vehicle and in the simulation environment. We employ an eight-shaped trajectory originating 1 meter ahead with respect to the current pose of the robot. The two circles have a diameter of respectively 18 and 12.5 meters.

Figure 6 shows the results of six experiments done with the real vehicle, in it we have plotted the reference path, the robot position, estimated by ROAMFREE, and the raw GPS readings. Especially in the first experiment (Figure 6a), but also in the other ones, it is possible to see how the trajectory is followed with reasonable accuracy, and with ROAMFREE being able to account for substantial multi-path effect compromising GPS readings.

Figure 7, instead, shows the results of the same experiments done with the simulator In this case the GPS, like all the other sensors, is simulated and it is possible to appreciate its simulated faulty behavior. The robot position is estimated using measurements given by the simulated sensors as they were real, no special configuration is necessary to use them.

In Figure 7a, and 7b, it is possible to see that multipath effect compromises the real GPS sensor readings. This happens when the receiver tracks a replica of the GPS signal which is reflected by environmental features such as buildings and trees. This effect is hard to model and it has not been simulated in V-Rep, even though, if we restrict to its effect on localization, a noise model which accounts for a random transformation to be applied occasionally on the GPS readings could be considered.

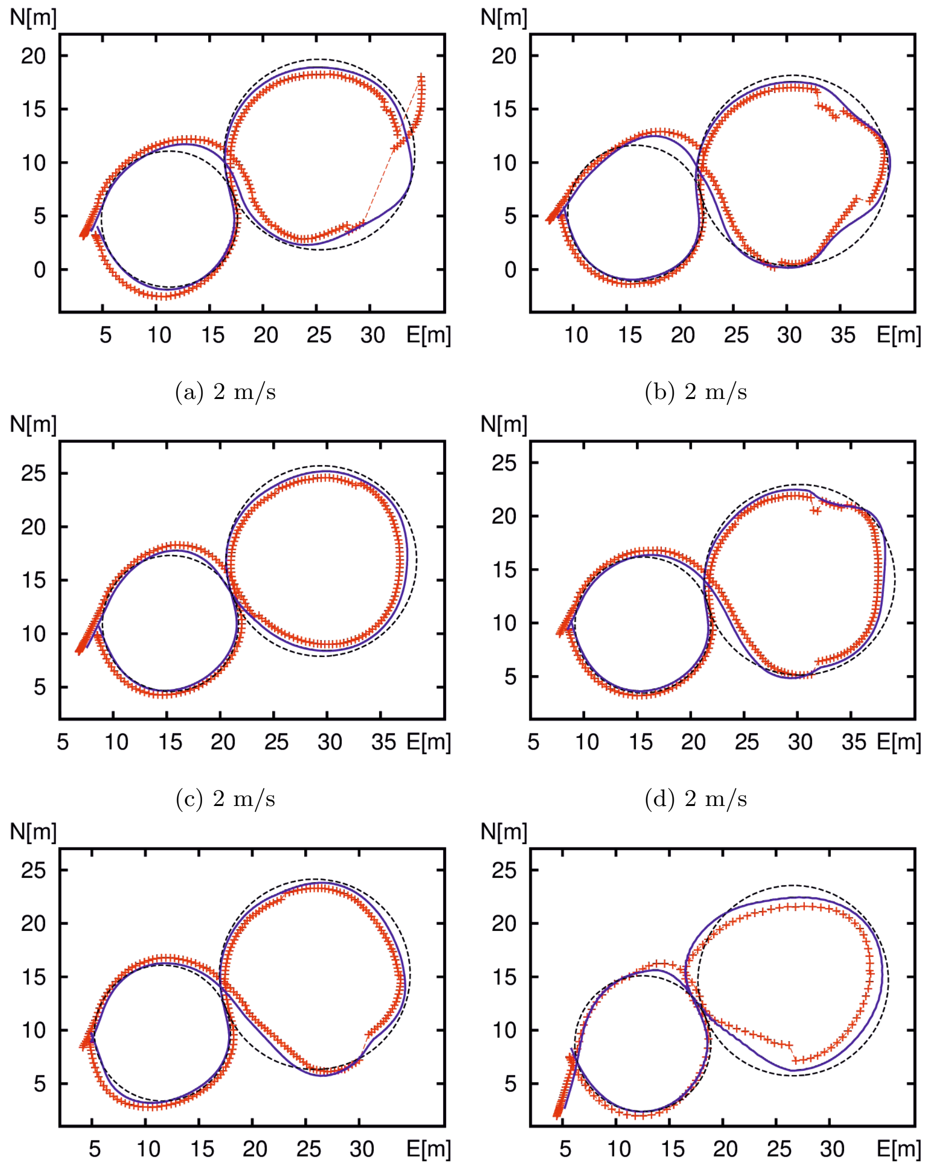
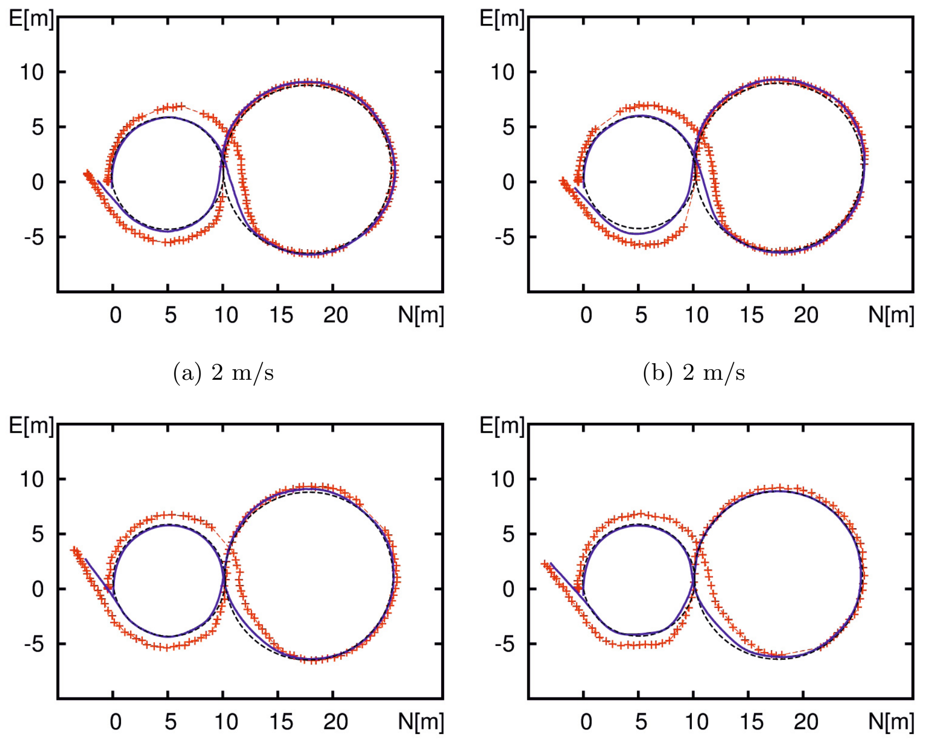


Fig. 6. Online trajectory following results. Reference path for the trajectory follower (black dashed line), the ROAMFREE position output (blue line), and the GPS readings (red crosses).



(a) 2 m/s (b) 2 m/s

(c) 3 m/s (d) 4 m/s

Fig. 7. Simulation trajectory following results. Reference path for the trajectory follower (black dashed line), the ROAMFREE position output (blue line), and the GPS readings (red crosses).

5 Conclusions

In this work we have presented and validated a simulated environment which provides an alternative when experiments on the real robots cannot be afforded, and ultimately simplifies the development and the testing of complex robotic architectures. As described in Section 3, the simulator transparently substitutes the real vehicle and its sensors. This is possible thanks to the highly modular ROS architecture and to the native integration of V-Rep with it. The simulator does not account for latency of sensors, either internal or caused by the communication, and this makes a simulation more ideal than we would like it to be, therefore a possible improvement to the current work could be addition of latency to sensors. The next step is to exploit the features of V-Rep to test the robot on rough terrains, since the simulator permits to add complex terrains that can be difficult to find in the real world, or that are too risky for the vehicle.

References

1. http://wiki.ros.org/sbpl
2. Bascetta, L., Magnani, G.A., Rocco, P., Zanchettin, A.M.: Design and implementation of the low-level control system of an all-terrain mobile robot. In: 2009 International Conference on Advanced Robotics (ICAR), pp. 1–6. IEEE (2009)
3. Bascetta, L., Cucci, D., Magnani, G., Matteucci, M., Osmankovic, D., Tahirovic, A.: Towards the implementation of a mpc-based planner on an autonomous all-terrain vehicle. In: Proceedings of Workshop on Robot Motion Planning: Online, Reactive, and in Real-time (IEEE/RJS IROS 2012), pp. 1–7 (2012), http://cs.stanford.edu/people/tkr/iros2012/schedule.php
4. Brück, D., Elmqvist, H., Mattsson, S.E., Olsson, H.: Dymola for multi-engineering modeling and simulation. In: Proceedings of Modelica, Citeseer (2002)
5. Cucci, D.A., Matteucci, M.: Position tracking and sensors self-calibration in autonomous mobile robots by gauss-newton optimization. In: 2014 IEEE International Conference on Robotics and Automation (ICRA). IEEE (to appear, 2014)
6. Cucci, D.A., Matteucci, M.: On the development of a generic multi-sensor fusion framework for robust odometry estimation. Journal of Software Engineering for Robotics 5(1), 48–62 (2014)
7. Freese, M., Singh, S., Ozaki, F., Matsuhira, N.: Virtual robot experimentation platform V-REP: A versatile 3D robot simulator. In: Ando, N., Balakirsky, S., Hemker, T., Reggiani, M., von Stryk, O. (eds.) SIMPAR 2010. LNCS, vol. 6472, pp. 51–62. Springer, Heidelberg (2010)
8. Hellstrom, T., Ringdahl, O.: Real-time path planning using a simulator-in-the-loop. International Journal of Vehicle Autonomous Systems 7(1), 56–72 (2009)
9. Jayakumar, P., Smith, W., Ross, B.A., Jategaonkar, R., Konarzewski, K.: Development of high fidelity mobility simulation of an autonomous vehicle in an off-road scenario using integrated sensor, controller, and multi-body dynamics. Tech. rep., DTIC Document (2011)
10. Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004), vol. 3, pp. 2149–2154. IEEE (2004)
11. Kümmerle, R., Grisetti, G., Strasdat, H., Konolige, K., Burgard, W.: g^2o: A general framework for graph optimization. In: 2011 IEEE International Conference on Robotics and Automation (ICRA), pp. 3607–3613. IEEE (2011)
12. Laue, T., Spiess, K., Röfer, T.: SimRobot – A general physical robot simulator and its application in roboCup. In: Bredenfeld, A., Jacoff, A., Noda, I., Takahashi, Y. (eds.) RoboCup 2005. LNCS (LNAI), vol. 4020, pp. 173–183. Springer, Heidelberg (2006)
13. Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.: ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software, vol. 3 (2009)
14. Rohmer, E., Singh, S., Freese, M.: V-REP: A versatile and scalable robot simulation framework. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 1321–1326 (2013)
15. Vasconcelos, J., Elkaim, G., Silvestre, C., Oliveira, P., Cardeira, B.: Geometric approach to strapdown magnetometer calibration in sensor frame. IEEE Transactions on Aerospace and Electronic Systems 47(2), 1293–1306 (2011)