POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in
Computer Science and Engineering

A MODEL PREDICTIVE CONTROL ARCHITECTURE FOR AN UNMANNED ELECTRIC VEHICLE

Relatore: Ing. Matteo MATTEUCCI

Correlatore: Ing. Luca BASCETTA

Tesi di Laurea di:

Pietro BALATTI Matr. 837376

Alessandro CIANFEROTTI Matr. 836855

Anno Accademico 2015 - 2016

alla realizzazione dei nostri sogni,
per molti utopie.

Ringraziamenti

Innanzitutto vogliamo ringraziare il Professor Matteo Matteucci per averci dato l'opportunità di lavorare su un campo così interessante e per averci indirizzato sulla strada giusta durante lo sviluppo della tesi. Successivamente vogliamo ringraziare il Professor Luca Bascetta per averci seguito nelle fasi fondamentali del lavoro. Inoltre, esprimiamo la nostra gratitudine a Davide Cucci, che ci ha assistito da lontano, e a Gianluca Bardaro per averci aiutato all'inizio del nostro lavoro. Infine, vogliamo ringraziare l'AIRLab e tutti i ragazzi che si sono susseguiti nel corso di questi mesi per aver animato le nostre giornate in laboratorio.

Pietro e Alessandro

Il mio primo pensiero non può che andare ai miei genitori, Piera e Ulisse, che hanno sempre creduto in me supportando le mie scelte e cercando di indirizzarmi sempre sulla via che ritenevano più corretta, senza tuttavia impormi mai nessuna decisione. Li ringrazio in particolare per avermi trasmesso valori e passioni che fanno di me quello che sono, l'aspirazione a sognare in grande e la forza di non arrendersi mai di fronte alle difficoltà. Il modo in cui mi hanno supportato e aiutato a superare questo periodo difficile dal punto di vista salutare ne è il più grande esempio.

Ringrazio la splendida grande famiglia di cui faccio parte, zii/e e cugini/e. Siamo tutti consapevoli di avere una famiglia speciale e unica, che non perde l'occasione per festeggiare e che si stringe forte e compatta in ogni momento di difficoltà. Credo che questa sia una delle più grandi fortune che si possano desiderare; grazie di cuore a tutti voi, perché ci siete stati in momenti bui e perché so che ci sarete sempre a condividere momenti di gioia e soddisfazione come questo.

Ricordo tutti gli amici e coinquilini che mi hanno accompagnato in questi anni. Grazie a Gnagneo che mi ha insegnato cos'è l'assurdo, alla squadra di Zii e Nipoti che mi ha trasmesso valori importanti nella vita quali l'essere RIC, ai Fratelli Grimm che mi hanno insegnato a raccontare fiabe e in generale a tutta la compagnia di Giancarlos che con caparbietà non ha mai smesso di lottare per il Trofeo. Non dimentico anche chi c'è sempre stato e che ora è fisicamente dall'altra parte del mondo, ma sentimentalmente ancora qui.

Un pensiero particolare va a Lisa che è la persona che mi è stata più vicina in questi anni; avrei preferito scoprirlo in altri modi, ma ho capito che sarà un grande medico da ogni punto di vista.

Ricordo le persone che hanno reso la mia vita universitaria più piacevole e con cui ho condiviso intere sessioni di esami e anni di lezioni, in particolare Toto e Vale. L'università mi ha anche dato l'opportunità di svolgere una delle esperienze più belle della mia vita, l'Erasmus a Valencia; senza di voi non sarebbe stato lo stesso: grazie ai boludos Lorenzito y Fede.

Per ultimo, non in ordine di importanza, ringrazio Cianfe, con cui ho condiviso lo sviluppo di questa tesi ma non solo; da vero amico non ha mai dubitato di intraprendere questo percorso nonostante le difficoltà che sapeva avrebbe potuto comportare.

Pietro

Milano, dicembre 2016

I miei più sentiti ringraziamenti vanno ai miei genitori, Teresa e Vero, per aver reso possibile tutto questo e per aver sempre creduto in me. Se sono arrivato fino a questo punto è solo grazie a loro e gliene sarò per sempre grato. Non mi hanno mai fatto mancare nulla e mi hanno sempre aiutato nei momenti di difficoltà. Con loro voglio ringraziare anche mio fratello Gianfranco per essere sempre stato la mia principale fonte di ispirazione e per essere sempre stato presente nei momenti di solitudine. A loro tre, che sono la mia colonna portante, un grazie di cuore.

Un ringraziamento speciale va a mio nonno Gianfranco per avermi accompagnato lungo tutto il percorso universitario e per avermi sempre dato importanti lezioni di vita. Ringrazio anche i miei zii, Maria e Biagio, che considero come dei secondi genitori e mia cugina Genoveffa per essersi sempre interessati ai progressi del mio percorso. Voglio ringraziare anche mio nonno Elia per avermi insegnato, attraverso le sue gesta, i valori della famiglia.

Grazie a Denny, Luchino, Sorbi e Mauro, per aver condiviso gioie e sofferenze durante le sessioni di esami e per aver reso la mia esperienza universitaria più piacevole.

Dedico l'ultimo ringraziamento, ma non in ordine di importanza, al mio amico Pietro, compagno di tesi e di viaggio, con cui ho avuto l'onore di vivere e condividere la parte finale dalla mia esperienza universitaria. Un pensiero speciale va a lui per avermi insegnato a non mollare mai e per essere stato un esempio di forza e tenacia.

Alessandro

Milano, dicembre 2016

Abstract

In this thesis, a Model Predictive Control architecture for an autonomous electric vehicle has been designed and developed together with the dynamic simulation used for its validation. The system has been realized first using a single-track model and then simulating a real vehicle, i.e., a *"Polaris Ranger XP 900"* via a 3D physics simulation. The work of this thesis starts from an MPC controller already developed in an other thesis in *Simulink* which was validated under ideal conditions. We developed also a localization system necessary to test our system in a simulation environment. The development of the software architecture has been done using the well-known and widely used framework for robotics ROS (Robot Operating System). 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. To verify and analyze the system behavior in control conditions, we have developed two simulators: one representing an ideal and simplified situation, exploiting the mathematical description of the single-track model, and one using Gazebo, a simulator that allows to model complex scenarios. The contribution of this thesis can be divided in four parts: the first one regards the general architecture, the second one is relative to the vehicle modeling, the third one is dedicated to the vehicle localization and the fourth one concerns the MPC controller. The overall design of the control architecture has been validated through an extensive experimental activity performed within the two simulation environments.

Keywords: Unmanned vehicles, Model Predictive Control, single-track, ROAMFREE, Gazebo, odeint, CPLEX.

Sommario

In questa tesi, è stata progettata e sviluppata l'architettura di controllo di un veicolo autonomo implementando un controllore predittivo basato su modello MPC (Model Predictive Control) insieme alla simulazione dinamica utilizzata per la sua validazione. Il sistema è stato validato infatti prima utilizzando un modello single-track, poi con la simulazione di un veicolo reale, ovvero un *"Polaris Ranger XP 900"*.

Il lavoro di questa tesi parte da un controllore MPC precedentemente sviluppato in un'altra tesi tramite l'utilizzo di un linguaggio di programmazione a più alto livello, MATLAB. E' stato quindi necessario convertirlo ripensando all'architettura in modo da ottimizzare le sue performance ed aumentarne la sua modularità. E' stato poi implementato un sistema di localizzazione necessario per poter testare il sistema in un ambiente di simulazione.

Lo sviluppo dell'architettura è stato effettuato usando un framework per la robotica noto e ampiamente utilizzato, i.e., ROS (Robot Operating Systems), caratterizzato da un'elevata modularità, implementata con lo stile architetturale publish-subscribe, e dalla disponibilità di diversi moduli già sviluppati e testati. Ciò ha permesso di sviluppare un'architettura flessibile appropriata per un prototipo, che può essere sottoposto a svariate modifiche durante il suo sviluppo, rimanendo comunque un sistema sufficientemente robusto e attuabile per un utilizzo a lungo termine.

ROS semplifica l'integrazione di più moduli sviluppati separatamente. In particolare, uno dei principali moduli che è stato integrato nell'architettura è ROAMFREE (Robust Odometry Applying Multisensor Fusion to Reduce Estimation Error), che è una libreria utilizzata per fare una fusione dei dati provenienti da diversi sensori con lo scopo di stimare la posizione di un robot. Oltre all'integrazione di questa libreria, è stato aggiunto un modulo proposto all'interno di un'altra tesi, chiamato fastPredictor, che partendo dalla posizione stimata da ROAMFREE integra le informazioni di odometria per ottenere una stima più veloce e ridurne i ritardi di computazione.

Nel campo dei veicolo autonomi, il ruolo della simulazione risulta essere fondamentale. Prima di poter utilizzare il sistema su di un veicolo reale, è richiesta la verifica e analisi dello stesso. A riguardo, abbiamo simulato il funzionamento della nostra architettura su Gazebo, un simulatore che permette di modellizzare scenari complessi utilizzando anche modelli di robot preesistenti. Inoltre, Gazebo offre la possibilità di simulare diversi sensori e di interfacciarsi facilmente con ROS. Il risultato è un ambiente di simulazione che può sostituire completamente il veicolo reale e che permette di effettuare esperimenti complessi.

Il contenuto di questa tesi può essere diviso in quattro parti: la prima riguarda l'architettura generale, la seconda è relativa alla modellizzazione del veicolo, la terza è dedicata alla localizzazione e la quarta è inerente al controllore MPC. Prima di tutto forniamo una descrizione generale dell'architettura sviluppata, spiegandone la suddivisione e come avviene la comunicazione tra i vari componenti del sistema. Successivamente, analizziamo dettagliatamente la modellizzazione del veicolo e i vari step di simulazione dello stesso. Inoltre, nella stessa sezione, vengono descritti i sensori utilizzati ai fini della localizzazione. In seguito, viene illustrato il modulo riguardante la stima della posizione del veicolo, specificandone il funzionamento e come viene migliorato. Si fornisce successivamente una breve descrizione del MPC preso come riferimento per il nostro lavoro e la relativa implementazione in ROS. Nell'ultima parte della tesi è possibile trovare i risultati sperimentali ottenuti durante i test effettuati.

Parole chiave: Veicoli autonomi, Model Predictive Control, single-track, ROAMFREE, Gazebo, odeint, CPLEX.

Contents

List of Figures

List of Tables

Introduction

The purpose of this thesis is to design and develop a software architecture of a self-driving electric vehicle.

A driverless car is an automobile that has an autopilot system allowing it to safely move from one place to another without help from a human driver. Ideally, the only role of a human in such a vehicle would be indicating the destination. The implementation of driverless cars could theoretically lead to many improvements in transportation. There are, however, many obstacles to successfully implementing the concept as a common and effective method of transportation. This is especially true in situations in which a car on autopilot would need to safely navigate alongside normal cars directed by human drivers. To be useful, a driverless car must be able to navigate to a given destination based on passenger-provided instructions, avoid environmental obstacles, and safely avoid other vehicles. There are many potential advantages to using a driverless car instead of a traditional human-controlled vehicle. A driverless car would not be subject to human error, one of the most common causes of car accidents. There would be little need for driver's licenses, highway patrols, extensive traffic laws, and even stop signs or street lights. Such vehicles would not be affected by erratic human drivers and would, therefore, be able to drive very close together. This could lead to a situation in which high road density would not have a detrimental effect on speed, so many cars could travel close together while maintaining a high average speed.

The great improvement in control strategies and electronic equipment, which are becoming more and more efficient, powerful and economical, is one of the main reasons of the growing interest of various automotive brands in autonomous vehicles, which are no longer a dream, but can be reality in the near future. Another important development that has allowed the growth of this sector is the considerable progress in data elaboration and sensors, such as camera, radar, LIDAR, which are necessary for the autonomous perception.

In this thesis, the Model Predictive Control architecture of an au-

tonomous electric vehicle has been designed and developed together with the dynamic simulation used for its validation. The system has been realized at first using a single-track model and then simulating a real vehicle, i.e., a *"Polaris Ranger XP 900"*.

This technique exploits a model of the process under investigation to obtain the control output, through the minimization of an objective function under some operational restrictions. Model Predictive Control has been employed since the last years of the 70's, mainly to control the processes in chemical plants and oil refineries. However, due to its prediction properties, MPC is very suitable for autonomous vehicle control applications. In fact, it calculates future changes of the state variables and related quantities, basing on the current measurements and the current dynamic state and these changes are used to compute the optimal control sequence.

The work of this thesis starts from an MPC controller already developed in an other thesis with a high-level programming language, MATLAB. It has therefore been necessary to convert it reorganizing the original architecture in order to optimize its performance and to improve its modularity. Moreover we developed a localization system needed to test our system in a simulation environment.

Our goal has been to develop a flexible software architecture to make a vehicle reach autonomously a target point avoiding obstacles. In order to achieve a high level of modularity, the development of the software 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.

Since for autonomous driving the localization is crucial, 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 implement a localization system and we couple it with a module developed within an other thesis that integrates odometry data to achieve a faster local estimate based on the absolute one provided by ROAMFREE.

Working with an autonomous vehicle, the role of the simulator is fundamental. Before using the system on a real vehicle, it is required to verify

and analyze its behavior. Thus, we have developed two simulators: one representing an ideal and simplified situation, exploiting the mathematical description of the single-track model, and one using Gazebo, a simulator that allows to model complex scenarios. It offers preexisting robot models with multiple sensors and provides a native and simple integration with ROS. 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.

The contents of this thesis can be divided in four parts: the first one regarding the general architecture, the second one relative to the vehicle modeling, the third one dedicated to the vehicle localization and the fourth one concerning the MPC controller. First of all we provide an overview of the whole architecture describing how it is composed and how the communication among them is handled. Then, we analyze the vehicle modeling and the simulation steps. Moreover, in the same section, the sensors used to localize the vehicle are described. Afterwards, the pose estimation module is illustrated, specifying how it works and how the estimation is improved. Lastly, we provide a short description of the reference MPC and our relative implementation in ROS.

In the last part of the thesis it is possible to find the experimental results obtained during the tests we carried out.

Outline

In order to help the reader to better understand how to read this work here we suggest how to proceed. For every chapter it is provided an initial relative background of the covered topic. For those who already have the relative knowledge it is possible to skip the first section of the chapter.

The structure of this thesis is the following:

- Chapter 1: some examples of driverless vehicles proposed in literature, with a special focus on the DARPA Challenge, are described. Moreover an overview of the main software architectures in unmanned vehicles is proposed.
- Chapter 2: an overview of the software architecture is described, followed by a brief description of each module developed or integrated.
- Chapter 3: the reference vehicle and sensors are described. Furthermore there is a detailed description of the simulators.
- Chapter 4: the localization module is illustrated describing the ROAMFREE library configuration and its improvement.

- Chapter 5: the application of the Model Predictive Control technique for the reference vehicle control is described.
- Chapter 6: parameter estimation and experimental results are presented and analyzed.
- Chapter 7: all the work done within this thesis is briefly summarized and some possible future extensions and improvements are presented.

Chapter 1

Driverless cars state of the art

1.1 Unmanned vehicles

From the very beginning of vehicles history, vehicular automation was one of the most pursued goals; nowadays 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 and dynamic environments as city streets remain an ambitious goal. This kind of vehicles could improve everyday life, for instance reducing the risk of accidents, or they could be used to explore difficult to reached places such as mines or planets.

Unmanned ground vehicles (UGV) come 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. In the latter category, over the years, various vehicles were used as base, 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.

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. Various approaches have been adopted in order to build and manage an UGV; in the following we present a section about

autonomous cars that belong to the unmanned vehicle category while maintaining they primary goal, i.e., transporting people.

1.2 Self-driving cars

For more than a century the automotive industry has contributed towards innovation and economic growth. In the last decade the momentum of innovation is phenomenal and the world is now at the cusp of greatest technological revolution: "self-driving" vehicles. The main focus is to keep the human being out of the vehicle control loop and to relieve him/her from the task of driving. The main components of self driving vehicles are sensors (to acquire information about vehicle surrounding), computers (to process sensor information and send warning/control signals) and actuators (responsible for lateral and longitudinal control). The basic responsibility of self driving cars include lateral and longitudinal control of vehicle along with vehicle to vehicle (V2V) and vehicle to infrastructure (V2I) communication. [55]

In 2013 the U.S. Department of Transportation's National Highway Traffic Safety Administration (NHTSA) announced a new policy concerning vehicle automation where it has been proposed a formal classification of the existing systems. NHTSA defines vehicle automation as having five levels [75]:

- **Level 0 - No-Automation:** The driver is in complete and sole control of the primary vehicle controls – brake, steering, throttle, and motive power – at all times;
- **Level 1 - Function-specific Automation:** Automation at this level involves one or more specific control functions. Examples include electronic stability control or pre-charged brakes, where the vehicle automatically assists with braking to enable the driver to regain control of the vehicle or stop faster than possible by acting alone;
- **Level 2 - Combined Function Automation:** This level involves automation of at least two primary control functions designed to work in unison to relieve the driver of control of those functions. An example of combined functions enabling a Level 2 system is adaptive cruise control in combination with lane centering;
- **Level 3 - Limited Self-Driving Automation:** Vehicles at this level of automation enable the driver to cede full control of all safety-critical functions under certain traffic or environmental conditions.

The driver is expected to be available for occasional control, but with sufficiently comfortable transition time;

- **Level 4 - Full Self-Driving Automation:** The vehicle is designed to perform all safety-critical driving functions and monitor roadway conditions for an entire trip. The driver provides destination or navigation input, but is not expected to be available for control at any time during the trip. This includes both occupied and unoccupied vehicles.

Among the Full Self-Driving cars (Level 4) benefits we can list [2]:

- Fewer accidents: humans can get distracted easily, computers do not. Moreover, reaction time is larger for humans. Also we all know driving drunk is dangerous. Drowsy driving is just as risky or even more;

- No more parking worries: most cars are parked 95% of the time. Self-driving cars would eliminate the need for much of that parking. Best of all, you would get back all that time you now spend looking for parking;

- Less traffic congestion: autonomous cars by communicating with each other and with customized infrastructures would be able to manage the traffic by themselves. Moreover if you could use an app to get a car to come pick you up anytime, you might not feel the need to buy your own;

- Lower emissions: an estimated 30% – 60% of the cars driving around a downtown area are circling for parking. That's good evidence that self-driving cars, which would not be cruising for a place to park, would lower emissions. If they also replace a good chunk of car ownership, that should further reduce emissions. Chances are that self-driving cars will also be electric, which will lower emissions even more;

- Savings: With self-driving cars, we are likely to save in several ways. Less car ownership certainly means savings: you will not have to spend on the car itself, repairs, or insurance. If your taxi or Uber comes without a driver, the ride will be cheaper. And another benefit of not having to park is that you do not have to pay to park. Plus, when fewer people own cars, they will not be as desirable to steal. And because driverless cars will be integrated and tracked in a network, it will be more challenging to steal them;

- Reduced stress: even driving enthusiasts get stressed by driving in bad traffic. Add the stress of finding parking, and it gets serious. In fact, it seems that stress from driving is even worse than we think. It can lead to adverse health effects and can even make you feel less satisfied with your job and with your life;

- Transportation for those who can not drive: self-driving cars will be a boon to the elderly, allowing them mobility and freedom far beyond what they enjoy now. They will also be great for people who can not drive because of other physical issues;

- Police and road signs reduction: autonomous cars will receive necessary communication via an electronic way;

- Higher velocity limits: due to a lower reaction time of an unmanned vehicle.

Despite the benefits brought by the use of autonomous cars, there are some technological and legal challenges that need to be overcome:

- Software reliability: depending totally from the software, self-driving cars must guarantee an high reliability rate to avoid malfunctions that can lead to accidents;

- Security: cyber-attacks on autonomous vehicles would put human lives at immediate risk in a way most other hacks do not [61];

- Radio spectrum need: to let the communication among the vehicles be possible [41];

- Sensitivity to different weather conditions: the navigation system of an autonomous car shall adapt to different meteo situations, above all in case of snow or rain because in these particular conditions sensors are not fully reliable;

- Need of creating (and maintaining) maps for self-driving cars [1];

- Road infrastructures may need changes to work properly with autonomous cars, e.g. traffic lights shall be able to communicate with cars;

- Privacy loss: car localization will always be tracked and other info will be shared with the other vehicles and with road infrastructures;

- New laws and government acts to rule responsibility in case of accidents;
- Instinctive human resistance to handing over control to a robot, especially given fears of cyber-hacking [31].

1.3 Existing projects

In this section we describe some existing projects from the very first unmanned vehicles to the modern self-driving cars. A specific subsection is dedicated to the DARPA Grand Challenge, a prize competition for American autonomous vehicles [18].

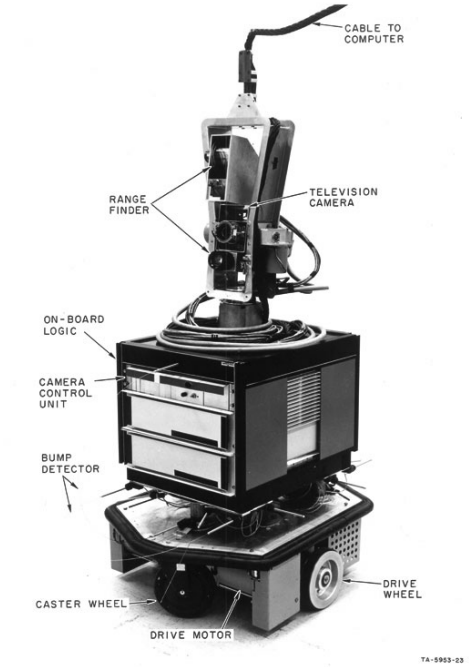
1.3.1 First unmanned vehicles

In the following we present the first unmanned vehicles projects; the first ones are more oriented to robots, while more recent ones are about real self-driving cars.

Shakey [45] (SRI International, United States, 1966-1972, Figure 1.1a) 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 [42] [43] (Stanford University AI Lab, United States, 1973-1981, Figure 1.1b) 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 [71] [42] (DARPA's Strategic Computing, United States, 1985-1988, Figure 1.1c) 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



(a)

(b)

(c)

(d)

Figure 1.1: *Some of the first prototypes of unmanned vehicles: Shakey (a), Stanford Cart (b), DARPA ALV (c) and Ground Surveillance Robot (d).*

(a) (b)

Figure 1.2: *Two prototypes of autonomous car: VaMP (a) and ARGO (b).*

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 [52] [26] (Naval Ocean Systems Center, United States, 1985-1986, Figure 1.1d) 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 [59] [20] [60] (Bundeswehr University of Munich, Germany, 1993-1995, Figure 1.2a) 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 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 [35] (University of Parma, Italy, 1998, Figure 1.2b) was a Lancia Thema passenger car equipped with a stereoscopic vision system consisting of two synchronized cameras able to acquire pairs of gray 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 on the Italian highways [3], 94% of the total trip was performed autonomously.

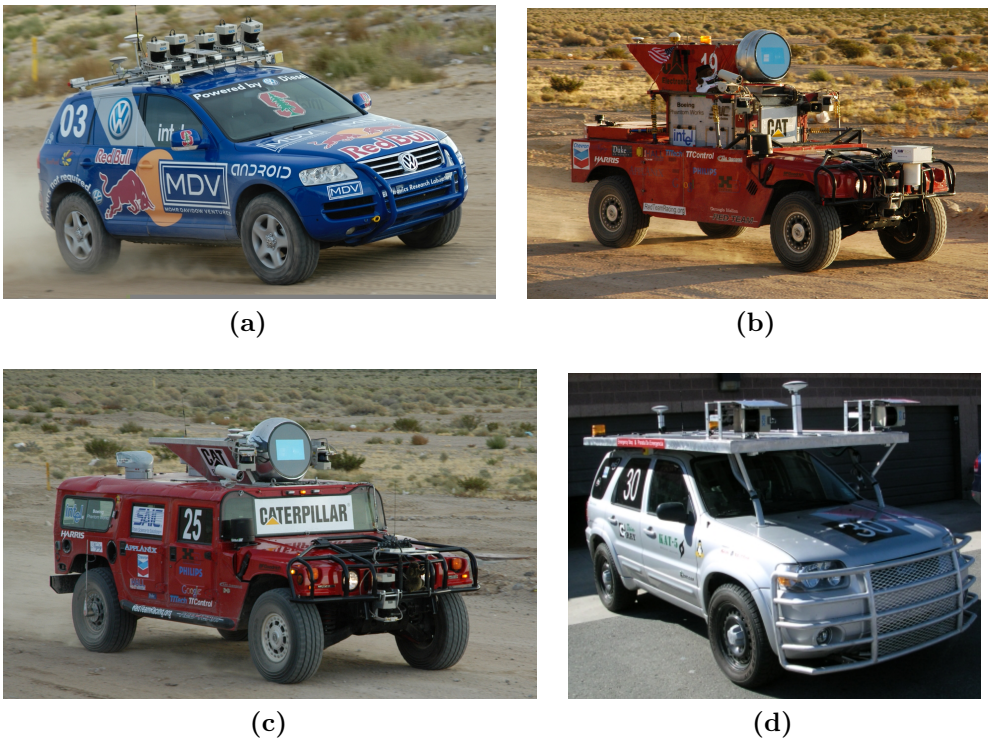
1.3.2 DARPA Grand Challenge

The DARPA Grand Challenge is a prize competition for American autonomous vehicles, funded by the Defense Advanced Research Projects Agency, the most prominent research organization of the US. Congress has authorized DARPA to award cash prizes to further DARPA's mission to sponsor revolutionary, high-payoff research that bridges the gap between fundamental discoveries and military use. The initial DARPA Grand Challenge was created to spur the development of technologies needed to create the first fully autonomous ground vehicles capable of completing a substantial off-road course within a limited time. The third event, the DARPA Urban Challenge extended the initial Challenge to autonomous operation in a mock urban environment.

In the following paragraphs we list the most noteworthy projects that participated in the 2005 and 2007 editions.

Stanley [9] (Stanford University, United States, 2005, Figure 1.3a) is an autonomous car that participated and won the second edition of the DARPA Grand Challenge in 2005. Stanley is based on a diesel-powered Volkswagen Touareg R5 with a custom interface that enables direct electronic actuation of both 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 of the 2005 DARPA Grand Challenge in 6 hours and 54 minutes.

Sandstorm [6] (Carnegie Mellon University, United States, 2004-2005, Figure 1.3b) 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 with drive-by-wire modifications control 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.



(a) (b) (c) (d)

Figure 1.3: *The four best participants of the DARPA Grand Challenge 2005 edition: Stanley(a), Sandstorm(b), H1ghlander(c) and Kat-5 (d).*

H1ghlander [6] (Carnegie Mellon University, United States, 2004-2005, Figure 1.3c) Created by Red Team, the same as the previous one, it is a heavily modified 1999 HUMMER H1. It competed in the 2005 DARPA Grand Challenge. The sensors used by H1ghlander include LIDAR laser-ranging units, one steerable LIDAR (in the globe on top), GPS and an inertial navigation system. H1ghlander completed the race in 7 hours and 14 minutes, placing 3rd out of the five vehicles to complete the 132 mile (212 km) course. It was preceded, in second place, by Sandstorm, its sister vehicle with which shares the software and the sensors.

Kat-5 [58] (GrayMatter, Inc., United States, 2005, Figure 1.3d) 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

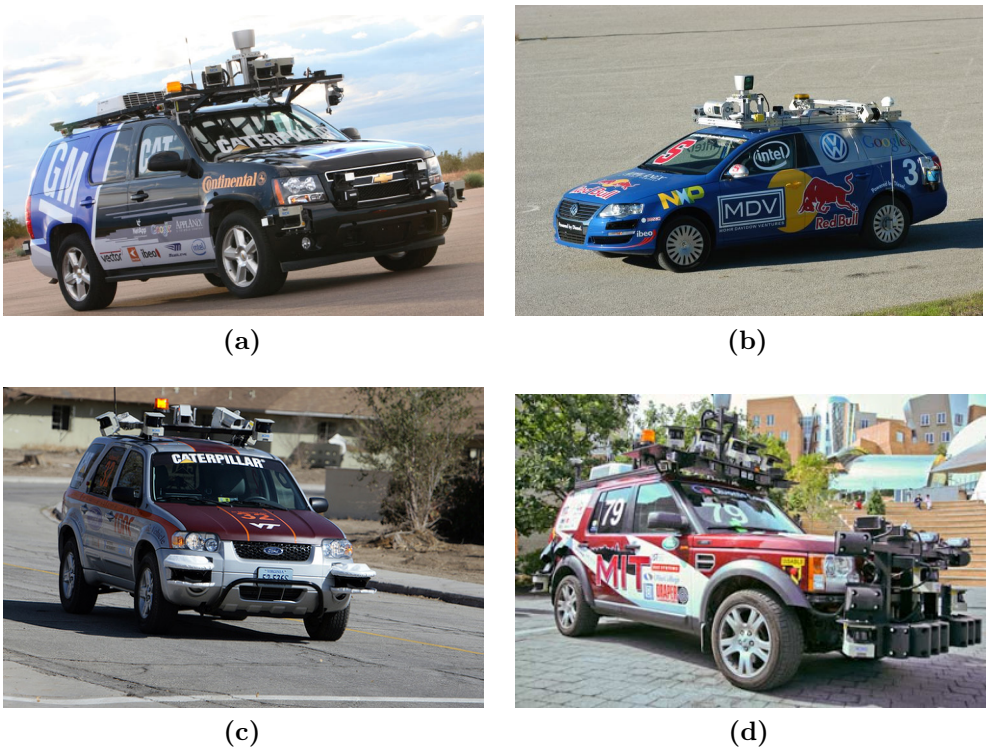
unit to create a picture of the surrounding environment and drive-by-wire systems to control the vehicle.

Boss [68] (Carnegie Mellon University, United States, 2007, Figure 1.4a) is a heavily modified Chevrolet Tahoe. Boss is equipped with more than a dozen lasers, cameras and radars to perceive the world. It completed the 2007 edition race of the DARPA Urban Challenge in 4 hours and 10 minutes winning it with an average of approximately 23 km/h throughout the course.

Junior [69] (Stanford University, United States, 2007, Figure 1.4b) is a modified 2006 Volkswagen Passat Wagon equipped with five different laser measurement systems, a multi-radar assembly, and a multi-signal inertial navigation system; specifically Junior uses the Applanix POS LV 420 Navigation system for state estimation (location, orientation, velocities). The POS LV 420 system comes with three GPS antennae, mounted on the roof of the vehicle, a high quality Inertial Measurement Unit, mounted in the trunk over the rear axle, and an external wheel encoder, attached to the left rear wheel. For external sensing, Junior features a Velodyne HD LIDAR laser range finder. Additional range sensing is achieved through two IBEO Alasca XT sensors, mounted on the front bumper of the vehicle. Junior also uses an omni-directional Ladybug camera, manufactured by PointGray. It participated to the 2007 DARPA Urban Challenge and finished with a time of 74 hours and 29 minutes, placing second.

Odin [22] (Virginia Tech University, United States, 2007, Figure 1.4c) is a modified 2005 Hybrid Ford Escape. The largest portion of Odin's detection coverage is provided by a coordinated pair of IBEO Alasca XT Fusion laser range finders. This system comprises two four-plane, multireturn range finders and a single external control unit (ECU) that covers a 260-deg field of view. A single IBEO Alasca A0 unit with a field of view of 150 deg is used to detect approaching vehicles behind Odin and navigate in reverse. For short-range road and obstacle detection, two additional SICK LMS 291 laser range finders are angled downward on the front corners of the roof rack. Two side-mounted SICK LMS 291 single-plane range finders are used to cover the side blind spots of the vehicle and ensure 360-deg coverage. In combination, the cameras cover a 90-deg horizontal field of view in front of Odin. It was able to complete the 96 kilometers urban area course in 4 hours and 36 minutes, placing third.

Talos [40] (MIT, United States, 2007, Figure 1.4d) is a modified



(a) (b)

(c) (d)

Figure 1.4: *The four best participants of the DARPA Urban Challenge 2007 edition: Boss(a), Junior(b), Odin(c) and Talos (d).*

Land Rover LR3. It perceives the state of the environment using LIDAR range finders together with vision sensors and its own motion through the combination of GPS and an IMU. The LIDAR suite includes "push-broom" sensors for analyzing the neighboring terrain along with a complementary 360 degree system for obstacle detection and tracking. Additionally, these scanners were augmented with vertically-oriented LIDARs that the vehicle used to fuse the push-broom data into a 3D mesh used to identify drivable terrain. Talos completed the race in approximately 6 hours, placing 4th.

1.3.3 Modern self-driving cars

After 40 years of research, the technology is close to leave the prototype stage. Late in 2007, six autonomous vehicles successfully completed a 90 kilometer test course of simulated urban traffic. In 2012, Vislab, Italy has tested their solar powered autonomous car for 13000 Km from Milan, Italy to Shanghai, China [15]. This journey completed successfully in three months and acquired few Terabytes of data for further analysis and

processing.

Toyota has recently presented Toyota Lexus model to help drivers observe and respond to the vehicles surroundings. Fujitsu Japan has revealed ROPITS (Robot for Personal Intelligent Transportation System) [65] with GPS, Laser and Gyroscope along with full navigation support. The passengers are required to enter the destination and vehicle will drive them autonomously and safely. For the first time ever, a joystick replacing steering wheels, which can be operated in case of any emergency, has also showcased. This could be a boon for the elderly persons. Infiniti Q50 of Nissan [32] (today, drive by wire; tomorrow drive by robot) uses sensor to get steering wheel angle, low speed maneuvers or high speed stability, lane departure system with windshield mounted camera [55].

In the following we present some of the self-driving cars that are still in development phase in order to make the reader understand the state of the art technologies in this area:

AnnieWAY [67] (Figure 1.5a) was developed in Germany at the Karlsruhe Institute of Technology (KIT) since 2007. AnnieWAY is equipped with several modifications over a VW Passat base vehicle; electronically controllable actuators for acceleration, brakes, transmission and steering have been added, each of which can be enabled individually. A CAN gateway allows sending requests to these actuators and receiving selected signals like wheel speeds and status information. It additionally implements a low-level safety disengagement of autonomous functions in case the driver needs to interfere. Several complementary sensors are available for cognition: a high definition laser scanner (Velodyne HDL64-E) delivers several all around 3D point clouds per second and multiple cameras can be mounted in different configurations on a roof rack. A third source of environmental information is the vehicle stock radar, which can be used to supplement the communication-based information about other vehicles. Self localization of the ego-vehicle is realized by a combined inertial- and satellite-based navigation system (OXTS RT 3003), which can optionally be augmented by reference stations (differential GPS). Additionally, it is equipped with a vehicle to vehicle communication system based on the wireless 802.11p standard that allows communication till a 800m distance. AnnieWAY participated in the 2007 DARPA Urban Challenge without reaching the end because of a software problem while in 2011 participated and won the Grand Cooperative Driving Challenge (GCDC), a competition that has the aim to deliver the most effective cooperative vehicle-infrastructure system in predetermined traffic scenario. This car has been developed within the KITTY project, a collaboration between

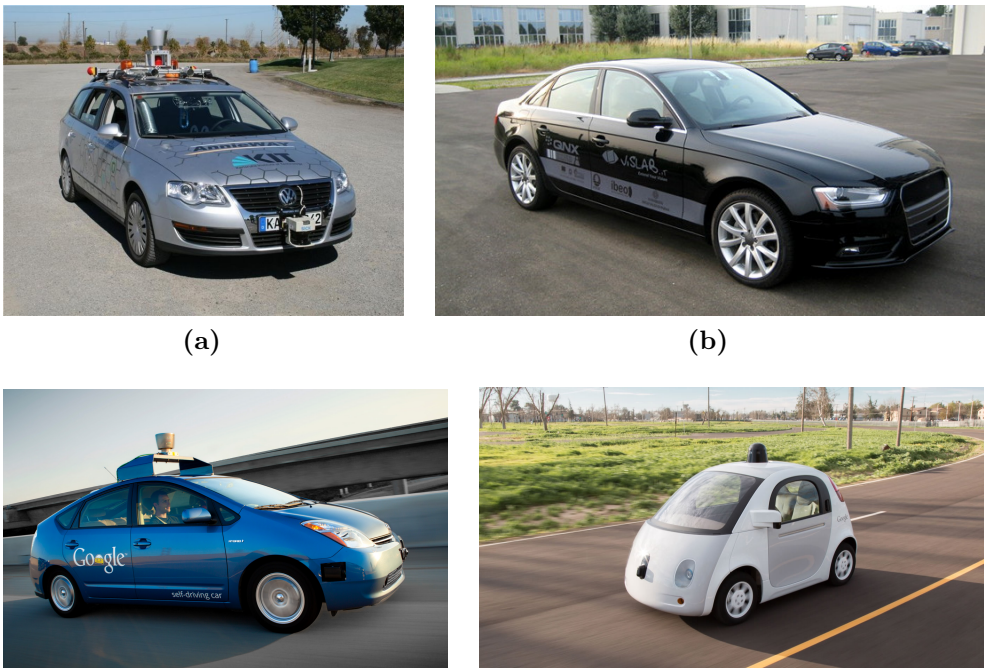


Figure 1.5: *Some models of modern self-driving cars: AnnieWAY(a), DEEVA(b), Toyota Prius Google Car (c) and the Google Car model customized by Google.*

Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago.

DEEVA [25] (Figure 1.5b) has been developed in Italy at the Artificial Vision and Intelligent Systems Laboratory (VisLab) since 2015. It is equipped with fully integrated sensors (more than 20 cameras, 4 lasers, GPS, IMU); the vehicle is able to cover a very detailed 360°view of the surroundings. The use of a technology based on artificial vision allows to achieve two main objectives, therefore making it possible to consider this concept car as very close to a final product: low cost and high integration level. DEEVA is heavily based on the VisLab's 3DV stereo-vision technology, which VisLab also provides to third parties to power their sensing systems and robots. This vehicle follows and improves the BRAiVE vehicle project: presented by VisLab in 2009 at the IEEE Intelligent Vehicles Symp in Xi'An (China), in July 2013 BRAiVE drove in a totally autonomous mode -with no driver- in an urban environment together with real traffic completing the first auto-driving test of the kind

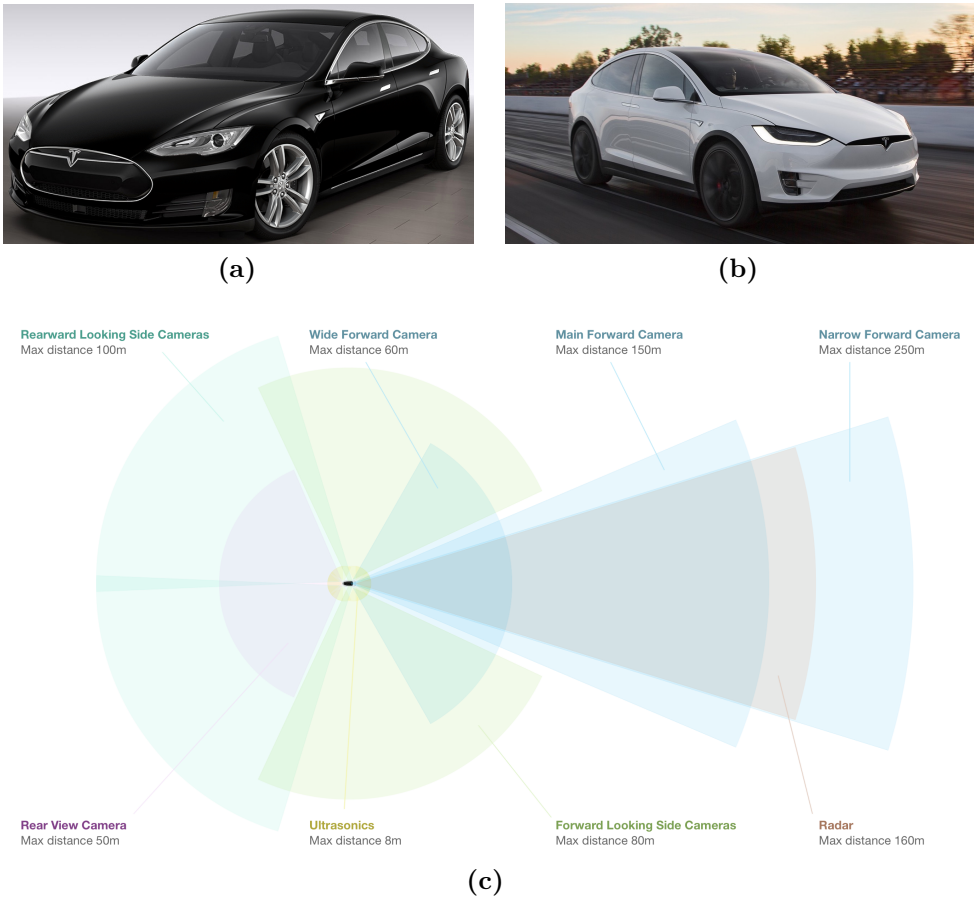


Figure 1.6: *The two Tesla models, model S and model X, and a representation of the Advanced Sensor Coverage of every Tesla model.*

in the world.

Google car [30] is a project developed in the United States at Google. Inc. since 2009. This project is probably the most known worldwide among the ones regarding self-driving cars. The project team has equipped a number of different types of cars with the same self-driving equipment, including the Toyota Prius (Figure 1.5c), Audi TT, and Lexus RX450h; Google has also developed its own custom vehicle (Figure 1.5d), which is assembled by Roush Enterprises. The equipment mounted is composed by the following sensors: a Velodyne HDL-64E scanning LIDAR, 3 RADAR in front of the car, a radar on the back bumper, a camera next to the rear-view mirror, an encoder mounted on the rear wheels, a GPS and

an IMU. Google cars have self-driven more than 2 million miles and are currently out on the streets of Mountain View, Austin, Kirkland and Metro Phoenix.

TESLA Autopilot [70] All Tesla vehicles have the hardware needed for full self-driving capability at a safety level substantially greater than that of a human driver, according to Tesla. Eight surround cameras provide 360 degrees of visibility around the car at up to 250 meters of range. Twelve updated ultrasonic sensors complement this vision, allowing for detection of both hard and soft objects. A forward-facing radar with enhanced processing provides additional data about the world on a redundant wavelength that is able to see through heavy rain, fog, dust and even the car ahead. A quite interesting point is made by Tesla's Over-the-Air updates that is considered one of the best example yet of the Internet of Things.

Looking at the prototypes listed above it is possible to note that in the origins 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 competitions, like the DARPA Grand Challenge that offers a prize in millions of dollars, still exist, also low cost prototype with complex functionalities are developed.

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.

1.3.4 Software architectures

While all autonomous vehicles aim at the same goals, they adopt different kind of architectures. In the following we list three relevant examples of software architectures from the 2007 DARPA Urban Challenge.

Annieway [53] Annieway's software architecture consists mainly of four modules. As shown in Figure 1.7 the first one is the perception module. It analyses all the sensor data, classifies all seen objects and maps the environment. This data, coming from the sensors, is sent to the planner where it is integrated in a global high-level scene description. Based on this description the planner analyses the current situation and chooses an appropriate behavior. This behavior is then executed and generates a trajectory which is passed to the next module, the low-level-collision-avoidance. Since the trajectory is generated based on abstract information, it has to be checked for drivability by taking into account the overall

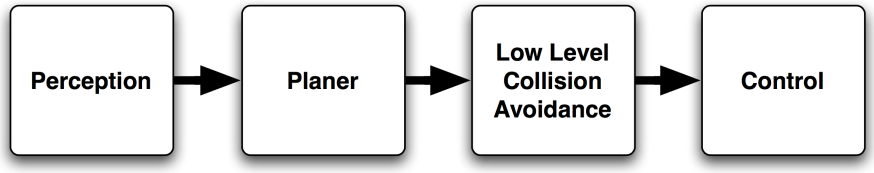


Figure 1.7: *Annieway's software architecture.*

environment. If there is some probability that the car will hit an obstacle, the collision avoidance module plans an alternative trajectory. At the last stage the control module drives the car according to the trajectory.

Boss [10] The software system that controls Boss, the Carnegie Mellon University vehicle, is divided into four primary subsystems: Perception, Mission Planning, Behavioral Executive, and Motion Planning. Their dominant communication paths are shown in Figure 1.8, and they use a message-passing protocol according to the anonymous publish-subscribe [63] pattern. The Perception subsystem processes sensor data from the vehicle and produces a collection of semantically-rich data elements such as the current pose of the robot, the geometry of the road network, and the location and nature of various obstacles such as roadblocks and other vehicles. The Mission Planning subsystem computes the fastest route to reach the next checkpoint from any point in the road network; it publishes a Value Function that maps each waypoint in the road network to the estimated time to reach the next checkpoint. The Behavioral Executive follows the Value Function from the robot's current position, generating a sequence of incremental Motion Goals for the Motion Planning subsystem to execute. The Motion Planning subsystem is responsible for the safe, timely execution of the incremental goals issued by the Behavioral Executive; it publishes its progress on the current goal, used by the Behavioral Executive to cue the selection and publication of subsequent or alternate goals as appropriate.

Junior [77] Junior's software architecture is designed as a data driven pipeline in which individual modules process information asynchronously; each module communicates with other modules via an anonymous publish/-subscribe message passing protocol as for Boss. The software is roughly organized into five groups of modules (shown in Figure 1.9):

- **Sensor interfaces** manage communication with the vehicle and

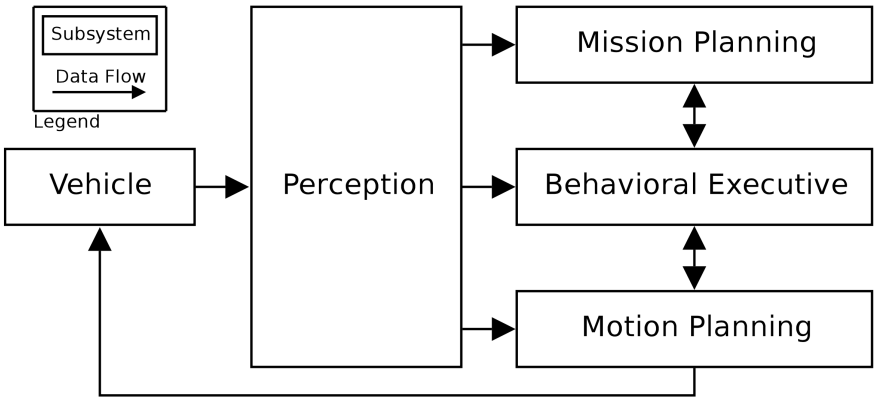


Figure 1.8: *Boss' software architecture.*

individual sensors, and make resulting sensor data available to the rest of the software modules;

- **Perception modules** segment the environment data into moving vehicles and static obstacles. They also provide precise localization of the vehicle relative to the digital map of the environment;

- **Navigation modules** determine the behavior of the vehicle. The navigation group consists of a number of motion planners, plus a hierarchical finite state machine for invoking different robot behaviors and preventing deadlocks;

- **Drive-by-wire interface** controls the vehicle by passing back commands through the drive-by-wire interface. This module enables software control of throttle, brake, steering, gear shifting, turn signals, and emergency brake;

- **Global services** are a number of system level modules provide logging, time stamping, message passing support, and watchdog functions to keep the software running reliably.

As we can see, in literature there exists different kind of architectures, less or more complex, that share a common base and cyclical schema represented by a first part of perception of the environment, followed by a planning phase and then a stage where the robot carries out the actions previously planned.

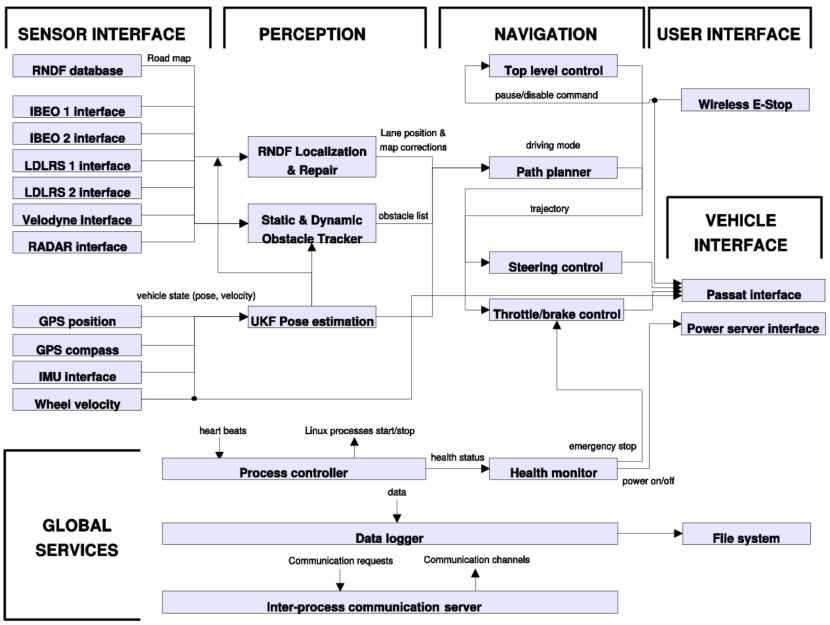


Figure 1.9: *Junior's software architecture.*

Taking the cue from these examples we decided to implement our own architecture based on the Sense-Plan-Act paradigm. In the next chapter we present a general overview of the designed architecture, while in the following chapters we focus the attention specifically on each module.

Chapter 2

Software architecture

This chapter describes the software architecture designed and developed within this thesis from an high level prospective; in the following, we provide a chapter for each main component: simulation (Chapter 3), localization (Chapter 4) and Model Predictive Control (Chapter 5). The aim of this chapter is to give a general overview of the architecture and how the communication between all its components is implemented.

2.1 Relevant background

Writing software for robotics purposes is challenging because different types of robot can have extremely diverse hardware, making code reuse hardly possible. Moreover, the modules developed must implement a deep stack starting from driver-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), developed by the Stanford Artificial Intelligence Laboratory and by Willow Garage, is a flexible framework for writing robot software. It is a collection of tools, libraries and conventions that aim to simplify the task of creating complex and robust robot behaviors across a wide variety of robotic platforms.

Creating truly robust, general-purpose robot software is hard. From the robot's perspective, problems that seem trivial to humans often vary wildly between instances of tasks and environments. Dealing with these

variations is so hard that no single individual, laboratory, or institution can hope to do it on its own. As a result, ROS was built from the ground up to encourage collaborative robotics software development. For example, one laboratory might have experts in mapping indoor environments, and could contribute a world-class system for producing maps. Another group might have experts at using maps to navigate, and yet another group might have discovered a computer vision approach that works well for recognizing small objects in clutter. ROS was designed specifically for groups like these to collaborate and build upon each other's work. [4]

ROS was designed to be as distributed and modular as possible, so that you can pick and choose which parts are useful for you and which parts you'd rather implement yourself. The distributed nature of ROS also fosters a large community of user-contributed packages that add a lot of value on top of the core ROS system. At last count there were over 3,000 packages in the ROS ecosystem, and that is only the ROS packages that people have taken the time to announce to the public. These packages range in fidelity, covering everything from proof-of-concept implementations of new algorithms to industrial-quality drivers and capabilities. The ROS user community builds on top of a common infrastructure to provide an integration point that offers access to hardware drivers, generic robot capabilities, development tools, useful external libraries, and more.

Over the past several years, ROS has grown to include a large community of users worldwide. Historically, the majority of the users were in research labs, but increasingly we are seeing adoption in the commercial sector, particularly in industrial and service robotics. The ROS community is very active. According to our metrics, at the time of writing, the ROS community has over 1,500 participants on the ROS-users mailing list, more than 3,300 users on the collaborative documentation wiki, and some 5,700 users on the community-driven ROS Answers Q&A website. The wiki has more than 22,000 wiki pages and over 30 wiki page edits per day. The Q&A website has 13,000 questions asked to date, with a 70% percent answer rate. [78] Figure 2.1 shows how the ROS vibrant community is spread all over the world.

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

Figure 2.1: *The costantly growing ROS vibrant community.*

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 (Figure 2.2), 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

Figure 2.2: *The Publisher/Subscriber protocol.*

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 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. Figure 2.3 shows the performed steps to set up the communication.

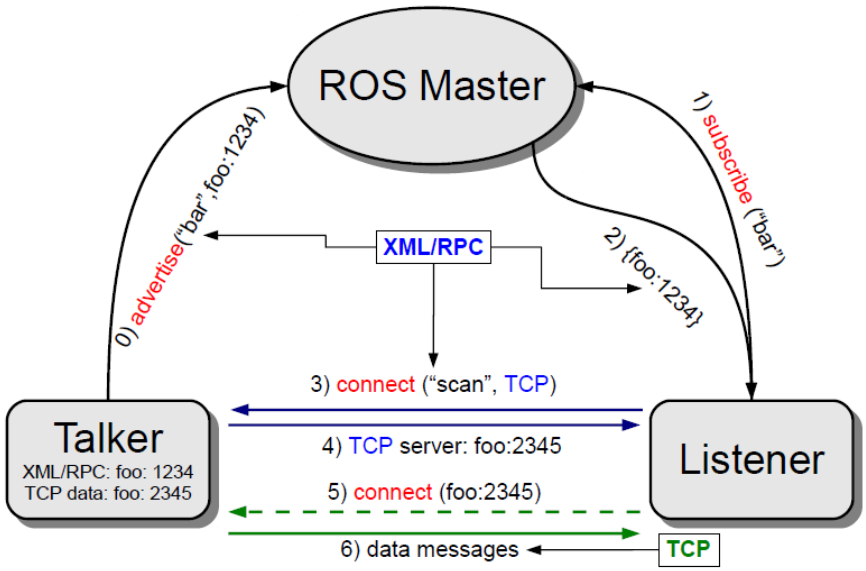


Figure 2.3: *Set up of the ROS communications.*

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

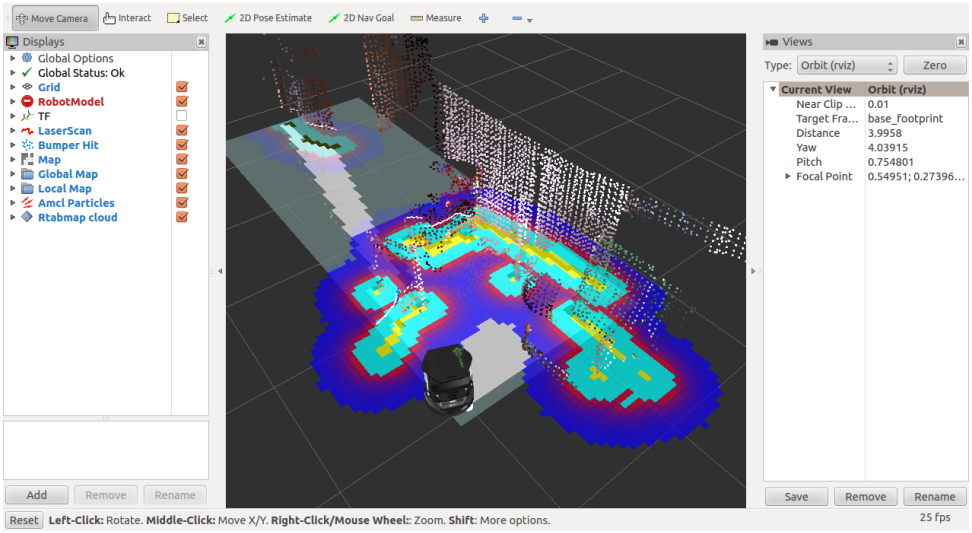


Figure 2.4: *A typical rviz window.*

have simple functionalities, e.g., showing all the messages published on a topic, while others are more complex. For example, rviz [27] (Figure 2.4) 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.

2.2 Architectural overview

In order to develop a flexible and modular architecture and to take advantage of some existing simulators, we decided to use ROS, which is up to now considered the standard de-facto for robot applications. Different reasons led us to this choice: first of all the possibility to design and develop different independent modules to handle all the single parts of the architecture. This allows us to test and debug separately all the system functionalities and to add on demand the modules we need. In addition, the open source logic adopted by ROS allows the developers to share their works in a official repository. This appeared to be useful for our work since we reused some of them. Another reason that drove our choice was the integration with simulators. Since simulation is one of the main phases of a robot development, we needed a system able to easily interfacing with one of them.

The software architecture can be divided into three main parts: the vehicle (Chapter 3), the localization (Chapter 4) and the Model Predictive

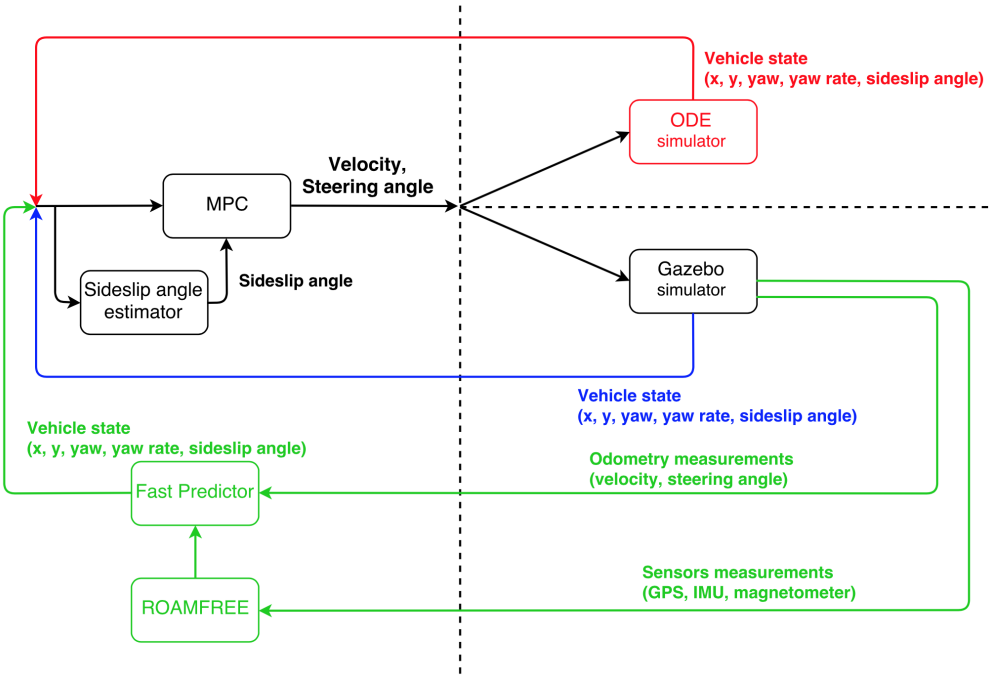


Figure 2.5: *General software architecture.*

Control (Chapter 5). More specifically this division can be seen as what has to run in the vehicle and the vehicle itself. Therefore the first part, localization and MPC modules, is independent from the second one, vehicle simulator, since it works regardless of the associated platform. Regarding the vehicle simulation it is possible to define three simulation profiles:

- **ideal simulation**: the vehicle is described as a single-track model (Section 3.3.1) and a ROS node has been developed to simulate its behavior based on *odeint*. Localization is provided by the simulation itself;
- **Gazebo simulation**: the ideal simulator is substituted by Gazebo (Section 3.3.2) and the localization is performed using the position provided by the simulator itself;
- **Gazebo simulation with ROAMFREE**: the simulation is performed by Gazebo while the localization is estimated by ROAMFREE library (Section 4.1).

In Figure 2.5 it is possible to see an high level representation of the control architecture with the three profiles. We decided to put the localization

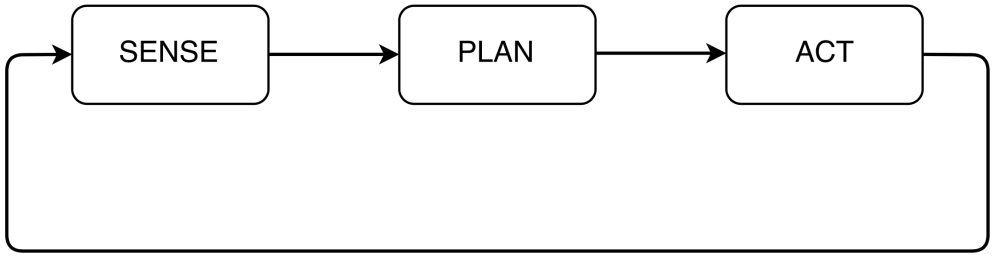


Figure 2.6: *Sense-Plan-Act paradigm.*

module performed with ROAMFREE in the left half of the schema since this module is independent from the vehicle and it has to run close to the MPC.

2.2.1 Main modules

In order to describe the main modules of the architecture, we followed the hierarchical paradigm Sense-Plan-Act (Figure 2.6) [62].

1. **Sense** - the robot needs the ability to sense important things about its environment, like the presence of obstacles or navigation aids;
2. **Plan** - the robot needs to take the sensed data and figure out how to respond appropriately to it, based on a preexisting strategy;
3. **Act** - finally, the robot must actually act to carry out the actions that the plan calls for.

The core part of the sense module consists in the localization node (Chapter 4), which is based on ROAMFREE. This framework provides pose tracking combining the information coming from an arbitrary number of sensors. In the current configuration the localization module estimates the robot poses exploiting a GPS, a magnetometer and an Inertial Measurement Unit, which includes both a gyroscope and an accelerometer. Since ROAMFREE provides vehicle position and orientation and since for the planning module we needed also information about the sideslip angle of the vehicle, to complete the perception part we implemented a sideslip angle estimator (Section 5.5).

The planning part of the architecture is represented by the MPC node (Chapter 5). As its name suggests, this node implements a Model Predictive Control. The goal to reach is given through a configuration file which is

read when the node is launched. It cyclically reads the current position of the vehicle and elaborates the commands to be sent to the actuators.

The act section can be represented by the simulators. For the ideal one it is the integration of the differential equations that describe the single-track model (Section 3.3.1) which represent the vehicle. About Gazebo it is the plugin developed to control the gas pedal and steer (Section 3.4.4).

2.2.2 ROS topics and messages

For the development of the architecture, even though ROS provides different types of messages, it has been necessary to create several custom messages to satisfy our needs. Here we provide a list of them specifying the contest in which they are used:

- **vehicle_state**: it includes information about the position of the vehicle, i.e., the Cartesian coordinates, the sideslip angle, the yaw angle and its rate (Section 3.3.1);
- **mpc_velocity**: it includes the output of the MPC controller, i.e., the value of the velocity over x-axis and the value of the velocity over y-axis (Section 5.2);
- **speed_steer**: it includes the values of the desired velocity and steering angle to be assigned to the vehicle (Section 5.3);
- **pointP**: this message includes the vehicle current state plus the Cartesian coordinates of the vehicles plus the Cartesian coordinates of the relative linearization (Section 5.3).

The values used in these messages are described as **float64**, that is the ROS representation of the C++ *double* [56]. Furthermore, regarding the message types offered by ROS we used the **std_msgs::Float64** to send single information about the yaw angle, the yaw rate and the sideslip angle. In Figure 2.7 it is possible to see in detail the composition of the custom messages.

The ROS topics used within the project, shown in Figure 2.8, are the following:

- **/vehicle_state**: it is used to publish the current state of the vehicle;
- **/mpc_vel**: this topic is used by the node MPC to publish its own output;

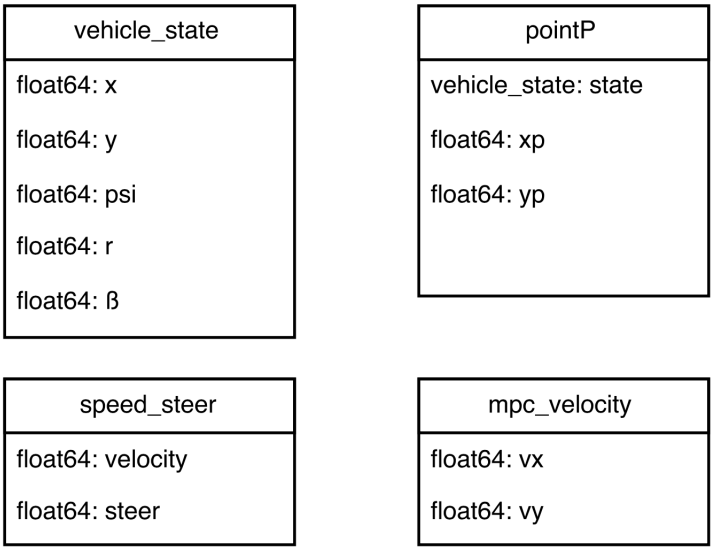


Figure 2.7: *Custom messages.*

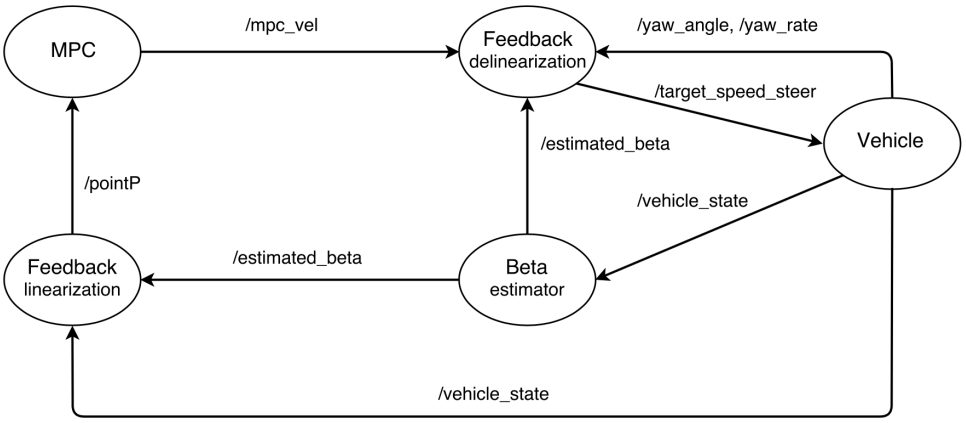


Figure 2.8: *Topics structure.*

- **/target_speed_steer**: on this topic the node Feedback delinearization publishes the resulting velocity and steering angle;
- **/estimated_beta**: on this topic it is possible to read the value of the estimated sideslip angle;
- **/yaw_angle**: topic on which it is possible to read information about the yaw angle of the vehicle;
- **/yaw_rate**: topic on which it is possible to read information about the yaw rate of the vehicle, i.e., the angular velocity with respect to the z-axis;
- **/pointP**: on this topic it is published the result of the feedback linearization.

While integrating Gazebo, it has been necessary to change the topic structure to take advantage of the preexisting simulator API. In fact there were already some topics on which to publish messages to control the vehicle:

- **/polaris/gas_pedal**: topic used to send command for the gas pedal;
- **/polaris/brake_pedal**: topic used to control the brake pedal;
- **/polaris/hand_wheel**: topic used to set the value of the hand wheel of the vehicle.

Chapter 3

Thesis reference robot platform

In this chapter we present the reference vehicle, the used sensors and their simulations. For our work we decided to adopt as vehicle the *"Polaris Ranger XP 900"* (Figure 3.1) [54]. Our choice was driven by the fact this platform has been recently used in the DARPA Robotics Challenge in 2015 [19] where one of the 8 tasks to be performed by the humanoid robots was to drive a Polaris Ranger XP 900 EPS" to a specified location. Thanks to this, a model of this vehicle was already implemented in the repository of the chosen simulation environment and we could take advantage of it.

In the relevant background we describe some works related to the same vehicle category and we give an overview of the main robotics simulation environments with a special focus on the chosen one. Among the implemented simulation profiles, we used also a mathematical model described by differential equations. The integration of these equations, needed to simulate the vehicle behavior, requires the usage of a library able to solve ordinary differential equations. For this reason we describe the used library and afterwards how it has been implemented.

3.1 Relevant background

3.1.1 Related works

Some similar projects on automated road shuttles, like golf cars and mini-buses, have already been developed. This type of vehicle typically operates at lower speeds in pedestrian environments and serves as a form of public transit.

Figure 3.1: *Polaris Ranger XP 900 EPS.*

Auro

The startup Auro says its self-driving golf cart (Figure 3.2a) will lead to autonomous shuttles for theme parks, vacation resorts, and retirement communities. The current prototypes are golf carts modified with laser scanners, radar, cameras, GPS, computers, and other components needed to actuate the golf cart actuators. Auro Robotics company is focused on the more modest goal of ferrying people on autonomous vehicle around the private grounds of universities, retirement communities, and resorts. Auro's vehicles require a detailed 3-D map of the environment where they operate, and collecting that data for a private campus and keeping it up-to-date is easier. Such environments are also less physically complex, have lower speed limits, and present fewer complicated traffic situations [66].

USAD

The Urban Shuttles Autonomously Driven (USAD) project (IRAlab, Univ. Milano - Bicocca in cooperation with Politecnico di Milano) aims at the development of vehicles capable to drive autonomously in a urban setting [76]. The specific aspects of the robotic research involved, w.r.t. extra-urban autonomous driving, is the need to localize the vehicle, despite the absence of the GPS signal and/or its not-good-enough accuracy, which implies a research focus on the perception side of the navigation. The

(a)

(b)

(c)

Figure 3.2: *Some examples of self-driving projects involving golf cars: Auro(a), USAD(b), SMART(c).*

vehicle is equipped with a GPS, two high-definition cameras to detect road markings, two single plane laser scanners installed in the front edges of the vehicle, and a third laser scanner in a central position with less field of view but able to detect four planes (Figure 3.2b).

SMART

The Singapore-MIT Alliance for Research and Technology (SMART) designed a fleet of autonomous golf cars (Figure 3.2c) which were demonstrated in public trials in Singapore's Chinese and Japanese Gardens, for the purpose of raising public awareness and gaining user acceptance of autonomous vehicles. The golf cars were designed to be robust, reliable, and safe, while operating under prolonged periods of time. The overall system design foresees that any member of the public has to not only be able to easily use the system, but also not to have the option to use the system in an unintended manner.

A Yamaha YDREX3 electric golf car was used as vehicle base platform and retrofitted by the SMART team to incorporate necessary actuation, sensing, computing, and power systems along with various additional features to enhance passengers' comfort and safety. Two non-contact magnetic encoders are mounted to the rear axle of the golf car, one on each side of the drive shaft. A MicroStrain 3DM-GX3-25 Inertial Measurement Unit (IMU) is rigidly mounted to the chassis above the center of the rear axle to provide attitude and heading of the vehicle. The encoder and the IMU readings are combined to provide vehicle's odometry information in 6 degrees-of-freedom. Environmental sensing is achieved through several 2D LIDARs and a webcam. One SICK LMS 151 LIDAR is mounted facing downward in the front of the vehicle roof, where the data returned is fused with odometry readings to achieve localization as described in [23]. A second SICK LMS 151 is mounted horizontally in the lower front of the vehicle for obstacle detection. Two SICK TiM551 LIDARs are mounted at the rear corners of the golf car to provide all around obstacle detection [51].

3.1.2 Physical simulation of vehicles

A robotics simulator is used to create embedded robotics applications without depending physically on the actual machine, thus saving cost and time.

When working with an autonomous vehicle, the role of the simulator is fundamental. The robot characteristics, the typical operating environment

and the complexity of the system architecture make it challenging to develop and test all the software components directly on the real platform. Therefore, a simulation that mimics 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 it is 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 robots, these can be validated by similar ones in the simulation. Among well-known simulation benefits we can list [44]:

- Reduce costs involved in robot production;
- Diagnose source code that controls a particular resource or a mix of resources;
- Simulate various alternatives without involving physical costs;
- Robot or components can be tested before implementation;
- Simulation can be done in stages, beneficial for complex projects;
- Demonstration of a system to determine if is viable or not;
- Compatibility with a wide range of programming languages;
- Shorter delivery times.

There are various types of simulators available. Some of them are specific for robotic applications, focusing on simplicity and fast prototyping or 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 the multiple vehicle subsystems. In the following we provide a list of most common simulators with a brief description for each of them:

Gazebo [34] is an open source simulator. It was originally integrated with ROS, but now it is an independent project [73]. Gazebo can simulate, with relatively good 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.

Webots [39] (Figure 3.3a) 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. Moreover, it offers an integrated IDE to develop controllers, which can be then directly transferred to commercially available real robots.

V-Rep [49] [64] (Figure 3.3b) 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.

OpenHRP3 [29] (Figure 3.3c) 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 [57] (Figure 3.3d) 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 it uses the Bullet library for physics simulation.

Dymola [28] (Figure 3.3e) is a commercial modeling and simulation environment based on the open Modelica modeling language. It offers unique multi-engineering capabilities, which means 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 and 3D visualization.

20-Sim [12] (Figure 3.3f) 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.

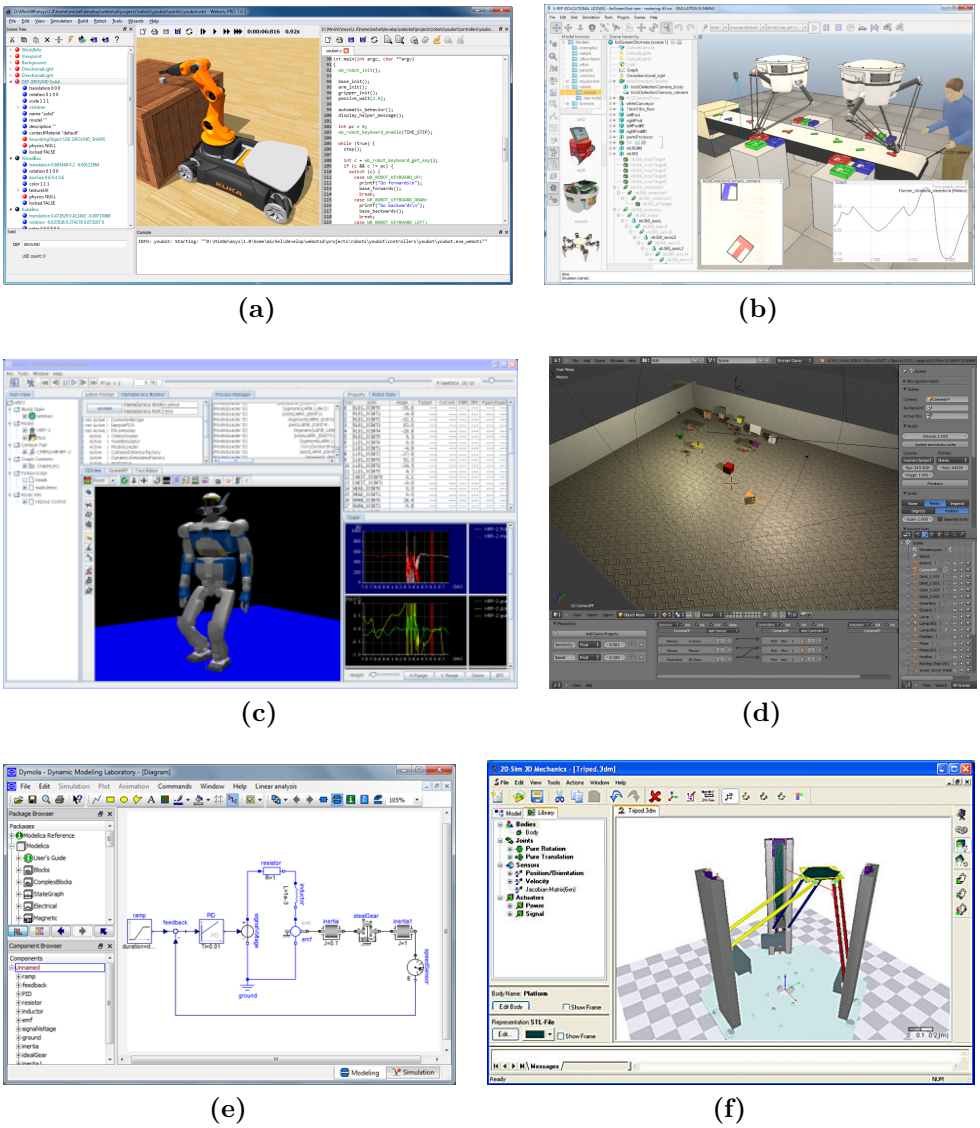


Figure 3.3: *Some examples of simulators for robotic applications: Webots(a), V-Rep(b), OpenHRP3(c), MORSE(d), Dymola(e) and 20-Sim(f).*

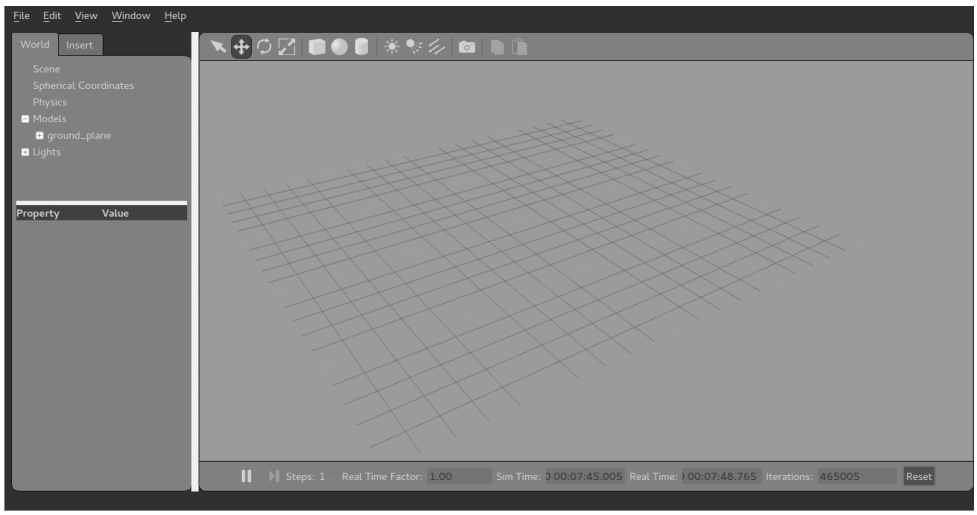


Figure 3.4: *Main window of Gazebo.*

For our work, we chose to adopt Gazebo. It has a native integration with ROS, which simplifies the interaction with our robot architecture. Moreover it is now considered the de facto standard in ROS robot simulation. Simulators like Dymola are currently 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.

3.1.3 Gazebo

Robot simulation is an essential tool in every roboticist toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, and perform regression testing using realistic scenarios. Gazebo (Figure 3.4) offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It sports a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces [24].

Its main features are:

- **Dynamics Simulation:** By default Gazebo is compiled with support for ODE as physics engine, but it also has access to other high-performance physics engines including Bullet, Simbody, and DART;
- **Advanced 3D Graphics:** Using OGRE, Gazebo provides realistic rendering of environments including high-quality lighting, shadows,

and textures. OGRE (Object-Oriented Graphics Rendering Engine) is a scene-oriented, flexible 3D engine written in C++ designed to make it easier and more intuitive for developers to produce applications utilizing hardware-accelerated 3D graphics;

- **Sensors and Noise:** Gazebo offers the possibility to generate sensor data, optionally with noise, from laser range finders, 2D/3D cameras, Kinect style sensors, contact sensors, force-torque sensors, and more;

- **Plugins:** Robot-independent Gazebo plugins for sensors, motors and dynamic reconfigurable components are available within the plugin package. It is also possible to develop custom plugins for robot, sensor, and environmental control. Plugins provide direct access to Gazebo's API;

- **Robot Models:** Many robots are provided including PR2, Pioneer2 DX, iRobot Create, and TurtleBot. It is possible to build your own using SDF and to contribute your model to Gazebo's online-database to benefits you and every other user of Gazebo;

- **TCP/IP Transport:** It allows to run simulations on remote servers, and interface to Gazebo through socket-based message passing using Google Protobufs;

- **Cloud Simulation:** There is the possibility to use CloudSim, a framework for modeling and simulation of cloud computing infrastructures and services, to run Gazebo on Amazon, Softlayer, or your own OpenStack instance;

- **Command Line Tools:** Extensive command line tools facilitate simulation introspection and control.

While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs. Typical uses of Gazebo include:

- testing robotics algorithms;

- designing robots;

- performing regression testing with realistic scenarios.

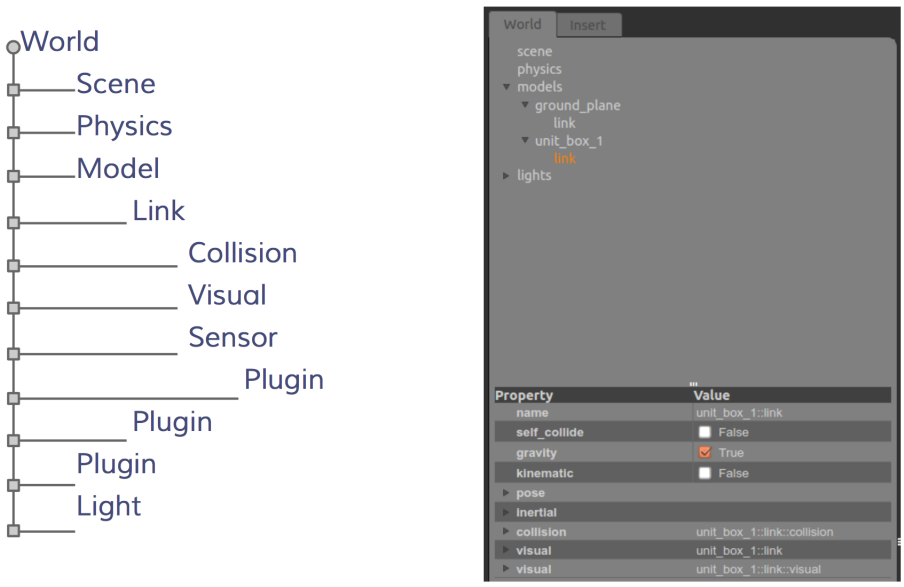


Figure 3.5: *The Gazebo element hierarchy.*

Recent Gazebo models are described using a new format called the Simulation Description Format (SDF) that was created for use in Gazebo to solve the shortcomings of the old URDF format. SDF is a complete description for everything from the world level down to the robot level. It is scalable, and makes it easy to add and modify elements. The SDF format is itself described using XML, which facilitates a simple upgrade tool to migrate old versions to new versions. It is also self-descriptive [74].

The main element of an SDF file is the *world* element, which encapsulates an entire world description: models, scene, physics, joints and plugins. The *model* element is the one used to define a complete robot. The description of a model is given by the definition of a set of *links*, i.e., a collection of *Collision* and *Visual* objects. *Collision Objects* is a geometry that defines a colliding surface, while *Visual Objects* is a geometry that defines visual representation such as meshes. Gazebo offers the possibility to modify the main elements, i.e., *World*, *Model* and *Sensor*, taking advantage of plugins. A plugin is a C++ library that is loaded by Gazebo at runtime. It has access to the relative Gazebo's API, which allows a plugin to perform a wide variety of tasks including moving objects, adding/removing objects, and accessing sensor data. In Figure 3.5 is shown the Gazebo element hierarchy.

As of Gazebo 1.9 and ROS Hydro, Gazebo no longer has any direct ROS dependencies and is now installed as an Ubuntu stand-alone package.

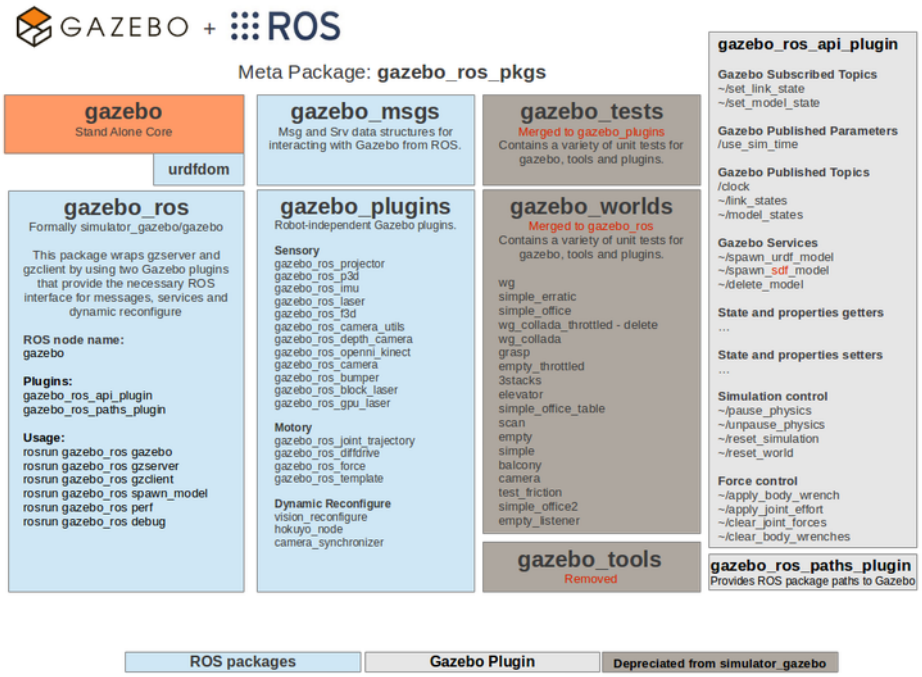


Figure 3.6: *Overview of the* ***gazebo_ros_pkgs*** *interface.*

Historically using Gazebo with ROS required a specific version of Gazebo to be built with the legacy 'simulator_gazebo' stack. To achieve ROS integration with stand-alone Gazebo, a new set of ROS packages named **gazebo_ros_pkgs** has been created to provide wrappers around the stand-alone Gazebo. They provide the necessary interfaces to simulate a robot in Gazebo using ROS messages, services and dynamic reconfigure. An overview of the **gazebo_ros_pkgs** interface is shown in the diagram in Figure 3.6 [73].

3.1.4 ODEINT

Odeint is a modern C++ library for solving Ordinary Differential Equations numerically. It is developed in a generic way using Template Metaprogramming [5] which leads to extraordinary high flexibility at top performance. The numerical algorithms are implemented independently of the underlying arithmetics. This results in an incredible applicability of the library, especially in non-standard environments. For example, odeint supports matrix types, arbitrary precision arithmetics and it can be easily run on

CUDA GPUs.

Odeint consists of four parts:

- Integrate functions
- Steppers
- Algebras and operations
- Utilities

The integrate functions are responsible to perform the iteration of the ODE. They do the step size control and they might make use of dense output. The integrate functions come along with an observer concept which lets you observe your solution during the iteration. The steppers are the main building blocks of odeint. Several steppers for different purposes exists, like:

- the classical Runge-kutta steppers
- symplectic steppers for Hamiltonian systems
- implicit methods
- stiff solvers

Algebras and operations build an abstraction layer which lets you change the way *odeint* performs basic numerical manipulations like addition, multiplication, etc. In the utility part it is possible to find functionality like resizing and copying of state types.

A stepper in odeint performs one single step. Several different stepper types exist in odeint. The simplest one is described by the Stepper concept. It has only one method do_step which performs the step. An example is the classical Runge-Kutta stepper of fourth order:

```
runge_kutta4< state_type > rk4;
rk4.do_step( system, x, t, dt );
```

Most of the stepper have a set of template parameters which tune their behavior. In the Runge-Kutta 4 example above you have explicitly to state the state type. This type is also used to store intermediate values. For the same stepper a lot of other parameters exist which are called with some default values. For example, you can explicitly state the value type

that gives the numerical values used during computations, the derivative type, the time type, the algebra and the operations. Furthermore, a policy parameter for resizing exists. In most cases the default parameters are a good choice. For exotic applications like using Boost.Units or for exotic algebras you need to specify them explicitly. The documentation of odeint shows some examples [47].

The first parameter of the do_step method specifies the ODE. It is expected that the system is either a function or functor and must be a callable object with the signature:

```
system( x , dxdt , t )
```

Another concept is the ErrorStepper. Its basic usage is very similar to the Stepper concept:

```
runge_kutta54_ck< state_type > rk54;
rk54.do_step( system , x , t , dt , xerr );
```

the main difference is that it estimates the error made during one step and stores this error in xerr.

Another concept is the ControlledStepper, which tries to perform one step. A stepper which models the ControlledStepper concept has usually some error bounds which must be fulfilled. If try_step is called the stepper will perform one step with a given step size dt and checks the error made during this step. If the error is within the desired tolerances it accepts the step and possibly increases the step size for the next evaluation. In case the error bounds are reached the step is rejected and the step size is decreased.

The fourth stepper concept is the DenseOutputStepper. A dense output stepper has two basic method do_step and calc_state. The first one performs one step. The state of the ODE is managed internally and usually this step is made adaptive, hence it is performed by a controlled stepper. The second function calculates intermediate values of the solution. Usually, the precision of the intermediate values are of the same order as the solution.

The integrate functions are responsible for integrating the ODE with a specified stepper. Hence, they do the work of calling the do_step or try_step methods for you. This is especially useful if your are using a

controlled stepper or a dense output stepper, since in this case you need some logic of calling the stepping method.

Odeint defines four different integrate methods:

- integrate_const: performs the integration with a constant step size
- integrate_adaptive: performs adaptive integration of the ODE
- integrate_times: expects a range of times where you want to obtain the solution
- integrate_n_steps: integrates exactly n steps

For all these methods it is possible to call an observer at each step size.

3.2 Reference sensors

When dealing with autonomous vehicles, positioning sensors play a big role. It is possible to divide them in two main categories; sensors that provide absolute measurements, like GPSs, and sensors that provide relative measurements, like Inertial Measurement Units.

Within our project we make use of both categories. Specifically, we use a GPS, an IMU and a magnetometer for the localization of the vehicle.

In the following we provide a description of the reference sensors and how they have been simulated. An entire chapter is dedicated to the vehicle localization (Chapter 4).

3.2.1 Global Positioning System

In order to have an absolute position useful for localization, the vehicle is equipped with a GPS: the model is a Garmin 18 LVC (Figure 3.7).

It reaches an accuracy in the order of meters, typically less than 15 (95% of cases). The sensor can be connected to the computer using an USB 2.0 cable.

The Garmin 18 LVC provides the information using the NMEA standard at a 1 Hz rate. In this standard, each message is an ASCII string with a specific format. Each sentence 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:

Figure 3.7: *GPS Garmin 18 LVC.*

$GPRMC,220516,A,5133.82,N,00042.24,W,173.8,231.8,130694, 004.2,W,*70

- UTC of position fix (**220516**)
- Data status, A = Valid position, V = NAV receiver warning (**A**)
- Latitude with its direction (N for North, S for South) (**5133.82,N**)
- Longitude with its direction (E for East, W for West) (**00042.24,W**)
- Speed over ground in knots (**173.8**)
- Course Made Good (**231.8**)
- UT date of position fix (**130694**)
- Magnetic variation degrees and its direction (E for East, W for West) (**004.2,W**)
- Checksum used for parity checking data (***70**)

3.2.2 Inertial Measurement Unit and magnetometer

The vehicle is equipped with an Xsens MTi (Figure 3.8), which is a miniature Inertial Measurement Unit (IMU) with integrated 3D magnetometers (i.e., 3D compass). The IMU is composed by accelerometers and gyroscopes, and it can calculate roll, pitch and yaw in real time, as well as outputting calibrated 3D linear acceleration and rate of turn. The magnetometer

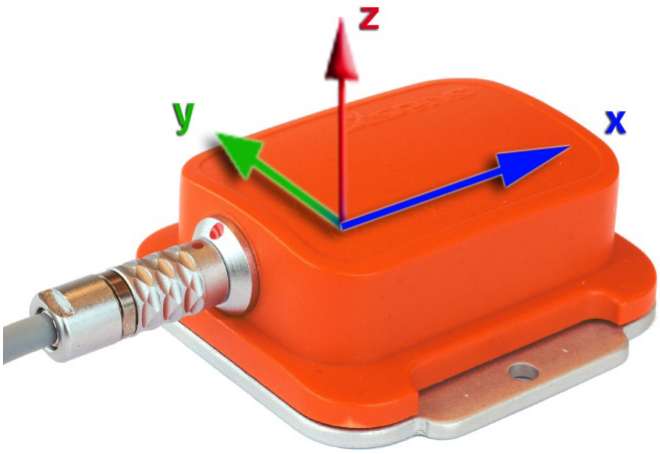


Figure 3.8: *IMU Xsens MTi.*

provides 3D earth-magnetic field data. The sensor provides measurements up to a 50 Hz frequency.

All calibrated sensor readings (accelerations, rate of turn, earth magnetic field) are in the right handed coordinate system as defined in Figure 3.8. 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.

The MTi can be directly connected with the computer using an USB 2.0 cable. The specification associated with measurements are listed in Table 3.1.

3.3 Thesis vehicle simulators

In this section we describe how the vehicle has been simulated in this thesis. Following the typical procedure, the simulation process went from an ideal situation, described in the following first section, to a realistic simulation using Gazebo, described in the following section.

	rate of turn	acceleration	magnetic field
unit	$[deg/s]$	$[m/s^2]$	$[mGauss]$
Dimension	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/$\sqrt{Hz}$]	0.0513	0.002	0.5 (1σ)
Allignment error [deg]	0.1	0.1	0.1
Bandwidth [Hz]	40	30	10
A/D resolution [bits]	16	16	16

Table 3.1: Calibrated data performance specification.

3.3.1 ODE based simulator

The first step necessary for the simulation is the realization of a model which describes the vehicle and its properties. The vehicle studied in this work has a rear-wheel drive layout with the steering working on the front wheels, so it would be best depicted by a car-like or four wheels model, but this would be computationally too expensive. For this reason a single-track or bicycle model has been adopted. In literature there are several examples of linear single-track models, but these neither include a description of the drive-train, nor they allow the representation of the vehicle behavior at larger steering angles or with higher lateral accelerations [8]. For these reasons a nonlinear single-track model has been adopted to describe the dynamics of the vehicle, which has already been proved successful in several texts of vehicle dynamics, and which is reported in Figure 5.9.

With reference to that figure the main parameters of the single-track model simulated in this work are:

- ψ: yaw angle;
- $\dot{\psi}$: yaw-rate, which is the angular velocity around the vertical axis;
- v: velocity of the vehicle;
- β: slip angle of the vehicle, which is the angle between the longitudinal axis and the velocity;
- δ: steering angle;
- α_f and α_r: front and rear tire slip angles;

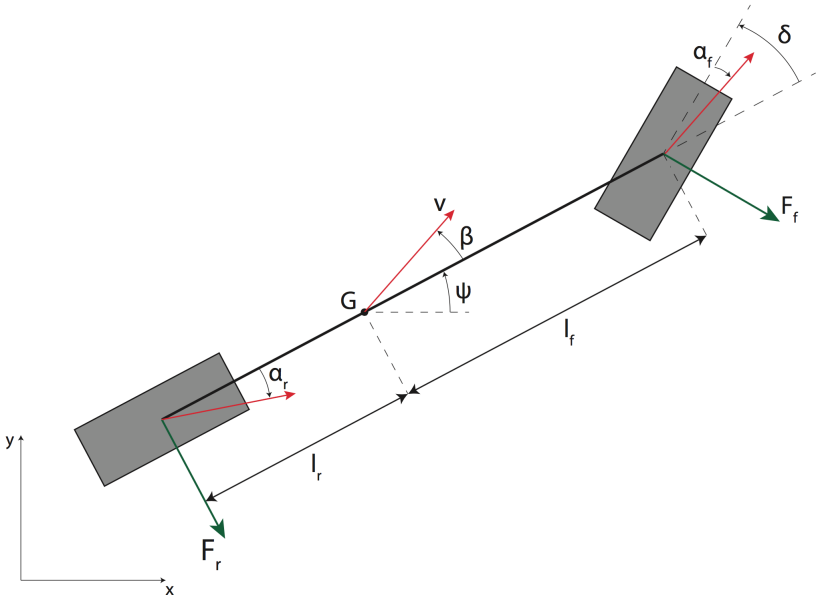


Figure 3.9: *Single-track model.*

- F_f and F_r: front and rear side forces;
- lf and lr: distances between the front and rear axle and the center of gravity;
- C_f and C_r: front and rear cornering stiffness;
- m: total mass of the vehicle;
- Izz: moment of inertia.

Based on the previous definitions, the single-track model is described by the following equations:

$$\begin{cases} \ddot{\psi} = \dfrac{l_r C_r \alpha_r - l_f C_f \alpha_f}{Izz} \\ \dot{\beta} = -\dfrac{C_f \alpha_f + C_r \alpha_r}{vm} - \dot{\psi} \\ \alpha_f = \beta + \dfrac{l_f \dot{\psi}}{v} - \delta \\ \alpha_r = \beta - \dfrac{l_r \dot{\psi}}{v} \end{cases} \tag{3.1}$$

These are nonlinear equations because the velocity v is the system input, together with the steering angle, and it appears at the denominator of some fractions. To complete the system described by Equations 3.1 two equations are necessary to describe the position of the center of gravity of the vehicle relative to a space-fixed frame of reference. Replacing the variables α_f and α_r with their expressions and changing $\ddot{\psi}$ as the first derivative of a new variable r, which is obviously the first derivative of ψ, it is possible to come to the final system of first-order ordinary differential equations:

$$\begin{cases} \dot{\psi} = r \\ \dot{r} = \dfrac{l_r C_r - l_f C_f}{Izz}\beta - \dfrac{l_f^2 C_f - l_r^2 C_r}{vIzz} r + \dfrac{l_f C_f}{Izz}\delta \\ \dot{\beta} = -\dfrac{C_f + C_r}{vm}\beta + \dfrac{l_r C_r - l_f C_f - v^2 m}{v^2 m} r + \dfrac{C_f}{vm}\delta \\ \dot{x} = v\cos(\beta + \psi) \\ \dot{y} = v\sin(\beta + \psi) \end{cases} \tag{3.2}$$

The next step has been the implementation of this system of equations in a ROS node in order to realize the simulator. After analyzing different options for solving ordinary differential equations (ODE) we decided to adopt odeint C++ library. Since the dynamic of the vehicle can change rapidly, a stepper which changes the step size over time has been adopted. Thus, we decided to use the stepper **runge_kutta_dopri5**. Having a dynamic step size, differently from the constant one, makes it possible to find the step size which best fits the integration for every step.

The creation of a simulator led us to implement a ZOH (Zero-order hold) model for the actuators; indeed the values that control the simulation are provided at discrete time, i.e., desired velocity and steering angle. A buffer has been added to the node to hold these inputs; in this way the simulator can run continuously using the latest actuator set points.

The simulator node runs at 50hz and the integration is performed for 0.02s each time. After every cycle, the updated state of the simulator is published using a custom message on the topic **/vehicle_state**.

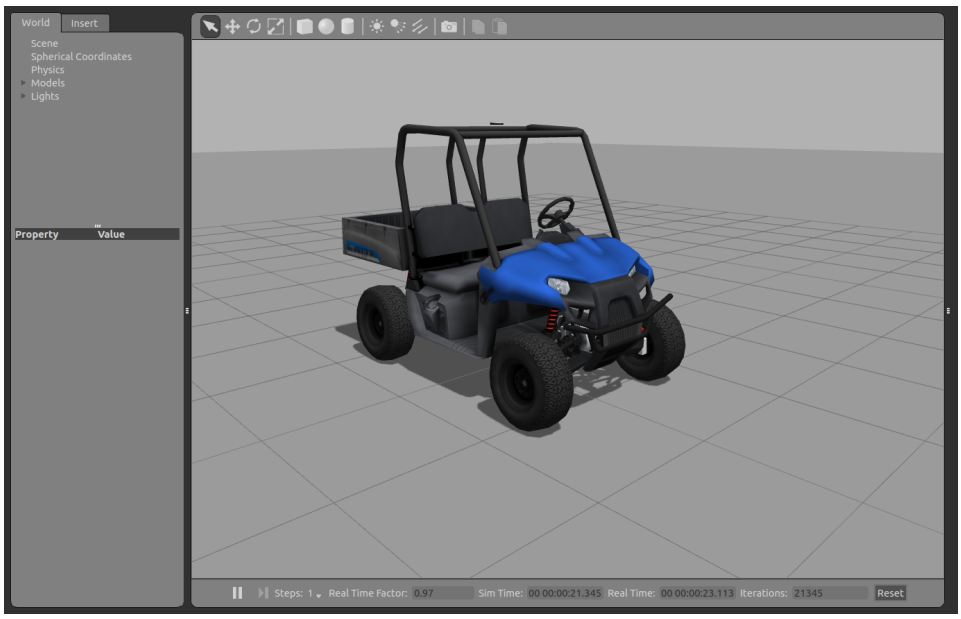


Figure 3.10: *Polaris model in Gazebo.*

3.3.2 Gazebo

The simulation with Gazebo required the creation of a vehicle model. Since the chosen vehicle was used for the DARPA challenge, it was possible to find the relative files to create the model in Gazebo (Figure 3.10). The SDF file that describes the DARPA Polaris has been modified in order to add GPS, IMU and magnetometer sensors.

First of all it was necessary to add a link for each sensor: we put the GPS and the magnetometer on the roof of the car while the IMU on the CoG (center of gravity) of the vehicle. Then, to connect the sensors to the Polaris, two joint elements have been added: both of them have been linked to the chassis. As last step we added the description of the plugin used to simulate the sensors. A detailed description of these steps is provided in the following section.

The next step was to connect Gazebo with ROS in order to retrieve information about the vehicle and the sensors and to send the command to control the Polaris. Again, as for the SDF file, a plugin, named **drcsim_gazebo_ros_plugin** developed within the DARPA challenge, was available. By default it already provided the following topics:

- **/hand_wheel/cmd**
- **/hand_wheel/state**

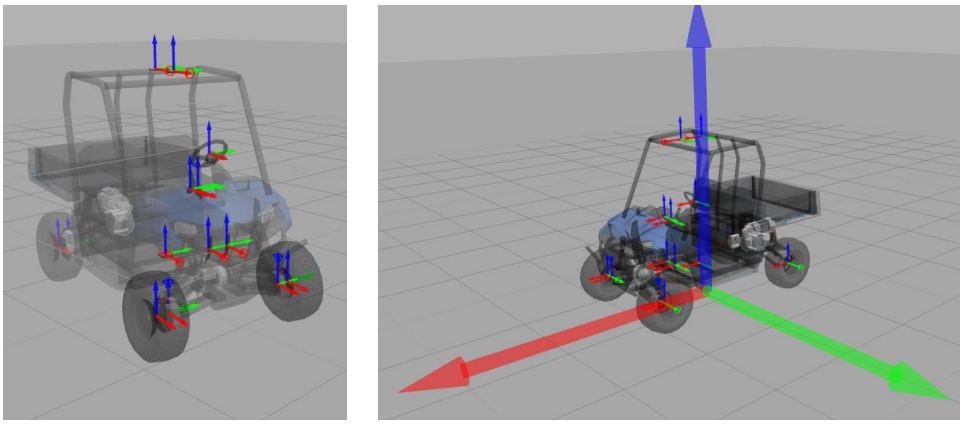
- **/hand_brake/cmd**
- **/hand_brake/state**
- **/gas_pedal/cmd**
- **/gas_pedal/state**
- **/brake_pedal/cmd**
- **/brake_pedal/state**
- **/key/cmd**
- **/key/state**

For each of them **cmd** represents the topic to be used to send commands to the relative element, while the **state** topic is used to publish information about it. Topics relative to the handbrake and the key accept as values only 0 and 1: for the handbrake 0 means that it is off, while for the key that the car is off, while for both, the value 1 has the opposite meaning. Topics for the gas and brake pedals accept values from 0 to 1: the value represents the pressing percentage of the pedal. Values exceeding the lower or upper bound are limited to the relative extreme. The hand wheel topic accepts values from -3.14 to 3.14 and it represents the steering angle. Again, for exceeding values the hand wheel is set to the relative limit. In the following section it is possible to find how the hand wheel and gas pedal are managed.

In the initial condition, the vehicle is turned on, the handwheel steering angle and the gas pedal pressing percentage are set to 0 with the hand-brake on.

3.4 Sensors and actuators

Managing sensors and actuators became necessary as soon as the simulation reached the Gazebo phase. Here we provide a description of how sensors are simulated and how their measurements are handled. Figure 3.11 shows the vehicle equipped with the sensors and their coordinate frames.



(a) (b)

Figure 3.11: *Polaris vehicle, sensors and coordinate frames.*

3.4.1 Global Positioning System

In order to retrieve information about an absolute position to feed the localization module (described in Chapter 4), a plugin in Gazebo has been implemented: **hector_gazebo_plugin**. This plugin, already present in the ROS repository, simulates a GNSS (Global Navigation Satellite System) receiver. As previously explained, it has to be attached to the car modifying the relative SDF file. It publishes ROS messages of type **sensor_msgs/NavSatFix** with the robot's position and altitude in WGS84 (World Geodetic System 1984) coordinates on the topic specified in the SDF file which is **/fix**.

3.4.2 Inertial Measurement Unit

As for the GPS, in order to simulate an IMU is required a plugin. The **hector_gazebo_plugin** plugin used for the GPS provides different simulated sensors, including an Inertial Measurement Unit. The simulated sensor is affected by Gaussian noise and low-frequency random drift. The data is sent through ROS using a **sensor_msgs/Imu** message, which stores IMU measurements (in body coordinates) and estimated orientation relative to the world frame.

It is possible to modify the characteristics of the sensor through the SDF file. There are several XML tags to define its own parameters, the most relevant for the scope of the simulation are:

- *<updateRate>*: update rate of the sensors in hertz;

- *<topicName>*: name of the sensors output topic;
- *<gaussianNoise>*: value used as standard deviation of Gaussian noise for acceleration and angular rate measurements.

To match the characteristics of the Xsens we set the update rate to 50hz and the topic used for publishing messages to **/imu_hector**.

3.4.3 Magnetometer

As for the previous sensors, it was possible to take advantage of the **hector_gazebo_plugin** plugin to simulate a magnetometer. The plugin simulates a 3-axis magnetometer and provides the simulated data through ROS using a **geometry_msgs/Vector3Stamped** message, which stores the magnetic field vector in body coordinates.

Through the SDF file we set the update rate of the sensors to 50Hz and the topic used for publishing messages to **/mag**. The plugin allows also to specify the values for the sensor noise.

3.4.4 Ackermann Odometry

Odometry information was not natively provided by Gazebo, so to retrieve its values it was necessary to modify the plugin used to control the vehicle. To do so a new ROS publisher was implemented within the **drcsim_gazebo_ros_plugin** plugin. Following the name convention used for publishing information about the vehicle, the publisher was set to publish on topic **/gazebo/odometry**.

Since ROS did not provide a message type for the Ackermann odometry, the type created in ROAMFREE package (Section 4.1), named **SingleTrackAckermannOdometryStamped**, has been used. It includes a header to specify time and frame, and two floats, one for the speed and one for the steer. The value of the steer was retrieved using a method provided by the plugin, `DRCVehiclePlugin::GetHandWheelState()`. Here it has been necessary to convert this value since it is bounded between -3.14 and 3.14 radians while for physical limit of the car it goes from -0.65 to 0.65 radians (from -37 to 37 degrees). Thus, a simple linear relation has been used to calculate the desired value. In order to get the speed information we used the one coming from the chassis link. The only way to get this value was to retrieve the information of the velocities of the chassis of the vehicle toward the x-axis and y-axis and then to calculate the resulting vector. After obtaining all the values, the message is published at the same publishing rate of all the other publishers, which is 50hz.

3.4.5 Velocity and steering angle control

It has been also necessary to make changes for managing the velocity and the steering angle of the vehicle. In order to do so we have taken advantage of the topics that Gazebo provides by default, which are:

- **/gas_pedal/cmd**;
- **/hand_wheel/cmd**.

Concerning the steer it was sufficient to convert the command sent within the interval [-3.14, 3.14], for the reason specified in the previous section. This conversion is performed in the plugin when the callback that handles the steering topic is called. Regarding the velocity, the plugin did not provide a direct velocity controller, in fact it was only possible to control the gas pedal, which handles the amount of torque to be provided depending on its pressure percentage. So, we implemented a simple proportional control system to manage the pedal in order to reach the target velocity:

$$gas\_pedal\_percentage = K \cdot e$$

where the error function has been defined as the difference between the target velocity, which is the velocity sent to the topic, and the current velocity, while the constant K has been set to 0.5 after having tested different values.

The *gas_pedal_percentage* value is then used to set the pressing percentage of the gas pedal; results bigger than 1 are handled as 1.

Chapter 4

Localization

Given the aim of this work, great attention has been given to the localization system. Although Gazebo provides information about the position, we use measurements coming from the sensors to feed ROAMFREE library to estimate an absolute position. This is fundamental since the next step, that is beyond the scope of this thesis, is to test the architecture on a real vehicle where the info provided by Gazebo are not present anymore. The maximum speed rate of ROAMFREE is limited and the estimation introduces a delay caused by the time necessary for the pose calculation; so, since for autonomous driving the localization is crucial, we integrated a new node, called fastPredictor, which takes the last pose published by ROAMFREE and integrates the odometer measurements in order to obtain a more precise position at a higher frequency. In Figure 4.1 it is possible to see the general architecture of the localization package.

4.1 Relevant background

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 sensors 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) [17] 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;

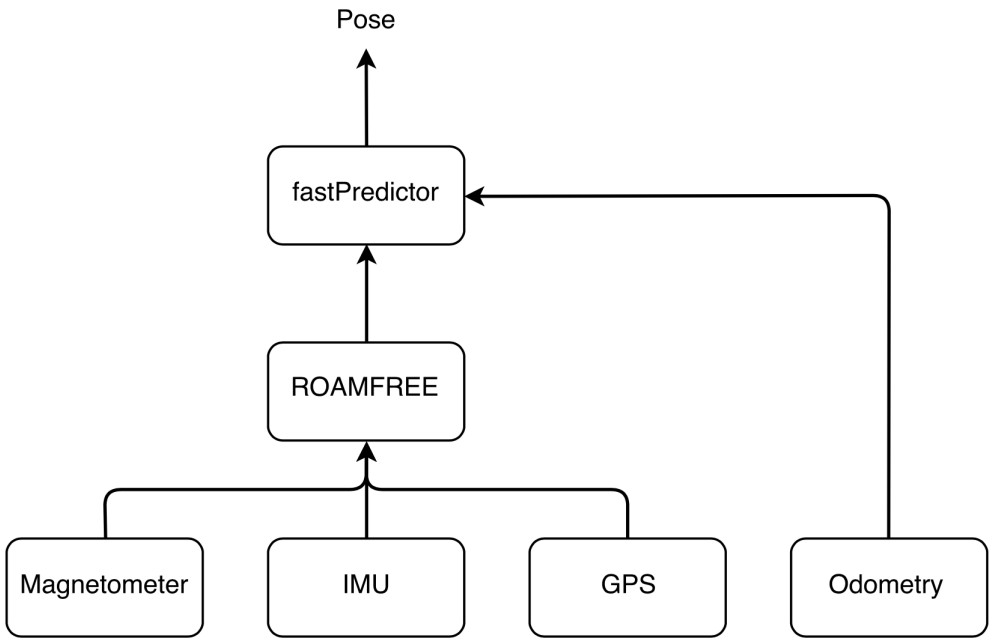


Figure 4.1: *Localization architecture.*

- 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 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 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,

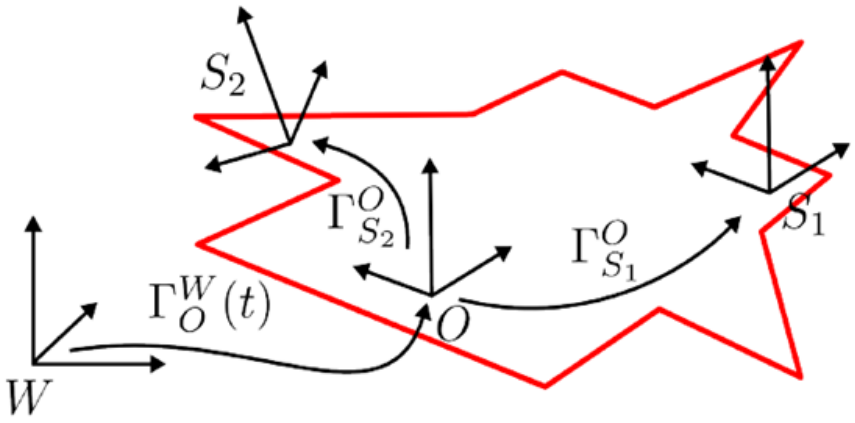


Figure 4.2: *Reference frames and coordinate transformations in ROAMFREE.*

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 4.2).

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 $\overline{e}_i(x_i) = e_i(x_i, \eta)|_{\eta=0}$. Since e_i can involve non-linear dependencies, the covariance of the error is computed through linearization.

$$\sum_{e_i} = J_i \sum_{\eta} J_i^T|_{\eta=0} \tag{4.1}$$

where $\sum_\eta$ is the covariance matrix of η and J_i is the Jacobian of e_i with respect to η. The optimization problem is define as follows:

$$P : \underset{x}{\operatorname{argmin}} \sum_{i=1}^{N} \overline{e}_i(x_i)^T \Omega_{ei} \overline{e}_i(x_i) \tag{4.2}$$

where $\Omega_{ei} = \sum_{e_i}^{-1}$ 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 of the problem can be found by means of the Gauss-Newton algorithm.

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, $\check{\Gamma}_O^W(t)$, 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 it. $z(t)$ 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 sensor parameters are known and the requirements are related to pose 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 in 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 subscribes to the sensors topics and periodically broadcasts the estimated position using **tf**.

4.2 ROAMFREE setup and configuration

In this section we provide the description of the localization process. The first part describes the configuration of the ROAMFREE library, while the second describes how it works. The last part includes the description of the fastPredictor node.

4.2.1 Configuration

The configuration process can be split in different steps. First of all we have the definition of the vehicle initial position and orientation. This may be not be strictly needed if there is a sensor that provides an absolute position, otherwise it is possible to set it through the **initial_pose.yaml** file. This position represents the origin of the global coordinate frame.

The second step is the definition of the sensors specifications through the **config.yaml** file. For each of them we specify the following elements:

- *type*: defines the type of the sensor;
- *is_master*: defines if the sensors is the master for the pose estimation;
- *frame_id*: the coordinate frame of the sensor;
- *topic*: ROS topic in which the sensor publishes its measurements;
- *topic_type*: defines the type of the message used by the sensor to publish its own measurement;
- *static_type*: optional element to specify the static covariance. When is set to true the covariance matrix needs to be specified.

It is also possible to specify some additional parameters for every sensor, for instance it is possible to set the IMU bias relative to the accelerometer and the gyroscope.

The coordinate frames definition is necessary because ROAMFREE needs to know where the sensors are displaced in order to estimate the robot pose. The structure of the reference frames on the vehicle (shown in Figure 4.3) is the following:

- **/world**: is the global coordinate frame. Its origin corresponds to the initial position of the vehicle;
- **/roamfree**: is the frame corresponding to the position estimated by ROAMFREE. The positive x-axis points in the direction of movement of the vehicle, the y-axis points to its left and the z-axis points up.
- **/base_link**: is the frame corresponding to the position predicted by the fastPredictor node. It has the same convention of the roamfree frame;
- **/imu_link**: is the frame of the IMU. Since we decided to set the roamfree frame in the same position of the imu_link, there is no need to define a roto-traslation between these two frames;

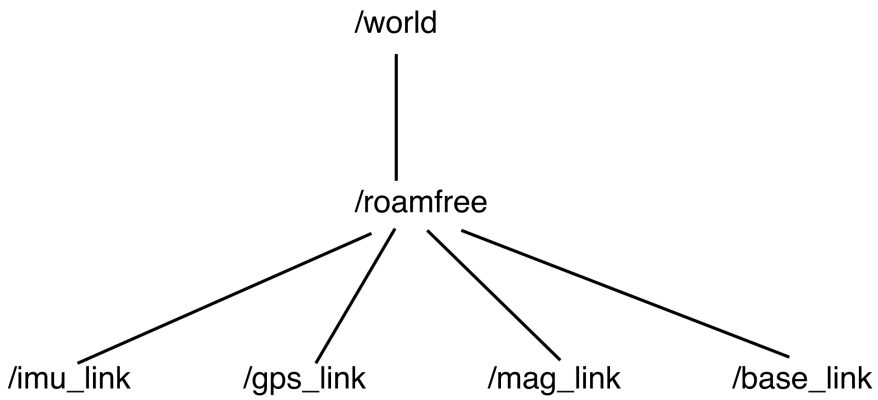


Figure 4.3: *Hierarchy of the coordinate frames.*

- **/gps_link**: is the frame of the GPS. Since the GPS is displaced on the top of the vehicle, there is a static translation between the roamfree and gps_link frames (x=0.0m, y=0.0m, z=1.54m). There is no rotation because the coordinate frame of the GPS is the same as the roamfree one;
- **/mag_link**: is the frame of the magnetometer. As the GPS, it is displaced on the top of the vehicle, so there is a static translation between the roamfree and mag_link frames (x=0.3m, y=0.0m, z=1.54m). Also in this case there is no rotation because the coordinate frame of the magnetometer is the same as the roamfree one;

The final step is given by the definition of the frequency at which ROAMFREE has to estimate the vehicle position and the size of the window used for the estimate. The choice of this last element requires a trade-off: a larger value means that more measurements are used to estimate the position, thus it is possible to obtain an higher accuracy, but this also implies a larger computational time, so the system may not reach the desired frequency. Due to this situation, after several tests, we decided to use a window length of 2 seconds and an estimation frequency of 10Hz.

4.2.2 GPS message conversion

Since the information coming from the GPS is given in geodetic coordinates and ROAMFREE works with ENU coordinates (i.e., East, North and Up), we had to integrate another node, **nmea_to_enu**, to read these messages and to convert the data from the first coordinate system

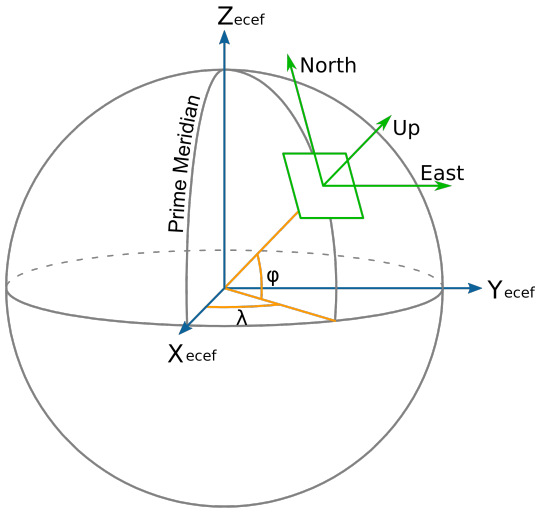


Figure 4.4: *Geodetic (yellow), ECEF (blue) and ENU (green) coordinates.*

to the latter. Then the result of the conversion is published as **geometry_msgs/PoseWithCovarianceStamped** message on the topic **/enu**.

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

$$
\begin{aligned}
X &= (N(\varphi) + h)cos\varphi cos\lambda \\
Y &= (N(\varphi) + h)cos\varphi sin\lambda \\
Z &= (N(\varphi)(1 - e^2) + h)sin\varphi
\end{aligned}
$$

Where:

$$
\begin{aligned}
N(\varphi) &= \frac{a}{\sqrt{1 - e^2 sin^2\varphi}} \\
e^2 &= f(2 - f)
\end{aligned}
$$

being a 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.

To transform from ECEF coordinates to the local coordinates a local reference point is necessary. In our node it is defined using geodetic

coordinates via ROS 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:

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

4.2.3 Estimation

The estimation process starts when the node **roamros** is launched. First of all it loads all the sensors configurations reading the relative parameters in the **config.yaml** file. Then it subscribes to sensor topics: **/enu** to read GPS measurement in ENU coordinates and **/imu_hector** to read the accelerometer and gyroscope measurements. After receiving the initial position, it starts collecting sensor measurements. As soon as it has enough information it estimates the vehicle position and broadcasts it periodically using **tf**.

The master sensor is the one used to build the hyper-graph as described in Section 4.1 since it provides enough information to predict the next pose. The default configuration of ROAMFREE predicts the next pose every time it receives a measurement from the master sensor and corrects the estimation integrating the measurements coming from other sensors. In our configuration we decided to adopt the IMU as master. Since the IMU publishing rate was to high compared to the GPS one, ROAMFREE did not have enough measurements coming from the other sensor to correct the estimation in time. To solve this problem we implemented a IMU handler to manage the IMU measurements. Since ROAMFREE is set to publish poses at 10Hz and the IMU runs at 50Hz, the handler reads and integrates 5 measurements so as to allow ROAMFREE to meet the publishing rate and furthermore to estimate a better position since it can integrate GPS measurements which have a 5Hz frequency.

4.2.4 Fast predictor

Localization plays a big role in autonomous vehicle, so it is very important to have an accurate and fast localization system. We could consider

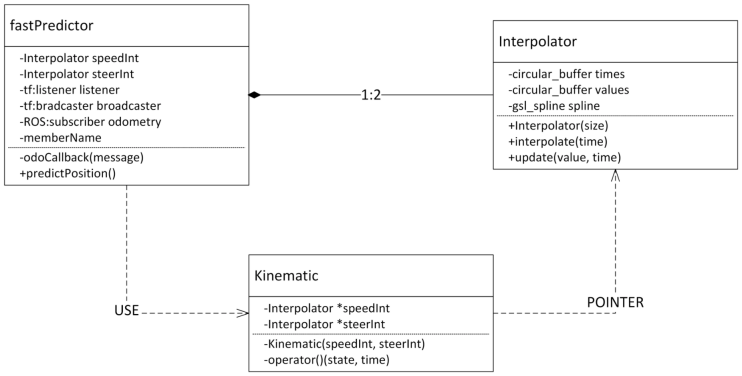


Figure 4.5: *The class diagram of the fastPredictor node.*

ROAMFREE accurate enough, but we need to face the delay introduced by the time required to compute the position. As first solution we considered the opportunity to increase the frequency of *roamros*, but it would have driven to a less accurate estimation and an higher computational load. Therefore we decided to implement a node, called **fastPredictor**, developed originally within another thesis [38], that predicts a relative position starting from the absolute position given by roamros and integrating the odometry of the vehicle. Figure 4.5 shows its class diagram.

The node subscribes to the topic **/polaris/odometry** to retrieve information about the speed and the steer of the vehicle. Furthermore, it implements a listener to receive the position broadcast by *roamros*. In order to use the node it is necessary to specify a few parameters: the coordinate frame of the absolute position, which is **/roamfree**, as stated previously, at which frequency this position is published, 10Hz in our case, and the frequency of the odometry, which is 50Hz. In Figure 4.6 is shown the relative ROS communication flow. The ratio between these two values represents the number of odometry measurement to store, since in the worst case the system needs at most $f_o/f_r + 1$ measurements to compute the relative position before roamros estimates a new pose, where f_o is the odometry frequency while f_r is the *roamros* frequency.

In order to perform the integration of the odometry, it is necessary to define the differential equations that describe the kinematic of the vehicle. In our case, which is a four-wheel model with an Ackermann steer, the required equation are represented by the following ones:

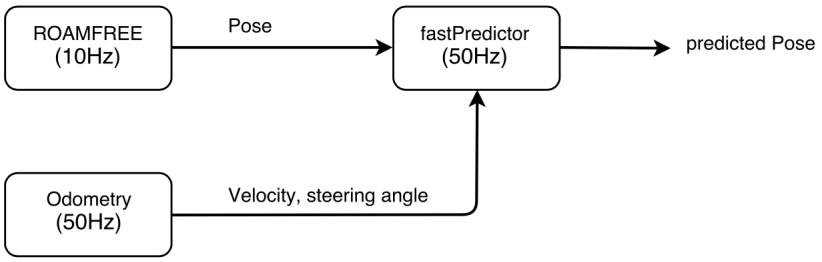


Figure 4.6: *The fastPredictor ROS communication flow.*

$$\begin{cases} \dot{x} = v(t) \\ \dot{\vartheta}_z = \dfrac{v(t)}{L} \tan(\varphi(t)) \end{cases} \tag{4.3}$$

Where $v(t)$ is the speed, $\varphi(t)$ is the steering angle and L is the wheelbase of the vehicle. The integration is performed using the *odeint* library. The starting time of the integration is set to the timestamp of the newest pose estimated by roamros, while the ending time is the current time plus a dt: this step forward in the future provides a prediction of the future position of the vehicle and grants the lowest possible delay. Velocity and steer values used in this prediction are computed by means of a linear extrapolation between the last two available values. Then the result is published as a transformation from the coordinate frame **/roamfree** to **/base_link** using **tf**.

Chapter 5

Model Predictive Control

This chapter provides a description of the MPC package developed within this thesis. Before going into detail about the implementation of the controller, an overview of Model Predictive Control is provided. Additionally it is explained how to integrate and use CPLEX, an optimization software package useful to solve optimization problem. In Figure 5.1 an high level representation of the structure of the developed package is shown. In the following a detailed description for each node that composes the package is provided.

5.1 Relevant background

Model Predictive Control (MPC) is a control framework which uses a model of the process under control to obtain a suitable control signal, by minimizing an objective function under operational restrictions [13]. A dynamic model is required to predict the effect of future control actions to the output.

The basic concept of MPC is that of transforming the classic control problem into a mathematical optimization problem, so that it is possible to simply insert constraints and limitations of the real system. In particular, the dynamic model of the system is used to predict, on a finite horizon, how the state variables will evolve starting from their values at current time and as a function of the future values of the control actions. In this way the prediction can be used in a cost function that has to be minimized so that the state adequately follows a specified target. The result is the optimal input sequence which allows to get the minimum of the cost function. MPC techniques have been largely discussed in the last few decades, as for example in [36] and in [13].

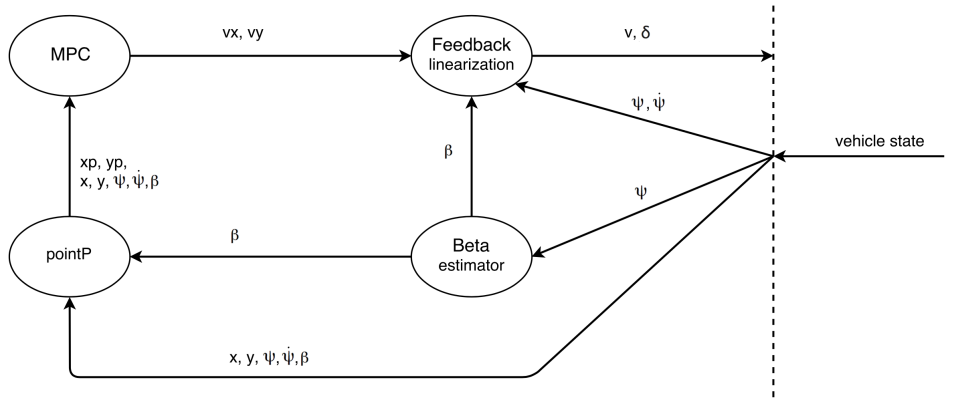


Figure 5.1: *MPC package.*

Human beings can easily drive a vehicle due to the brain ability of predicting obstacles and road direction; this should characterize also the controller in a self-driving car. For this reason, Model Predictive Control is very suitable in autonomous vehicle applications.

This kind of control has had a significant success in the industrial area starting from the 80's, when it started to be applied mainly in the petrochemical sector; the key of this was the simplicity of the algorithm. The increasing potential of digital computers has helped the spread and development of MPC in different sectors, such as indeed autonomous vehicles.

The main features which characterize this control logic are:

- formalization of the control problem as an optimization problem in which is possible to include several aims, that sometimes are even contrasting;
- explicit inclusion in the control problem of constraints on the state variables and the inputs;
- possibility of designing the regulator starting from empirical models of the process gotten from simple plant tests, such as step or ramp response.

The MPC technique can be applied on both linear and nonlinear systems, and in the latter case it is called NMPC. For linear systems the optimization problem leads to a linear quadratic problem, i.e., the problem of optimizing a quadratic function, with a massive reduction

in the computational time for optimization compared to the nonlinear situation. Indeed, the computational burden has been a critical aspect of the Model Predictive Control framework, especially in the past when the hardware was not as powerful as it is today. The optimization and the variable estimate processes are responsible for the biggest computational costs; the choice of using a linear MPC through the application of the feedback linearization shown in Section 5.3 guarantees a reduction of the first contribution, beside a simplification of the formulation.

The application of Model Predictive Control has raised recently in quantity and quality not only thanks to the increase in hardware calculus power, but also through the use of new optimization algorithms, which are more powerful and reliable, and in particular through the development of the theory that has led to new methods which guarantee fundamental properties such as stability and robustness.

Model Predictive Control of linear systems

Consider a linear discrete time system, in state-space representation:

$$x(t+1) = Ax(t) + Bu(t), \tag{5.1}$$

in which the state $x \in X$ is assumed measurable and $u \in U$ is the vector of control variables. X and U contain the origin as an interior point. The aim of the controller at time t is to establish the optimal control sequence $u(t), u(t+1), ..., u(t+N-1)$, where N is a positive integer quantity called prediction horizon. The controller determines the optimal control sequence which minimizes the quadratic cost function on a finite horizon, that can be generically written as:

$$J(x(t), u(\cdot), t) = \sum_{k=0}^{N-1} (\|x(t+k)\|_Q^2 + \|u(t+k)\|_R^2) + \|x(t+N)\|_S^2, \tag{5.2}$$

where Q is symmetric and positive semi-definite matrix and R and S are diagonal positive definite matrices, with proper dimensions.

The MPC optimization problem consists in finding at any time t the optimal control sequence:

$$u(t),\ u(t+1),\ ...,\ u(t+N+1), \tag{5.3}$$

which minimizes the cost function 5.2 subject to the constraints

$$x(t+k) \in X, \quad u(t+k) \in U, \quad x(t+N) \in X_f. \tag{5.4}$$

MPC as a quadratic program

The optimization problem can be cast as a suitable quadratic program, starting from the Lagrange formula for discrete-time systems:

$$x(t+k) = A^k x(t) + \sum_{i=0}^{k-1} (A^{k-i-1} B u(t+i)), \quad k > 0, \tag{5.5}$$

where the first contribution represents the free motion, while the second is the forced motion. It is therefore possible to write the evolution of the system state within the finite prediction horizon as:

$$\mathbb{X}(t) = \mathbb{A}x(t) + \mathbb{B}\mathbb{U}(t), \tag{5.6}$$

where:

$$\mathbb{X}(t) = \begin{bmatrix} x(t) \\ x(t+1) \\ x(t+2) \\ \vdots \\ x(t+N-1) \\ x(t+N) \end{bmatrix}, \quad \mathbb{U}(t) = \begin{bmatrix} u(t) \\ u(t+1) \\ u(t+2) \\ \vdots \\ u(t+N-2) \\ u(t+N-1) \end{bmatrix}, \quad \mathbb{A} = \begin{bmatrix} I \\ A \\ A^2 \\ \vdots \\ A^{N-1} \\ A^N \end{bmatrix}$$

$$\mathbb{B} = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 0 \\ B & 0 & 0 & \cdots & 0 & 0 \\ AB & B & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ A^{N-2}B & A^{N-3}B & A^{N-4}B & \cdots & B & 0 \\ A^{N-1}B & A^{N-2}B & A^{N-3}B & \cdots & AB & B \end{bmatrix} \tag{5.7}$$

It is necessary now to write also the cost function as a product of proper matrices and in particular the minimum of the objective function 5.2 is equal to the minimum of the following one:

$$\overline{J}(x(t), u, t) = \mathbb{X}^T(t)\mathbb{Q}\mathbb{X}(t) + \mathbb{U}^T(t)\mathbb{R}\mathbb{U}(t), \tag{5.8}$$

where $\mathbb{Q}$ and $\mathbb{R}$ are:

$$\mathbb{Q} = \begin{bmatrix} Q & 0 & \cdots & 0 & 0 \\ 0 & Q & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & Q & 0 \\ 0 & 0 & \cdots & 0 & S \end{bmatrix}, \quad \mathbb{R} = \begin{bmatrix} R & 0 & \cdots & 0 & 0 \\ 0 & R & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & R & 0 \\ 0 & 0 & \cdots & 0 & R \end{bmatrix}. \tag{5.9}$$

Expanding the functional 5.8 following the expression 5.6:

$$\begin{aligned}\overline{J}(x(t),u,t) &= (\mathbb{A}x(t)+\mathbb{B}\mathbb{U}(t))^T\mathbb{Q}(\mathbb{A}x(t)+\mathbb{B}\mathbb{U}(t))+\mathbb{U}^T(t)\mathbb{R}\mathbb{U}(t)\\ &= x^T(t)\mathbb{A}^T\mathbb{Q}\mathbb{A}x(t)+2x^T(t)\mathbb{A}^T\mathbb{Q}\mathbb{B}\mathbb{U}(t)+\mathbb{U}^T(t)(\mathbb{B}^T\mathbb{Q}\mathbb{B}+\mathbb{R})\mathbb{U}(t).\end{aligned} \tag{5.10}$$

The resulting optimization problem can be solved using quadratic programming solvers, such as QUADPROG [72] and CPLEX [16].

The Receding Horizon (RH) control principle

The optimization of the control problem over a finite horizon window leads to the definition of the optimal control input vector $\mathbb{U}^o(t)$, which contains the N control actions, i.e., the current value at time t and the future values.

The *Receding Horizon* or *moving horizon* principle consists of the fact that at each time t the optimization problem is solved on a finite horizon $[t, t+N]$ and just the first action $u^o(t)$ of the input vector $\mathbb{U}^o(t)$ is applied to the system. At the next sampling time $t+1$ the optimization and all the operation is repeated within the temporal window $[t+1, t+N+1]$. This explanation is also shown in Figure 5.2.

The new control sequence obtained at time $t+1$ is generally different from the previous one, since the moving horizon shifts towards and the controller predict the future values and establish the control inputs starting from the new value of the state $x(t+1)$.

CPLEX

IBM® ILOG® CPLEX® Optimization Studio is an analytical decision support toolkit for rapid development and deployment of optimization models using mathematical and constraint programming. It combines an integrated development environment (IDE) with the powerful Optimization Programming Language (OPL) and high-performance ILOG CPLEX optimizer solvers. [48]

The use of CPLEX to solve an optimization problem requires several steps. First of all it is necessary to build the environment: an instance of `IloEnv` object is created, it represents the container of the problem.

After creating the environment it is possible to create the optimization model. The definition of a model is performed using a `IloModel` object that needs to be linked with its environment passing the `IloEnv` object as argument to the constructor.

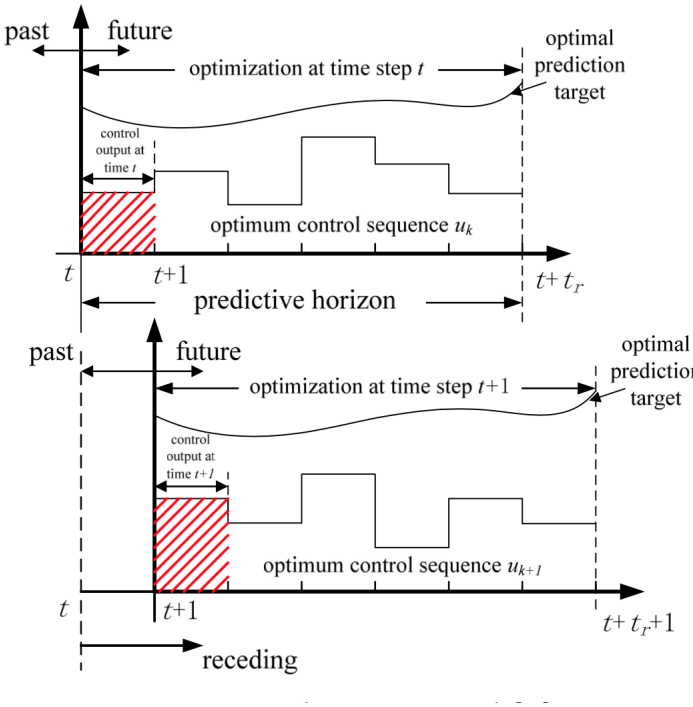


Figure 5.2: *Receding Horizon principle [7].*

After an `IloModel` has been instantiated, it is populated with the executables that define the optimization problem. The most important classes are:

- `IloNumVar`: representing modeling variables;
- `IloRange`: defining constraints of the form $l \leq expr \leq u$, where $expr$ is a linear expression;
- `IloObjective`: representing an objective function.

It is possible to create objects of the previous classes for each variable, constraint and objective function of the optimization problem. Then to add the objects to the model it is sufficient to call, for each object, the following element:

```
model.add(object);
```

Modeling variables are constructed as objects of class `IloNumVar`, by defining variables of type `IloNumVar`. There are several constructors for doing this; the most flexible form is:

```
IloNumVar x1(env, l, u, type);
```

This definition creates the modeling variable $x1$ with lower bound l and upper bound u. The choice about the type can be done among `ILOFLOAT` for continuous variables, `ILOINT` for integer variables and `ILOBOOL` for Boolean variables. For each variable in the optimization model a corresponding object of class `IloNumVar` must be created.

After all the modeling variables have been constructed, they can be used to build expressions, which in turn are used to define objects of class `IloObjective` and `IloRange`.

After the optimization problem has been created in an `IloModel` object, the next step is to create the `IloCplex` object for solving the problem by creating an instance of the class `IloCplex`. It is possible to add directly to the `IloCplex` object the model object or to add the environment object and then to extract the model. Then the object cplex is ready to solve the optimization problem defined by the model. To solve the model it is sufficient to call:

```
cplex.solve();
```

This method returns an `IloBool` value, where `IloTrue` indicates that CPLEX successfully found a feasible (yet not necessarily optimal) solution, while `IloFalse` indicates that no solution was found. More precise information about the outcome of the last call to the method solve can be obtained by calling:

```
cplex.getStatus();
```

The returned value tells what CPLEX found out about the model: whether it found the optimal solution or only a feasible solution, whether it proved the model to be unbounded or infeasible, or whether nothing at all has been proved at this point. Even more detailed information about the termination of the solve call is available through the method `getCplexStatus`.

5.2 MPC

Here it is described the application of this technique for our control problem. Since we dealt with an optimization problem, we followed the standard form [37] in order to describe the regulation problem. In the following we give an overview of the controller which has been developed within the thesis of M. Spaliviero [11]. First of all we consider the objective function then we move on to the evaluation of the constraints which bound the function. Lastly, we explain the implementation process with its relative toolkit.

Cost function

The regulation problem consists in reaching a target position with null velocity and free orientation without violating vehicle limits and constraints, starting from a generic initial position. In order to achieve this result it is necessary to minimize a certain cost function. This is the approach that has been followed: the controller itself generates a reference position $\tilde{x}(t)$ that is forced to get close to the final target x_{goal} though the objective function:

$$J = \sum_{k=t}^{t+N-1} (\|x(k) - \tilde{x}(t)\|_Q^2 + \|\delta u(k)\|_R^2) + \gamma \|\tilde{x}(t) - x_{goal}\|^2, \tag{5.11}$$

in which x_{goal} is a vector that contains the x-y coordinates of the destination point. Through this function we force the x-y position to get close to the point $\tilde{x}(t)$ and at the same time the reference to approach to the target position x_{goal}. Also the reference point is an optimization variable, as the value of $\delta u(k)$ for $k = t, ..., t + N - 1$, so the minimization of the function must return the following control action, here expressed in vectorial notation:

$$\Theta = \begin{bmatrix} \delta u(t) \\ \delta u(t+1) \\ \vdots \\ \delta u(t+N-1) \\ \tilde{x}(t) \end{bmatrix} = \begin{bmatrix} \delta U(t) \\ \tilde{x}(t) \end{bmatrix}. \tag{5.12}$$

In order to obtain these optimal control action through a quadratic programming algorithm, it is necessary to write the optimization problem (and therefore the objective function) as a quadratic one:

$$\min_{\Theta(t)} J(\Theta, t) = \min_{\Theta(t)} \frac{1}{2} \Theta(t)^T H(t) \Theta(t) + f^T(t) \Theta(t) + cost, \tag{5.13}$$

where H and f are respectively the Hessian and the linear term in a quadratic optimization problem, while the constant term is independent of the optimization variable Θ and so it is not involved in the minimization process.

Zero terminal constraint

In order to guarantee feasible trajectories and stability in the regulation problem, it has been chosen to allow the final velocity of the vehicle in both

Cartesian direction to be null, hence imposing a zero terminal constraint of the type:

$$u(t+N) = u^{APP}(t+N-1) = 0, \tag{5.14}$$

where $u^{APP}(t)$ is the actual velocity of the vehicle at time t. This hypothesis is also important since it allows the system to possibly stop within a certain distance. In fact in this way it is guarantee that at each cycle of the controller the system can possibly be at rest at the end of the horizon prediction T_p. Besides, it has also been inserted a terminal constraint on the position consistently, i.e.:

$$x(t+N) = \tilde{x}(t). \tag{5.15}$$

Acceleration, velocity and position constraints

In order to delimit the operational field of the vehicle and to take account of the saturation of the actuators it is necessary to limit the possible values of position, velocity and acceleration. This can be done as follows:

$$\begin{aligned} -x_{max} \leqslant x_{x,y}(k) \leqslant x_{max}, &\quad k = t, t+1, \ldots, t+N-1, t+N, \\ -u_{max} \leqslant u_{x,y}^{APP}(k) \leqslant u_{max}, &\quad k = t, t+1, \ldots, t+N-1, \\ -a_{max} \leqslant a_{x,y}(k) \leqslant a_{max}, &\quad k = t, t+1, \ldots, t+N-1. \end{aligned} \tag{5.16}$$

While for position and velocity it is possible to easily add their relative constraints since they are part of the state vector, for the acceleration it has been necessary to define this variable as the ratio between the variation of velocity $\delta u_{x,y}(k)$ and the time interval $\tau(k)$:

$$a_{x,y}(k) = \frac{\delta u_{x,y}(k)}{\tau(k)}, \quad k = t, t+1, \ldots, t+N+1. \tag{5.17}$$

It is also necessary to enforce another constraint of the velocity for guaranteeing consistency between the feedback linearized model (Section 5.3) and the nonlinear system variables. To guarantee that the longitudinal speed is positive, the following constraint is added:

$$v(k) = u_x^{APP}(k)cos(\beta+\psi) + u_y^{APP}(k)sin(\beta+\psi) \geq 0 \quad . \tag{5.18}$$

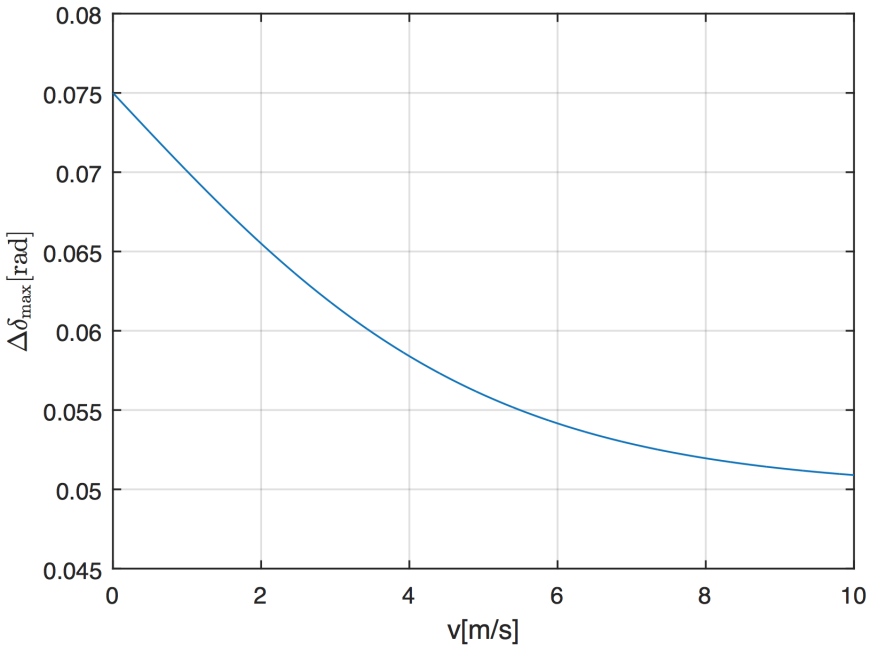


Figure 5.3: $\Delta\delta_{max}$ *as function of the velocity* v.

Steering angle constraints

In order to approximate the real behavior of a vehicle and its limitations, it is important to introduce in the controller also a constraint on the steering angle δ:

$$\begin{cases} \delta(t+k) \leqslant \delta_{max} \\ -\delta(t+k) \leqslant \delta_{max} \end{cases} . \tag{5.19}$$

It is also necessary to impose a constraint on the variation of the steering angle $\Delta\delta(k) = \delta(k) - \delta(k-1)$. In particular we want to limit it, so that it is lower than a certain value $\Delta\delta_{max}$. This prevents the swerves from being too abrupt, following the real dynamics of the vehicle.

The value of $\Delta\delta_{max}$ has to be function of the velocity of the vehicle. In fact it is a common experience that the danger of a steering maneuver is function of the velocity. The analytical expression of this behavior is:

$$\Delta\delta_{max} = 0.05 + \frac{0.05}{1 + e^{0.4v}} \quad , \tag{5.20}$$

which is shown in Figure 5.3.

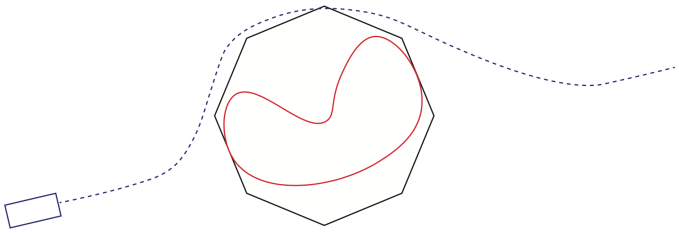


Figure 5.4: *Approximation of an obstacle with a regular polytope.*

Obstacle avoidance constraints

One of the task of the controller is to avoid obstacles, generating trajectories which guarantee no collisions during the path. Since the collision detection goes beyond the scope of this thesis, we implemented an obstacle avoidance using static obstacles. The representation of an obstacle is performed using a 2-dimensional polytope. In particular the considered geometric figure is:

- bounded: there is a circle with a finite radius that contains it;
- convex: it is also a set of points in the 2-dimensional space R^2. The polytope can be represented by a system of linear inequalities:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 &\leq b_m \end{aligned} \tag{5.21}$$

- regular: which is the most symmetrical and simple kind of polytope.

In this way we approximate the obstacle with a regular polytope, which is circumscribed by a circumference of radius r as shown in Figure 5.4.

The implementation of the obstacle avoidance requires to verify if the predicted trajectory is inside or outside the polytope. In order to perform the verification, the distances between the edges of the polytope which circumscribes the obstacle and the predicted positions are calculated.

Every edge of the polytope, considering its extensions on both the sides, divides the x-y plane into two half planes. The sign of the distances ρ conveys in which part of the plane the vehicle is with respect to the edges:

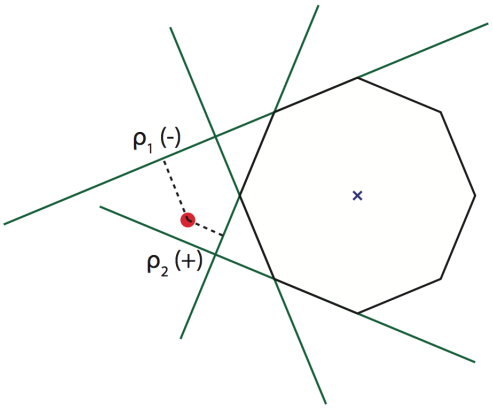


Figure 5.5: *Distances between the vehicle and the polytope edges.*

if it is positive the vehicle is in the half which does not contain the center of the polytope, if it is negative the vehicle is in the half with the center. Figure 5.5 shows this consideration.

If the trajectory is outside the polytope at least one difference for each predicted point will be positive. Thus, a constraint for each predicted point is created stating that the relative maximum distance has to be positive:

$$\overrightarrow{\rho}_{max_{(N+1)x1}} = \tilde{D}_{(N+1)x2}(P_{veh} - P_{obs})_{2x(N+1)} - \overrightarrow{1}_{(N+1)x1} \geq 0, \qquad (5.22)$$

where P_{veh} is the vector of the predicted positions, P_{obs} is a vector with the same dimension of P_{veh} which contains the origin of the obstacle, and $\tilde{D}$ is the matrix that represents, for each predicted position, the furthest edge.

In this way the obstacle avoidance has been implemented just as an additional constraint on the optimization and thanks to this choice the computational burden of the Model Predictive Control does not increase significantly. This guarantees that, if the vehicle does not change actions with respect to ones predicted at the previous instant, then obstacle avoidance is guaranteed.

5.3 Feedback linearization

Since the model that describes the vehicle is a nonlinear model, we decided to use a technique, known as feedback linearization [33], to linearize the input-output relation of the nonlinear system.

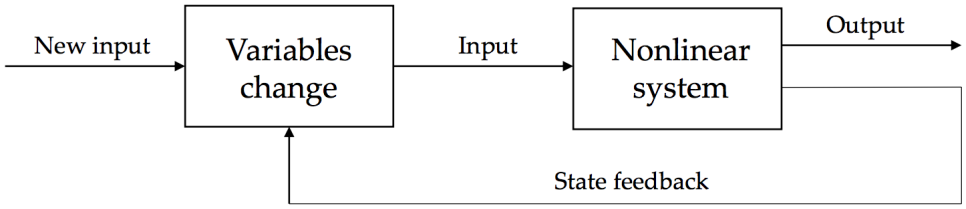


Figure 5.6: *Block diagram explaining how the feedback linearization works.*

This approach leads to an exact linearization through a change of variables and a state feedback that removes the nonlinearities of the system under control, as represented in Figure 5.6.

The feedback linearization technique allows to represent, from an external point of view, the relationship between input and output as a chain of integrals. Therefore the system becomes linear just to external effects, while it does not change internally, and this is the reason why the linearization is exact and not approximated.

The linearization process goes through the definition of a point P at distance p from the center of mass of the vehicle along the longitudinal direction (Figure 5.7), whose coordinates are:

$$\begin{aligned} x_p &= x + p \cdot cos(\varphi + \beta), \\ y_p &= y + p \cdot sin(\varphi + \beta), \end{aligned} \tag{5.23}$$

where x and y are the Cartesian coordinates of the vehicle while φ and β are respectively the yaw and sideslip angle.

Differentiating Equation 5.23 with respect to time, we can obtain:

$$\dot{x_p} = \dot{x} - psin(\beta + \psi)(\dot{\beta} + \dot{\psi}) = vcos(\beta + \psi) - psin(\beta + \psi)(\dot{\beta} + \dot{\psi}), \tag{5.24}$$

$$\dot{y_p} = \dot{y} + pcos(\beta + \psi)(\dot{\beta} + \dot{\psi}) = vsin(\beta + \psi) + psin(\beta + \psi)(\dot{\beta} + \dot{\psi}). \tag{5.25}$$

More specifically, Equations 5.24 and 5.25 describe the dynamics of point P. The inputs to the feedback-linearized model are defined as the velocity of point P with respect to the x-axis and y-axis, respectively defined as:

$$v_{P_x} = vcos(\beta + \psi) - psin(\beta + \psi)(\dot{\beta} + \dot{\psi}), \tag{5.26}$$

$$v_{P_y} = vsin(\beta + \psi) + pcos(\beta + \psi)(\dot{\beta} + \dot{\psi}). \tag{5.27}$$

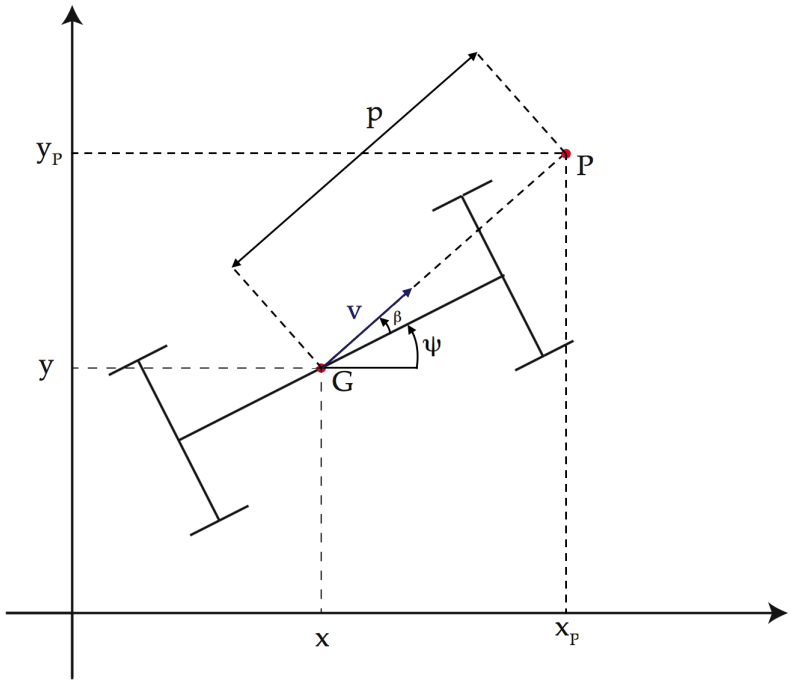


Figure 5.7: *Representation of the point P under control.*

The feedback-linearized model is then:

$$\begin{aligned} \dot{x_p} &= v_{P_x}, \\ \dot{y_p} &= v_{P_y}. \end{aligned} \tag{5.28}$$

Now it is possible to show that, given v_{P_x} and v_{P_y}, there exist output values for the final model (i.e., values of δ and v) such that the dynamics of point P actually corresponds with Equations 5.28. To do so, first Equation 5.26 is multiplied by $cos(\beta+\psi)$ and Equation 5.27 by $sin(\beta+\psi)$ and then, the two equations are added up together. We obtain that:

$$v = v_{p_x} \cdot cos(\beta+\psi) + v_{p_y} \cdot sin(\beta+\psi) \tag{5.29}$$

i.e., the velocity of the center of mass.

Also, we multiply Equation 5.26 by $sin(\beta+\psi)$ and Equation 5.27 by $cos(\beta+\psi)$ and we subtract them, we obtain that:

$$p(\dot{\beta}+\dot{\psi}) = v_{p_y} \cdot cos(\beta+\psi) - v_{p_x} \cdot sin(\beta+\psi). \tag{5.30}$$

Furthermore, recalling System 3.1 and denoting $\dot{\beta}+\dot{\psi} = \omega$, we obtain that:

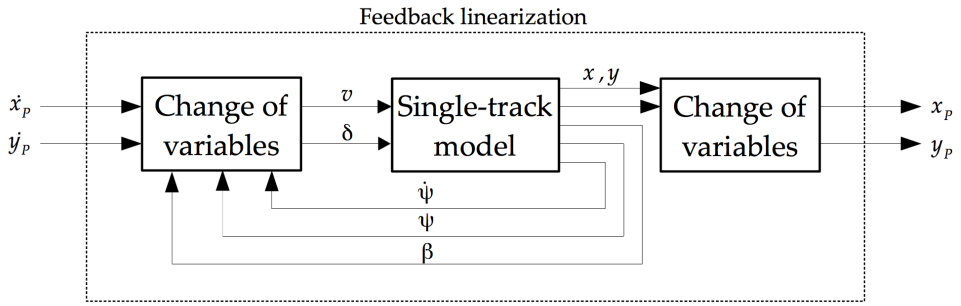


Figure 5.8: *Feedback linearization applied to the single-track model.*

$$\omega = \frac{1}{vm}\left[-(C_f + C_r)\beta + \frac{\psi}{v}(C_r l_r - C_f l_f) + C_f \delta\right]. \tag{5.31}$$

Therefore, also δ can be retrieved from ω (i.e., from the linear system inputs v_{p_x} and v_{p_y}) and the system variable as follows:

$$\delta = \frac{vm}{C_f}\left[w + \frac{C_r + C_f}{vm}\beta - \frac{C_r l_r - C_f l_f}{v^2 m}\dot{\psi}\right]. \tag{5.32}$$

Since velocity appears at the denominator of two of these formulas, it has been necessary to elaborate another formulation to be used when the velocity is near to a null value. The chosen equation to complete the alternative formulation of the model is:

$$\delta = \frac{C_f + C_r}{C_f}\beta. \tag{5.33}$$

Figure 5.8 shows how the feedback linearization is applied to the single-track model.

For this first step on linearization, i.e, the point P definition, we developed a node, called **pointP**, that subscribes to three topics:

- /**estimated_beta**: topic on which the node **beta_estimator** publishes the estimated sideslip angle (described in Section 5.5);
- /**vehicle_state**: topic on which simulators or ROAMFREE publish the current state of the vehicle which includes its Cartesian coordinates and orientation;
- /**yaw_angle**: topic on which it is possible to read information about the yaw angle of the vehicle with respect to z-axis;

and it publishes the resulting point P coordinates to feed the MPC on topic **/pointP**.

The second step is to evaluate the MPC output, i.e., the velocity on x-axis and y-axis, to produce the resulting velocity vector and steer angle to control the vehicle, considering also yaw and sideslip angles. To achieve this task, we have developed a new node, called **feedback_linearization**, which reads on three topics; /**estimated_beta** and /**vehicle_state**, described above, and on:

/**mpcVel**: topic on which MPC controller publishes its own output.

The resulting values, calculated by means of the Equations 5.29 and 5.32, are then published on a custom message, **velocity_steer**, on the topic /**vehicle**/**set_velocity_steer**.

5.4 MPC implementation

In order to implement this controller we developed a ROS node, called **MPC**. Since almost every operation is performed among matrices, we decided to take advantage of the Eigen C++ library, which is a template library for linear algebra [21], which provides a data type to store matrices and implements several operations, such as product, division, and transpose. Furthermore the library performs its own loop unrolling [50] which allows to achieve better performance.

In order to work properly the controller needs to know the vehicle parameters. Thus, we created a configuration file, called *vehicle.yaml*, which contains the vehicle data. It also contains information about the obstacle, i.e., the $x-y$ coordinates of its center, the number of the polytope sides which describe the obstacle and the radius length of the circumference which circumscribes it. Furthermore it includes the Cartesian coordinates of the goal and the value of the N intervals of the prediction horizon. Therefore, when the node is launched it reads this file, loads all the values and then starts the execution.

In order to make the code more flexible and readable, we created two objects to store the values included in the configuration file, one for the parameters regarding the controller and one for the parameters regarding the vehicle. The first is called *mpc_parameters* and contains the following elements:

- _A_MAX: maximum acceleration $[m/s^2]$;
- _V_MAX: maximum velocity $[m/s]$;

- _X_MAX: maximum x allowed $[m]$;
- _Y_MAX: maximum y allowed $[m]$;
- _DELTA_MAX: maximum steering angle allowed $[deg]$;
- _X_GOAL: x coordinate of the goal $[m]$;
- _Y_GOAL: y coordinate of the goal $[m]$;
- _N: number of intervals of the prediction horizon;
- _Q_VAL: value characterizing the weights matrix on the state;
- _R_VAL: value characterizing the weights matrix on control;
- _GAMMA: weight to push $\tilde{x}$ toward the goal;
- _LAMBA: slack variable that allows to violate the obstacle constraint;

while the second is called *vehicle_parameters* and contains the following elements:

- _M: mass of the vehicle $[kg]$;
- _IZZ: moment of inertia on z-axis $[kg \cdot m^2]$;
- _LF: distance between the vehicle CoG and the front axle $[m]$;
- _LR: distance between the vehicle CoG and the rear axle $[m]$;
- _CF: front cornering stiffness $[N/rad]$;
- _CR: rear cornering stiffness $[N/rad]$;
- _P: distance between the CoG and the point P $[m]$ (Section 5.3).

The **MPC** node subscribes to three topics, **/estimated_beta** to read the last value of the sideslip angle (Section 5.5), **/yaw_rate** to read the last value of the yaw rate and to **/pointP** to read the state of the vehicle and its relative linearization. It runs continuously and at every iteration it calculates the values of velocity to be applied to the vehicle. The execution of this node can be divided in four phases:

1. create the objective function and the constraints matrices for the optimization problem;

2. execute the optimization;

3. get the output of the optimization, elaborate the resulting velocities and perform the prediction to calculate the future N positions;

4. send the velocities on topic **/mpc_vel**.

Optimization

To perform the optimization problem we decided to adopt CPLEX. Therefore, we developed a C++ class, called CPLEXsolver, in order to implement this library. To take advantage of all the provided functionalities, it is necessary to create a CPLEXsolver object and to set the problem: this is done by specifying all the required parameters to set up the solver, i.e., the number of variables within the objective function, the number of equality and inequality constraints and the type of solver to be used for the optimization. After the initialization, it is possible to specify through the `setProblem` method the objective function and the constraints matrices.

The objective function has to be specified using two elements: the hessian, which represents the quadratic part of the problem, and the gradient, which represents the linear part. Regarding the constraints it is necessary to specify distinctly the equality and inequality constraints since they are handled differently: for the equality constraints we use the `IloRange` method provided by CPLEX which allows to specify both lower and upper bounds, while for the inequality constraints we use the `setUB` method, which specifies only the upper bound.

After setting up the problem it is possible to call the `solveProblem` method to launch the execution. To retrieve the output, CPLEX provides two functionalities: one to get the resulting matrix and one to get the optimizer status. The optimizer status is useful to interpret the output, the main values are the following:

- optimal solution: the resulting matrix contains the optimal values for the problem;

- feasible solution: the resulting matrix contains a feasible solution which is not optimal;

- unfeasible solution: the resulting matrix contains an unfeasible solution for the problem.

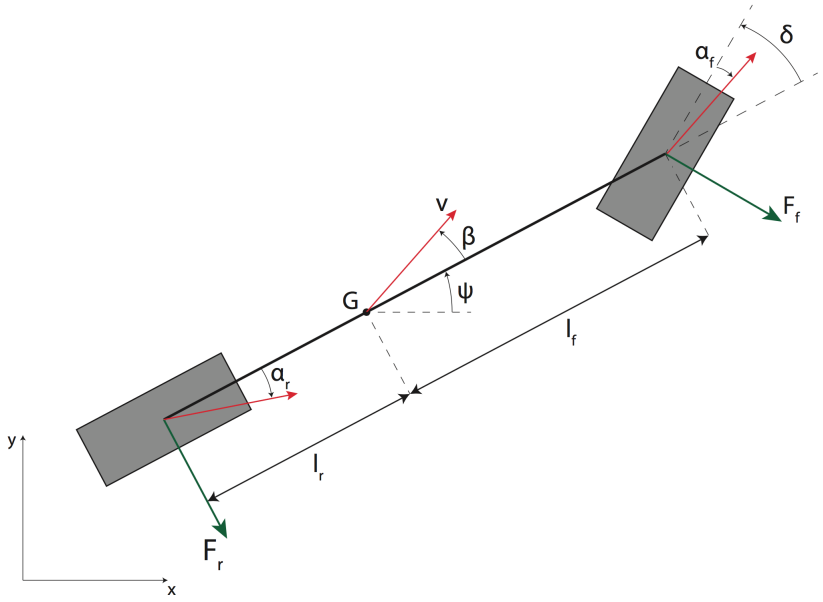


Figure 5.9: *The sideslip angle β in the single-track model.*

5.5 Sideslip angle estimation

In vehicle dynamics, the sideslip angle (shown in Figure 5.9 as β) is the angle between a rolling wheel's actual direction of travel and the direction towards which it is pointing. In a real life scenario, to have a direct measurement of vehicle sideslip angle, a complex and extremely expensive equipment is required, which cannot be considered a suitable solution. Therefore, in order to obtain this crucial variable we implemented a sideslip angle observer, originally developed in another thesis [11], which combines the available measurements with a static or dynamic model to estimate the unknown quantity.

The basic idea of the estimation process is that of linearizing the relationship between positions and accelerations in a unicycle model of the vehicle through the feedback linearization so that, from an external point of view, it is possible to represent the system as a chain of integrators. This methodology is represented in Figure 5.10.

The kinematic model of a unicycle model for the single-track is:

$$\begin{cases} \dot{x} = v \cdot cos(\psi + \beta) \\ \dot{y} = v \cdot sin(\psi + \beta) \\ \dot{\psi} + \dot{\beta} = w \end{cases} \tag{5.34}$$

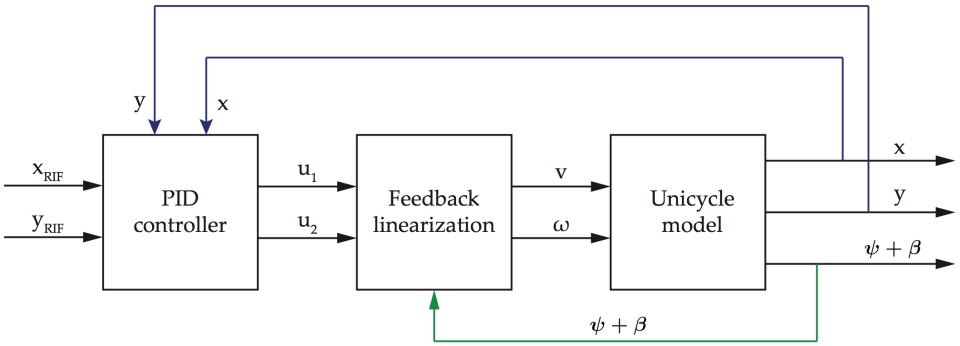


Figure 5.10: *Methodology for sideslip angle estimation through feedback linearization.*

while the feedback linearization is:

$$\begin{cases} \dot{\vartheta} = a_x \cdot cos(\psi + \beta) + a_y \cdot sin(\psi + \beta) \\ w = \dfrac{a_y \cdot cos(\psi + \beta) - a_x \cdot sin(\psi + \beta)}{v} \end{cases} \tag{5.35}$$

where x and y are the Cartesian coordinates of the vehicle, the term ψ and β are the yaw angle and the sideslip angle, v is the driving velocity and ω the steering velocity.

In order to perform the integral calculus we took advantage again of ODEINT [46], a C++ library for numerically solving ordinary differential equation [5]. With this library it is possible to set the initial time and end time of the integration and, furthermore, to select which kind of stepper to use. In order to have a dynamic step size we decided to use a *runge_kutta_dopri5* stepper, which dynamically changes the size of the integration.

The estimation is performed every time the position of the vehicle is available and the time of the integration is set to the difference between the timestamps of the last two poses. The final result of the estimation is the sum of yaw and sideslip angles of the vehicle; in order to calculate the sideslip angle the node subtracts from the estimated angle the actual yaw angle of the vehicle read on topic **/yaw_angle**.

Chapter 6

Experimental results

In this chapter we provide the results obtained from the experiments done both with ODE and Gazebo simulations. In the first section we present the tests performed to estimate the real parameters of the vehicle used in Gazebo in order to feed the MPC and to tune the ODE simulation. Then we present the localization experiments done driving the vehicle manually, both with and without sensors noise. As first test for the architecture we provide the results obtained following a single point trajectory with and without an obstacle, with three different simulation profiles: ODE based, Gazebo and Gazebo integrated with ROAMFREE. The last section contains the results achieved using a multi-point trajectory.

6.1 Parameters estimation

In order to use the MPC with the reference vehicle it has been necessary to configure the controller with the real parameters; this is requested also when dealing with a real vehicle. Some of these parameters were already available in the configuration file of the Polaris Gazebo model, while for others an estimation has been performed. In this section we present the procedure we followed for the estimates.

6.1.1 Sideslip angle

The sideslip angle estimate described in Section 5.5 has been proved suitable for the ODE based simulator. Figure 6.1 shows a comparison between the angle estimated by the observer, represented by the blue line, and the one calculated with the single-track equations, represented by the red line. It is possible to notice how the two angles are almost overlaid for all the

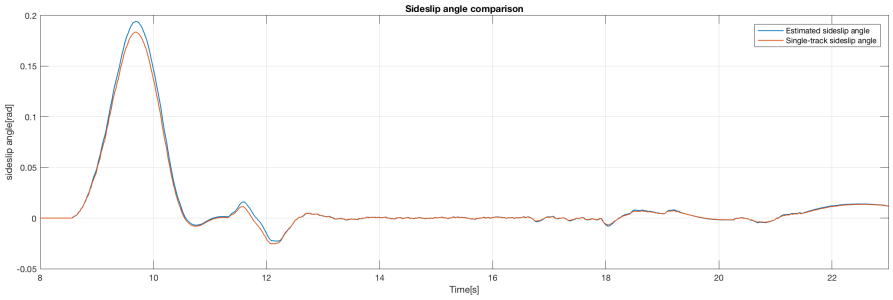


Figure 6.1: *Observer and single-track sideslip angle comparison.*

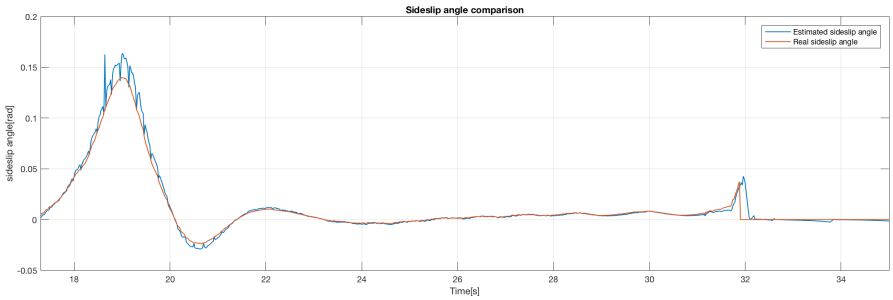


Figure 6.2: *Observer and real sideslip angle comparison.*

simulation long. Tests performed with both values turned out to have the same behavior.

Despite the previous result, the observer has not guaranteed the desired behavior with the Gazebo simulator. Although the values of the angles are quite similar, as shown in Figure 6.2, the presence of slower actuators led to an undesired behavior of the feedback linearization causing difficulties in steering variations. The blue line represents the estimated sideslip angle, while the red one the actual sideslip angle.

To deal with this situation we decided to calculate the sideslip angle in a different manner, i.e., taking advantage of the information provided by Gazebo about the velocities over x-axis and y-axis exploiting the following relation:

$$\beta = \psi - arctan\frac{v_y}{v_x}, \tag{6.1}$$

where ψ is the yaw angle of the vehicle.

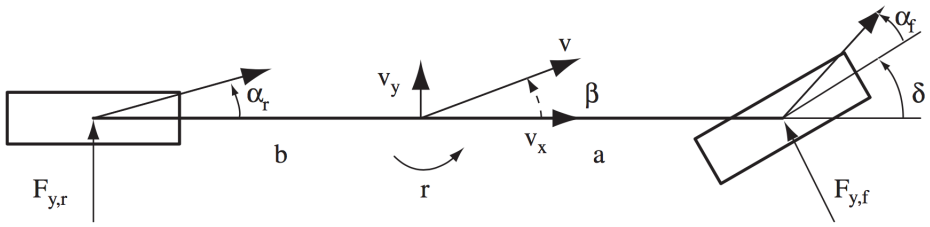


Figure 6.3: *Single-track model.*

6.1.2 Cornering stiffness

Cornering stiffness is the ratio of cornering force over slip angle. Since the usage of our MPC necessitates the identification of the cornering stiffness of the Polaris vehicle and there was no information about this parameter we needed to calculate them. This was accomplished, as detailed below, following the method illustrated in [14].

Empirical tyre curves are generated from data of a quasi-steady state ramp steer manoeuvre. In this manoeuvre, the front steer angle is slowly ($0.5\ deg/s$) increased so that the vehicle can be assumed to be in a steady state condition ($\dot{r} = \dot{v}_y = 0$). Under this assumption, the lateral acceleration a_y can be approximated as in:

$$a_y^{ss} = r v_x \tag{6.2}$$

where v_x represents the vehicle's longitudinal velocity and r the yaw rate.

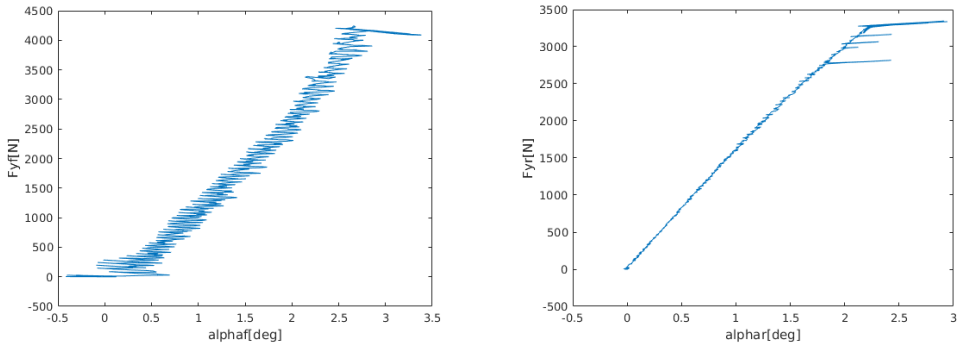
Applying this steady state approximation to the single-track model (Figure. 6.3) leads to the expressions below for the front and rear tyre lateral forces:

$$F_{y,f} = \frac{mb}{(a+b)cos\delta} v_x r = \frac{mb}{(a+b)cos\delta} a_y^{ss} \tag{6.3}$$

$$F_{y,r} = \frac{ma}{a+b} v_x r = \frac{ma}{a+b} a_y^{ss} \tag{6.4}$$

where m is the vehicle mass, a the distance from the vehicle CoG to the front axle, b the distance from the vehicle CoG to the rear axle, $F_{y,f}$ the front tyre lateral force, $F_{y,r}$ the rear tyre lateral force, v_y the lateral velocity and δ is the front tyre steer angle.

By using Equations 6.3 and 6.4 to calculate the tyre lateral forces and the relations in Equations 6.5 and 6.6 to calculate the tyre slip angles, it is possible to generate experimental tyre curves like the ones given by the scatter plots in Figure 6.4a and 6.4b.



(a) **(b)**

Figure 6.4: *Empirical tyre curves: (a) front and (b) rear.*

Front cornering stiffness	$C_{\alpha,f}$	57000	(N/rad)
Rear cornering stiffness	$C_{\alpha,r}$	91700	(N/rad)

Table 6.1: Tyre parameter estimates.

$$\alpha_f = arctan\frac{v_y + ar}{v_x} - \delta, \tag{6.5}$$

$$\alpha_r = arctan\frac{v_y - br}{v_x}. \tag{6.6}$$

By means of the relation in Equation 6.7, from these curves it is possible to calculate the value of the cornering stiffness shown in Table 6.1 for the vehicle in the Gazebo simulation.

$$F_y = C_\alpha \alpha \tag{6.7}$$

6.2 Localization

In this section we present the results obtained testing the localization module using at the beginning the ROAMFREE library alone and then coupling it with the **fastPredictor** node. The considered scenarios take into account, at first, ideal sensors and then sensors affected by noise in order to better simulate a real life situation. The tests were performed driving the vehicle manually following an eight-shape trajectory.

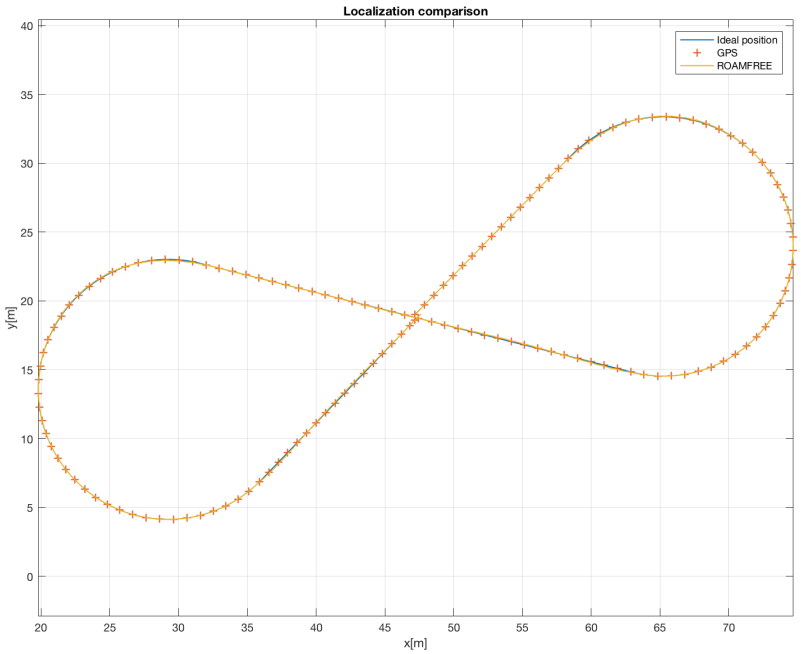


Figure 6.5: *Trajectory comparison among ideal position, GPS and ROAMFREE.*

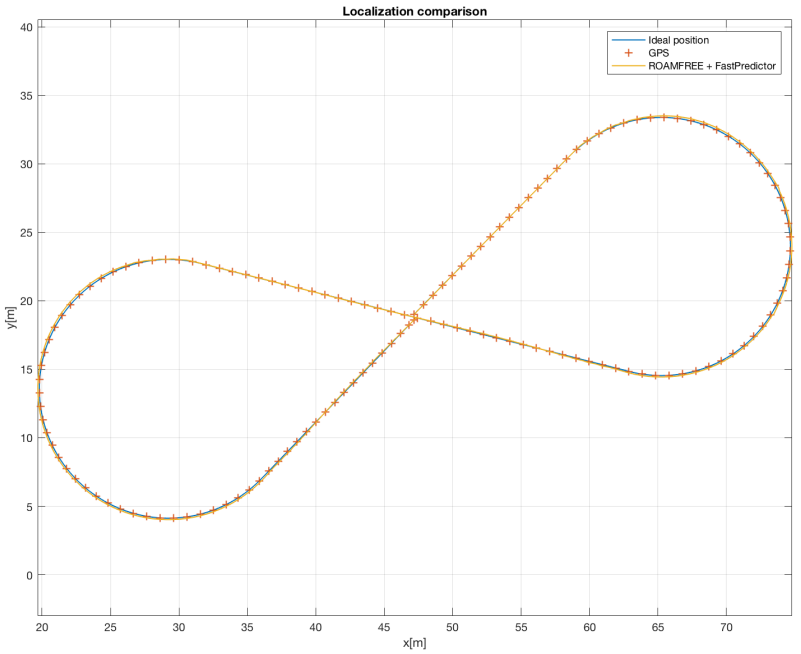
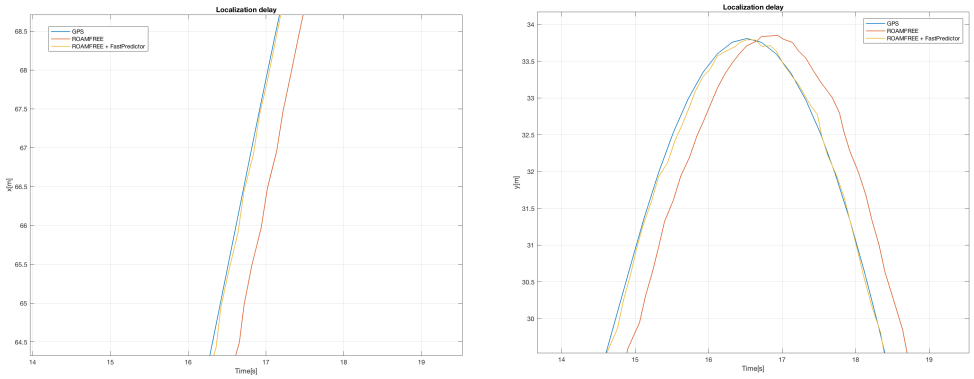


Figure 6.6: *Trajectory comparison among ideal position, GPS and ROAMFREE with fastPredictor.*

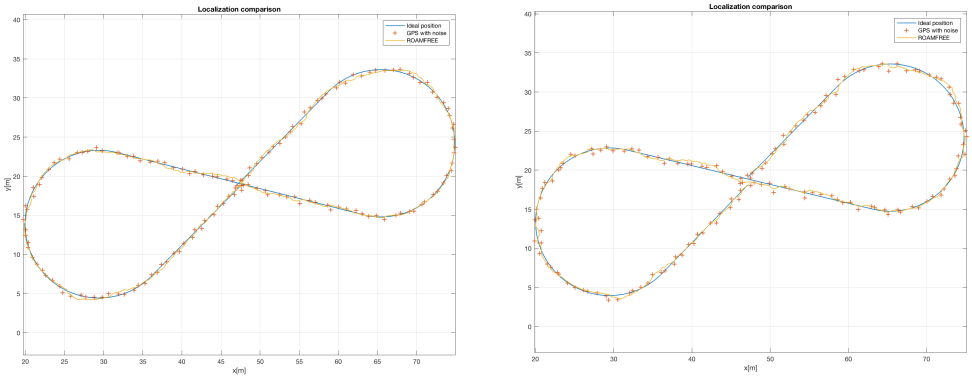


(a) (b)

Figure 6.7: *Localization delay and its compensation on x (a) and y (b) coordinates.*

In Figure 6.5 and 6.6 are shown the results obtained with ideal sensors. The blue line represents the followed path, the orange crosses represent the GPS measurements and the yellow line represents the estimated pose. It is possible to notice how the pose estimated by ROAMFREE matches with a certain degree of precision the real one highlighting the accuracy of this library in ideal situations. In the second image it is presented the estimation calculated coupling ROAMFREE with the **fastPredictor** node; also in this case the estimated pose is highly accurate. This is not surprising since the **fastPredictor** node just takes in input the pose estimated by ROAMFREE and integrates the odometry measurements to obtain an higher frequency estimation. The advantage of this configuration is represented by the delay compensation since the ROAMFREE estimated pose presents a delay due to the computational time. The compensation is performed integrating in the future the odometry measurements starting from the last ROAMFREE estimated pose. Figure 6.7a and 6.7b show how this delay is compensated by the **fastPredictor**. The blue line represents the followed path, the orange line represents the pose estimated by ROAMFREE, while the yellow one represents the pose estimated by the **fastPredictor**.

In Figure 6.8a and 6.8b two similar scenarios are shown but using sensors affected by Gaussian noise; GPS is affected by Gaussian noise with a standard deviation of $0.1m$ for each axis in the first case and $0.3m$ for each axis in the second case, while IMU accelerometer is affected by Gaussian noise with a standard deviation of $0.1m/s^2$ for each axis. It is possible to observe how the noise affects the estimation and how bigger



(a) (b)

Figure 6.8: *Trajectory comparison among ideal position, GPS and ROAMFREE with fastPredictor.*

noises lead to a worse estimation.

ROAMFREE, together with the position, estimates the orientation too, so we thought it was opportune to make a comparison between the actual orientation and the estimated one. In Figure 6.9, the blue line represents the real yaw angle of the vehicle, while the orange line represents the yaw angle estimated by ROAMFREE. It is possible to notice how the estimation is accurate, with a small delay ($< 250ms$) due the computational time. This delay is conform to the one resulting from the position estimation. Coupling to ROAMFREE the **fastPredictor** allows to compensate this delay.

6.3 Regulation problem

In this section we describe the results obtained through a single-point trajectory. For each test the target position we want the vehicle to reach is $goal = [30, 30]^T$ with null initial velocity and starting position in $[0, 0]^T$. In the first part we provide the ODE simulation experiment, while, in the second one, we focus on the experiments performed with Gazebo. In both cases we present different scenarios, with and without an obstacle. The obstacle, with radius of 2.5 meters, has been placed in $[15, 12.5]^T$.

6.3.1 ODE based simulator

The first scenario we want to analyze is the one performed with the ODE simulator. In Figure 6.10 the trajectories followed by the vehicle are shown;

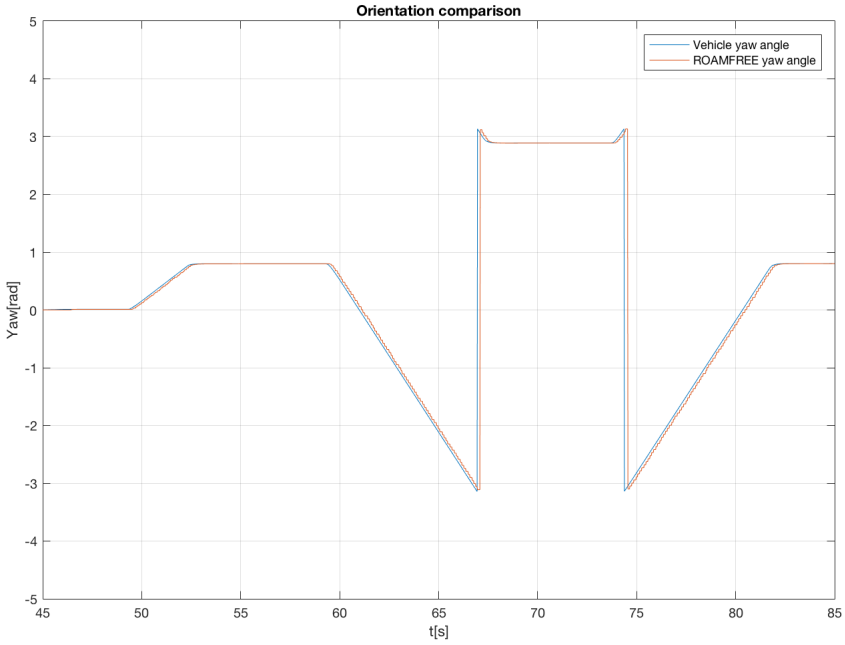


Figure 6.9: *Yaw comparison between ideal and ROAMFREE trajectories.*

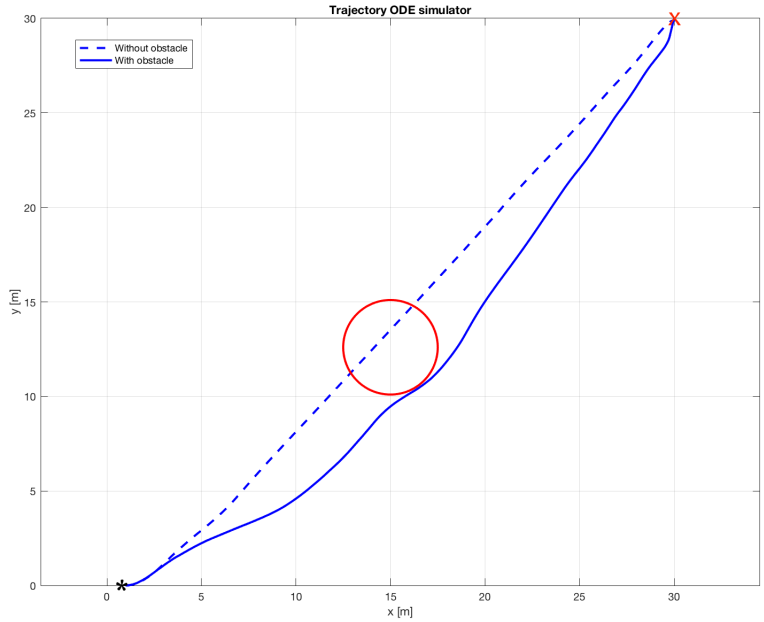


Figure 6.10: *Trajectory of the vehicle with and without obstacle.*

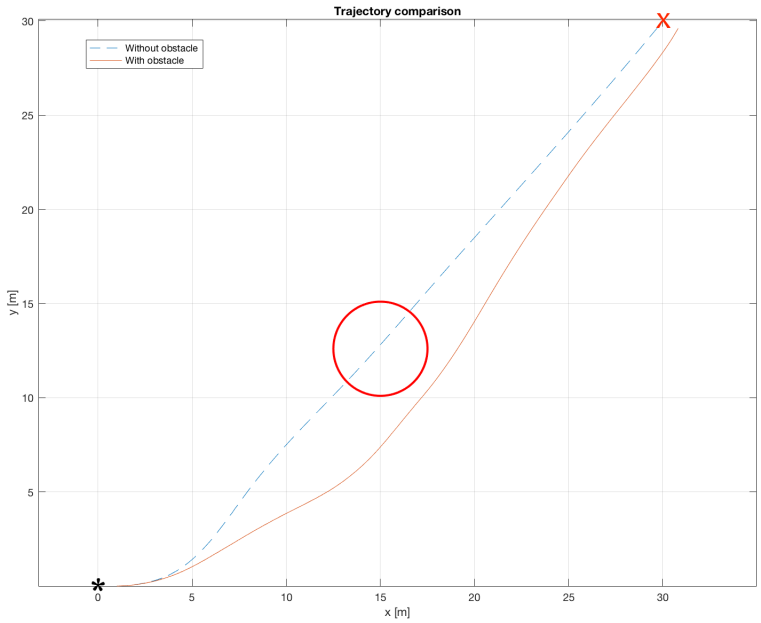


Figure 6.11: *Trajectory of the vehicle with and without obstacle in Gazebo.*

the dashed line represents the path that the car would follow if no obstacle was placed, while the solid line represents the actual path adopted to avoid the obstacle. The initial position is indicated by a black asterisk while the goal is marked with a red cross. It is therefore possible to observe how big is the contribution of the obstacle avoidance constraint in the definition of the trajectory. In fact due to the obstacle, the vehicle changes its original path since the very beginning of its trajectory.

6.3.2 Gazebo

In this section we present the results with a single-point trajectory using Gazebo. As previously stated, we analyze two different situations, one with the presence of an obstacle and one without.

The experiments were performed using different setups: in the first one the localization is performed by Gazebo, in the second one by ROAMFREE, while, in the third one with the integration of the **fastPredictor** node. In the last two scenarios we made the same tests using both ideal and noisy sensors.

In Figure 6.11 the trajectories obtained using the ideal localization are shown. The dashed line represents the path followed without the obstacle, while the solid one represents the path followed in presence of the obstacle.

In Figure 6.12a and 6.12b the trajectories obtained using ROAMFREE are shown. In this case we decided to compare the results when using ideal and noisy sensors, in order to better simulate a real life situation; GPS is affected by Gaussian noise with a standard deviation of $0.1m$ for each axis in the first test and $0.3m$ in the second case, while IMU accelerometer is affected by Gaussian noise with a standard deviation of $0.1m/s^2$ for each axis. The blue line represents the ideal situation, the yellow line represents the noisy one, while the red crosses represents the noisy GPS measurements. It is possible to notice how different estimates take to different trajectories. In particular the noise leads to a worse path since the MPC takes as input the position estimated, so for different inputs the final trajectories turn out to be distinct. In the second image it is possible to notice how a worse GPS signal affects both the estimate and the final path; also in this case, the MPC is still able to find a sequence of actions to reach the final point.

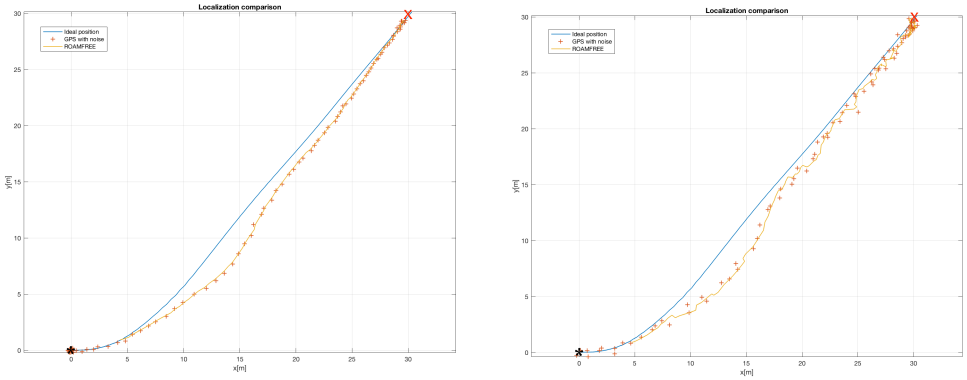
In Figure 6.13 is presented a similar scenario, i.e., the comparison between the trajectories with and without noise, but with the presence of an obstacle. In Figure 6.13a the dotted line represents the followed path using ideal sensors, while the other lines represent five different tests performed using the same level of noise. The aim of this comparison is to show how the MPC is influenced by noise starting from the same initial conditions; in all cases the MPC is able to drive the vehicle to the final goal. In Figure 6.13b we show, just for a single test, the GPS measurements compared with the followed path.

Lastly, Figure 6.14 compares all three localization methods in a noisy scenario. The blue line represents the trajectory followed using Gazebo localization, the orange line represents the path followed using ROAMFREE and the dotted lines the ones with the **fastPredictor** node. It is possible to notice how the dotted lines generally represent a compromise between the ideal localization and the ROAMFREE one. This result is obtained thanks to the higher frequency of the fastPredictor and its better accuracy.

6.4 Trajectory

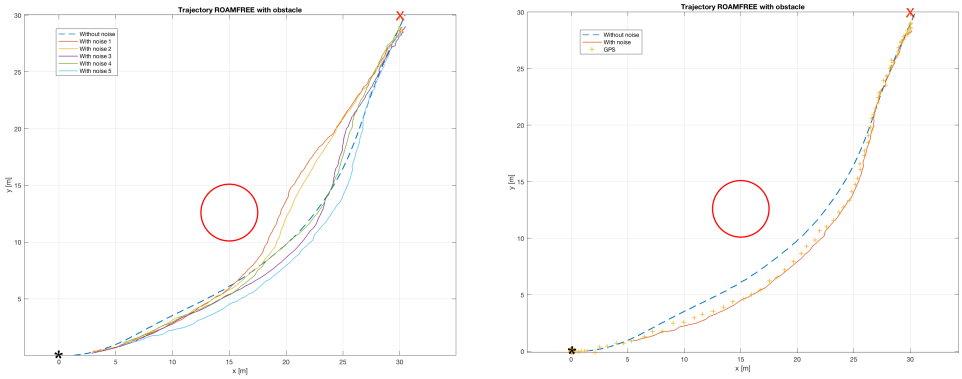
The structure of the original MPC was not thought to implement a path follower: it would have been necessary to modify the MPC itself, in particular its cost function and the constraints that bound the controller to reach a goal with null final velocity and free orientation, but this goes beyond the scope of this thesis.

In this section we present the results obtained through a waypoint



(a) **(b)**

Figure 6.12: *Trajectories of the vehicle with ROAMFREE with and without noise: GPS std <0.1, 0.1, 0.1> (a), GPS std <0.3, 0.3, 0.3> (b).*



(a) **(b)**

Figure 6.13: *Trajectories of the vehicle with ROAMFREE with and without noise (with obstacle): comparison of various tests(a) and GPS noisy signal of one the tests (b).*

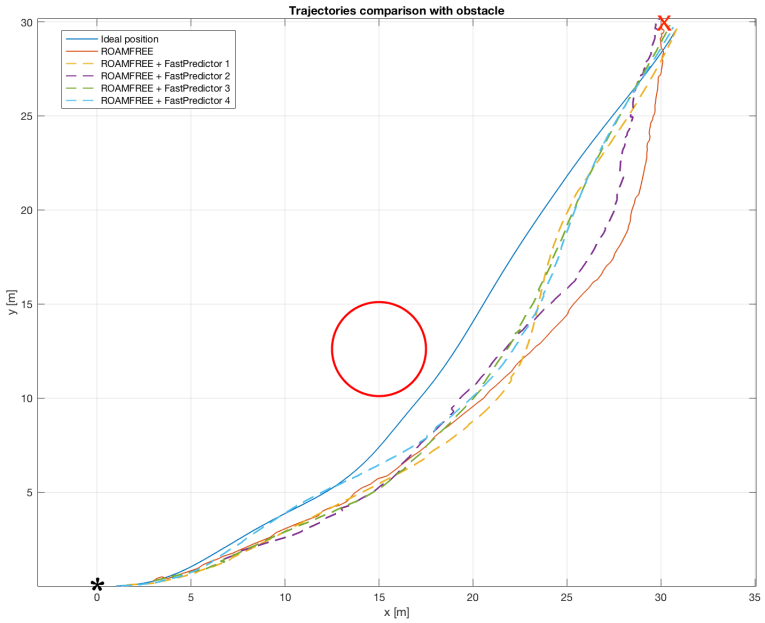


Figure 6.14: *Trajectories comparison among the ideal situation in Gazebo, ROAMFREE affected by noise and ROAMFREE with fastPredictor affected by noise.*

changing trajectory, i.e., using a sequence of long-range goals. This has been made possible adding a rule to the **MPC** node, so as to update the goal as soon as the vehicle reaches a distance that is less than 2 meters far from the current goal. In the following we present the results obtained with both ODE and Gazebo simulators.

In these scenarios we want the vehicle to follow a waypoint trajectory composed by these points: $goal_1 = [10, 0]^T$, $goal_2 = [30, 10]^T$, $goal_3 = [40, 10]^T$, $goal_4 = [60, 0]^T$, $goal_5 = [70, 0]^T$, $goal_6 = [90, 10]^T$, $goal_7 = [100, 10]^T$, $goal_8 = [120, 0]^T$ with the initial position of the vehicle set to $[0, 0]^T$ and null velocity.

6.4.1 ODE based simulator

In this first part we present the results obtained with ODE simulator. Figure 6.15 shows the trajectory followed by the vehicle. The initial point is indicated by a black asterisk while the goals that are to be reached are marked with red circles.
From the figure it is possible to notice how the trajectory does not cross exactly the goals, because the next goal is updated when the current goal is almost reached as stated above.

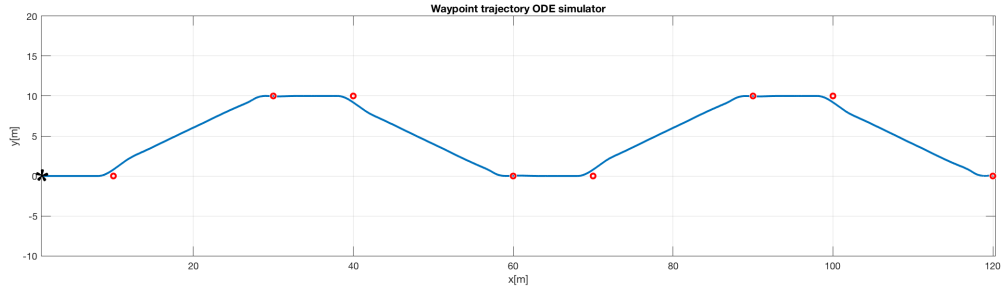


Figure 6.15: *A waypoint trajectory followed by the ODE simulator.*

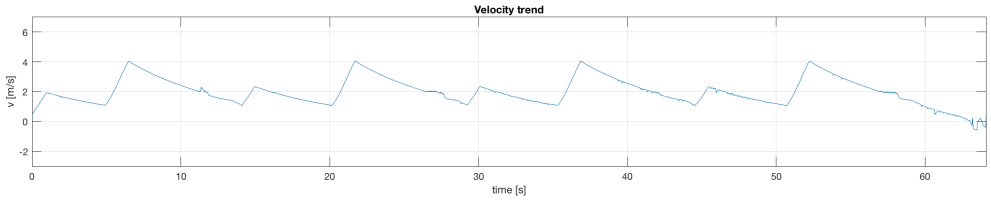


Figure 6.16: *Velocity trend.*

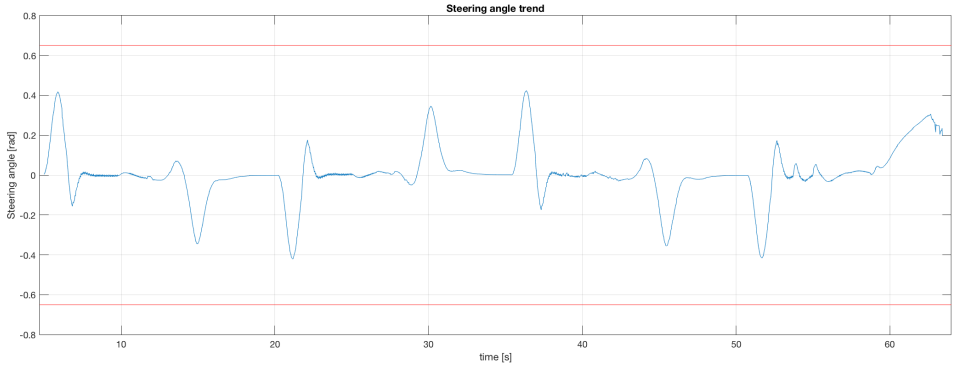


Figure 6.17: *Steering angle trend.*

In Figure 6.16 and 6.17 velocity and steering angle trends are shown. From the first image it is possible to notice how the velocity trend changes in the updating-goal and in the reaching-goal phases: the more it gets close to the goal the more the velocity decreases, because of the zero terminal constraint. In the second image it is possible to observe the steering angle trend, which never exceeds its bounds ($\pm 0.65 rad$), delimited by the red lines.

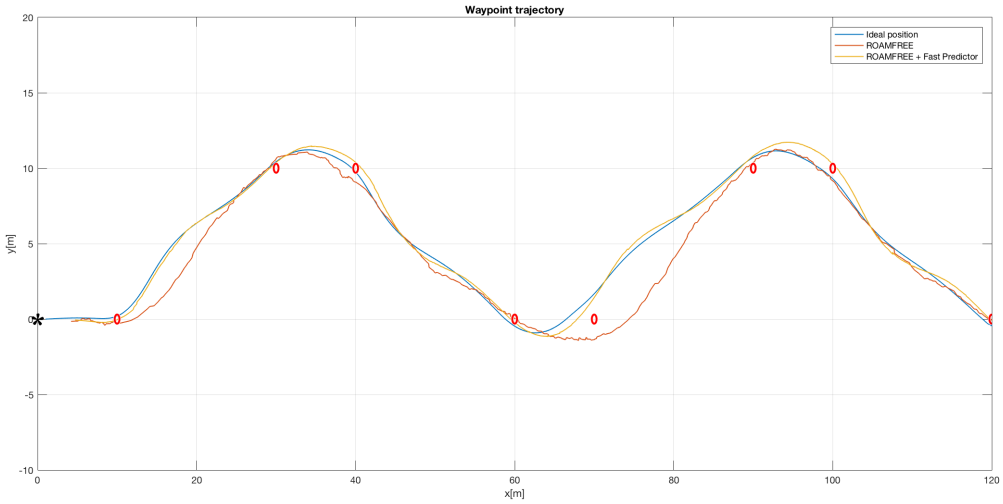


Figure 6.18: *Trajectories comparison in Gazebo.*

6.4.2 Gazebo

In this section we present the results about the trajectory obtained using Gazebo simulator. Three kind of experiments have been performed, the main difference among them is given by the localization method used. At the beginning we used the localization offered by Gazebo, then the one provided by ROAMFREE and lastly the one coupling ROAMFREE and the fastPredictor. In the last two cases the position is estimated using noisy sensors, since we thought they can represent a more realistic scenario. GPS is affected by Gaussian noise with a standard deviation of $0.1m$ for each axis, IMU accelerometer is affected by Gaussian noise with a standard deviation of $0.1m/s^2$ for each axis.

In Figure 6.18 we show the obtained results. The initial point is indicated by a black asterisk while the goals that are to be reached are marked with red circles. The blue line represents the trajectory followed using the ideal localization provided by Gazebo, the orange line represents the trajectory followed using ROAMFREE localization and the yellow line the one obtained using the fastPredictor.

As previously described, also in this case not every goal is crossed due to the goal-updating rule. The reference trajectory is given by the blue line since it represents the ideal scenario. Observing the orange line it is possible to notice how the trajectory is influenced by the estimated position, since the sensors suffer from a certain noise. In fact even though ROAMFREE is able to estimate a reasonable position, the obtained trajectory is hardly affected, although the vehicle is still able to reach the final goals. The

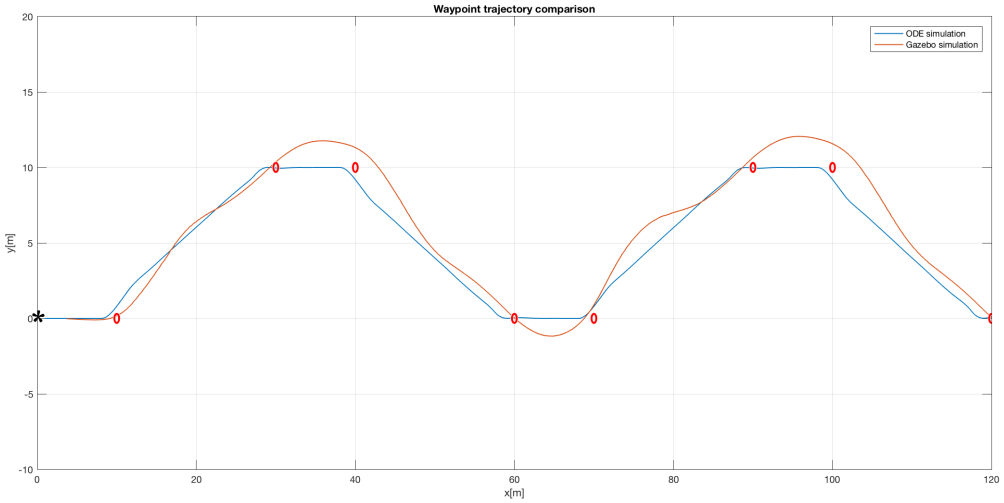


Figure 6.19: *Ideal trajectories comparison.*

yellow line represents a compromise between the previous ones: it is not as good as the ideal trajectory, but there is a notable improvement with respect to the one generated using only ROAMFREE. This is because the pose estimation is performed at a much higher frequency and because the odometry integration allows the controller to work with a more precise position.

6.4.3 Ideal comparison

In this section we want to compare the two ideal simulation, i.e., the one performed with ODE and the one performed with Gazebo using its own localization method. The aim of this comparison is to highlight the differences in the behavior of the controller switching from the single-track model to the real one. To make this experiment we took advantage of the same trajectory we already used previously.

In Figure 6.19 the blue line represents the ODE trajectory, while the orange line the Gazebo one. It is possible to notice how the first path appears to be less smooth compared to the second one; this is because in the first case the steering variation happens instantaneously, while in the latter the steering variation requires more time, due to a real vehicle dynamic. This leads also to a reduced accuracy in the final path.

Chapter 7

Conclusions and future work

The result of this work is a flexible and modular Model Predictive Control architecture developed with ROS that includes localization and static obstacle avoidance. All the modules have been developed and tested independently. First of all, the proposed system has been tested with a single-track simulator; the performance is satisfactory and the vehicle reaches its target position without violating the considered constraints. After that, we have used a more accurate model to describe the vehicle using a simulated *Polaris Ranger XP 900* within the Gazebo environment. This allowed us to test the localization module and the controller behavior in a more realistic situation. This architecture has proven effective during the experiments, obtaining good results in both localization and single-point trajectory generation although some limitations are clear from the actuator dynamics.

7.1 Future work

One of the first functionalities that could be added to the architecture is the obstacle detection. Since the implemented Model Predictive Control handles obstacles as polytopes, in order to add this feature it would be necessary to equip the vehicle with at least a laser scanner and to develop a new module able to detect objects and to model them as polytopes to be sent to the MPC node.

The next step could be to modify the controller to add the path following functionality; in order to do so it would be necessary to modify the cost function and the MPC constraints. For instance the zero terminal constraint should be modified and substituted with constraints regarding velocity and orientation that the vehicle should assume for each point.

It is also possible to improve the system behavior working on the localization module. Being inspired by other well-known unmanned vehicle projects, the vehicle could be equipped with more sensors, such as one or more laser scanners and cameras. Moreover the vehicle odometry measurements could be added to this module. All this additions could allow the system to compensate in a more efficient manner the lack of GPS measurements.

Once added all the previously specified functionalities, the architecture would be ready to be implemented on a real vehicle. In order to do so, the vehicle should be equipped with an actuation system able to control the handwheel and the gas/brake pedals. Furthermore it would be necessary to implement a new module to handle the communication between the controller and the actuators.

Bibliography

[1] *5 big challenges that self-driving cars still have to overcome*. Apr. 2016. URL: http://www.vox.com/2016/4/21/11447838/self-driving-cars-challenges-obstacles (cit. on p. 8).

[2] *7 Benefits of Self-Driving Cars*. Mar. 2016. URL: http://blog.mydomino.com/7-benefits-self-driving-cars/ (cit. on p. 7).

[3] M. Bertozzi A. Broggi and A. Fascioli. "The 2000 km test of the argo vision-based autonomous vehicle". In: *IEEE Intelligent Systems* (1999), pp. 55–64 (cit. on p. 12).

[4] *About ROS*. URL: http://www.ros.org/about-ros/ (cit. on p. 24).

[5] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison Wesley Professional, Dec. 2004 (cit. on pp. 43, 86).

[6] C. Urmson; C. Ragusa; D. Ray; J. Anhalt; D. Bartz; T. Galatali; A. Gutierrez; J. Johnston; S. Harbaugh; W. Messner et al. "A Robust Approach to High-Speed Navigation for Unrehearsed Desert Terrain". In: *Journal of Field Robotics* 23.8 (2006), pp. 467–508 (cit. on pp. 12, 13).

[7] Changbin Hu et al. "Energy Coordinative Optimization of Wind-Storage-Load Microgrids Based on Short-Term Prediction". In: *Energies* 8.2 (2015), pp. 1505–1528 (cit. on p. 72).

[8] Changbin Hu et al. "Energy Coordinative Optimization of Wind-Storage-Load Microgrids Based on Short-Term Prediction". In: *Energies*. Vol. 8. 2. 2015, pp. 1505–1528 (cit. on p. 49).

[9] S. Thrun; M. Montemerlo; H. Dahlkamp; D. Stavens; A. Aron; J. Diebel; P. Fong; J. Gale; M. Halpenny; G. Hoffmann et al. "Stanley: The robot that won the DARPA grand challenge". In: *Journal of Field Robotics* 23.9 (2006), pp. 661–692 (cit. on p. 12).

[10] C. Baker and J. Dolan. "Traffic Interaction in the Urban Challenge: Putting Boss on its Best Behavior". In: *Intelligent Robots and Systems, 2008*. IEEE, 2008 (cit. on p. 20).

[11] M. Spaliviero; L. Bascetta and M. Farina. "Control of an autonomous all-terrain vehicle with Model Predictive Control". MA thesis. Politecnico di Milano, 2016 (cit. on pp. 73, 85).

[12] Jan F. Broenink. "20-sim software for hierarchical bond-graph/block-diagram models". In: *Simulation Practice and Theory*. Vol. 7. 5. 1999, pp. 481–492 (cit. on p. 38).

[13] Eduardo F. Camacho and C. Bordons Alba. *Model Predictive Control*. Springer-Verlag London, 2007 (cit. on p. 67).

[14] Rami Y. Hindiyehb Christoph Vosera and J. Christian Gerdes. "Analysis and control of high sideslip manoeuvres". In: *Vehicle System Dynamics* 48 (Mar. 2010), pp. 317–336 (cit. on p. 89).

[15] A. Broggi; P. Medici; P. Zani; A. Coati and M. Panciroli. "Autonomous vehicles control in the VisLab Intercontinental Autonomous Challenge". In: *Annual Reviews in Control* 36.1 (Apr. 2012), pp. 161–171 (cit. on p. 15).

[16] IBM ILOG CPLEX. "V12. 1: Users manual for CPLEX". In: *International Business Machines Corporation*. Vol. 46. 53. 2009, p. 157 (cit. on p. 71).

[17] D. Cucci and M. Matteucci. "A flexible framework for mobile robot pose estimation and multi-sensor self-calibration". In: *Proceedings of the 10th International Conference on Informatics in Control, Automation and Robotics - ICINCO* 2 (Jan. 2013), pp. 361–368 (cit. on p. 57).

[18] *DARPA Grand Challenge: overview*. 2007. URL: http://archive.darpa.mil/grandchallenge05/overview.html (cit. on p. 9).

[19] *DARPA Robotics Challenge Finals: Rules and Course*. 2015. URL: http://spectrum.ieee.org/automaton/robotics/humanoids/drc-finals-course (cit. on p. 33).

[20] M. Maurer; R. Behringer; S. Furst; F. Thomanek; E.D. Dickmanns. "A compact vision system for road vehicle guidance". In: *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*. Vol. 3. IEEE, 1996, pp. 313–317 (cit. on p. 11).

[21] *Eigen*. URL: http://eigen.tuxfamily.org/index.php?title=Main_Page (cit. on p. 82).

[22] A. Bacha; C. Bauman; R. Faruque; M. Fleming and C. Terwelp et al. "Odin: Team VictorTango's Entry in the DARPA Urban Challenge". In: *Journal of Field Robotics* 25.8 (2008), pp. 467–492 (cit. on p. 14).

[23] ZJ Chong; Baoxing Qin; Tirthankar Bandyopadhyay; Marcelo H. Ang; Emilio Frazzoli and Daniela Rus. "Synthetic 2d lidar for precise vehicle localization in 3d urban environment". In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1554–1559 (cit. on p. 36).

[24] *Gazebo*. URL: http://gazebosim.org (cit. on p. 40).

[25] A. Broggi; S. Debattisti; P. Grisleri and M. Panciroli. "The deeva autonomous vehicle platform". In: *Intelligent Vehicles Symposium (IV), 2015 IEEE* (Aug. 2015) (cit. on p. 17).

[26] S. Harmon. "The Ground Surveillance Robot (GSR): An Autonomous Vehicle Designed to Transit Unknown Terrain". In: *IEEE Journal on Robotics and Automation* (July 1987), pp. 266–279 (cit. on p. 11).

[27] David Gossow; Adam Leeper; Dave Hershberger and Matei Ciocarlie. "Interactive Markers: 3-D User Interfaces for ROS Applications". In: *IEEE Robotics & Automation Magazine* 18.4 (2011), pp. 14–15 (cit. on p. 27).

[28] Dag Brück Hilding Elmqvist and Martin Otter. *Dymola-user's manual*. Dynasim AB, 1996 (cit. on p. 38).

[29] Fumio Kanehiro; Hirohisa Hirukawa and Shuuji Kajita. "Openhrp: Open architecture humanoid robotics platform". In: *The International Journal of Robotics Research* 23.2 (2004), pp. 155–165 (cit. on p. 38).

[30] *How Google's Self-Driving Car Works*. Oct. 2011. URL: http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/how-google-self-driving-car-works (cit. on p. 18).

[31] *Humans are the main obstacle to the driverless revolution*. Aug. 2016. URL: https://www.ft.com/content/e961f914-6ba3-11e6-ae5b-a7cc5dd5a28c (cit. on p. 9).

[32] *Infiniti Q50*. 2016. URL: https://www.infiniti.it/cars/new-cars/q50.html (cit. on p. 16).

[33] Alberto Isidori. *Nonlinear control systems*. Springer Science & Business Media, 2013 (cit. on p. 78).

[34] N. Koenig and A. 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*. Vol. 3. IEEE, 2004, pp. 2149–2154 (cit. on p. 37).

[35] G. Conte M. Bertozzi A. Broggi and A. Fascioli. "Vision-based Automated Vehicle Guidance: the experience of the ARGO vehicle". In: *IAPRVA - Tecniche di Intelligenza Artificiale e Pattern Recognition per la Visione Artificiale* (Apr. 1998), pp. 35–40 (cit. on p. 11).

[36] Lalo Magni and Riccardo Scattolini. *Complementi di controlli automatici*. Pitagora, 2006 (cit. on p. 67).

[37] *Math 407A: Linear Optimization*. URL: https://www.math.washington.edu/~burke/crs/407/lectures/L4-lp_standard_form.pdf (cit. on p. 73).

[38] G. Bardaro; M. Matteucci and D. Cucci. "High Level Control Architecture and Dynamic Simulation for an Autonomous All Terrain Robot". MA thesis. Politecnico di Milano, 2014. Chap. 4.5.3 (cit. on p. 65).

[39] O. Michel. "Webots: a powerful realistic mobile robots simulator". In: *Proceedings of the Second International Workshop on RoboCup, LNAI*. Springer, 1998 (cit. on p. 38).

[40] *MIT DGC: Technology*. URL: http://grandchallenge.mit.edu/technology.shtml (cit. on p. 14).

[41] Linda K. Moore. *Spectrum Needs of Self-Driving Vehicles*. Feb. 2015. URL: https://www.fas.org/sgp/crs/misc/IN10168.pdf (cit. on p. 8).

[42] Hans P. Moravec. *Obstacle avoidance and navigation in the real world by a seeing robot rover*. Tech. rep. Robotics Institute Carnegie-Mellon University Pittsburgh, Pennsylvania 15213, 1980 (cit. on p. 9).

[43] Hans P. Moravec. "The Stanford Cart and the CMU Rover". In: *Autonomous Robot Vehicles*. Springer New York, 1990, pp. 407–419 (cit. on p. 9).

[44] *Most Advanced Robotics Simulation Software Overview*. Mar. 2016. URL: https://www.smashingrobotics.com/most-advanced-and-used-robotics-simulation-software/ (cit. on p. 37).

[45] Nils J. Nilsson. *Shakey The Robot*. Tech. rep. AI Center, SRI International, 1984 (cit. on p. 9).

[46] *ODEINT*. 2012. URL: http://headmyshoulder.github.io/odeint-v2/ (cit. on p. 86).

[47] *ODEINT: All examples*. 2012. URL: http://headmyshoulder.github.io/odeint-v2/doc/boost_numeric_odeint/tutorial/all_examples.html (cit. on p. 45).

[48] *Optimization model development toolkit for mathematical and constraint programming*. URL: http://www-03.ibm.com/software/products/en/ibmilogcpleoptistud (cit. on p. 71).

[49] 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*. Springer, 2010, pp. 51–62 (cit. on p. 38).

[50] David Patterson and John L. Hennessy. "Computer Architecture, A Quantitative Approach". In: Fifth. Morgan Kaufmann Publishers, 2011. Chap. 4 (cit. on p. 82).

[51] Scott et al. Pendleton. "Autonomous Golf Cars for Public Trial of Mobility-on-Demand Service". In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2015 (cit. on p. 36).

[52] S. Harmon ; G. Bianchini ; B. Pinz. "Sensor data fusion through a distributed blackboard". In: *Robotics and Automation. Proceedings. 1986 IEEE International Conference on*. Vol. 3. IEEE, 1986, pp. 1449–1454 (cit. on p. 11).

[53] Tobias Gindele; Daniel Jagszent; Benjamin Pitzer and Rüdiger Dillmann. "Design of the planer of Team AnnieWAY's autonomous vehicle used in the DARPA Urban Challenge 2007". In: *Intelligent Vehicles Symposium*. IEEE, 2008 (cit. on p. 19).

[54] *RANGER XP® 900 EPS: Overview*. 2016. URL: http://www.polaris.com/en-us/ranger-utv/2016/ranger-xp-900-eps-velocity-blue (cit. on p. 33).

[55] Aishwarya Singh Rathore. "State-of-the-Art Self Driving Cars: Comprehensive Review". In: *International Journal of Conceptions on Computing and Information Technology Vol. 4, Issue. 1, January' 2016; ISSN: 2345 - 9808* (2016). URL: http://www.worldairco.org/IJCCIT/January2016Paper11T.pdf (cit. on pp. 6, 16).

[56] *ROS messages*. 2016. URL: http://wiki.ros.org/msg (cit. on p. 30).

[57] Michael Karg Jim Mainprice Alexandra Kirsch Séverin Lemaignan Gilberto Echeverria and Rachid Alami. "Human-robot interaction in the morse simulator". In: *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*. ACM, 2012, pp. 181–182 (cit. on p. 38).

[58] C. Armbrust; T. Braun; T. Föhst; M. Proetzsch; A. Renner; B.H. Schäfer and K. Berns. "KAT-5: Robust Systems for Autonomous Vehicle Navigation in Challenging and Unknown Terrain". In: *The 2005 DARPA Grand Challenge*. Vol. 37. Springer Berlin Heidelberg, 2007, pp. 103–128 (cit. on p. 13).

[59] E.D. Dickmanns; R. Behringer; D. Dickmanns; T. Hildebrandt; M. Maurer; F. Thomanek; J. Schiehlen. "The seeing passenger car 'VaMoRs-P'". In: *Intelligent Vehicles '94 Symposium, Proceedings of the*. Vol. 3. IEEE, 1994, pp. 68–73 (cit. on p. 11).

[60] M. Maurer; R. Behringer; D. Dickmanns; T. Hildebrandt; F. Thomanek; J. Schiehlen and E. D. Dickmanns. "VaMoRs-P: an advanced platform for visual autonomous road vehicle guidance". In: *Proceedings of SPIE - The International Society for Optical Engineering* (Jan. 1994) (cit. on p. 11).

[61] *Self-driving cars and cybersecurity: What are the risks of car hacking?* June 2016. URL: http://betanews.com/2016/06/09/self-driving-cars-and-cybersecurity-what-are-the-risks-of-car-hacking/ (cit. on p. 8).

[62] *Sense Plan Act (SPA)*. URL: http://www.education.rec.ri.cmu.edu/products/teaching_robotc_vex/reference/hp_spa.pdf (cit. on p. 29).

[63] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996 (cit. on p. 20).

[64] 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*. IEEE, 2013, pp. 1321–1326 (cit. on p. 38).

[65] *Single-passenger Mobility-support Robot "ROPITS"*. 2012. URL: http://rraj.rsj-web.org/en_atcl/1343 (cit. on p. 16).

[66] *Startup Aims to Beat Google to Market with Self-Driving Golf Cart*. 2015. URL: https://www.technologyreview.com/s/540751/startup-aims-to-beat-google-to-market-with-self-driving-golf-cart/ (cit. on p. 34).

[67] A. Geiger; M. Lauer; F. Moosmann; B. Ranft; H. Rapp; C. Stiller and J. Ziegler. "Team AnnieWAY's entry to the Grand Cooperative Driving Challenge 2011". In: (Apr. 2011) (cit. on p. 16).

[68] *Tartan Racing @ Carnegie Mellon*. 2007. URL: http://www.tartanracing.org/tech.html (cit. on p. 14).

[69] Stanford Racing Team. *Stanford's Robotic Vehicle "Junior:" Interim Report*. Tech. rep. Stanford University, Stanford, CA 94305, USA, Apr. 2007 (cit. on p. 14).

[70] *Tesla Autopilot*. 2016. URL: https://www.tesla.com/autopilot (cit. on p. 19).

[71] James W. Lowrie; Mark Thomas; Keith Gremban; Matthew Turk. "The Autonomous Land Vehicle (ALV) Preliminary Road-Following Demonstration". In: *Intelligent Robots and Computer Vision IV*. 1985 (cit. on p. 9).

[72] Berwin A. Turlach and A. Weingessel. "quadprog: Functions to solve quadratic programming problems". In: *R package version*. 2007, pp. 1–4 (cit. on p. 71).

[73] *Tutorial: ROS integration overview*. 2014. URL: http://gazebosim.org/tutorials?tut=ros_overview (cit. on pp. 37, 43).

[74] *Tutorial: Using a URDF in Gazebo*. URL: http://gazebosim.org/tutorials?tut=ros_urdf (cit. on p. 42).

[75] *U.S. Department of Transportation Releases Policy on Automated Vehicle Development*. May 2013. URL: http://www.nhtsa.gov/About-NHTSA/Press-Releases/U.S.-Department-of-Transportation-Releases-Policy-on-Automated-Vehicle-Development (cit. on p. 6).

[76] *USAD - Urban Shuttles Autonomously Driven*. URL: http://www.ira.disco.unimib.it/research/robotic-perception-research/urban-shuttles-autonomously-driven/ (cit. on p. 34).

[77] Michael Montemerlo; Jan Becker; Suhrid Bhat; Hendrik Dahlkamp; Dmitri Dolgov; Scott Ettinger; Dirk Haehnel; Tim Hilden; Gabe Hoffmann; Burkhard Huhnke; Doug Johnston; Dirk Langer; Anthony Lev; Jesse Levinson; Julien Marcil; David Orenstein; Johannes Paefgen; Isaac Penny; Anna Petrovskaya; Mike Pflueger; Ganymed Stanek; David Stavens; Antone Vogt and Sebastian Thrun. "Junior: The Stanford Entry in the Urban Challenge". In: *The DARPA Urban Challenge*. Springer Berlin Heidelberg, 2009, pp. 91–123 (cit. on p. 20).

[78] *Why ROS?* URL: http://www.ros.org/is-ros-for-me/ (cit. on p. 24).