Universidade de Aveiro Departamento de Engenharia Mecânica
2012

Joel Filipe Pereira

Estacionamento autónomo usando perceção 3D

Autonomous parking using 3D perception

Universidade de Aveiro Departamento de Engenharia Mecânica
2012

Joel Filipe Pereira

Estacionamento autónomo usando perceção 3D

Autonomous parking using 3D perception

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Mecânica, realizada sob orientação científica de Vítor Manuel Ferreira dos Santos, Professor Associado do Departamento de Engenharia Mecânica da Universidade de Aveiro

Agradecimentos / Acknowledgements

O cumprimento dos objetivos estipulados para este trabalho deve-se, em grande parte, à ajuda e apoio demostrados por algumas pessoas.

Ao Professor Doutor Vítor Santos por ter aceite tão prontamente ser meu orientador e por me ter mostrado e encorajado nos novos mundos da perceção 3D.

Aos meus colegas do Laboratório de Automação e Robótica por todo o espírito de equipa e boa disposição demonstrada nas longas noites de trabalho.

Ao Jorge Almeida pela imensa sabedoria e opiniões fornecidas.

Finalmente, ao Miguel Oliveira, por todo o tempo e paciência depositados no meu trabalho. Miguel, funcionou!

Palavras-chave

Estacionamento paralelo; Veículo não-holonómico; ROS; Kinect; Trajetórias compostas.

Resumo

Este trabalho enquadra-se no contexto da condução autónoma, e o objetivo principal consiste na deteção e realização de uma manobra de estacionamento paralelo por parte de um veículo não-holonómico à escala de 1:5, utilizando um ambiente de programação ROS. Numa primeira fase são detetados os possíveis lugares vagos com recurso a uma nuvem de pontos proveniente de uma câmara 3D (Kinect), analizando volumes ao lado do carro. Assim que é encontrado um lugar vazio, inicia-se o estudo de possíveis trajetórias de aproximação. Estas trajetórias são compostas e são geradas em modo *offline*. É escolhido o melhor caminho a seguir e, no final, envia-se uma mensagem de comando para o veículo executar a manobra. Os objetivos traçados foram alcançados com sucesso, uma vez que as manobras de estacionamento foram realizadas corretamente nas condições esperadas. Para trabalhos futuros, seria interessante migrar este algoritmo de procura para outros veículos e tipos de manobra.

Keywords

Parallel parking; Non-holonomic vehicle; ROS; Kinect; Composed trajectories.

Abstract

This work fits into the context of autonomous driving, and the main goal consists of the detection and execution of a parallel parking manoeuvre by a 1:5 scaled non-holonomic vehicle, using the ROS programming environment. In a first stage, the possible parking locations are detected by analysing a point cloud provided by a 3D camera (Kinect) and specifically by analysing volumes on the side of the car. Whenever an empty place is found, the study of possible paths of approach begins. These are composed trajectories, being generated offline. The path to follow is evaluated, and then the commands needed to the vehicle perform the selected path are sent. The outlined objectives were successfully achieved, since parking manoeuvres were performed correctly in the expected conditions. For future work, it would be interesting to migrate the search algorithm to other types of vehicles and manoeuvring.

Contents

List of Figures

List of Tables

Code sections

Part I

Guidelines

Chapter 1

Introduction

It did not take many decades since the mass production of automobiles (1913) for companies to start thinking about autonomous driving. At the Norman Bel Geddes's Futurama exhibit, sponsored by General Motors at the 1939 World's Fair, appeared the very first idea of an autonomous vehicle - an electric car, controlled by radio and powered by circuits embedded in the roadway [O'Toole, 2009].

Despite having an early start, the legislation didn't allow this type of vehicles on public roads until June of 2011, when the State of Nevada became the only place in the World to authorize the use of autonomous cars on highways [Markoff, 2011]. The approved law defines an autonomous vehicle as *"a motor vehicle that uses artificial intelligence, sensors and global positioning system coordinates to drive itself without the active intervention of a human operator"* [Nevada, 2011].

As it was forbidden to circulate with autonomous cars on the street, technological developments followed another path - the creation of small driving aids. Nowadays it is clear that there is an increasing use of electronic components in cars, which are intended to improve the safety of vehicle's occupants. The new trend is to incorporate long-range sensors such as cameras or radars, to improve passengers and other cars safety [Fossati et al., 2011].

1.1 Autonomous cars

In the second half of the twentieth century some research institutions began to develop their prototypes of autonomous vehicles. Some of the most remarkable concepts are presented below.

The first project to achieve success in the development of a driverless vehicle was concluded in 1977 by the Tsukuba Mechanical Engineering Laboratory in Japan. It tracked white street markers (on a clearly marked course) using computer vision and achieved speeds up to 30 km/h. One of the biggest difficulties of that project was the hardware required, since commercial computers were much slower than they are today [Schmidhuber, 2011, Chiafulio, 2010].

In the 1980s Ernst Dickmanns and his group at the University Bundeewehr Munich (UniBW) built robot cars using parallel computers and applying techniques such as saccadic vision (cameras focus on the most relevant points of interest) and probabilistic approaches (use of Kalman filters). They started the project "VaMoRs" equipping a

Mercedes-Benz van, table 1.1, with cameras and computers to track lane markings of an highway, achieving speeds up to 100 km/h. This application was helped by the strong geometric constraints available from knowledge of highways. Despite doing a safe driving, the initial experiments took place without traffic.

Since 1994 the Robot Car "VaMoRs-P", in short "VaMP" (table 1.1), managed to drive all by itself, at speeds up to 130 km/h. This car was equipped with a range of sensors for autonomous navigation comprising the sense of vision and inertial sensors for accelerations and angular rates. Road and object recognition was performed both in a look-ahead and in a look-back region.

In 1995 Ernst Dickmanns finishes the "VITA-2" project which is a twin car of the "VaMP". They were two autonomous vision based Mercedes 500SL (tracking up to 12 cars) which have driven more than 1000 km on the Paris multi-lane ring reaching speeds of 130 km/h, automatically passing slower cars. One year later, the "VaMP" Mercedes drove from Munich to Copenhagen and back (more than 1600 km) exceeding speeds of 170 km/h and completing the journey with 95% of autonomous driving [Schmidhuber, 2011, Chiafulio, 2010, Aloimonos, 1997].

In 1995 began the *'No Hands Across America'* project. During this tour of America, which was sponsored by Delco Electronics, AssistWare Technology, and Carnegie Mellon University, two researchers drove more than 4000 km with an autonomous vehicle, the Navlab 5 (table 1.1), which uses video images, to determine the location of the road ahead, and GPS/gyroscope information to estimate the current position. The appropriate steering position was calculated using all the data received from the sensors. One of the project's limitations was the need to humanly operate the throttle and brake pedals [Schmidhuber, 2011, Chiafulio, 2010, Jochem et al., 1995].

In the late 90s an Italian project (ARGO) modified a car, a Lancia Thema 2000 (table 1.1), which could follow the white marks in a highway. The vehicle was equipped with two black-and-white video cameras. The images acquired by the cameras were analysed in real-time and the results of the processing were used to drive an actuator mounted onto the steering wheel, which was the only fully autonomous component of the car. However, this car was very important on the path of autonomous driving, because of the use of low-cost components [Schmidhuber, 2011, Chiafulio, 2010, Bertozzi et al., 1998].

The year of 2005 was the debut of the *DARPA Grand Challenge*, which is *"(...) a field test intended to accelerate research and development in autonomous ground vehicles that will help save American lives on the battlefield."* [Darpa, 2007]. The Grand Challenge is not limited to college students. It brings together organizations from the industry to backyard inventors who are looking for a technological challenge. The course took place in the desert and had a large amount of GPS points to follow [Schmidhuber, 2011, Chiafulio, 2010]. The large majority of participants used new forms of perception of the external environment, such as LiDAR systems, which can provide a very accurate 3D map of the surrounding region. The winner of the competition was the 'Stanley' car (table 1.1) from the Stanford University.

In 2006 began the ELROB (European Land Robot Trial) which is not a competition, but a pure demonstration of what European robotics is able to achieve today. The ELROB is an annual event and alternates between a military and a civilian focus each year. This competition brings great benefits to the development of algorithms for navigation and perception, because the circuits have obstacles that were not known initially (the track is maintained secret until the day of the event) [Schmidhuber, 2011, Chiafulio, 2010, Schneider, 2012].

The DARPA Challenge returned in 2007 now with the name of “DARPA Urban Challenge”. This time the autonomous vehicles must run trough an urban environment respecting traffic signals. The competition was won by Carnegie Mellon University (table 1.1).The sensors began to be more elegant, and some of the semi-autonomous vehicle characteristics started to be included by some motor companies like Audi, Volvo and GM. [Schmidhuber, 2011, Chiafulio, 2010].

Table 1.1: Autonomous cars achievements through the years

Year	Project	Achievement	Image
1980s	VaMoRs [VaMoRs, 2012]	Track road markers on highway	
1994	VaMoRs-P [Behringer, 2007]	Road and objects recognition	
1995	Navlab 5 [NavLab, 2012]	4000 km with autonomous steering-wheel control	
1998	Argo [Vislab, 2009]	Autonomous driving using low-cost components	
2005	Stanley [Hudson, 2008]	Champion of the 1st Darpa event	
2007	Tartan [Tartan, 2012]	Champion of the Urban Darpa	
2010	Google car [Ackerman, 2010]	First legal autonomous car	

Most recently Google showed the world its vision of what an autonomous car is (table 1.1). They took a normal Toyota Prius and assemble on it a LiDAR sensor (a Velodyne), some radars, a camera, a GPS and IMU, and a speed encoder fixed to the rear wheel. They manage to navigate autonomously in several occasions, using the maps provided by Google maps. Nowadays, this car is the only one in the World which is legal to be autonomously driven, in the State of Nevada.

1.2 The parking manoeuvre

Over the years, the growth of metropolitan areas has led to an exponential increase on the number of cars and consequently to a decrease on the available parking spaces.

From all the types of parking manoeuvres, parallel parking is the one that best optimizes the available space on the street, but also the most difficult to achieve. Driving forward into a parking space on the side of a road is usually not possible. The driver should reverse into the spot to take advantage of a single empty space.

Because of all of that, the parking manoeuvre is something that worries many people, not only for its complexity but also for the need to be done quickly to prevent the formation of traffic jams. The car is the principal responsible for this complexity because it is a non-holomic system (figure 1.1) where the number of control commands available is less than the number of coordinates that represents its position and orientation. Here the final state of the system depends on the intermediate values of its trajectory through the space. So, there are positions which are reachable by holonomic systems, but which are not achievable by non-holonomic ones in the presence of obstacles.

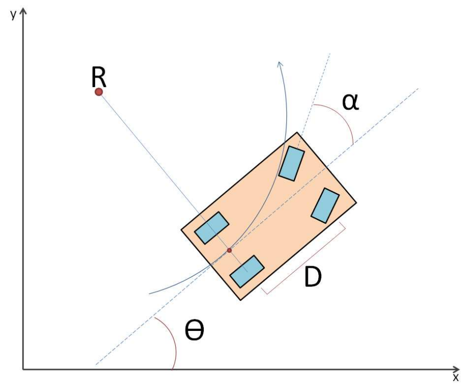


Figure 1.1: Non-holonomic model of a car

In 1934 was presented the first prototype of a vehicle with an easy parallel parking system (figure 1.2). That model consisted on the raise of the vehicle with four hydraulic jacks with wheels that allowed the car to move sideways to fit on the available space. Despite the great technological advance presented this model was never produced [Brown, 1934].

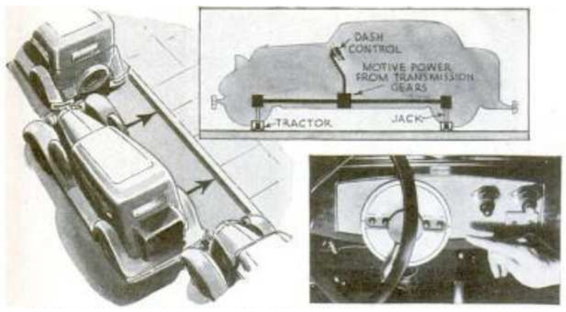


Figure 1.2: First easy parallel parking prototype [Brown, 1934]

The first vehicle to be able to do a parallel parking without human intervention was developed at INRIA (Inventeurs du monde numérique, France) in the 90s. The car was an electric Ligier model equipped with sonars at the front and rear bumpers to measure distances (figure 1.3). It was possible to drive the car until the appearance of a row of parked cars. Then, the turn of a switch put the car in automatic mode making it move forward and looking for empty spaces with the help of the sonars. If a spot is found, a computer calculates all the distances required and send information to the electric motors (engine and steering wheel) to perform the manoeuvres. This project allows the car to leave the parking lot automatically too, but all the movements were limited to flat ground [INRIA, 1998].

In 2000 this project was extended to the perpendicular parking manoeuvre, but the car employed a LiDAR system to sense the surrounding space.

Figure 1.3: Inria auto parking car [Paromtchik, 2012]

Nowadays, car brands have been investing large amounts of money to develop electronic components to aid the driver in parking manoeuvres. Systems that go from the parking spot detection to the most advanced ones that control the steering wheel on the parking manoeuvre have been released in the past few years.

1.3 The ATLAS project

The ATLAS project begun in 2003 in the group of Automation and Robotics from the Department of Mechanical Engineering of the University of Aveiro. The main goal of the project was the development of advance sensing and active systems to implement in mobile robots which were created to participate at Autonomous Driving competition (AD) taking place at Portuguese Robotics Open [FNR, 2012].

The AD represents a technical challenge, in which autonomous robots must travel along a road like scenario, which is composed of several visual patterns (figure 1.4). This contest is divided in 3 stages, of increasing complexity. The fastest robot to complete the course is the winner. Time penalties are given when robots run of the road, bump into obstacles or disrespect traffic signs.

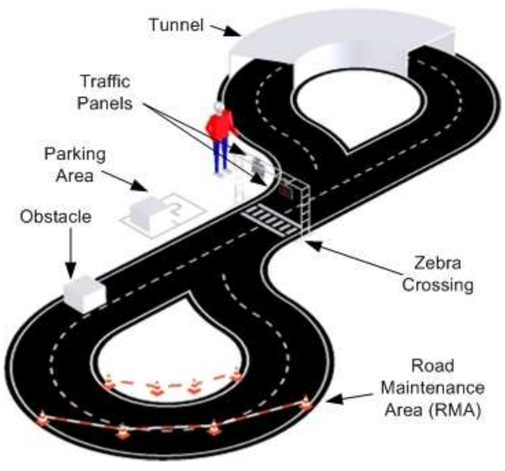


Figure 1.4: Autonomous driving competition track

The first robot of the Atlas series (figure 1.5) was based on an aluminium frame with two wood layers. It has a mechanical differential to provide traction and only one webcam looking at a v-shape mirror to allow the entire visualization of the sides of the road.

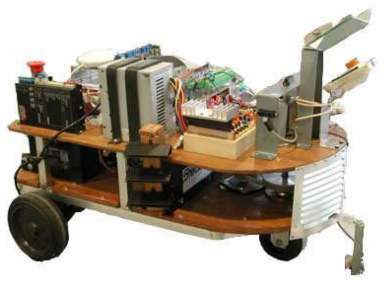
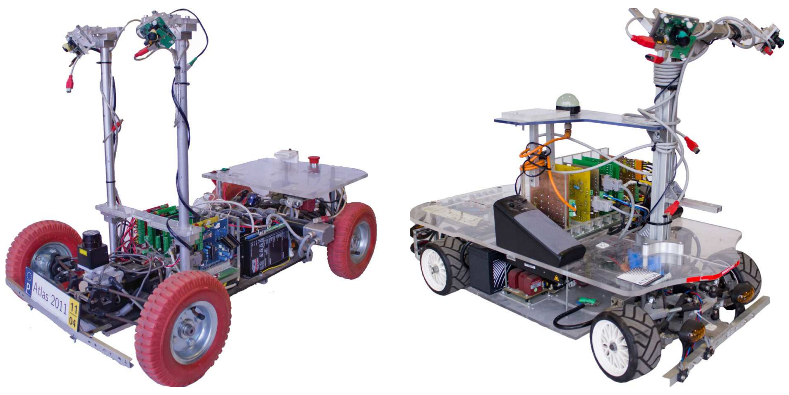


Figure 1.5: First atlas robot prototype

The next prototype to be made was the ATLAS 2000 (figure 1.6(a)) a 1:4 scale model used by model makers to achieve a similar relation to a common car. In 2006 the ATLAS team won for the first time the AD competition with this car, repeating the victory in 2007. Several upgrades were carried out to obtain better performances from the robot.

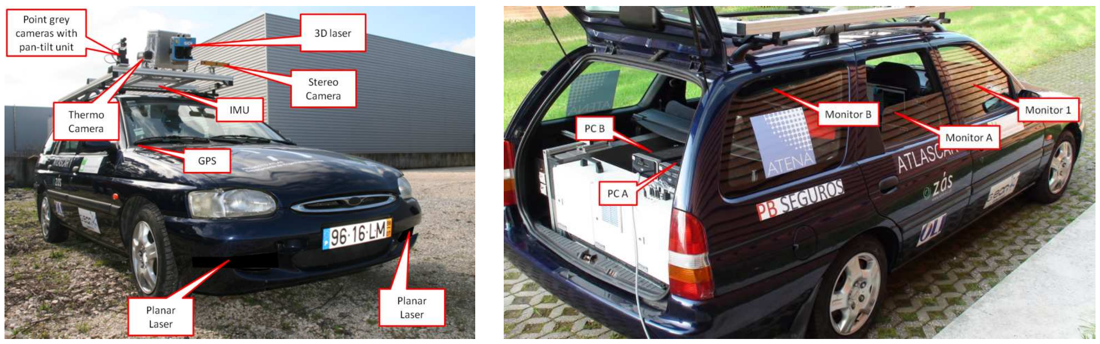
In 2008 a new robot was developed, the AtlasMV (figure 1.6(b)), to improve the performances of its predecessor. That robot was designed to be smaller (1:5), lighter and faster. New steering control mechanisms, pneumatic braking systems (later replaced by hydraulic brakes) and active perception unit were developed.



(a) ATLAS 2000 (b) AtlasMV

Figure 1.6: Autonomous driving competition champions

Nowadays the Atlas group is evolving to deal with real road scenarios. To achieve this objective, a full sized prototype (1998 Ford Escort), the ATLASCAR 1 (figure 1.7), was equipped with several state of the art equipment. A 200A alternator, a 3000W inverter, lasers sensors, a stereo camera, a IMU, a GPS, among others [ATLAS, 2011, Santos et al., 2010].



(a) Atlascar 1 front-view (b) Atlascar 1 rear-view

Figure 1.7: Atlascar 1 - The first full scale prototype [ATLAS, 2011]

1.4 The Robot Operating System

“*ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications* [ROS, 2012]”.

It provides services that are available on an operating system such as hardware abstraction, low-level device control functionality and message passing and packages management.

To fully understand the ROS philosophy it should be noted that it has three levels of concepts. The filesystem, the computation graph and the community.

The filesystem is composed by the resources found on disk, such as:

- Packages - the unit of ROS organization. They may contain ROS runtime processes (nodes), libraries, data-sets, etc.
- Manifests - provide information about a package, like the library dependencies and compiler flags.
- Stacks - a collection of packages.
- Message types - define data structures to the messages sent in ROS.

The computation graph (figure 1.8) represents the network of ROS processes. There are a few computation concepts:

- Nodes - as ROS is designed to be modular, nodes are processes that perform computation.
- Master - it provides names registrations and lookup for the rest of computation graphs.
- Messages - are a simple data structure which allows communication between nodes.
- Topic - is the name used to identify the concept of the message.
- Bags - are the format to save and playback ROS messages data.

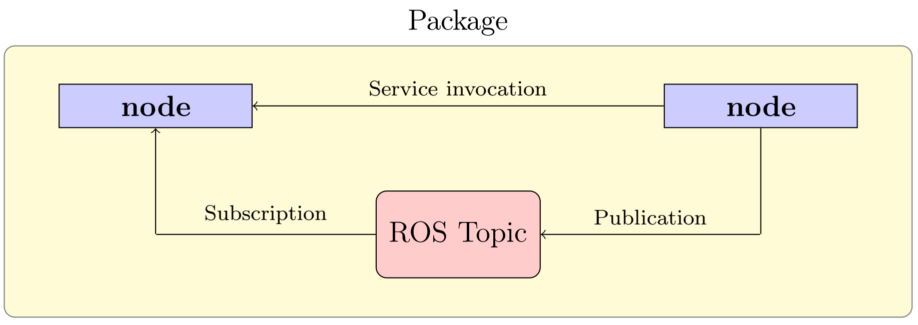


Figure 1.8: ROS - simple network of process

The ROS community allows the exchange of stacks, packages and knowledge between the global users.

These characteristics turn ROS into a good platform to develop code. However, ROS currently only runs on Unix-based platforms.

1.5 Objectives

Due to the great importance that has been given to the autonomous parking, the primary objective of this thesis is on the programming and implementation of that manoeuvre into an autonomous driving vehicle, the ATLASCAR 1.

Conducting a parking manoeuvre requires making two fundamental tasks, the first is the search for an empty space where to park the car and the second one consists of the approach to the required final position of the vehicle.

1.5.1 Parking spot detection

To begin the search for a parking place, it is very important to know first what can be considered an empty spot. Humans encounter locations to park the car in many different situations. However, it is very difficult to 'tell' a machine what are all the parking possibilities, since they are endless. So, this thesis must cover only the most difficult parking manoeuvre, which is the parallel one (due to the non-holonomic nature of the vehicle).

The information about the surrounding environment should be given by a Kinect® sensor (figure 1.9) not only because it is a new type of hardware present on the laboratory, but also because of the huge acceptance that this sensor has taken on the researchers community and the big precision/price ratio.

Figure 1.9: Kinect® camera for Xbox 360

The selection of the parking space must be variable in accordance to the vehicle dimensions and mechanical restrictions keeping always in mind that the robot is a non-holonomic vehicle in a world with some obstacles and traffic rules.

One last thing to have in mind, is that another project related to the ATLASCAR prototype is being developed to allow the automatic actuation of the gearbox (once the original car transmission is manual), so it may be necessary to operate another navigation robot of the laboratory (*e.g.* the AtlasMV) on a smaller scale scenario.

1.5.2 Planning the parking manoeuvre

After the parking spot has been detected and selected, an algorithm must generate some approaching trajectories to achieve the desired coordinates and orientation. Those trajectories must assume a complex form, which means that the vehicle may change its turning angle while moving forward or backwards.

The choice of one trajectory to follow should be studied in another algorithm because of the importance to complete the manoeuvre without collisions and by the shortest path, always having in consideration the non-holonomic nature of the model.

After all, and if the program assumes that there is an empty spot which is reachable, a message should be sent to the low level controls of the robot so it can follow the programmed path.

Summing up, the entire process can be outlined by the scheme presented in figure 1.10.

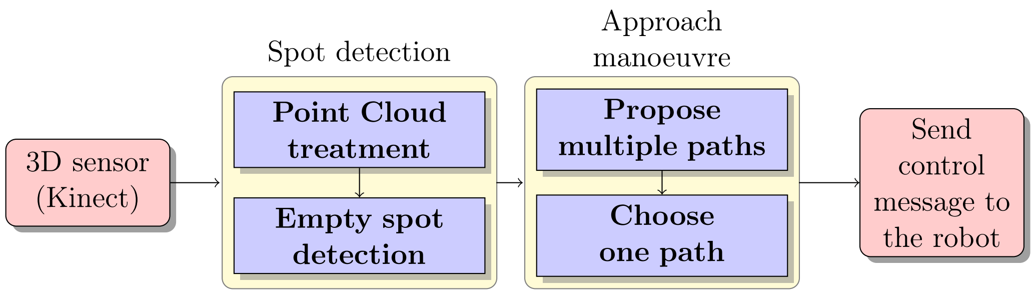


Figure 1.10: Scheme of the programming procedure

1.5.3 ROS utilization

This thesis must be done using the ROS environment not only because of all the libraries, drivers and packages existing, but also because the Laboratory of Automation and Robotics was at the time migrating all of its code to that platform. All of the work should be done not only to satisfy a single task, but also to be useful to the community of researchers of the laboratory.

Chapter 2

State of the Art

2.1 Perception of the external environment

Since its emergence, autonomous vehicles make use of perception systems. The surrounding environment and the state of the car are sensed by the use of techniques such as computer vision, LiDAR, radar or GPS/INS.

2.1.1 Computer Vision

"*The goal of computer vision is to make useful decisions about real physical objects and scenes based on sensed images* [Shapiro and Stockman, 2000]".

Many of the activities performed by human beings in day-to-day would not be possible without the use of vision, so it becomes clear why the first prototypes of autonomous driving cars used computer vision to detect obstacles and the road.

From the camera, with the information gathered, there is a path to follow, as shown in figure 2.1.

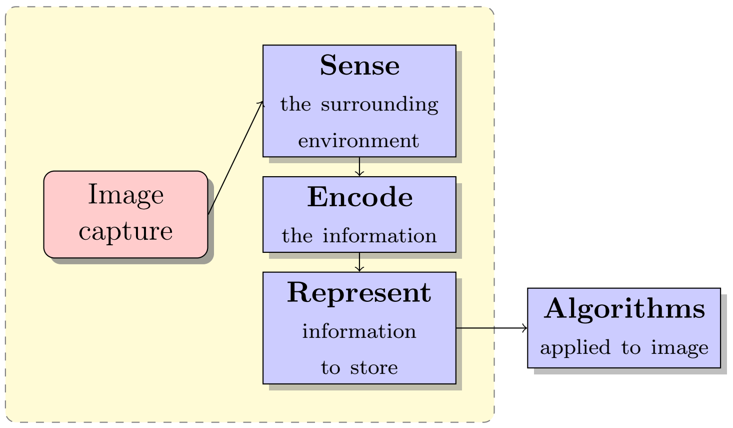
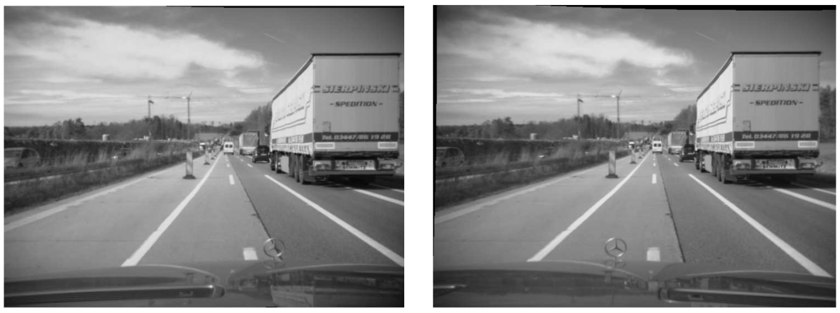


Image capture
Sense the surrounding environment
Encode the information
Represent information to store
Algorithms applied to image

Figure 2.1: Camera information path

One of the techniques applied on autonomous vehicles consists of the use of a stereo camera (or two cameras) which generates a stereo pair of images (figure 2.2). In those two images it is possible to identify pairs of corresponding points. Those points allow the calculation of the distance between the projected point in the three-dimensional world and the recording cameras [Klette and Liu, 2008].



(a) Left (b) Right

Figure 2.2: Stereo pair of images [Klette and Liu, 2008]

Another technique consists of the analysis of a video sequence (multiple frames). Motion estimation for these sequences should provide information about movements of objects for each sequence, identifying possible conflict paths. By applying several filters (such as Canny edge detector) to the image sequence it becomes possible to identify obstacles and the road in different positions through the time. This allows the creation of a vector with speed and trajectory to each object identified [Klette and Liu, 2008].

2.1.2 3D image cameras

3D image cameras work much like an ordinary camera to capture the two dimensions of an image. To add the third dimension (which gives the depth information of the scene, figure 2.3) there are two different methods which can be used. The time of flight imaging (ToF) and the structured light imaging.

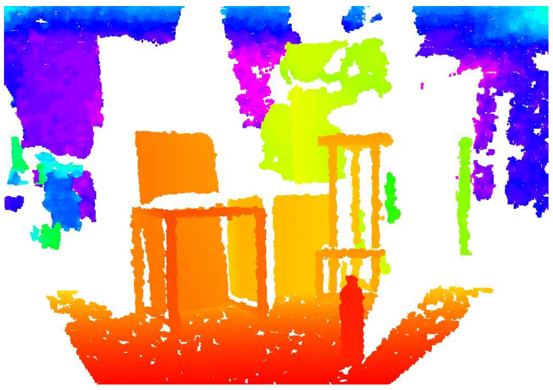


Figure 2.3: Kinect example of depth image

The ToF technique consists of measuring the depth of a scene quantifying the changes that the emitted light encounters. It can be divided in two different principals:

- Pulsed modulation - Measures distances to objects by measuring the total time that a light pulse needs to travel to an obstacle and bounce back.

- Continuous wave modulation - Measures distances evaluating the phase of the wave reflected from the obstacles encountered.

One example of a sensor which works with the ToF principals is the D-Imager (figure 2.4) which was announced by Panasonic in 2010.

Figure 2.4: Panasonic D-Imager sensor [Koifman, nd]

On the other hand, the structured light imaging technique consists of the projection of a known light pattern to the scene. The distortion provoked on the patern will give the information about the distances, figure 2.5.

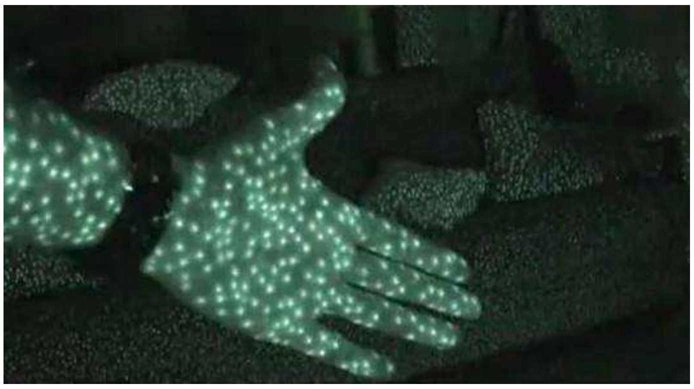


Figure 2.5: Infra-red pattern emitted by the Kinect sensor [Fisher, 2012]

The Kinect, figure 1.9, which was first announced on June of 2009 under the code name “Project Natal”, is currently the most popular of this kind of systems because of its low price and the open nature of the communication code. The sensor specifications are presented in table 2.1.

Table 2.1: Kinect sensor specifications

Field of view		Data stream		Range
Horizontal	Vertical	Depth	RGB	min-max
57°	43°	320×240 30 Hz	640×480 30 Hz	0.6-8.0 m

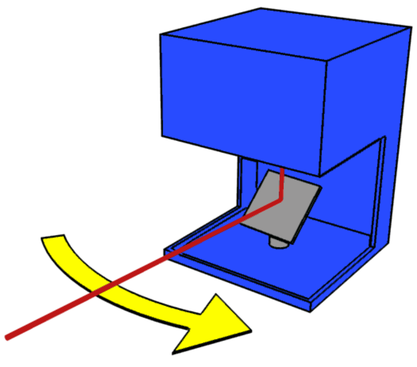
2.1.3 LiDAR Sensors

LiDAR - Light Detection And Ranging - (also known as LADAR) is a technology that can measure distances to objects using infrared, visible or ultraviolet light. This technology has several decades, yet its commercial application have only been developing in recent years.

In this system the higher the power of the light beam the greater is the achievable range. However, this system should compromise the power of the laser to prevent injury to the eyes of people who may be affected [Schwarz, 2010]. Also, LiDAR has some problems with certain weather conditions because it uses a form of light, which is easily reflected, dispersed, and sometimes absorbed by rain or fog, resulting in loss of information.

The basic elements of a LIDAR system (figure 2.6) are a laser scanner, a rotating mirror and a cooling system. The scanner is designed to record the time that the laser pulse takes to be reflected and return since it was emitted. The rotating mirror causes the laser beam to disperse through an angle which allows the construction of a 2D map.

By adding another degree of freedom (*e.g.* rotation axis, figure 2.7) to the LiDAR system, it becomes possible to obtain a three dimensional representation of the surrounding environment.


(a) LiDAR System components

(b) LiDAR sensor - Sick S3000 [Sick, 2012]

Figure 2.6: LiDAR system

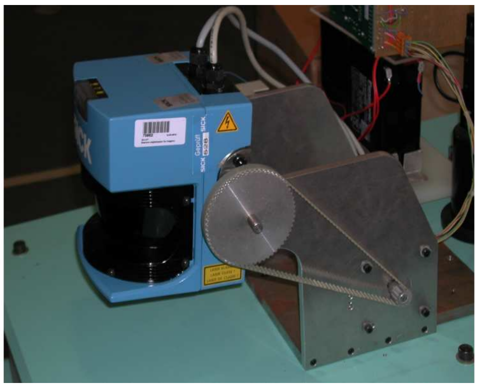


Figure 2.7: Sick LiDAR system with a rotating axis [Matos, 2003]

2.1.4 Radars

Today, some cars are equipped with sensors that measure the distance up to other vehicles or large objects in front of them.

Automotive radar sensors are less expensive and still offer the advantages of microwave-based sensing with respect to laser sensors and video cameras [Rohling, 2008].

There are two primary methods of measuring distances using radar:

- **Direct propagation** - consists of the measurement of the delay associated with the emission and reception of the signal. This delay is function of the speed of radio waves and its period.
- **Indirect propagation** - also known as FMCW (Frequency Modulated Continuous Wave), consists of the emission of a modulated frequency. The difference between the emitted and received frequency can be used to directly determine the distance as well as the relative speed of the object.

The main applications for the automotive radars can be divided in three main groups, as shown in figure 2.8.

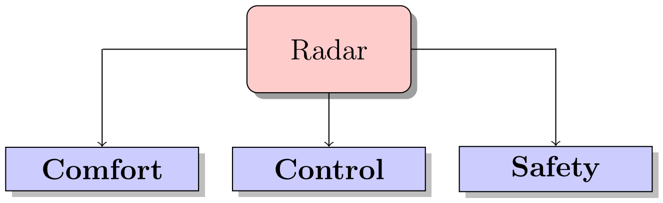


Figure 2.8: Automotive Radar applications

- **Comfort** - This area is ensured by parking aid sensors (which avoid the need of drilling holes in the bumpers) and blind spot warning system.
- **Control** - The active cruise control (ACC) is similar to a regular cruise control, but it verifies if there is any slower car in front of the vehicle, and regulates the travelling speed.

- **Safety** - If the radar sensors detect an eminent collision (Closing Velocity Sensing) a signal is sent to the vehicle control system to deploy some safety features (like the seat belt pre-tensioners).

2.1.5 Ultrasonic Sensors

Ultrasound is an acoustic wave with a frequency beyond human hearing (more than 20 kHz) and its propagation speed (approximately 340 m/s) is slower than light or radio waves, which allows measurements using low speed signal processing.

Ultrasonic sensors are used to determine the distance or direction of an object from the time the waveform takes to make a round trip to the target. The range of a sensor of this type is much smaller than that of a LiDAR system, so their use is greater in short-range applications, such as in parking assistance systems (figure 2.9). In these systems, sensors in the front and rear bumpers emit ultrasonic signals that are reflected by an obstacle in the range of detection. Traces of the reflected ultrasound are received by the sensors and their travel time is measured, calculating then the distance to the obstacle [Hikita, 2010].

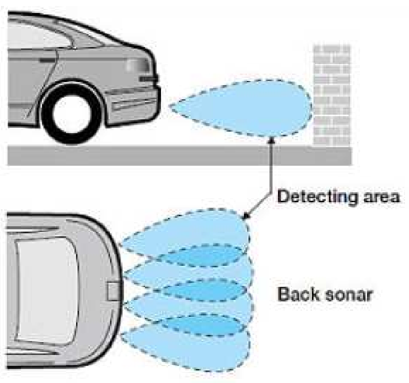


Figure 2.9: Ultrasonic parking sensors [Hikita, 2010]

2.1.6 GPS and INS

Driverless vehicles require a very precise information about its position, so they use a system which works similarly to the scheme shown in figure 2.10.

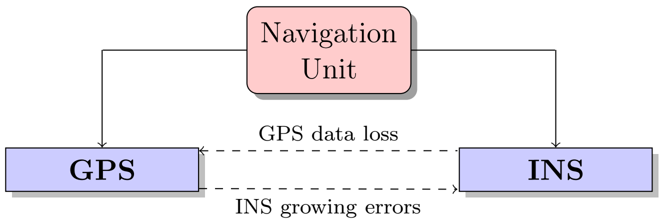


Figure 2.10: Operation of the navigation unit

Most navigation systems, these days, depend on the availability of information from their GPS coordinates to estimate the position of the vehicle. GPS satellites transmit information to receivers and, through the use of triangulation, the exact position of the user is estimated. Despite being very accurate, it is common that the GPS information is not reliable, because there can be loss of data (in tunnels or near high buildings). To overcome this loss of signal is used a INS (Inertial Navigation System) which uses accelerometers and gyroscopes to check changes in the position of the vehicle at a certain period of time. However, the GPS is very useful to correct the growing errors of the INS. So, both sensors can work in a single way, but their results are better if working together [Baker et al., 2006].

2.2 Autonomous parking

In recent times there have been many brands of automobiles that released to the market mechanisms to help the parking manoeuvre, since the technology is still expensive and not very reliable and the law does not allow fully autonomous parking.

It began with the use of ultrasonic sensors which detected the proximity of vehicles from the front and rear bumpers of the car which was performing the parking manoeuvre.

Later, ultrasonic sensors that warned the driver if there was a parking space big enough to perform the manoeuvre safely were introduced (figure 2.11). After the introduction of this system appeared a mechanism that gave the driver the instructions about the direction that the steering-wheel should have at every moment. In vehicles which have electric powered direction, the steering-wheel may even be controlled independently, letting the driver control only the pedals.

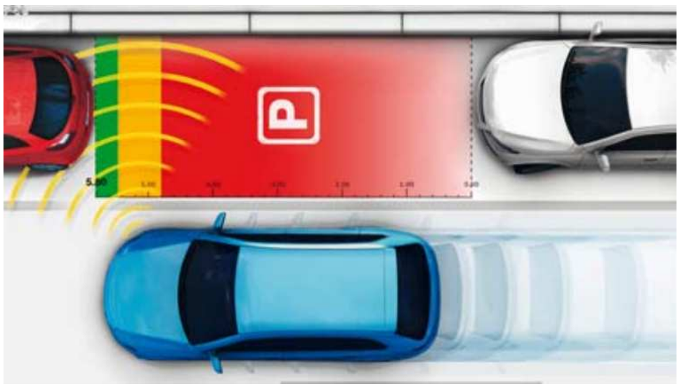


Figure 2.11: Bosch parking space detector system [Bosch, 2011]

These systems are virtually equal in all vehicles, being the Lexus LS460 (figure 2.12) the first production vehicle to be presented with a semi-autonomous parking system (the Lexus Park Assist). After this one, car brands like Toyota, Mercedes, Ford among others, presented their own solution (all of them very similar).

Figure 2.12: Lexus LS460 parking itself [AutomotiveAdicts, 2006]

All systems developed to date have encountered problems when the car does not meet the pre-programmed situations. For example, when there is a car parked in second row, or when there are pedestrians near the side-walk.

The need for perfect conditions make it nearly impossible to have a reliable self-parking system.

Part II

Methods and Programming

Chapter 3

Parking spot detection

Humans look for an empty parking space with a certain logic and criteria which is hard to 'teach' to a machine. So, in order to programme an autonomous parking spot detector, some simplifications should be done. This part of the thesis will describe the paths followed to allow the search for open spaces to park a robotic car. Some of the code made for this purpose will appear in a yellow shaded rectangle with an explanation at the bottom.

3.1 Analysis of the parking spot and vehicle

3.1.1 Parking spot definition

The first thing to be done when developing a parking spot detector algorithm is the definition of what a parking space is. There are several types of parking situations, like parallel, perpendicular, angled, among others. From all the kinds of manoeuvres the most difficult one to be done quickly and in safe conditions (due to the non-holonomic characteristics of car-like vehicles) is the parallel parking.

For those reasons, the chosen parking approach method to be done autonomously is the parallel manoeuvre.

3.1.2 Used vehicle

As explained on section 1.5.1, the work developed on this thesis must be inserted on the Atlascar global project. However, the vehicle used on that project was not ready (until the present date) to operate the gear box autonomously, so it became necessary to apply the parking algorithm to a robotic vehicle available at the laboratory, the AtlasMV, which is a 1:5 scale model of a road vehicle.

The code developed should be generic enough to allow its use on different kind of vehicles with simple and quick modifications. Due to that, the migration of the code to a full scale model (the Atlascar) should be pretty easy and will be explained in each part of the code.

3.1.3 Search method

The surrounding environment is perceived by a point cloud which, in this case, is generated by a Kinect sensor. The parking space coordinates (figure 3.1) are obtained by treating the information which comes from the cloud generated.

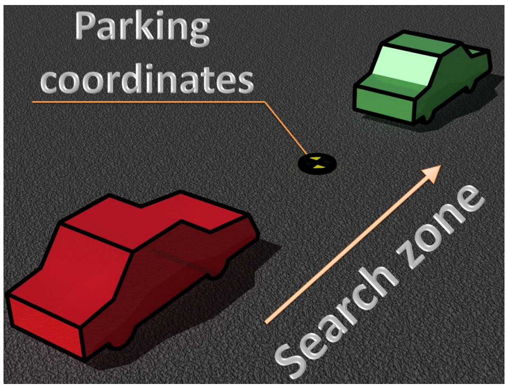


Figure 3.1: Empty parking spot coordinates

To this type of parking space there is only one limitation: the parking spot must be always preceded by another parked car (or object). This is due to the fact that it is very difficult for a machine to realise what a parking space is. So, on the search zone, the algorithm only considers a parking space if there is another vehicle behind that place.

3.2 Point cloud reconstruction

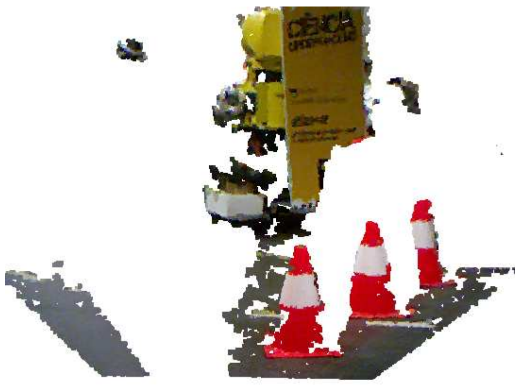
In order to reconstruct the external environment it was necessary to build some hardware and code a few ROS packages. Those processes are explained on the following subsections.

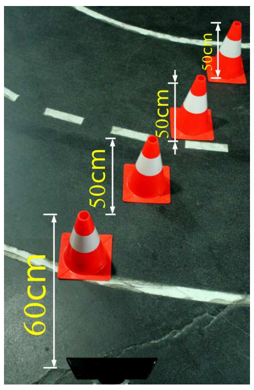
3.2.1 Kinect assembly

The Kinect sensor was not built to operate in outdoor conditions, since when it is exposed to direct sunlight the infrared sensor becomes saturated. As consequence, there is no information about the depth of the scene exposed. Due to that, the use of the Kinect sensor on the Atlascar vehicle will be very limited. Only in low light conditions (*e.g.* cloudy weather or at night) this sensor will provide an accurate point cloud information. However, the navigation should not be limited by the weather, so in non favourable conditions to the Kinect, the point cloud should be provided by another sensor, like the stereo camera or lasers.

The AtlasMV vehicle is an indoor robot, so the use of the Kinect is perfectly allowed. To receive the point cloud provided by this sensor it was used an existent driver on the ROS repository, the *openni_kinect*.

As it was said on section 2.1, this camera has a range of 0.6 to 8.0 meters when measuring the depth of a scene. In figure 3.2 it can be seen that the closest cone to the camera is not perceived by the depth sensor and it generates a lack of information projection shadow.


(a) Point cloud minimum range limitation



(b) Top view of minimum range test

Figure 3.2: Kinect test of minimum range

In order to maximize the use of the optimal range of the camera and to allow the search for parking spaces at a reduced distance from the line of parked cars, the Kinect was mounted on the right side of the vehicle and facing the left side (figure 3.3). So, the parking manoeuvre performed by this vehicle will be done following the left side of the road and not the right, like most of the countries legislation assumes.

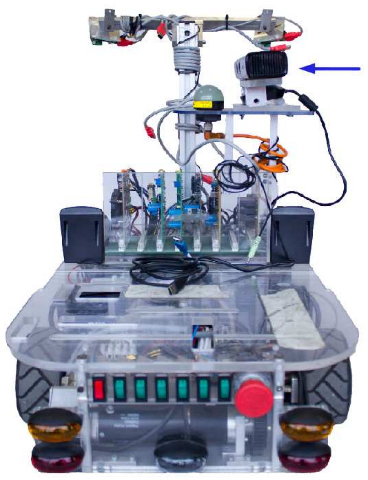
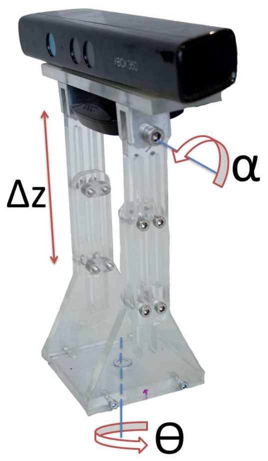
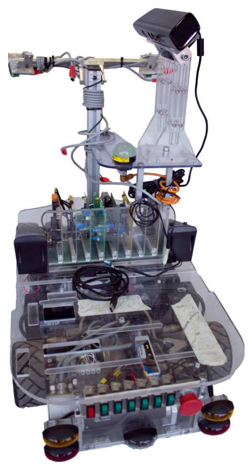


Figure 3.3: Kinect mounting position

Because of the chosen mounting position and the limited field of view of the camera there would be hidden point cloud information. To overcome that loss, an adjustable mounting device was built. This device can be regulated in height and in two different angles (figure 3.4) in order not only to make the parking spot search more accurate, but also to allow different utilizations of the sensor in future works.



(a) Kinect support



(b) Rear view of the mounted support

Figure 3.4: Device constructed to accommodate the Kinect sensor

After the Kinect assembly on the mounting device and posterior fixation to the AtlasMV robot, it was needed to define the position of the camera frame in relation to the one of the car. It was made by a ROS node which is responsible to publish the transformation between frames.

Code section 3.1: Transformation publisher

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <std_msgs/Float64.h>
#include <math.h>

int main(int argc, char** argv)
{
    ros::init(argc, argv, "pub_transformations");
    ros::NodeHandle n;
    tf::TransformBroadcaster broadcaster;
    ros::Rate r(10);

    float alpha=(38.0)*(M_PI/180);// the angle of the camera
    tf::Transform transform(btMatrix3x3(0,-1,0, cos(alpha),0,sin(alpha), -sin(↩
        alpha),0,cos(alpha)),
                                btVector3(0.236, -0.05, 0.68));
    while(n.ok())
    {
      broadcaster.sendTransform(tf::StampedTransform(transform, ros::Time::now()↩
          ,"/vehicle_odometry", "/openni_camera"));
      r.sleep();
      ros::spinOnce();
    }
}
```

The first thing which is done in this piece of code is the variable initialization. The ROS node is called *pub_transformations*, and there is a broadcaster of transformations called *broadcaster*.

After the determination of the tilt angle of the camera, which may be done using the ROS package *kinect_aux*, the transformation matrices were defined. The matrices correspondent to the rotation (equation 3.1) and distances between frames (equation 3.2) are calculated by replacing the values of the angles and distances in the correspondent place.

$$Rot = \begin{bmatrix} \cos\theta & -\sin\theta\cos\alpha & \sin\theta\sin\alpha \\ \sin\theta & \cos\theta\cos\alpha & -\cos\theta\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \tag{3.1}$$

$$\Delta r = \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} \tag{3.2}$$

In the present case (figure 3.5), the matrices parameters are:

- $\theta = -\pi/2$ rad
- $\alpha = (38.0) * (\pi/180)$ rad
- $\Delta x = 0.236$ m
- $\Delta y = -0.05$ m
- $\Delta z = 0.68$ m

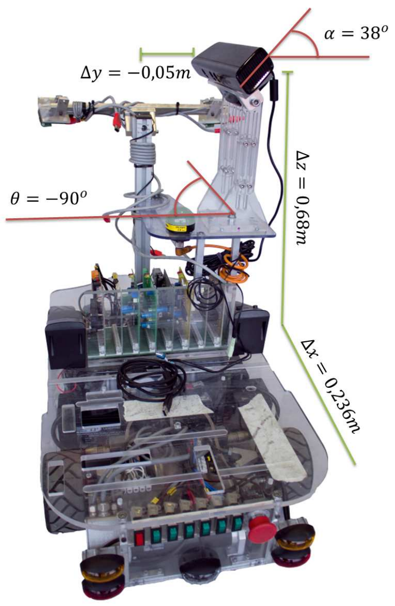


Figure 3.5: Final assembly of the sensor

After the definition of the transformation to broadcast (named *transform*), composed by a 3×3 rotation matrix and a 3×1 Δr vector, a while loop start to broadcast the transformation between the *vehicle_ odometry* (center of the rear axle) and the *openni_ camera* (center of the kinect sensor) frames. The last two lines of cycle are responsible to keep the loop at a rate of 10 Hz.

To migrate this piece of code to the Atlascar vehicle the only part which needs to be changed is the transformation matrix that should assume values measured after the kinect assembly on the car.

3.2.2 Frequency modulator

The *openni_kinect* package available on ROS repository publishes the point cloud information at a relatively high rate. Later this became undesirable when other process were in execution in the same processor. So another node was created, responsible to receive the point clouds from the *openni_kinect* and publish them back, but at a slower rate.

Code section 3.2: Kinect point cloud re-publisher

```
(...)
void conversion (const sensor_msgs::PointCloud2ConstPtr & pcmsg_in)
{
  pcl::PointCloud<pcl::PointXYZ> pc_cut;
  pcl::PointCloud<pcl::PointXYZ> processed_pc;
  sensor_msgs::PointCloud2 pcmsg_out;

  pcl::fromROSMsg(*pcmsg_in,pc_cut);
  pc_cut.header.frame_id=pcmsg_in->header.frame_id;
  for (int i=0;i<(int)pc_cut.points.size();i++)
  {
    if (pc_cut.points[i].z>0.6 && pc_cut.points[i].z<1.5)
      processed_pc.push_back(pc_cut.points[i]);
  }
  processed_pc.header.frame_id=pc_cut.header.frame_id;
  pcl::toROSMsg(processed_pc,pcmsg_out);

  pcmsg_out.header.stamp=pcmsg_in->header.stamp;
  cloud_pub.publish(pcmsg_out);
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "kinect_freq_mod");
  ros::NodeHandle n;
  ros::Rate loop_rate(10);
  (...)
  // Subscribe of the Kinect point cloud message
  ros::Subscriber sub = n.subscribe ("/point_cloud_from_kinect", 1, conversion);
  // Advertise of the resutant point cloud
  cloud_pub = n.advertise<sensor_msgs::PointCloud2>("/point_cloud_input", 1);

  while(ros::ok())
  {
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

At the main function, this code subscribes a point cloud from the Kinect, and executes a callback each time a message is received. The callback is responsible to reduce the amount of points existent in the original point cloud. This is achieved by confining the depth information of the message. In this case, only the points which are between 0.6 and 1.5 meters in the depth axis are considered acceptable, since the other points do not bring any additional information to the parking spot localization.

In the “while” cycle the new point cloud is published at a rate defined by the ros::Rate, which in this case is 10 Hz.

To reduce even more the size of the message transmitted, only the geometric parameters of the point cloud (x, y and z) are sent. The Kinect also provides colour information about the scene, but to detect volumes it is not necessary (figure 3.6).

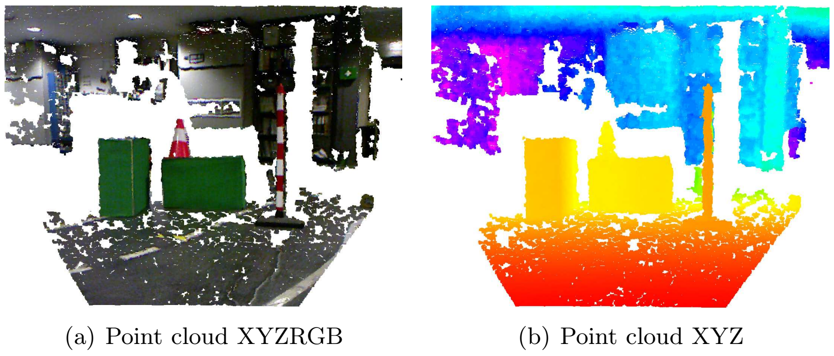

(a) Point cloud XYZRGB (b) Point cloud XYZ

Figure 3.6: Comparison between point cloud XYZRGB and point cloud XYZ

To migrate this code to another vehicle, the only thing which needs changes is the callback. There, the distances which limits the depth of the point cloud are optimized to the use on the AtlasMV vehicle. New values should be defined to use on another vehicle.

3.2.3 Point cloud accumulator

The Kinect horizontal field of view does not allow the search for an empty parking space at the normal search distance (figure 3.7). To overcome this limitation there were two possible solutions. The first one consisted of a search for a parking space made at a bigger distance, however this may be impossible due to the traffic regulations restrictions. The other solution assumed a reconstruction of the point cloud (figure 3.8). To proceed to the reconstructions the only two things that are needed are the point cloud and the odometry information of the vehicle (the vehicle position on the global frame at each time).

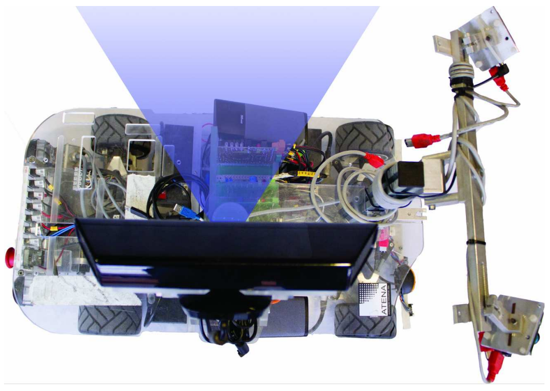


Figure 3.7: Kinect horizontal field of view

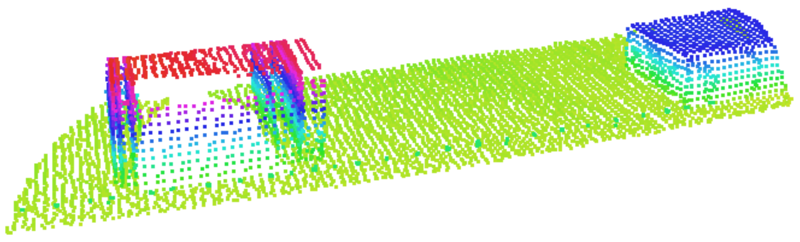


Figure 3.8: Reconstructed point cloud with odometry information

There was already a C++ class developed in the Laboratory of Automation and Robotics server which was responsible to accumulate points to a certain frame of accumulation. To use that class it was just necessary to include the node on the launch file (file which manages the processes to launch).

Code section 3.3: Nodelet launch file

```
<launch>
(...)
  <node name="pc_accumulation_nodelet" pkg="pc_accumulation" type="↩
      pc_accumulation_nodelet" >
                <param name="distance_from" value="2.5" type="double"/>
                <param name="timer_value" value="0.1" type="double"/>
                <param name="acc_frame" value="/world"/>
                <param name="voxel_size" value="0.03" type="double"/>
                <param name="removed_from" value="/vehicle_odometry"/>
                <param name="odometry_topic" value="/atlasmv/base/odometry"/>
                <remap from="/pointcloud0" to="/point_cloud_input"/>
  </node>
</launch>
```

In the pc_accumulation node there are some parameters which need to be defined.

- "*distance_from*" - means the distance accumulated in meters.
- "*time_value*" - is the period of the message.
- "*acc_frame*" - is the accumulation frame (where the point cloud must be reconstructed).
- "*removed_from*" - is the frame of the odometry message.

This node also provides a tool which can reduce the point cloud density of the resultant accumulated point cloud, the *voxel_size*. In this case, a voxel grid of 0.03 meters means that each point is separated from the others by at least 0.03 meters.

The parameters that mandatorily need to be changed when migrating the code are the odometry message, the frame of accumulation and the accumulated distance.

3.3 Volume detection

The volume detection is made by the use of functions which provide informations about the presence or not of points at a certain zone. All the processes and code are shown in the next subsections.

3.3.1 Points from volume extraction

To verify if there were any points from a point cloud at a certain region of space it was created a C++ class which, with some input parameters, determines the existence or non-existence of points. That class assumes that the input parameters (figure 3.9) are:

- Convex hull point cloud - Those points (at least three) define the geometric figure of the solid which will be extruded. This solid is in fact the region of search for points.
- Positive/negative offset - It defines the height of the extruded solid. The positive offset indicates the amount of positive extrusion, and the negative do the opposite.
- Zone flag - This flag is very important because it establishes if the search zone is inside the convex hull (negative flag) or outside it (positive flag).

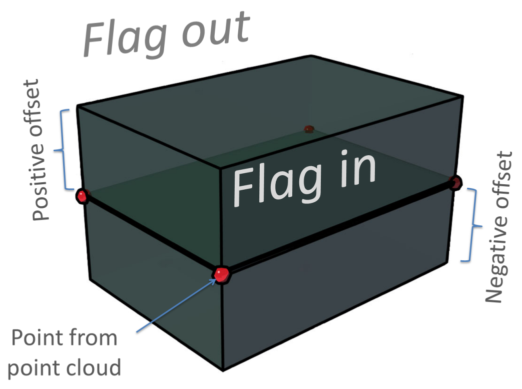


Figure 3.9: Points from volume input parameters

Code section 3.4: Convex hull extraction

```
(...)
pcl::ExtractPolygonalPrismData<T> epp;
(...)

pcl::ExtractIndices<T> extract; //Creates the extraction object
pcl::PointIndices::Ptr indices;
indices.reset();
indices = pcl::PointIndices::Ptr(new pcl::PointIndices);

(...)

if ((int)indices->indices.size()!=0)
{
    extract.setInputCloud(pc_in.makeShared());
    extract.setIndices(indices);
    extract.setNegative(flag_in_out);
    extract.filter(pc_in_volume);
}
(...)
```

On the C++ class code it is defined a polygonal prism data of extraction (epp) which is responsible to verify if the points from the point cloud in study are inside or outside the limits of search. The indices of the points which satisfies the condition are returned and indicate the points from the point cloud which are in the required area.

One of the public parameters of the class which may be obtained is a point cloud containing only the points inside (or outside if pretended) the defined convex hull (figure 3.10).

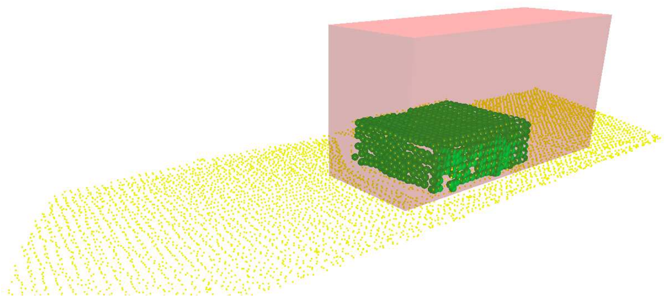


Figure 3.10: Green points inside a convex hull

3.3.2 Empty volume detection

This is the section responsible for the main execution of the package "*parking_ detection*", which looks for empty volumes at a certain search zone. Thanks to the classes and execution nodes that were previously made, the main goal of this executer is to make all the nodes work together.

Code section 3.5: Empty spot detector

```
int main(int argc, char **argv)
{
    (...)
    tf::TransformListener listener(n,ros::Duration(10));
    (...)
    Publisher = n.advertise<trajectory_planner::coordinates>("/msg_coordinates",↩
         1000);
    // ______________________
    //|________Markers________|
    car_pub = n.advertise<visualization_msgs::Marker>( "car", 0 );
    (...)
    // ______________________
    //|_____ConvexHulls______|
    // ConvexHull 1
    convex_hull1.header.frame_id="/vehicle_odometry";
    pcl::PointXYZ pt1;
    pt1.x = spot_length/2 + spot_length/2; pt1.y= (spot_wide + spot_distance) + ↩
        spot_wide/2; pt1.z= 0.02;
    convex_hull1.points.push_back(pt1);
    (...)
    pfv.set_convex_hull(convex_hull1);
    (...)
    // ______________________
    //|______PointCloud______|
    //Point Cloud publications
    cloud_pub = n.advertise<sensor_msgs::PointCloud2>("/pc_ahead", 1);
    (...)
    // ________________________
    //|______PCL subscr.______|
    // Creates a ROS subscriber for the input point cloud
    ros::Subscriber sub = n.subscribe ("/pc_out_pointcloud", 1, cloud_cb);

    ros::Rate loop_rate(30);
    ros::spin();
}
```

Initially a listener is defined and will wait for transformations information between frames. After that it is important to initialize a publisher, which will be very useful later to send the parking coordinates message.

There are other publishers defined to represent markers (objects drawn on rviz, which is a visualizer from ROS). The coordinates that will define the vertices of the figure to extrude into a convex hull are also initialized in this section of code. These points are geometrically defined by the position of the convex hull vertices in relation of the search vehicle (figure 3.11).

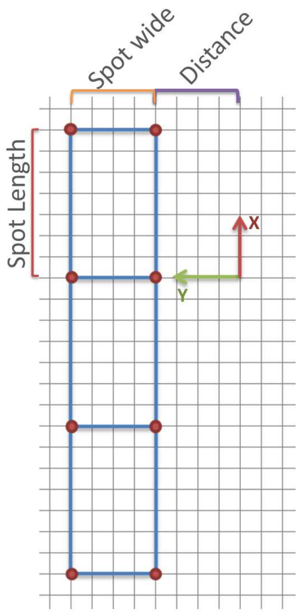


Figure 3.11: Points from convex hull coordinates

At the end, the point clouds resultant from the class "*points_from_volume*" application are published. One of the last parameters is a subscriber, which runs a callback function every time a message of type "*pc_ out_pointcloud*" is received.

Part of the code present in the callback is shown bellow.

Code section 3.6: Callback function

```
#define VEHICLE_FRAME "/vehicle_odometry"

void cloud_cb (const sensor_msgs::PointCloud2ConstPtr & pcmsg_in)
{
    // STEP 1: Create the point_cloud input
    pcl::PointCloud<pcl::PointXYZ> pc_in;
    pcl::fromROSMsg(*pcmsg_in,pc_in);

    //STEP 2: Query for the transformation to use
    tf::StampedTransform transform;
    (...)
        p_listener->lookupTransform(pcmsg_in->header.frame_id, VEHICLE_FRAME, ↩
            ros::Time(0), transform);
    (...)

    //STEP3: transform pc_in using the queried transform
    pcl::PointCloud<pcl::PointXYZ> pc_transformed;
    pcl::PointCloud<pcl::PointXYZ> pc_ahead, pc_spot, pc_behind, pc_ground;
    pcl_ros::transformPointCloud(pc_in,pc_transformed, transform.inverse());
    pc_transformed.header.frame_id = VEHICLE_FRAME;

    // STEP 4: Aplying the ConvexHull class
    pfv.convexhull_function(pc_transformed, 0.0, -0.6, false);
    pc_ahead=pfv.get_pc_in_volume();
    pc_ahead.header.frame_id = VEHICLE_FRAME;
    (...)

    // STEP 5: Convert to ROSMsg
    sensor_msgs::PointCloud2 pcmsg_out;
    pcl::toROSMsg(pc_ahead, pcmsg_out);
    (...)

    // STEP 6: Markers
    visualization_msgs::Marker marker_car;
    marker_car.header.frame_id = VEHICLE_FRAME;// Frame name
    (...)
    marker_car.type = visualization_msgs::Marker::CUBE;// Marker type
    marker_car.pose.position.x = 0.8/2;
    marker_car.pose.position.y = 0;
    marker_car.pose.position.z = spot_high/2;
    (...)

    // STEP 7: Publish markers and PClouds
    car_pub.publish(marker_car);
    (...)
    cloud_pub.publish(pcmsg_out);
    (...)
}
```

There is a transformation applied to the point cloud in order to draw the points on the "*world*" frame. After that, the class "*points_from _volume*" which verifies if there are points inside a certain volume is executed. The returned point cloud is converted into a ROS message to be published later. This callback is also responsible for the construction of the markers which represents the convex hulls latter published.

To migrate the code to another vehicle, the changes to apply are only in the dimensions of the convex hull (they must respect the new vehicle geometry), and the size of markers, which are a representation of the convex hulls. The "_VEHICLE_FRAME_" should be also the same of the one published by the vehicle low level.

3.3.3 Parking coordinates message

As soon as some conditions are met, a message with the parking spot coordinates (position and orientation) is send to the trajectory planner module. The conditions (figure 3.12) necessary to assume that there is a parking space are indicated on the following items.

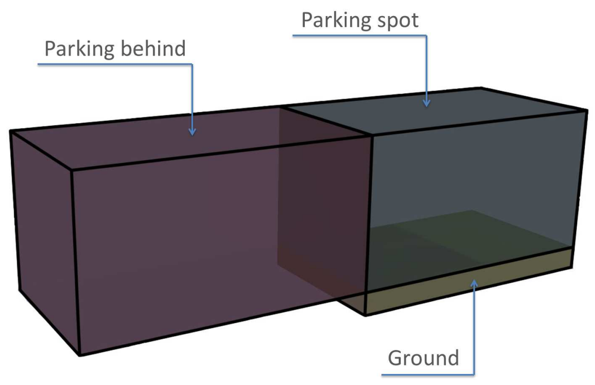


Figure 3.12: Hulls of empty parking spot search

- Parking behind convex hull - This convex hull should have points in it, because it was assumed that to found an empty parking space it is necessary first to find an occupied zone.

- Parking space convex hull - As expected there is a zone were no points should exist. This zone will be the parking space.

- Floor convex hull - This complementary zone determines if the ground has a certain amount of points, because if there are holes on the ground the parking manoeuvre may be very dangerous.

If the conditions are respected, a position message is sent.

This message contains information about the geometric position of the parking spot and the orientation on the base frame - *world* (figure 3.13).

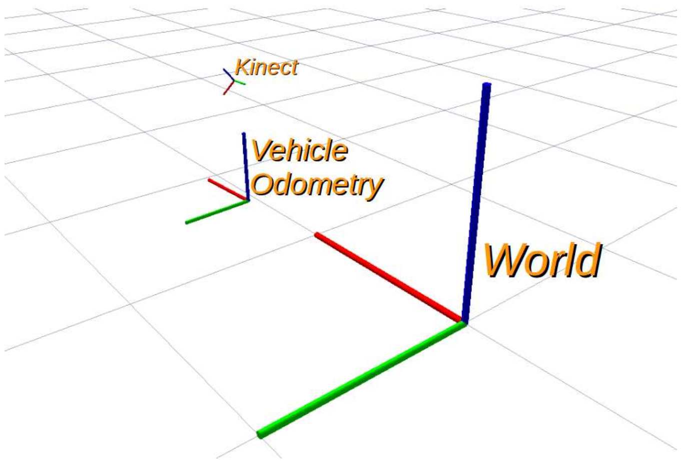


Figure 3.13: Search frames

3.4 Possibilities of package launch

One of the advantages of ROS utilization is the possibility to organize the code in nodes and create “launch files” which just execute some of the nodes.

In the case of the parking spot detection this may be very useful, because of the possibility of replay some recorded situations.

3.4.1 Record data

The launch file ‘*parking_detection_bagrecord.launch*’ has the responsibility of launch the necessary nodes to record a ‘bagfile’ which contain the reconstructed point cloud and the transformations between all the frames.

Code section 3.7: Recorder launch file

```
<launch>
  <node name="openni_node" pkg="openni_camera" type="openni_node"/>
     (...)
  <node name="kinect_freq_mod" pkg="parking_detection" type="kinect_freq_mod" ↩
       args="point_cloud_from_kinect:=/camera/depth/points"/>
  <node name="pub_transformations" pkg="parking_detection" type="↩
       pub_transformations" />
  <node name="pc_accumulation_nodelet" pkg="pc_accumulation" type="↩
       pc_accumulation_nodelet" >
               (...)
  </node>
  <node name="recorder" pkg="rosbag" type="rosbag" args="record /↩
       pc_out_pointcloud /tf -O /home/joel/bag1.bag"/>
</launch>
```

3.4.2 Playback mode

If there is any bagfile stored on disk (just bagfiles recorded with the necessary information), the launcher ‘*parking_detection_bagplay.launch*’ may be executed. This will treat all the incoming data, as if it were happening at the moment. However, it will be just a playback action.

To publish the recorded information there is a ROS node which must be used, the ‘*rosbag play*’.

Code section 3.8: Playback launch file

```
<launch>
<node name="parking_detection" pkg="parking_detection" type="parking_detection" ↩
    />
      <param  name="/use_sim_time"   type="bool"  value="true" />
</launch>
```

3.4.3 Live action

The last launch file is the one responsible to perform live action. It is very similar to the launcher used to record the information. However, on this one there is no need to launch the last node which was only responsible to record the selected information. The '*parking_ detection.launch*' launcher is the one to be used when executing the complete parking manoeuvre (search, plan and execution).

Chapter 4

Planning a manoeuvre

Planing a non-holonomic vehicle trajectory is harder than planning for an holonomic one. This kind of mobile robots, by not being able to rotate around themselves, have positions in space that are simply not achievable (figure 4.1). So, in order to plan a manoeuvre, it is necessary to use some method which define the path for a car-like vehicle.

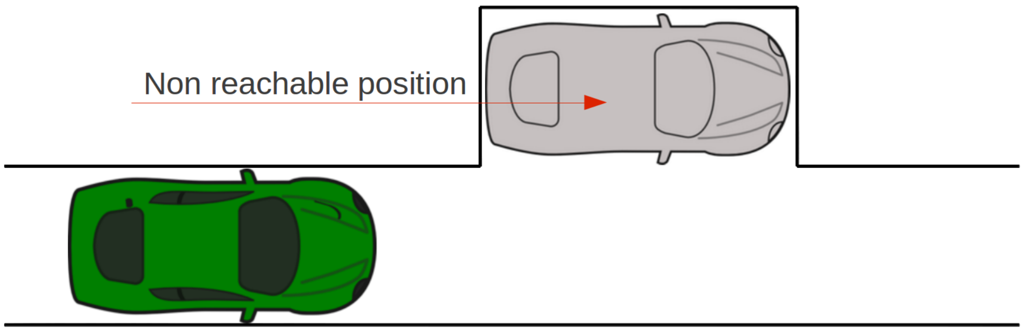


Figure 4.1: Non reachable position to a non-holonomic vehicle

4.1 Planning approaches

The very first ideas of path planning were published in the first International Joint Conferences on Artificial Intelligence (late 60s). Nowadays there are three main families of methods to find out the better path to follow; the ‘roadmap’, the ‘cell decomposition’ and the ‘potential field’ approaches [Laumond et al., 1997, Latombe, 1990].

4.1.1 Roadmap

A roadmap is composed by lines of free space which a robotic vehicle may follow between obstacles (figure 4.2).

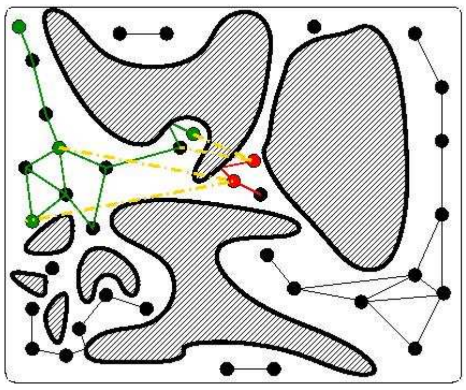


Figure 4.2: Example of roadmap navigation lines [Morales, nd]

As soon as the roadmap is obtained, it is necessary to identify three different paths:

- 1 - From the starting point up to some point on the roadmap;
- 2 - A continuous roadmap path;
- 3 - From the roadmap up to the desired final point.

If these paths exist and are continuous, then there is a path from the initial until the goal point. The principles to the roadmap creation may lay on different techniques, such as visibility graphs, voronoi diagrams, freeway net and silhouette [Latombe, 1990].

4.1.2 Cell decomposition

This method consists of the sub-sampling of the navigation space in small cells, forming the nodes of a connectivity graph. Two nodes are said connected if the cells which represent them are adjacent to each other.

The path (channel) to follow is defined, if there are adjacent cells from the starting point up to the goal point. An optimization algorithm is responsible to ensure that the best possible channel is found [Latombe, 1990, Lingelbach, 2004].

4.1.3 Potential field

The potential fields are obtained by dividing the navigation scene in a small grid, and then calculating the forces applied on the vehicle. Usually, the navigation robot is reduced to a single point and subject to the attraction or repulsion fields. The presence of obstacles will generate repulsion forces on the vehicle, while the goal point produces an attractive field to the robot. The resultant 'virtual' force is responsible to conduct the movement of the robot.

Despite being very efficient when compared to other methods, this technique may lead into a non return situation if a local minima is achieved [Latombe, 1990].

4.2 Trajectory generator

The traditional methods described on the previous section are responsible essentially to perform path planning. Since the AtlasMV is a non-holonomic vehicle, the paths planned should be converted into non-holonomic trajectories. After that it would be necessary to perform the movement with some kind of control to ensure the correct execution of the manoeuvre (complex paths would require an accurate position control).

So, in order to avoid the need for a big precision on the vehicle position estimation, this work will make use of a simple method (based only on circumferences arcs) which is responsible not only for the path planning but also for the execution task. This method will count on a set of predefined typical paths, and the best one, even if not the optimal, will be chosen. Despite the limited number of paths, this will allow a good performance on the trajectory planning algorithm, as will be shown.

Due to the simplicity of the path to follow, there is no strong necessity to perform close loop control, avoiding this way the need to plan alternative trajectories when following the chosen one. However, there is an odometry information which advises the distance travelled by the vehicle, and informs when the vehicle should turn or stop the movement.

This method must be very flexible since it may be used in other simple navigation applications, continuing the work already started by [Oliveira et al., 2012].

The use of the term ‘trajectory’ is sometimes a language abuse, since trajectories have not only geometric information, but also the time details of when each action or changes on velocities happens. On the case of this thesis, there is no time information because all the trajectory planning and execution is made in an open loop control, so what is named here by ‘trajectory’ should be in most of the cases ‘path’. But, the most usual name to define a route is ‘trajectory’, so the nodes and packages created will have this name on them.

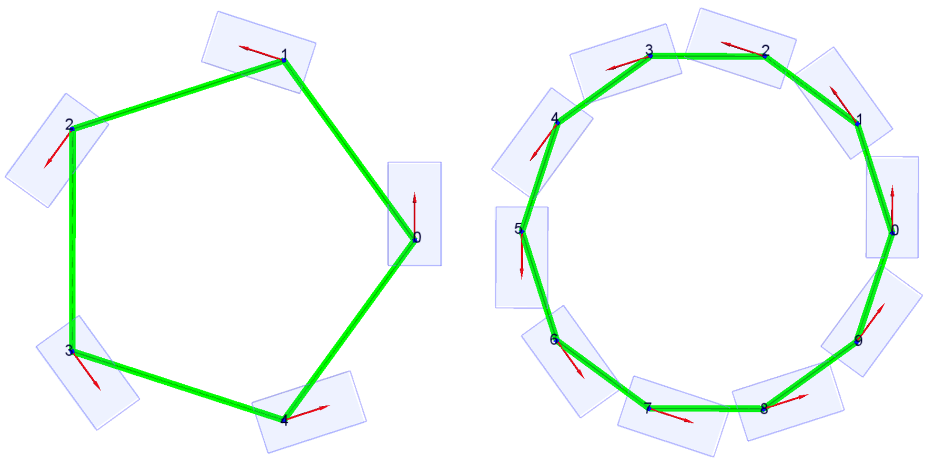
Since the analysis and planning of the trajectories demand the knowledge of the initial and the final states, the choice criteria and the actions to take [LaValle, 2006], a large amount of trajectories is generated, followed by the study of the most adequate one. One simple way to generate non-holonomic trajectories is by using simple geometrical expressions. Those expressions were already defined on the AtlasMV navigation module [Oliveira et al., 2012].

This section will describe the processes and equations needed to generate trajectories for a non-holonomic vehicle using geometric equations.

4.2.1 The input file

The first thing to be done when generating a trajectory with this approach, is the creation of two vectors which contain information about the turning angle of the vehicle and the length of the desired trajectory. In other words, there are just two parameters to define a trajectory, the angle of the turning wheels at each position (α angle) and the distance travelled (arc) with that angle. Those two values need to be combined with the inter-axis dimension of the vehicle to define the trajectory.

Obviously, the planned trajectory will be an approach to the real intended trajectory, once the equations will only bring information about the positions of the vehicle at the calculation points. The bigger the vector with parameters which defines the trajectory, the most accurate to the reality the trajectory will be (figure 4.3). So it is very important to trade off a balance between the accuracy and the computational cost of the trajectory generator.



(a) Trajectory generated with 5 nodes (b) Trajectory generated with 10 nodes

Figure 4.3: Circular trajectories generated with different number of nodes (segments)

The advantage of using a planned trajectory generated with more nodes (segments) will be evident on section 4.3, where the proximity to the real trajectory is fundamental.

4.2.2 Generation on the base frame

The use of some mathematical expressions allows the prediction of a non-holonomic vehicle's position, knowing in advance the turning angle and the distance travelled with this angle.

Usually, non-holonomic vehicles change direction using only one pair of wheels (front or rear pair). The vehicle used for this thesis has front turning wheels, so the car base frame will be the center of the rear axle.

By looking at figure 4.4 it becomes clear that the arc travelled by the vehicle with a certain turning angle, α, has a radius, R, which may be obtained by the use of equation 4.1 [Oliveira et al., 2012].

$$R = \frac{D}{\tan\alpha} \tag{4.1}$$

The parameter D represents the inner-axis length of the used vehicle.

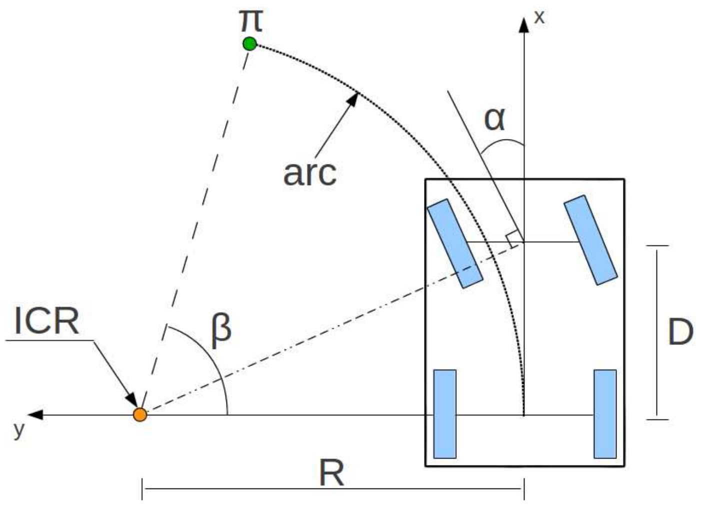
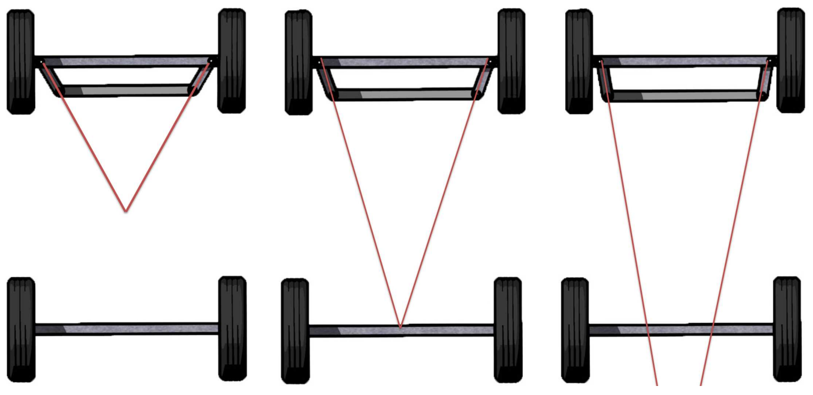


Figure 4.4: Geometric parameters of a car-like vehicle

The value of the turning angle is actually a virtual value, once the wheels do not turn exactly the same angle. This is very usual in car-like vehicles, so in order to obtain a single value of α, the four wheeled car should be reduced to a tricycle model with a single front wheel.

If front wheels spinning is not desired when turning at low speeds, the vehicle should follow a perfect example of an Ackerman model. This means that the prolongation of a line which intersects the center of wheel turning and the point of direction actuation should intersect also the rear axle center (figure 4.5). If it does not happen, wheel spinning may occur because wheels would not be in a tangent line to the circle they must describe.



(a) Supra Ackerman angle (b) True Ackerman angle (c) Sub Ackerman angle

Figure 4.5: The three different types of Ackerman steering

Knowing the value of the instantaneous center of rotation (ICR), and having information about the distance travelled (arc) with a certain turning angle,it becomes possible to calculate the vehicle's orientation (β angle) using equation 4.2 [Oliveira et al., 2012].

$$\beta = \frac{arc}{R} \tag{4.2}$$

With the use of some trigonometry it is possible to determine the x, y position of the vehicle after the movement (equation 4.3) [Oliveira et al., 2012].

$$\begin{bmatrix} \Pi x \\ \Pi y \end{bmatrix} = \begin{bmatrix} R \times \sin\beta \\ R - R \times \cos\beta \end{bmatrix} \tag{4.3}$$

All the position parameters (Πx, Πy, β) are defined in relation to the previous frame of the vehicle's localization. So it is necessary to transpose them to the global frame, by multiplying the new points by all the previous transformations.

Code section 4.1: Frames transformations

```
(...)
for(size_t i=0; i<alpha.size(); ++i)
{
    (...)
    transform.setOrigin(tf::Vector3(lx[i], ly[i], 0));
    transform.setRotation(tf::createQuaternionFromRPY(0, 0, ltheta[i]));
    ltrans.push_back(transform);
    (...)

    for(int j=i;j>=0;--j)
    {
        pcl_ros::transformPointCloud(ponto_teste, ponto_result, ltrans[j]);
        (...)
        ponto_teste.points.push_back(ponto_result.points[0]);
        (...)
    }
    (...)
}
```

First of all, a transformation is created by observing the local positions of the car in relation to the frame. Then, this transformation will be applied from the last to the first node of the trajectory recursively.

After that, it becomes possible to obtain the global coordinates of each trajectory node using just two input vectors which contains the arc and α values (figure 4.6).

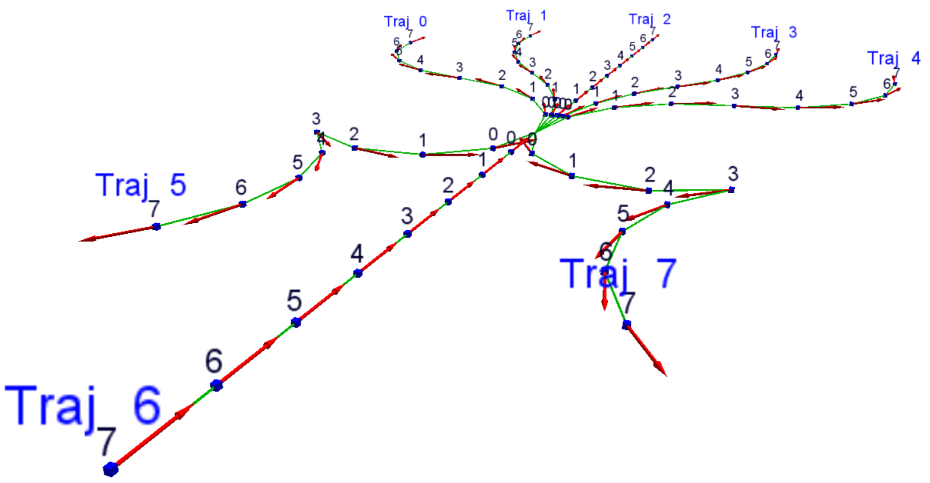


Figure 4.6: Example of a set of trajectories - the first planned position is node 0

4.3 Manage trajectories

Since a large amount of possible trajectories is defined, it is very important to assess which one is better to follow, maximizing the accuracy but having always in consideration possible risks. The next subsections will describe the choice process.

4.3.1 Distance to attractor point

When studying a trajectory it is very important to have an attractor point. This point has in fact the coordinates of the parking spot localization which are sent by the *parking detector* package.

The functions responsible to calculate the trajectory distance to the attractor point only consider the distance to the closest node. This means that the trajectory will only be followed until the closest node, since after that the vehicle would be at a bigger distance of the desired point.

The distance d which each node of the trajectory has to the attractor point (Ax, Ay) is an Euclidean distance (equation 4.4).

$$d = \sqrt{(\Pi x - Ax)^2 + (\Pi y - Ay)^2} \tag{4.4}$$

Logically, the closest node will be the one which has the shortest distance to attractor point.

To evaluate the results obtained, it is better to trust in a normalized value of the distance (equation 4.5).

$$d_{norm} = 1 - \frac{d}{\Delta} \tag{4.5}$$

In this equation, the Δ value is the admissible maximum distance.

Code section 4.2: Distance to attractor point

```
t_func_output c_manage_trajectory::compute_DAP(c_trajectoryPtr& trajectory,↩
    t_desired_coordinates& AP)
{
    trajectory->score.DAP=10e6;
    trajectory->closest_node=-1;

    for(size_t i = 0; i < trajectory->x.size(); ++i)
    {
        double DAP_prev = sqrt( pow(trajectory->x[i]-AP.x,2)+ pow(trajectory->y[↩
            i]-AP.y,2));
        if(DAP_prev < trajectory->score.DAP)
        {
            trajectory->score.DAP = DAP_prev;
            trajectory->closest_node = i;
        }
    }
    (...)
}

t_func_output c_manage_trajectory::compute_trajectories_scores(void)
{
    double maximum_admissible_to_DAP=8.0;
    //normalize DAP
    vt[i]->score.DAPnorm=1-(vt[i]->score.DAP)/maximum_admissible_to_DAP;
}
```

On the code there is a *for* loop which tests the nodes distances to the attractor point. If the distance calculated to the current node is smaller than the ones previously calculated it becomes the shortest distance and the node number is memorized as the closest node.

After that the shortest distance is normalized. In this case the maximum admissible distance is 8.0 meters, being this the only value which needs to be changed when migrating the code to other platforms.

A score of $d_{norm} = 1$ means that the closest node is in fact coincident with the attractor point (figure 4.7).

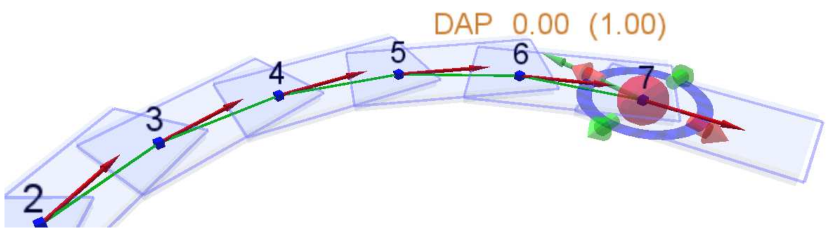


Figure 4.7: Distance to attractor point (DAP) on node 7 = 0

4.3.2 Angular difference

When it comes to a parking manoeuvre, it is important not only the final position (x, y) of the vehicle, but also the orientation. Due to that it is very important to evaluate the angular difference between the orientation that the car will have at the closest node and the intended orientation of the parked vehicle.

The angular difference is the absolute value of the difference of the closest node angle and the intended one (equation 4.6).

$$\delta\theta = |\Pi\theta - A\theta| \quad (mod\ \ \pi) \tag{4.6}$$

Code section 4.3: Angular difference

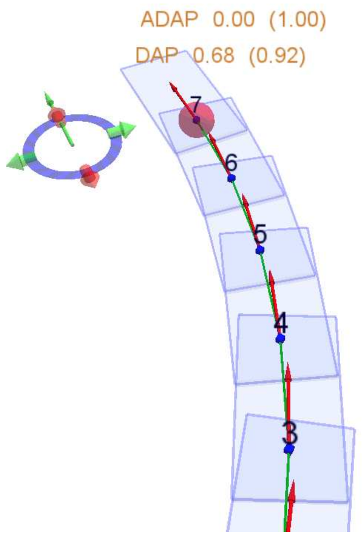
```
double c_manage_trajectory::compute_ADAP(c_trajectoryPtr& trajectory,↩
    t_desired_coordinates& AP, int i)
{
    double adap=abs(trajectory->theta[i]-AP.theta);
    if (adap>M_PI)
        adap=2*M_PI-adap;
    return adap;
}

t_func_output c_manage_trajectory::compute_trajectories_scores(void)
{
    (...)
    //normalize ADAP
    vt[i]->score.ADAPnorm=1.0-(vt[i]->score.ADAP/(M_PI));
}
```

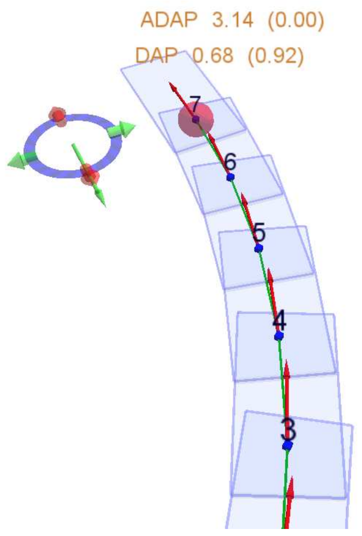
The $\delta\theta$ values must be between 0 to π radians, so to normalize the angular difference to attractor point (ADAP) it is required the use of equation 4.7.

$$\delta\theta_{norm} = 1 - \frac{\delta\theta}{\pi} \tag{4.7}$$

If the value of $\delta\theta_{norm}$ is 1 it means that the angle of the closest node is the same that the one of the attractor point, $\delta\theta = 0$, while a 0 value means that they are with an offset of π radians, $\delta\theta = \pi$ (figure 4.8).



(a) ADAP = 1



(b) ADAP = 0

Figure 4.8: Angular difference to attractor point (ADAP)

4.3.3 Free space

During a manoeuvre, it is very important to ensure that the execution of a trajectory would not guide into possible collisions to obstacles. To be sure that this will never happen a weight function was created to determine if the trajectory is in a collision route to obstacles up to the nearest node.

A collision may be represented by the intersection of two lines. So the car contour is drawn with four lines (approaching the vehicle to a rectangle) in each trajectory node. If these lines intersect other obstacle lines the trajectory has at least a collision point (figure 4.9). In the case of being in the presence of a very small obstacle (a small line), the intersection of this line and the vehicle is ensured by a well discretized trajectory, generating many car contour lines and consequently many collision possibilities. By doing that, the risk of an obstacle be inside of the car contour without touching any line is minimum.

The Laboratory of Automation and Robotics server had already a node (*mtt*) which publishes obstacle lines from an input point cloud of the scene.

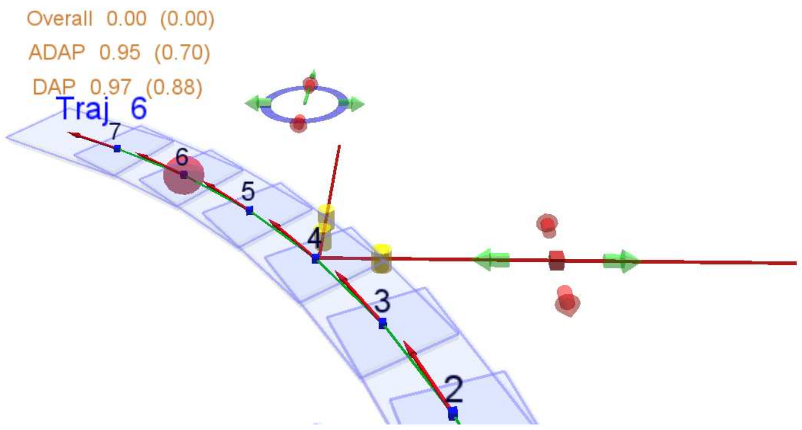


Figure 4.9: Collision points - yellow cylinders

Code section 4.4: Free space

```
//cycle all nodes until the closest node
for (int n=0; n<= trajectory->closest_node; ++n)
{
    (...)
    //cycle all vehicle lines
    for (size_t l=0; l< trajectory->v_lines[n].size(); ++l)
    {
        // Define points from lines
        (...)
        //cycle all obstacles
        for (size_t o=0; o< vo.size(); ++o)
        {
            //cycle all lines inside each obstacle
            for (size_t lo=1; lo< vo[o].x.size(); ++lo)
            {
                (...)
```

```
                    int ret=lineSegmentIntersection(Ax,Ay,Bx, By,Cx, Cy,Dx, Dy,&X, &↩
                        Y);
                    if (ret==DO_INTERSECT)
                    {
                        (...)
                        trajectory->score.FS*=0;
                    }
                }
            }
        }
}
```

During the node execution, all vehicle lines, from the first up to the closest node, are tested with all possible obstacles. If the return of the *lineSegmentIntersection* function (already on LAR's server) is "DO_INTERSECT" then the free space value (FS) will be turned into 0, otherwise it will remain 1.

4.3.4 Distance to obstacles

During a trajectory execution it is not only important to know if there is a collision or not. It is also relevant to determine the minimum distance between the vehicle and obstacles, because a situation may occur where there is no collision but a dangerous passing (very close) in relation to the obstacle.

The distances are measured with the Euclidean distance between the vehicle vertices (rectangle shaped vehicle) and the points of obstacles (points which define the obstacle lines). Here, a well discretized trajectory will be better, since the two points which form a long line (low discretization) may not provide enough information about the closest points of the trajectory.

Code section 4.5: Distance to obstacles

```
trajectory->score.DLO = 10.0;
//cycle all nodes until the closest node
for (int n=0; n<= trajectory->closest_node; ++n)
{
    (...)
    //cycle all vehicle lines
    for (size_t l=0; l< trajectory->v_lines[n].size(); ++l)
    {
        // Define points from lines
        (...)
        //cycle all obstacles
        for (size_t o=0; o< vo.size(); ++o)
        {
            //cycle all lines inside each obstacle
            for (size_t lo=1; lo< vo[o].x.size(); ++lo)
            {
                double DLOprev = sqrt(pow(trajectory->v_lines[n][l].x[0]-vo[o].x↩
                    [lo-1],2)+pow(trajectory->v_lines[n][l].y[0]-vo[o].y[lo↩
                    -1],2));
                    if(trajectory->score.DLO > DLOprev)
                        trajectory->score.DLO=DLOprev;
                (...)
            }
        }
    }
}

t_func_output c_manage_trajectory::compute_trajectories_scores(void)
```

```
{
    double maximum_admissible_to_DLO=10.0;
    //normalize DLO
    vt[i]->score.DLOnorm=(vt[i]->score.DLO)/maximum_admissible_to_DLO;
}
```

On the code it is defined a value from which the distance of closest obstacle became no longer relevant. In this case it was established to 10 meters (this parameter may change in different situations or vehicles). During the node execution the *for* cycles look for a lower value of the distance to obstacles (DLO), being the lower value the one that is saved.

After that, the value must be normalized. This is achieved by dividing the lower DLO value found by the maximum admissible (Φ) (equation 4.8).

$$DLO_{norm} = \frac{DLO}{\Phi} \tag{4.8}$$

4.3.5 Measurement function

After having defined the evaluation criteria it is necessary to choose the weights to generate the best trajectory to follow. Those values (table 4.1) were manually optimized to the current situation and vehicle, and must be updated if any of these two constrains change.

Table 4.1: Trajectory evaluation weights

Parameter	DAP_{norm}	$ADAP_{norm}$	DLO_{norm}
Weight	40%	35%	25%

The *free space* (FS) will represent 100% of the decision since trajectories with collisions are not even considered. The trajectory score will be given by the formula present on equation 4.9.

$$Score = FS \times (0.40 \times DAP_{norm} + 0.35 \times ADAP_{norm} + 0.25 \times DLO_{norm}) \tag{4.9}$$

Only trajectories with a final score higher than 75% must be considered, since the parking manoeuvre must be something that requires some precision.

The chosen trajectory may be seen highlighted on ROS *rviz*, and in the case of figure 4.10 the one to follow will be "traj 2", since trajectory 3 (the closest one) has collision points with an obstacle in its intermediate nodes.

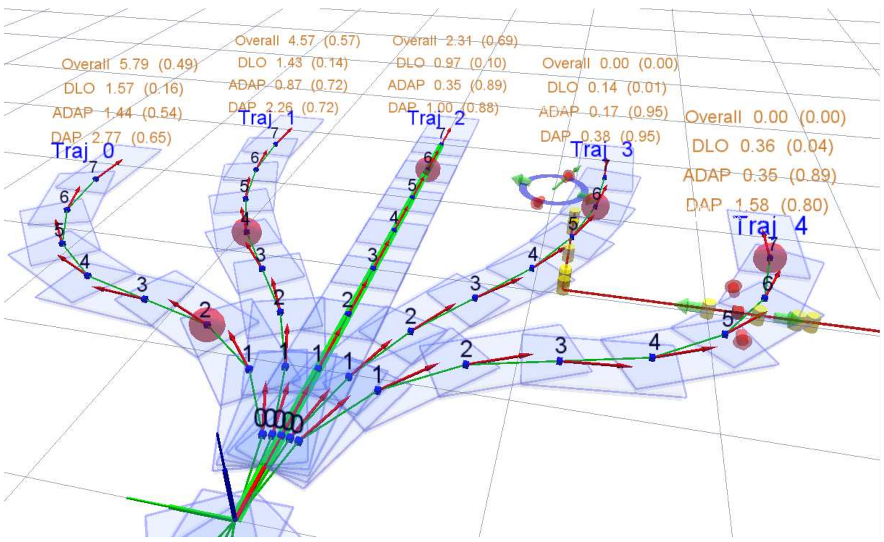


Figure 4.10: Chosen trajectory - traj 2

4.4 Trajectory execution message

As soon as a trajectory is chosen, the *trajectory_planner_nodelet* node send a message with two vectors containing information about the turning angle values (α) and each trajectory segment speed. The speed vector will be positive or negative according to the arc value being positive or negative.

Code section 4.6: Speed modulator

```
#define SPEED_REQUIRED 0.25
#define SPEED_SAFETY 0.05

vector<double> set_speed_vector(boost::shared_ptr<c_trajectory> t)
{
    vector<double> speed_setted;
    for(size_t i=0;i<_NUM_NODES_;++i)
    {
        if(i < (_NUM_NODES_ - 1))
        {
            if((t->arc[i])*(t->arc[i+1])<0.0)
                speed_setted.push_back((t->arc[i]/fabs(t->arc[i]))*SPEED_SAFETY)↩
                    ;
            else
                speed_setted.push_back((t->arc[i]/fabs(t->arc[i]))*↩
                    SPEED_REQUIRED);
        }
        else
            speed_setted.push_back((t->arc[i]/fabs(t->arc[i]))*SPEED_REQUIRED);
    }
    return speed_setted;
}
```

The vehicle will travel at a required speed (0.25 m/s), positive or negative. However, there is a situation when the vehicle must travel at a safety speed (0.05 m/s). This happens when the product of two consecutive arc values is negative, meaning that there will be a change from reverse to forward (or the opposite) movement. So to avoid sudden accelerations the trajectory between this two nodes is travelled at lower speeds (figure 4.11).

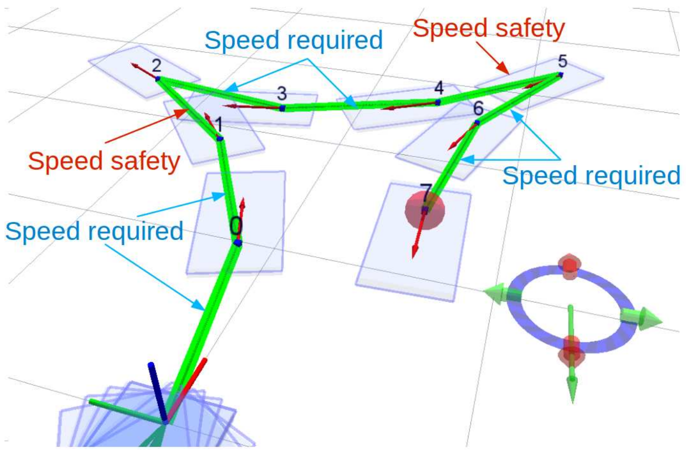


Figure 4.11: Speed modulation on forward/reverse movement

The *trajectory_ executive* node is responsible to send the command message to the low level control of the vehicle.

Code section 4.7: Command message

```
void send_command_message(int current_node)
{
    if(current_node != −1 && static_cast<int>(info.alpha.size())>current_node)
    {
        command.dir=info.alpha[current_node];
        command.speed= info.speed[current_node];
        command.lifetime=INFINITY;
        (...)
    }
    else
    {
        command.dir=0.0;
        command.speed=0.0;
        command.lifetime=INFINITY;
        (...)
    }
    command.header.stamp=ros::Time::now();
    commandPublisher.publish(command);
}
```

With the information from the vehicle odometry it is possible to determine the travelled distance and, consequently, the current node of the trajectory. Knowing that, this executable only has to send the command message related to the actual node.

When the last node is reached, a message sends a stop command to the vehicle (0 m/s of speed and 0 radians of direction).

All the parking trajectories were defined in offline mode. Those were very similar, changing only the steering angle, or the distance travelled.

4.5 Mobile robot package

Generally, the search, planning and trajectory execution requires some cooperation and exchange of information between nodes and packages. Due to that, the launching process of the autonomous parking module must be well explained. The next sections will describe the important details of the nodes to launch.

In this work, all the tasks were performed by the AtlasMV robot which already had a developed communication module. The only thing needed to do was the code migration to ROS, since this robot previously used another architecture, *Carmen* [CarmenTeam, 2012].

After the code migration, the fundamental structure of the "AtlasMV package" remained the same, having a central node which is responsible to publish vehicle information messages, like the odometry, and to treat the received command orders which will actuate the motors.

4.5.1 Gamepad control and priority messages

It is possible to send control messages to the AtlasMV vehicle using a remote control device. This advantage was used not only to control the robot on some simulation situations, but also to implement a module responsible to perform the movement of the vehicle when searching for an empty parking spot.

Since no search navigation module was created until the date, it was necessary to generate a simple one that would enable the start of the robot movement. This would emulate the situation when a driver passes next to a line of parked cars.

The created module makes that when the 'Y' button of the gamepad (figure 4.12) is pressed, a control message of straight forward movement at a speed of 0.2 meters per second is sent to the *Atlasmv_ base* node.

Code section 4.8: Gamepad control message

```
void GamepadSearch(int value,void*parameters)
{
    atlasmv_base::AtlasmvMotionCommand command_local;
    if(value==1)
    {
        command_local.dir=0;
        command_local.speed=0.2;
        command_local.lifetime=INFINITY;
        command_local.priority=1;
        command_local.header.stamp=ros::Time::now();
        commandPublisher.publish(command_local);
    }
    else
    {
        command_local.dir=0;
        command_local.speed=0.0;
        command_local.lifetime=0.1;
        command_local.priority=1;
        command_local.header.stamp=ros::Time::now();
        commandPublisher.publish(command_local);
    }
}
```

As it may be seen on the code presented in code section 4.8, a message of priority 1 is sent every time the 'Y' button is pressed or released. The use of priority numbers has the same effect on control messages as traffic lights have on traffic regulation. In case multiple command messages arrive at the same time, the vehicle will respect the one with higher priority.

The search module created publishes low priority messages. When a parking spot is found, a command message to follow the parking manoeuvre is sent with a priority of 2, so the vehicle will respect this message and ignore the forward search movement.

Pressing any other buttons of the gamepad activates a message with priority 3. These buttons may act as an 'emergency interrupt', since the vehicle will ignore the previous message and respect the new command.

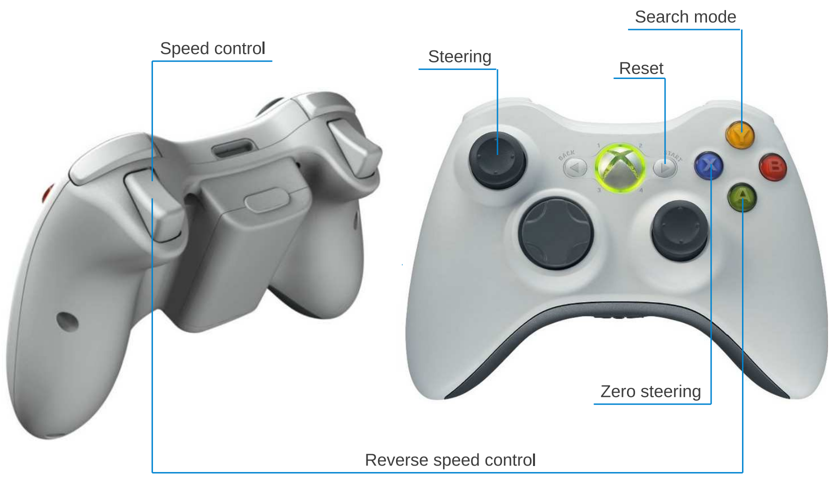


Figure 4.12: Gamepad control buttons

After choosing the best trajectory to follow, the vehicle starts its movement, assuming that all the action is being performed correctly due to the open loop control of the process. However, the approach is compatible with changing environment conditions in case that it can be monitored. In those situations new paths can be continuously recalculated.

4.5.2 AtlasMV launcher

Since the search for an empty parking spot assumes the utilization of a car, it is necessary to launch the package that ensures the communication with the vehicle. In this case, this is achieved by executing two different launch files, the *pre_ launch_ atlasmv.launch* and the *atlasmv_ base.launch*.

The first one is responsible to guarantee that all the vehicle connections are properly made by doing a mapping of all the devices connected (*e.g.* gamepad).

The other launcher is the one which activates the remote control and the *atlasmv_ base* nodes allowing the control of the vehicle and the broadcast and subscription of some informations messages.

With the use of these two launch files it is possible to obtain all the informations needed to perform the point cloud reconstruction and to follow accurately the chosen trajectory.

4.6 Generated trajectories

After the initiation of the search module, and as soon as the node responsible to detect empty parking spot locations send the coordinates message the process which will choose the best trajectory to follow comes into operation.

The trajectory to follow is chosen from a large amount of trajectories possibilities defined since the beginning of the search process. Those trajectories are created in a way that they must converge into the same goal zone. Since the parking spot is always detected approximately in the same region, in theory the approach trajectories may end at a very similar position and orientation.

To define a trajectory there must be created two vectors of the same size. One for the turning angle values (α, in degrees), and the other for the arc distance travelled with that angle (arc, in meters). The α angle must be limited by the maximum turning angle of the used vehicle, in this case that value was 21 degrees in each direction.

The trajectories generation was an iterative process, planning random parking manoeuvres movements, and then overlap them with the reconstructed point cloud. When a reasonable trajectory was found, some small variations were applied to other path parameters, resulting a set of trajectories similar to the one on figure 4.13.

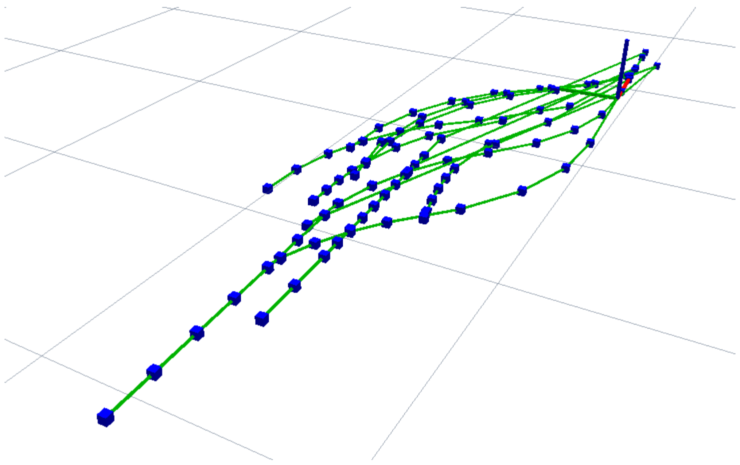


Figure 4.13: Some of the generated trajectories

The node responsible to evaluate all the trajectories weights chooses the one with higher value. If this value is above a predetermined parameter (75% in this case) a control message with priority number of 2 is sent.

4.7 Communication scheme

When performing the complete parking manoeuvre (search, plan and execution) there are nodes from multiple packages exchanging information messages. A simplified scheme of communications is shown on figure 4.14.

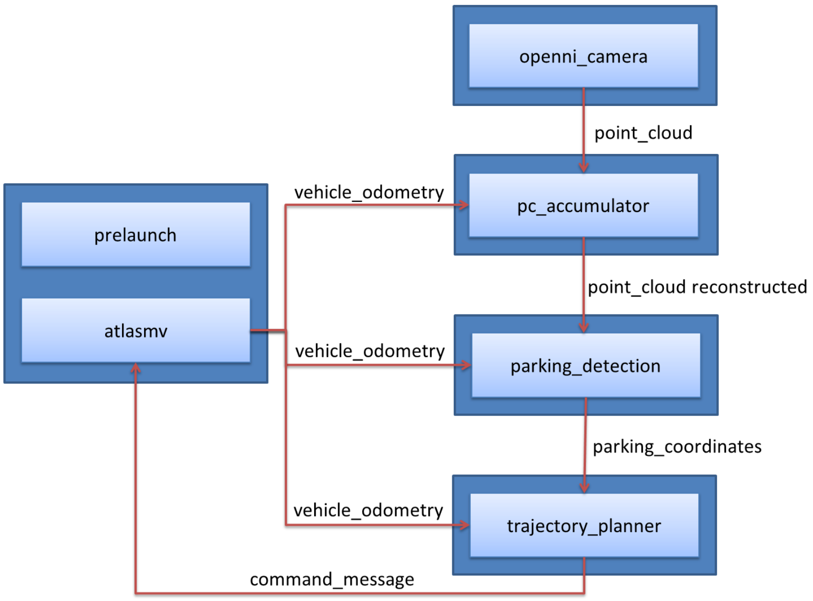


Figure 4.14: Simplified Communication scheme

This types of schemes are very useful to understand the paths which messages follow when a certain process is being executed.

Part III

Results and Discussion

Chapter 5

Experimental results

When programming, all executable nodes must be tested experimentally to confirm the results. There are always little details that are not perceptible on code writing but appear to be evident on the field trials.

5.1 Vehicle odometry calibration

In order to reconstruct the point cloud of the surrounding environment and to follow correctly the planned trajectories, the odometry information should be the most accurate possible.

It was necessary to calibrate the odometry message sent by the AtlasMV vehicle, since initially it was not very precise.

To achieve satisfactory results, the first test carried out was the speed calibration. Here, the vehicle must travel in a straight line for four meters and the odometry should exhibit the same movement. After a few attempts and some changes on the calibration parameter, the result achieved was the one present on figure 5.1.

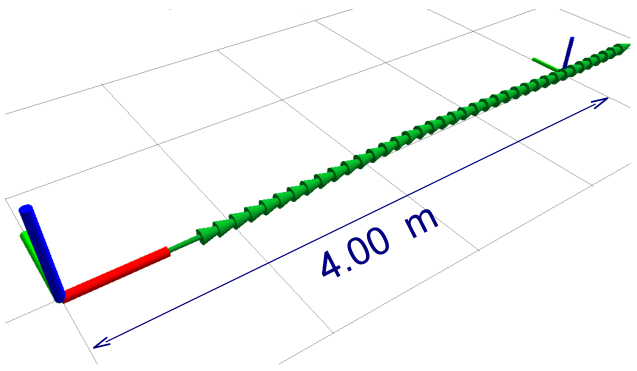
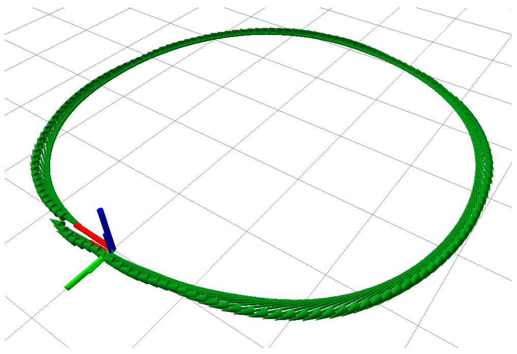


Figure 5.1: Straight forward motion to calibrate the odometry

Having reached a satisfactory straight line movement, it became necessary to calibrate the direction parameters. It was possible by controlling (manually) the vehicle in a circular movement (clockwise and anticlockwise) and stop the vehicle in the same position were the movement began. The odometry should confirm that the initial and final position have the same coordinates and orientation. The results achieved (figure 5.2) were very accurate, since the vehicle movement and the odometry information were in an acceptable harmony.


(a) Clockwise movement

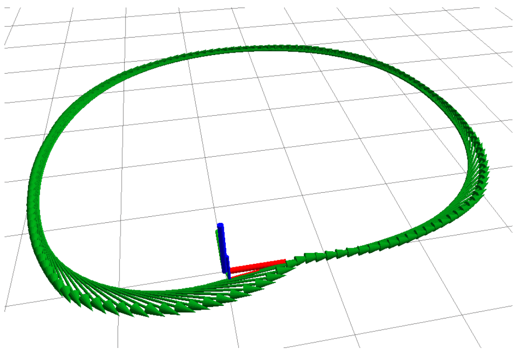
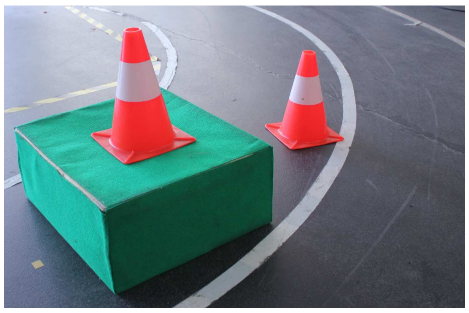

(b) Anticlockwise movement

Figure 5.2: Final odometry calibration

5.2 Point cloud reconstruction

Since the horizontal field of view of the Kinect was too short to catch all the scene information necessary to found a parking space, it was required to reconstruct the point cloud while the vehicle was moving forward.

As it was said on section 3.2.3, the reconstruction was performed by an existing package on the LAR's server. The result of the reconstruction may be seen on figure 5.3.


(a) Scene to reconstruct

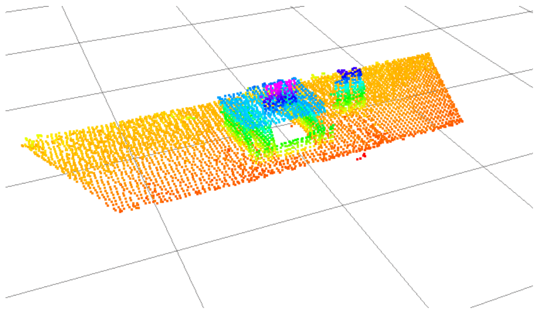

(b) Reconstructed point cloud

Figure 5.3: Point cloud reconstruction

A small error on vertical walls was detected on all the reconstructions. This may occur due to the large amount of information combined with small odometry imperfections, resulting on the representation of the same obstacle in different positions.

These overlaps, on small scale vehicles like the AtlasMV, generate a considerable error (± 0.20 m) but on full scale vehicles this error will not be so representative.

5.3 Search for an empty parking spot

In order to begin the search for an empty parking spot, it was necessary to press the gamepad ‘Y’ button which sent a message to the vehicle to move straight forward. If, during the movement, the robot finds a parking spot, it sends a message which contains an attractor point.

Usually, non-holonomic vehicles only fit into spaces with approximately 1.5 times its length. However, the AtlasMV has a low limit on the turning angle (0.37 radians), so the aimed parking spot should be at least 1.8 times bigger than the length of the car, meaning the search for empty spots of 1.5 meters in length.

Since in this work a parking space is defined as something after another car (or obstacle), there are two possibilities for parking situations: In front of a parked vehicle (type I) and between to cars (type II).

The results of the search for an empty parking spot, in both situations are described in the next subsections.

5.3.1 Type I parking situation

While moving forward, and analysing the point cloud which has been reconstructed, a parking spot is found in the coordinates represented by the green sphere (figure 5.4). These coordinates represent the position of the rear axle of the vehicle in the desired position. The identified parking spot is in front of a vehicle, because the marker ahead has no points inside it, meaning no obstacles in front of the possible parking space.

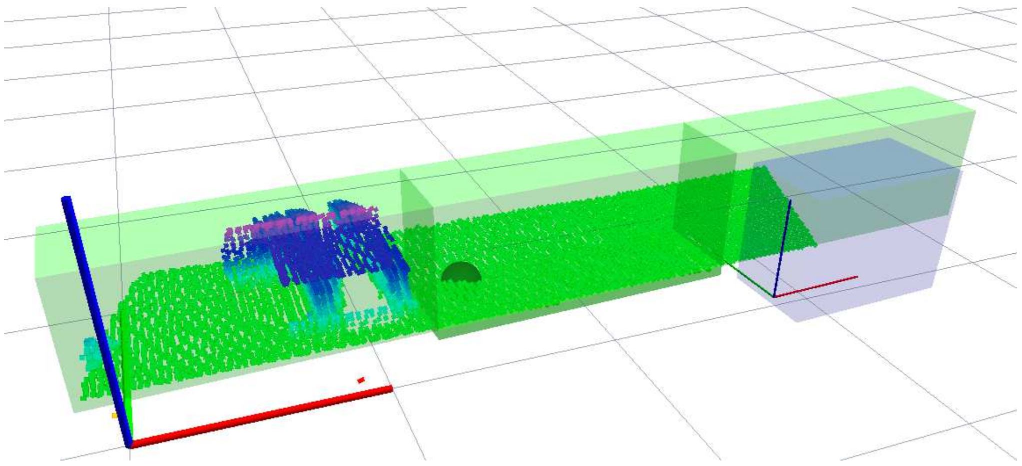


Figure 5.4: Parking detection - type I

5.3.2 Type II parking situation

This parking situation is similar to the described previously except for the fact that in this case, the parking space found is between two vehicles (figure 5.5).

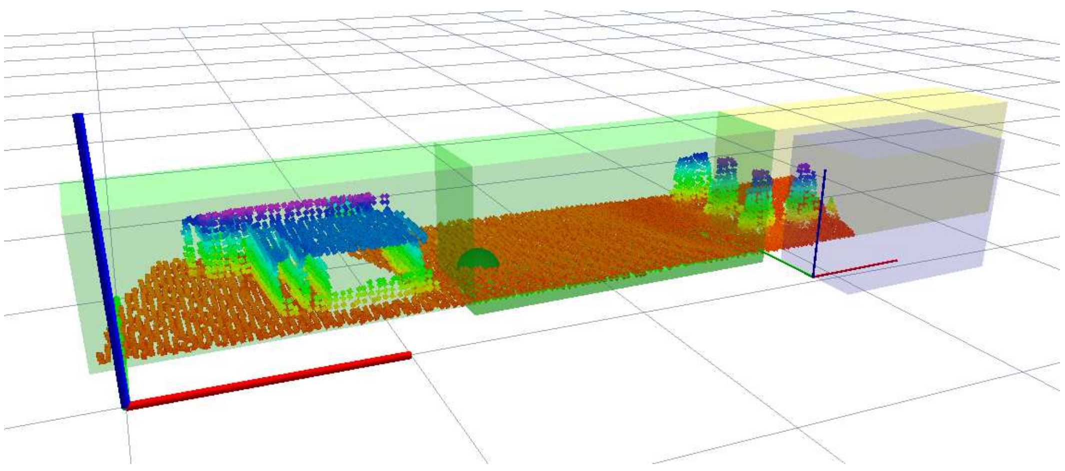


Figure 5.5: Parking detection - type II

5.4 Processing time

As it was said on section 4.2, one of the advantages of using predefined paths (offline planning) is the lower computational cost when compared to online planners. In the present case, the computational time increases linearly with the number of generated trajectories. The results achieved on the trajectories evaluation on a quad-core 2.80 Ghz processor were the ones shown in figure 5.6.

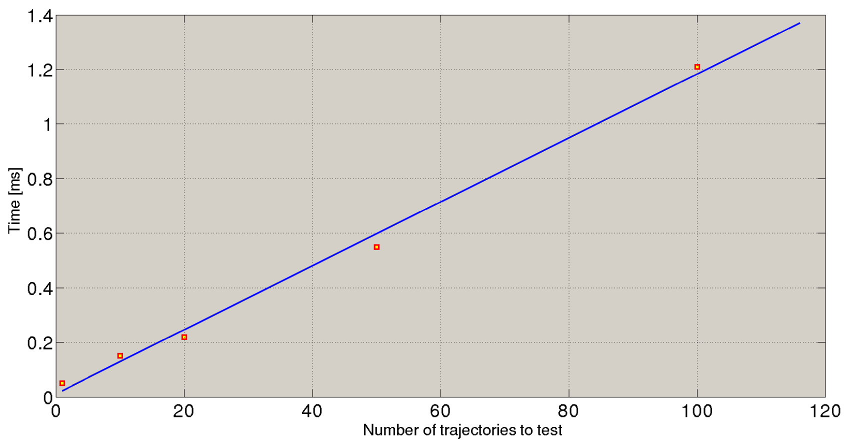


Figure 5.6: Processing time on trajectories evaluation

The trajectories evaluation module is performed only one time per parking spot found, making this process fast enough to perform a parking manoeuvre.

5.5 Trajectories *vs* accuracy

Making a vehicle to follow a trajectory requires a very accurate and predictable odometry information and vehicle control.

Since the AtlasMV robot was not built accordingly to the Ackerman model (figure 5.7), a small deviation between the trajectory planned and the executed due to the front wheels skid was expected.

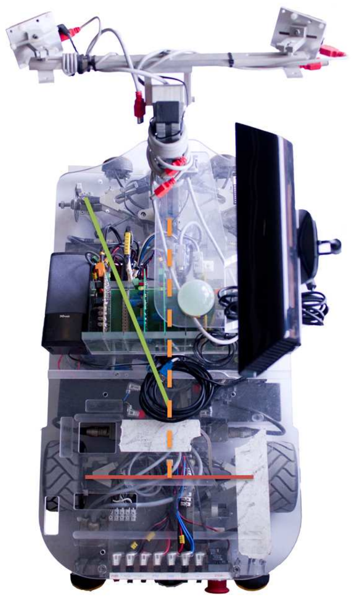


Figure 5.7: AtlasMV Ackeman angle

The test carried out to verify the 'amount' of trajectory deviation consisted of the plan of a wide circular trajectory (diameter of four meters) and the overlap of the vehicle odometry while completing the path. The test was performed with the robot lifted up the ground, which made the wheels to spin freely without touching the ground. The results are shown on figure 5.8.

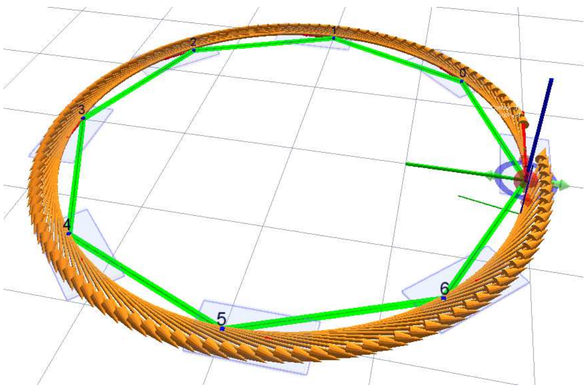


Figure 5.8: Circular trajectory planed and executed (lifted up the ground)

As expected, the results were very accurate, since the odometry was properly calibrated and the front wheels did not has the chance to skid because they were lifted. So the next test to do was the same as the previous, but now with the vehicle on the floor.

The results were a little different from the first essay (figure 5.9).

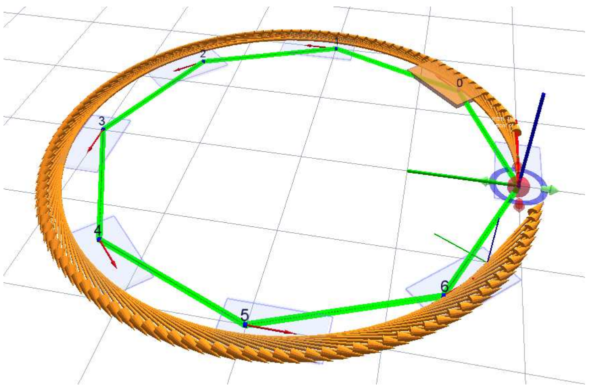


Figure 5.9: Circular trajectory planed and executed (on the floor)

It was observed that robot performed a little wider trajectory than the one planned, indicating skid from one of the front wheels. Another factor that may have contributed to the trajectory disparity was a loose wheel support which made the left front wheel to get off the path planned.

However, the results obtained only began to become less accurate at longer distances travelled, so in short trajectories they should be good enough. Even in a long 'S' shaped trajectory all the results were pretty accurate (figure 5.10).

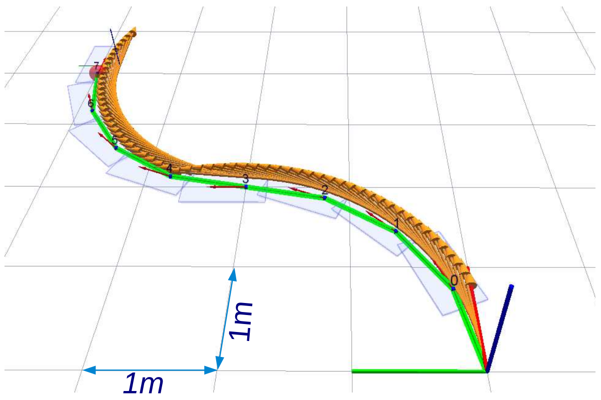


Figure 5.10: 'S' shaped trajectory

On this king of trajectory, where there are an ample turning angle change, the trajectory performed will not be composed by two tangent arcs. This is due to the fact that when the change of direction occurs, the vehicle is still moving, generating a clothoid shaped trajectory (figure 5.11). However, this effect is not noticeable if the manoeuvre is performed at low speeds.

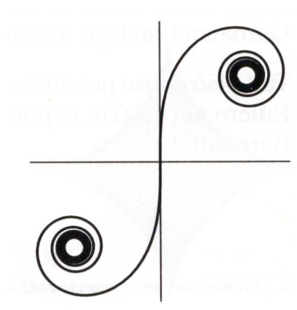


Figure 5.11: Clothoid path[Nasa, nd]

5.6 Trajectory weights

To evaluate the generated trajectories, it was necessary to determine the weight that each evaluation parameter should have on the final score.

For safety reasons, it was decided that the free space (FS) parameter, should be multiplied by all the other parameters, and if a collision is detected it will turn the global score of the trajectory to 0. Otherwise the trajectory will depend on the other parameters weights.

In a first approach, the remaining three evaluation parameters were weighted equally, 33.3%. However, and after a few parking simulations, it became clear that the distance to the attractive point (DAP) should be more important than the angular difference (ADAP), since a parked vehicle may have small orientation deviations.

The DLO (distance to obstacles) parameter should have less weight than the other two, because this parameter will only choose the trajectory which passes at a longer distance from obstacles, and in parking manoeuvres, sometimes this is inevitable.

So, after empirical trials, the final results achieved were 40% for the DAP, 35% for the ADAP and 25% for the DLO.

5.7 Parking manoeuvre

As soon as the attractive marker which represents a parking spot is received, some pre-defined trajectories are generated on the current point of the vehicle. Those trajectories were already established on a file which contained the information about the arc values and the α angles.

After the generation, all trajectories are evaluated, and the one with the highest score, if above 75%, is executed.

When the execution message starts to be followed, the trajectory search button (‘Y’ button of the gamepad) may be released, since from that moment there are messages with higher priority.

The trajectory is perfectly followed at the desired speeds until the closest node. All the small variations are due to some loose suspension, and a front end of the vehicle which has not been built accordingly to the Ackerman model of a car.

On both types of parking manoeuvre the node responsible to the execution is the same, being the obstacle locator the responsible to avoid a collision with the front car (in the case of a type II manoeuvre) by choosing another optional trajectory to follow.

On figure 5.12, it may be seen a type I parking manoeuvre, whereas figure 5.13 represents the second type.

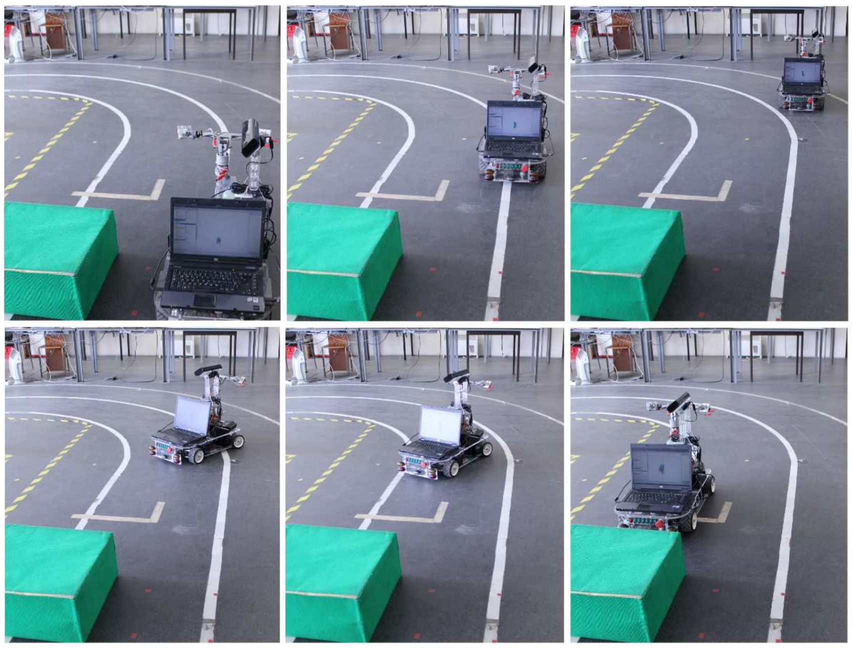


Figure 5.12: Parking manoeuvre - type I

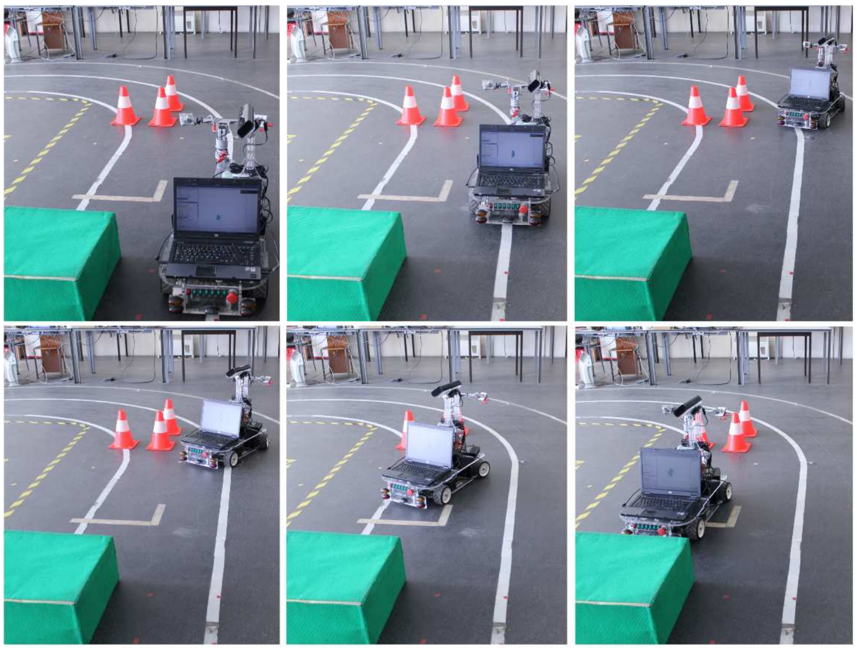


Figure 5.13: Parking manoeuvre - type II

Chapter 6

Conclusions and Future Work

Nowadays, car brans are delivering to the market new types of aids to perform the parking manoeuvre in a semi-autonomous way. So, it became clear that a fully autonomous car should be capable to auto park itself.

The main goal of this work, which was the detection and execution of a parking manoeuvre, was successfully achieved despite the small perception errors. The use of a small scale vehicle was very useful, since the risk of collisions and equipment damage was more controllable.

In respect to the command messages it became clear that the adopted philosophy (messages with different priorities) has proved to be very useful in the management of the commands to follow.

The trajectory planner approach revealed to be fast and quite accurate for the type of required manoeuvre. The chosen path is followed in a 'blind' way, since there were no sensors to control the rear empty space of the vehicle. However, if this information was available, the trajectory monitoring and re-planing would be of easy implementation. Another easy implementation task is the code migration in case of platform changing, like the implementation of the autonomous manoeuvre on the Atlascar.

Since the parking manoeuvre is so vast, the execution of this work opened doors to the realization of new tasks. Either for development or small corrections, all the tasks that need to be done are extremely important to the ATLAS project integrity.

6.1 Parking search module

The search for a parking space assumes that there is a module responsible to direct the attention from the road ahead to the parked cars on the side of it. This change of point of interest may be called as the parking navigation module.

In this work, a very basic navigation module was created, to perform the parking spot search. So, in order to explore the parking manoeuvre in all its potential a module dedicated to the parking navigation should be created, keeping well located not only the area of parked cars, but also the presence of traffic signs which could give informations about the parking spot nature, or even parking regulations.

6.2 Environment reconstruction

The Kinect point cloud reconstruction achieved on this work was not totally satisfactory, since the reconstruction had errors on vertical walls. In order to perform a perfectly 'clean' parking manoeuvre, everything from the reconstruction to the execution must be "perfect", so the *point_ cloud_ accumulator* nodelet should be improved to assume high rate information and a better correlation between the point clouds and the vehicle odometry. If the errors pursuit, a new principle of reconstruction may be started, not from the odometry information, but by merging the point clouds with obstacles comparison.

6.3 Steering optimization

The use of a small scale vehicle was very useful, not only to perform a few essays, but also to simulate some manoeuvres with lower risk than if they were performed with a full scale prototype. So, despite the existence of the Atlascar, the use of this 'little' robots should continue.

There are however, a few improvements to be made on the AtlasMV vehicle. As it was explained on section 5.5 this vehicle was not built accordingly to the Ackerman model of a car, so the trajectories executed may be a little different from the ones planned.

To make this vehicle correct from the point view of Ackerman, there is only one modification that needs to be done. The steering actuator must be replaced by one which acts slightly to the outside than the current one (figure 6.1).

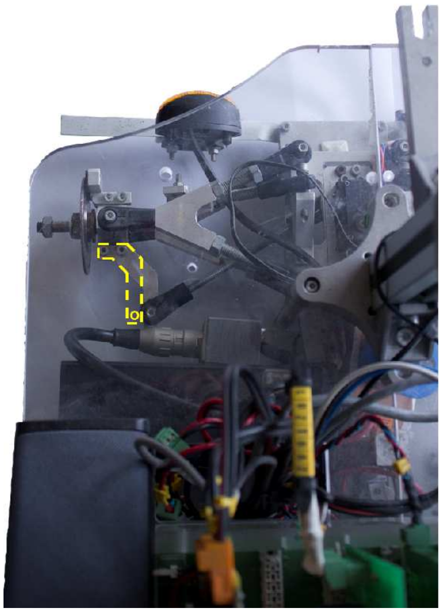


Figure 6.1: Corrected direction actuator position

Another correction to be made on the AtlasMV is the tune of the front suspension, since there are some screws with small gaps, making all the front end of the car inaccurate.

6.4 Empty parking space variations

In a parking manoeuvre it is very important to have a large range of options of which final parked position the car should assume. In real live situations vehicles are not parked always in the same configuration. So, it would be very useful to have a module which allows the detection of several parking spots arrangements, like the perpendicular, angled, and even in the side-walk when the situation demands so.

This module would make the search for an empty parking spot more vast and flexible, being more reliable to the human being actions.

6.5 Code migration

Possibly, the essay which should generate more social impact would be the autonomous parking manoeuvre applied to the full scale prototype, the Atlascar. This kind of manoeuvre being performed by a full size vehicle shows that the process is closer to its final phase. So, as soon as the gear box actuator of the Atlascar is ready, the code migration should start, since there are some parameters that demand some measurements and essays to be fully operational.

This migration would close a cycle which started this year with the application of the algorithm and manoeuvre to a small scale prototype.

References

[Ackerman, 2010] Ackerman, E. (2010). Google autonoumous car. Available from: http://www.botjunkie.com/2010/10/12/googles-autonomous-car-takes-to-the-streets/. Accessed March 2012.

[Aloimonos, 1997] Aloimonos, Y. (1997). *Visual Navigation: From Biological Systems To Unmanned Ground Vehicles (Computer Vision Series)*. Routledge.

[ATLAS, 2011] ATLAS (2011). Atlas project - hardware setup. Available from: http://atlas.web.ua.pt/hardwaresetup.html. Accessed March 2012.

[AutomotiveAdicts, 2006] AutomotiveAdicts (2006). The automotive adicts blog. Available from: http://automotiveaddicts.com/blog/2006_09_01_archive.html. Accessed May 2012.

[Baker et al., 2006] Baker, C., Debrunner, C., Gooding, S., Hoff, W., and Severson, W. (2006). Autonomous vehicle video aided navigation; coupling INS and video approaches. In *Proceedings of the Second international conference on Advances in Visual Computing - Volume Part II*, pages 534–543, Berlin, Heidelberg. Springer-Verlag.

[Behringer, 2007] Behringer, R. (2007). Vamp car - 1986. Available from: http://www.flickr.com/photos/reinholdbehringer/1919685529/. Accessed January 2012.

[Bertozzi et al., 1998] Bertozzi, M., Broggi, A., Conte, G., Fascioli, A., and Fascioli, R. (1998). *Vision-based Automated Vehicle Guidance: the experience of the ARGO vehicle*. World Scientific Publishing.

[Bosch, 2011] Bosch (2011). Automotive electronics parking made easy - parking assistance systems from Bosch. Available from: http://www.boschautoparts.com/parkassist/pages/parkassist.aspx. Accessed May 2012.

[Brown, 1934] Brown, R. J. (1934). Four wheel on jacks park car. *Popular Science*, page 58.

[CarmenTeam, 2012] CarmenTeam (2012). Carmen robot navigation toolkit. Available from: net/intro.html. Accessed June 2012.

[Chiafulio, 2010] Chiafulio, M. (2010). The history of autonomous vehicles: How far we've come. Available from: http://www.mikechiafulio.com/RIDE/history.htm. Accessed January 2012.

[Darpa, 2007] Darpa (2007). Darpa grand challenge. Available from: http://archive.darpa.mil/grandchallenge05/. Accessed February 2012.

[Fisher, 2012] Fisher, M. (2012). Matt's webcorner. Available from: http://graphics.stanford.edu/~mdfisher/Kinect.html. Accessed May 2012.

[FNR, 2012] FNR (2012). Festival nacional de robotica. Available from: http://www.robotica2012.org/12/. Accessed May 2012.

[Fossati et al., 2011] Fossati, A., Schonmann, P., and Fua, P. (2011). Real-time vehicle tracking for driving assistance. *Machine Vision and Applications*, 22(2):439–448.

[Hikita, 2010] Hikita, M. (2010). An introduction to ultrasonic sensors for vehicle parking. Available from: http://www.newelectronics.co.uk/electronics-technology/an-introduction-to-ultrasonic-sensors-for-vehicle-parking/24966/. Accessed January 2012.

[Hudson, 2008] Hudson, T. (2008). Winer of the 2005 darpa grand challenge. Available from: http://cache.jalopnik.com/assets/images/12/2006/10/darpa_stanley.jpg. Accessed February 2012.

[INRIA, 1998] INRIA (1998). Rapport d'activite scientifique 1997. In *Theme INRIA 3B, INRIA Projet SHARP*.

[Jochem et al., 1995] Jochem, T., Pomerleau, D., Kumar, B., and Armstrong, J. (1995). Pans: A portable navigation platform. In *IEEE Symposium on Intelligent Vehicles*, pages 107 – 112.

[Klette and Liu, 2008] Klette, R. and Liu, Z. (2008). Computer vision for the car industry. In *Multimedia Imaging Report 9*.

[Koifman, nd] Koifman, V. (n.d.). Image sensors world. Available from: http://image-sensors-world.blogspot.pt/2010/06/panasonic-announces-d-imager-3d-imager.html. Accessed March 2012.

[Latombe, 1990] Latombe, J. (1990). *Robot Motion Planning*. Springer.

[Laumond et al., 1997] Laumond, J., Sekhavat, S., and Lamiraux, F. (1997). *Guidelines in Nonholonomic Motion Planning for Mobile Robots*.

[LaValle, 2006] LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.

[Lingelbach, 2004] Lingelbach, F. (2004). Path planning using probabilistic cell decomposition. In *in Proc. of the IEEE Int. Conf. on Robotics and Automation*, pages 467–472.

[Markoff, 2011] Markoff, J. (2011). The New York Times - Google Lobbies Nevada to Allow Self-Driving Cars. Available from: http://www.nytimes.com/2011/05/11/science/11drive.html?_r=1&emc=eta1. Accessed January 2012.

[Matos, 2003] Matos, M. (2003). Scanner 3D para aplicações em modelação e navegação. Master's thesis, University of Aveiro.

[Morales, nd] Morales, M. (n.d.). Algorithmic techniques for probabilistic motion planning. Available from: https://parasol.tamu.edu/~marcom/. Accessed June 2012.

[Nasa, nd] Nasa (n.d.). The clothoid. Available from: http://www-spof.gsfc.nasa.gov/stargaze/Sclothoid.htm. Accessed June 2012.

[NavLab, 2012] NavLab (2012). Navlab 5, no hands across america. Available from: http://www.cs.cmu.edu/image_archive/07robot_hof/navlab5/navlab5_2.jpg. Accessed January 2012.

[Nevada, 2011] Nevada, S. (2011). Nevada legislature - ab511. Available from: http://www.leg.state.nv.us/Session/76th2011/reports/history.cfm?ID=1011. Accessed February 2012.

[Oliveira et al., 2012] Oliveira, M., Santos, V., and Sappa, A. D. (2012). Short term path planning using a multiple hypothesis evaluation approach for an autonomous driving competition. In *Submitted to evaluation*.

[O'Toole, 2009] O'Toole, R. (2009). *Gridlock: why we're stuck in traffic and what to do about it*. Cato Institute Press.

[Paromtchik, 2012] Paromtchik, I. (2012). Inria - driver assistance. Available from: http://emotion.inrialpes.fr/~paromt/index_drv.html. Accessed March 2012.

[Rohling, 2008] Rohling, H. (2008). Smart fm / cw radar systems for automotive applications. In *Radar Conference, 2008. RADAR '08. IEEE*.

[ROS, 2012] ROS (2012). Ros - robot operating system. Available from: http://ros.org. Accessed April 2012.

[Santos et al., 2010] Santos, V., Almeida, J., Ávila, E., Gameiro, D., Oliveira, M., Pascoal, R., Sabino, R., and Stein, P. (2010). Atlascar - technologies for a computer assisted driving system onboard a common automobile. In *Intelligent Transportation Systems (ITSC), 2010 13th International IEEE Conference on*, pages 1421 –1427.

[Schmidhuber, 2011] Schmidhuber, J. (2011). Prof. schmidhuber's highlights of robot car history. Available from: http://www.idsia.ch/~juergen/robotcars.html. Accessed February 2012.

[Schneider, 2012] Schneider, F. (2012). Elrob is a trial! Available from: http://www.elrob.org/. Accessed January 2012.

[Schwarz, 2010] Schwarz, B. (2010). LIDAR: Mapping the world in 3D. *Nature Photonics*, 12:429–430.

[Shapiro and Stockman, 2000] Shapiro, L. and Stockman, G. (2000). *Computer Vision*. Prentice Hall.

[Sick, 2012] Sick (2012). Sick - laser intelligence. Available from: http://www.sick.com/. Accessed March 2012.

[Tartan, 2012] Tartan (2012). Tartan racing car. Available from: http://www.cmu.edu/news/image-archive/Boss400x300.jpg. Accessed February 2012.

[VaMoRs, 2012] VaMoRs (2012). Image of the vamors. Available from: `http://www.unibw.de/lrt8/institut/mitarbeiter/prof_wuensche/vamors/pic1_preview`. Accessed February 2012.

[Vislab, 2009] Vislab (2009). The argo project car. Available from: `http://vislab.it/img/prototypes/argo.jpg`. Accessed February 2012.