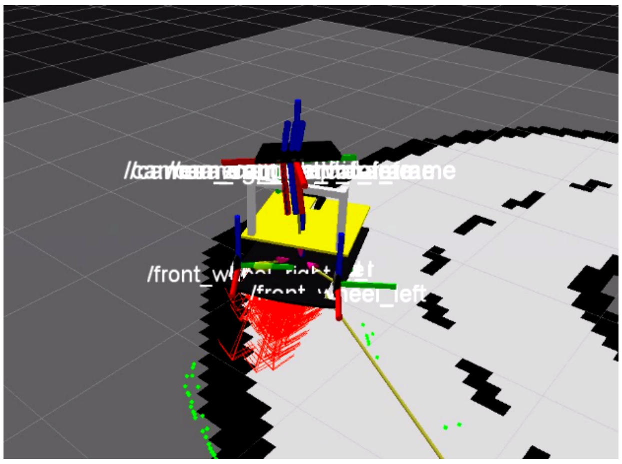
Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática
2013

Leandro Ferreira

Localização e Navegação de um Veículo de Condução Autónoma

Localization and Navigation in an Autonomous Vehicle



The greatest discovery is yet to come.

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática
2013

**Leandro
Ferreira**

Localização e Navegação de um Veículo de Condução Autónoma

Localization and Navigation in an Autonomous Vehicle

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações. Dissertação desenvolvida sob a supervisão cientifica de Artur José Carneiro Pereira e José Luís Costa Pinto de Azevedo, Professores do Departamento de Electrónica, Telecomunicações e Infomática da Universidade de Aveiro e com a colaboração de João Cunha e Eurico Pedrosa, alunos de Douturamento e investigadores no Instituto de Engenharia Electrónica e Telemática de Aveiro.

o júri / the jury

presidente / president	Professor Doutor Armando José Formoso de Pinho professor associado com agregação da Universidade de Aveiro
vogais / examiners comittee	Doutor Agostinho Gil Teixeira Lopes investigador auxiliar da Universidade do Minho
	Professor Doutor Artur José Carneiro Pereira professor auxiliar da Universidade de Aveiro
	Professor Doutor José Luis Costa Pinto de Azevedo professor auxiliar da Universidade de Aveiro

Agradecimentos

A realização deste trabalho não seria possível no tempo previsto sem a colaboração de um conjunto de pessoas e colegas a quem quero deixar um especial agradecimento. Quero agradecer a colaboração do Orientador Professor Artur Pereira e colaboradores Eurico Pedrosa e João Cunha, que foi crucial no desenvolvimento de todo um conjunto de software incluído neste projeto, bem como o apoio do Professor António Neves que me orientou e ajudou no desenvolvimento de algoritmos de visão usados neste projecto. Também a colaboração do Co-orientador Professor José Azevedo, Professor Bernardo Cunha e o auxiliar técnico do departamento José Simões foram essenciais no desenvolvimento do hardware subjacente à implementação do robô e das alterações efetuadas a este. Também não posso deixar de agradecer a todas as pessoas que partilharam comigo o laboratório de robótica, para além dos demais já mencionados, João Miranda e Luís Santos, com quem troquei valiosas experiências na área da robótica e com quem discuti ideias para a implementação deste projecto. Agradeço também a todos aqueles que me ajudaram na correção e escrita deste documento.

Gostaria também de agradecer em especial à familia, aos amigos e em especial à Ana Ferreira por todo o apoio incondicional que mostraram para comigo.

Resumo

A área da condução autónoma tem sido palco de grandes desenvolvimentos nos últimos anos. Não só se tem visto um grande impulso na investigação, existindo já um número considerável de carros autónomos, mas também no mercado, com vários sistemas de condução assistida a equipar veículos comercializados.

No trabalho realizado no âmbito desta dissertação, foram abordados e implementados vários tópicos relevantes para condução autónoma. Nomeadamente, foram implementados sistemas de mapeamento, localização e navegação num veículo autónomo dotado de um sistema de locomoção Ackerman. O veículo é capaz de construir o mapa da pista e de usar esse mapa para navegar.

O mecanismo de mapeamento é supervisionado, no sentido em que o veículo tem de ser remotamente controlado de modo a cobrir a totalidade da pista. A localização do veículo na pista é realizado usando um filtro de partículas, usando um modelo de movimento adequado ao seu tipo de locomoção. O planeamento de percurso faz-se a dois níveis. A um nível mais alto, definem-se pontos de passagem na pista que estabelecem o percurso geral a realizar pelo veículo. A definição destes pontos está diretamente relacionada com a concretização de tarefas impostas ao veículo. A um nível mais baixo, o percurso entre pontos adjacentes anteriores é detalhado numa sequência mais fina de pontos de passagem que tem em consideração as limitações do modelo Ackerman da locomoção do veículo. A navegação é adaptativa, no sentido em que se adequa à existência de obstáculos, entretanto detetados pelo sistema sensorial do veículo.

O sistema sensorial do veículo é essencialmente baseado num dispositivo com visão RGB-D (Kinect) montado num suporte com dois graus de liberdade (pan&tilt). Este sistema é usado concorrentemente para ver a estrada e os obstáculos que nela possam existir e para detetar e identificar sinais de trânsito que aparecem na pista. A aquisição e processamento dos dados sensoriais e a sua transformação em informação (localização do veículo na pista, deteção e localização de obstáculos, deteção e identificação dos sinais de trânsito) foi trabalho realizado pelo autor. Um agente de software foi desenvolvido para gerir o acesso concorrente ao dispositivo de visão.

O veículo desenvolvido participou na Competição de Condução Autónoma, do Festival Nacional de Robótica, edição de 2013, tendo alcançado o primeiro lugar.

Abstract

The autonomous driving field has been a stage of major developments in the last years. Not only has been seen a major push in the research, already existing several self driving cars, but also in the market, with several assisted driving systems equipped in commercialized vehicles.

In the work developed in the scope of this dissertation, it were approached and developed several relevant topics to the autonomous driving problem. Namely, it were implemented mapping systems, localization and navigation in an autonomous vehicle with an Ackerman locomotion system. The vehicle is capable of building the map of the track and use that map to navigate.

The mapping mechanism is supervised, the vehicle has to be remotely controlled to cover the entire track. The localization of the vehicle in the track is accomplished using a particle filter, using the adequate motion model to its locomotion system. The path planning is performed at two levels. At a higher level, the overall course to be performed by the vehicle is defined by passage points. At a lower level, the path between the aforementioned points is detailed in a thiner sequence of points that take into account the limitations of the Ackerman motion model. The navigation is adaptive since it adapts to the existence of the obstacles detected by the robot's sensory system.

The vehicle's sensory system is essentially based on a device with RGB-D vision system (Kinect) mounted over a structure with two degrees of freedom (pan&tilt). This system is concurrently used to see the track and the obstacles that may exist and to detect and identify traffic signs that appear on the track. The acquisition and processing of the sensory data and its transformation in information (localization of the vehicle in the track, detection and localization of obstacles, detection and identification of traffic signs) was work developed by the author. A software agent was developed to manage the concurrent access to the vision device.

The developed vehicle participated in the Autonomous Driving Competition, from the Portuguese Robotics Open, 2013 edition, having achieved the first place.

Contents

List of Figures

Acronyms

6D	6 DoF.
API	Application Programming Interface.
ATRI	Transverse Activity on Intelligent Robotics.
CAN	Controller Area Network.
EKF	Extended Kalman Filter.
GPS	Global Positioning System.
KLD-Sampling	Kullback-Leibler Divergence Sampling.
LED	Light-emitting Diode.
LRF	Laser Range Finder.
MCL	Monte Carlo Localization.
MHT	Multi-Hypothesis Tracking.
MOM	Message-oriented Middleware.
PID	Proportional Integral Derivative.
RANSAC	RANdom SAmple Consensus.
RGB	Red Green Blue.
RGBDSLAM	6DoF SLAM for Kinect-style cameras.
ROS	Robotic Operating System.
SBP	Search-Based Planning.
SBPL	Search Based Planner Library.

SLAM	Simultaneous Localization and Mapping.
UKF	Unscented Kalman Filter.
URDF	Unified Robot Description Format.
USB	Universal Serial Bus.
XML	eXtensible Markup Language.

Chapter 1

Introduction

From this chapter is possible to understand how the proposal for this dissertation was born, the objectives and goals to fulfil. The goals are defined by the dissertation theme itself and also by the Autonomous Driving Challenge in which the robot developed should participate. It this chapter, is also possible to have an overview of this project and how this document is organized.

1.1 Motivation and Context

The project in which this dissertation fits is ROTA, which is a project of the Transverse Activity on Intelligent Robotics (ATRI) [1]. The objective of the project is to develop an autonomous driving robot capable of driving in a known and controlled environment. The environment in which this robot is aimed to drive is described in the rules of the Autonomous Driving Challenge from the Portuguese Robotics Open. The robots for this competition have to be autonomous, capable of driving through a known track that is composed of two lanes. The track is a closed loop that has a crosswalk in its course, as it is shown in Figure 1.1. As a part of the competition there are several obstacles that the robot must overcome.

In this competition the robot should complete a pre defined number of laps in track considering the signs that are placed over the crosswalk and vertical signs in unknown positions. The robot has also to overcome obstacles obstructing one of the lanes, a roadwork course and also a tunnel where the visibility is reduced.

The competition includes traffic light signs recognition, where the traffic light is a TFT screen with signs to be identified, placed over the crosswalk. The signs in the TFT that the robot must recognize are the ahead sign, for going ahead in the track, the left sign, to turn left after the crosswalk, the right

[1] wiki.ieeta.pt/wiki/index.php/Transverse_Activity_on_Intelligent_Robotics

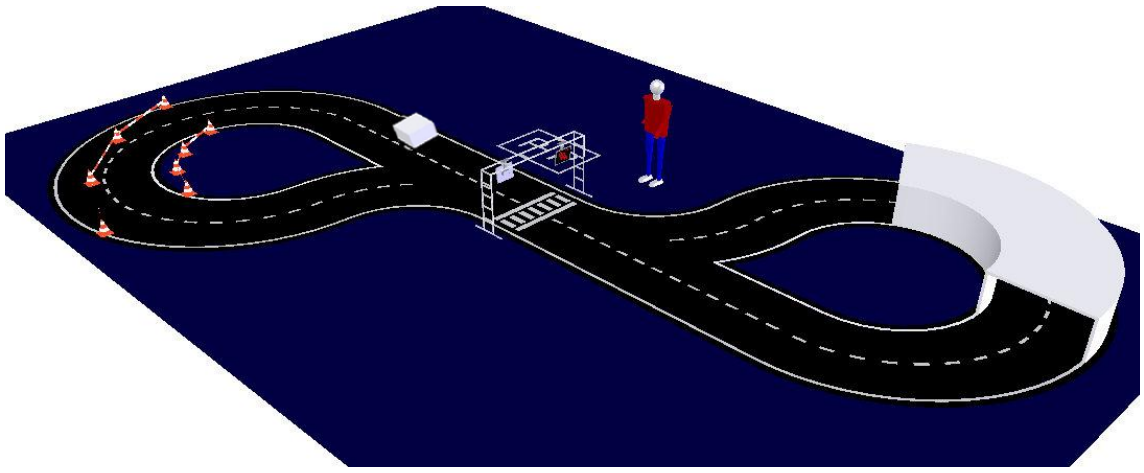


Figure 1.1: Overview of the Portuguese Robotics Open, Autonomous Driving Challenge

sign, to park the robot in the parking area, the chess sign, indicating to the robot that the is entering the last lap, and the stop sign, indicating that the robot must stop by the crosswalk and wait for other sign. Also the competition has vertical signs placed in unknown positions in the map witch the robot should be able to detect and identify.

As it occurs in a real driving scenario, a lane of the track may be obstructed and the robot should be able to detect and localize the obstacle in order to overcome it. In the competition the robot may encounter two kinds of obstacles, a green box obstructing a lane, and the orange and white cones, which delimit a roadworks zone, where the robot may have to leave the track to follow a detour delimited by those cones.

Having described the main elements of the competition in which the robot should be able to participate, the motivation is clear. In the described environment above, the most efficient way to fulfil all requirements, is to have the robot localized and intelligent enough to plan a trajectory. The described requirements for the robot require localization, which in turn uses mapping and navigation, which includes motion planning.

1.2 Objectives

The primary objective of this dissertation is to make the existing robot of the ROTA project able of localize itself on the track that it is driving and use that information to navigate, using as primary sensor the Kinect camera. It is also an objective to use Robotic Operating System (ROS)[2] as the basis for the robot software. In Appendix A is given an overview of ROS, and please refer to it for details on the system and how is organized a system developed over ROS.

[2] www.ros.org

The previously defined objectives have branched into other objectives necessary to achieve the main goals. Since the Kinect has not been used before in the ROTA project, to develop a completely new vision system based on it became also an objective of this work. Since the ROS has most of the software developed for holonomic robots, it became an objective as well to study and adapt the use of ROS to robots with Ackerman locomotion system.

The navigation and localization objectives demand the study of other subjects. Besides the vision system as adicional objective, there is the mapping problem. For the purpose of the competition the track is known, being described on the rules, and in that case it is possible to build a map for localization purposes. However, the objective it is not to build a navigation system only valid for a specif track, but instead, for a generic one. In that case the map should be assumed to be unknown and the robot should build its own map, and in that case the navigation becomes generic regardless of the track map. This introduces another objective to this dissertation, make the robot capable of performing Simultaneous Localization and Mapping (SLAM) so that the robot can build its own map. In sum, there are several objectives for this dissertation that can be defined:

- Develop the vision system for ROTA, based on the Kinect;
- Study how to use the mapping, namely SLAM algorithms available in ROS, in this project;
- Implement, test and validate the mapping;
- Study how to use the localization algorithms available in ROS, in this dissertation study case;
- Implement, test and validate the localization algorithm;
- Study how to use the navigation algorithms available in ROS;
- Implement, test and validate the navigation, which includes motion planning;
- Implement other necessary software for the competition objectives such as traffic light signs recognition, obstacle recognition and avoidance, vertical sign detection and identification, etc.;
- Implement the main navigation robotic agent.

1.3 Structure of the Document

This document is dived into six chapters including the present one. The next chapter, Chapter 2, is meant to situate the reader in the main goals of this dissertation project. The mapping, localization and navigation problems are addressed and explained. The state of the art of each problem is also presented.

The robotic platform used in this work is also addressed and its structure is explained in this document in Chapter 3. This chapter is meant to briefly describe the robotic platform used and its constraints and implementation options. The robotic platform was already available in the ROTA project and was adapted to accommodate the new features and implementation options. In Chapter 3 this changes are addressed and explained.

As for the software implemented, is described in Chapters 4 and 5. Chapter 4 describes the software architecture as well as the options made to implement it. It is also given an overview of the algorithms used in each software module implemented and the design options, as well as the justifications to support them. Chapter 5 shows the software implementation details, over ROS, of the algorithms and design options stated in Chapter 4. In Chapter 5 are also shown the results of such implementation options and algorithms.

The last chapter, Chapter 6, contains the conclusions taken of the resulting work developed in this dissertation project. In this chapter are also stated some ideas for future work in this project based on the obtained results. The ideas are also based on the experience acquired with the realization of this project.

Chapter 2

Mapping, Localization, Navigation

This chapter describes what is the state of the art of the main objectives of this dissertation. Essentially, since ROS is used as basis, this chapter shows the state of the art of the subjects necessary to accomplish the primary objectives of this dissertation, and how they are represented in ROS. Since ROS, in most cases, uses open source algorithms with a ROS wrapper, the algorithms used by ROS tend to be the ones that represent the state of the art at the moment.

2.1 Mapping

Mapping, in this case, refers to map-making. The map-making implies the spacial data collection in order to build a map. The problem in building a map with collected data is how to store that data in order to represent a useful map in terms of occupied and unoccupied space and in terms of accuracy thus usefully representing the environment. The mapping problem, in mobile robotics, is often referred to as the SLAM problem. For a mobile robot to build a map it needs a good estimate of its location and, to have that, a robot needs a consistent map. It is not possible to separate the mapping problem from the localization problem[39]. The described mutual dependency between map and robot pose makes the SLAM problem a complex and at the same time, an interesting problem and the difficulty arises due to localization errors that are incorporated in the map. The SLAM problem consists in merging two main sources of information of a mobile robot in order to obtain a representation of the environment that is consistent enough to allow a mobile robot to operate autonomously. Those two main sources of information are the idiothetic, that refers to motion, as dead reckoning that gives the absolute position of the robot using the accumulation of the odometry sensors, and allothetic, that refers to the observation sensors that is information from the surrounding environment. In Figure 2.1 is show an example of a robot performing SLAM, building a 2D map of the environment while navigating.

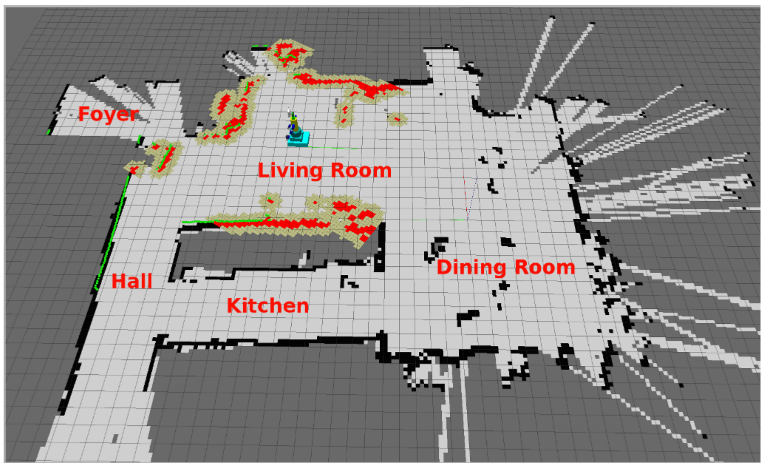


Figure 2.1: Example of a Robot Performing 2D SLAM[18]

For any mobile robot, to know the surrounding environment is very important. It is necessary to know the surrounding environment to implement any kind of deliberative navigation and path planning, because navigation is only reliable with localization, and localization can only be reliable with a reliable map of the environment. So, SLAM is not just only an interesting problem but useful as well. If the robot builds its own map, when it is reliable, the localization will be reliable as well, since the robot will see the world just as it is on the map. Also, it is very useful to have a robot building its own map since it spares the user from exploring and analyzing the environment to build it. This operation can be very difficult in environments with a lot of details and even more if those environments change in short periods of time, obligating to build or change the map for every application. Also, SLAM can have numerous applications like house helping robots that would require to be personalized to every home, which is not necessary recurring to SLAM since the robot may learn the map of the house. Another direct application of SLAM may be the exploration of dangerous or inaccessible places for humans.

2.1.1 Problems and Different Approaches

One of the problems in mapping, in the field of robotics, is how to store and represent the spatial information of an environment in order to make this information perceptible and useful for navigation and localization that are the main uses for the map. The representation problem is an ambiguous one and there are different approaches for this problem which depend on the application.

In terms of spatial organization of the data there are solutions from two dimensions (x and y) to six dimensions (x, y, z and Red Green Blue (RGB) for example). In Figure 2.2 is show a map resulting from a six dimensional SLAM algorithm. The representation of the data is a problem as well. The information should be concise, due to processing and storing problems, but should be enough to

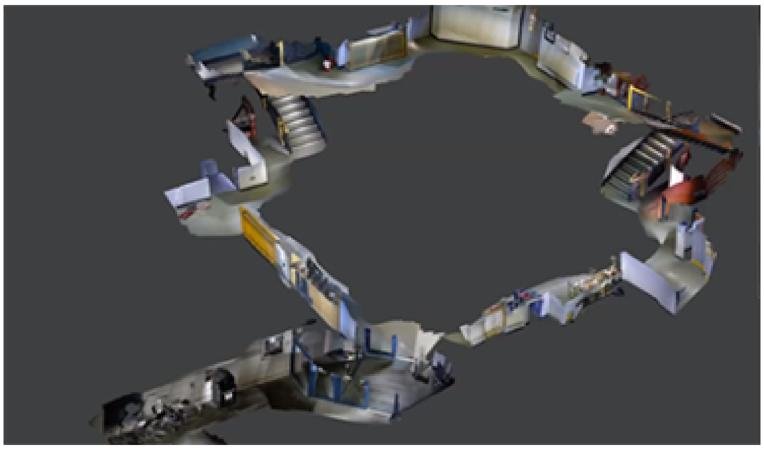


Figure 2.2: Example of a 3D Map[2]

represent the world to a certain level of detail in order to decrease the level of uncertainty of the localization process. The representation of data is always concerning the occupied or unoccupied space. In the case of six dimensions representations, besides the occupation, the color is also represented. The representation of the color improves the localization, since the color disambiguates spaces that geometrically look alike. It is very difficult to have a complete deterministic localization because the sensors have errors. When the map is built by the robot itself this errors become more significant since the map is built with the errors from the sensors, and the measurements for localization are done with errors as well. In a map where there are two similar areas, it should be possible to represent that the robot may be at one of those areas, not deterministically, representing both possibilities with a probability. Because of this problem it does not make sense to build a deterministic map and have a deterministic localization, map is probabilistic and the localization is also probabilistic. The probabilistic map is built considering the error from the sensors, the error from the observation sensors and from odometry sensors. To accommodate this problem, the occupancy is not represented only by two values, occupied or unoccupied, but by predefined values that allow to represent the probability of a certain space to be occupied or unoccupied and even situations of unknown space. The representation of the occupancy state has many solutions, where the quality and performance depends on the application and, as it would be expected, it is very different to represent a 3D map rather than a 2D map.

In the case of SLAM the map is built with collected data by the observation sensors and this data is arranged in the map according to the displacement of the robot measured by the odometry system, in order to have a representation of the environment so that a mobile robot can use that information to navigate autonomously. Usually, in robotics, neither the observation sensors nor the odometry sensors acquire information continuously, which means that the data collection is done at a given sample rate which means that the collected data from the environment is discrete. In robotics, typically, everything is processed in the digital domain, due to sample rates imposed by the sensors, due to all the processing is done by a computer, and also, due to simplicity. For the aforementioned reasons, the map is built and processed in the digital domain, and so, the map is discrete. There are some different approaches for representing the collected data into a discrete and processable map.

One of the most common representations for maps is the topological representation. This representation, uses a graph to represent the environment. It represents intersections with nodes and these are attached by lines that represent a path between the nodes. The path connecting the nodes may or not may represent the path correctly, for example, it may not contain the curves in the path. It is common to see these types of maps for city or urbanization representations since it is easy to read and occupies less space than other types of representations. This kind of representation for maps may also be used in robotics by identifying each node with specific features however, if the links between nodes do not represent the real path between them, as it often occurs, the navigation in these has to be done using other mechanisms besides localization. The topological representation also presents the advantage on being extremely simple for performing path planning and occupies less space in memory compared to other types of representation.

Another commonly seen representation of environment are metric maps, where every detail of the environment is represented as it is physically. These maps are very commonly used for cities representation or road mapping in cases that the environment needs to be accurately represented. This type of representation is used, for instance, in Global Positioning System (GPS).

An environment representation commonly used in robotics is the landmark list. The land mark list may be represented as an image with a pre defined resolution for converting pixel values to metric values. This list may also be a simple text file where each landmark is identified and located. This maps are very memory efficient and make it simple to implement a localization system depending on the quantity and singularity of the landmarks. Of course this map has limited range of applications, since most of the times it is not easy to describe environments with few landmarks that have to be distinguishable among each other.

There is also the forcefield map. This type of map represents the environment through forcefields, where the forcefields may point towards or outwards obstacles. This map not only indicates the obstacles position, but also how far they are. Using this type of world one can implement path planning for avoiding the obstacles as far as possible just by looking for the lowest resulting forcefield in the map. This type of representation is also useful for reactive implementations since the robot reacts according to the force field to avoid obstacles, which is why this type of representation is often associated to other map. The force field map is used for deciding a reaction, and there is another map used for localization.

From the various solutions available for the representation of the environment, the far most used is the metric map in a grid form. The grid divides the map into cells, and these cells are filled with the information that represent the environment in that area and depending on the application, this information varies [39]. For example, in the case of a 6D SLAM, like RGB SLAM, the information in the cell represents a color and the cell is a 3D cell. However, in the case of 2D SLAM, the most common is to have the cell filled with a value that represents occupancy where the cell is 2D. One of the advantages of the grid cell representation is the possibility of changing the resolution, and so, the cell may be smaller in order to have a finer map, having more accuracy, or the cell may be larger in a more uniform map thus reducing process times and memory usage. In the case of the 2D SLAM each map cell represents the occupancy of that area with a single value, where this value represents a probability of being occupied, considering the errors from the sensors, or the density of the occupation in that cell.

Due to the intense research on the SLAM problem on last few years there are different SLAM methods and algorithms for different kinds of applications. Most of the SLAM algorithms that exist are available on the OpenSLAM [1] website, which is a platform created so that researchers of the SLAM problem can publish their algorithms, making them available to the world and so that those algorithms may be used and tested by other researchers.

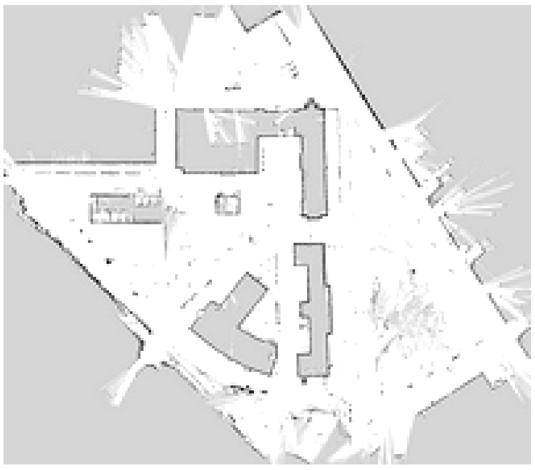
The great majority of SLAM algorithms build a grid map due to their vast applicability and flexibility and mainly because they are are 2D maps. However, there are algorithms for 3D grid maps and there are also SLAM algorithms to build other kinds of maps. An example of a 2D algorithm that is not available on OpenSLAM, however it is in ROS, is the hector SLAM algorithm[24]. This algorithm consumes very few computational resources and it is quite efficient when a good pose estimate is available. This algorithm is very useful for performing SLAM in unstructured environments since it uses robust scan matching allied to the sensor 3D pose estimation, using to measure it, an inertial system.

For building 3D grid maps there are algorithms like the SLAM with 6 DoF (6D) [2], which is basically a software to register 3D point clouds into a common coordinate system and a GUI interface to display the scene and change the algorithms parameters [28]. Another example of a SLAM algorithm for 3D grid map construction is the 6DoF SLAM for Kinect-style cameras (RGBDSLAM) [3][12]. The RGBDSLAM algorithm is aimed for quickly acquire maps or models from indoor scenes and objects with a hand-held Kinect-style camera [11] and thus it is not required to have an odometry system nor even a robot, but only a Kinect-style sensor.

2.1.2 State of the Art

The state of the art at the moment, in what concerns mapping, is the GMapping[20][21] [4] algorithm. In Figure 2.3 are shown two maps obtained using the GMapping algorithm. The GMapping algorithm is available on the OpenSLAM website and is integrated over ROS in the *slam_gmapping* [5] package. The GMapping algorithm builds grid maps from laser range data and

[1]www.openslam.org
[2]www.openslam.org/slam6d
[3]www.openslam.org/rgbdslam
[4]www.openslam.org/gmapping
[5]www.ros.org/wiki/gmapping


(a) Map of the Freiburg Campus

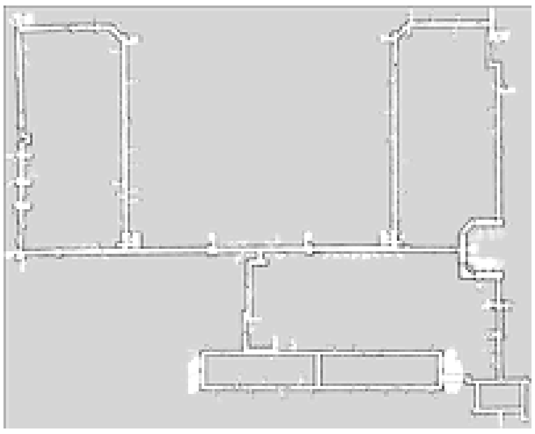

(b) Map of the MIT Killian Court

Figure 2.3: Maps Obtained using GMapping on two Famous Datasets

is a highly efficient Rao-Blackwellized particle filter [19].

Common to all SLAM algorithms, there is the localization problem, the mapping problem and the mutual dependency problem to be solved. Every SLAM algorithm has its approach to solve these problems. As for GMapping, it uses a Rao-Blackwellized particle filter in which each particle of this has its own and individual map of the environment [20] dealing thus with the mutual dependency problem between mapping and localization. The individual map of the environment that each particle carries is built using scan matching, were the scan match is computed according to the motion of each particle. The problem of having a particle filter in a SLAM algorithm, is how to select the correct particle, which is, the particle that has the map that represents the environment more accurately and that has the correct location on that map. Selecting the correct particle, or at least a particle close to the correct one, forces to deal with two of the main problems of the particle filters:

- Reduction of the number of particles;
- Particle depletion.

In a particle filter there could be hundreds of particles, and in this case, each one carrying its own map, which means that there are hundreds of maps, and it is necessary to choose one. For that matter, the reduction of the number of particles is very important and thus by reducing the number of particles it is possible to choose a particle that best fits the environment. To reduce the number of particles in the Rao-Blackwellized particle filter, GMapping computes not only the odometry data, but also the last observation of the environment using a scan matching procedure and so the last observation is taken into account for the next particle selection, which improves the efficiency of the particle filter[20].

The reduction and resampling of the particles leads to another problem, the particle depletion. The particle depletion problem is a critical one, because these problem may cause the particle filter to fail completely. When the particles are selected in the resampling step of the filter, it may happen that the particle that best represents the robot does not got sampled, or even that all the particles sampled converge to only one narrow blob, and this blob does not represent the state of the robot. In the case of such thing happens, the particle filter would not have any particle to represent the robot, causing the filter failure, being this problem known as particle depletion. So, the particle depletion problem is described as when in the all set of the particles that still live in the particle filter, none of them represent the robot. To cope with the particle depletion problem, the GMapping algorithm uses an improved proposal distribution and an adaptive resampling technique[21], that allows to have a variety of particles reducing the risk of particle depletion.

Almost all the described mapping methods, and even the state of the art algorithms use laser scan data from Laser Range Finder (LRF) as input. However there are algorithms to perform SLAM with a Kinect, such as the SLAM with 6D or the RGBDSLAM as described in by [12] and [28]. Furthermore, there are methods to transform point cloud data into laser scan data, making it possible to use most of SLAM algorithms with a Kinect instead of the usual LRF. Since the algorithms are optimized for large distances in the collected data, the quality of the results using fake laser scan data generated by the Kinect depends on the algorithm (the Kinect has a short range when compared with most of the LRF).

2.2 Localization

The localization problem, applied to mobile robotics, has been a strong field of research. In order to have a deliberative mobile robot, it is necessary to have perception of the surrounding environment. Most of the times, the perception of the robot results in a localization, allowing to have a robot with a deliberative navigation, which is almost a requirement in mobile robots with complex tasks. For example, for a vacuum cleaner robot, localization is not a requirement, with reactive navigation for deviating obstacle the robot may perform its task. However, considering a service robot, it is almost impossible to complete most of the tasks without localization.

2.2.1 Problems and Different Approaches

The localization problem and its implementation is highly dependent on the application and the problems that this has to solve. The implementation of a localization algorithm depends on many things, the initial knowledge of the environment, the type of the environment, if it is required a global or local localization among others. These problems are divided into dimensions that define the problems of localization [39].

One of these dimensions is defining the problem as a global localization or a local localization problem. In a local localization approach, the uncertainty of the localization is local, confined to a region near the robot's true pose, and often approximated to an unimodal distribution. In local localization, usually the initial pose is known and what is performed is position tracking accommodating the noise in robot motion. As for the global localization approach, there can not be any boundaries for the uncertainty of the localization since the initial localization is not assumed. The global localization problem is then clearly of more difficult implementation than local localization once it is inappropriate to use unimodal probability distributions for a problem like this.

The implementation of localization is also highly dependent on the environment in which the robot has to localize. It is clear to imagine that the localization problem becomes difficult in a dynamic environment. For localization, the robot uses a map of the environment in which is navigating, and of course, if the environment is dynamic, the perception of the world will not correspond to the map that the robot has. On the other hand, if the environment is static, the perception of the world will match with the description and map that the robot has, making easier to implement comparison between perception and map, making thus more probable the localization to succeed.

The localization problem is not confined only to observations of the world in order to localize in a passive manner. It is possible to have a localization implementation which is active and has control of the robot in order to place the robot in a position that favours localization. If the localization of the robot is ambiguous and there is a good landmark nearby that may reduce the uncertainty of the localization, active approaches on localization will drive the robot to that landmark. However, this implies that the localization has control over the robot, which is much of the times not compatible with other tasks that the robot may use. Most of the times active localization approaches yield better results, but being these incompatible with other tasks, most of the times active methods are implemented on top of passive approaches. This is the case of the *coastal navigation*[39] represented in Figure 2.4. It is possible to see that the route performed to the goal is not the shortest, however, this route has the advantage of making the robot travel closer to the obstacles, thus enhancing the localization.

There is also another possibility for localization implementations, the use of a single robot or multiple robots. The use of a single robot is the most common studied case, and even in a scenario with multiple robots cooperating, each can perform single robot localization with no cooperation for localization. In a multi robot scenario there is the possibility of combining the data of the robots for localization, or use the relative position of each robot to combine a localization from all the robots seeing each other. However, in a multi robot localization system, there are other problems to solve like the communication between robots and the computation of the different beliefs of the robots.

Most of the localization approaches rely on world perception data and motion data as well, but there are algorithms for solving the localization that rely almost entirely on the world perception

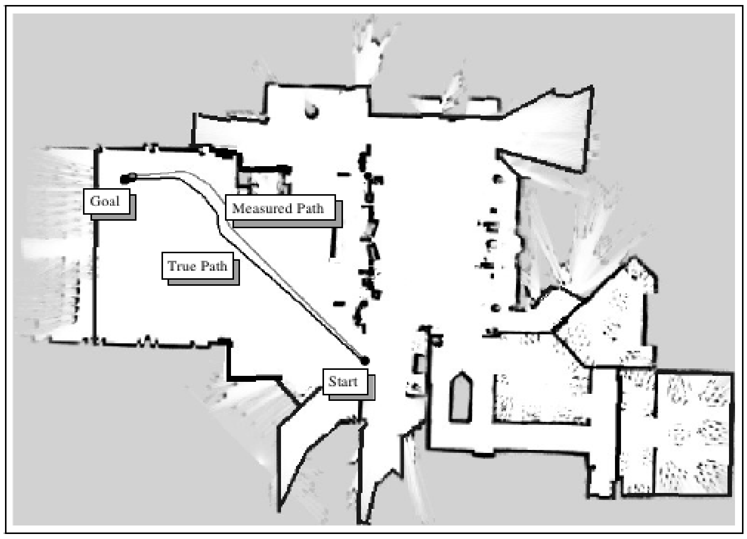


Figure 2.4: Coastal Navigation Route[39]

[27]. The localization approaches that rely of both odometry and perception data need to merge this information into a resulting pose. Also, the two used data sources are very different with very different results, models and errors. The resulting pose using such data has to be probabilistic and represented by a belief. The described process, for acquiring a probabilistic pose from odometry and perception data combined, is in all similar to the Bayes filter, making the probabilistic localization algorithms variants of it[39]. In Figure 2.5 it is shown the Bayes network characterizing the aforementioned

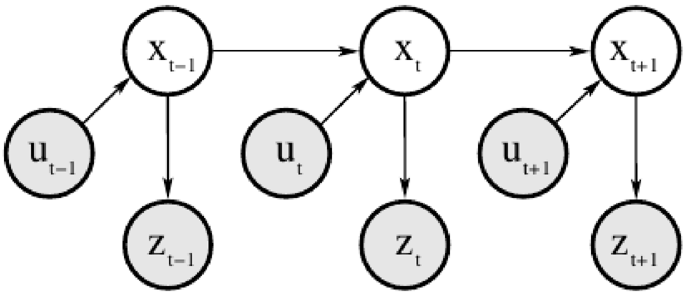


Figure 2.5: Evolution of Controls, States, and Measurements[39]

evolution of movement and measures that is necessary to combine into a result and a belief for that result, u_t are the motion commands, x_t the result and the z_t a measurement. Applying this network to the localization problem, the u_t are motion commands that should move the robot to the pose x_t, and the fact that the robot is in the pose x_t should originate the measurement z_t.

The computation of a belief for a result position, considering the probability of the action commands applied causing the movement to a given position, and considering also that the probability of that position originates a given measurement, is the application of the Bayes filter to localization. The belief is represented as a function of position, and each probable position of the robot is represented with a Gaussian which mean and variance indicate the probability of that location representing the robot true pose. Such application is represented in Figure 2.6 where the m represents the map that is used for the measurement model.

Most of the localization algorithms are based on the scheme represented in Figure 2.6 and the

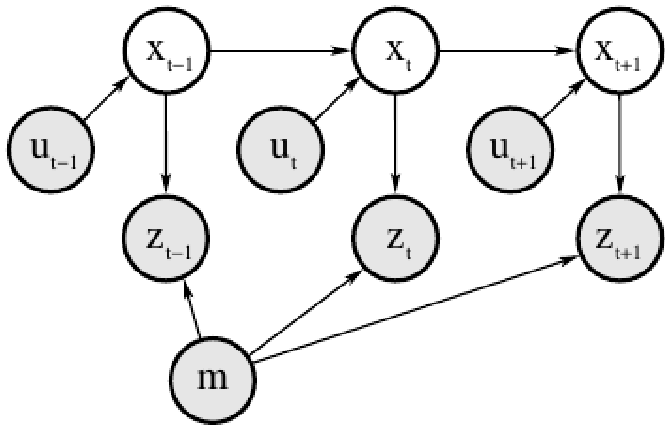


Figure 2.6: Bayes Filter Applied to Localization[39]

obvious utilization of the Bayes filter into localization is the Markov localization algorithm and it functions just as the previously description of the Bayes filter applied to localization. The Markov localization algorithm is a fully functional method for static environments, it addresses the global localization problem, position tracking and kidnapped robot problems[39].

Another variation of algorithms based on the Bayes filter, is the Extended Kalman Filter (EKF) localization[39], which is a special case of Markov localization. Just as Markov localization algorithm this addresses most of the localization problems. The two algorithms are very similar, the particularity of the EKF is that this assumes to have a feature based map, and that all landmarks are uniquely identifiable. Just like Markov localization, the EKF computes the resulting pose with a belief that is represented by a Gaussian. The difference is that EKF represents the belief with a single Gaussian since it considers uniquely identifiable landmarks.

To overcome the default of EKF where there is only a Gaussian, there are extensions to the algorithm. As explained before, the reason why the produced result using the EKF is a single Gaussian is due to the data association problem, since the landmarks are considered uniquely identifiable, and most of the times this is not the case. One of this extensions, and a solution for the data association problem, is the Multi-Hypothesis Tracking (MHT). The MHT is derived from a full Bayesian implementation of EKF under unknown data association[39], which allows to describe the belief by multiple Gaussians. As the MHT, there are other extensions and implementations of the Kalman Filter applied to localization like the Unscented Kalman Filter (UKF) which use the unscented transform to linearize the motion and measurement models.

The Bayes filter has also a discrete variation, which is more suitable considering that a robot has to have a discrete model of the world, and the map is discrete as well. Such discrete implementation is called histogram filter[39]. The histogram filter is applied to a discrete world, typically divided into cells, and it functions just has the Bayes filter, however, the histogram represents the belief by attributing a probability to each cell of the world. Considering a two dimensional world, each cell has three dimensions (x, y, θ), being θ the orientation. The probability is not only attributed to a cell but to orientation has well, meaning that to a given cell more than one probability may be associated due to multiple orientations possible.

There is another alternative to the histogram filter for discrete environments, the particle filter. The particle filter is very similar to the histogram filter and it is also based on the Bayes filter, however the particle filter is different in the way it represents its belief and on how the space state is populated[39]. One implementation of a particle filter is explained in the next section, since it has been a very used method for solving the localization problem and it is considered the state of the art for probabilistic localization.

2.2.2 State of the Art

The state of the art algorithm for localization is a probabilistic particle filter which implements the Monte Carlo Localization (MCL) approach with Kullback-Leibler Divergence Sampling (KLD-Sampling)[39]. This algorithm implements a probabilistic localization approach in which the belief is represented by a set of particles. These particles represent possible poses of the robot in 2D (x, y, θ), and they are computed to originate a resulting probabilistic pose. In Figure 2.7 is represented the implementation of the particle filter for a one dimensional localization where the belief $(bel(x))$ is represented by particles, and each particle is represented by a pose and a probability.

Assuming that at first the particles are randomly created, the first question that arises with a particle filter implementation is on how to manage the particles in terms on how to define them, maintain them over the time and how to generate a resulting pose. A particle is defined, besides being a possible pose of the robot, with a probability. This means that each particle represents a pose and a probability of being the particle representing the true pose of the robot.

As for the problem on how to maintain the particles over time, this algorithm uses the KLD-Sampling[39] approach. The problem of the particle maintenance is a struggle between computational effort and accuracy. Since each particle represents a possible pose, on a filter using less particles the uncertainty arises and eventually it may lead to the problem of particle depletion in which none of the particles represent the robots pose. On the other hand, if a particle filter uses too many particles it becomes computationally heavy to process the particles at each processing cycle, lowering the rate of the pose publishing. For that reason, this algorithm uses the KLD-Sampling approach which is a variante of the original MCL algorithm. Basically, the ideia of KLD-Sampling is to adapt the

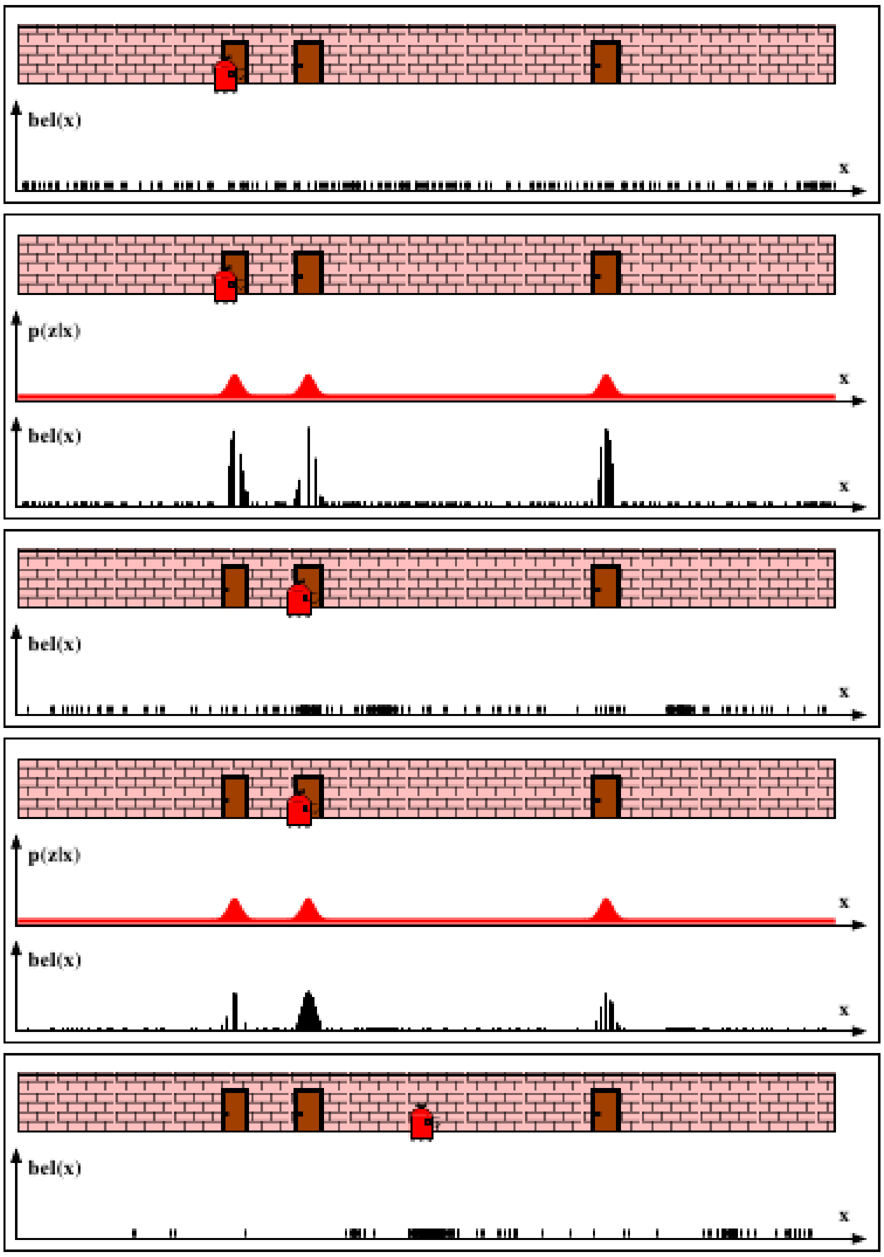


Figure 2.7: Representation of a Particle Filter Implementation[39]

number of particles over time based on belief of the particle filter. This approach allows to have a lower number of particles when the belief is larger, and a higher set of particles when the belief is lower allowing to have more possible representation of the robots pose. The KLD-Sampling approach defines another problem in the particles maintenance. It is possible to the KLD-Sampling approach to eliminate particles that actually represent the robots pose, and even though there are less scattered particles in which results a higher belief, the pose is mistakenly represented.

For the aforementioned reason, the used algorithm also implements an algorithm for recovering from failures, the Augmented MCL [39]. The base ideia of this approach is to add random particles to the particle set, in order to improve the possibility of having a particle representing the robots pose. The process results in a larger set of particles, resulting in a lower belief which in turn will result in a recovery of the particle filter.

As for computing a resulting pose, there are some valid approaches that depend on the application itself. One may calculate the resulting pose by at least three different ways, selecting the best particle, calculating an average of the particles or an average of the best representative blob of particles.

At this point, the only explanation missing is on how to update the particles. Each particle, as it was mentioned, is represented by a pose and by a probability, and it is necessary to update the state of the particles. The main core of the MCL algorithm as described in [39] is summarized as three main steps:

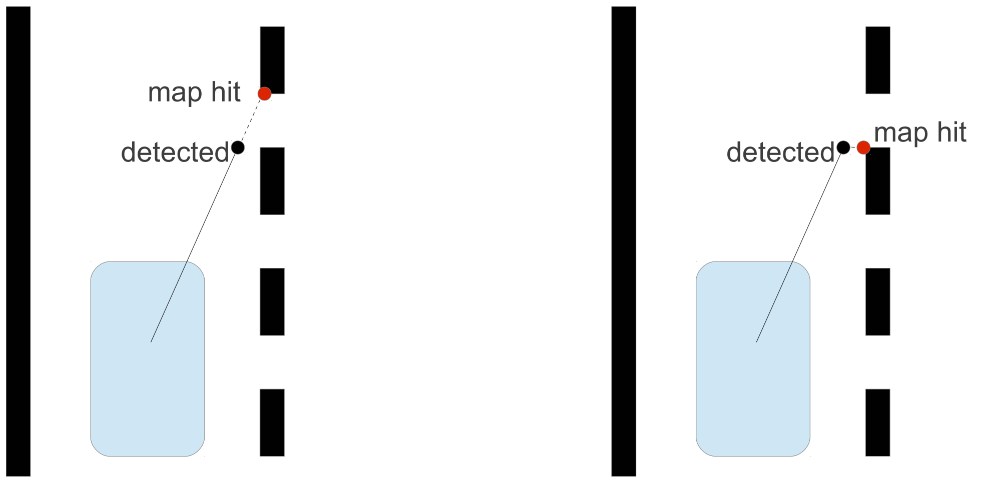
- update the pose of the particles (motion update);
- update the probability of the particle (measurement update);
- resample particles according to their probability.

The motion update, is where the robots movement is added to the particles and there are two different methods, described in [39], for considering errors in motion measurement systems and velocity measurement systems for odometry. The addition of motion to the particles is not performed in a deterministic manner, meaning that it is not the displacement measured by the odometry system that is added to all the particles. The motion added to the particles is the odometry displacement plus random noise that is generated for each particle. The generated random noise has a standard deviation that is defined by the error of the odometry system, and this error is generated according to the locomotion system, considering the covariance between the odometry components ($\Delta x, \Delta y, \Delta\theta$)). For example, in an Ackerman locomotion system, the random error should not be added to the (x, y, θ) components directly, but instead, the error has to be added to the distance covered by the wheels and to the steering angle since these are the components used to calculate the odometry displacement. Only after, the (x, y, θ) is recalculated and applied to the particles. By adding motion to the particles with an error, this error being generated randomly for each particle, the particles randomly scatter increasing the probability of having one particle that has been updated with the real movement.

The second step of the particle filter is the measurement update. In this step the measurement done by the perception sensor is added to the particles. This addition implies to update the probability of the particle comparing the measurement done with the measurement that each particle should have done. The measurement, is compared to the map to find where it fits best, and with this it is possible to have a measurement of which particles fit better with the measurement performed, and depending on that error, the probability of the particle is computed.

Any perception sensor has errors associated with the measurement which are intrinsic to the sensor itself or caused by any filtering or processing of sensor data. These erros have to be accounted for when measuring the error between the measurement and how this is projected on the map for computing the particle probability.

The perception of the world is, most of the times, based on LRF sensors, where each beam represents the measure of a landmark. There are two models, defined in [39], for considering measurement errors in a LRF sensor type, the beam model and the likelihood field model. The way that each model considers sensor errors and associates a map landmark with a measurement is represented in Figure 2.8. The black dot represents the performed measurement by the sensor and the red dot the association of that measurement to the map performed by the sensor model. Figure



(a) Beam Sensor Model (b) Likelihood Field Model

Figure 2.8: Representation on How the Sensor Model Corresponds a Measure to the Map

2.8(a) represents how the beam model matches in the map a real distance obtained by measurement. Basically, the beam which defines the measurement, is extended in the map until a landmark is found. The beam, for considering errors, is not extended in a straight line, but in form of a cone[39], being the opening of this defined by the standard deviation of the measurement error in distance and bearing. As for the likelihood model, which is represented in Figure 2.8(b), the measurement is extended in all directions, creating a field where this measurement may hit a map landmark[39].

The third and last step of the particle filter is the resampling step, which has various ways of being implemented like the KLD-Sampling and the augmented MCL[39]. Also, after the resampling step, the probabilities of the particles are normalized. The basis of the sampling step, however, is the same in every implementation. The resampling step is where the particles get resampled based on its probability, however, even those particles with low probability should be sampled. The particle filter is a probabilistic one, and the sample of the particles can not be deterministic, since even the particles with the lowest priority may be the ones that actually best represents the robots pose.

In the sampling step, the particles are sampled with greater probability if their probability is higher, meaning that the sampling probability is directly proportional to the particles probability, and also means, than even the particles with lower probability have a probability of being sampled to the next step. With this probabilistic resampling approach, the problem of discarding less probable particles, that may represent the robots pose, gets diminished.

2.3 Navigation

The navigation problem is a complex one and it is a recurring problem in mobile robotics. There are many applications for autonomous mobile robots, from planetary exploration[29] to service robots. The field of autonomous driving cars has pushed a major breakthrough in the last few years. There are, for instance, many cars from various different teams competing at the DARPA Grand Challenge [6] which is one of the greatest test-benches for autonomous driving cars. The evolution of the autonomous navigation has been remarkable and there are robots capable of navigating autonomously in unknown environments and even adapting to very different situations with the implementation of machine learning.

The navigation is not only confined to the two dimensional problem, there is also three dimensional navigation and path planning[6]. The three dimensional navigation is often associated to the aerial robotics since these robots navigate into three dimensions. The navigation problem may even be expanded to more than three dimensions like it is the case of arm navigation where six dimensional navigation is required.

2.3.1 Problems and Different Approaches

The autonomous navigation not always implies having a central processing unit with software for the navigation. For example, it is possible to implement a mobile robot just using hardware and simple electronic components depending on the environment. Namely, considering an environment with lines drawn on the floor, it is possible to have a robot following those lines just using infra red sensors and simple integrated circuits for reacting to the position of the lines. These type of implementation for navigation is defined as reactive navigation.

The reactive navigation, as its name implies, is implemented just for reacting to acquired world perception. This is an architecture that is very easy to implement but has a narrow range of

[6]`www.darpagrandchallenge.com`

applications being, nevertheless, very useful for certain situations. For example, it is very common to have a reactive architecture for mapping purposes. In the mapping process, the robot does not know the environment in which is going to navigate, and also the mapping algorithms tend to consume many computational resources. For those reasons, it is very common to implement a reactive agent for avoiding obstacles, wandering around and use the acquired information to perform mapping.

The reactive implementations also have an important part in space exploration. There are many robots designed for space exploration that are reactive. These robots are sent to other planets where they wander while gather information. These robots are mainly reactive since its function is simple, avoid obstacles and gather information. Also, since the resources are limited and the robot has to survive as much time as possible, one can spare on hardware and computational power thus implementing a reactive robot that very well serves its purpose.

There are a wide range of problems for mobile robots that the reactive navigation can not solve. For example, a service robot may use reactive navigation to build the map of the house in which it will serve, but it can not use the reactive navigation to perform any task within the map. For more complex and intelligent navigation, there is the deliberative approach. The deliberative approach on navigation is a more complex implementation for the navigation, and allows to perform more complex tasks. Considering the tasks they have to perform and the environment where they have to navigate, service robots and autonomous driving robots have to implement deliberative navigation.

Deliberative implementations for navigation often are implemented using world models and perception. The deliberative implementation, most of the times, uses world perception for build a world model in which the decisions for navigation are based. To make a decision or build a navigation plan, it is required to analyse the world model and take decisions according to the perception of the world in relation to the model available.

Reactive agents have a very short response time, which is desirable for real time applications and dynamic environments. However, reactive navigation approaches can not perform complex tasks unlike deliberative approaches. Deliberative approaches can perform numerous tasks but may originate response times not suitable for dynamic environments. To join the best of the two approaches it is common to see hybrid architectures, with a high level agent with a deliberative implementation which coordinates low level modules that are implemented in a reactive manner[42].

Most of the times, in a deliberative implementation, the navigation problem is addressed as a path planning or motion planning problem. This naming comes from the fact that more than a half of the navigation work is to compute the best navigable path to make the robot travel from one point to another. Also, the path and motion planning are common factors to most of all deliberative navigation approaches being the rest intrinsic to the application.

The motion planning problem, is a specific case of the path planning. Motion planning is the local path planning. Once the path planning generates a path that allows the robot to travel from one point to another, the motion planning is defined as the computation of the action commands required to follow the given path.

The motion planning is much more dependent of the robots motion model than the path planning itself. The path outputted by the path planning algorithm may be computed and filtered to be smooth and followable by the robot. However, the motion planning is directly linked to the action commands and thus directly linked to the robots motion model.

There is no key algorithm for the motion planning being this, in the majority of cases, handled by control algorithms such as Proportional Integral Derivative (PID) based or multi-modal[39]. Multi-modal control approaches tend to produce much better results since these are based on Markov processes, which imply a probabilistic approach that is far more safe and accounts to most of the errors a control system may have. However, PID control serves the motion planning purpose most of the times and depending on the application, it may have the same results as the multi-modal control. Also, multi-modal control is highly dependent from the observation of the state spaces and it is of complex implementation.

The PID control it of very simple implementation and common to most of the motion models. To use this is just necessary to compute the error signal, adjust the constants through experiments and apply the action commands accordingly. The simplicity and applicability of the PID control makes it most of the times preferable to the multi-modal control.

Before applying motion planning, it is required to have a path to follow. The most important part of navigation with a deliberative implementation is the path planning. The path planning is the computation of an optimal path from one point to another. The parameters which define a path as optimal depend on the application and the environment itself. However the two most common parameters considered for computing a path are the distance to be covered and the navigability considering obstacles avoidance and smoothness of the path. There are other parameters used for computation of an optimal path which depend on the application. For example, using an active localization system, the optimal path is considered the shortest possible that passes close to known landmarks so that these can be used while navigating for localization.

There are many algorithms available for computing the shortest path which is the basis of path planning. One of the first search algorithms published, and serves as basis for many others, is the Dijkstra algorithm[9]. This algorithm is a graph search algorithm which will find the shortest path between two nodes, considering the cost of each node transition. Considering that a grid map may be represented as a graph, representing cells as nodes and each cell transition has a cost, the Dijkstra algorithm may be applied to path planning. In Figure 2.9 is shown the application of the Dijkstra algorithm to the path planning problem. The algorithm will explore all possible routes in the graph until it finds a path to the goal node, it then continues to explore the graph trying to encounter any shortest path, any route search will be aborted if the cost exceeds the already found path. The Dijkstra algorithm can be slow depending on the map and the position of the goal, however, it is the base of many path planning algorithms based on graph search and it is still very used.

One of the algorithms based on Dijkstra that is also very used for path planning, is the A* algorithm[8]. The A* star algorithm, just like Dijkstra, explores the graph in order to find the shortest

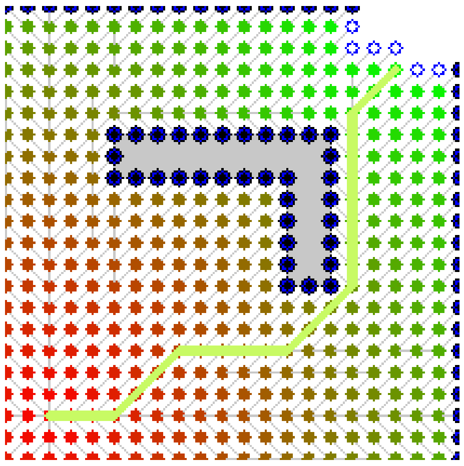


Figure 2.9: Representation of Application of the Dijkstra Algorithm for Path Planning[4]

path to the goal. However, the exploration of the graph performed by the A* algorithm is based on an heuristic function. The heuristic can not overestimate the distance to the goal, meaning that this can represent a straight line to the end point. With the heuristic, the exploration of the graph is more efficient since the algorithm may expand first the nodes that have a lower heuristic value. Considering this, it is more likely to find the shortest path to the goal without having to perform a long search through all the graph. In Figure 2.10 it is shown the application of the A* algorithm

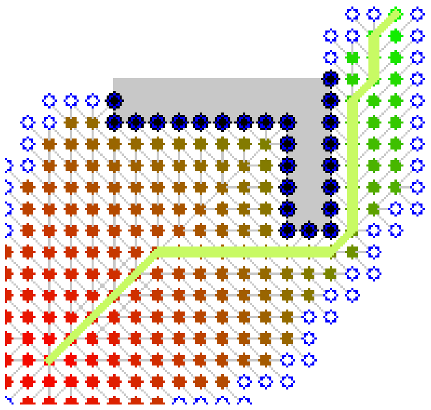


Figure 2.10: Representation of Application of the A* Algorithm for Path Planning[3]

to the path planning problem. Using the A* algorithm, the search through the graph becomes more efficient, however it is necessary to have an heuristic function for the goal.

There are many more variants of the Dijkstra and A* algorithms like D*[37], Field D*[38], GAA*[23] and many others. However, in all these implementations the resulting path is the shortest cost path independently of this being navigable or not by the robot. There are some changes that can be performed to these algorithms to accommodate this problem, for example, the cost function for cell transition can be dynamic, meaning that the costs in each node transition could accommodate the motion model of the robot. These accommodations are however not trivial, and the accommodation of the motion model is performed after the optimal path computation.

It is very common to use Dijkstra or A* algorithms for path planning in robotics. To accommodate the motion model of the robot, a smoothing operation is performed to the generated path in order to make it practicable by the robot. This smoothing operation is a filtering application to the computed path, in order to relocate points so the robot can follow the path. However, the smoothing operation may not be always successful originating a non-navigable path by the robot, relocating, for instance, path points to non-navigable cells.

There are path planning algorithms that search for an optimal path considering the motion model of the robot. The Search-Based Planning (SBP) algorithms compute an optimal path, considering motion constraints and shortest cost path. This type of algorithm is considered the state of the art for the path planning problem and it is described in following section.

2.3.2 State of the Art

The state of the art of navigation *per se* is not definable. There are many applications and different approaches for each application. However, one of the most developed navigation algorithms is applied to autonomous driving navigation. There have been some major developments also in indoor environments navigation applied to service robots.

The navigation implies several aspects, it includes artificial intelligence, path planning, motion planning and even machine learning in some applications. The most recent developments concerning navigation have been towards machine learning and the intelligent path planning.

The state of the art of path planning, which is the core of the navigation approach of this project, is the SBP algorithm[26]. This algorithm is considered the state of the art since it solves most of all path planning problems and it is highly configurable to any mobile robot.

Just like the previously mentioned path planning algorithms, the SBP computes the shortest path between two points, and it even uses an algorithm based on A* for finding the shortest path the graph. The particularity of the SBP regarding other algorithms, is the graph used for searching the optimal path. The Figure 2.11 shows the representation of a graph used by SBP for searching an optimal path. The graph used is built by the algorithm itself based on motion primitives. These motion primitives define the connections between nodes where each connection represents a movement dynamically feasible by the robot[26]. Also, each connection has a pre defined cost which represents the motion model of the robot.

Using this approach, the resulting graph is a tree of movements that is placed over the map. This tree is then used for finding an optimal path that leads to goal pose, and the result is defined as a vector of movements that can be translated to a vector of points used for motion planning. The use of

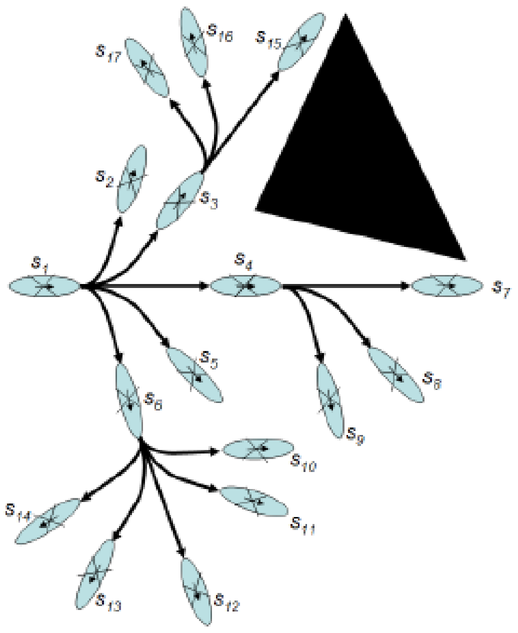


Figure 2.11: Representation of the Graph Used by the Search-Based Planning[26]

this algorithm, with the correct motion primitives, define a path that is optimal, but more important, it does not require filtering or processing for smoothing since it was built considering the motion model of the robot and thus is inherently feasible.

Figure 2.12 is shows a path that is a result of the use of the SBP algorithm using car like motion

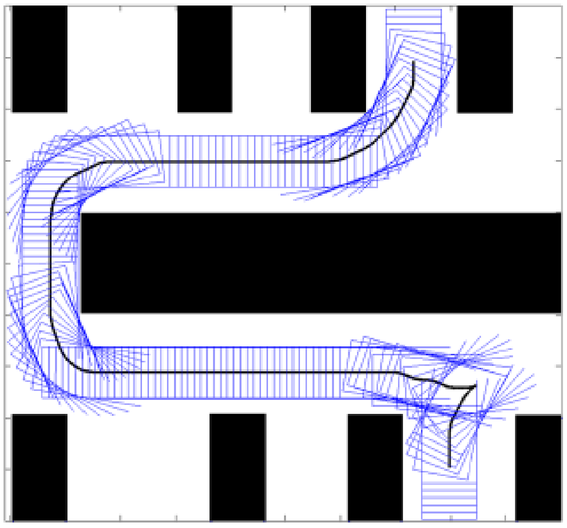


Figure 2.12: Representation of a Result Path Using the Search-Based Planning Algorithm[26]

primitives. As can be seen in the figure, the result is completely feasible by the robot, and it even includes maneuvers for more complex movements.

The real advantage of this algorithm compared with others, as it was mentioned, is its ability to include the motion model of the robot. It encompasses the motion model being this represented by the motion primitives, meaning that the motion primitives have to be created according to the robots motion model. However, the construction of the motion primitives is a small price to pay when compared to the advantages that this algorithm presents.

Being the construction of the motion primitives the user responsibility, which even allows to have different costs for each primitive, not only shows the high range of applications where it can be used but also shows this algorithm is highly configurable. The creation of the motion primitives is done with the knowledge of the robots motion model, but its resolution its not defined. This means that one can build a high number of primitives for a high resolution path planning, but can also define a lower number of primitives for reducing the computation time, originating a resolution versus computation time trade-off. However, just like it happens with other mentioned algorithms, one can use low resolution motion primitives, that are feasible by the robot, and then handle the result path with motion planning thus reducing the computation time.

Chapter 3

Robotic Platform

This chapter describes the robotic platform used in this project. The used robotic platform was already available in the ROTA project and some modifications were made to accommodate new features and new approaches on the autonomous driving problem. These modifications are documented in this chapter as well as the choices made to support them.

The robotic platformed used in this thesis is an adaptation of the robotic platform that was already available in the ROTA project. The mechanical and software details of implementation of this robot were first described in [1]. The physical infrastructure is based on an Ackerman steering system, equipped with several sensors, being the main one a vision system. Control is split into two layers, called low level and high level control. The low level control is close to the physical devices and is based on a set of interconnected microcontroller modules. The high level control run in a computer and is connected to the low level control through a communication link.

Two robotic platforms were built, one named ROTA, after the project's name, and the other named Zinguer. The main difference between the two platforms is the size, since the latter was built bigger in order to have space to encompass a laptop to run the high level control, while in the former it runs in a single board computer. Zinguer was the platform used and adapted during this work. Figure 3.1 shows this robotic platform, after all the adaptation process. In the following subsections more detailed descriptions of the different modules are given, highlighting the adaptations done.

3.1 Physical Infrastructure

The used platform is a car like robot, with an Ackerman locomotion system. It has a rectangular form of sixty by forty centimeters which is compatible with the rules of the Autonomous Driving Challenge and the size of the track. The size of the robot is merely to accommodate the various

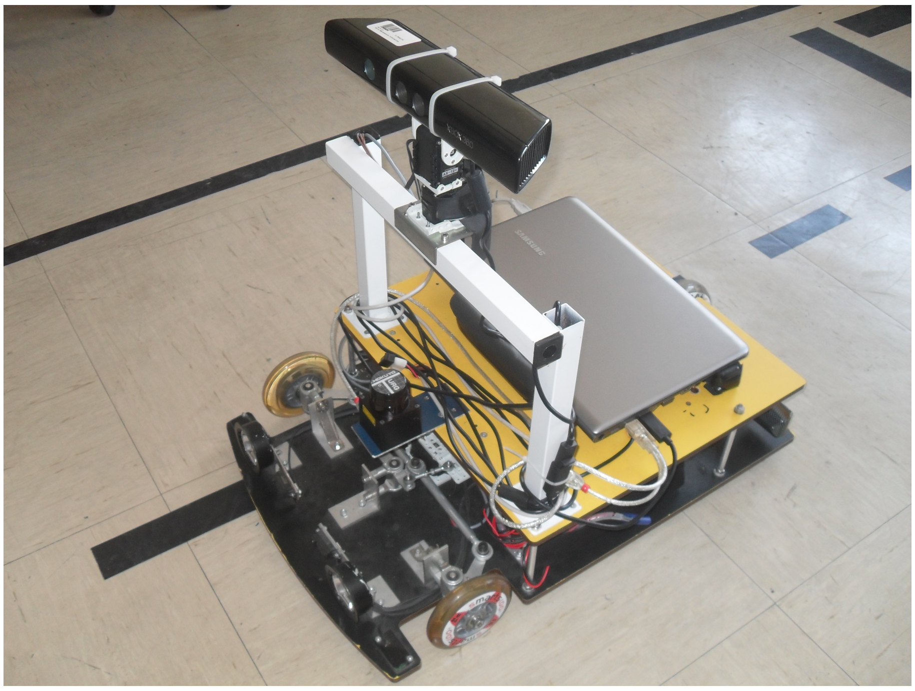


Figure 3.1: Picture of the Robotic Platform used, the Zinguer car

sensors and actuators as well as the laptop which is the central processing unit in the robot. The sensors and the actuators of the robot were chosen and projected aiming the autonomous driving problem and the Autonomous Driving Challenge of the Portugal Robotics Open. By consequence, the sensors and actuators reflect the needs of the robot.

The robot has two infrared sensors placed under the front bumper on its extremities. Since the track is drawn with white lines in a black ground, these sensors allow to detect the lines of the track. This feature is not aiming the autonomous driving itself but it allows to have an easy line detection system which may be used as ground base. These sensors can be used for crosswalk detection allowing to correct localization errors when approaching the crosswalk. Also, it could be used to detect the external lines of a lane allowing to implement security features and ground truth for transversal localization on the track.

The robot has also on its bumper, but on the top of it and facing forwards, two infrared distance sensors. These sensors are placed in the mentioned pose to allow a low level obstacle detection. With these two sensors it is not possible to identify the form of the obstacle or what is its nature. Also, these sensors have a short detection range, making these only usable, as the line sensors, for security and ground truth purposes.

The robot also has, as primary sensorial source, a vision system which is used for signs detecting, obstacle detection and line detection for localization purposes. In the first implementation of the robot, this was composed of two fire-wire web cameras both directed towards the robot front, one aiming up for sign identification, and other aimed to the ground for driving purposes[1]. This vision system, was changed to a single vision sensor, which implied an addition of actuators for controlling the camera pose, namely a pan and tilt structure. Due to its major importance in the robotic platform, the vision system is explained and detailed further ahead in this chapter.

The robot also has, as from this thesis, another perception sensor, an LRF. This sensor was not available in the previous implementation of the robotic platform and it was added in this year project. The LRF was added after the participation of this thesis project on the 2013 Autonomous Driving Challenge. It was added due to the difficulties felt on the detection of the obstacles that came from the new approach of the vision system and all the tasks it has to perform.

The robot also has motion sensing that was improved in this project. In the previous implementation the motion sensing was only performed by the back wheel. The steering did not had feedback, meaning that the position of the servo had to be estimated from the last applied order in order to calculate odometry values. This was improved by changing the steering servo for one with feedback.

Since the robot is a car like robot with an Ackerman locomotion system, it is inferred that the robot has a traction system and a steering system. The traction system adopts a tricycle solution with only one traction wheel. This traction system is easy to implement since it only requires a single motor and no mechanical differential as a two wheel traction system would require[1]. The locomotion system is inherited from the previous implementation with the already mentioned exception of the steering servo. The change of the steering servo did not infer any change on the physical infrastructure of the platform, the modifications performed to accommodate the new servo with feedback where made at the low level control, level which is explained further ahead in this chapter.

As it would be expected from a car like robot it has a set of lights that are typically encountered in a car. The robot has two turn signals, one for left and other for right, head lights that is aimed to be used when light conditions are not favorable and a stop light. The stop light is only mounted in the ROTA robot, Ziguer does not have one due to lack of a proper space to mount it.

The two robots, Zinguer and ROTA have the same infrastructure, being the powering system no exception. In the previous implementation, the robot was powered by two NiMH of 12 Volts. One battery for powering the control electronics and another for the motor and lights[1]. Within this project, the powering system of the Zinguer robot was changed, and the Zinguer robot is now powered by two lithium-ion polymer batteries, often abbreviated to LiPo.

This change was made by several reasons. The first reason that led to this change was the fact that the lithium-ion batteries were already worn out with very reduced autonomy. The second reason, that led to the alteration instead of acquiring new NiMH batteries, was the fact that the CAMBADA robots, also a project of the ATRI, had a powering system based on LiPo batteries. In an attempt to standardize the powering system of the ATRI projects, and since the Zinguer had to acquire new batteries, it was opted to install the new powering system with LiPo batteries. The third and last reason for remodeling the powering system, was the installation of the new vision system, that with the previously installed power system would require a third battery. LiPo batteries have, as main advantages, an improved autonomy and a much higher discharge current when compared to NiMH batteries.

The installation of the powering system led to other changes in the low level control. The first alteration was the replacement of the circuit board in charge of generating all the necessary output voltages for the several componentes of the system. The second was the addition of control modules for the batteries. Since LiPo cells can not discharge more than a given threshold voltage, or the cell will no longer accept a full charge, these have to be monitored preventing that situation to occur. For such monitoring, the LiPo batteries have two connectors, one that is the source which is the output of the set of cells, and other that allows to measure the voltage in each cell.

3.2 Low Level Control

The low-level control layer is composed of a set of nodes interconnected by means of a Controller Area Network (CAN) network[1]. A gateway node interconnects this layer to the computer running the high-level control, through an Universal Serial Bus (USB) port. The various nodes that constitute the layer are based on the same underlying hardware module. Figure 3.2 shows a general diagram of the low-level control layer.

Some of the nodes have been kept almost unaltered since the first version of the platform. The

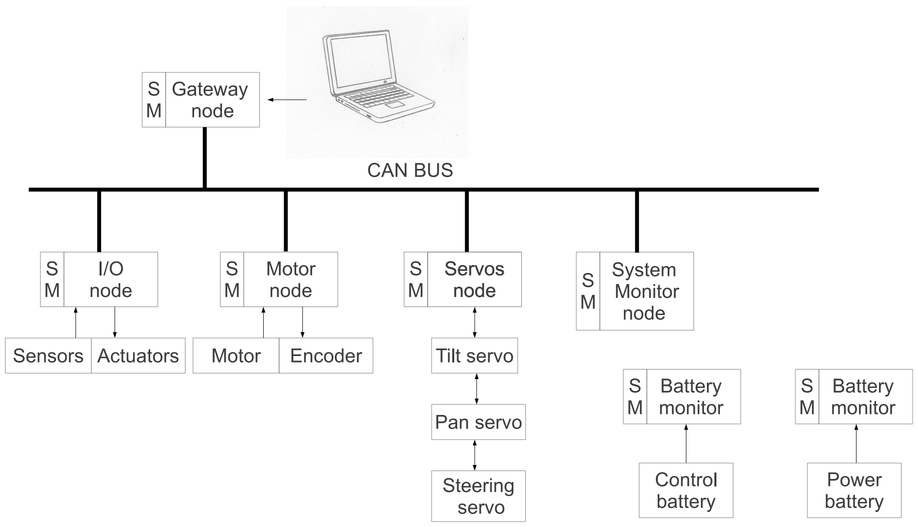


Figure 3.2: Zinguer Low-level Architecture[1]

motor node and the I/O nodes fall in this category. Some nodes have suffered alterations, namely the gateway and the monitor nodes. Finally, there is a new node for the pan and tilt structure as well as for the steering servo.

Gateway node The function of the gateway is to interconnect the micro-controller based low-level sensing and control system to the high-level control running on the laptop. The communication is performed through an USB port of the laptop that is connected to the gateway node.

The gateway node receives data messages from the various nodes attached to the CAN BUS and sends them to the laptop. The inverse operation is also performed, which means that the gateway receives data messages from the laptop and sends them to the nodes attached to the CAN BUS.

Motor node The motor node is responsible for interface the motor and its encoder with the higher-level. This node will receive, via CAN, actuation orders for the motor, namely a velocity, and will use this velocity as set point for its internal closed loop controller, a PI[1]. As feedback for the controller, the motor node uses the data sent from the encoder attached to the motor. The feedback data sent by the encoder is also sent to the higher-level by the motor node.

I/O node The function of the I/O node is to translate orders received by the higher-level to the actuators, and to send sensor readings to the higher-level. This node is associated to the previously described infrared distance sensors and line sensors, and also to the lights.

System Monitor node The monitor node has as function to monitor all the distributed system attached to the CAN bus. Its function is to monitor reset situations from the modules, and answer to *alive* requests sent by the higher-level layer. This way, the system is monitored for any irregular situation and the higher-level can test connectivity with the low-level control. In the previous implementation described in [1] this node was also responsible for monitoring the state of the batteries, task that is now performed in a different module.

Battery Monitor The battery monitor is responsible to read the state of the batteries and send the read state to the higher-level. As previously mentioned, the powering system of the robot was altered to support LiPo batteries, which led to the alteration of the batteries state monitor node. The cells that constitute LiPo battery can not discharge under 3.2 Volts. This means that the monitor node, for safety reasons, has to inform in a very visible manner that the batteries can not discharge any longer. For that reason the battery monitor is not attached to the CAN bus, but it is an objective to the future so the robot may also have the battery state information and may be implemented high level behaviours for dealing with the aforementioned situation.

This module is oriented to the user instead of the robot itself. It contains a buzzer and a blue Light-emitting Diode (LED). When the battery monitor connector is attached to the monitor the buzzer beeps indicating that is functioning. While in normal monitoring the blue LED blinks. When any of the cells of the battery reaches a 3.2 Volts, the buzzer beeps until the battery is disconnected, and the LED blinks with a higher frequency.

Servos Control node In the previous implementation, the robot only had a servo, the steering servo. This means that, in the previous implementation, this node was the responsible for traducing the steering orders from the higher-level into action commands for the servo. This node, in the current implementation, remains with the same function however, it contemplates also the pan servo and the tilt servo, which were added as a part of the new vision system. Since all three servos have feedback, this node also comprises the function of sending this feedback to the higher-level.

All three servos are controlled by the same node due to the servos itself. The servos used, all three of them, are servos from the AX-Series from Dynamixel, namely the AX-12 servo[10]. These servos allow to have a multi drop link to connect all servos to a single controller node as represented in Figure 3.3. The messages are then sent to all servos through this link, identifying in the message

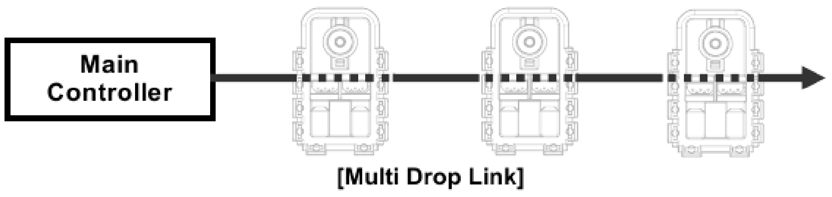


Figure 3.3: Multi Drop Link for AX-Series servos[10]

the destination servo(s).

These servos were chosen to implement the pan and tilt structure because they have been a study element of the ATRI group, being the servos already available for usage, as well as an already implemented control module. As for the steering servo, it is the same as the servos used for the pan and tilt since no more hardware is required for installing it. The steering servo is attached to the already created link for the pan and tilt servos and the firmware of the control node is altered to accommodate one more servo.

3.3 Vision System

Vision is the main source of sensory information of the robot. It is used to acquire road information and traffic sign information. Before the modification carried on during the work underlying this thesis, it was composed of two fixed cameras, one pointing down, used to acquire the road information, and another pointing up, used to acquire the traffic sign information.

This configuration presented some problems. The vision system needed to be calibrated in order to get the correct function between pixel information and world coordinates. The slightest change in the camera pose, would have major effects in the line detection accuracy.

Another problem of this configuration has to do with the type of connection used. To connect the two firewire cameras to the laptop, an express card was used. This offered many connectivity problems related with drivers and poor contact that often resulted into a series of plug and unplug until the cameras were finally connected. A firewire bus solution was also attempted, using a laptop with a firewire port. However this solution also revealed driver problems and fragility in the connection to the laptop.

ROS provides a driver for firewire and USB cameras, however it is not compatible with all the models of cameras. The cameras used in the previous vision system were partially supported by ROS, it required some modifications to the offered driver for a proper functioning.

Because of the aforementioned problems, it was decided to substitute the firewire cameras. For the new vision system, a Kinect camera put on top of a pan and tilt module, was chosen.

Being an RGB-D camera with two degrees of freedom, a number of new challenges were

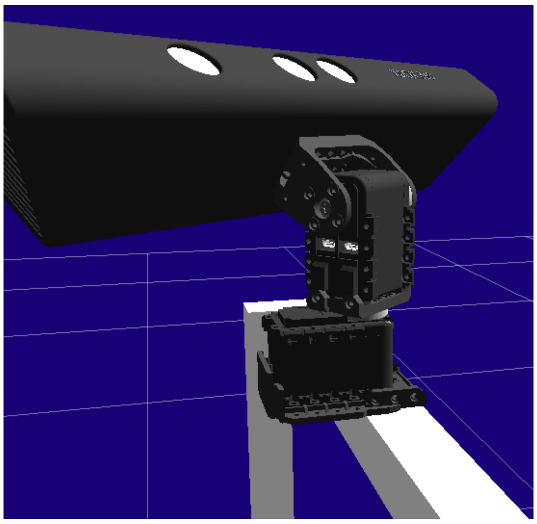


Figure 3.4: Vision System of the Zinguer Robot

put on the project. With the used of an RGB-D camera, it is not necessary to have calibration for transforming the pixel into world coordinates. It is possible to register the depth and RGB images so that each pixel has a distance associated. Using these two images it is possible to compute the lines of the track.

The use of the depth image is also an advantage for sign detection. Knowing the signs shape and displacement, one can use the depth image to detect the sign and dynamically define a region of interest. With a region of interest defined it becomes easier to use the RGB image for identification.

The use of the depth image also gives a new possibility for obstacle detection. Using the depth image for obstacle detection is less error prone than using the RGB image. Also it is possible to estimate the full location and size of the obstacle using depth perception.

The problem that arose with the use of the pan and tilt configuration was on how to set a pose. Not only to detect the lines on the floor, but also to detect vertical signs and the semaphore signs. Even to detect the lines on the floor, the camera pose should not be fixed favouring the capture of information for detecting points of interest in the track. The use of this approach is also very interesting to research: a human being is able to drive a car only using a set of eyes, choosing, in each moment, where to look. This is an interesting problem to model using artificial intelligence.

Chapter 4

Software System Architecture

This chapter describes the high level control layer of the system. It starts by presenting the overall picture of this software layer, going then in more detail in the description of the several constituting modules. Implementation details and obtained results are kept out of this description, being postponed to the next chapter.

As stated before, the high-level control layer is a piece of software that runs in a laptop and implements the global control of the robotic vehicle. It, in a short statement, collects data from the Kinect camera and from the low-level sensing system and sends actuation orders to the motion control and to the pan and tilt module.

It had to be almost completely rewritten, due to a number of reasons. It was a requirement that the software should run under the ROS middleware. This, by itself, make it necessary to adapt any existing software that one would want to port. Additionally, ROS encompasses several packages for autonomous driving purposes, which are almost mandatory to use. The vision system is completely different from the previous, making it easy to rewrite from scratch the image processing code than to adapt the existing one.

Since little code was to be ported, it was decided to follow a new design approach, taking full advantage of the ROS communication layer. This led to a software architecture based on nodes, topics and services, following a publisher-subscriber architecture, as is encouraged by ROS.

The vehicle has to accomplish two different main tasks. As a project's objective it has to autonomously navigate in a known map, a road like environment, where some known objects, obstacles and roadwork cones, can be placed in unknown locations. Also, it was a requirement that an RGB-D vision system with pan and tilt capabilities should be used as the main sensor device. This navigation task is hereafter referred to as the normal operation of the vehicle. Figure 4.1 shows an overview of the high-level control software during the normal operation of the vehicle.

The vision module is responsible for gathering the world perception information and it is divided

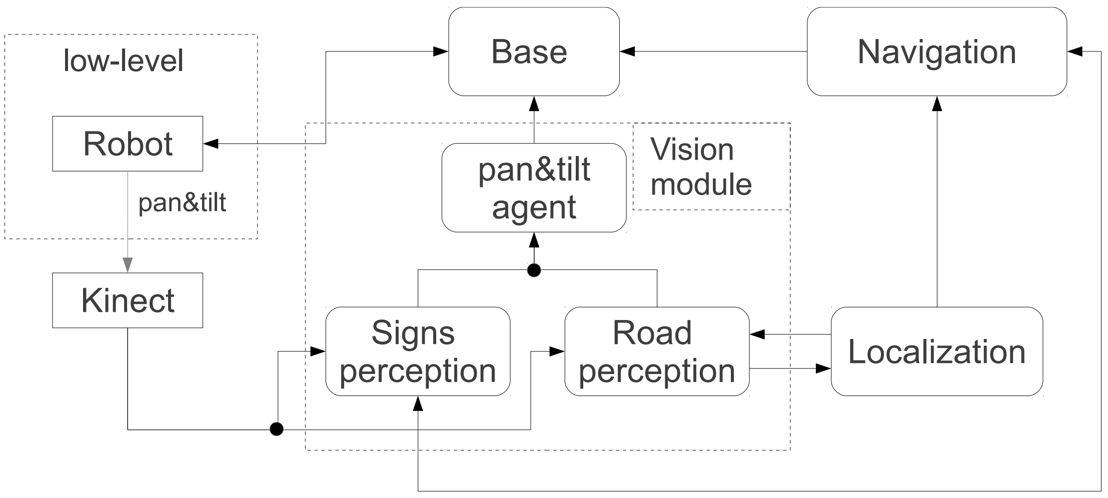


Figure 4.1: System Overview for Navigation

into three sub modules. The first, road perception, is responsible for gather information of the delimiting lines of the track for localization. It also has the function to compute the best pose for the Kinect to detect that information. The road perception also includes obstacle detection that is used by the navigation module to update the navigation map. This module is also responsible for detecting and identifying the vertical signs and traffic light signs.

Since the vision module has different functions, and the vision system is composed only by a single vision sensor, the Kinect camera is mounted over a pan and tilt device. This device is controlled by a dedicated agent which receives requests from all vision modules and sets a pose accordingly. This agent is the only agent that directly applies a pose to the pan and tilt.

The base module is the module responsible for establishing the communication between the high-level software and the low-level. It is responsible for translating the action commands sent by the high-level software to hardware readable values. This is applicable in the inverse course. The base module is also responsible for translating the values sent by the low-level into meaningful values and make the state of the robot available for the high-level software.

The localization module is responsible for providing the robot's pose within the map. It take as input the state of the robot and world perception data, and outputs the pose of the robot in a known map.

As for the navigation module, it contains all the software related to the navigation. It contains the requirements for path planning and motion planning. It is responsible for updating the cost maps, adding or clearing areas, with the received world perception data. With the cost maps, path planning and motion planning, it computes a path to a given pose and follows it. It also contains the main navigation agent responsible for the deliberative navigation of the robot. It manages and monitors the target poses in which the robot should pass by and implements the driving behaviour.

The official layout of the track is known in advance, making it possible to build the navigation map by hand and use it afterwards for navigation. But, the actual track can have small differences due to assembly faults. Thus, it was decided to set up a framework to automatically map the track. Additionally, this approach make it possible to navigate in environments not known in advance.

A second project's objective was to allow the vehicle to automatically acquire the map of the environment where it is going to navigate afterwards. Figure 4.2 shows an overview of the software for mapping purposes. Some of the modules are the same as for the normal operation of the vehicle

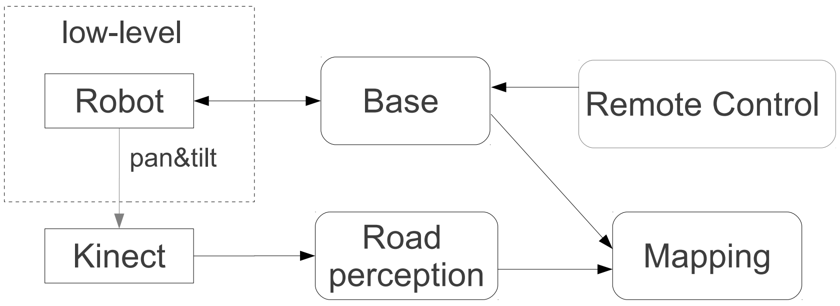


Figure 4.2: System Overview for Mapping

and so their purpose was already presented. The new modules are the mapping and the remote control modules.

The function of the mapping module is to build a map of the environment, namely the track, with the information gathered by the road perception module and the odometry information given by the base module. The remote control module allows to interface a joystick, or other input device, with the robot, for controlling the motion of the robot and the camera pose. This allows the supervised data collection for mapping.

The passing of information between modules is accomplished using either ROS messages or ROS services. The modules functionality will be further detailed in the next sections.

4.1 Base Module

The base module is responsible for the communication between the computer and the robot's base. It is also responsible for publishing the state of the robot. In one direction, its function is to translate the high-level orders into proper low-level orders, encode them into proper messages and send those messages to the low-level control layer. This module also functions for the inverse course, being responsible for the reception and parsing of the messages received from the low-level control layer, and also, to transform these into meaningful information.

In order to make the translations, it is necessary to have the model of the robot, or, in this case, the model of the car. As an example, the translation of a velocity order to the proper traction motor set point requires a number of parameters associated to the gear system. As another example, the translation of the encoder count to odometry data requires the knowledge of the conversion factor between the encoder counting and number of turns of the motor, as well as gear and wheel parameters to convert the encoder count into a covered distance.

This module, besides communicating with the car, is also responsible for publishing the dead reckoning state of the car as well as all the other sensorial information received from the low-level. The dead reckoning as well as the state of the robot are processed by this module in order to publish the complete tf tree for the robot. This allows to have the complete state of the robot available in the ROS system. With the state of the robot published through the tf, any ROS component can access it using the tf Application Programming Interface (API).

To be able to translate the sensorial values sent by the low-level, the base module has a set of configuration files in which are defined all the necessary parameters of the car to perform such translation. It contains the traction gear parameters for translating the encoder count into a covered distance, the length between axis to perform the odometry calculation, the functions for converting a pan, tilt and steering angles into hardware values and others.

Another important part of the robot's configuration, is its physical description. This description contains all its frames and movable links between them and its implemented using a Unified Robot Description Format (URDF) file. With this file, it is possible to associate the information sent by the low-level to the joints of the robot. The combination of both, makes possible to publish the state of the robot through the tf tree. The publication of the state of the robot, in which is included the dead reckoning, is performed using ROS components and for further details please refer to 5.1.

This module contains the software that supports the communication between the high-level and the low-level at the laptop's end. The laptop end link is supported by a software piece in the base module that serves as a drive. The two main functions of this piece of software is to read and write to the USB port, parsing the received messages and constructing the messages to send. It also traduces the low-level values from the parsed messages into high-level values which will then be used within the base module to produce the state of the robot. This task is accomplished by means of the configuration files and robot's model previously mentioned.

4.2 Remote Control Module

The remote control module allows to interface a joystick or other input device with the robot. This module receives commands from a joystick device and translates them into action commands for the robot actuators, namely the motion system, steering and traction, the pan and tilt and also the lights. This module allows a user to fully control the robot.

There were two main reasons for the inclusion of this module on the system: aiding in the mapping procedure and supporting development. Mapping is accomplished in a two-fold procedure.

First, road perception and dead-reckoning data is acquired and stored. This is done using the bagging support from ROS. The data is afterwards used to run the mapping algorithm. The quality of the acquired data is determinant for the quality of the computed map. Thus, data can be acquired in a supervised mode, using the remote control device to both make the vehicle follow an appropriated trajectory and point the Kinect in the best way.

To test the localization capabilities of the vehicle, a joystick can be used to force the vehicle to follow a given trajectory and check if it localizes well. This play mode is not represented in the diagram of Figure 4.1, but it corresponds to substitute the Navigation module with the Remote Control module. Additionally, the remote control capabilities can be used to drive the vehicle from one place to another, avoiding having to carry it by hand.

This module interfaces with the joystick using an available device driver and publishes action commands based on key's strokes. To map the keys the module has a set of configuration files, one for each device used as remote controller, that uniquely identifies each key. The purpose of the configuration files is so that the software reads the joystick commands in the same manner independently of the device used. The read keys are then translated into action commands that are then published.

With the modification of the steering servo of the robot it was necessary to build a new function for translating an angle value to a servo set point, readable by the low-level. To create this new translation function one has to measure a set of applied steering angles from a set of servo set points. This measure is performed indirectly by measuring the diameter of the circumference described by the robot at a given servo set point.

To make the construction of such a function easier, a piece of software was implemented in the remote control module. This piece allows to select a servo set point and use the remote controller just to control the velocity, allowing the user to describe a full circumference. This piece of software also maps remote controller keys to perform a measure of a radius from a completed circumference, make averages with multiple measures and show the resulting value of the steer angle for a given servo set point in the terminal.

4.3 Road Perception Module

This module captures images from the color and depth cameras of the Kinect device, processes them in order to extract the data required by the Localization and Mapping modules and, finally, publishes that data. So as to get the best view of the road to accomplish the previous task, it also computes and publishes the corresponding Kinect's pose.

The Localization and Mapping modules require points from the road lane delimiting lines in the form of a laser scan. Thus, the data extracted from the color and depth images has to be transformed into laser scan data. The followed approach unfolds into three steps:

- extract delimiting line points from the color image;
- validate these points;
- convert the selected points to a laser scan.

The extraction of delimiting points is done through the use of a set of image scan lines, radial distributed over the gray scale image, as can be seen in Figure 4.3. This set of image scan lines are referred to as a radial vision sensor. A set of parameters can be used to configure the vision sensor,

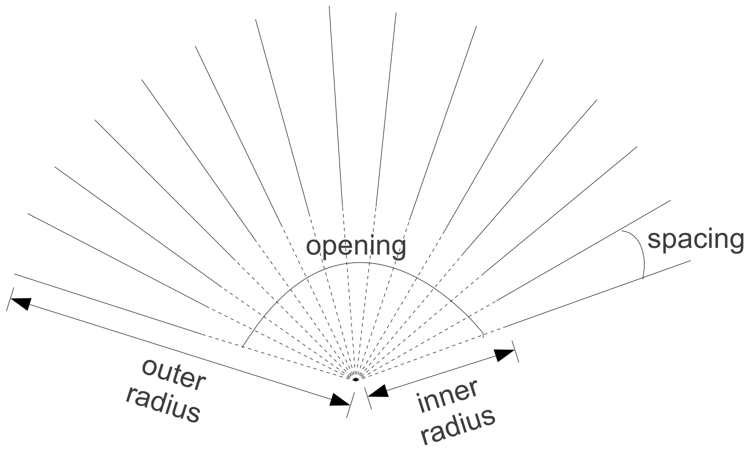


Figure 4.3: Representation of Radial Sensor

such as the inner and outer radius or the spacing between scan lines, among other attributes. To speed up processing time, each scan line is converted to a list of pixel coordinates.

The scan lines are processed inside out looking for transitions that may possible represent delimiting points. The detection of transitions is accomplished using a derivative operator. The discrete derivative operator, the differential operator, is applied between adjacent pixels until its value is higher than a given threshold, meaning that a significance color transition occurred which may represent a delimiting track line start point. After detect the first transition in a scan line, instead of making the difference between adjacent pixels, the interval for differentiation raises until the difference between the first and the following pixels is higher than the defined threshold. The increase of the differential interval stops when the differential value shows a value bellow the threshold. The described method for detecting the delimiting line points is accomplished by the pseudo-code shown bellow.

```
for $i = 1 \to sizeof(scan\_line)$ do
    $k \leftarrow 1$
    if $|Image[scan\_line[i-1]] - Image[scan\_line[i]]| > threshold$ then
        $j \leftarrow i - 1$
        $k \leftarrow 2$
        while $j + k < sizeof(scan\_line)$ and $|Image[scan\_line[j]] - Image[scan\_line[j + k]]| > threshold$ do
            $k \leftarrow k + 1$
        end while
    end if
    if $k \geq step\_thresandj + k \neq sizeof(scan\_line)$ then
        add scan_line[i] to line point list
        add scan_line[j+k-1] to line point list
    end if
    $i \leftarrow i + k$
end for
```

With this method, when a scan line crosses a delimiting line of the track, a step as wide as the line in the image is produced. Both pixels representing the extremities of the step are considered delimiting line points. The expected result when a line is detected by a scan line is represented in Figure 4.4. In the figure is shown, in blue, if the pixel is considered to be in a line and, in red, the value of the gray scale pixels normalized between zero and one.

The width of the step allows to filter much of the noise since the minimum and maximum allowed

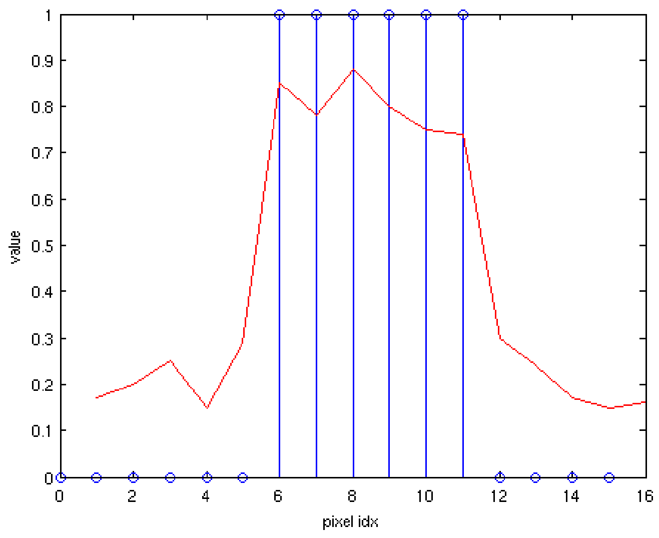


Figure 4.4: Graph representing the Detection of a Line

width, for considering the step a line, are user definable. The differential threshold value to consider a transition is also user definable allowing to adapt the delimiting line point detection to light or track conditions. In Figure 4.5 is shown the final result of the aforementioned procedure. The red lines

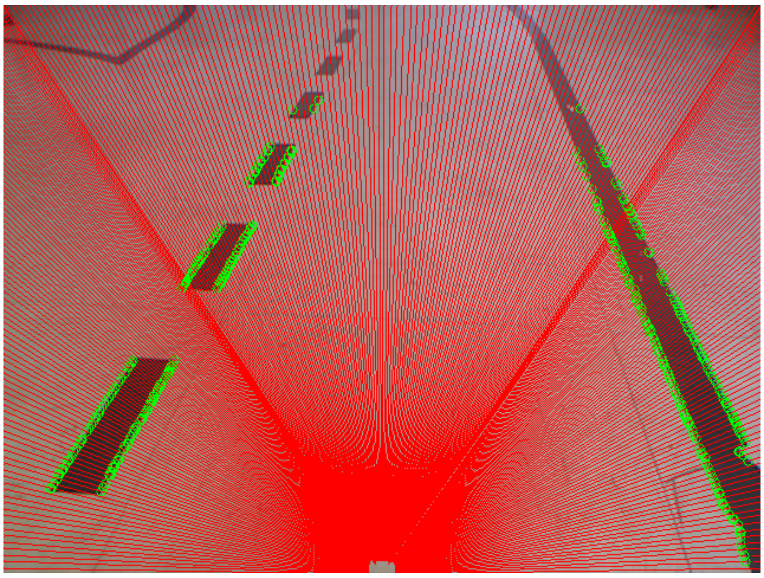


Figure 4.5: Final Result for the Delimiting Line Points Search

represent the radial sensor and the green circles represent detected line points. It is possible to see that the ground is noisy due to the floor pattern and the aforementioned procedure allows to detect the nearby delimiting line points without outliers, at least in the case represented in Figure 4.5.

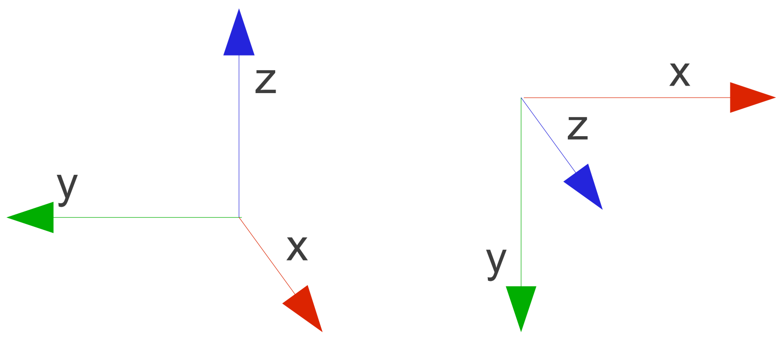
The number of lines of the radial sensor may seem exagerated however, it is necessary to detect as many points as possible to enhance the localization and mapping process. Also, the number of points will improve the outlier rejection capability of the localization and mapping algorithms. The large number of points detected is also related to the laser scan data generation as it will be explained further ahead.

The detected points are just good candidates to be delimiting points. A validation step follows towards the selection of those being actually delimiting line points. It is possible to have outliers, not due to noise on the image or track, but due to obstacles on the track. For example, since the roadwork cones are orange and white, a white stripe of the cone may be detected as a delimiting track line. It is necessary to account for these kind of outliers and eliminate them. The validation of the points is only performed using its position information so that points originated from obstacles may be eliminated. Hereafter is explained how to transform pixel coordinates into spacial coordinates in order to validate the position of the delimiting points detected. This transformation is also required in order to build a laser scan or point cloud so that the Mapping and Localization modules may use this information.

It is possible to transform pixel information from the Kinect camera into spacial coordinates using also the depth image. To do that, one can use the transformations that ROS uses for generation of the point cloud[22]. The z coordinate of the point is directly the distance provided by the depth image. To obtain the x and y coordinates of each pixel one can apply the equations shown in 4.1 which combines the depth and color images to transform (u, v, z) into (x, y, z).

$$\begin{cases} x = (u - \frac{w}{2} \times (z + minDistance) \times scaleFactor \times \frac{w}{h} \\ y = (v - \frac{h}{2} \times (z + minDistance) \times scaleFactor \\ z = z \end{cases} \tag{4.1}$$

In equations 4.1 the *minDistance* and *scaleFactor* are parameters that come from the Kinect specifications. The *w* is the width and *h* is the height of the captured image. The coordinate system resulting from the transformation between pixel coordinates to the spatial coordinates does not have the configuration desired as it is shown in Figures 4.6(a) and 4.6(b). With this transformation, the points are represented relatively to the Kinect's pose and it is necessary to transform them to be represented relatively to the robot's pose. The situation described by the Figures 4.6(a) and 4.6(b)



(a) Robot's Pose frame Axis (b) Kinect's Pose frame Axis

Figure 4.6: Representation of the Coordinate Systems for *rgb_camera_link* and *base_link* frames

can be solved by applying a rotation and translation considering the displacement of the Kinect's coordinate system relatively to the robot's coordinate system. After the transformation the delimiting line points are represented in spatial coordinates relatively to the robot's pose.

When the result pretended is a point cloud, this is directly the one obtained by the aforementioned method, otherwise, if the pretended result is a laser scan it is necessary to build it using the already obtained points. To transform a (x, y, z) point into proper laser scan it is necessary to understand how the laser scan is defined. A laser scan data set is a vector of radial ranges where all the ranges begin in the same point and are equally spaced with a fixed angle. Every range has a specific angle, often mentioned as bearing, which indicates in which direction the measure occurred. The vector of ranges is filled from its initial range to the last, according to the sweeping of the laser scan, that may be clockwise or counterclockwise. Also, all the ranges in the laser scan data set are assumed to be taken from the same plan.

To guarantee that all ranges come from the same plan, the ground plan, only the points extracted that belong in a certain range of z are admitted. This step also serves as outlier rejection since the line points are on the ground plan. Since the points are represented relatively to the robot's pose, the range of z admitted can be $[-0.06, 0.06]$ meters to allow some error in the measurements, transformations and to take into consideration the depth resolution of the Kinect that is of 1 centimeter. The range admitted to assume that a point is on the ground plan was experientially obtained by tunning the range until all the outliers were rejected and non real point was rejected. If the z coordinate value of the point belongs to the pretended plan, then it can be added to the laser scan data set.

To calculate the laser scan components of each point, range and angle, it is used simple trigonometry functions. The range is calculated using $range = x^2 \times y^2$ and the angle of this supposed beam with $angle = -atan2(x, y)$. The range then is stored in the ranges vector and its position in the vector is determined by its angle. It is possible that is not found a point to satisfy every range position and it may occur that multiple points are found for each range. For that reason, when the vector for the ranges is initialized it is filled with the maximum range plus one, invalidating the positions of the vector that do not get replaced. Also, it is important to build a point cloud with as much points as possible to choose the one that best fits a position in the vector. To allow such situation, the laser scan has less points than the sensor, and in that case, one may chose a better fitting of the detected points to the laser scan.

The algorithm for converting the point cloud data into a laser scan is based on the *pointcloud_to_laserscan* [1] node. This node is not used directly to avoid having another layer of processing between the delimiting line points detection and the Localization module. The detection of points, transformation, validation and insertion into the proper data structure is performed in the same processing cycle thus gaining processing time.

The detection of the delimiting line points of the track is conditioned by the Kinect's pose. This should be setted in order to detect as many points as possible. In order to do that, the Kinect must be posed so that the captured image has the most lines possible considering the field of view of the robot.

Since the map is a binary image, the main idea of the algorithm for choosing the camera pose is to load the binary image of the map, build possible vision windows according the robot's pose and the field of view of the Kinect, choose the window with highest sum value and then calculate a pose for that given window.

The first passe of the algorithm is to calculate a set of windows that represent the image that the robot may acquire. There are many problems on the calculus of the windows. First of all it is necessary to define which is the field of view of the robot considering the field of view of the Kinect and the maximum and minimum pan and tilt angles.

The maximum and minimum poses for the pan and tilt are not truncated by the physical limits of

[1] `www.ros.org/wiki/pointcloud_to_laserscan`

the pan and tilt, but rather for what is useful to see. It does not make sense for the robot, in order to localize, to look down at a ninety degrees angle because it will see itself. To avoid this, the minimum distance, measuring from the center of the robot towards its front, that the robot may look is 0.4 meters. This distance avoids that the Kinect captures the robot. The same applies for the maximum tilt angle. It does not make sense for the robot to look for more than simply ahead (0 degrees for tilt) because in that case the robot will see other than road. The maximum distance that the robot can see is defined as 2.8 meters. More than this and the information is not useful since the lines to detect will be too small in the RGB image and by that, untraceable when compared to the noise.

The pan is limited so that the image captured dos not contain the robot. If the pan angle becomes greater that $\frac{\pi}{3}$ with the previous defined minimum tilt angle, it will cause the robot to see the structure holding the Kinect, and by that it is being lost vision information. Having defined maximum and minimum distances, one can calculate a spatial window to where the robot may look.

The field of view of the robot, considering its description, is delimited by two semi circumferences. To make the processing easier and enforceable with simple image processing functions, the field of view is approximated to a rectangular window.

In Figure 4.7 is shown a representation of the field of view of the robot, where the dashed semi

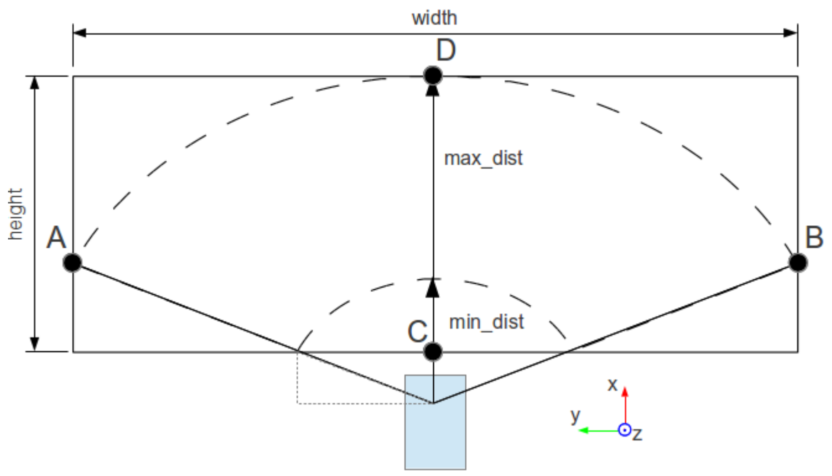


Figure 4.7: Illustration of the Field of View of the Robot

circles delimit the real field of view, and the rectangle represents the approximation. To calculate the *height* and *width* of the defined window one can use the equations shown in 4.2 that are supported by the scheme represented in Figure 4.7.

$$\begin{cases} width = \frac{\overline{AB}}{map\_resolution} = \frac{2 \times max\_dist \times \sin{(opening/2)}}{map\_resolution} \\ \\ height = \frac{\overline{CD}}{map\_resolution} = \frac{max\_dist - min\_dist \times \cos{(opening/2)}}{map\_resolution} \end{cases} \tag{4.2}$$

The *opening* is the opening angle of the field of view, in this case the maximum angle allowed is $\frac{2\pi}{3}$, however, this is a configurable parameter.

Having defined the window for the complete field of view of the robot, it is necessary to divide the defined field of view into windows that represent each pose of the Kinect. The size of the windows depend on the Kinect's field of view and also its pose. The Kinect has 57 degrees of horizontal opening and 47 degrees of vertical opening. To define the multiple windows, the developed algorithm starts by computing the maximum and minimum tilt angles that allows the image to be inside the defined field of view of the robot. The maximum and minimum tilt are represented by two points, one at the top and one at the bottom of the robots field of view. The windows that these points originate can not go out of the defined field of view of the robot, and to guarantee that, it is necessary to take into account the vertical opening of the Kinect camera.

Figure 4.8 represents the scheme used for calculation of the point at the top extreme of the field of

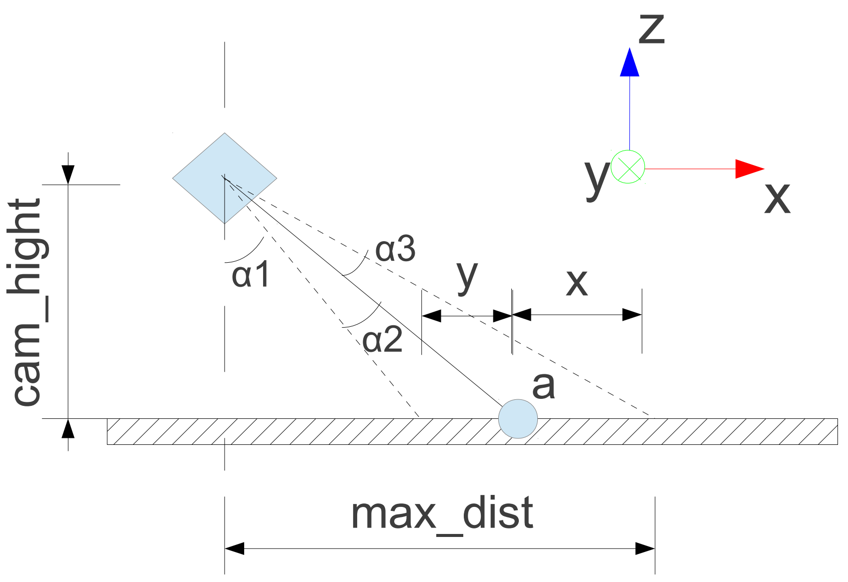


Figure 4.8: Scheme for Calculation of the Top and Bottom Points for Defining the Windows

view. In Figure 4.8, *a* represents the point to calculate and *cam_height* the Kinect's height relatively to the floor. The *max_dist* is the maximum distance configured for the field of view. To calculate point *a* one can calculate *x* and subtract it to the maximum distance for the field of view. For that, one has to consider both schemes represented in Figure 4.7 and 4.8 which will originate the equations represented in 4.3 through 4.5.

$$\begin{cases} \alpha_1 = 90^o - tilt - \frac{kinect\_vertical\_open}{2} \\ \alpha_2 = 90^o - tilt \\ \alpha_3 = 90^o - tilt + \frac{kinect\_vertical\_open}{2} \end{cases} \tag{4.3}$$

The equations represented in 4.3 define the angles necessary for the calculus of the intermediary variables.

$$\begin{cases} \tan(\alpha_3) = \frac{max\_dist}{cam\_height} \\ \tan(\alpha_2) = \frac{max\_dist - x}{cam\_height} \\ \alpha_2 = \alpha_3 - \frac{kinect\_vertival\_open}{2} \end{cases} \tag{4.4}$$

As for the equations represented in 4.4 allow to define the angles recurring only to known variables. This allows to have a single equation for calculating the distance *x* where this is the only dependable variable which is represented in 4.5.

$$x = max\_dist - cam\_height \times \tan(\arctan(\frac{max\_dist}{cam\_height}) - \frac{kinect\_vertical\_open}{2}) \tag{4.5}$$

The top point of the window can be calculated by subtracting to the height of the window the distance x converted into pixel coordinates, obtained using the map resolution. To calculate the bottom point the procedure is similar, but in this case, considering the *y* distance and the minimum distance allowed, which originates the equation represented in 4.6.

$$\begin{aligned} y = cam\_height \times \tan(\arctan(\frac{min\_dist \times \cos(\frac{opening}{2})}{cam\_height}) + \\ + \frac{kinect\_vertical\_open}{2}) - min\_dist \times \cos(\frac{opening}{2}) \end{aligned} \tag{4.6}$$

After having the top and bottom points, the algorithm creates points spaced by two pixels from top to bottom in the vertical center line of the field of view. These points will be used as reference for calculating multiple pan and tilt poses which represent windows to where the robot may point the Kinect at.

For each point created vertically in the field of view, the left and right extreme points are computed so that the windows, to be defined by these points, fit inside the field of view defined for the robot. In order to calculate the left and right extreme points of the field of view, one has to consider the scheme represented in Figure 4.9 where w represents the distance to the right and z to the left counting from the center point of the Kinect's field of view. The Kinect has a constant inclination in its horizontal axis when changing the pan angle, resulting in an equal opening of the field of view for the left and right extreme points.

Considering the schemes represented on Figure 4.9 one can define the equations represented in

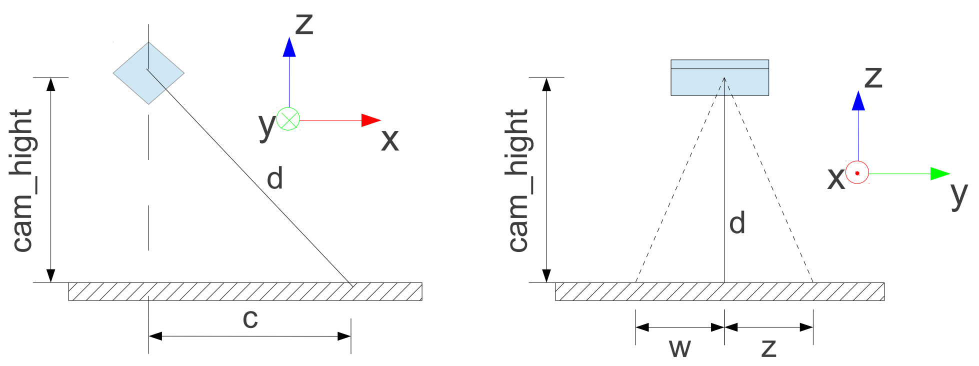


Figure 4.9: Scheme for Calculation of the Right and Left Points for Defining the Windows

4.7 to calculate the right and left distances from the center point of the window to the extremes so that the widow fits inside the robot's field of view.

$$\begin{cases} d = \sqrt{cam\_height^2 + c^2} \\ w = z = d \times \tan\left(\frac{kinect\_horizontal\_open}{2}\right) \end{cases}$$
$$\Rightarrow w = z = \sqrt{cam\_height^2 + c^2} \times \tan\left(\frac{kinect\_horizontal\_open}{2}\right) \tag{4.7}$$

With the aforementioned procedures and equations, it is possible to define points on the field of view of the robot from top to bottom and left to right. These points respect the maximum and minimum distances allowed so that the captured image is inside the defined boundaries for the robot's field of view.

The field of view, at this point, contains several points that will define a set of pan and tilt poses, which in turn, will define possible windows that the robot may acquire. The described situation is represented in Figure 4.10 where are shown the multiple points calculated inside the robot's field of view. It is an image created by the algorithm itself, where the window is the field of view of the robot, and the points will define pan and tilt poses. The extreme top and bottom points, represented in Figure 4.10, are not at the same distance of the respective extremes of the field of view. That result is expected since as far as the Kinect is pointed at the largest will be its field of view projected on the floor.

Having all the points defined, it is necessary to calculate a pan and tilt for each point. Since the points are well defined in x and y coordinates, and using the map resolution to convert between pixel and spatial coordinates, the expressions for the calculus of pan and tilt become as shown in equations

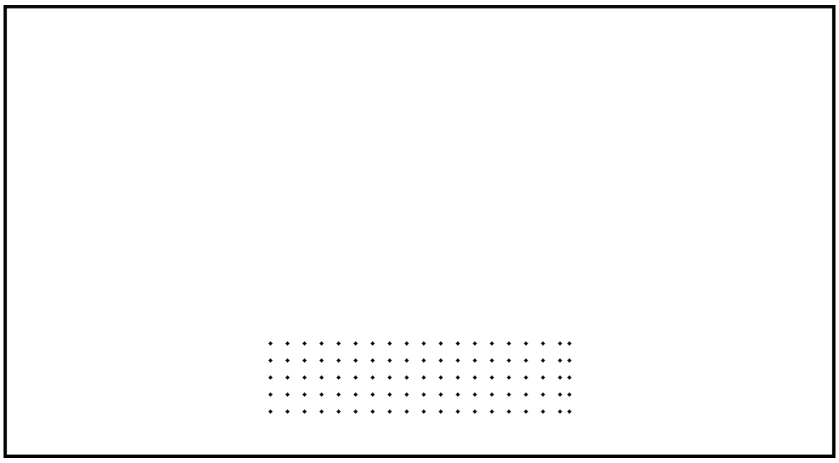


Figure 4.10: Field of View and all Possible Poses Within

4.8.

$$\begin{cases} pan = \arctan\left(\frac{y}{x}\right) \\ tilt = 90^o - \arctan\left(\frac{\sqrt{x^2+y^2}}{cam\_height}\right) \end{cases} \tag{4.8}$$

Calculating the pan and tilt poses, it is possible to calculate the surrounding window for that specific point using a similar procedure as the one used to calculate the top, bottom, right and left extreme points. Considering the schemes represented in Figures 4.8 and 4.9, that the point (x, y) and the pan and tilt pose are available, it is possible to calulate the projection of the Kinect's field of view on the floor using the equations represented in 4.9.

$$\begin{cases} x = cam\_height \times \tan\left(90^o - tilt + kinect\_vertical\_open\right) - \sqrt{x^2+y^2} \\ y = \sqrt{x^2+y^2} - cam\_height \times \tan\left(90^o - tilt - \frac{kinect\_vertical\_open}{2}\right) \\ w = z = \left(\sqrt{cam\_height^2 + x^2 + y^2}\right) \times \tan\left(\frac{kinect\_horizontal\_open}{2}\right) \end{cases} \tag{4.9}$$

With the calculation of x, y, w and z, variables from Figures 4.8 and 4.9, it is possible to associate a window size for all the calculated pan and tilt poses.

The center points computed earlier do not represent the center of the windows calculated due to its projection on the floor. To facilitate the transformation of the windows into the map according to the robot's pose, the computed points are centered in the calculated windows. The centering of the points into the calculated windows originate a new arrangement for the points which is represented in Figure 4.11. Has it can be seen, the displacement of the points is consistent to the theoretical expectation. The projection of the field of view of the Kinect on the ground plan is larger with the

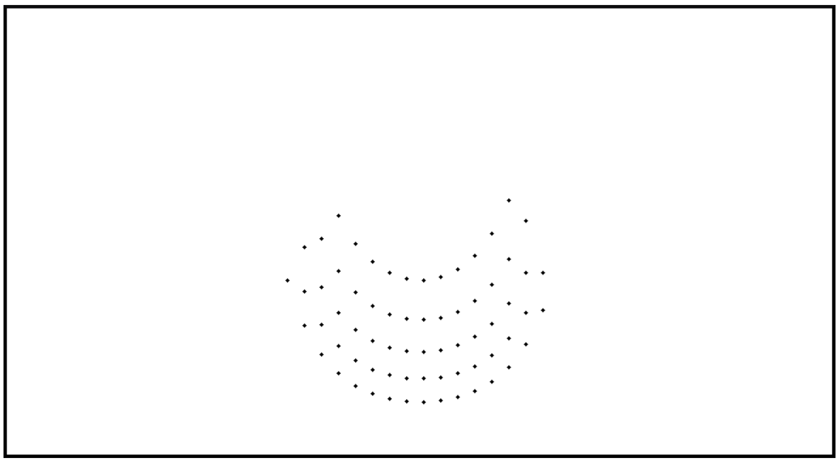
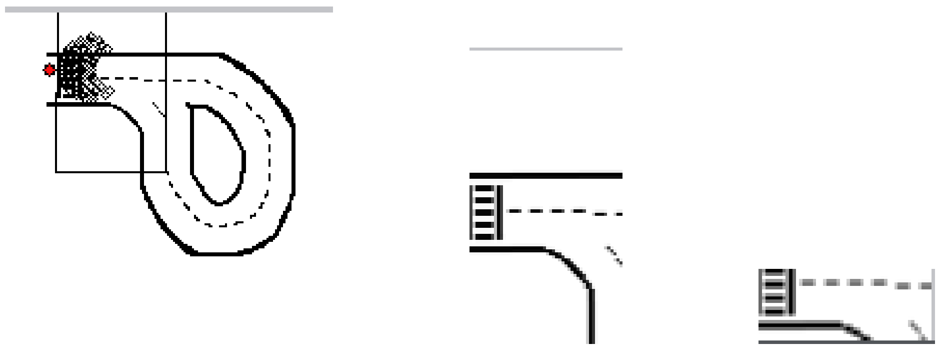


Figure 4.11: Center Points for the Pre Calculated Windows

distance. Also, the Figure 4.11 has less points than the Figure 4.10. Due to the approximation done to the field of view of the robot, there are generated windows that go outside the defined field of view and so, these are excluded.

Everything that is needed to compute the best window, and so the pose that best serves the localization algorithm, is calculated at this point and it is necessary to transform the window's pose to the map according to the robot's pose. This is performed at each processing cycle to evaluate where to point the Kinect. The transformation is performed using a translation followed by a rotation operation. After the point within the map is calculated, according to the robot's pose, the respective window is extracted from the map.

In Figures 4.12(a) through 4.12(c) are shown the results of the aforementioned algorithm. In this



(a) Map and Field of View

(b) Field of View Cropped from Map

(c) Window with Most Weight Cropped

Figure 4.12: Results from the Selection of the Camera Pose

document were already shown some maps of the test track, and in the image shown in Figure 4.12(a) the map is inverted. The reference of the map is changed so that the it can be accessed in pixel coordinates since the $(0, 0)$ of an image is on the top left corner and in the real map the reference is on the bottom left corner. In Figure 4.12(a) is shown the loaded map, the red point represents the robot's pose in it and the rectangle the field of view possible within the map according to the robot's pose. Inside the filed of view, draw on the map, are draw the several points representing possible pan and tilt poses. Figure 4.12(b) shows the robot's field of view extracted from the map, which is used for further processing. Using the extracted field of view from the map and the several windows, originated by the pre computed pan and tilt poses, are processed in order to evaluate the best pan and tilt pose in which results an expected area of the map acquired by the Kinect image. This expected area, resulting from the best pan and tilt pose computed, is shown in Figure 4.12(c), the process for selecting the best pan and tilt pose is explained hereafter.

Using the several pan and tilt poses it is possible to extract several expected map areas from the map. With the extracted areas from the map it is possible to calculate the amount of information on that area. However, the importance of a given area, and the respective pan and tilt pose, is not calculated using only the information it contains, but also the required movement to apply the respective pose. Since the localization algorithm requires as much information as possible, it is important to reduce the movement of the camera as much as possible, because while the camera moves, the acquired information is not usable. As a measure to reduce the movement of the camera, it is considered the movement that has to be performed to apply a pose. To measure the importance of a window, it is added to the importance, or weight, the information inside of it. It is also subtracted to the weight of a window an expression containing the movement, in this way, the weight increases with the information and decreases with the movement necessary to set it. In Figure 4.12(c) it is possible to see the result of the selection, and it is in fact the area with most information on the map that is possible to point the Kinect within the defined field of view for the robot.

The perception of the world can not be limited only to the track. The environment in which the robot should be able to navigate contains obstacles, being these, boxes obstructing a lane or a roadwork area delimited by cones. For that reason one should account for obstacles not on the map.

The algorithm used for detection of the obstacles is somehow similar to the one used for detecting the lines on the floor, but the sensor is different and the depth image is used. It is not made a distinction between the detection of cones or other obstacles since the navigation approach taken will handle both situations in the same manner. This happens because the tape connecting the cones is also detected which will generate a continuous obstacle on the track and the robot will plan its course accounting for it.

To the detect obstacles the depth image is filtered first so that only obstacles closer than a pre defined distance are considered. While performing this filtering, and since the Kinect is pointed at the ground, the image row that corresponds to the user pre defined distance is found and stored.

In this filtered image a sensor from the top to the bottom is defined, being the bottom limited by the previously defined row that represents the threshold distance. Then, with the sensor defined, a differential operation, similar to the one performed in the detection of delimiting line points, is performed. Since the edges of the obstacle will present a step in the depth image, the limits of the obstacle are detected by this procedure. Also, the line which represents the threshold distance is analysed to detect lateral points of the obstacle and its geometry. The Figure 4.13 represents the

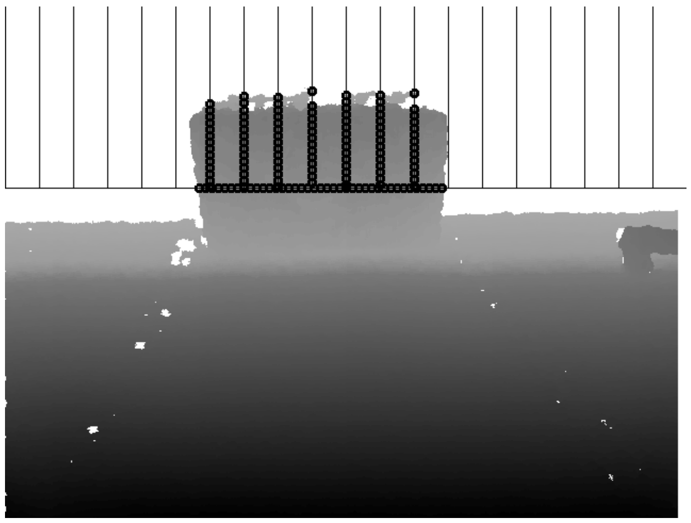


Figure 4.13: Detection of the Obstacle Edges Using a Vertical Sensor

detection of the edges of the obstacle in the already filtered image. The detected points in the depth image are then validated using the knowledge that one has about the possible obstacles. These can only be rectangular boxes or cones with defined measurements.

After detecting and validating the obstacles it is necessary to publish this information. The publishing of the points representing the obstacle should be done using either laser scan data or point cloud data and, as it is performed in the delimiting line points detection, the points detected in the depth image are transformed into a point cloud.

To represent the obstacle properly in the cost map, to be accounted for path planning, it is created a blob using the detected points and its geometry. Namely, to the points already detected, are added points with a shape of a circumference, a blob, using the estimated center of the obstacle and its size. As it was said, the size is measured using the bottom line of the sensor. With the size of the obstacle it is possible to determine if it is a box or a cone and create a blob accordingly.

4.4 Signs Perception

The robot should be able to detect traffic light signs (semaphore) and vertical signs since it is an objective of the Autonomous Driving Challenge. Since the only vision sensor available in the robot is mounted over a pan and tilt structure the first obvious problem to solve is on how to actually capture an image from the sign in order to identify it.

The robot pose may be used to compute a camera pose for identifying traffic light signs since the position of the traffic light sign (TFT), in which are shown the traffic light signs, is fixed. Having a fixed pose for the traffic light sign as represented in Figure 4.14 one may calculate the pan and tilt pose for the camera in order to capture an image containing it.

It is possible to calculate the angle α between the robots pose and the traffic light sign pose. The

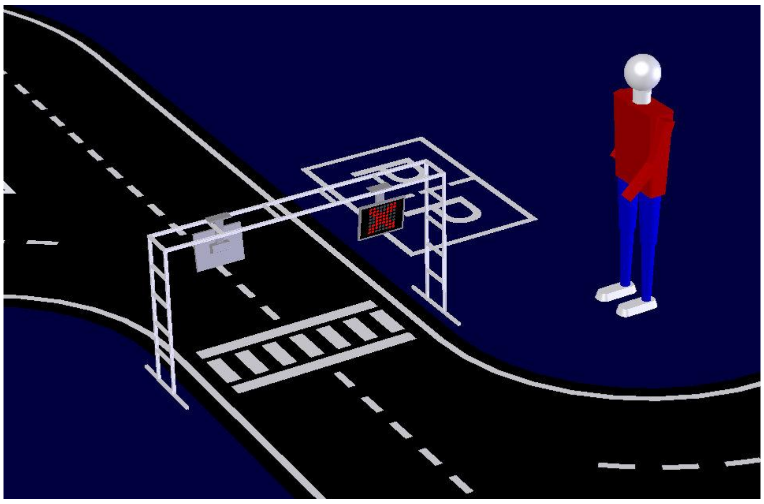


Figure 4.14: Representation of the Traffic Light Sign Pose on the Track

calculation of this angle is done using the expression shown in 4.10.

$$\alpha = \arctan\left(\frac{x_{robot} - x_{semaphore}}{y_{robot} - y_{semaphore}}\right) \tag{4.10}$$

With α calculated one has to take into consideration the heading of the robot to obtain the pan pose which guarantees that the Kinect is pointing towards the traffic light sign (equation 4.11).

$$pan = -robot\_heading - \alpha \tag{4.11}$$

For the calculation of the tilt pose one has to consider the height of the traffic light sign relatively to the vision sensor and the distance to the traffic light sign resulting on a expression for the tilt angle represented in 4.12.

$$tilt = \arctan\left(\frac{semaphore\_height - camera\_height}{\sqrt{(x_{robot} - x_{semaphore})^2 + (y_{robot} - y_{semaphore})^2}}\right) \tag{4.12}$$

With both pan and tilt calculated, and considering an accurate localization information, it is assured that using the aforementioned procedure calculating the pan and tilt pose the traffic light sign will be in the captured image.

The selection of a camera pose to identify the vertical signs is more complex since the position of these is not known in advance. It is necessary to detect the sign first in order to compute a proper pose for the pan and tilt. To detect the sign it is used the depth image that is being acquired while the robot is navigating[13]. The base of the vertical sign has a specific signature in the depth image and this fact is used for detecting the base of the sign selecting then a pose for identification accordingly.

For identifying the base of the vertical sign in the depth image it is used template matching. In

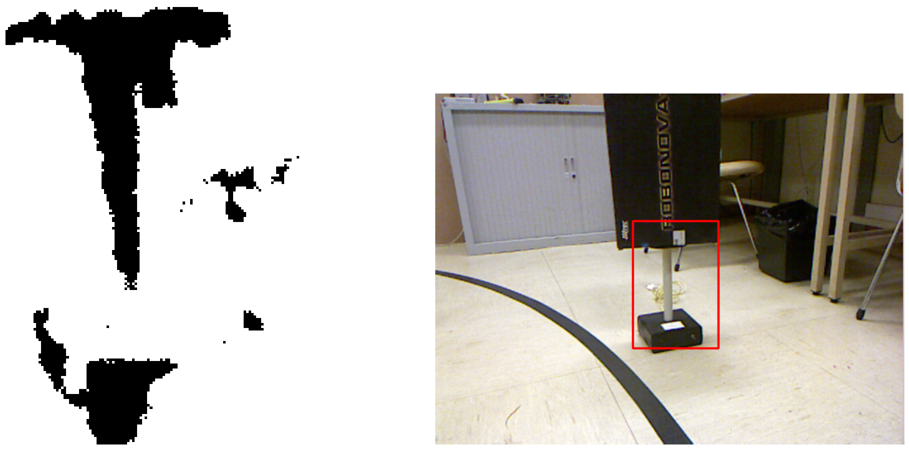

(a) Base of Vertical Sign Depth Template (b) Base of Sign Identified

Figure 4.15: Vertical Sign Identification[13]

Figure 4.15(a) is shown the template of the base a test vertical sign used. The template shown is binary so that the reader may understand the signature of the vertical sign's base in the depth image. In Figure 4.15(b) is shown, in the RGB image, the area where the base of the vertical sign was found using template matching in the depth image. The pan angle for identifying the sign itself is defined by the position of the base, being the tilt angle defined by the height of the sign.

After selecting a proper pose for the pan and tilt, guaranteeing that the actual sign is captured by the Kinect, it is possible to attempt an identifications. The identification of vertical and traffic light signs is very similar and the first step is common to both. To improve the identification, it is taken advantage from the fact that the sign is represented in a plan. This is used in the first step of the identification to extract the sign from the scene.

A two sided threshold is applied to the depth image considering an estimation of the distance at which the sign is. In Figures 4.16(a) and 4.16(b) is shown the depth image before and after applying the threshold. This operation already eliminates much of the scene and noise, and it is a preparation

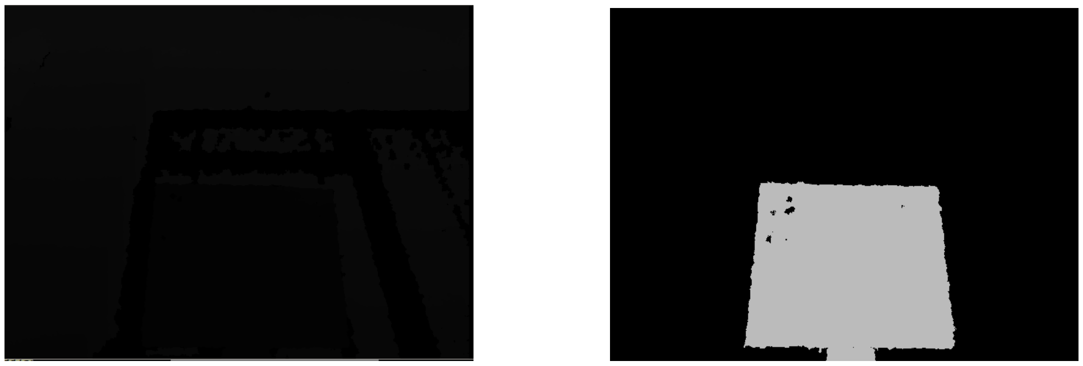

(a) Depth Image without the Threshold Applied (b) Depth Image after Threshold Applied

Figure 4.16: Demonstration of the Application of the Threshold to the Depth Image

for the next step.

In the depth image in which was applied the threshold, it is applied a region growing method to identify only the area of the sign. The pixel for starting the region growing method is the closest one in the depth image. This selection for the first point comes from the assumption that the Kinect is correctly pointing at the sign's plan. The region growing method, assuming that the Kinect

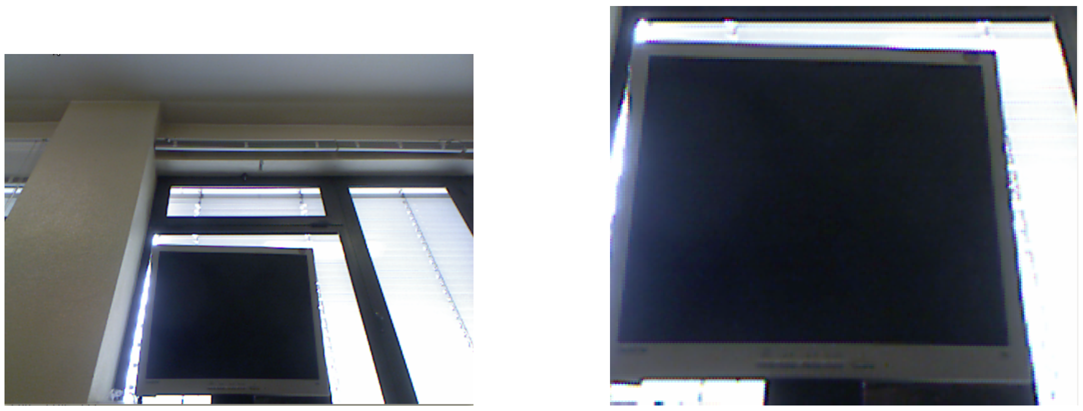

(a) Original RGB Image Captured Containing the Traffic Light Sign (b) Extracted Sign from Scene in RGB Image

Figure 4.17: Isolated Sign

has a correct pose, identifies the region of the sign's plan. Since the depth and RGB images are registered, it is possible to correspond the extracted area from the depth image to the RGB image. The result of the extraction of the sign's plan from the scene is represented in Figure 4.17(a) and 4.17(b). With the extracted sign from the scene it is possible to attempt a more accurate identification.

For the identification of the traffic light signs it is used a robust matcher[25] algorithm. The algorithm is not aimed for image matching as it is being used in this application. The algorithm is aimed for matching two images taken from two different points of view aiming stereo vision. However, the key point is the same, find matches in two similar images, and since this is a very robust algorithm for matching, it is applicable to this particular case.

The algorithm for matching sums up to the following steps[25]:

- Feature detection and extraction of the two images to compare;
- Match the nearest neighbours (at least two) from image one to image two and from image two to image one;
- Evaluate matches for each feature and if the ratio between the best match and worst match is greater that a given threshold eliminate the worst match, this is done until the ratio between the best match and the worst is lower than the defined threshold;
- Eliminate matches that are not symmetrical between image one and image two;
- Validate surviving matches with RANdom SAmple Consensus (RANSAC) eliminating outliers.

The last step of the algorithm was proven to be sometimes destructive in the matching process and so it may not be applied. It is destructive for this particular application since it is aimed for eliminating the points that do not obey to the epipolar constraint[25] like it would be expected in a stereo vision system. However it is not possible to guarantee the alignment of the template and the captured image, making the last step of the algorithm, in this case, eliminate good matches. Besides the algorithm extreme robustness, it is adaptable and configurable. It even allows to choose the feature detection algorithm as well as the extrator one.

To choose the sign that is being captured by the image acquired it are evaluated the features from the template, the features from the captured image and the surviving matches. Based on the ratio between features and surviving matches a score of each template is calculated. The identification and scoring process may be performed more than once in different acquired images. The score is accumulated to each template and the sign is selected based on it.

Due to the importance of the identification of the traffic light signs, the identified sign is published along with a belief. This belief is computed from the scores of all templates and the one with the higher score. This belief allows to the navigation agent, who will be using this information, evaluate if the identification is accurate enough to proceed or if it should request another identification in order to raise the belief in the result.

The identification of the vertical signs is not so critical as the identification of the traffic light signs. Another aspect of the vertical signs is that these are less ambiguous and easier to distinguish from each others. Also, a vertical sign identification does not need a belief associated since it will not be used by the navigation agent, the identification should only be informed, somehow, to the user.

For the aforementioned reasons, the matching algorithm used in the vertical sign is less robust than the one used for identifying traffic signs. The matching algorithm, is the application of the first three steps of the robust matcher explained earlier. The score of a vertical sign template is computed as it is for the traffic light signs. However, the identification process is only performed once and the template with the highest score is shown on the screen of the laptop.

4.5 Pan and Tilt Agent

So far in the vision module it was possible to see that there are multiple components of the vision system requiring different poses for a single vision sensor. For that reason, the system requires an agent to handle different requests so that no component prevents others from having vision information.

The agent for the pan and tilt structure is the only agent that sets the camera pose. It receives requests from the various modules requesting vision and based on the requests it sets a pose.

The agent has a waiting queue where it stores the requests. It also has a priority list of all modules that request vision. The fixed priority of the module, requesting a camera pose, is merged with a dynamic priority that is sent along with the vision request. This dynamic priority is meant to represent how much the module wants to see[14]. For example, the road perception module, changes this priority, that sends with the vision request, according to the belief in the localization. If the belief is high then this dynamic priority is low allowing other modules to see, if not, then the dynamic priority sent is high.

The dynamic priority allows to conciliate the signs identification with the navigation. For example, when the robot approaches the crosswalk the Kinect should be pointed at the traffic light sign to identify it while moving so that if the sign is not the stop sign the robot continues its march. However, the pose for the sign is set only if the localization has a high belief allowing to navigate a short period of time just with the dead reckoning information. If the belief in the localization is low, then the robot will approach the crosswalk and stops and only then will attempt an identification, since when the car is stopped the belief on the localization is no longer relevant for navigation.

As for the requests, besides the pose and the identification of who is requesting, they contain a tolerance for the pan and tilt angles. With this tolerance a pan and tilt window is defined that is used to conciliate, if possible, multiple requests on the waiting queue[14]. At first is selected the request in the waiting queue with the higher priority, and then the queue is searched for unattended requests which window intercepts the already generated one. The pose for the pan and tilt is then adjusted to fit both windows that are coincident but favoring the most priority request. This situation is represented in Figure 4.18. The red circle represents the pose requested with the highest priority, the green rectangle

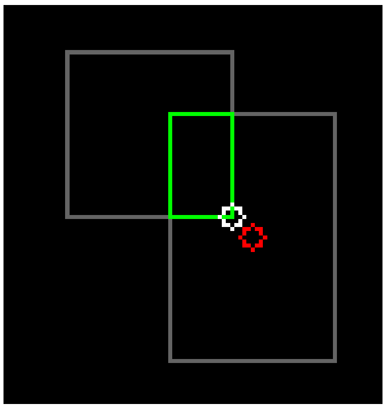


Figure 4.18: Merging two Vision Requests[13]

the window resulting from merging the two gray windows, representing two different requests, and the white circle the resulting pose, which favours the request with higher priority attending also other request. This means that the pose will be in the limit of the resulting window but closer to the request with the highest priority. When a request is attended it remains in the waiting queue but with a flag representing that is already attended. When a pose is set, confirmed by the pan and tilt feedback, the algorithm choses another request and the intercepting requests to attend.

The algorithm for the pan and tilt agent sums as follows:

- receive and store requests in the waiting queue;
- chose unattended request with the higher priority;
- define request window with the defined tolerance;
- find the second most priority request that fits the window;
- recalculate the pan and tilt pose fitting the interception window favoring the request with higher priority;
- redefine the widow, being this the interception of the two previous windows;
- repeat the three last steps until no more requests in the queue fit the window.

The agent of the pan and tilt as described in [14] was altered so it could be applied to this dissertation case. Namely it was added the option to cancel requests, so these would not stay in the waiting queue even if not valid anymore by the requesting module.

4.6 Localization Module

As objective of this dissertation, the robot should localize itself on the track using for that the Kinect as main sensor. For a mobile robot to localize it self it should account for its odometry sensory data and for its perception of the environment. Again, since ROS is being used, one should start by evaluating the available possibilities. As it was said in Chapter 2, particle filters have been a very popular and efficient method for solving the localization problem in mobile robots, and ROS has a particle filter implementation applied for localization available.

The algorithm used for localization is a probabilistic particle filter which implements the MCL approach with KLD-Sampling [39] since this is implemented over ROS in the *amcl*[2] package from the *navigation* stack and presents many advantages regarding other methods [39]. This algorithm was already explained in Chapter 2.

The *amcl* node tracks the pose of the robot against a known map. To properly work, the input data is, besides the map, perception data, in the form of laser scan data, and odometry data. The filter publishes the estimate pose and the set of pose estimates maintained by the filter. There is a number of parameters that can be used to configure the operation of the node, which are related to the filter, the laser model and the odometry model.

After analyzing the subscribed topics and the parameters of the *amcl* node there are two main aspects important for the use of the node that should be addressed: the laser model and the odometry model. The amcl node only supports laser scan data, which needs to be generated from the Kinect camera images. This data is generated in the vision module from a point cloud obtained from the cameras images. Also, the *amcl* node can be adapted to support point cloud data as input perception data. This adaption was performed (Chapter 5.4.2) however, it was not fully tested and it is not fully functional yet.

The *amcl* node, as it is available in ROS, only supports two odometry models, the differential and the omnidirectional, whereas the vehicle has an Ackerman steering system. The use of a different motion model in the particle filter adds error at every motion step, since the particles will move mistakenly, reducing thus the probability of having a particle representing the real robot's pose. The Ackerman motion model may be added to the *amcl* node by adding another model to the odometry sensor implemented by the amcl.

In order to the particles motion get updated respecting the Ackerman motion model, the random error can only be added to the covered distance by the traction wheel and to the steering servo which are the direct odometry measurements. The addition of random error only to the two mentioned components originate a scattering of the particles coherent with the Ackerman motion model. In Figure 4.19 the arrows represent the particles, and its disposition represents the expected scattering of the particles using the Ackerman motion model. The implementation of this alteration is inherent

[2]www.ros.org/wiki/amcl

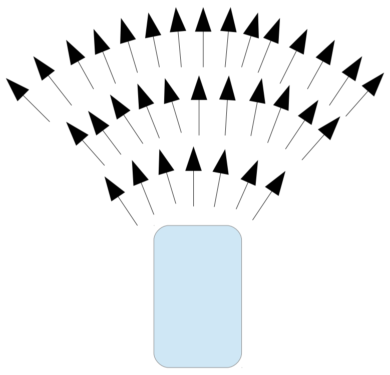


Figure 4.19: Scattering of the Particles with the Ackerman Motion Model

to the organization of the *amcl* package, and how it internally functions. For details on this alteration please refer to Chapter 5.4.1.

As for the output of the *amcl* it publishes the transformation from the map frame to the odometry frame allowing to have the transformation from the map frame to the robot's frame. The algorithm also publishes a topic with the localization, however, with a slow rate, which makes this topic only useful for debugging purposes. The pose, published in the topic, has however a very useful information. The pose is published with a covariance matrix which indicates the covariance of the particles in (x, y, θ). This covariance matrix is used by the navigation module to enhance the driving safety.

In sum the localization module is an adaption of the *amcl* package provided by ROS. The adaption was performed so that the algorithm would support the Ackerman motion model and point cloud as input perception data.

4.7 Navigation Module

The navigation module implements all the software related to the navigation. This means that this module contains the software for deliberative driving, path planning and motion planning. The main idea of the navigation module is to use the localization as starting point for autonomously drive the robot through the track. Using localization one can define a starting point and an end point on the map, compute a plan between the two points and follow the plan using control.

First of all, and since it is an objective to use ROS, its navigation[3] support is used as basis. ROS navigation support is implemented on the *move_base*[4] package. The *move_base* package completely implements autonomous navigation having internal software for navigation control and behaviour definition. It has as input the robot's state, localization and world perception.

[3]`www.ros.org/wiki/navigation`
[4]`www.ros.org/wiki/move_base`

In Figure 4.20 is shown the architecture of the software when using the *move_base* package.

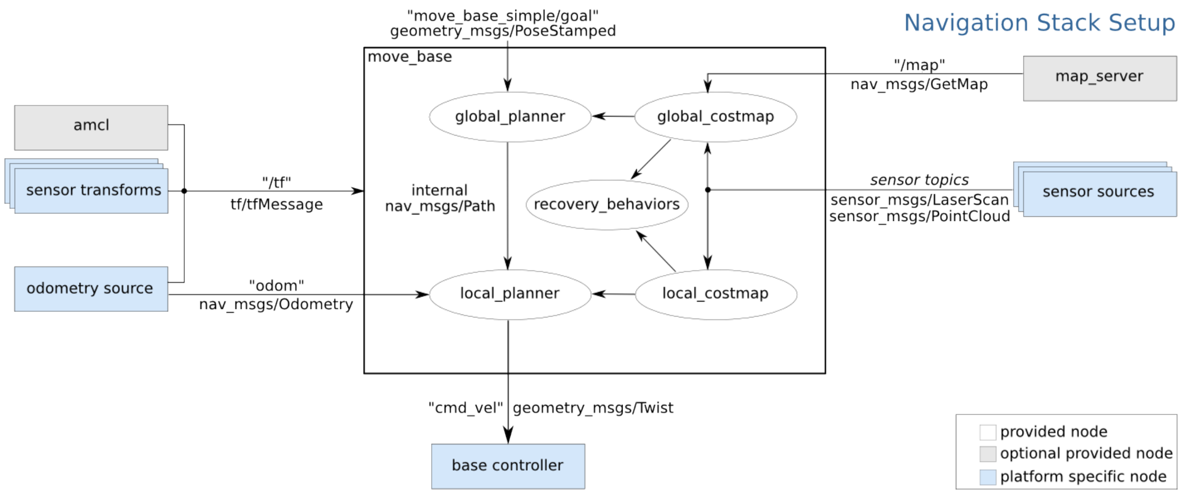


Figure 4.20: Overview of the *move_base* Package Functioning[16]

It is possible to see through the architecture, that one can publish a goal pose, and with properly published information of localization, robot's state and world perception, the *move_base* package will publish action commands to follow a planed path. It allows to define a global planner and a local planner for the path planning and even allows to set recovery behaviours for flaws.

However, the *move_base* package only supports omnidirectional and differential motion models in the local planning. This means that this can not be used in this particular case for the car like robot. Without the proper motion model, the computed plan may not be feasible by a car like robot, and even if it is, the action commands can not be applied.

However, this package is used because it publishes the local and global cost maps which allows to have the perception of the world within ROS environment. Also it is used for global planning since it is possible to define a global planner compatible with a car like robot and use it, through a service, to request a global plan without the computation of action commands and local planning.

The global planner used, and that is supported by the *move_base* package, is the SBP planner. This planner is not defined for any kind of locomotion, instead, it is possible to define a set of motion primitives. The algorithm will use the motion primitives to build a path containing only movements feasible by the robot. This allows to have any kind of locomotion system with the definition of motion primitives to use in the process of finding a path. For details on how these motion primitives are created for ROS implementation of SBP, please refer to Chapter 5.5.

Using a search based algorithm, a smooth and feasible path is produced. This means that no processing or smoothing is needed after planning. The produced path is discrete and represented by a set of points that can be used for computing the action commands. Using the *move_base* package as main core for the navigation, one has to create software for handling the path and produce the action commands accordingly.

The objective of the software created to complement the *move_base*, is to substitute the local planner of the *move_base* package since it does not support car like robots. The idea is to handle requests for driving to a given point described by a pose and a velocity profile. It should compute the path to that goal, using the service available in the *move_base* package, and then apply steering and velocity control to reach the goal at the given velocity profile. Also, this part of the navigation module, should use the local cost map available through the *move_base* package to account for obstacles not described in the map, and recompute a path that deviates from the detected obstacle. The state of each goal should be published as well, so that the main agent has feedback on how the goal is being pursued, if it is possible to pursue it, etc . . .

The steering control is performed based on the cross track error[40]. The cross track error is measured between a point in the front bumper of the car to the closest point of the computed path. The measure of the error is performed in relation to a point in front of the robot, and not on the center of the steering as defined in [40], because it was experienced that the path following is a lot more stable using a point in front of the car.

The scheme represented in Figure 4.21 is aimed for calculation of the expression for the steering

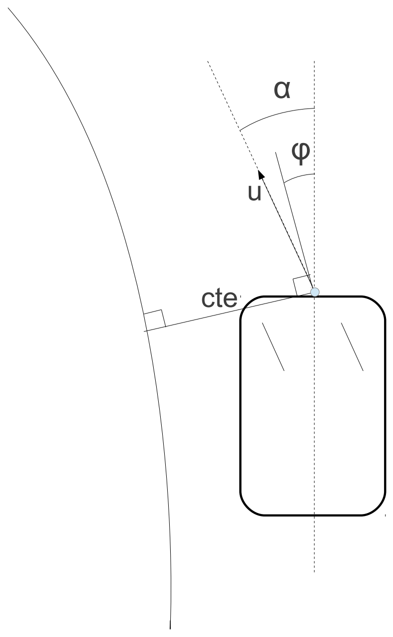


Figure 4.21: Representation Scheme for Cross Track Error Calculation[40]

angle that allows to guide the robot towards the path. With the represented scheme one can have for the steering ($\alpha(t)$) the following expression[40] represented in 4.13.

$$\alpha(t) = k_\phi \times \phi(t) + \arctan\left(\frac{k_{cte} \times cte(t)}{u(t)}\right) \tag{4.13}$$

α is the steering angle to apply, ϕ the heading error, cte the cross track error and the $u(t)$ the velocity. Also, the k_{cte} is a constant to define the importance of the cross track error in the expression and the k_{phi} for defining the importance of the heading error, added to have a easily tunable controller. As for the heading error, this is measured between the robot's heading and the next point in the path.

Since the path is discrete, the path is approximated to a series of lines, making possible to measure a perpendicular error to the path. To avoid oscillations in the control due to localization noise, the approximation to a line is not performed point by point, but with a spacing of a pre defined number of points. This line is shifted point by point as the robot drives through the path as it is shown in Figure 4.22.

To calculate the cross track error, instead of calculating the perpendicular intersection between

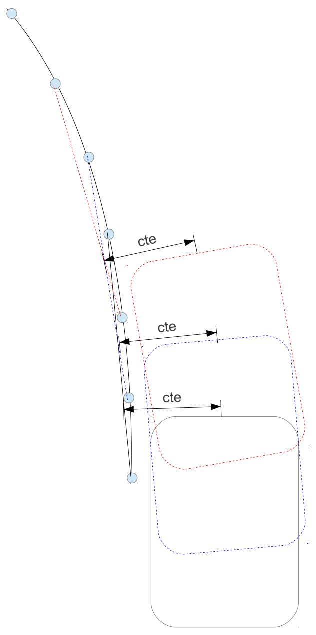


Figure 4.22: Scheme Representing the Reference Line for Cross Track Error Calculation

the reference line and the front point of the robot, all the points considered are transformed to the origin of the coordinate system as is shown in Figure 4.23. The transformation of all the points involved in the cross track error calculation, as shown in Figure 4.22 is accomplished by applying to all points a translation of $p1$, which is the first point of the reference line, and applying the rotation of $p2$, the last point of the reference line.

With the aforementioned transformations, the cross track error is the y coordinate of the

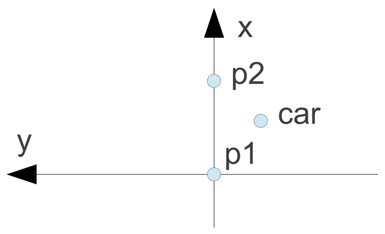


Figure 4.23: Points Transformed for Simpler Cross Track Error Calculation

transformed car point and to evaluate if the reference line has to be shifted in the path, it is only necessary to compare the x coordinate between the transformed car point and the point for which the reference line has to be shifted. Then, using the calculated cross track error, one can calculate the steer action commands with the already mentioned expression.

The same exact algorithm is used for backwards movement, but instead of using the point in front of the car, it is considered a point in the back of a car. Also, for safety reasons, the velocity is reduced.

Please refer to Chapter 5.5.3 for further details on the implementation of the local planner.

To implement navigation it is also necessary to create the main deliberative agent for producing the goals to drive to, following an approach on driving point by point. This agent also has to handle the approach to the crosswalk to set in motion the necessary mesures, namely, the identification of the traffic light sign.

The main agent will handle publishing goals accordingly to their state, that is published by the implemented local planner. This enables a goal by goal driving approach where the main agent chooses which goal to publish next when another point is reached or cancelled. This approach allows to define a driving profile by choosing a list of points to follow on the track accordingly. For example, if one wishes to drive favoring the lap time, then the points should not consider the lanes and should try to cut the distance on the curves.

The navigation agent has very well defined objectives and the navigation is defined by a set of goal poses. For that reason, the main agent for navigation is implemented as a state machine where the tasks are well defined and the driving goals are defined by the user. The states for the navigation agent are directly linked to the goal by goal driving method and with the Autonomous Driving Challenge objectives. For more details on the navigation agent and its implementation please refer to Chapter 5.5.4.

The chosen implementation requires communication between the navigation agent and the local planner implemented. This communication is based on the one provided by the *move_base* package itself, but it is adapted for this particular case, the car like robot and the goal by goal driving approach.

The communication, from the navigation agent to the implemented local planner, carries the goals in which the robot should pass along with the velocity it should have when passing it. From the local planner to the navigation agent, the communication carries the state of each goal published. This allows the navigation agent to know when it should publish the next goal. It also allows to the navigation agent to know if a goal can not be pursued, if it was cancelled due to impossibility of being reached, if the path to it was replanned due do obstacles, among others. Also, this communication carries the action commands to be applied, so that the main navigation agent may publish them or not according to the situation.

4.8 Mapping Module

In order to have a correct navigation, based on a localization approach, the map should be accurate. Even if a plan is followed to build the physical environment, in this case the track, this will contain irregularities due to assembly faults. Also, a map built by hand do not account for what the robot sees, it instead describes exactly what the robot should see. For that reason it was decided to implement mapping capabilities to the robot.

With the option of performing mapping, one should choose how to implement it. Since it is an objective to use ROS, one should start by evaluating what is available in it. Considering the robot and the environment there are some options available in ROS for performing SLAM.

Since a Kinect is being used, one of the algorithms available in ROS is the RGBDSLAM, which is an algorithm for performing mapping with a hand held Kinect like camera[12]. As explained in Chapter 2 this algorithm constructs a dense point cloud map registering the collected data set, which is perfectly fit for the Kinect. The produced map offers some problems for localization algorithms and the computational effort, due to the dense map, is too high for a robot that only needs 2D localization. The aforementioned reasons make the RGBDSLAM suitable for performing 3D models of objects or small indoor environments.

One of the 2D SLAM approaches that ROS offers is the *hector_slam*[24], which provides a SLAM algorithm for unstructured environments. However, this requires a good pose estimation, and since it uses perfect matching for building the map, it can fail completely. This algorithm can also perform mapping using dead reckoning from inertial sources only, which is the primary functionality of this algorithm. This algorithm was tested in the robot but did not presented satisfactory results due to the environment, in which the perfect matching algorithm fails.

For the implementation of the mapping module the GMapping algorithm is used since it is available in ROS and it is described as the state of the art SLAM algorithm. As it was explained in Chapter 2 the GMapping algorithm uses a Rao-Blackwellized particle filter as main core in which each particle carries an individual map. The GMapping algorithm is very efficient and the ROS implementation is very adaptable, making easy to apply it.

The ROS implementation of the GMapping algorithm subscribes laser scan data and dead reckoning state of the robot. The dead reckoning state is published by the base module, based on the odometry data and steering servo position, received from the robot base and published through the tf. The laser scan data is generated and published by the vision module, based on a point cloud obtained from the Kinect image data. Thus, the ROS *slam_gmapping* node, the implementation of GMapping in ROS, can be used without modifications.

The mapping operation is performed off line using previously recorded data. This allows repeatability for testing the GMapping parameters. Also, the mapping process is computationally heavy and it should be performed off line to guarantee a proper functioning of the algorithm. For details on the mapping operation, please refer to Chapter 5.6.

Chapter 5

Implementation Details and Results

This chapter is a complementation of the Chapter 4, where all the details of the implementation are described. In this chapter the approaches described in Chapter 4 are implemented, describing the necessary steps of implementation to make the earlier described approaches and options possible.

Also, in this chapter, the modules overviews given in Chapter 4 are linked to the implemented ROS software, making possible to understand the links between them.

The software, as standard in ROS, is organized into packages that form a stack called *rotapkgs*. The *rotapkgs* stack is organized into four main packages. The name of this packages is self explanatory about its content and each one of the packages mentioned in this chapter is directly related with the modules explained in Chapter 4 or are its direct software implementation.

Since the robot has two main functions, are these, mapping and navigation, there are different configurations for each task. The configurations have a lot in common in which concerns the nodes, topics and packages used. In Figures 5.2 and 5.1 are shown overviews of the two configurations of the software in a form of a graph. The boxes represent the packages, and the arrows represent their connections. The connections between packages are done through topics or services that support the message passing between the packages and their nodes. In Figure 5.1 is represented the package arrangement needed for navigation and in Figure 5.2 for mapping.

The choices and options mentioned in Chapter 4 are implemented in this chapter over ROS. In the following sections shown some details of the previous explained software as well as details inherent to the ROS implementation.

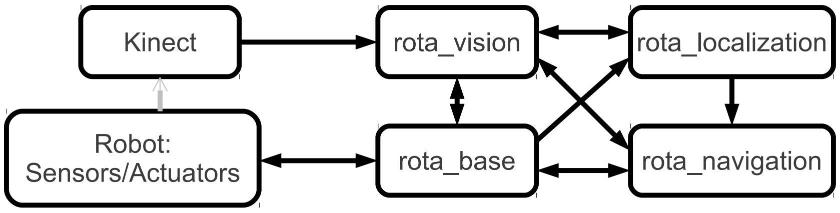


Figure 5.1: Package Overview for Navigation Process

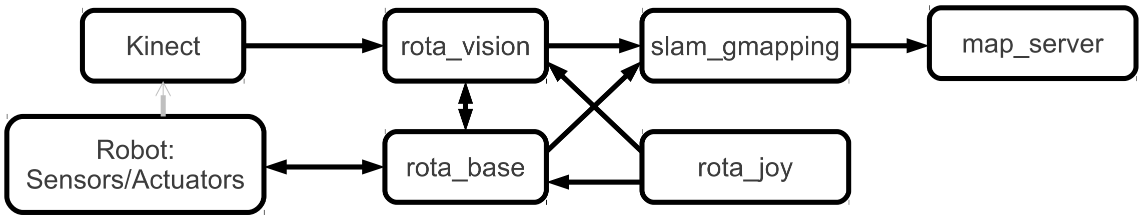


Figure 5.2: Package Overview for Mapping Process

5.1 Base Module

The implementation of the base module is the *rota_base* package. The needs of this package are defined in Chapter 4.1 and it is necessary to take the approach into practice developing the software to support it. The necessary software is divided into three main parts, the configuration software, the software for communication with the robot and the software for publishing the robot's state.

5.1.1 Configuration and Communication Set Up

Part of the robot's description is done through configuration files. The configuration files define a set of constants that make possible the translation of sensor values into values with meaning. In these constants are defined among others the wheel diameter, the encoder resolution, the gear relation, etc . . . The configuration files defining the constants were written aiming for the use of the *libconfig++* [1] library. Other configuration files needed are files containing the lookup tables, the conversion functions, for the steering, pan and tilt servos.

The software for handling the communication and the configuration files was inherited from previous software versions and was altered to support new hardware components like the new steering servo and the pan and tilt structure for the Kinect and then it was adapted to work over ROS. The software is divided into three C++ classes that are then used in a ROS node. The three classes are the *rtCarModel*, the *rtCommChannel* and the *rtConfig*.

The *rtCarModel* class makes use of the *rtConfig* class to load the configuration files and the lookup tables for the servos, providing functions to translate low level values into meaningful values. The

[1]`www.hyperrealm.com/libconfig`

rtCommChannel class provides the support to send and receive frames for and from the hardware. The support for the communication is given through functions to transform a given order into a valid message for the hardware and functions to send messages to the hardware. The inverse support is also given, there are functions to receive and to translate the messages from the hardware into processable values that are then ready to be processed by the *rtCarModel* class to be transformed into meaningful values. These configuration files and classes not only support the communication between the robot and the computer, but also the robot's model for high-level software layer.

5.1.2 Physical Robot Description

To have the state of the robot within ROS environment, it is necessary to first define the robot's model. The robot's model will define the joints of the robot, the coordinate system for each frame and the performed movement for each joint. For simulation purposes, the physical model may also contain inertial values, weights and other relevant constants that model the system and make possible a realistic simulation. In Figure 5.3 is shown the appearance of the physical model made for the Zinguer robot.

The physical model is extremely important in the software implemented over ROS since this

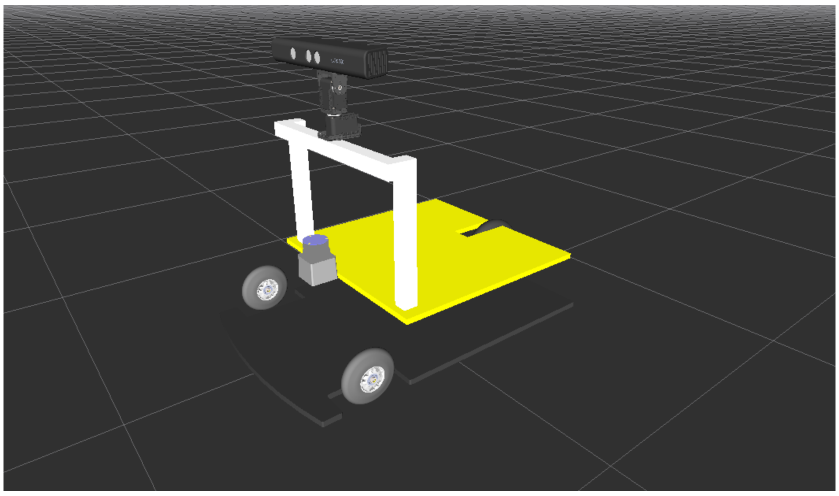


Figure 5.3: Physical Model Appearance for Zinguer Robot

model is used to create the tf tree and to make the transformations between coordinate systems along the tf tree of the robot. It is also very useful for debugging with ROS tools. The displacement of the sensors has to be as accurate as possible, otherwise, the transformations between coordinate systems will introduce error. Please refer to Appendix B.1.1 were is shown the tf tree generated by the physical model created for the Zinguer robot. In the tree are shown all the frames of the robot and also its connections.

In ROS the physical model of the robot is build using the URDF specification and the language used is eXtensible Markup Language (XML). In the code for the URDF model is possible to identify the various frames that are represented in the tf tree and how their connected by the means of joints. The origin for the coordinate system for each frame was measured from the robot since this already existed.

5.1.3 Robot State Publisher

To have an accurate and consistent tf tree it is better to have a single node publishing all the transformations at once. Other necessary tf like the \map $\rightarrow$ \odom, representing the transformation from the map frame to the odometry frame will be published by the localization node. ROS has a specific package available for publishing all the tf associated to the robot's state, the *robot_state_publisher* [2], which is integrated in the *robot_model* stack. The *robot_state_publisher* package contains only one node that is the *state_publisher*. This node is easy to use and has very few input parameters. It has as input the URDF XML robot's description file, a tf prefix, to build the tf tree with a name space prefix, and the desired publish frequency for the transformations. The *state_publisher* node subscribes to a topic where the joint states should be published using the *sensor_msgs/JointState*[3] message type.

Since the *state_publisher* needs the joint states to be published, another node, also called *state_publisher* was created in the *rota_base* package to acomodante this. For simplicity reasons, this new node is also responsible for publishing the tf \odom $\rightarrow$ \base_link, representing the dead reckoning, and thus keeping the approach on centralize all the tf.

In Figure 5.4 is represented the structure for the software responsible for the robot state publishing. Analysing Figures 5.6 and 5.4 it is possible to note a link between the software for communication with the robot and the software for publishing the robot's state, */hwcomm/baseKinematicData*. Links like this will be appearing in the unroll of this chapter, making possible to understand the connections between packages represented in Figures 5.2 and 5.1. The created node takes the feedback from the car base, sent trough the topic *hwcomm/baseKinematicData*, and publishes this information in a form of a joint state. The created *state_publisher* node besides publishing the joint states also publishes directly the tf \odom $\rightarrow$ \base_link, and since the tf is published in a form of a displacement between origins, this tf is the odometry displacement itself. The initial values for the dead reckoning are given through input parameters of the node, initial_x, initial_y and initial_theta. To calculate the dead reckoning of the robot, since it is a car like robot, the bicycle model is used.

In Figure 5.5 is represented a displacement of the robot, where d is the distance covered by the back wheel, L the distance between axis, R the radius of the circumference described by the robot's

[2] `www.ros.org/wiki/robot_state_publisher`
[3] `www.ros.org/doc/api/sensor_msgs/html/msg/JointState.html`

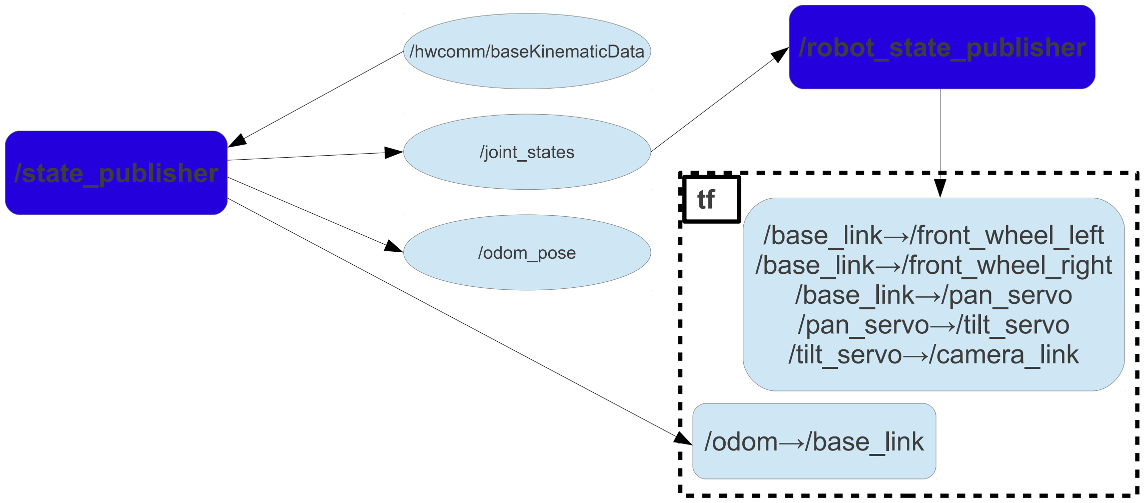


Figure 5.4: Software Architecture for Robot State Publisher

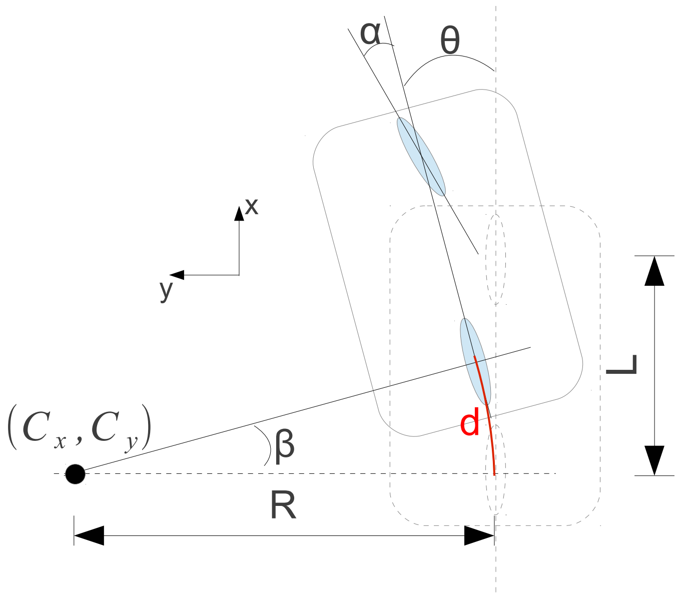


Figure 5.5: Representative Scheme of a Displacement of the car

movement with (C_x, C_y) as its center, β the angle of the described arc, α the steering angle and $\Delta\theta$ the heading displacement. The pose of the car is defined in the 2D space with x, y and θ. Having defined all the variables, the objective is to calculate the new x, y and θ after a displacement where the only known variables of the scheme represented in the Figure 5.5 are d, α and L. To calculate the new x, y and θ one can start by defining that $\beta = \frac{d}{L} \times \tan\alpha$ and with β one can define that $R = \frac{d}{\beta}$. Considering the previous pose as x_{old}, y_{old} and θ_{old} and the new pose as x_{new} ,y_{new} and θ_{new}, the center of the circumference described by the movement of the robot can be calculated with the equations shown in 5.1.

$$\begin{cases} C_x = x_{old} - \sin(\theta_{old}) \times R \\ C_y = y_{old} + \cos(\theta_{old}) \times R \end{cases} \Rightarrow \begin{cases} x_{new} = C_x + \sin(\theta_{old} + \beta) \times R \\ y_{new} = C_y - \cos(\theta_{old} + \beta) \times R \end{cases} \tag{5.1}$$

To obtain θ_{new} it is necessary to add the heading displacement to θ_{old}, and since $\Delta\theta = \beta$, $\theta_{new} = (\theta_{old} + \beta) \pmod{2\pi}$. Since $\beta = \frac{d}{L} \times \tan\alpha$ if d or α are zero, β becomes zero invalidating the calculus of R. However, if $|\beta|$ is small, let small be less than 0.001, the update on the robot's pose may be approximated to the equations shown in 5.2.

$$\begin{cases} x_{new} = x_{old} + d \times \cos(\theta_{old}) \\ y_{new} = y_{old} + d \times \sin(\theta_{old}) \\ \theta_{new} = (\theta + \beta) \pmod{2\pi} \end{cases} \tag{5.2}$$

Having calculated the dead reckoning data of the robot it is needed to create a tf structure to hold it, input the values for x, y and θ, the parent and child frame names, in this case the parent frame is *odom* and the child frame *base_link*, include the time stamp and then broadcast it. With this the dead reckoning data is available for all nodes that may need it, and all the transformations associated with the robot are published.

5.1.4 Communication Software

The communication with the car, on the ROS software point of view, has to be a topic or a set of topics in which to publish orders to be applied and where the state of the car and sensor values can be obtained. To solve the ROS interface problem it was created a node called *cb*, which stands for *car base*, since this node will the basis of all software for the car. The car base node is responsible for receiving orders from the upper levels software, translating them into hardware values and send those values to the hardware using serial communication through an USB port from the computer.

In Figure 5.6 is shown the software architecture for the communication between the robot and the high-level ROS software. The scheme represented in Figure 5.6 represents the node *cb* and the topics that it subscribes and publishes. Topics are represented with elipses, letting the arrows indicating if they are published or subscribed, and nodes are represented with boxes. All the topics associated with the car base node begin with the name space */hwcomm* to indicate that it is a topic aimed for hardware communication. The name of the topic after the name space is given according to its function.

baseActuationData The function of this topic is to receive messages for hardware actuation, namely the car related one. The message type associated to this topic includes velocity commands, steering

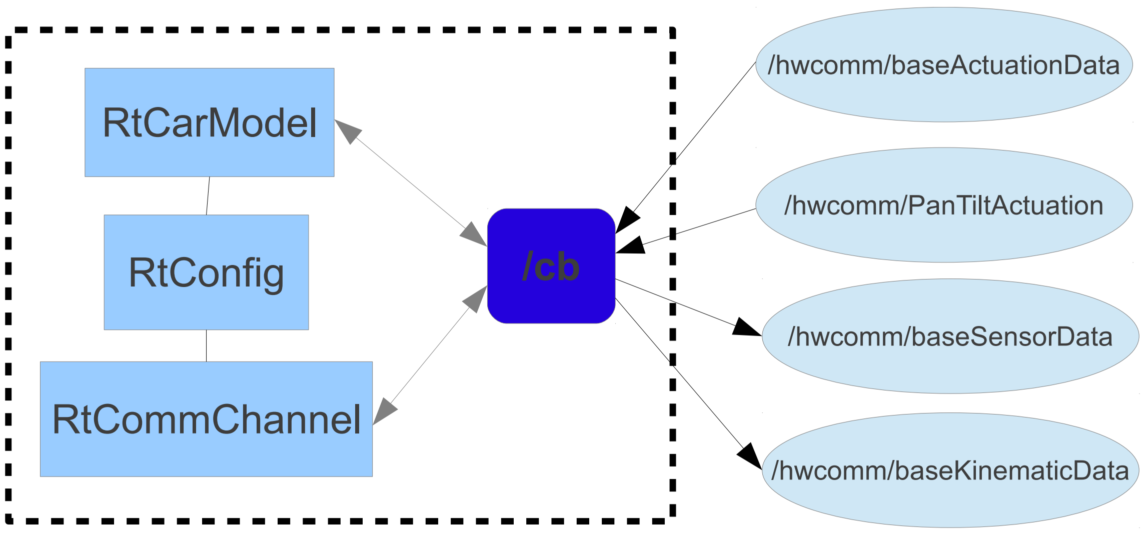


Figure 5.6: Software Architecture for Computer/Robot Communication

commands and light commands. Velocity and light commands have its own message type while the steering commands are divided into to floats, the steer angle and the velocity for the movement. The message associated with the velocity command contains two floats, one for linear velocity and other for angular velocity, having thus the possibility to apply a linear velocity and separated steer commands, or the possibility to apply an angular velocity that basically will be traduced into a velocity for the traction and a steering angle. As for the lights commands there is an array of the light type message in the *baseActuationData* message. This array has as purpose to send more that a light order at once since the car has more that one light. The light type message has two char type fields, are these to indicate the identification of the light and the order, where the order can be for blink, for turn on or turn off.

PanTiltActuation From the beginning of the implementation of the software the pan and tilt and the driving orders were meant to be sent by different agents. In order to guarantee the modularity and the separation of the orders there were created two different topics, one for driving actions and other for pan and tilt actions. The *PanTiltActuation* message type has four float type fields, are these for the pan angle, pan movement velocity, tilt angle and tilt movement velocity.

baseSensorData The *baseSensorData* topic is published by the *cb* node, giving to other nodes the information of the sensors. In the *baseSensorData* type message is included the information of the ground sensors, the obstacle sensors and the state of the batteries. Each sensor has its own message type which means that the *baseSensorData* is composed by other messages. The *GroundSensors* message type used inside the message *baseSensorData* has two variables of the type bool, one for the left sensor, and other for the right sensor. The *ObstacleSensors* message is organized in the same way, but instead of two bool fields it has two floats, indicating the distances measured by the left and right obstacle sensors. The *BatteryCharge* message type appears two times in the *baseSensorData* message, one for the battery of the control circuits, and other for the battery of the power circuits. These message contains two floats, one for the current charge of the battery and other for its nominal charge.

baseKinematicData The *baseKinematicData* topic and its associated message type are one of the most important parts of the *rota_base* package messages. This topic is published by the *cb* node to give information of the kinematic data of the robot, it provides odometry updates, and the feedback of the servos. For the odometry data the *baseKinematicData* message has five float fields, the curvature, the steering angle, the partial distance covered by the traction wheel, the total distance covered by the traction wheel and the velocity. As for the joint states, the *baseKinematicData* message has three fields, the pan angle, the tilt angle and the steer angle.

5.2 Remote Control Module

The *rota_joy* package is a package aimed for interfacing the robot with gaming remote controllers. ROS already provides a package of joystick drivers, *joystick_drivers*[4], making possible to have abstraction from the hardware and the communication with it. The *joy_node*, available in the package, is a ROS generic driver for joysticks that can be supported by Linux. The *joy_node* takes as input parameter the device to do the interface (eg /dev/input/js0), and publishes the input information of the joystick through a topic, making it available to any node that subscribes it. The topic published by the *joystick* node uses the *sensor_msgs/Joy* message type. The *sensor_msgs/Joy* type message has two arrays, one float array for the axis values and an int array for the buttons values. Having the input information from the joystick already published in a topic it is necessary to translate this information into commands.

The *rota_joy* package is divided into two main parts, the configuration files and the software for interfacing the joystick commands with the robot it self.

5.2.1 Configuration Files

The configuration part of the software uses the libconfig++ library. It has a set of files for mapping the keys of the joystick, one file for each joystick type, and the file contains the positions in the respective array, for buttons and for axis, in the ROS provided message. To build the configuration

[4]www.ros.org/wiki/joystick_drivers

files, which contain the mapping information, it is necessary to know which key corresponds to each position in the arrays of the *sensor_msgs/Joy* message, using the *joystick_drivers*. It is possible to know the configuration of the joystick recurring only to ROS software. To see which key corresponds to which array position one can use four commands. First launch the ROS environment in a terminal, then in another terminal set the device if the default is not applicable, eg

```
$ rosparam set joy_node/dev "/dev/input/jsX"
```

then run the joy_node with

```
$ rosrun joy joy_node
```

and after the launching, in another terminal use the rostopic tool to look into the topic and see which positions of the array change with which key

```
$ rostopic echo joy
```

After this process, and having in mind which array position corresponds to which key, it is just necessary to add another configuration file with the mapping information.

5.2.2 *rota_joy* Software

The developed software for the package *rota_joy* consists into two nodes, one for controlling the robot with the joystick and the other was built specifically to calibrate the steering table, that as was mentioned in Chapter 4 is the table used to convert a steering angle into a servo set point.

Driving the Robot This node was created to control the robot with the joystick and it is called *joy_com*. It loads the configuration file, indicated by the input parameter, and maps the keys for the functions according to the configuration file. In Figure 5.7 is shown the configuration of the keys, and their functions, for driving purposes. From the Figure 5.7 it is possible to understand what each button does at the exception of the *dead man* button. The *dead man* button exists for security purposes, if this button is not pressed, the commands are not assumed, and the velocity is set to zero. The velocity and steering control are done in a manner that the position of the joystick is directly linked to the position of the servo, and the velocity.

To link the value of an axis to an action, one has to consider the joystick response. The response of the joystick is not linear. It varies between zero and one with a form of a sigmoid function. To accommodate the sigmoid, the value of the joystick is translated to velocity using three different gains, approximating thus the sigmoid to three lines. The same is applied to the steering commands.

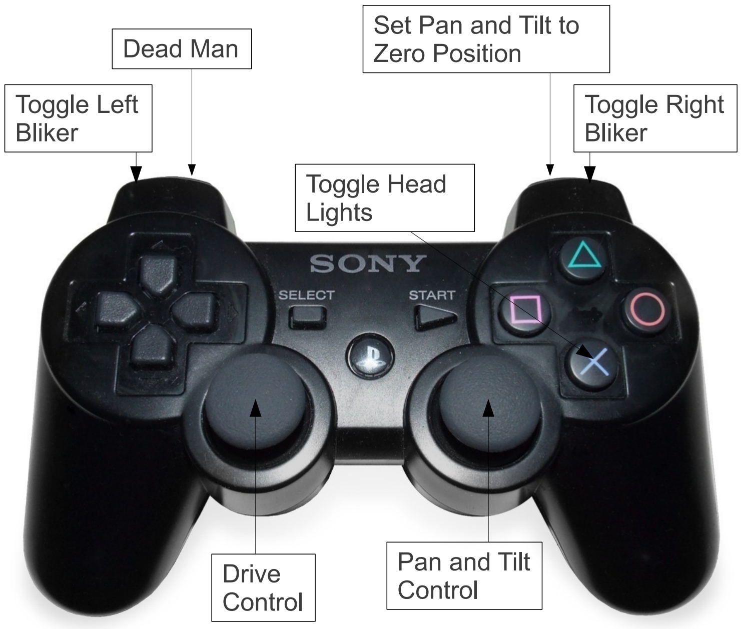


Figure 5.7: Mapping of Joystick Keys for Driving Purposes

The pan and tilt controls are accumulative, which means that the joystick is integrative and adds to the position of the pan and the tilt. This is for the user to be able to set a position for the pan and tilt and then drive without having the concern of holding the joystick to maintain a pan and tilt position. That is also the reason for having a button to set the pan and tilt to zero.

Having the node description in mind the overview for the system, when this node is launched, is shown in Figure 5.8. The *Request_Vision* topic appears in the Figure 5.8 because it is possible

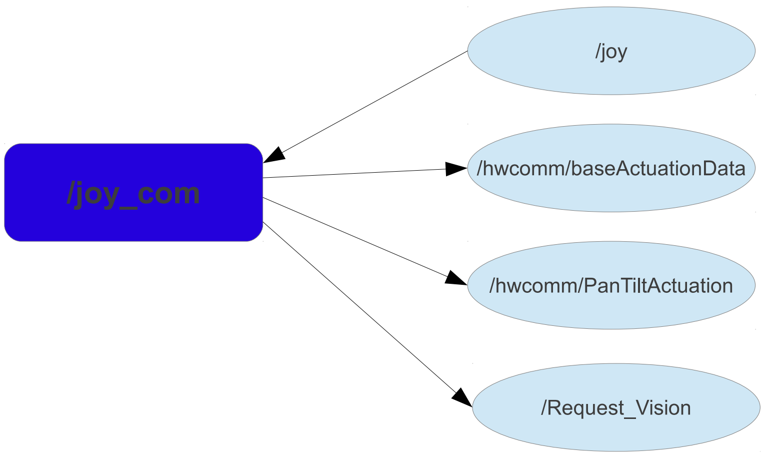


Figure 5.8: Software Architecture for Driving Purposes

to control the pan and tilt structure directly, sending the commands to the *cb* node through the topic

/hwcomm/PanTiltActuation, or through the vision agent, publishing a request for a pan and tilt pose through the topic *Request_Vision*. The configuration on how to control the pan and tilt structure is set through parameters of the *joy_com* node.

Building the Steering Map Table As for the *steering_table_gen* node, the one created to build the steering table, it has a well defined function, and so the commands given from the joystick are different and well defined as well. The main ideia of this node is to have an easy way of building the mapping table from a steer angle, into servo set point. In order to do that, it is not enough to have the model of the steering servo, it is necessary to measure, for a given order, which is the steering angle.

From the calculus of the odometry, in Section 5.1.3, and assuming the same names for the variables, $\beta = \frac{d}{R}$ and also $\beta = \frac{d}{L} \times \tan(\alpha)$, with this, one can make $\alpha = \arctan(\beta \times \frac{L}{d})$, which simplified becomes $\alpha = \arctan(\frac{L}{R})$. With the aforementioned information it is possible to define a set point for the steering servo and measure the radius of the described circumference with a given set point making thus possible to calculate α.

To measure the radius of the described circumference by the robot, without introducing to much error, the simpler way is to use the odometry system of the robot, namely the measure of the covered distance by the back wheel. To do that one can mark a departing point for the robot, define a set point for the steering servo and describe a complete circumference making the robot stop on the departing point. In this case, the distance covered by the back wheel of the robot is exactly the perimeter of the circumference described, which means that one can obtain the radius directly with $R = \frac{d}{2\pi}$.

With the described approach for measuring α for a steer set point the use the joystick for interfacing with this node is the logical implementation. In Figure 5.9 is shown the mapping of the

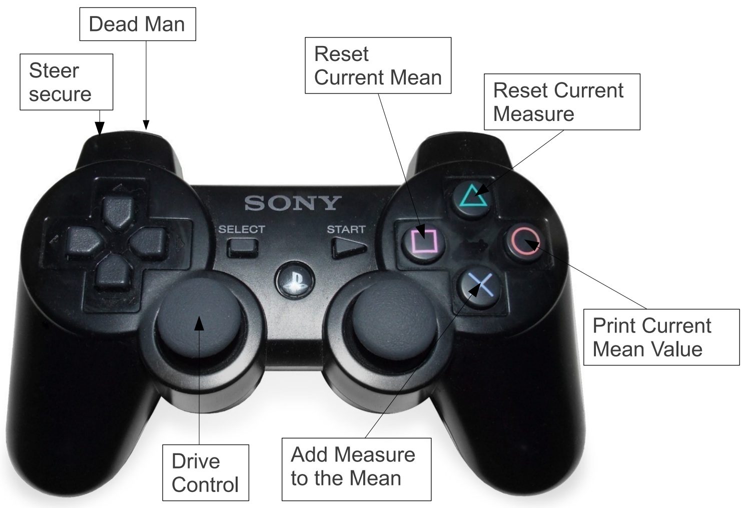


Figure 5.9: Mapping of the Joystick Keys to use the *steering_table_gen*

keys of the joystick for interfacing with the *steering_table_gen* node. The *Drive Control* joystick is for driving the robot, but in this case, when the *Steer Secure* button is pressed, the velocity commands are not assumed but only steer commands. When the *Steer Secure* button is not pressed, the steer is blocked in the last set point, and only velocity commands are assumed. The steering set point is printed in the terminal while is being adjusted.

The node accumulates measures of perimeters into a mean every time the *Add Measure to the Mean* button is pressed. This is meant for reducing the error on the observations. Once that are enough measures, one can press the *Print Current Mean Value* to obtain the resulting α for the set point at the moment. Also, reset buttons where added to correct measures or reinitialize the measuring process.

With the described process one can make measures off strategic set points and build a table for every set point available by the hardware using interpolation or linear regression depending on the results. The acquired table of values for the Zinguer robot using the aforementioned method, as well as the graph for the regression, are shown in Appendix B.2.1.

As for the software architecture, in the case of using this node, is more or less the same as using the *joy_com* node, with the exception of the topics related to pan and tilt control, since there is no need for those when using this node.

5.3 Vision Module

The *rota_vision* package, as its name indicates, is the package that olds all the software related to the vision software system. This package not only includes the image processing software, but also software for controlling the pan and tilt and compute according poses.

This package, is the direct implementation of the vision module approached in Chapter 4.

5.3.1 Road Perception Module

In this section is detailed the road perception module described in Chapter 4. It is detailed the software implementation as well as how it reflects over ROS.

***vision_sensor* node** Since the creation of a sensor over the image may be an heavy operation and it can be calculated only once and every time the parameters of the sensor change, a node to create the sensor was developed. The *vision_sensor* node is responsible for calculating the image sensor that will be used to process the image for detecting the lines of the track. The input of this node is a set of parameters for the sensor, and the output a set of image coordinates that define the sensor.

Listing 5.1: Message for Defining the Sensor Parameters

```
int32[2] picsize
int32[2] center
float32 alpha
int32 radius
float32 opening
int32 radius_min
```

The Listing 5.1 represents the message *rota_vision/sensor_params*. This message is used to define the sensor generated by the *vision_sensor* node. The parameters for the sensor are defined in image coordinates and parameters. The *alpha* defines the spacing in radians between lines of the sensor, the *opening* the opening in radians of the sensor, the *radius* defines the radius of the sensor in pixels and the *radius_min* defines where the lines of the sensor begin in relation to the defined center by the *center* parameter defined in pixels with x and y coordinates.

To calculate the sensor, since it is radial, one can calculate the end point for each line using the alpha parameter and the radius, and then calculate the line function $y = m \times x + b$ in which to substitute x to obtain y, defining thus the points belonging to the sensor. There are precautions in calculating the points for each sensor line. Because the points are in pixel coordinates, and so they are discrete, it is necessary to take into account the angle of the line in order to have always adjacent pixels in the point array. To guarantee that, one needs to consider the angle of the sensor line and instead of using $y = m \times x + b$, use the equation $x = \frac{y-b}{m}$ for the cases of the lines with angles minor that $\frac{3\pi}{4}$ and for angles greater than $frac\pi 4$. Another particular case to consider is when the angle of the line sensor is $\frac{\pi}{2}$ and it is not possible no have a function for the line, the x is fixed, and the y varies from the center to center plus the radius.

After generation of the sensor with the received parameters, the node publishes a message containing the definition of the sensor.

Listing 5.2: Message Defining the Vision Sensor

```
int32[] points
int32[] xpoints
int32[] ypoints
int32 nlines
int32[] npoints
rota_vision/sensor_param gen_params
```

The Listing 5.2 represents the message *rota_vision/sensor_points* and it contains an array *points* for direct access to the image pixels and two other arrays *xpoints* and *ypoints* for accessing the image pixels through x and y coordinates. The message also contains the *nlines* variable, indicating the number of lines in the sensor, and the array *npoints*, for defining the number of point in each sensor line. Also, the returned message by the *vision_sensor* node contains the *rota_vision/sensor_params* message to inform which parameters generated that sensor.

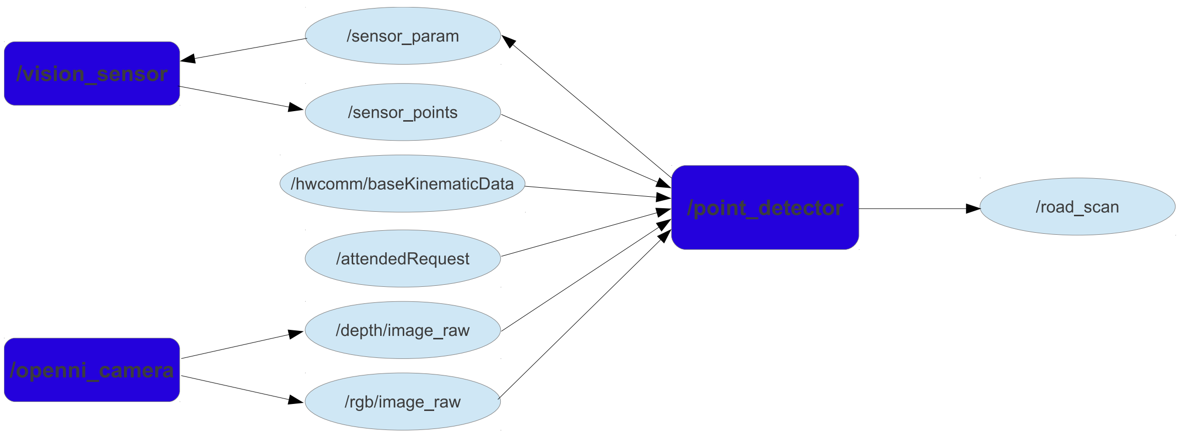
***point_detector* node** The point detector node is the direct implementation of the algorithm described in 4 for detecting the delimiting line points of the track. It subscribes the RGB and depth image published by the Kinect and outputs a laser scan or a point cloud, depending on definable parameters, containing the detected points to be used by the Mapping and Localization modules. It also subscribes the topic related to the pan and tilt agent, the *attendedRequest* topic, in order to know when to use the information captured by the Kinect camera.

***camera_pose_selector* node** The *camera_pose_selector* node is the node responsible for computing a pose for the Kinect camera that favours the detection of the lines of the track at any given moment. It subscribes the map, the pan and tilt feedback as well as the robot's pose. It publishes the computed pose for the Kinect to the pan and tilt agent in order to be applied. The pose is published with a frequency two times lower than the rest of the system to avoid to much oscillation on the camera pose.

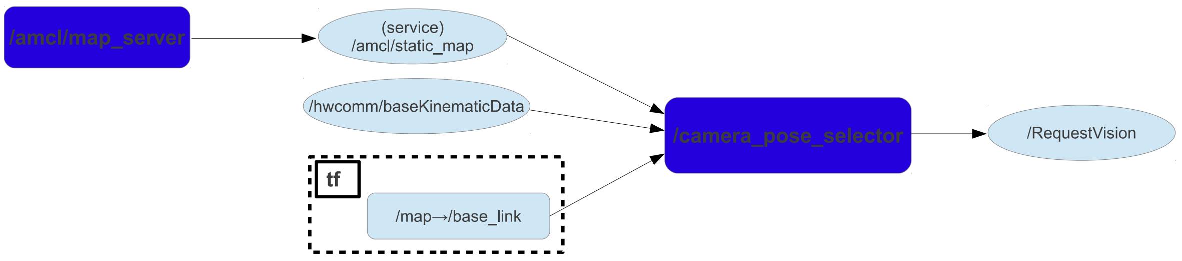
***obstacle_sensor* node** The *obstacle_sensor* node is the direct software implementation of the explained approach for obstacle detection. This node subscribes to the depth and RGB image to produce a point cloud containing the obstacles detected. It also subscribes to the topic *attendedRequest* so that obstacle information is only produced when the Kinect is pointed at the ground.

With the nodes for the vision aimed for localization and mapping all defined, one can now define a software architecture for this part of the *rota_vision* package. The Figures 5.10(a) and 5.10(b) show the software system architecture for the software responsible for the vision system aimed for localization purposes. For mapping purposes the system is the same with the exception with the *camera_pose_selector* node since the Kinect pose is controlled by the user using the joystick. In Figure 5.10(a) is shown the architecture for the software responsible for generating data to the localization algorithm, in this case the *road_scan* topic which may contain point cloud data or laser scan data. In Figure 5.10(b) is shown the architecture for the software responsible for selecting a camera pose, pan and tilt, in order to favour the *point_detector_node* on data collection. In Figures 5.10(a) and 5.10(b) are shown some topics that where not mentioned yet in this document like is the case of the *attendedRequest* and *RequestVision*. These are topics to communicate with the agent for the pan and tilt and will be explained further ahead in this document.

In Figure 5.10(b) it is also shown the *map_server* and the service announced by this node as



(a) Software Architecture for Point Detection



(b) Software Architecture for Selecting Camera Pose

Figure 5.10: Software Architecture of the Vision Software for Localization

static_map. This service provides the map loaded by the *map_server* node and the */amcl/* is just the name space where the service and node where launched. The choosing of a name space for these particular node is going to be explained further ahead in this document.

5.3.2 Signs Detecting

One of the objectives on the Autonomous Driving Challenge from the Portuguese Robotics Open is to recognize vertical and traffic light signs. Aiming for the participation of the robot into such challenge, software was created to detect the signs.

***sign_detector* node** The identification of traffic light signs is suitable for the server/client paradigm. The navigation agent requests the identification of the traffic sign and it is responded with a sign. However, the identification of the traffic light signs is meant to be performed while moving and servers are blocking. This means that if the identification of a traffic light sign is performed over a service the navigation agent stays blocked after sending an identification request until the response is sent by the service. This approach is not wise since the navigation agent is the main piece of software.

To deal with this situation, the traffic light signs identification is implemented over a node but with service behaviour. The node is blocked until a request is performed through a publication in a request topic. When the request is received, the node identifies the sign, sends the response and then blocks again. With this implementation, the navigation agent may send a request for identification and wait for a response over a topic without being blocked.

The topic for requesting an identification is associated to a simple integer message. This integer represents the number of consecutive identifications the node should do before publishing a result. On the other hand the topic where to publish the response is associated with a message created specifically for the effect, and is represented on Listing 5.3.

Listing 5.3: Message used for Publish the Result of Sign Identification

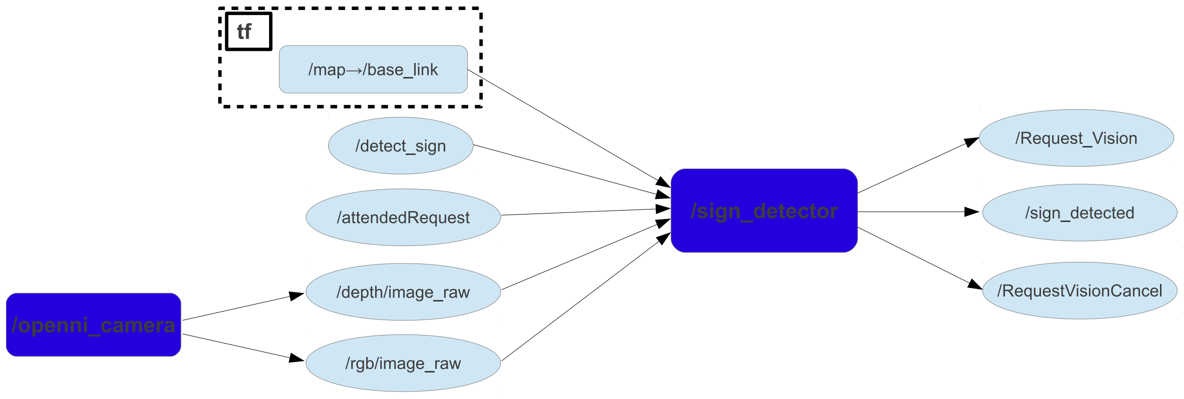
```
uint32 sign
float32 probability
#defined as in the file include/sem_signs.hpp
uint32 AHEAD = 0
uint32 STOP = 1
uint32 LEFT = 2
uint32 RIGHT = 3
uint32 CHESS = 4
uint32 NOT_CLEAR = 5
```

The response is composed by the sign number, identifying the sign it self, and the probability associated to the identification, indicating the confidence of the identification performed. The numbering of the signs defined in this message is related to definition of the templates used for comparison as explained hereafter.

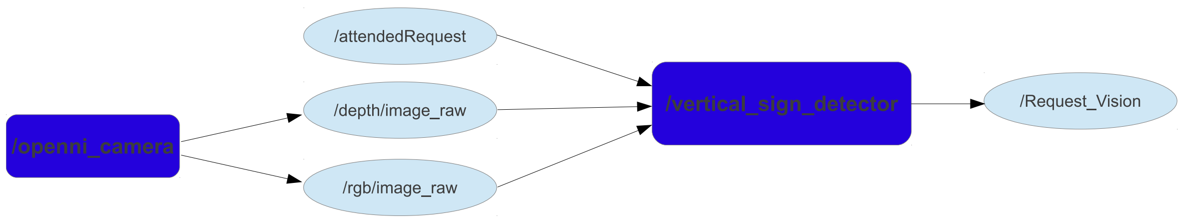
In order to load the templates for the identification and associate an id to each sign, a class was created to provide abstraction over the sign's templates. This class is implemented in a single *hpp* file and when this class is instantiated it creates two arrays, one containing the path for the loadable templates, and other containing the sign's name. The positions of the signs in the arrays is fixed and correspond to the numbering defined in the response message shown in Listing 5.3.

***vertical_sign_detector* node** This node contains the software necessary to detect and identify vertical signs. This node is completely independent, it does not subscribes to any topic that is not related to the vision. This means that this nodes subscribes to the RGB and depth images provided by the Kinect, publishes and subscribes the topics related to pan and tilt agent and nothing else. Since the vertical signs only have to be identified, and the agent does not have to do anything with this information, this is only indicated to the user by means of the terminal and visually, showing the identified sign in a window.

Having described the software responsible for signs identification and its implementation, one may now define a software architecture for this part of the *rota_vision* package. The Figures 5.11(a)



(a) Software Architecture for Traffic Light Sign Detection



(b) Software Architecture for Vertical Sign Detection

Figure 5.11: Software Architecture for Signs Detection

and 5.11(b) show the system software architecture concerning the signs detection. It shows the nodes involved as well as the topics published and subscribed by each one.

5.3.3 *ag_PanTilt* node

This node is the implementation over ROS of the pan and tilt agent described in Chapter 4. It is the node responsible for coordinating the different requests for setting a camera pose. This is the only node that directly publishes pan and tilt orders to the car base.

In Figure 5.12 is shown the software architecture of the agent for the pan and tilt. The topics and the architecture are defined by the node functions and its necessities. The message associated to the topic for requesting a given pose, the *Request_Vision*, is composed by the pan and tilt angles, the tolerance for the pan and tilt and the id of the requesting node. The message associated to the topic indicating which nodes are attended, the *attendedRequest*, is composed by a pan and tilt pose and an array of ids, indicating which nodes the pose serves. As for the topic for cancela given request, the *RequestVisionCancel*, it is associated with a message only containing an id which identifies the node

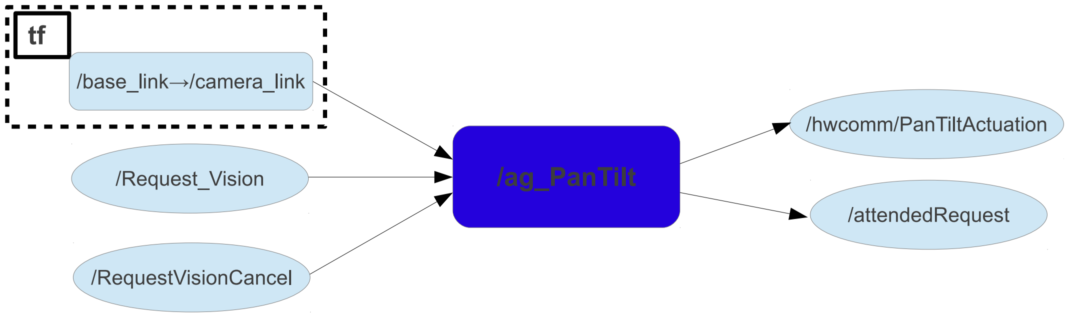


Figure 5.12: Software System Architecture for the Pan and Tilt Agent

cancelling the request.

5.4 Localization

To adapt the *amcl* package to support the Ackerman motion model and point cloud as input data, one has to take into consideration the organization of the software in order to alter it correctly. The *amcl* package software is divided into four main parts:

- *amcl* node;
- map related software;
- particle filter related software;
- sensor related software.

The package is organized so it can be altered and sensors may be added easily. The software has to be altered in the sensor software since odometry and point cloud are sensory information. The main class of this part of the software is the *AMCLSensor* class, which is an abstract class, and that will support sensor type classes. This organization facilitates the implementation of new sensors just by adding support classes for the *AMCLSensor* class. The already implemented classes in the *amcl* package for the sensory part are the *AMCLOdom* and the *AMCLLaser*.

5.4.1 *amcl* Ackerman Motion Model Support

From the organization of the *amcl* package software, to include the Ackerman motion model, one has to add it to the already implemented class *AMCLOdom*. In order to include support for the Ackerman motion model in the *amcl* package, one has to add another option to the *odom_model_type* parameter which is done in the code of the *amcl* node. Also, it is necessary to add the *SetModelAckerman* method to the class.

The really important alteration is in the motion step which is performed by the *UpdateAction* method of this class. There, it is necessary to update the motion of the particles accordingly to the Ackerman's motion model. In the motion update step of the particle filter, instead of adding to the particles the motion measured, one has to add the motion plus random Gaussian noise. The difficulty in this step, since the functions for generating the error are already provided by the package, is on how to add this error. The motion that the *amcl* node has access to is the Δx, Δy and $\Delta\theta$. However, the error can not be added directly to this values.

As stated in Chapter 4, it is necessary to add the error to the direct odometry measurement sources which are the covered distance and the steering angle. To account for that fact, one has to do the inverse calculation that is performed for calculating the odometry thus obtaining the covered distance and the steering angle. Having those values, one can add error to them and only after recalculate the Δx, Δy and $\Delta\theta$ to update the particles.

Using the same approach used on Chapter 5.1.3 but on reverse, one may obtain the equations shown in 5.3 for d, the covered distance, and for α, the steering position.

$$\begin{cases} \alpha = \arctan\left(\frac{L \times \sin(\Delta\theta)}{\Delta x}\right) \\ d = \frac{\Delta\theta \times L}{\tan(\alpha)} \end{cases} \tag{5.3}$$

The L value is not known since it depends on the car, however, and since the *amcl* node has five values for the odometry model configuration, and the the Ackerman motion model only needs two values, one for representing the standard deviation for d and another for defining the standard deviation for α, one may use another of these five parameters for defining the distance between axis L.

Having α and d calculated, one may use the already provided function for generation of random Gaussian error to add noise to d, using the standard deviation defined in parameter *alpha1*, and to α, by using the standard deviation defined in parameter *alpha2*. After the noise is added, one has to recalculate the odometry calculating first the new $\Delta\theta$ and the new R as it is shown in the equations represented in 5.4.

$$\begin{cases} \Delta\theta' = \frac{d_{noise}}{L} \times \tan(\alpha_{noise}) \\ R' = \frac{d_{noise}}{\Delta\theta'} \end{cases} \tag{5.4}$$

With R' and $\Delta\theta'$, that already contain noise, one can recalculate the motion to update the particle position as it is shown in the equations represented in 5.5.

$$\begin{cases} \Delta x' = R' \times \sin(\Delta\theta') \\ \Delta y' = R' \times (1 - \cos(\Delta\theta')) \end{cases} \tag{5.5}$$

With the aforementioned information the update step sums to the following steps:

- calculate α and d;
- enter loop to all particles;
- add random noise to α and d;
- recalculate odometry step;
- update particle position.

5.4.2 *amcl* Point Cloud support

As mentioned, another problem of the implementation of the *amcl* package is that it only supports laser scan as input perception data. However, due to its organization, it is possible to add point cloud support. To make the *amcl* capable of accepting point cloud as perception data input it was created a copy of the already existing *AMCLLaserData* class named *AMCLPointCloudData*. The created class has the same configuration as the class already existing and it was altered to support the point cloud data type.

The code for the update sensor step is essentially the same since the laser scan data is transformed into (x, y) coordinates before being used and the point cloud provides directly this coordinates. The main difference between the classes are the data type, and the sensor model.

The *amcl* package supports two algorithms for sensor update, the beam model and the likelihood field model. In the beam model, to estimate the best point in the map that corresponds to the measurement done, the beam of the reading is extended looking for the nearest point with the bearing correspondent to that beam. While the likelihood field model searches for the nearest point in the map in a given area around the measurement. Please refer to Chapter 2.2 for detail on the sensor models, especially to Figures 2.8(b) and 2.8(a) where is shown how these approaches find the measurement done in the map. Understanding the models for the sensor update step, it is relatively easy to assume that the beam model does not make sense for point cloud data input, and so, the support for this model in the *amcl_pointcloud* was eliminated.

With the two described alterations to the *amcl* package, the *amcl* node is fully functional for a car like robot and for main perception sensor used, and it may be used with much better results than the results expected than using the unaltered *amcl* package.

5.4.3 Experiments and Results

The localization was tested in two ways. The first test performed to the localization was done comparing the results of the localization provided by the *amcl* node with the measured position. This test was not very rigorous due to the lack of means for testing and lack of time to create such methods. The procedure effectuated to execute such test will be explained latter in this section.

The second test was performed by the navigation algorithm. The navigation, considering the approach, can only be successful with an accurate localization system. The ultimate test was performed in the Autonomous Driving Challenge in the Portuguese Robotics Open where the robot acquired the first place, proving the success of the implementation.

The test for the localization by it self was performed in robotics lab by launching the localization node and the required nodes for providing the necessary information for the localization algorithm. Then the robot was driven through the track using the remote control module and the localization computed by the *amcl* was checked using the floor of the lab. The floor is composed by squares of half a meter side where the lines of the track are draw using adhesive tape. Using the squares of the floor as reference, and setting the same squares of the floor in the Rviz environment, it was possible to perform a small test, with the precision that the test set implies, to comprove the results of the localization algorithm.

In Figures 5.13(a) through 5.13(e) is shown the evolution of the particles performing the

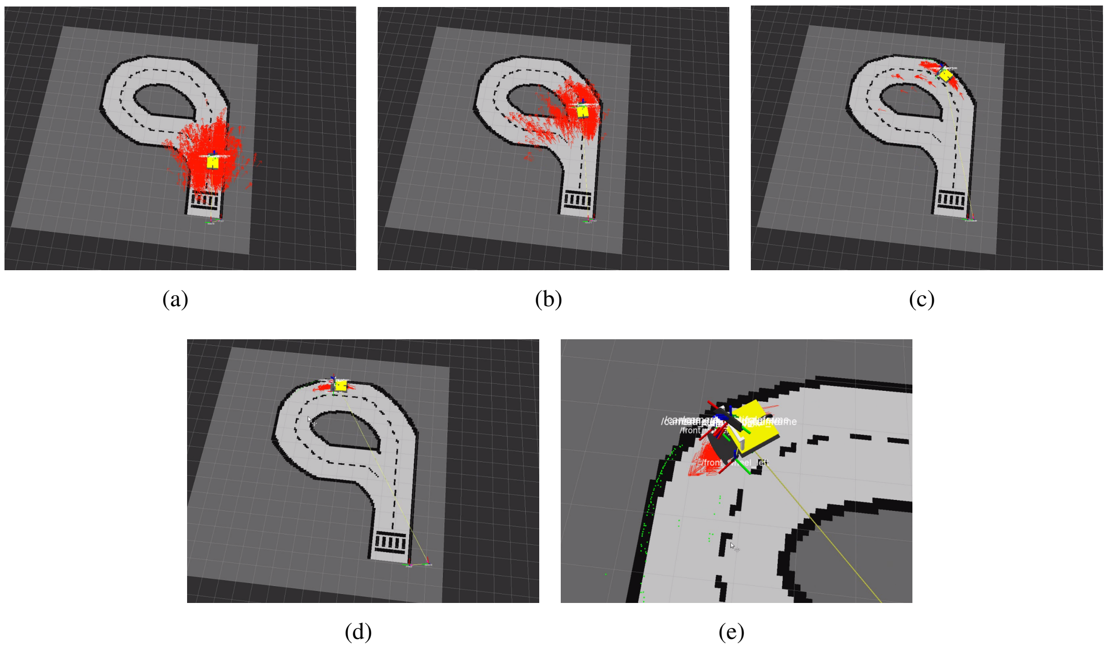


Figure 5.13: Evolution of the Particles

aforementioned method. The images are not simulated, they are screen shoots of the Rviz environment while performing the localization test. It is possible to see in these figures, that the particles have the expected evolution. These start as a big blob in order to have a high probability on having the real pose represented. The size of the starting blob is definable by parameters, however, if the blob is to narrowed, one has to position the robot correctly in the starting pose. The blob then evolves into a more concentrated and narrowed blob as the robot moves through the track and fuses the movement with the detected landmarks. The importance of the landmarks is remarkably notable in the first curve that the robot performs. The size of the blob before and after the curve is immensely different which is consistent with the expected result.

In the final two shown images it is possible to see that the blob is really narrow and concentrated meaning that the localization is precise with a great confidence of its value, having thus, low values for the covariance matrix. This is consistent with the expected and shows results precise enough for the application in hands.

5.5 Navigation

As explained in Chapter 4.7, the navigation module, which is implemented by the *rota_navigation* package, is composed by two main nodes, one for the path planning and motion planning, defined by control, and another for the navigation.

Also, the navigation module is mainly supported by the *move_base* package, which will handle loading the map and setting the cost maps, the creation of a path to a goal and the maintenance of the cost maps inserting obstacles where detected and also clearing the cost map where no detection is made.

The global path planning, the only one used, is based on the use of the Search Based Planner Library (SBPL) over ROS which is already available. But before its usage, one has to create the motion primitives which will describe the tree of motions to use in the searched based algorithm for planning.

Another need of the approach taken, is the implementation of a communication language between the main agent node (*navigation_agent*) and the node in charge of the path planning and the control (*goal_reacher*). It is required that the *navigation_agent* node communicates to the *goal_reacher* node indicating which goals to follow and also to cancel in any case. Also, in order to the navigation agent know when to publish these goals, and if they were accomplished, the *goal_reacher* node has to communicate the state of each goal to the *navigation_agent* node.

5.5.1 Creation of Motion Primitives

For using the SBPL implemented over ROS, one has to provide a motion primitives file which will allow to the search based algorithm to compute a path feasible by the robot. This file should contain all movements that the robot can perform assuming a discrete world decomposed by cells. Each primitive describes a possible motion each is described by a starting pose and an end pose, in cell coordinates, the cost of this motion and the intermediate poses between the start and end cell in meters. Depending on the robot and the locomotion system, the motion primitives file may contain hundreds of primitives, making the creation of the file by hand very difficult and burdensome.

The SBPL implementation over ROS, namely the *sbpl_lattice_planner* package, which is the SBPL implementation for two dimensional planning, provides Matlab functions for generating the motion primitives[5]. It provides a function for producing unicycle primitives, unicycle plus sideways, unicycle plus sideways plus back turn and unicycle plus sideways plus back turn plus diagonal. The function that best fits a car like robot is the simple unicycle function.

Using this Matlab function, and configuring its internal parameters, one can create the motion primitives file. The parameters have to be configured to the robot and to the application. The main parameters altered were the vectors which define the movements that can be performed. These vectors define an end pose (x, y, θ), in cell coordinates and considering as start point the $(0, 0, 0)$, for turning 11.25 degrees, 22.5 degrees, 33.75 degrees and 45 degrees. These four angles define sixteen simple motions, eight for left, being four for forward movements and four for backward movements, and the same for the right.

Another parameter, to be defined is the number of primitives for each angle, which was defined as eleven. This defines the number of primitives extrapolated from each of the previous defined ones meaning that there will be created 176 motion primitives. If the number is to high, the algorithm may take to much time to compute a path, if the number is two low, the resulting path may not be smooth enough to be followed or it may occur that a path is not computed at all. This parameter is setted by experiencing the primitives and analysing the computed path using them.

Once the parameters are setted one can use the function to create the primitives file for the *sbpl_lattice_planner*. In Figures 5.14(a) through 5.14(d) are shown as an example the first four primitives created. These figures are an output of the Matlab function used for creating the primitives. As an example, in Appendix B.3.1, are shown four primitives of the primitive file created.

5.5.2 Communication Between Agent and Control

The communication between the *navigation_agent* and the *goal_reacher* nodes has two directions, from the first to the second for indicating goal poses to pursue, and from the second to the first for indicating the state of the goals and action commands. The implemented communication is based on the one already implemented by the *move_base* package.

The communication is supported by the ROS message passing system, meaning that the communication is based on messages. The message created to allow the *navigation_agent* to

[5]`ros.org/wiki/sbpl`

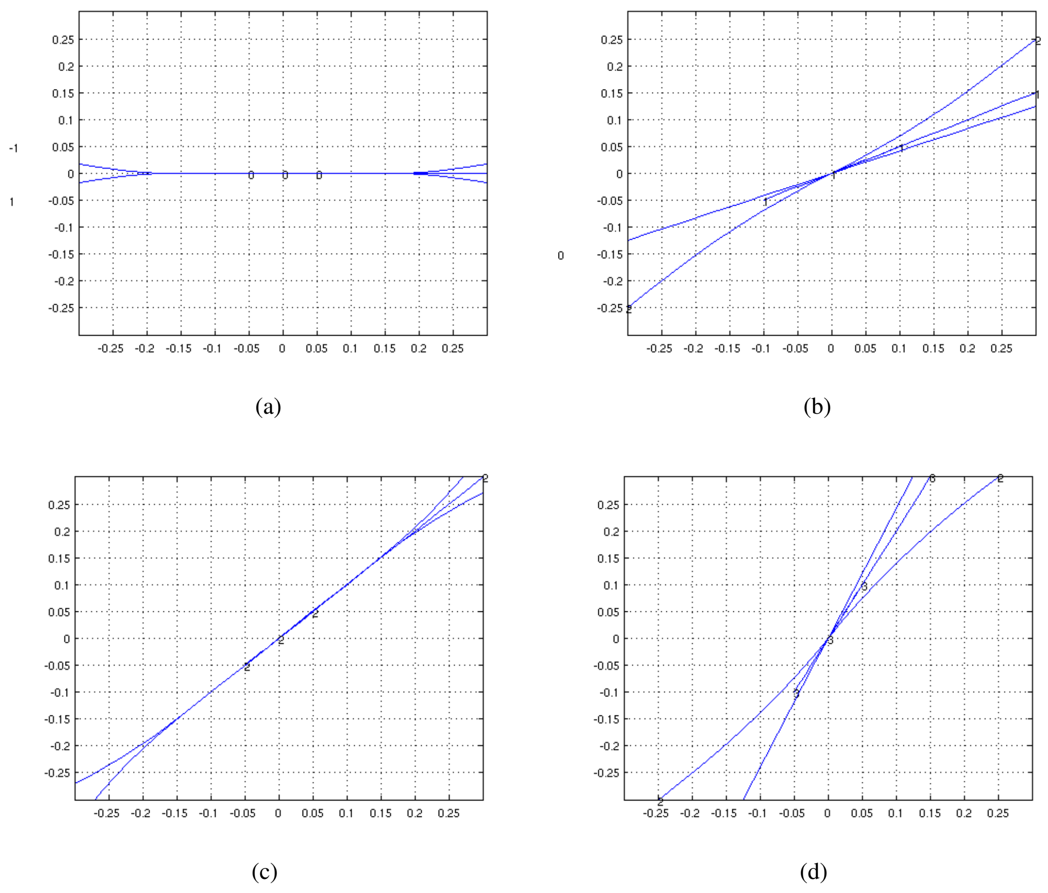


Figure 5.14: Graphical Representation of Four Pre Defined Motion Primitives

communicate the goals to pursue to the *goal_reacher* has the following implementation:

Listing 5.4: Message for Publishing Goals to Pursue (rota_navigation/goal)

```
rota_navigation/goal_status goal_status
geometry_msgs/PoseStamped target_pose
float32 tg_vel
```

The first field of the message is the message defining the state of the goal which will be described further ahead in this section. The second field is the target pose, the goal, to be pursued by the *goal_reacher* defined as (x, y, θ), which is complemented by the third field which indicates the velocity which the robot should have on that goal. This message implements the first objective of the communication between the two nodes, indicating which goal to pursue.

The second objective of this communication is to allow the *goal_reacher* to communicate the state of each goal, and this is supported by a message which is an array of the following message:

Listing 5.5: Message Used to Communicate the State of a Goal (rota_navigation/goal_status)

```
actionlib_msgs/GoalID goal_id
uint32 status
uint32 ASKING = 0
uint32 PENDING = 1
uint32 CANCEL = 2
uint32 CANCELLED = 3
uint32 ABORTED = 4
uint32 PURSUING = 5
uint32 SUCCEDED = 6
uint32 READY = 7
```

The *goal_id* is to identify each goal uniquely and is setted by the *navigation_agent* node when publishing a goal to pursue. The *status* field defines the state of the goal identified by the *goal_id*, and the states possible are defined in the message as well.

The defined states for a goal are a direct consequence of what may happen when pursuing a goal. The *READY* state is a dummy state, just used by the *goal_reacher* node to indicate that is ready for receive goal requests. Also, the *ASKING* state is used by the *navigation_agent* node to publish a request, making this state the first state of a goal. As for the rest of the states, are defined as follows:

- PENDING: the goal has already been assumed by the *goal_reacher* node and it is on the waiting queue for being processed;
- CANCEL: used by the *navigation_agent* node to cancel goal requests already assumed by the *goal_reacher* node;
- CANCELLED: this state represents that the *goal_reacher* node already assumed the cancel order on the request and this is no longer in its waiting queue nor being followed;
- ABORTED: used by the *goal_reacher* node to indicate that the goal was aborted and will not be pursued, this may be caused by the impossibility on reaching it;
- PURSUING: the goal is being pursued by the *goal_reacher* node;
- SUCCEEDED: this state indicates that the goal as already been reached successfully.

Using this communication, the *navigation_agent* may implement the point by point driving approach and can monitor the defined points and the navigation it self, publishing the goals to pursue accordingly.

5.5.3 *goal_reacher* node

The *goal_reacher* node is the one responsible for pursuing the goals published by the main navigation agent. This node has a waiting queue of goals received, it computes a path for the next goal on the waiting queue and computes action commands for following the computed path to a goal. This action commands are published and then are applied by the *navigation_agent* node. Also, and since this node is responsible for the path planning, it is responsible as well for monitor the local cost map to evaluate if it is possible to reache a given goal and if it is necessary to compute a new path if the conditions of the cost map change.

To handle the receiving goals the *goal_reacher* has a small waiting queue and handles the requests sequentially one by one. Also, each goal as at least three stages in its life cycle, the planning state, the pursuing state and the ending state. Two handle the waiting queue and the states of each goal, the *goal_reacher* node is implemented by a state machine with four states, and functions to handle the waiting queue and publish the state of the goals every time a state of a goal changes.

The states implemented are the following:

- WAITING: this state is setted at the beginning, when the queue is empty, and also when the pursuing of a goal has finished to evaluate if there are any goals waiting to be pursued;
- GETTING_PATH: in this state the *goal_reacher* node calls the *move_base* service to compute a path, or to pursue the next goal and, if the environment changes in a way that the path has to be recalculated, to recompute a path to avoid obstacles;
- PURSUING: this is the main state of the *goal_reacher* node, it is where steering and velocity control values, in which result action commands, are computed (explained in Chapter 4.7);
- LOST: this state indicates that the *goal_reacher* node got lost in the pursuing of a goal, either by missed calculated action commands or by wild variations in the localization, which leads to a recalculation of the path or selecting the next goal to pursue.

All the states, at the exception of the PURSUING one, are of simple or direct implementation.

In the PURSUING state, before computing the action commands, are evaluated the following conditions which invalidate the computation of the action commands if true:

- if the goal in pursuing has a cancel request, which leads to the abandonment of the pursuing of that goal and choosing another goal to pursue from the waiting queue;
- if the distance to the pursuing goal is smaller than a user defined threshold, which leads to the same aforementioned situation;

- if the distance to the closest point in the path being followed is greater than a user defined threshold, which leads to the LOST state.

If none of the aforementioned conditions are true, then, the *goal_reacher* node, computes the action commands.

5.5.4 *navigation_agent* node

The *navigation_agent* node contains the higher level of intelligence and decision of the robots software. This node is responsible for coordinating the navigation and monitor it. It is responsible for publishing the goals for the *goal_reacher* node and monitor them, request the identification of the traffic light sign and above all, it contains the deliberative behaviour for driving, being the only node in the system sending action commands to the car base node.

As stated in Chapter 4.7 this node is implemented with a state machine due to the well defined objectives of the Autonomous Driving Challenge and due to the pose by pose driving approach that is easy to implement recurring to a state machine. The state machine is configured by a list of points in which the robot should pass, by the number of laps to complete and also by two points that represent the position of the crosswalk in the track. The points which define the crosswalk, define where the robot should stop when approaching the crosswalk for identifying the traffic light sign if this was not successfully identified while approaching.

The state machine is implemented by the following represented states.

INIT This state is the initial state of the navigation agent. In this state, the agent as already loaded all the parameters, including the points used for driving. This state is abandoned to the IDLE state when the *goal_reacher* node communicates that is ready for receiving goals to follow, and also, when the *camera_pose_selector* node informs that is ready for computing the best pose for the pan and tilt for driving purposes.

IDLE This state is a waiting state. When the state machine is in this state means that is already fully operational for navigating as well as all the nodes required by it. This state allows to the user to set the start order. When the start order is given, this state is abandoned to the ADD_GOAL state.

ADD_GOAL In this state the next goal to pursue is chosen and published to the *goal_reacher* node. It evaluates the state of the goals published by the *goal_reacher* node in order to decide which goal to publish next. It also takes into consideration if the points defined for driving are already all published. If so, the next points to publish will the points for approaching the crosswalk. This state leads automatically to the MONITOR_GOALS state.

MONITOR_GOALS This state starts by evaluating if there are any goals sent, if not, then it sets the ADD_GOAL state. If there are goals published this state monitors them. The monitoring of the states is dived into three main tasks. The first, is monitor if the last goal published is already in the PURSUING state, if so, it is setted the ADD_GOAL state so that the *goal_reacher* node does not have to wait for the next goal. The second, is evaluate if the point being pursued by the *goal_reacher* is the point for approaching the crosswalk, if so, then the CROSS_WALK state is setted. The third and last task is to evaluate if the goal being pursued can in did by reached. This evaluation is performed be monitoring the local cost map and see if the goal area is occupied. If the area is occupied, the pursuing goal is cancelled and it is relocated to the closest area not occupied. If the relocation is not possible the next goal for pursuing is published. The same situation happens if the goal being pursued goes into the ABORTED state.

CROSSWALK This state is setted when the robot is approaching the crosswalk. In this state is performed the request for traffic light sign identification. The identification starts to be performed in movement. The position of the robot as well as the ground sensors are monitored in order to stop the car if this reaches the crosswalk without a valid traffic light sign identification. When the sign is identified, and it is not the stop sign, the state changes to END_LAP. If the sign is the stop, then a new sign identification request is published.

END_LAP This state is setted when the robot completes a half lap to the track. It starts by evaluating if the pre defined number of laps was performed, if so then the car does one of two possibilities. If it is defined the parameter for parking, then it evaluates the cost map to choose the parking spot publishing the parking point to the *goal_reacher* node, if not, then the END state is setted. If the number of laps is not completed then the points for driving are reset and ordered depending on the sign identified, right or left. With a new set of goal points defined, the next goal is published and the state is changed to MONITOR_GOALS to complete another lap.

END This state is setted when the navigation has ended. In this state the *navigation_agent* just waits for all nodes to end and waits for the ROS system to be killed.

LOST This state is not setted by any other state. This state is set when the robot is considered lost and this evaluation is performed outside the state machine. In this state the car stops and the turning signals blink so that the user may see that the car is lost.

To evaluate if the car is lost, the covariance matrix published by the *amcl* node is evaluated. Basically are allowed values for the covariances that may represent the robot still inside the track. This means that the evaluation also depends on the position of the car on the track.

With the previously defined nodes, one can define the software architecture, within ROS system, created for navigation. This architecture is represented in Figures 5.15 and 5.16. It is possible to

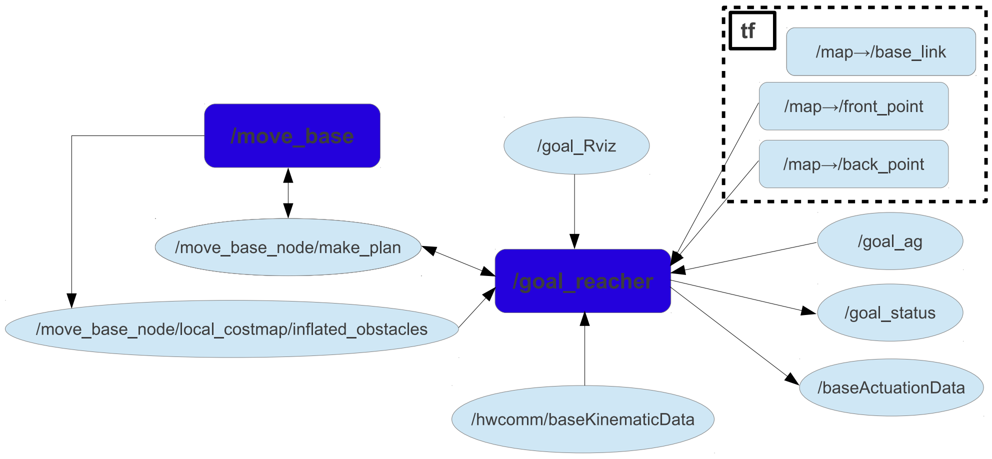


Figure 5.15: Software System Architecture for Motion Control and Path Planning

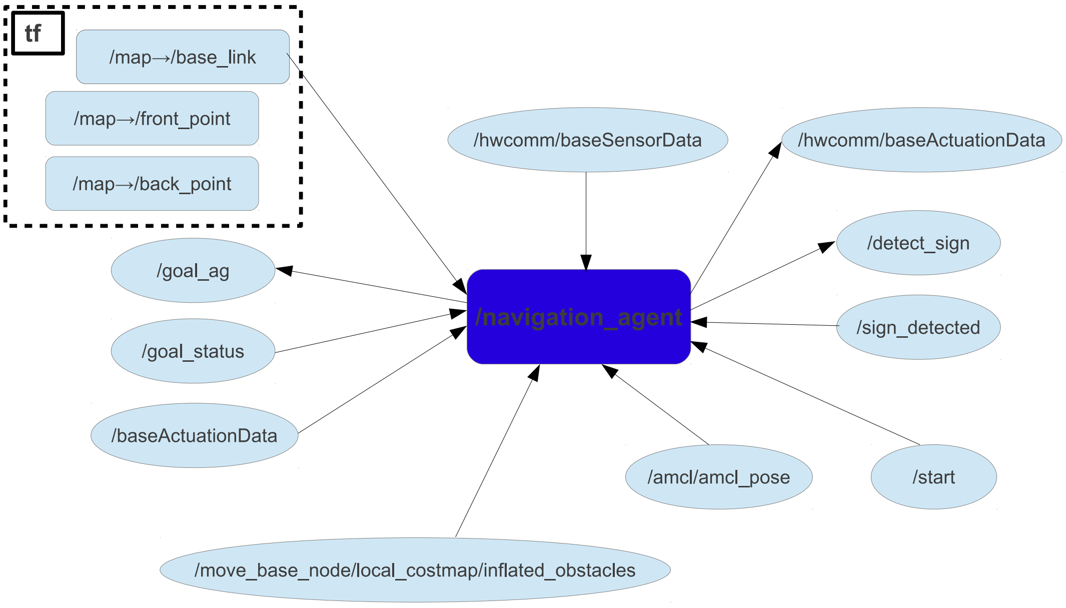


Figure 5.16: Software System Architecture for Navigation

see in Figure 5.15 that the *goal_reacher* node also subscribes a */goal_Rviz* topic. This topic is aimed to publish a target pose on the map using Rviz, making possible to test the *goal_reacher* node alone without the need of the navigation agent.

It is also possible to see that there are some topics within the *amcl* name space. The creation of this name space is to have two independent maps at the same time in the system. One map used for localization, in the *amcl* name space, and other map for navigation.

The map used for navigation does not contain the dashed line that divides the two lanes. The dashed lane is erased for navigation since it will be considered as obstacle in the map and it could condition the navigation. Namely, the dashed line could condition the change of lanes for deviation of obstacles.

5.5.5 Experiments and Results

The test of the navigation, as it occurs with the localization, was fully tested in the Autonomous Driving Competition. The ROTA project acquired the first place with a good performance that could only be achieved with a good navigation. As for individual tests, the path planning and the motion planning were tested separately.

The path planning was tested using Rviz. The robot was placed in the test track and a target pose was published using Rviz. The computed path is observable in Rviz and it is possible to evaluate if it can in deed be followed by the robot. In Figure 5.17 it is possible to see a computed path by the path

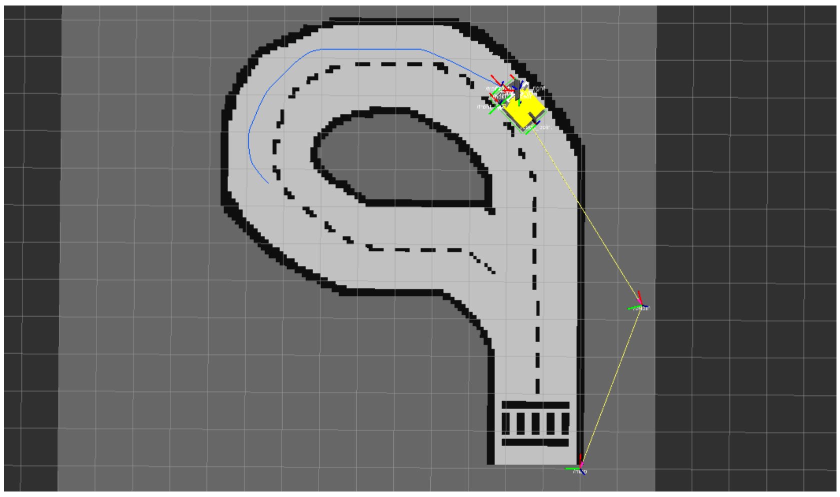


Figure 5.17: Path Planning Test

planning module.

The resulting path is drawn as a blue line in the map. It is possible to observe that the computed path is smooth and also respects the motion constraints of the robot as it can be seen in the path computed to complete the curve. It is possible to see that in order to the robot complete the curve of the test track it has to use the exterior lane which is true since the robot does not have enough steering power to complete the curve using the inner lane.

As for the test of motion planning, the procedure was the same. It was published a goal using Rviz, which resulted in the path represented in Figure 5.18. This path was then used by the *goal_reacher* node to accomplish the goal published in Rviz.

To have a control on the test, the robot was pushed instead of using its velocity controller. The

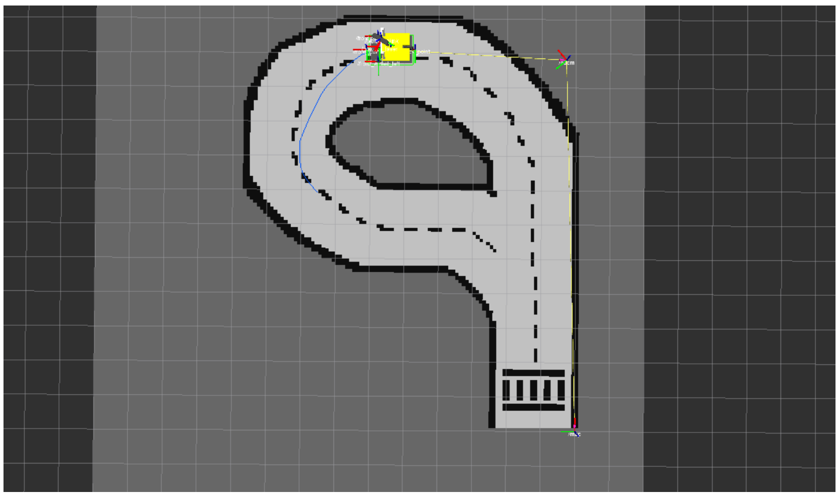


Figure 5.18: Used Path for Motion Planning Test

test was performed in this way so that the localization noise could be perceptible in the resulting cross track error since the robot was pushed slowly. This method allowed to identify the peaks in the cross track error that are a reflection of the localization noise.

Figure 5.19 shows the resulting cross track error while following the computed path. It is possible

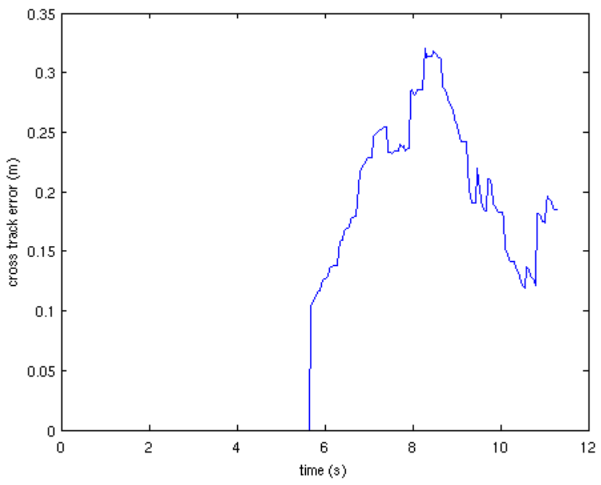


Figure 5.19: Graph of the Cross Track Error

to see that at start the error has a massive spike. This pikes was produced by the localization noise. When the robot started to move, the localization suffered a major correction that was reflected in the cross track error. After the localization stabilizes the cross track error tends to zero. Even with the cross track error diminishing it is possible to see some pikes in the cross track error that are a product of the localization noise.

The pikes in the cross track error are stated as localization noise since with this procedure it is impossible to have a displacement of almost 15 centimeters in such a short time. However, due to the smoothing effect of the steer controller, these pikes, provoked by the localization adjustments, are not reflected in the steering. The resulting driving is smooth and stable enough for the application that is aimed to.

5.6 Mapping

As it is explained in Chapter 4.8 the mapping module is implemented by the GMapping algorithm. The implementation of GMapping used, is the implementation available in ROS, which uses the revision 39 of the GMapping algorithm available on `www.openslam.org/gmapping` and it is available in the *slam_gmapping* package. Having an efficient SLAM algorithm already implemented over ROS it is necessary to understand how to use it in this particular application. To understand its use one should read the page for usage of the algorithm under ROS[6].

As explained in Chapter 4.8, the GMapping implementation over ROS has as input laser scan data as world perception data and the dead reckoning data from the tf tree, using the transformation from the odometry frame to the robot's frame. As for the laser scan generation it is explained in Chapter 5.3.1 and the generation of odometry data is explained in Chapter 5.1.3. With this, one has already information for the *slam_gmapping* to produce a map.

Besides making the information available for the algorithm, it is necessary to configure the it. The configuration of the algorithm over ROS is done using parameters for the *slam_gmapping* node. Some of the parameters are easy to set, like it is the case of the *maxUrange* parameter or *maxRange* that are parameters to crop the laser scan data. Other parameters are theoretically calculable, like it is the case of the dead reckoning errors. However, the calculation of the parameters, is nothing but an estimative which makes the effort to obtain that estimative ungrateful because the SLAM operation is not aimed to be done online and at runtime. Performing SLAM off line and with recorded data sets there is the possibility for repeatability, and the launch files in ROS, as well as the parameter server, allow to easily change the parameters of the *slam_gmapping* node and test those parameters with exactly the same information. This fact makes the theoretically estimation unnecessary.

As mentioned, the data for the mapping process is pre generated. To generate the data for mapping there is a software scheme similar to the one shown in Figure 5.2, but without the *slam_gmapping*

[6]`www.ros.org/wiki/gmapping`

and the *map_server* packages. It is launched the base software for communicating with the robot and publish the robot's state. It is launched also the software for remote controlling and the vision software for generation of the laser scan data.

With this software launched one can drive the robot on the track and control the pan and tilt pose using the remote controller. This possibilities the generation of data as the user see fits, namely, point the Kinect at the lines of the track and drive the robot through the track. With this setting one can generate data and record it to latter, off line, perform mapping.

To record the data one can use the *rosbag* tool, explained in Appendix A. Also, one can later play the recorded data using the same tool and with the published data perform mapping. To perform mapping with recorded data it is necessary to record the tf topic and the *road_scan* topic, the first containing the dead reckoning data and the second the world perception data.

With the recorded data, for simplicity, one can have a launch file for launching the *slam_gmapping* node with all its parameters. With this, one can play the recorded data and launch the *slam_gmapping* node, testing various sets of parameters easily. This parameters can then be tuned until a satisfactory result is obtained with the recorded data set.

Having the mapping procedure defined, it is necessary to save the resulting map. For recording or even publish a map, ROS offers the *map_server*[7] package from the *navigation* stack. The map generated by the *slam_gmapping* node is published in a topic, and the *map_server* offers a command line tool for saving the map into the computer.

The generated map is composed by two files, the *.pgm* image file containing the occupancy grid, and a *.yaml* file containing the properties of the map, namely the origin of the map, thresholds associated to the occupancy of the cells and the resolution, in meters, of the cells. Using the *map_server* one can the publish a map through a topic or by service, providing as parameter the *.yaml* file to be loaded.

To perform SLAM the robot uses the *slam_gmapping* node from ROS with no alteration. The only thing done to successfully obtain a map using ROS implementation of the GMapping algorithm was parameter tunning until an acceptable result was obtained and the production of the necessary information for the algorithm. Also, it were recorded various data sets for testing since the data it self influences the resulting map.

5.6.1 Experiments and Results

The Figure 5.20 represents three results from three different data sets of the afore mentioned method for mapping. All the data sets were recorded in the robotics lab (room 303) of the Electronics, Telecommunications and Informatics department. In this room, there is an half track drawn on the floor with reduced dimensions for testing.

[7]`www.ros.org/wiki/map_server`

The two first shown maps are acceptable results, and were even used to navigate successfully on

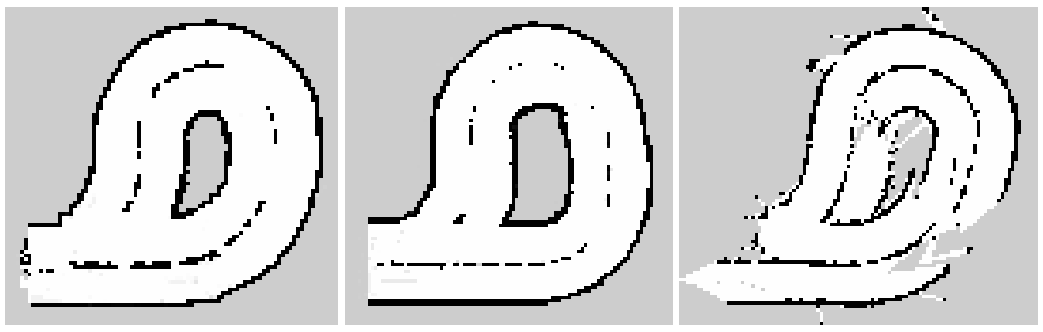


Figure 5.20: Maps Built from 3 Different Data Sets

the test track. All three shown maps were built using the same parameters however the third is an unacceptable result for navigation.

Although the two first maps shown seem acceptable, it is possible to see that the geometry is different, namely the loop. The closed loops in the GMapping are a problematic thing to map and one has to perform parameter tunning to the odometry parameters in order to account more or less the odometry to open or close more a loop.

Another problem encountered by using the GMapping algorithm to this particular case, was the dashed center line of the track, which is not exactly represented. The little details, like the dashed line, are a problem in almost every SLAM algorithm since they get deleted in the scan matching step. The little details can then be enhanced by using an image processing software. The dashed line was corrected to use the map for localization since it helps the localization algorithm.

Chapter 6

Conclusions and Future Work

In this chapter are made some conclusions to the performed work and added hardware. Also, are made some observations and conclusions to the implementation performed and all the modules implemented for the various objectives and challenges that the autonomous driving problem has. In this chapter are described some ideias for future work related to this dissertation. In this chapter are described some features that can be improved or added to the existing project, but also, are explained some different approaches that one can use to solve the autonomous driving problem.

6.1 Conclusions

With the work developed on this dissertation a lot of work was performed in the ROTA project and basis were created to support future work. For instances, the robot has now a new vision sensor that proved to be reliable and efficient. The Kinect added to the robot provides three dimensional vision which is a more interesting problem from the research point of view than two dimensional vision. Also, the new vision sensor provides support for research in other areas and provides support for new autonomous driving algorithms.

To complement the vision sensor, the pan and tilt structure was added and were created new possibilites for research on the artificial intelligence in the autonomous driving field. The pan and tilt structure, allied with the vision sensor, provides a robotic platform that is more similar to the human, making the robot a more interesting platform for research in this field. The agent for the aforementioned pan and tilt structure also provides a base for research for artificial intelligence algorithms applied to the vision system, not only for the ROTA project, but also the CAMBADA@HOME robot which possesses the same vision system as the one implemented now in ROTA.

Another important sensor added to the robot was a LRF which gives the possibility on implementing new approaches for obstacle detection, localization and even mapping. With this sensor one can localize and navigate using the surrounding environment instead of the lines of the floor. Also, with the LRF one can have a support for the autonomous driving, performed using the Kinect, as it happens with the already existing assisted driving algorithms and technologies.

Still in the hardware area, the replacement of the batteries and the powering system, was proven to be a very good option. This alteration provided more running time to the robot and the possibility of adding more hardware to the robot. Also, with the replacement of the powering system, it is possible to work much more time with the robot without the concern of changing and recharging batteries which really improves the working time with the robot, improving, as consequence, the final result.

As for the software developed, namely the navigation approach, is only suited for the Autonomous Driving Competition and it is not robust enough for any environment. However, many of the software developed, presented very good results and is an important base for future research on the autonomous driving field. The mapping and the localization algorithms functioned just fine and are only one step away for being used in a real autonomous driving environments. In the localization software it was possible to notice how important is the motion model. The differential motion model was first used and the results, even though the parameters were carefully tunned, were not as good as expected and the robot could localize very well and would become lost very often. Also, the software for signs detecting, traffic light and vertical, functioned very well, although it requires some work for improving the time of the detection and the robustness.

As final conclusion, the performed work within this dissertation as proven to be a good implementation and approach for the autonomous driving problem. This conclusion is made, corroborated by the performance of the ROTA project in the Autonomous Driving Challenge in the Portuguese Robotics Open, where the ROTA project acquired the first place.

6.2 Future Work

Use Gazebo for Simulation

The ROS framework has support for simulation in various ways. For example, ROS allows to record data sets which may be ran off line has if the robot was present in the system. It is possible with this tool to support simulation using previously recorded data. However, with this tool, it is not possible to reproduce action commands but only data processing. Also, the data has to be recorded first.

It is possible to implement a full simulation environment using ROS and Gazebo. ROS provides libraries for interfacing with Gazebo using the underlying publish/subscribe paradigm. It is possible to set environments and accurate physical models of robots to fully simulate navigation localization and other problems. The environment may be set using grid maps, has the one built by the GMapping algorithm. It also contains sensor models to simulate data acquisition from the map.

One of the most important requirements is the geometric, physical and kinematic description of the robot in order to simulate it in Gazebo. This is achieved by implementing the URDF file with all the frames and joints of the robot as well as the physical constants which define the kinematics of the robot.

As future work, one could use the created URDF file within this dissertation to built a simulation environment with Gazebo. To do this, since the created URDF model already contains all frames and joints required by Gazebo, one has to measure and add the physical constants and the file would be ready to be used in Gazebo. This would allow to simulate the software implemented since there are already created maps.

Improving the Localization

With the addition of the LRF, a new localization and mapping approach may be taken. At first, the data that the LRF provides, is directly compatible with most of localization and mapping algorithms, which by it self possibilities the use of a wide range of algorithms already implemented and tested. Another advantage of the LRF, is that the environment captured by it tends to be structured and less ambiguous than the one captured at this point by the Kinect.

The track, has two closed loops exactly symmetrical which diminishes the confidence, and by consequence, it degrades the results of the localization process. With the use of the LRF as sensor for world perception, the aforementioned problem tends to be reduced since the surrounding environment tends to be less ambiguous. However, the map of the track should by used for navigation since it is what defines the course and the driving approach.

The ideia for improving localization is not to use the Kinect or the LRF as singularities for world perception. The ideia is to perform mapping with the LRF and with the Kinect at the same time, originating two different maps, one with the surrounding environment and other with the track. With these two maps, while navigating, one could perform two localization algorithms in parallel and merge the results of the two algorithms to have a better localization. Also, this possibilities to use less the Kinect for localization and more for the navigation. For example, this possibilities to recognize the vertical signs and the traffic light signs while driving, since when the Kinect is pointed at them there is still information for localization from the LRF.

Improving Motion Planning

The motion planning is robust and with results good enough for this concrete application. However, this should be complemented with adaptive techniques to make the motion planning more stable and less inherent to the localization errors. This could be performed by evaluating the covariance matrix of the localization and applying filtering functions to the cross track error before applying control.

The path planning should also be more dynamic. In the current implementation, the path is computed from goal to goal and it is recomputed in the presence of a change in the local cost map. The path planning should be implemented as it is in the *move_package*. The *move_base* package does not compute a path only from goal to goal. It is constantly computing a path that is merged with the previous computed ones. This even improves the goal transition oscillations provoked by the computation time of a plan between goals since the goals could be added dynamically and the path computed would be adapted to previous one. Of course that this alteration can not be performed using the service that is being used since the response is slow for a real time operation. This could be implemented by adapting the existing *move_base* package to support a car like robot or even reimplement it in a more suitable way for this application.

Improving Signs Identification

The implemented algorithm for sings identification can be improved. For instances, the detection of vertical signs, since its position is not known, could be improved with the use of machine learning. The robot could indeed look for the base of the sign for detecting it. However, this should be performed only once. In the next laps, the robot could use the knowledge of the previous detection and identification. Also, the vertical sign detection should be complemented with tracking so that the camera pose could follow the sign performing the identification while moving.

Has for the traffic light signs identification, the implemented algorithm is robust enough although it takes to much time for performing a valid identification. The processing time could be improved by having an already trained classifier with many signal samples. With it, the algorithm did not have to compute the features of the template, and just perform the comparison.

The previously mentioned improvement is also applicable for the vertical sign detection. Both of the algorithms could have, instead of templates, trained classifiers in order to perform a quicker identification.

Another Approach on the Driving Intelligence and Agent

The approach on driving at this moment is based on localization entirely. The robot has pre defined points for passing on, depending on the type of driving chosen, and it plans a path from one point to another. Then, the robot uses localization to apply velocity and steering control to follow the computed path.

However, this approach on driving, is counterintuitive. A human, for driving, just stays on the lane, changing lanes when needs to overcome an obstacle, and it only uses localization for intersections and so. Even more, the localization used by a human being, is completely visual and with a precision of no more than half a meter depending on the car and on the driver.

The aforementioned fact leads to another idea for implementing the autonomous driving, one more human like. The idea is to implement the driving based on the same manner of a human, not using localization for the navigation and not having a metrical map. The robot should not use a metrical map, but a topological one, where each node represents an intersection and each intersection is represented by features. In this approach, the path planning is performed over a topological map, and it is used control for maintaining in a lane.

However, this implementation obligates to have local path planning for maneuvers, and also behaviours for specific situations. These described situations imply, parking, passing obstacles and others. The use of a topological map implies other problems, namely, on how to perform mapping and how to identify each node. To identify each node, one could use a coarse localization algorithm allied to specific features of each intersection for identification, the localization would only serve as ground basis for validating the identified node of the map. As for mapping, it had to be a different method from the one used where the robot is driven around identifying intersections, constructing thus a topological map containing the intersections and the connections between them.

Bibliography

[1] José Azevedo et al. “ROTA: A Robot for the Autonomous Driving Competition”. In: (2007).

[2] Brian Benchoff. *3D mapping of rooms, again*. URL: `hackaday.com/2013/06/08/3d-mapping-of-rooms-again/`.

[3] Subhrajit Bhattacharya. *A* algorithm progress animation*. URL: `en.wikipedia.org/wiki/File:Astar_progress_animation.gif`.

[4] Subhrajit Bhattacharya. *Dijkstra’s progress animation*. URL: `en.wikipedia.org/wiki/File:Dijkstras_progress_animation.gif`.

[5] K. Birman and T. Joseph. “Exploiting virtual synchrony in distributed systems”. In: *Proceedings of the eleventh ACM Symposium on Operating systems principles* (1987), pp. 123–138.

[6] Joseph Carsten, Dave Ferguson, and Anthony Stentz. “3D Field D*: Improved Path Planning and Replanning in Three Dimensions”. In: (2006).

[7] Edward Curry. “Middleware for Communications”. In: John Wiley & Sons, Ltd, 2004. Chap. 1 - Message-Oriented Middleware.

[8] Rina Dechter and Judea Pearl. “Generalized best-first search strategies and the optimality of A*”. In: *Journal of the ACM* (1985), pp. 505–536.

[9] E.W. Dijkstra. “Numerische Mathematik”. In: 1959, pp. 269–271.

[10] *Dynamixel AX-12 User’s Manual*. 2006.

[11] Felix Endres et al. *GBDSLAM - 6DOF SLAM for Kinect-style cameras*. URL: `www.openslam.org/rgbdslam`.

[12] Nikolas Engelhard et al. “Real-time 3D visual SLAM with a hand-held RGB-D camera”. In: ().

[13] Leandro Ferreira. *Vertical Sign Detection and Identification*. Tech. rep. Final Project Report for Computer Vision Course. DETI - UA, 2012.

[14] Leandro Ferreira and Michael Costa. *Agent to Control a Vision Sensor Pose in Multi-Agent Systems*. Tech. rep. Final Project Report for Mobile and Intelligent Robotics Course. DETI - UA, 2012.

[15] Tully Foote, Eitan Marder-Eppstein, and Wim Meeussen. *tf Package Summary*. URL: `www.ros.org/wiki/tf`.

[16] willow garage. *navigation Package Summary*. URL: `www.ros.org/wiki/navigation`.

[17] willow garage. *ROS.org*. URL: `www.ros.org/wiki`.

[18] Patrick Goebel. *Robot Cartography: ROS + SLAM*. URL: `www.pirobot.org/blog/0015/`.

[19] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. *GMapping*. URL: `www.openslam.org/gmapping`.

[20] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. "Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters". In: (2006).

[21] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. "Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling". In: ().

[22] *Kinect Image Transformations*. URL: `www.openkinect.org/wiki/Imaging\_Information`.

[23] S. Koenig et al. "Incremental Heuristic Search in Artificial Intelligence". In: *Artificial Intelligence Magazine* (2004), pp. 99–112.

[24] S. Kohlbrecher et al. "A Flexible and Scalable SLAM System with Full 3D Motion Estimation". In: *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE. 2011.

[25] Robert Laganière. "OpenCV 2 Computer Vision Application Programming Cookbook". In: Packt Publishing, 2011. Chap. 9 - Estimating Projective Relations in Images, pp. 233–242.

[26] Maxim Likhachev and Dave Ferguson. "Planning Long Dynamically-Feasible Maneuvers for Autonomous Vehicles". In: *International Journal of Robotics Research* (2009).

[27] Artur Merke, Stefan Welker, and Martin Riedmiller. "Line Based Robot Localization under Natural Light Conditions". In: ().

[28] Andreas Nuechter et al. *SLAM6D - Simultaneous Localization and Mapping with 6 DoF*. URL: `www.openslam.org/slam6d`.

[29] L. Pedersen et al. "A Survey of Space Robotics". In: ().

[30] *Robotic Operating System*. URL: `en.wikipedia.org/wiki/Robot_Operating_System`.

[31] *ROS Launch*. URL: `www.ros.org/wiki/roslaunch`.

[32] *ROS Master*. URL: `www.ros.org/wiki/Master`.

[33] *ROS Messages*. URL: `www.ros.org/wiki/Messages`.

[34] *ROS Nodes*. URL: `www.ros.org/wiki/Nodes`.

[35] *ROS Services*. URL: `www.ros.org/wiki/Services`.

[36] *ROS Topics*. URL: `www.ros.org/wiki/Topics`.

[37] Anthony Stentz. “The Focussed D* Algorithm for Real-Time Replanning”. In: *Proceedings of the International Conference on Robotics and Automation* (1995), pp. 3310–3317.

[38] Anthony Stentz and Dave Ferguson. “Field D*: An Interpolation-based Path Planner and Replanner”. In: ().

[39] Sebastian Thrun, Dieter Fox, and Wolfram Burgard. “Probabilistic Robotics”. In: The MIT Press, 2005.

[40] Sebastian Thrun et al. “Stanley: The Robot that Won the DARPA Grand Challenge”. In: *Journal of Field Robotics* (2006), pp. 684–685.

[41] *URDF - Joint*. URL: `www.ros.org/wiki/urdf/XML/joint`.

[42] C. Urdiales et al. “A hybrid architecture for autonomous navigation in dynamic environments”. In: ().

Appendix A

Robotic Operating System

ROS is a software framework, primarily developed by Willow Garage, to help software developers create to robot applications. It presents middleware features providing operating system functionalities like hardware abstraction, device drivers, graphic tools, debugging tools, libraries of software commonly used, message-passing between processes and package management[17][30]. ROS is manly developed for Unix systems, being Ubuntu supported.

Besides software framework and middleware, ROS is a development platform for robotic applications providing the two main features expected from a development platform, the development environment and the library suite. The development environment, is the aforementioned operating system like environment that allows easy software development with a simple compiling system, the workspace environment, the message-passing system between processes, monitoring, launching and other features. As for the library suite, this is implemented by *ros-pkg* that is composed of packages provided by ROS users and developers that are organized into stacks, forming a file system hierarchy that organizes software developed under ROS.

Besides all the tools and packages ROS provides, one of the most important features of ROS is the possibility of abstraction from some basic and low level aspects of software development. ROS provides a communication system between processes that can be easily used for implementing a distributed system with multiple processes. Using the implemented communication framework, the system is based on a graph like architecture. Nodes are responsible for the processing and these may receive, send and multiplex sensor and actuator information, planning, state and other messages.

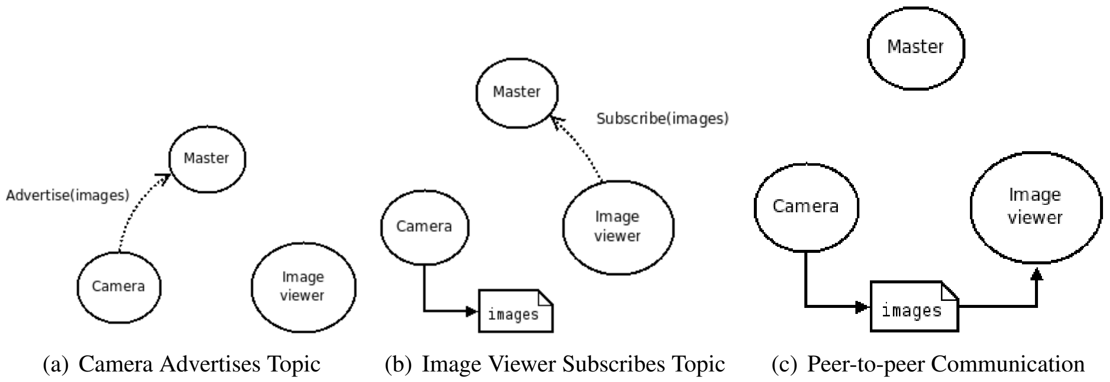
A.1 Publish-subscribe Architecture

ROS is a Message-oriented Middleware (MOM) that implements the publish-subscribe pattern. The MOM is an infrastructure designed to support message passing between distributed systems, being these modeled in hardware or software. This type of infrastructure allows abstraction for communication between systems and thus reduces the complexity on developing applications that use multiple systems, languages or network systems[7]. In the case of the MOM implemented by ROS, it provides the APIs necessary for using this message passing system across diverse platforms and languages.

The publish-subscribe pattern is a part of the MOM architecture implemented over ROS. The implemented architecture is a topic-based system where the sender of the messages, called publisher, sends the message through a topic, without knowing if there is a receiver, named subscriber, listening to the topic. The sender does not program the message that is sent through the topic, the messages are defined through classes without the knowledge of what, if any, subscriber that may be[5]. The same happens to a receiver that will use the classes characterizing the messages and receive the messages of interest without the knowledge of what, if any, publishers that may be[5].

The publisher-subscriber pattern is similar to the message queue paradigm, which is also supported by the MOM implemented by ROS. This implementation allows to have asynchronous communication between publishers and subscribers since the messages sent over the topics are stored in containers that the publisher and subscribers are unaware of. Also, using this type of pattern, and as happens with ROS, it is possible to create a server-client communication, where requests and responses may be sent asynchronously between the client and the server.

To coordinate all these components, ROS uses a node in its graph structure, the ROS Master[1]. This master node of ROS is launched within a ROS system automatically, and this node coordinates and tracks all publishers, subscribers and services within the ROS system[32]. The master node enables other individual nodes to locate each other and once they located each other, these communicate peer-to-peer[32]. This situation is represented in Figure A.1. Also, the master node is responsible for



(a) Camera Advertises Topic (b) Image Viewer Subscribes Topic (c) Peer-to-peer Communication

Figure A.1: Representation of the Role of the Master node[32]

naming and registering other nodes within the ROS system. Another function of the master node, is the maintenance of the parameter server. The parameter server, as its name implies, is a server

[1]www.ros.org/wiki/Master

containing all parameters defined in the ROS system. This functionality allows to redefine node parameters dynamically in run time.

With the ROS implementation and paradigm aforementioned it is possible to understand and identify the several components inherent and comprised with a ROS base software implementation. Each of this components are detailed and explained in the following sections.

A.1.1 Messages and Topics

Publishers and subscribers communicate sending and receiving messages[33] respectively. The message, as explained, is implemented by a class which defines a certain data type. Using ROS, the class which implements the message is generated automatically from a simple text file with the *.msg* extension where the message is defined using a metalanguage.

The definition of a message over ROS supports natively a wide range of variable types, which are then used to create more complex data structures and messages. This definition system supports the basic data types, float, char, integer, and other types as well as its arrays with or without a predefined size. It is possible also, to define a message using simple data types and other messages. With this type of definition for the messages, it is possible to create any kind of message to implement any communication need.

ROS already has defined several messages that can be used within user defined messages or as singularities, it has messages defined for image, action commands, sensor data and many others. One particular message implemented within ROS is the header message which is a special type of message. This message contains tracking information of the message containing the header, it contains a time stamp, a sequence id the frame which the message is associated with.

Listing A.1: sensor_msgs/Image.msg Message Definition

```
Header header
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

In Listing A.1 is shown an example of a message definition. The example is a message already defined and available in ROS. In contains a field header, explained earlier, which is a message by it self, and it contains simple basic data type to define the information it self, that in this case, is an image.

The publishers and the subscribers need to a have a support for communication, a channel. The communication channel is what defines the message type that the subscriber and publisher exchange, this channel is like a container for the messages. This container is called topic[36] within ROS.

The topic is the support for the communication between publishers and subscribers, the publisher will be publishing messages to the topic, and the subscriber will be watching the topic for messages received, making the topic a one way communication channel. Also, the topic do not know who is publishing to it and who is subscribing it, it completely detaches the production of the consumption of the information and can have multiple publishers and multiple subscribers attached.

When a subscriber or a publisher are instantiated, the topic which the subscriber or publisher will use as support is passed as argument. The topic is defined through a path allowing to have topics associated to the same kind of message and even with the same name but with different paths, enabling different connections for the same information, useful for multi robot environments for example. Since a topic is a communication channel, and it functions as a container for the messages passed between publisher and subscriber, this means that each topic can support one, and only one, message type.

A.1.2 Nodes

To hold the publishers and the subscribers, there are *nodes*. In ROS, a node is a process, a piece of software compiled into an executable, and each node may carry many subscribers and many publishers. Nodes are the core of the processing and software. Nodes use the publishers and the subscribers to communicate with each other through topics using the aforementioned communication system. Besides topics nodes may also communicate through services and the parameter server. To monitor the topics, the nodes implement callback functions associated to each topic, that may be called, using ROS API, when the topic associated has new messages to be processed.

Each node has a unique identifier, even if the nodes have the same executable name, they can be renamed to be unique in the These oscillations can cause errors in the odometry prejudicing the localization system. systems graph. The nodes in ROS allow modularity, each node has its specific function, publishing the result to other nodes, allowing to have a well defined structure for processing, and a chain to culminate in a final product, that typically in robotics, begins with sensor information and ends in action commands.

Another advantage of the use of nodes, and the underlying graph structure, is that in ROS nodes provide additional fault tolerance that isolate individual node crashes form the rest the graph. Also, with the modularity the nodes implement, the complexity of the software is reduced since the graph usually contains many nodes having each one a specific function. The offered modularity as well as the MOM implemented by ROS makes possible to implement nodes in different languages since the implementation details are hidden from the rest of the graph since nodes only expose minimal APIs[34].

The implementation of nodes may follow a synchronous or an asynchronous approach based on topics subscribed. This is possible using ROS API for publishers and subscribers. The node may function synchronously following the rate of the received messages or may function asynchronously consulting the topic subscribed at any given rate or even without a defined rate.

A.1.3 Services

Services are implemented and provided within nodes and are identified by a string name. The service is the ROS solution for the request-reply communication type. This type of communication is implemented over ROS because although the publish-subscribe paradigm is a very flexible one, often request-reply communication is required by distributed systems[35].

Like topics, services are defined by a specific message type that define the service. Instead of defining a message with the *.msg* extension is defined a message with the *.srv* extension. The particularity of this file, is that instead of having only a set of fields, *.srv* files have two sets of fields separated by a '—' line. The first set is the request part of the message, where data to be processed by the service is inserted, and the second part is the response, where the service will insert the response.

Listing A.2: Simple Server File for a Service to add two Numbers

```
int64 A
int64 B
———
int64 Sum
```

In Listing A.2 is shown a server message for a simple service that adds two numbers. Like messages for topics, the server files may have another messages in its fields, allowing to have complex servers, enabling complex services. Unlike publishers and subscribers, the service does not have a topic associated.

To use a service advertised by a node, one has to instantiate a container, the server generated by the *.srv* file, and call the service by using the name of the service, advertised using the container. The call for the service is blocking, and the node calling the service will block until a response is sent, and the response is received in the container used to call the service. The services are very useful for tasks that are clearly of the server type tasks. However the implementation of a service requires some cares, since the node using the service will block waiting for a response, and if the service is advertised but no response is sent, one may be blocked indefinitely waiting for a response.

A.1.4 Launch files

Another important part of software organization using ROS are the launch files. The launch files are XML files with a *.launch* extension. These files allow to easily launch an entire system using primitives to launch nodes and defining input parameters for the launched nodes directly[31].

Besides defining parameters for the launched nodes, the launch files also have primitives for publishing fixed tf frames, define environment variables, remap topics changing their names and even define name spaces allowing to launch complex systems at one launch file. It even allows launching complex systems with many nodes and topics using for this the inclusion of launch files within a launch file in order to have a better organization without having an extensive launch files.

A.2 Available Software

One of the advantages of using ROS is the software that is provided, and that can be directly used. ROS provides many packages for most of the robotic applications in nowadays. It has packages containing software for motion planning, whether if it is for an arm or for the robot it self, for sensor processing, action and many more. The organization of the stacks and its packages, allied to the *ros-pkg* system, allow to find and use the software available simply, providing even terminal commands for navigating within the ROS file system.

Because all software implemented over ROS is developed in nodes, it is easy to launch the nodes containing the desired algorithms and then publish the required topics to produce the expected results. For example, the *amcl*[2] package implements the MCL localization approach and one can use in a ROS system. To obtain the pose of the robot using this package it is necessary to have the map to be used by the main *amcl* node, publish the odometry information and publish the laser scan data. It is even possible, using the parameters of the *amcl* node, to use this it in an omnidirectional robot, or in a differential robot. The implementation already available of such packages allows the developers to work on other problems without having to create such algorithms from scratch. Also, all the available stacks are well documented, allowing to use them at *user* level and not at *developer* level.

Besides libraries and stacks of algorithms, ROS also provides software for interfacing with hardware. ROS has packages for interfacing with joysticks, vision sensors (eg Kinect), LRF and more, allowing hardware abstraction since the information from the hardware is directly available through ROS topics. This type of software allows to have hardware information published in the ROS system just by launching and using and using its API. Some of the available software for device drivers even implemente filtering function, allowing to have a high level of abstraction from the hardware.

A.3 Transformation Frames

The transformation frames (tf) package is one of the most useful and important ones of ROS. The tf package provides the possibility of keeping track of the various coordinate frames of the robot over time [15]. A tf tree, is the tree that represents how the frames of the robot are connected from parent

[2]`www.ros.org/wiki/amcl`

frame to child frame, in which each frame has its own coordinate system.

The tf is implemented as a topic although it is a complex topic, and offers an API of its on. The API provided by the tf allows to publish and subscribe to tf, using broadcast and listening methods respectively. The broadcast and listening operations are confined to a specific transformation, meaning that a broadcast operation publishes only a transformation between two frames and the listening operation only provides a transformation between frames. Just like topics, the tf may have multiple listeners and multiple broadcasters, although it is typically to have only one broadcaster for the tf tree in a ROS system.

By keeping track of the coordinate frames, the tf package provides functions to transform sensor data represented in a coordinate frame into another coordinate frame indicating only the names from which frame to which frame the transformation is required. Also, the tf provides functions to obtain the state of each frame relatively to another frame. For example, if one wants to know the position of the robot, represented as *base_link* frame, in the map, represented as *map* frame, one may use the tf package to instantiate a *transform_listener* from *map* frame to *base_link* frame, allowing to ask for the transformation, and consequently the pose of the robot in the map, at any given time, provided that the given tf and frames are available within the tf tree. The same applies if one wants the know the Kinect pose relatively to the robot asking for the transformation from *base_link* frame to *camera_link* frame, obtaining thus the Kinects pose relatively to the robot.

Using the tf package, it is relatively easy to transform from any coordinate system to another, and transform sensor data accordingly, provided that a consistent and correct URDF model is provided as well as the tf tree generated by it. For this, all the required frames have to be represented in the tf tree and the position of each joint has to be periodically published.

The tf package also provides tools[3] for monitor and debug the tf tree and the transformations. For example, as shown in Figure A.2, one can use Rviz to visually debug the tf tree, evaluating if the coordinates of each frame are in the correct place of each frame, and if the tf axis moves accordingly to the frames movement.

The tf also offers command line tools for monitoring the transformations in the tf tree. For example, the *tf_echo* tool allows to monitor the transformation between any two frames within the tf tree, with the command:

```
$ rosrun tf tf_echo \map \base_link
```

will print the transformation between the map frame to the base link frame, allowing to debug the transformation.

The tf also provides a tool to see all the transformations available and debugging information for each one. The *view_frames* tool will generate a pdf file containing the tf tree represented in a graphical way, and for each transformation between frames the file will contain the information on who is broadcasting this transform, the rate of the publishing, the time of the last frame published and

[3]www.ros.org/wiki/tf/Debugging%20tools

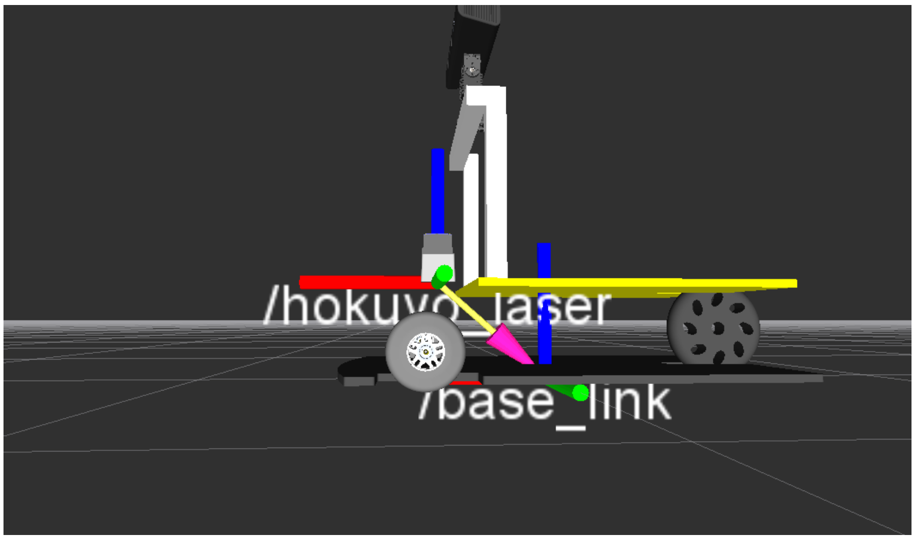


Figure A.2: Debug tf Using Rviz

also the time length of the transformation buffer.

A.4 Hardware Models and Abstraction

From the many libraries and tools available in ROS, the ones providing hardware abstraction and support for interfacing and modeling are worth to be particularly mentioned. As it was mentioned earlier ROS provides a set of libraries to implement device drivers between hardware and ROS directly. This software allows to use sensors directly over ROS without implementing other software.

Besides the abstraction, ROS allows to the user to define a model for the robot, specifying the kinematics and the physical model of the robot. The definition of the robot is a very importante part of the software, as it allows to have a complete robot description at software level which allows a certain degree of abstraction from the real hardware. Also, the description of the robot, allows to automatically publish all the necessary transformations between frames of the robot.

The definition of the robot model allows to have support for simulation. Simulation using robot description over ROS is directly supported by the Gazebo simulator [4]. Also, Gazebo has support for the software implemented over ROS as well, meaning that with an accurate robot description, one can test software without the need of the actual robot. Gazebo allows to built an environment model, which can be built with maps that were built by ROS algorithms with ROS interface. Also, it allows to have sensor models, many already available to use, meaning that one can simulate acquisitions of the environment. All this together, allows to fully test software over simulation helping developers test their software without the need for a robot.

[4] `gazebosim.org`

The description files use URDF[5] which is an XML format for describing a robot model. This format is held by a ROS package that offers a C++ parser for the descriptions files, allowing to generate the necessary code for the description and handling from the single XML file. The URDF format allows the user to define kinematics, physical, collision proprieties, inertial proprieties, joint dynamics and visual information of the robot. Using the URDF format, not only one has a physical and mathematical description but also visual description for direct debugging from the simulator or even from the real robot using this model for visualization in rviz. As an example, for defining the physical and kinematic description one has primitives in the URDF formar like the Listing A.3 which in this case, defines the proprieties of a frame with its mass value and its rotational inertia matrix.

Listing A.3: Defining Inertial Proprieties with URDF

```
<inertial>
        <mass value="10"/>
        <inertia ixx="1.0" ixy="0.0" ixz="0.0"
           iyy="1.0" iyz="0.0" izz="1.0"/>
</inertial>
```

As for the visual elements, the URDF format supports primitives for defining spheres, cubes and many others with the respective center position and rotation. However, describing a complete robot with elementary forms may originate a long description file and also, it is not intuitive to build a visual model with XML code. To avoid long description files and the complexity of building a visual model from elementary forms, the URDF format allows to include as CAD files as visual elements. These included CAD files allow to have a more accurate robot description since its easy to built the robot in a CAD software rather than XML code, and also, the CAD files may include the physical and kinematic proprieties of the frame.

Another important part of the robot description files are the joint primitives. These allow to specify the links between robot frames, as in their position, which frames are linked through the specified joint, and to specify the movement performed by this joint.

The joint primitive has many primitives that may be included within it. The joint is defined by the parent frame and child frame which are the mandatory parameters of this primitive. However, it is possible to include in the primitive, its position and rotation, the axis in which it preforms movement, the maximum performed movement and even simulation parameters like maximum tension, maximum velocity and acceleration, friction parameters and others. Basely, the joint may be completely defined in physical terms and parameters and even its control parameters. Listing A.4 shows a URDF description of a joint containing some of the definable physical parameters.

[5] www.ros.org/wiki/urdf

Listing A.4: Example Definition of Joint in URDF[41]

```
<joint name="my_joint" type="floating">
      <origin xyz="0 0 1" rpy="0 0 3.1416"/>
   <parent link="link1"/>
   <child link="link2"/>

   <calibration rising="0.0"/>
   <dynamics damping="0.0" friction="0.0"/>
   <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
   <safety_controller k_velocity="10" k_position="15"
      soft_lower_limit="-2.0" soft_upper_limit="0.5" />
</joint>
```

Defining the joints correctly in the description file, allows to have a correct tf tree published within ROS using the *robot_state_publisher* node as it is explained in Chapter 4.

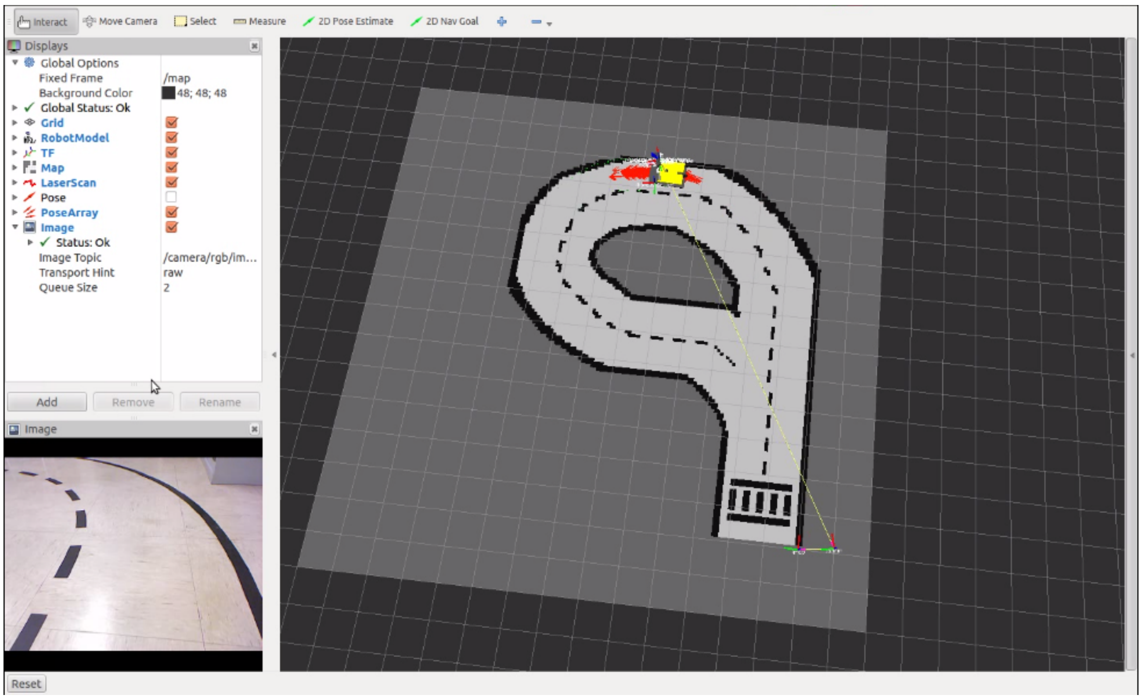
A.5 Debugging and Development Tools

ROS also has available a set of tools[6] for development and debugging in ROS environment, being these in the command line or even graphical.

One of the most used tools for debugging in ROS environment is indubitably Rviz[7]. Rviz is a graphical tool that allows to see almost everything that is going on in a ROS developed system. Since Rviz can display the information visually, it is possible to subscribe to topics and really see what is being sent over those topics. The debugging of a 3D environment, for example, only by numbers is very difficult, and Rviz allows to see everything the robot is seeing, including point clouds, laser scans, obstacles, etc. Also, Rviz has the notion of coordinate frames making it possible to show the information in every coordinate frame that is available in the tf tree of the system, allowing to immediately debug the tf tree and the transformations that are being made.

In Figure A.3 it is possible to see a typical usage of Rviz. In a typical usage of Rviz, it is possible to what the robot is seeing in the main window, as long as the Rviz is subscribing it. The subscription are set in the left window of the Rviz, and there they are shown allowing to set the subscription on or off. It is possible to see also the tf tree, being the origin of each frame drawn in its place. Also the tf tree is visible, opening the TF subscription in the left window of the Rviz. It is also possible to see any kind of processing information, as long it is published in a topic, like it is the case of the point cloud or the particle cloud draw in the main Rviz window.

[6]www.ros.org/wiki/Tools
[7]www.ros.org/wiki/rviz



Global Options	
Fixed Frame	/map
Background Color	■ 48; 48; 48
✓ Global Status: Ok	
Grid	☑
RobotModel	☑
TF	☑
Map	☑
LaserScan	☑
Pose	☐
PoseArray	☑
Image	☑
✓ Status: Ok	
Image Topic	/camera/rgb/im...
Transport Hint	raw
Queue Size	2

Figure A.3: Print Screen of a Rviz Usage for Debugging

Another useful graphical tool is the rxgraph[8] which allows to vizualize the graph for the entire running ROS system. It allows to visualize the nodes that are running and the connections between them showing also the topics involved.

ROS also provides a set of bash commands for simpler access to ROS files and packages. For example, the *rosed* command allows to edit any source file within any package with simple usage. With the command

```
$ rosed rota_base cb.cpp
```

the terminal will open the source file *cb.cpp* from the package *rota_base* with vim, without having to know the full path for the file, it is just necessary to know the package and the source file name.

A important tool that ROS provides for development is the *rosbag*[9] command. It allows to record any topic within the running system, and these can be played latter with no need for the robot. This tool allows to test algorithms without the robot, since the information can be recorded and the played, and more important, it offers repeatability that is very useful for testing algorithms.

The described tools and many more are available in all ROS distributions, and all of them are well documented in the `www.ros.org/wiki/Tools`.

[8]`www.ros.org/wiki/rxgraph`
[9]`http://www.ros.org/wiki/rosbag`

A.5.1 Bagging

An important tool provided by ROS is the *rosbag*[10] tool. The *rosbag* tool allows to record and play data within ROS environment. With this tool one can record data from ROS topics and use this data without having the robot. This tool is very important because it offers repeatability in the used data set which is very useful for testing algorithms. The *rosbag* tool is a command line tool, and as for the *groovy* distribution it as also a gui interface *rqt_bag*, that allows to record specific topics and play them later respecting the time stamps. For example, one can record the topic *road_scan* using the following command:

```
$ rosbag record \road_scan
```

This will generate a *.bag* file containing the information published on the topic for the time recorded. The *.bag* file containing the recorded data can be played later with the following command:

```
$ rosbag play ——clock recorded.bag
```

It will publish the topics recorded in the *.bag* file. The *–clock* is for correcting the time stamp in the messages, since they contain a latter date. For use this feature is important to set the *use_sim_time* parameter as true, in order to publish simulated times.

The *rosbag* also offers a code API allowing to set the action for recording topics within executables, which is very useful since without this is necessary to know the names of the topics and use the command line. Also, with this API one can choose from subscribing topics from a bag or available online within the executable, very useful for testing and debugging.

[10]www.ros.org/wiki/rosbag

Appendix B

Implementation Details

B.1 *rota_base*

B.1.1 TF tree for Zinguer robot

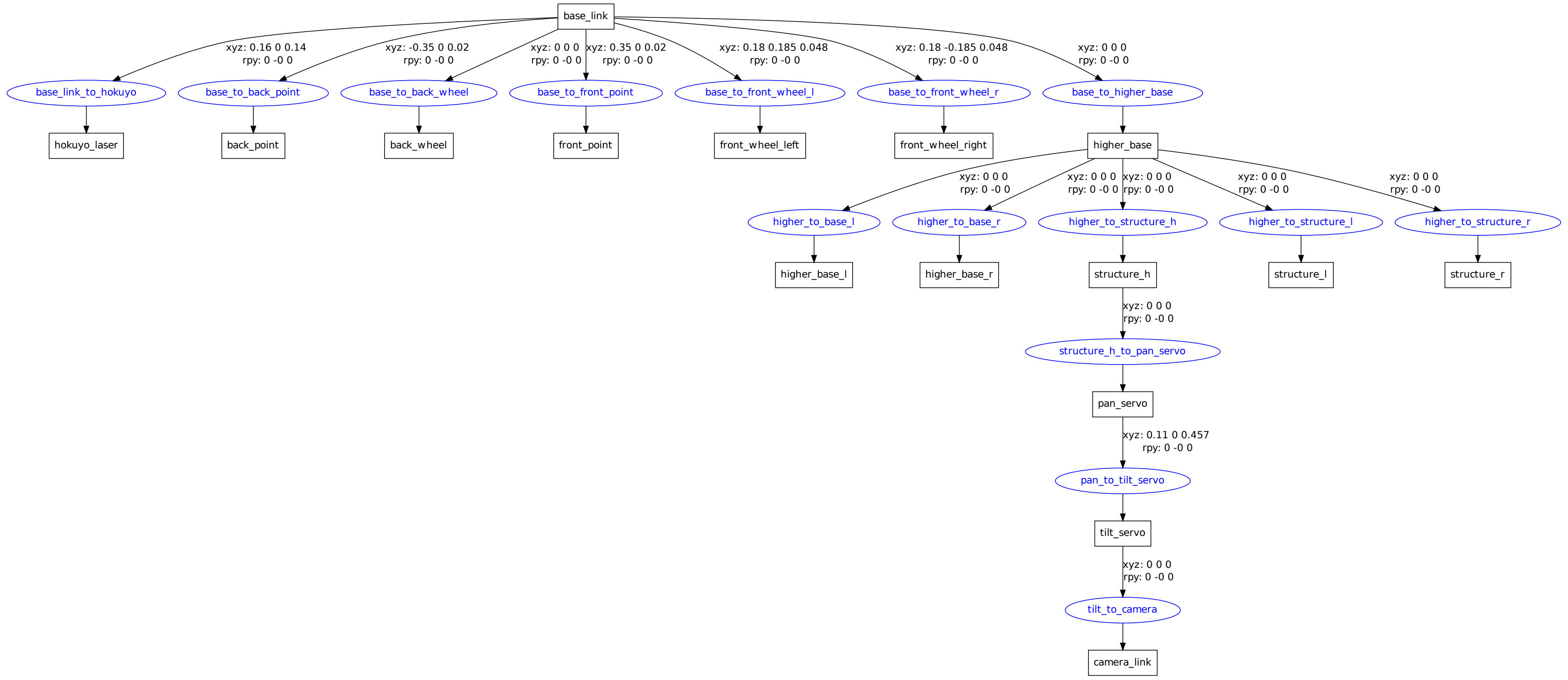


Figure B.1: Generated tf tree for Zinguer Robot

In the tf tree, Figure B.1.1, the boxes represent frames, each with its own coordinate system. The elipses represent joints that may or not be movable and the arrows correspond to links between frames. Each link has a coordinate system origin associated, in meters, and the starting point for each coordinate system is the first frame of the tf tree. The first frame may not have its origin of the coordinate system coincident to the $x = 0$, $y = 0$ and $z = 0$.

B.2 *rota_joy*

B.2.1 Building the Steering Map Table

In Table B.2.1 are shown the values measured for building the lookup table for mapping the values of the servo set point into steering angles. With the values from the Table B.2.1 a linear regression was

Set point	Perimeter (m)	α (rad)	Radius (m)
97	8.21452	0.317757	1.30738
94	8.57809	0.305126	1.36525
86	9.67054	0.272435	1.53911
73	11.0103	0.240458	1.75365
60	13.4302	0.198521	2.13748
51	17.3575	0.154415	2.76253
40	21.2742	0.126321	3.38589
-40	20.2881	-0.132391	3.22895
-51	14.7979	-0.180589	2.35515
-62	13.0134	-0.204679	2.07143
-72	10.3178	-0.256106	1.64212
-83	9.23368	-0.284654	1.46959
-100	7.76117	-0.334993	1.23523

made since it was the best fit for the data set. The regression done resulted in the graph represented in Figure B.2. The resulting graph shown in Figure B.2 has as function $y = 0.0033 \times x - 0.0067$. Although the function may turned as linear, if lower values of set point may been measured, the resulting function could be different and non linear. It was not possible to measure values lower than -40 for the set point due to the high radius it originates that would require a large room. The -100 and 97 set point values are the physical limits of the steering mechanical structure. It is possible to observe from the data set represented in Table B.2.1 that the mechanical set for the steering is not symmetrical in terms of limits since the maximum and minimum mechanical values are different.

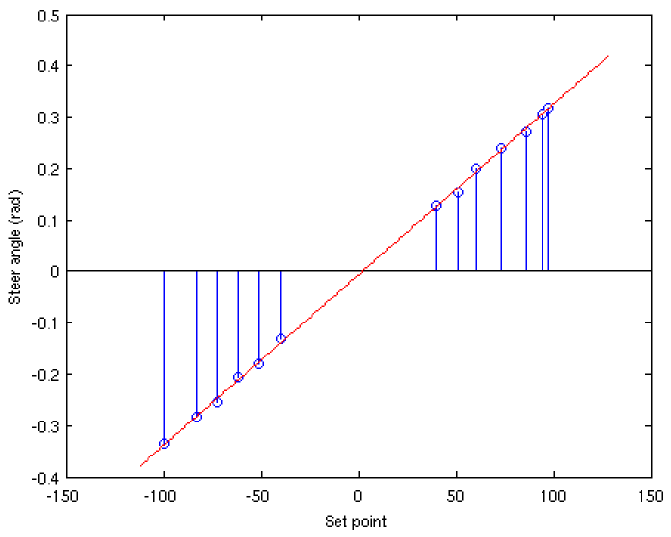


Figure B.2: Linear Regression from the Measured Data Set

B.3 rota_navigation

B.3.1 Motion Primitives

```
resolution_m: 0.050000
numberofangles: 16
totalnumberofprimitives: 176
primID: 0
startangle_c: 0
endpose_c: 1 0 0
additionalactioncostmult: 1
intermediateposes: 10
0.0000 0.0000 0.0000
0.0056 0.0000 0.0000
0.0111 0.0000 0.0000
0.0167 0.0000 0.0000
0.0222 0.0000 0.0000
0.0278 0.0000 0.0000
0.0333 0.0000 0.0000
0.0389 0.0000 0.0000
0.0444 0.0000 0.0000
0.0500 0.0000 0.0000
```

```
primID: 1
startangle_c: 0
endpose_c: 8 0 0
additionalactioncostmult: 1
intermediateposes: 10
0.0000 0.0000 0.0000
0.0444 0.0000 0.0000
0.0889 0.0000 0.0000
0.1333 0.0000 0.0000
0.1778 0.0000 0.0000
0.2222 0.0000 0.0000
0.2667 0.0000 0.0000
0.3111 0.0000 0.0000
0.3556 0.0000 0.0000
0.4000 0.0000 0.0000
primID: 2
startangle_c: 0
endpose_c: −1 0 0
additionalactioncostmult: 5
intermediateposes: 10
0.0000 0.0000 0.0000
−0.0056 0.0000 0.0000
−0.0111 0.0000 0.0000
−0.0167 0.0000 0.0000
−0.0222 0.0000 0.0000
−0.0278 0.0000 0.0000
−0.0333 0.0000 0.0000
−0.0389 0.0000 0.0000
−0.0444 0.0000 0.0000
−0.0500 0.0000 0.0000
primID: 3
startangle_c: 0
endpose_c: 8 1 1
additionalactioncostmult: 2
intermediateposes: 10
0.0000 0.0000 0.0000
0.0452 −0.0000 0.0000
0.0904 −0.0000 0.0000
```

```
0.1355 −0.0000 0.0000
0.1807 0.0008 0.0488
0.2257 0.0045 0.1176
0.2703 0.0114 0.1864
0.3144 0.0213 0.2551
0.3577 0.0342 0.3239
0.4000 0.0500 0.3927
primID: 4
startangle_c: 0
endpose_c: 8 −1 −1
additionalactioncostmult: 2
intermediateposes: 10
0.0000 0.0000 0.0000
0.0452 0.0000 0.0000
0.0904 0.0000 0.0000
0.1355 0.0000 0.0000
0.1807 −0.0008 −0.0488
0.2257 −0.0045 −0.1176
0.2703 −0.0114 −0.1864
0.3144 −0.0213 −0.2551
0.3577 −0.0342 −0.3239
0.4000 −0.0500 −0.3927
```