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

Robust Odometry, Localization and Autonomous Navigation on a Robotic Wheelchair

Relatore: Ing. Matteo Matteucci

Correlatori: Ing. Davide Cucci
Ing. Giulio Fontana

Tesi di Laurea di:
Luca Calabrese, matricola 783214

Anno Accademico 2012-2013

A mio nonno, che porto sempre nel cuore

Abstract

This thesis concerns autonomous robotics, a branch of robotics that deals with the study and design of vehicles able to fulfil tasks without the need for human intervention. In particular, a software system for a robotic powered wheelchair intended for people with motor disabilities has been designed and implemented on a prototype, previously developed by Politecnico di Milano. The aim of the thesis was providing the wheelchair with autonomous features like path planning and collision avoidance, while keeping it safe for both users and people around it. A widely used framework for robotic applications, named ROS (*Robot Operating System*), has been adopted. With its publish-and-subscribe paradigm and high portability, this framework improves extensibility and reuse of software modules. Moreover, the issue of robot localization has been studied. To this end, a new library for multi-sensor fusion and pose estimation, ROAMFREE (*Robust Odometry Applying Multisensor Fusion to Reduce Estimation Errors*), has been used. ROAMFREE fusion engine allows to merge odometry data coming from different sensors, in order to provide an estimate for the robot pose which is robust, meaning that it is less prone to errors. This method has been combined with an algorithm known in literature as AMCL (*Adaptive Monte Carlo Localization*), in order to increase the robustness of the estimate by compensating, in many cases, the absence of absolute position sensors. The robot has been tested in different situations, involving static and dynamic obstacles (such as objects and people), and has shown good performance in both cases.

Sommario

La presente tesi riguarda la robotica autonoma, un ramo della robotica che si occupa dello studio e della progettazione di veicoli in grado di eseguire compiti senza il bisogno dell'intervento umano. In particolare, un sistema software per una carrozzina robotica per disabili motori è stato progettato e implementato su un prototipo realizzato in precedenza dal Politecnico di Milano. Lo scopo di questa tesi è stato dotare la carrozzina di funzionalità di guida autonoma, come la pianificazione di percorsi e l'evitamento delle collisioni, mantenendola sicura sia per gli utenti sia per le persone che la circondano. È stato adottato un framework largamente usato in applicazioni robotiche: ROS (*Robot Operating System*). Grazie al suo paradigma di tipo publish-and-subscribe e alla sua alta portabilità, questo framework migliora l'estensibilità e il riuso dei componenti software. Inoltre si è studiato il problema della localizzazione. A questo proposito è stata adottata una nuova libreria per la fusione multi-sensoriale e per la stima della posizione, chiamata ROAMFREE (*Robust Odometry Applying Multisensor Fusion to Reduce Estimation Errors*). Il motore di fusione di ROAMFREE permette di fondere i dati odometrici provenienti da più sensori, in modo da fornire una stima della posizione del robot che sia robusta, ovvero meno soggetta ad errori. Questo metodo è stato combinato con un algoritmo noto in letteratura come AMCL (*Adaptive Monte Carlo Localization*), in modo da irrobustire la stima compensando, in molti casi, l'assenza di sensori di posizionamento assoluto. Il robot è stato testato in diverse situazioni, in cui sono stati coinvolti sia ostacoli statici sia dinamici (come oggetti e persone), e ha mostrato buone prestazioni in entrambi i casi.

Ringraziamenti

Desidero ringraziare innanzitutto il Prof. Matteo Matteucci per avermi offerto l'opportunità di occuparmi di questo progetto così importante, e per aver confidato fin da subito nelle mie capacità. Ringrazio inoltre gli ingegneri Davide Cucci e Giulio Fontana per aver seguito da vicino le diverse fasi del lavoro, e per avermi insegnato ad affrontare ogni problema con approccio critico e costruttivo. Grazie anche ai ragazzi conosciuti in AIRLab, che hanno alleggerito anche le giornate più faticose.

Giunto alla conclusione del mio percorso di studi, non posso fare a meno di ringraziare i miei genitori, per l'affetto che mi donano tutti i giorni e per avermi supportato in ogni momento della mia vita. Grazie a mia sorella Laura, per la quale questo è un anno molto speciale, e al mio futuro cognato Carlo. Grazie anche a mia nonna, che fa sempre il tifo per me. Un ringraziamento speciale va a Martina, che ha reso questi anni bellissimi e indimenticabili.

Ringrazio tutti coloro che hanno condiviso con me questo cammino, in particolare Alberto, Alessandro, Francesco, Jacopo e Riccardo, compagni di giochi e momenti spensierati, oltre che di studi. Un grazie va anche ai miei amici "storici", Andrea e Walter, e alle compagnie con cui trascorro le ore di svago. Ringrazio inoltre Fabrizio, il mio insegnante di basso, per avermi fatto capire come lo studio della musica sia non solo dilettevole, ma anche formativo in qualsiasi ambito.

Grazie a tutti coloro che hanno creduto e che credono in me.

Luca

Contents

List of Figures

Chapter 1

Introduction

This thesis concerns autonomous robotics, a branch of robotics that deals with the study and design of vehicles able to fulfil tasks without the need for human intervention. In particular, a software system for a robotic powered wheelchair intended for people with motor disabilities has been designed and implemented on a prototype, previously developed by Politecnico di Milano, named LURCH (*Let Unleashed Robots Crawl the House*).

The aim of the thesis was providing the wheelchair with autonomous features like path planning and collision avoidance, while keeping it safe for both users and people around it. A widely used framework for robotic applications, named ROS (*Robot Operating System*), has been adopted. With its publish-and-subscribe paradigm and high portability, this framework improves extensibility and reuse of software modules. Moreover, the issue of robot localization has been studied. To this end, a new library for multi-sensor fusion and pose estimation, ROAMFREE (*Robust Odometry Applying Multisensor Fusion to Reduce Estimation Errors*), has been employed. ROAMFREE fusion engine allows to merge odometry data coming from different sensors, in order to provide an estimate for the robot pose which is robust, meaning that it is less prone to errors. This method has been combined with an algorithm known in literature as AMCL (*Adaptive Monte Carlo Localization*), in order to increase the robustness of the estimate by compensating, in many cases, the absence of absolute position sensors.

The present work has been done in order to respond to the usability limitations of powered wheelchairs already on the market; some people, indeed, suffer diseases that hinder the normal usage of such wheelchairs, like cerebral palsy and Parkinson. Over the years, scientific research has tried to overtake those limitations, focusing on the development of wheelchairs with features comparable to those of mobile robots. Some of those prototypes are basically mobile robotic platforms able to transport a person; other projects use commercial wheelchairs as a base, and add robotic features while keeping the structure and the functionalities already present on those vehicles. LURCH belongs to this second category: a commercial wheelchair was extended, including a set of sensors (among which laser scanners and wheel encoders) that let it perceive the environment, and a circuit that allowed to communicate with the motors through a computer.

Although the first version of LURCH software system was able to show the potentialities of an autonomous wheelchair, its development has been dropped for the following reasons. First of all, the wide and fast diffusion of ROS over recent years has set up a new de facto standard in robot software development. This framework has the advantage of being easy to integrate with new technological solutions, such as ROAMFREE, and to test them more efficiently, thanks to the large number of tools provided. Moreover, ROS already includes modules that deal with typical robotic applications, and that have been tested on a wide variety of platforms; thus, using those modules increases the robustness of the system in general. The addition and changing of software components is much simpler, allowing to improve performances in future developments. Another reason was the need for creating an architecture that could be easily transferred to other platforms, because some functionalities, such as localization and motion planning, are common to most mobile robots.

The work done in this thesis can be divided into four logical parts. In the first part, modules that read sensory data have been implemented, giving attention to the estimation of time stamps, since they are critical in computing velocities; the obtained odometry information has been used along with ROAMFREE to retrieve pose estimates. In the second part, we focused on the communication with the motors, and on the realization of a PID software module for speed control. The third part involved command devices and assisted drive, and it has faced the addition of obstacle detection and collision avoidance features to manual drive. The last part has focused on autonomous navigation; to this end, ROS navigation modules have been used and configured. The AMCL package has been added in the architecture, in order to enhance the pose estimation process by integrating the map of the environment as a reference.

The robot has been tested in different situations, involving static and dynamic obstacles (such as objects and people), and has shown good performances in both cases.

The structure of this thesis is the following:

- In Chapter 2 some examples of robotic wheelchairs proposed in literature are illustrated, and the previous software architecture of LURCH is described; moreover, the ROS framework and the ROAMFREE multi-sensor fusion engine are discussed.
- In Chapter 3 the current hardware configuration of LURCH is illustrated.
- In Chapter 4 the software architecture is described, along with the explanation of all design choices.
- In Chapter 5 experimental results are analysed and discussed.
- In Chapter 6 conclusions are made, and some possible extensions and improvements for the project are illustrated.

Chapter 2

State of the art

In this chapter we summarize the state of the art on robotic wheelchairs, and we describe technologies and methods adopted in the present work.

2.1 Robotic wheelchairs

Electric and motorized wheelchairs increase the moving possibilities of users. Those wheelchairs are usually driven through an on-board joystick, and are designed for people who have not enough strength to move a manual one or have difficulties in doing that. Nevertheless, a category of people who are not able to safely control even such wheelchairs still remains. Among those we can count people who suffer cerebral palsy, Parkinson and, more generally, diseases that alter the ability to perform accurate movements, or cause involuntary motions.

The need to extend the use of wheelchairs to disabled people who cannot drive a normal one has led many universities and research institutes to the development of wheelchairs with more functionalities than those currently on the market. There are basically two research directions in that sense:

- creating a mobile robot with a seat for a person;
- using a commercial wheelchair as a base, changing the control system with an "ad hoc" system, and/or adding devices to it.

The former is the first direction that had been followed, but it turns out to be expansive and it requires much more effort in the design and realization of the mechanical structure and motorization. The second approach is today the most used one. Indeed, in a commercial wheelchair the mechanical structure is already working and tested, and the same holds for the low-level control of the motors. In addition, the use of a commercial wheelchair as a base allows to keep all the functionalities that are not directly related to motion. Such wheelchairs are, in fact, already designed to support disabled people in the best way possible. For instance, the seat is designed taking into account ergonomics, and in most cases it is adjustable to fit the person's posture.

On a functionality level, robotic wheelchairs can be divided into two categories: semi-autonomous and autonomous.

- Semi-autonomous wheelchairs have the objective of assisting the user, avoiding obstacles and helping moving through narrow passages. The path to follow is decided and controlled by the user, who communicates it through available devices (typically, the joystick, but some special devices might exist). This approach is known as *shared autonomy*, and it is defined as the full or partial replacement of a function that has to be executed by the robot [1]. In this case, for the most part, the driving function is carried out by the user.

- Autonomous wheelchairs offer similar functionalities to those usually implemented on autonomous robots. The user has to specify a destination, or goal; the control software plans a path to reach the goal and then executes it. In order to accomplish the task it is necessary to know where the wheelchair is located with respect to a reference pose, and to have information on static obstacles in the environment. Sometimes it is possible to integrate static obstacles information with new data, like unforeseen obstacles, to correct or change the path previously computed.

Obviously, these two modes can be combined, letting the user choose between them at runtime.

2.2 Existing projects

Over the years many different wheelchairs with extended features have been developed. What follows is a list including some of them [2] [3]:

SMART ALEC (Stanford, USA, 1980-1990, Figure 2.1(a)) [4] [5] [6] was the first semi autonomous wheelchair. It was based on a modified commercial structure, equipped with encoders on its wheels and sonars for obstacle detection. The driving system was based on the position of the head of the user, detected by using sonars. Among the functionalities offered there were: obstacle avoidance, keeping of a predefined distance from walls (useful when moving along corridors), and target following.

Madarasz wheelchair (Arizona University, USA, 1986) [7] was the first full-autonomous wheelchair. It was equipped with sonars and vision systems for the identification of artificial landmarks.

Mister Ed (IBM, USA, 1990) [8] was basically a mobile robot with a seat on it. Its software was based on a subsumption architecture, where "simple" behaviours were combined to produce more complex and high-level behaviours. Some simple behaviours implemented were passage through doors, wall following, and target following.

VAHM (University of Metz, France, 1992-2004) [9] [10], acronym for *Véichule Autonome pour Handicapé Moteur*, was based on a mobile robot with a seat on it. The control architecture was made of three levels, and allowed autonomous navigation based on an internal map, obstacle avoidance and keeping of a constant distance from walls. The choice of the operating mode was left to the user. Another prototype was made, with similar features, but based on a commercial wheelchair [11].

NavChair (University of Michigan, USA, 1993-2002, Figure 2.1(b)) [12] was based on a commercial wheelchair with some changes to the motor control system. It provided collision avoidance and a group of specific behaviours, like passage through doors and wall following.

TinMan (Kipr, USA, 1994-1999, Figure 2.1(c)) [13] represents a series of prototypes for autonomous wheelchairs based on electric wheelchairs. The first prototype used a mechanical device to move the joystick, while the following projects used a commercial wheelchair as a base, modifying the electronic motor control system.

CCPWNS (University of Notre Dame, USA, 1994-2000, Figure 2.1(d)) [14], acronym for *Computer Controlled Power Wheelchair Navigation System*, allowed following paths which had been learnt previously by the system. It used landmark identification. No obstacle avoidance feature was provided.

SENARIO (Greece, 1995-1998) [15] was a modified commercial wheelchair; it allowed interaction between user's driving and an obstacle avoidance

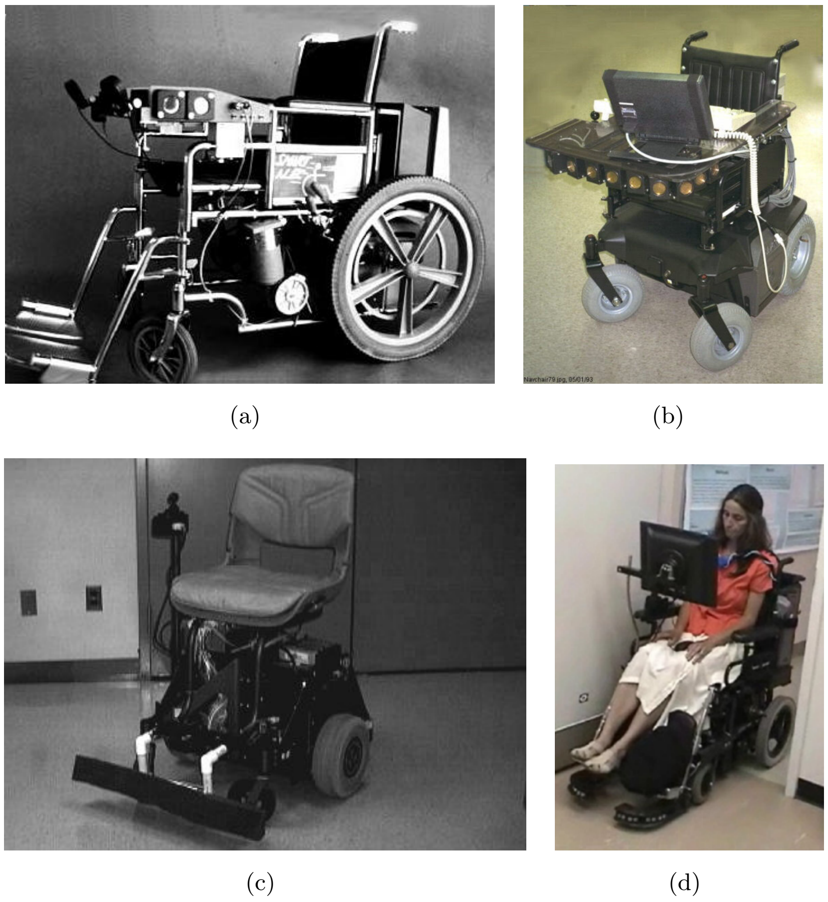


Figure 2.1: Some of the first prototypes: SMARTALEC (a), NavChair (b), TinMan 2 (c), and CCPWNS (d)

system. Autonomous navigation was managed with an internal map.

OMNI (University of Hagen, Germany, 1995-1999, Figure 2.2(a)) [16], acronym for *Office wheelchair with high Maneuverability and Navigational Intelligence*, was a modified omnidirectional electric wheelchair. Its features were organized in a hierarchical architecture: simple obstacle avoidance, custom behaviours for specific tasks and autonomous navigation.

Rolland I and II (University of Bremen, Germany, 1997-2002, Figure 2.2(b)) [17] [18] [19] have been the targets of many evolutions and studies. They were based on a commercial wheelchair (Meyra Genius 1.522) which already had interfaces for velocity commands and the measurements of the speed with encoders on the wheels. The first prototype provided autonomous navigation based on artificial landmarks and odometry, combined with collision avoidance through sonars, IR and bumpers. The operation modes were many and it was possible to follow previously learnt paths. In the second prototype only sonars were present, but the obstacle avoidance algorithm was more sophisticated. The basic behaviours to follow trajectories were in-place rotations and maintaining of constant distances from walls. In semi-autonomous mode the wheelchair adapted its velocity according to the presence of obstacles.

MAid (Germany, 1998-2003) [20], acronym for *Mobility Aid for elderly and disabled people*, was a commercial wheelchair with a custom motor control. It had two different operating modes: Narrow-Area Navigation (NAN) and Wide-Area Navigation (WAN). NAN mode allowed navigating from a starting pose to a goal, WAN mode allowed identifying and avoiding dynamic objects in the environment. At a later stage, the possibility of following moving obstacles was added. The odometry part was made out of a gyroscopic sensor and a couple of encoders mounted on the wheels.

Intelligent Wheelchair System (Osaka University, Japan, 1998-2003, Figure 2.2(c)) [21] interpreted head movements using a camera pointing toward the seat. A second camera, outward-facing, allowed object following and interpreting commands even when the user was not on board. The user's will was compared to the possibility of moving in a certain direction, basing both on a topological map and on measurements made with sonars. Localization was given by a system that integrated the odometric data. Thanks to the external camera, the system was able to recognize the presence of pedestrians and to move the wheelchair in order to avoid collisions.

Hephaestus (TRAC Labs, USA, 1999-2002) [22] was a kit for electric wheelchairs that implemented obstacle avoidance behaviours. It was compatible with many different wheelchair models and required no modification on the motor control system.

SIAMO (Alcala University, Spain, 1999-2003) [23] was a wheelchair equipped with many input devices, among which vocal control, head motion recognition, and ocular motion perceived with electrodes. It had an obstacle avoidance system with laser scanners and infra-red sensors which were also able to distinguish depressions and steep slopes. Localization was possible thanks to encoders on the motor wheels.

Smart Wheelchair (University of Kanazawa, Japan, 2000) [24] was able to locate itself by using radio beacons. Thanks to the knowledge of its pose, it could navigate autonomously. It could learn and reproduce paths thanks to the use of neural networks. It did not implement any obstacle avoidance system, but it was equipped with an odometry system on its wheels.

SPAM (Sciences, USA, 2003-2004, Figure 2.2(d)) [25], acronym for *Smart Wheelchair Assistance Module*, was developed as an accessory to motorize manual wheelchairs. It was compatible with many manual wheelchair models. The functionality provided was obstacle avoidance, and software architecture was based on behaviour rules.

Rolland III (Bremen University, Germany, 2005) [26] had two laser scanners placed at ground level, one facing forwards and the other one facing backwards. The wheels were equipped with encoders for odometry. The vision system was based on an omni-directional camera that perceived landmarks located in the environment.

In recent years research on autonomous wheelchairs has mostly focused on means to let disabled users interact with them. Furthermore, motion on irregular terrains has been studied, allowing wheelchairs to deal with steps and to move outdoors. Here we illustrate some recent examples.

BCW (National University of Singapore, 2007, Figure 2.3(a)) [27], acronym for *Brain Controlled Wheelchair*, is based on a commercial wheelchair, the Yamaha JW-I. It is equipped with optical encoders mounted on special small wheels, a proximity sensor installed on the front part and a barcode reader for localization. Control is made with a BCI interface.

MIT Wheelchair (Massachusetts Institute of Technology, 2008, Figure 2.3(b)) [28], is a wheelchair with a vocal interface that relies on Wi-Fi networks for indoor navigation, whereas obstacle avoidance is ensured by a SICK sensor together with a couple of Hokuyo laser scanners.

SILLA (University of Zaragoza, Spain, 2008) [29], is a wheelchair controlled with the brain by means of a BCI interface. It is equipped with encoders on its wheels for odometry, whereas obstacle avoidance relies on a SICK scanner placed between the user's legs.

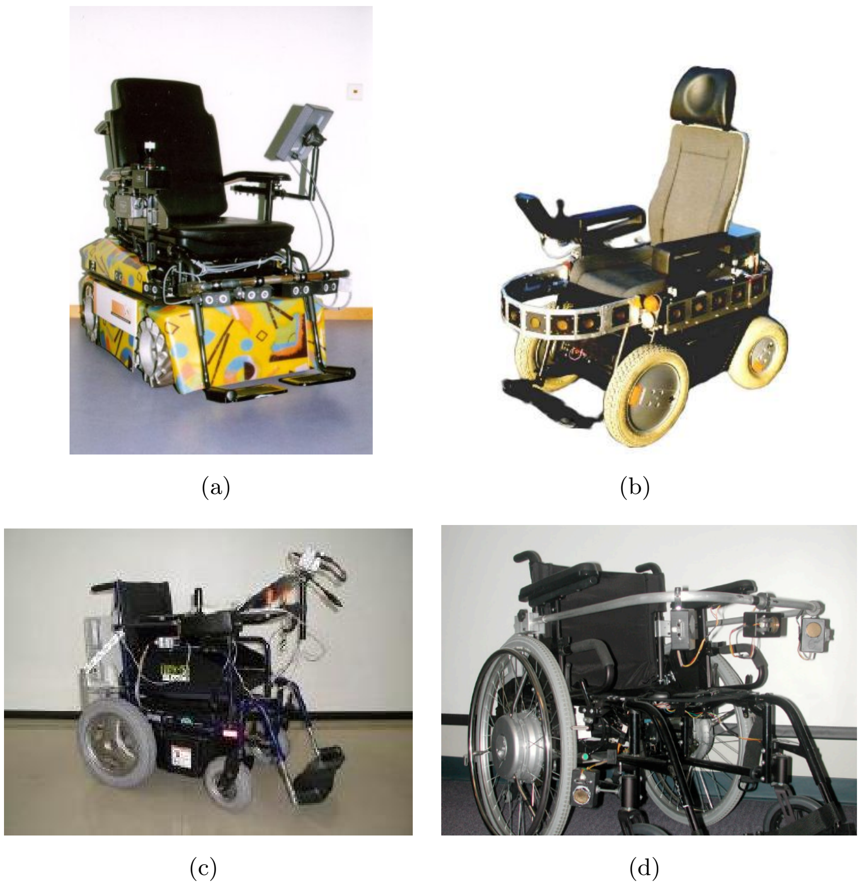


Figure 2.2: OMNI (a), Rolland II (b), Intelligent Wheelchair System (c), and SPAM (d)

TDS-PWC (Georgia Tech University, 2008, Figure 2.3(c)) [30], acronym for *Tongue Drive System-Powered Wheelchair*, takes advantage of a system known as Tongue Drive System to drive a commercial wheelchair by interfacing with the proprietary bus. The system uses a magnet placed on the user's tongue, whose movements are detected by some sensors located on a dedicated helmet.

RT-Mover (Chiba Institute of Technology, Japan, 2009, Figure 2.3(d)) [31] is a robotic wheelchair capable of moving through rough terrains. It uses a hybrid mechanism that combines the advantages of both wheeled and legged robots. To be more precise, it can change the roll of the wheel axes in order to lift wheels as they were legs, thus succeeding in passing through slopes and other irregularities of the terrain, while keeping the stability and

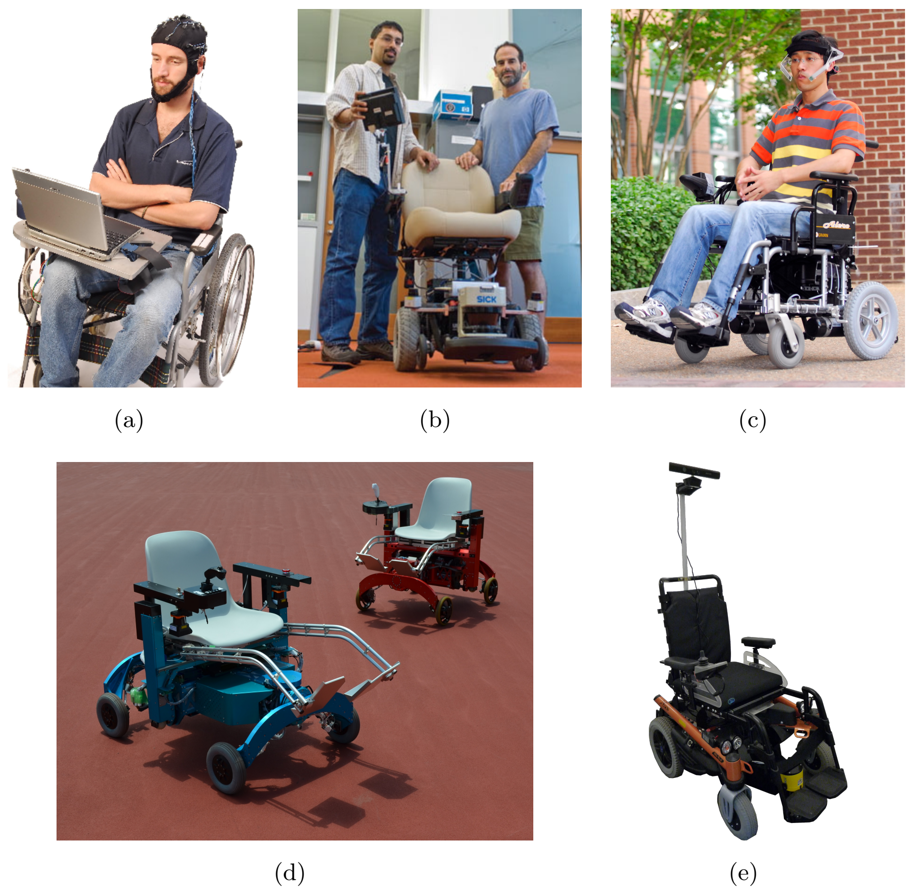


Figure 2.3: Some recent examples of robotic wheelchair: BCW (a), MIT Wheelchair (b), TDS-PWC (c), RT-Mover (d) and Intelligent Wheelchair (e)

minor complexity of a wheeled system. It also has a control mechanism that keeps the seat steady. It uses encoders and current sensors on the wheels in order to detect steps.

Wheelchair for users with cerebral palsy (University of Zaragoza, Spain, 2010) [32] is a wheelchair that has been specifically designed for people with cerebral diseases. It is based on a commercial wheelchair, and uses a planar laser and wheel encoders to perceive the environment. Users select goals through a touch screen showing a visual 3D representation of the wheelchair and the environment around it. The computer does not store any map: the map is dynamical, based on the current sensor readings, and always centred on the wheelchair.

Intelligent Wheelchair (Freie Universität Berlin, Germany, 2011, Figure 2.3(e)) [33] is a wheelchair that combines different input sources: it can be driven through a smartphone (*iDriver*), with an eye-tracking system (*eye-Driver*), or with a Brain Computer Interface (*brainDriver*). It also provides obstacle avoidance features. It is based on a commercial wheelchair.

IWS (University of Toronto, Canada, 2012) [34], acronym for *Intelligent Wheelchair System*, is an add-on for powered wheelchairs. Obstacles are detected by a stereo-vision camera; movements towards the obstacles are prevented. If the wheelchair remains stopped by an obstacle for a certain period of time (about 2 seconds), an audio prompt will be played, which helps the user navigate into free space surrounding the obstacle (e.g. "try turning left", "try turning right"). The audio prompt is repeated every 5 seconds if the wheelchair continues to remain stopped.

Among the prototypes listed above, we can identify some recurring features. For instance, most autonomous and semi-autonomous wheelchairs are equipped with more than one command interface. That is because of two main reasons: first of all, wheelchairs are often designed for many types of disabilities, so that the end user can choose the appropriate interfaces; in the second place, different situations can imply different input methods. As regards tasks for which a tutor is required, for example, the tutor may need to drive the wheelchair remotely, so another input device, different from the one adopted by the user, may be necessary to him/her. The need for many input possibilities entails a modular architecture both for hardware and for software: the addition of a device must be as transparent as possible to the rest of the system.

Another important aspect is that the security required for a robot that transports people is significant. In particular, it is important that the wheelchair is able to locate itself and detect obstacles in the best way possible, in order to avoid accidental collisions; for this reason, usually many different sensors are employed, and their data are merged to increase robustness.

2.3 LURCH

In 2007 the AIRLab (Artificial Intelligence and Robotics Laboratory) of Politecnico di Milano developed a robotic wheelchair called LURCH (acronym for *Let Unleashed Robots Crawl the House*) [2] [3]. It is based on a commercial wheelchair and implements both semi-autonomous and autonomous modes.

2.3.1 Main features

The first version of LURCH [2] was equipped with:

- 2 laser scanners in the front, each scanning the environment at 240 degrees;
- an IMU (Inertial Measurement Unit) to measure velocities;
- a camera that detects artificial landmarks;
- an on-board x86 computer;
- a touchscreen that helps the user interact with the computer.

Basically, the wheelchair could detect obstacles by means of the two laser scanners, localize itself with the help of the camera, and have a velocity feedback given by the IMU. In semi-autonomous mode, the wheelchair could be driven with the on-board joystick or a joypad alike; the software was able to facilitate the user's movement, avoiding collisions.

In autonomous mode, the user could select a goal position through the touchscreen; the control software had the task of computing the path to reach the goal and executing it by sending the proper commands to the motors. In order to make the computer give commands to the motors, the connection between the joystick and the motors was cut, and an interface circuit was realized and put in the middle. Such board has basically two functions:

- it reads the joystick positions and it sends them to the computer;
- it translates the commands sent by the computer into voltage values that control the motors.

A peculiarity of this solution is that the joystick still keeps its functionality, as its position can always be known and employed, and it can be used in conjunction with other input devices.

Over the years some changes to LURCH have been made. In particular [3]:

- the IMU was removed, for its low accuracy in velocity measurements and for its high costs;
- the velocity feedback is now given by encoders mounted on the wheels;
- an odometry board was made in order to interface the computer with the encoders.

As for the command interfaces, some different solutions have been tested, such as a wireless joypad, speech command, facial muscle control and brain-computer interface (BCI) [35]. A more detailed description of the current robot hardware is given in Chapter 3.

2.3.2 Previous software architecture

The control software was initially based on a framework known as DCDT (*Device Communities Development Toolkit*) [36]. Such framework facilitates the implementation of software agents that can communicate and exchange information. Each agent, also known as *member*, has the following lifecycle:

1. Initialization of the member.
2. Periodic repetition of the member specific function.
3. Closure of the member.

The desired behaviour is obtained by specifying the actions that the member has to follow at each step. Members communicate one another by exchanging messages. The communication is realized through TCP/IP, allowing two members to communicate even if they run on different machines. Messages have a type, in other words they are distinguishable according to a code. The mechanism that manages dispatch and reception of messages is based on a publish-and-subscribe paradigm. More specifically, each member specifies which messages it intends to receive, without the need to know which the sender is. This creates a quite flexible system that allows, for instance, to substitute a component with another that produces the same messages, leaving the other parts of the system unaware of the change; thus modifications in the receiving members are not required.

Another important feature implemented in the previous version of the software is BRIAN (acronym for *BRIAN Reacts by Inferring ActioNs*), a control system based on behaviours. It provides the choice of active behaviours and the fusion of results, basing on conditions defined in terms of state variables and sensory detections. A behaviour is represented as a set of fuzzy logic rules.

In the LURCH DCDT control software, members were called *experts*, since each of them was specialized in a particular task. The components were the following.

MotorExpert: it read raw data coming from the motor board, namely the position of the on-board joystick, and sent them to BrianExpert. At the same time, it transmitted data commands coming from BrianExpert to the motor board. After the addition of the encoders and the odometry board this module also published the total number of encoder ticks.

JoypadExpert: it mapped joypad buttons to commands to be sent to BrianExpert.

HokuyoExpert: it interacted with the laser scanners and sent information on the minimum distance from obstacles to BrianExpert. On request it also sent a list of points representing the shape of obstacles.

MTiExpert: it retrieved sensory data coming from the IMU and sent them to PoseExpert. It was removed after the addition of the encoders.

VisionExpert: it periodically acquired an image from the camera and computed, whenever possible, the wheelchair position with respect to the reference frame. The message was then sent to PoseExpert.

PoseExpert: it managed data coming from encoders and VisionExpert to compute the actual speed of the wheelchair, which was then sent to BrianExpert.

SpikeExpert: it is the path planning module. It waited for a planning request and returned a path, or a message that communicated the impossibility of building it. The chosen planner was Spike, which in turn used the A* algorithm to compute the path over a grid with cells of customizable size. The map could be dynamically updated.

SequencerExpert: it specified a starting point, a goal and information on new obstacles. It sent planning requests to SpikeExpert.

BrianExpert: it was the core of the whole architecture. It collected all data coming from the other experts, translated them into fuzzy variables, run the proper rules and returned commands for MotorExpert. The obstacle avoidance feature was treated as a high-priority behaviour, separated from the actual path planning, in order to make the wheelchair more reactive to dynamic obstacles.

GuiExpert: it managed the graphic user interface, which was used to display information and to set goals.

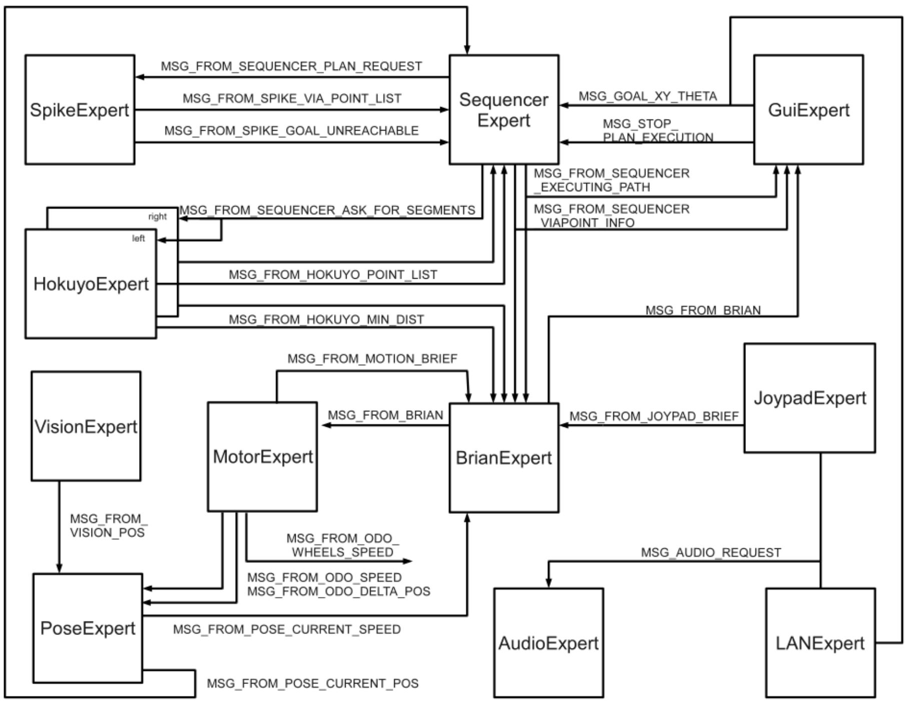


Figure 2.4: Software architecture of LURCH with DCDT [3]

ErrorExpert: its task was to detect and manage system failures by reading the messages sent by the other members.

Figure 2.4 shows the entire system architecture. `AudioExpert` was only used to reproduce multimedia contents, and `LANExpert` was predisposed to exchange messages through an Ethernet connection (used, for instance, by the BCI).

Although such system was able to demonstrate the potentialities of a wheelchair with extended features in a satisfactory way, there was the need of creating a system that was easier to extend and to integrate with newest technological solutions. Furthermore, the wide and fast diffusion of ROS framework over recent years, along with its benefits, has set up a new de facto standard in robot software development. This means that there is now the tendency to converge to a unique framework that allows to share solutions and to reuse components in other projects. The main advantages of ROS are described in the next section.

2.4 ROS

ROS (*Robot Operating System*) [37] is an open-source operating system for robots, developed by the Stanford Artificial Intelligence Laboratory and by Willow Garage. More precisely, it is a *meta-operating system*, as it provides a structured communication layer above a host operating system. Its aim is to provide a general framework, suitable for the most common use cases in robotic software development.

2.4.1 Basic structure

A system built using ROS is made of a certain number of processes, potentially on a number of different hosts, connected at runtime in a peer-to-peer topology. Those processes are called *nodes*. In a typical robot application, each node is responsible for a specific task, often related to a particular part of the hardware. On a conceptual level, nodes are the ROS equivalent of the members/experts used in DCDT framework (see Section 2.3.2).

Nodes communicate with each other by passing *messages*. A message is a typed data structure. Standard primitive types, such as integer, float, etc. are supported, but programmers can also create custom messages and combine different types to produce more complex messages.

A node sends a message by publishing it to a given *topic*. A node that is interested in a certain kind of data must subscribe to the proper topic. In general, many nodes publishing or subscribing to the same topic may exist, and a single node may publish or subscribe to multiple topics. As in DCDT architecture, publishers and subscribers are not aware of each others' existence.

Although this topic-based model, founded on the publish and subscribe paradigm, is very flexible and can be useful in many of the most common cases, it is not appropriate for synchronous transactions. To solve this problem, ROS provides the possibility to define a so-called *service*, that is a pair of messages, one for the request and one for the reply. This is similar to what happens on Web services, which have request and response documents of well-defined types.

In order to let processes locate each other at runtime there is a module called *master*, which provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics and services [38].

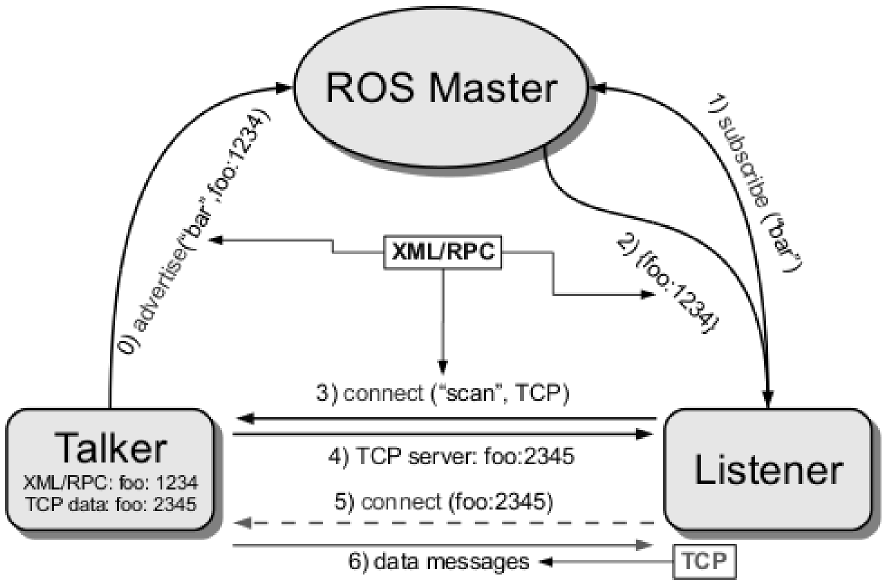


Figure 2.5: ROS basic structure [39]

2.4.2 Main properties

The structure, made of independent nodes and messages, improves the reuse and extensibility of software projects. In fact, the encapsulation of code, forced by this structure, makes it relatively easy to take a single node or a *package*, that is a set of nodes, from a project and put it into another project. The only required effort is to adapt the new project to the interface of the retrieved nodes, namely message types and topic names, without having to touch their inner code. For these reasons, many generic nodes are provided, by the ROS team or by the community of programmers, and can be used in many cases directly out of the box or with little tuning. Among those, there is a variety of drivers for the most famous or common devices for robots, like sensors and input devices. The growing diffusion of ROS as a standard for robot software developing has increased the number of available solutions for many typical problems.

Another important feature of ROS is that it allows communication between nodes written in different programming languages. This allows, for instance, to write some parts of the software in an interpreted language (like Python) to make those parts configurable and testable with less effort, and use more complex, compiled languages (like C++) to deal with tasks that have strict constraints in terms of time or memory consumption.

ROS is not a monolithic development and runtime environment. On the contrary, it is composed of many small tools, able to perform various tasks. All these tools can be run by means of bash commands, so they are integrated in the normal operating system usage. These tools allow to navigate the source code tree, get and set configuration parameters, run single nodes or sets of nodes, see which topics and nodes are running, visualize messages published on a topic, and so on. This modular structure is useful when debugging, specially if the scope of investigation is a single part of the project, such as a single node. In fact a node can be run, modified, and then rerun without having to restart the whole infrastructure: the graph composed by talkers and listeners is dynamically modifiable. ROS also provides specific tools for recording and playing back nodes, thus simplifying data analysis and research.

Among the tools provided by ROS, an important role is played by graphic tools. Programmers can plot data and visualize graphs containing nodes, topics and relations between them. There is also a complete tool for data visualization, called *rviz*. This tool allows to view maps, reference frames, landmarks, planned paths, sensor data, and so on.

In a nutshell, ROS allows to employ less effort in the coding and engineering parts, and to concentrate more on the core research.

2.4.3 The Navigation Stack

As we said above, ROS repositories already contain some packages that deal with the most common problems in robotics. One of them manages the autonomous navigation of the robot. The *Navigaton Stack* is a collection of packages for path planning, localization and environment mapping. Considering it as a whole, it takes information from odometry and sensors and outputs velocity commands to a mobile base [38]. Obviously, all packages needs to be configured for the kinematics, dynamics and shape of the specific robot.

Concerning the path planning and motion planning module, called `move base`, it is basically subdivided into two parts: a *global planner* and a *local planner*. The global planner has the task of building a path from starting pose to goal pose over the entire, static map of the environment. A local planner, on the other hand, reasons about a little portion of the environment, the one that immediately surrounds the robot, which is updated more often than the global map. It decides which action to choose from a set of possible actions in order to approach the global plan. The environment is represented as a grid, where each cell has a cost, which is higher in correspondence of obstacles and lower in free space. This cost is taken into account while planning: the planner tends to move in low-cost cells so, as a consequence, it makes the robot avoid obstacles.

The map of the environment can be directly provided (for instance, if the environment has a well-defined structure and a map is already available), or it can be built with a package called `gmapping`, which relies on the approach of the same name [40]. Given odometry data and distance measurements collected by a laser scanner, it solves the problem known in literature as SLAM (*Simultaneous Localization and Mapping*) to create a map.

The presence of a map can also be used to adjust the pose estimate given by odometry. The approach adopted by the ROS Navigation Stack is called *AMCL* (*Adaptive Monte Carlo Localization*), and it is described in [41]. Basically, it samples around the uncertain pose given by odometry data, and applies a weight to every sample. The weight contains information on the similarities between laser scans and static obstacles on the map. At each step the distribution of samples changes in accordance with their weights. In other words, the higher the weight, the higher the probability of drawing that sample. By doing so, after some steps the sample distribution tends to concentrate near the real pose of the robot.

Since there can be situations in which AMCL is uncertain on the robot position, such as when no map feature is detectable, or when many similar features are present, the accuracy of odometry data is important. Therefore AMCL has to be coupled with another localization system that combines the information given by sensors to produce a pose estimate that is as precise as possible.

2.5 ROAMFREE

In order to provide an estimate of robot location, in general it is necessary to combine data coming from multiple sources, like wheel encoders and laser scanners. Pose tracking, along with multi-sensor fusion, is one of the most important issues in mobile robotics: the effectiveness of autonomous navigation is strongly dependent on this factor.

ROAMFREE (*Robust Odometry Applying Multisensor Fusion to Reduce Estimation Errors*) [42] is a framework developed by Politecnico di Milano that has the double objective of fusing position measurements coming from an arbitrary number of sensors and solving the pose tracking problem on-line. Its aim is to provide a general approach, so it tries to abstract from the nature of the information sources. For this reason, it does not deal with physical sensors, but with *logical* sensors, based on their functionality, namely the kind of information they provide. In that way, the same physical

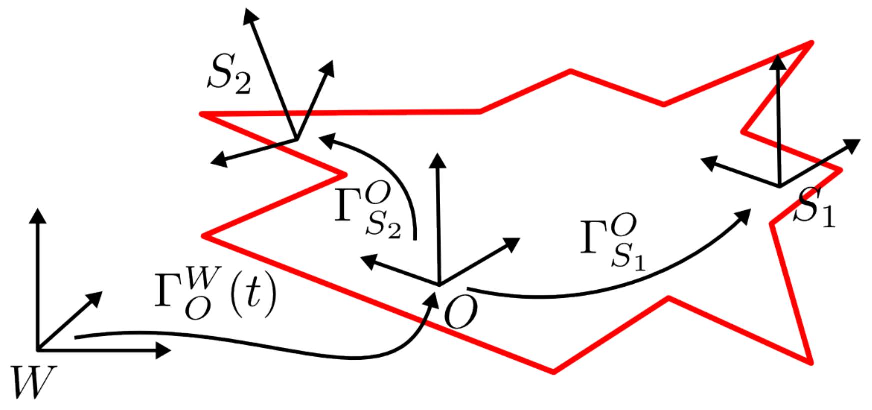


Figure 2.6: Reference frames and coordinate transformations in ROAMFREE

sensor can be associated to more than one logical sensor, and vice versa, one logical sensor can be composed of many physical devices that together provide the same kind of information.

The framework includes a library of sensor families, such as absolute position and/or velocity, angular and linear speed, acceleration and vector field (e.g. magnetic field). For each category an error model is defined that relates the state estimate with the measurement data, considering the most common sources of distortion, bias and noise. These models are taken into account during the estimation process, in order to handle the logical sensor readings according to their type.

ROAMFREE uses basically three reference frames: W, the fixed world frame, O, the moving reference frame, placed at the odometric center of the robot, and the i-th sensor frame, S_i. The tracking module estimates the position and orientation of O with respect to W, namely Γ^W_O (see Figure 2.6).

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

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

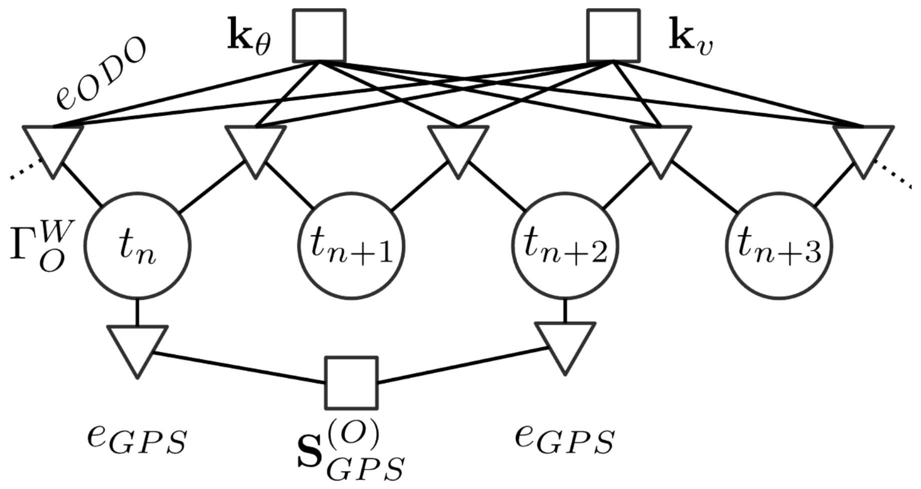


Figure 2.7: An instance of the hypergraph, with four pose vertices $\Gamma_O^W(t)$ in circles, odometry edges e_{ODO} (triangles), two shared calibration parameters k_v and k_θ (squares), two GPS edges e_{GPS} and the GPS displacement $\mathbf{S}_{GPS}^{(O)}$

error is computed through linearization:

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

where Σ_η is the covariance matrix of η and J_i is the Jacobian of e_i with respect to η.

The optimization problem is formulated as follows:

$$\mathcal{P}: \quad \underset{x}{\operatorname{argmin}} \sum_{i=1}^{N} \bar{e}_i(x_i)^T \Omega_{e_i} \bar{e}_i(x_i) \tag{2.2}$$

where $\Omega_{e_i} = \Sigma_{e_i}^{-1}$ and N is the total number of edges. If a reasonable initial guess for x is provided, a numerical solution to this problem can be found by means of non-linear least-squares methods such as Gauss-Newton or Levenberg-Marquardt algorithms.

In order to build the graph, first of all it is necessary to choose a high frequency sensor from which it is possible to predict $\Gamma_O^W(t+\Delta t)$ given $\Gamma_O^W(t)$ and its measurement $z(t)$. This sensor is called *master*. Each time a new reading for the master sensor is available, a new node $\Gamma_O^W(t+\Delta t)$ is instantiated, using the last pose estimate available, $\check{\Gamma}_O^W(t)$, and $z(t)$ to compute an initial guess for that node. $z(t)$ is also used to construct an odometry edge between poses at time t and $t + \Delta t$.

When a new measurement from a different sensor is available at time t, its corresponding edge is inserted between the poses that have timestamps nearest to t. The error introduced by this approximation is very low if the master sensor has a sufficiently high sampling rate. Since by increasing the graph size the time complexity of the optimization step increases, it is necessary to limit its scope, so a moving pose window is implemented, and only the last poses are considered.

An interesting feature of this framework is that it is also possible to include sensor parameters in the hyper-graph, in order to calibrate them on-line. This characteristic is useful when some parameters are unknown, or difficult to measure with accuracy. Some examples are sensor displacements with respect to the robot odometric center, wheel radius and axis length for encoders, scale factors and biases.

ROAMFREE exposes to programmers a simple Python interface, composed of functions that allow to add logical sensors, choose the master sensor, set the window size and the number of optimization iterations, select which parameters are going to be estimated, and so on. It can be wrapped in a ROS node that subscribes to sensor outputs and periodically publishes the current estimated pose.

Chapter 3

Robot configuration

What follows is a description of LURCH configuration prior to the current thesis work. The content of this chapter is the base upon the work has been done. The equipment of LURCH presented here derives from other works and theses, especially from [2] and [3].

3.1 Wheelchair

The wheelchair used as a base for the development of the system is *Rabbit*, produced by the German company Otto Bock. It is a wheelchair suitable for both indoor and outdoor environments, as it has two large rear wheels with tessellated tyres. The drive is realized by means of two independent motors that operate on each rear wheel, while the front wheels are free to turn and not controllable. The motors are about 200W each and the power source is given by two 12V - 70Ah serially-linked batteries located under the seat. The control system of the motors, along with the on-board joystick, is part of the DX System series produced by Dynamic Controls.

3.2 Sensors

Every robot needs sensors to perceive the environment and estimate its state. LURCH is equipped with several sensors, and here we describe them.

3.2.1 Laser scanners

In order to perceive the environment in the most precise and dependable way, the robot is equipped with two laser scanners; the model is Hokuyo URG-04LX. These sensors are able to perceive the range of obstacles on a

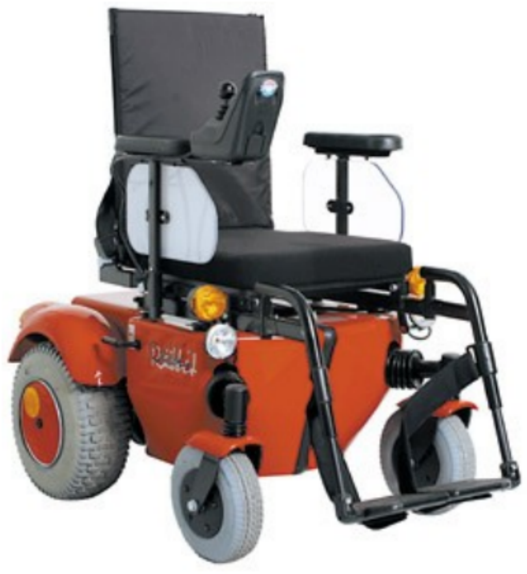


Figure 3.1: The Otto Bock Rabbit wheelchair

plane, with a field of view of 240° and a resolution of 0.36°. The maximum detectable distance is 5.6 m. The wiring of the laser scanners to the computer is made of a USB interface, and the required voltage is 5V.

The Hokuyo URG-04LX consists of a compact stacked structure with a spindle motor and the actual scanner on top of it. The motor rotates a small transmission mirror that deflects the vertical laser beam coming from the top of the sensor into horizontal direction. This allows the laser beam to scan a planar area around the sensor with an opening angle of 240°. A second mirror below, the reception mirror, deviates the horizontal laser beam captured by a lens into vertical direction again [43].

Distances are then computed by measuring the so-called *time-of-flight*, that is the time gap between the signal emission and its return after being reflected by a surface. This quantity is equal to twice the space divided by the wave velocity:

$$t = \frac{2d}{v} \tag{3.1}$$

from which the distance can be easily extracted:

$$d = \frac{tv}{2} \tag{3.2}$$

A full scan is performed every 100 ms.

The two lasers scanners are mounted on each side of the footplate. In order to maximize the scanning angular range, the two sensors are placed

Figure 3.2: An Hokuyo URG-04LX laser scanner

at about 90° with respect to the longitudinal axis of the robot. In this way the central area in front of the wheelchair is covered by both sensors, adding robustness to the detection of obstacles: if one scanner fails, the other one will keep a minimum degree of security.

3.2.2 Encoders

LURCH is also equipped with two encoders, one for each driving wheel. These sensors are able to measure the number of rounds completed by the wheels, expressed in *ticks* (detectable sections of a round), in order to abstract odometry information. The encoders currently mounted on the robot are implemented using two Honeywell's HOA0961 infrared sensors in quadrature. They are fork-shaped sensors composed of an infra-red emitting diode facing an Optoschmitt detector. The emitted ray passes through a toothed disc mounted on the wheel axis, and is caught by the receiver. The signal can be received if the ray passes through an open space, or not received if the ray intersects a tooth. This originates two different states that can be used to detect if a wheel has moved. The second sensor is used to detect the direction of rotation.

The disc has 45 teeth, which originate 45 rising edges and 45 falling edges. The total amount has to be doubled because of the use of two sensors, thus obtaining 180 ticks per round. In this configuration, considering a 20 ms timing between two readings of the cumulated counts, the minimum

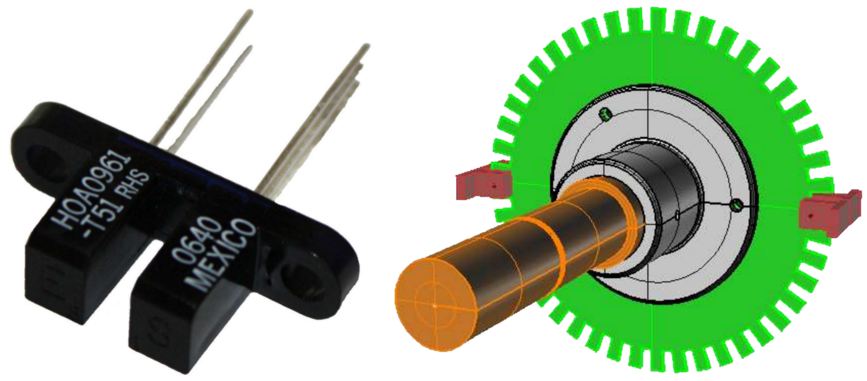


Figure 3.3: An Honeywell HOA0961 sensor and, on the right, a 3D rendering of the entire system

detectable velocity is given by:

$$v_{min} = \frac{\Delta ticks}{\Delta t \cdot ticks_{round}} \cdot circ \simeq \frac{1}{0.02s \cdot 180} \cdot 1m \simeq 0.28m/s \qquad (3.3)$$

where $\Delta ticks$ is the difference between the total number of ticks revealed by two subsequent readings, Δt is the difference between the timestamps of the two detections, $ticks_{round}$ are the number of ticks per round, and $circ$ is the wheel circumference, which in our case is 1 m.

The interface between the encoders and the computer is made of an electronic board, called *odometry board*, which sends messages with a period of 20 ms, specifying:

- a counter representing the number of messages sent since the start;
- the total number of ticks for the left wheel since the start;
- the total number of ticks for the right wheel since the start.

Communication is possible through a serial connection.

Although an electronic PID velocity controller based on encoder readings has been designed [3], it has never been added. Chapter 4 shows how a velocity controller based on encoders has been implemented as a software module by us in this thesis.

3.2.3 Other sensors

The robot is also equipped with sonars and a camera. In the previous software system, the wheelchair used the sonars to detect obstacles on the back, and the camera to detect landmarks located in the environment to provide

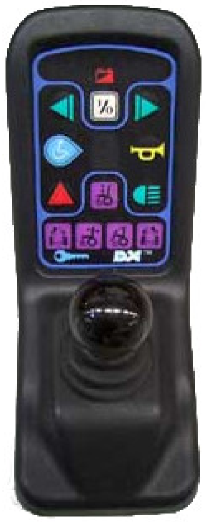


Figure 3.4: A DX-REM34: command panel

an absolute position estimate. Those sensors have not been used in the project, so we do not discuss them here. In the new software the absolute position is given by a Monte Carlo global localization algorithm named *Adaptive Monte Carlo Localization* (AMCL) (see Chapter 2), which uses the laser scans. AMCL has the advantage of not requiring any modification of the environment.

As for rear obstacles, since the wheelchair is meant to move mostly forwards, sonars are not strictly necessary to test the navigation features, although they could be added in future to improve security and add backward movements to the planner's action set. Anyway, sonars are not sufficiently accurate to be used for localization purposes (i.e. they can't be added to ROAMFREE multi-sensor fusion module). Nevertheless, in Chapter 6 we discuss some possible directions for improvement using additional sensors.

3.3 Low-level control

The low-level control of the actuators is done by the DX-REM34 module developed by Dynamic Controls and included in the wheelchair default configuration. This module has several functions among which:

- a joystick that can be used to control wheelchair speed and direction;
- a switch for selecting the gear among 5 different options (a seven segment display shows the currently selected gear).

In order to let the computer send commands to the motors and retrieve the commands given manually by the user, an interface circuit is present, known as *motor board*. Such board can read joystick positions and send them to the computer; at the same time, it can translate the commands given by the computer into appropriate voltage values for the motor controllers. This is done by converting the joystick analog signals into digital signals that can be sent to the computer by means of an ADC (Analog to Digital Converter). The output signal coming from the computer is then converted again into a compatible analog signal through a DAC (Digital to Analog Converter) to be sent to the motor controlling circuits. In this way the commands sent by the computer are treated exactly as they were sent directly through the joystick. It's important to clarify that velocity commands cannot be expressed as actual velocities (in m/s and rad/s), but rather as joystick positions. A position is given by two fields: position on forward-backward axis and on left-right axis. Both are expressed as integers between 0 and 255, where:

- for the forward-backward axis, 0 corresponds to maximum backward velocity and 255 corresponds to maximum forward velocity;
- for the left-right axis, 0 corresponds to maximum left velocity and 255 corresponds to maximum right velocity.

For both coordinates, 128 means 0 velocity. The actual velocity reached by the wheelchair obviously depends on the currently selected gear.

Even if the joystick is not directly connected to the motor controllers, it is still possible to drive the wheelchair in a normal way with the computer turned off. Indeed the motor board has two driving modes: *driven by user* and *driven by PC*. In *driven by user* mode the wheelchair can be normally driven through the on-board joystick, so the presence of the additional circuit is completely transparent; the computer cannot control the wheelchair in any way. In *driven by PC* mode the wheelchair is driven by the computer. The driving mode can be changed by means of a physical button, and the current state is displayed by two LEDs having different colours:

- a yellow LED indicates that the current mode is *driven by user*;
- a green LED indicates that the current mode is *driven by PC*.

If both LEDs are flashing intermittently, the wheelchair is in an intermediate state (*driving disabled*) in which it can be driven neither through the computer nor manually. This mode has been added for safety reasons. At

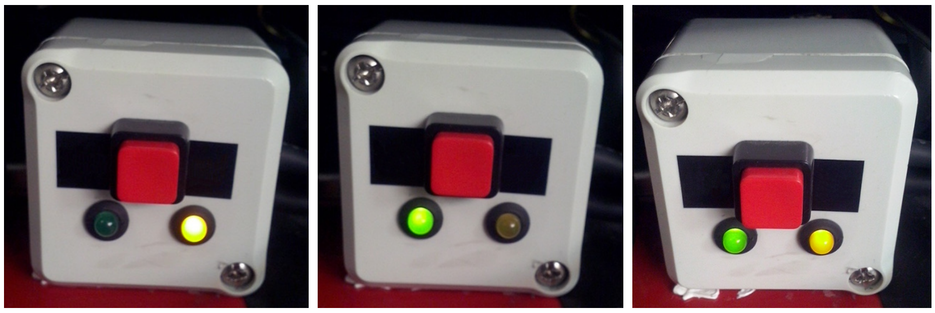


Figure 3.5: The three LED configurations associated to the driving modes. From left to right: driven by user, driven by PC, driving disabled

start, the wheelchair is in *driven by user* mode. Pressing the button can cause one of the following situations:

- if the computer is sending commands to the motor board, the wheelchair switches to *driven by PC* mode;
- otherwise the wheelchair switches to *driving disabled* mode.

Moreover, if the wheelchair is in *driven by PC* mode and no commands are sent by the computer for about 1 second, the wheelchair switches to *driving disabled* mode.

Messages coming from and sent to the board have the same format, only their interpretation is different. They are composed of four fields (each expressed as one byte). The first field represents the current mode: 0 for *driven by user*, 1 for *driven by PC*, 2 for *driving disabled*. The second field contains the joystick position on the forward-backward axis, expressed as an integer between 0 and 255. The third field is the joystick position on the left-right axis, again between 0 and 255. The last field is a checksum that can be used to verify message integrity; it is computed as the sum of the other three fields. Communication between the motor board and the computer is made possible through a serial connection.

3.4 Computer

The computer elaborates sensor readings in order to extract information about the environment and the robot location. In addition, it provides intelligent behaviours like obstacle avoidance and autonomous navigation, and automatically sends commands to the motor board.

A Zotac ZBOX ID83 has been chosen for its compact size and its good performances. It is equipped with an Intel Core i3 3120M processor, having

Figure 3.6: The Zotac ZBOX ID83

two physical cores running at 2.5 GHz. Concerning memory and storage, it has 4 GB of RAM and a 64 GB SSD. As for the graphics part, the ZBOX ID83 is equipped with an Intel HD 4000 integrated video card, supporting OpenGL 4.0. Ethernet and Wi-Fi connections are also included [44].

The on-board Operating System is Ubuntu-Linux. This OS has been chosen for its full compatibility with ROS. Furthermore, most of ROS packages and add-ons are already available in the Ubuntu repositories, which are constantly updated. As for the version, 12.04 has been chosen, since it is the last version available with long term support (5 years). The installed ROS distribution is Groovy Galapagos [38].

3.5 Touch screen

In order to provide a visual feedback and allow the user to select goals in a simple way, a touch screen monitor is present. The monitor, a Xenarc 700YYV [45], has a 7" display with a resolution of 800x480, and integrated speakers. In this project the touch screen has been used mainly to give a visual feedback of the localization during testing sessions, and to set navigation goals through *rviz* while on the wheelchair; no dedicated user interface has been designed yet.

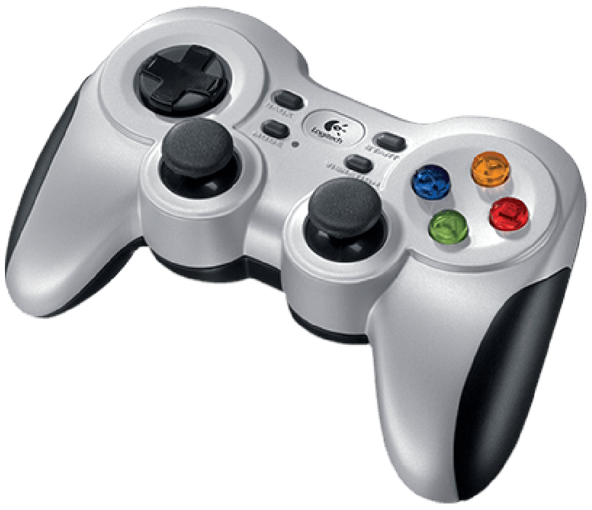


Figure 3.7: The Logitech F710 Wireless Gamepad

3.6 Joypad

The wheelchair can be driven remotely by means of a wireless joypad. The joypad that has been adopted is a Logitech F710 Wireless Gamepad [46], a device commonly used for video games. This device is connected to the computer by means of a USB wireless receiver. The joypad has 2 analog sticks having an angular range of 360°, an 8-way digital pad, and 12 digital buttons, two of which are accessible by pushing the analog sticks. This device allows to set up many different button configurations.

3.7 Emergency management

A security system has been added to the wheelchair; on one side, easy to reach for the user, there is an emergency button that, if pushed, blocks the DX-REM34 controller and stops the wheelchair. In that condition, the display on the joystick panel shows an horizontal segment. In order to unlock the system it is necessary to turn the button and reset it to its original position. The security block can also be triggered by pushing any button on a dedicated safety remote (Figure 3.8). If the system is blocked by the remote, in order to restore it the physical security button must be pushed and released again. Figure 3.9 shows the complete configuration of LURCH.

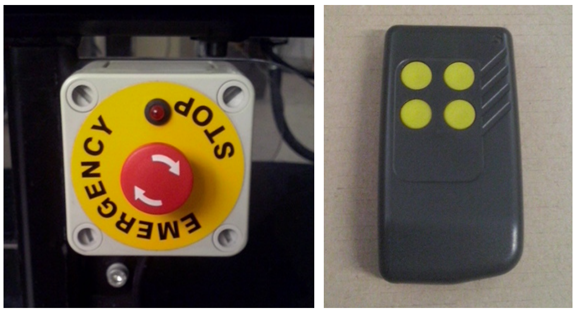


Figure 3.8: The emergency button and, on the right, the remote

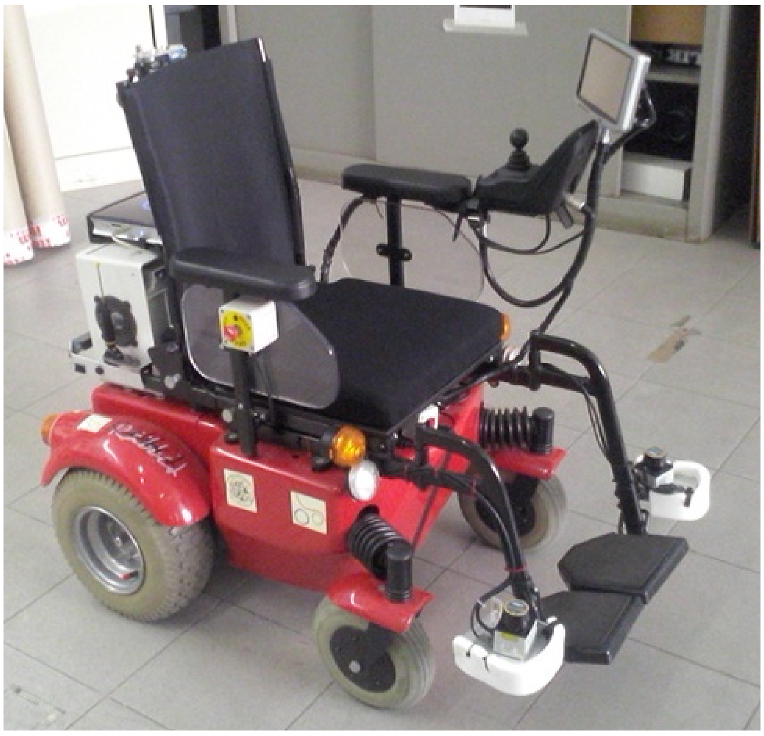


Figure 3.9: The complete LURCH robotic system

Chapter 4

Software project

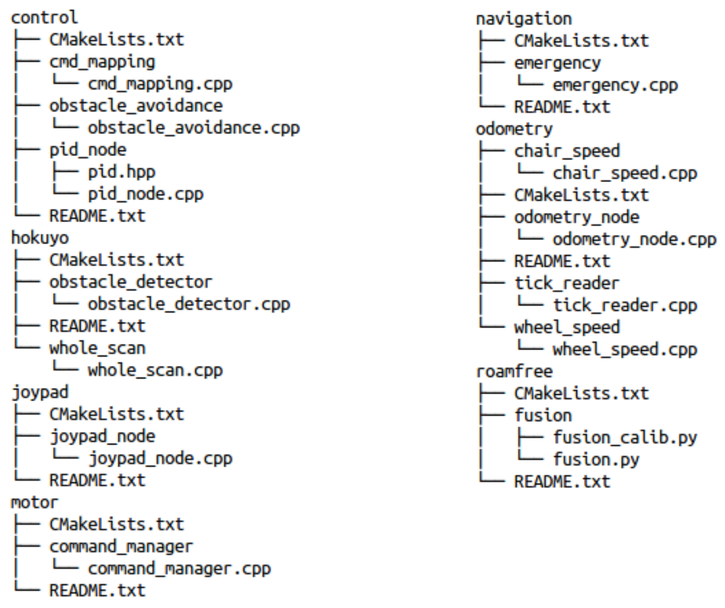
This chapter describes the design of the novel control architecture for LURCH. First of all, sensor readings are treated; then the multi-sensor fusion method for localization is discussed. The following part is devoted to assisted drive, autonomous behaviour, velocity control and management of the operating states.

4.1 Initial set-up

Since one of the aims of this project is to build a solid base structure for future developments, a significant amount of time has been spent to think about the organization of code. The first dilemma we have encountered concerned whether to deploy it as a single package or as a set of packages.

The first option makes it easier to port the whole project on other machines, thanks to its atomicity; on the other hand, if its content grows because of a large number of nodes, browsing the source files may be difficult. Developing a set of packages has the clear advantage of having all the code well partitioned, keeping track of the dependencies between modules, and allowing to distribute single parts easily; however working with many packages is more complex, running a node implies remembering to which package it belongs, and the project is spread throughout the workspace, meaning that packages involved in the project are placed together with packages belonging to other potential projects.

Eventually, we have chosen to develop a single package and give a hierarchical structure to the source files by means of folders, in order to improve code browsing and readability. At a high level, we have categorized code based on the logical part of the robot which it involves (odometry, naviga-



```
control
├── CMakeLists.txt
├── cmd_mapping
│   └── cmd_mapping.cpp
├── obstacle_avoidance
│   └── obstacle_avoidance.cpp
├── pid_node
│   ├── pid.hpp
│   └── pid_node.cpp
└── README.txt
hokuyo
├── CMakeLists.txt
├── obstacle_detector
│   └── obstacle_detector.cpp
├── README.txt
└── whole_scan
    └── whole_scan.cpp
joypad
├── CMakeLists.txt
├── joypad_node
│   └── joypad_node.cpp
└── README.txt
motor
├── CMakeLists.txt
├── command_manager
│   └── command_manager.cpp
└── README.txt
```

```
navigation
├── CMakeLists.txt
├── emergency
│   └── emergency.cpp
└── README.txt
odometry
├── chair_speed
│   └── chair_speed.cpp
├── CMakeLists.txt
├── odometry_node
│   └── odometry_node.cpp
├── README.txt
├── tick_reader
│   └── tick_reader.cpp
└── wheel_speed
    └── wheel_speed.cpp
roamfree
├── CMakeLists.txt
├── fusion
│   ├── fusion_calib.py
│   └── fusion.py
└── README.txt
```

Figure 4.1: File system of the project

tion, control, and so on). Inside each of these categories, a folder is dedicated to each node. Nodes are isolated because, in general, a single node can be made of more than one source file; for instance, `pid_node` (Section 4.7) is made of a header file and a `.cpp` file that implements it. In this way, retrieving a node means simply copying the correspondent folder. Figure 4.1 shows the file system of the project. In the architecture diagrams presented in this chapter, each folder is represented as a coloured rectangle with a label on it.

4.2 Laser scanners and scan matching

The Hokuyo laser scanners are very useful, not only to detect obstacles, but also to provide odometry information. Indeed, subsequent scans can be compared by means of a technique known as *scan matching*. Differences between scans imply a linear or angular movement, whose measurement can be used for odometry purposes. For this reason, the output of the scan matching module is given as input for ROAMFREE, in order to add a second odometry sensor to the one implemented through encoders.

4.2.1 Hokuyo drivers

As seen in Chapter 3, the two Hokuyo lasers are connected to the computer via USB. The driver that allows to communicate with them is already provided in the ROS repositories. It is called `laser_drivers` and it is basically a set of packages that deals with the most common laser sensors, Hokuyo's included.

The package we are interested in is `hokuyo_node`. This package contains a node of the same name that reads sensor raw data and publishes an array of 682 elements. Each element in the array represents the distance in meters to the closest obstacle detected in one section of the angular range. The order is counterclockwise around the z axis (which is considered to point upwards), in accordance to the mostly used convention in angle measurement. Like most of ROS message types, this message contains a *header*, a field including a sequence number, a time stamp, and a frame id, that is a string associated to the sensor reference frame. Obviously, in order to adjust the provided package to our specific case, some parameters had to be set.

First of all, since we have two sensors, two instances of the node have to be run. Each one needs a different name in order to be recognizable. We have chosen `hokuyo_left` and `hokuyo_right`. Also the published topics have been renamed, in order to separate the output of the two sensors, resulting in `scan_left` and `scan_right` respectively. Another required parameter is the logical port name corresponding to each device. It is important that left and right Hokuyos are always distinguishable from each other, regardless of the physical port they are attached to; moreover, the name of each sensor must always correspond to its geometrical position (sensors must not be "swapped"). For these reasons, we had to define univocal names for logical ports. In general, the name assigned to a device is not the same every time the operating system is booted up: each device is assigned a string terminating with an increasing number that represents the order in which it is detected by the system (in this case, `ttyACM0`, `ttyACM1`...). This order is not deterministic, so it is not always sufficient for an unambiguous identification. We have solved this problem by writing an `udev` rule. In Linux an `udev` rule is basically a file specifying what to do when a specific device is plugged in. In our case, we needed to build a symbolic link to the logical port of each Hokuyo, and use that link as a port name for that sensor's node instance. The link name has to identify the device univocally. In order to do that, we have used a program included in the drivers folder, named `getID`. This program returns the serial ID for a sensor, given its logical port name. Then, the resulting ID has been used as the symbolic link name. Here we report part of the rule:

```
PROGRAM=="/opt/ros/groovy/stacks/laser_drivers/hokuyo_node/bin/
getID /dev/%k q", SYMLINK+="hokuyo_%c"
```

`%k` is the kernel name for the device (e.g., `ttyACM0`), and `%c` gets the output value of the program specified by attribute `PROGRAM`. Once the geometrical position of the two devices has been identified, we have assigned the corresponding symbolic link names to the appropriate nodes.

4.2.2 Scan matching

For the scan matching part, a node developed by Politecnico di Milano has been used. It reads laser scans from either `scan_left` or `scan_right` topic, and publishes an odometry topic that provides the estimated pose of the correspondent sensor, with respect to that sensor reference frame. The pose is expressed as position and orientation. Position is declared as three coordinates (x, y, z), while orientation is expressed as a quaternion (q_x, q_y, q_z, q_w).

This node has the peculiarity of providing not only the pose estimate given by scans, but also the associated 6x6 covariance matrix:

$$\begin{bmatrix} \mathtt{V}(x) & \mathtt{C}(x,y) & \mathtt{C}(x,z) & \mathtt{C}(x,q_x) & \mathtt{C}(x,q_y) & \mathtt{C}(x,q_z) \\ \mathtt{C}(y,x) & \mathtt{V}(y) & \mathtt{C}(y,z) & \mathtt{C}(y,q_x) & \mathtt{C}(y,q_y) & \mathtt{C}(y,q_z) \\ \mathtt{C}(z,x) & \mathtt{C}(z,y) & \mathtt{V}(z) & \mathtt{C}(z,q_x) & \mathtt{C}(z,q_y) & \mathtt{C}(z,q_z) \\ \mathtt{C}(q_x,x) & \mathtt{C}(q_x,y) & \mathtt{C}(q_x,z) & \mathtt{V}(q_x) & \mathtt{C}(q_x,q_y) & \mathtt{C}(q_x,q_z) \\ \mathtt{C}(q_y,x) & \mathtt{C}(q_y,y) & \mathtt{C}(q_y,z) & \mathtt{C}(q_y,q_x) & \mathtt{V}(q_y) & \mathtt{C}(q_y,q_z) \\ \mathtt{C}(q_z,x) & \mathtt{C}(q_z,y) & \mathtt{C}(q_z,z) & \mathtt{C}(q_z,q_x) & \mathtt{C}(q_z,q_y) & \mathtt{V}(q_z) \end{bmatrix} \tag{4.1}$$

where $\mathtt{C}(x,y)$ is the covariance between x and y, defined as $\mathtt{E}[(x-\mu_x)(y-\mu_y)]$, μ_x being the expected value of x. $\mathtt{V}(x)$ is the variance of x, defined as $\mathtt{C}(x,x)$. The covariance matrix can be used to state how much the estimate is reliable when doing multi-sensor fusion: if covariances are high, more importance will be given to other odometry sources; on the contrary, if covariances are low, the scan matcher is quite sure of the estimate, therefore it will be considered more reliable than other odometry sources. Of course, also for this node two instances must be present, originating two topics: `odometry_left` and `odometry_right`. These topics are read by the ROAMFREE fusion node.

4.3 Encoders

Encoders are another important source for pose estimation. Since the frequency at which encoder data are provided is relatively high (50 Hz), they can be elected as master sensors for ROAMFREE. Moreover, since they are the fastest sensors that are able to measure the current speed of the wheelchair, they can also be used to close the loop in velocity control.

4.3.1 Ticks reading

In order to interact with the odometry board, which periodically sends a message containing the number of ticks detected on each wheel, we have written a node called `tick_reader`. This node reads raw data from the serial port corresponding to the board, extracts the number of ticks for the two wheels and publishes them on a topic, called `enc`.

The format of messages coming from the board is the following:

```
O [COUNTER][LEFT TICKS][RIGHT TICKS]
```

The node opens a serial connection with the board and constantly reads characters from the stream by means of the `fgetc()` function, which, on success, returns the character currently pointed by the internal file position indicator; then, the position indicator is advanced to the next character [47]. All characters are initially discarded until an *O* is read; all successive characters before the following *O* are parsed, in order to extract useful data. Those data are then wrapped in a custom ROS message we have created: `Encoders`. This message is composed of the following fields:

- `header.seq`: sequence number of the message, based on the counter read from the odometry board, and initialized at node start up;
- `header.stamp`: the time stamp of the message;
- `left`: the total number of ticks read from the left encoder;
- `right`: the total number of ticks read from the right encoder.

As for the sequence number, also left and right ticks are initialized at node start up. These initializations are needed otherwise the counter and the number of ticks would be computed based on the board starting time, which does not correspond to when the actual software starts.

Encoder ticks are used to estimate the velocity of the wheels. In order to do this, it is important that the time stamp difference between subsequent messages published on topic `enc` is as accurate as possible. However, the odometry board does not provide time information; this is a problem, because we can measure timestamps only via software (by means of functions like `ros::Time::now()`, which return the current wall time), with a non-null delay from the instant at which data were sent by the board. This delay is mostly given by the communication time. An additional delay is caused by the scheduling mechanisms that occur in the Operating System: the instant at which the reading process is given priority by the system scheduler is not deterministic.

Assuming that the frequency at which the board outputs data is approximately constant over time, a solution to this problem is setting the time stamp of the current message as the time stamp of the previous one plus the period:

$$\hat{t}(i) = \hat{t}(i-1) + \Delta t; \tag{4.2}$$

where Δt is the period; since the nominal frequency of the board output is 50 Hz, we could set $\Delta t = 0.02s$. However, the real frequency is not exactly equal to the nominal one, because the accuracy of the board hardware is not as perfect. For this reason, an estimate of the period has been computed:

$$\Delta\bar{t}(i) = \bar{t}(i) - \bar{t}(i-1); \tag{4.3}$$

$$\Delta\hat{t}(i) = w \cdot \Delta\bar{t}(i) + (1-w) \cdot \Delta\hat{t}(i-1); \tag{4.4}$$

$$\hat{t}(i) = \hat{t}(i-1) + \Delta\hat{t}(i). \tag{4.5}$$

First (Equation 4.3) the current difference between time stamps is measured by computing the difference between the new return value of function `ros::Time::now()` ($\bar{t}(i)$) and the one obtained for the previous message ($\bar{t}(i-1)$), thus getting one sample $\Delta\bar{t}(i)$. The unbiased estimated period is then computed recursively as a weighted average between the current sample $\Delta\bar{t}(i)$ and the period estimated for the previous message $\Delta\hat{t}(i-1)$ (Equation 4.4). We have chosen value 0.01 for w, thus giving much more importance to the period estimated so far than to the current sample. In fact, our goal is to reduce the impact of outliers, and change our estimate significantly only if many samples have a new value. Finally (Equation 4.5), the time stamp of the message is computed as in Equation 4.2, but using the new period estimate. This expedient has significantly reduced noise in the resulting velocity profile.

The node also publishes data on two separate topics, `lwheel` and `rwheel`, compatible with the `diff_tf` node, included in the `differential_drive` package, already provided by ROS. The latter computes odometry (pose and velocity of the robot) given the ticks, the number of ticks per meter and the distance between the wheels. Since ROAMFREE takes the angular velocity of the wheels and computes the robot pose by itself including information coming from the laser scanners, `diff_tf` has only been used to test the `tick_reader` node, while it has not been employed in the final implementation.

4.3.2 Wheel velocities

A node called `wheel_speed` is responsible for computing the angular velocity of the wheels. This node subscribes to topic `enc`, containing the number of ticks for each wheel and the correspondent time stamp, and publishes a topic called `wspeed`. The type of messages published on this topic is custom, and we have called it `Wheelspeed`. Its format is the following:

- `header`: it is equal to the header of the last `Encoder` message that has been received;
- `left`: the angular velocity of the left wheel, in rad/s;
- `right`: the angular velocity of the right wheel, in rad/s.

In order to output one speed message, at least two messages from the encoders are needed, so the node stores the last message retrieved. The computation is done as it follows (for simplicity, here we consider only one wheel, since the computation is the same for both):

$$\Delta ticks = ticks(i) - ticks(i-1); \tag{4.6}$$

$$\Delta t = \hat{t}(i) - \hat{t}(i-1); \tag{4.7}$$

$$\omega = \frac{\Delta ticks \cdot 2\pi}{\Delta t \cdot 180}; \tag{4.8}$$

where $ticks(i)$ is the number of ticks returned by the i-th `Encoders` message for the wheel, and $\hat{t}(i)$ is the time stamp of the i-th message estimated before.

4.4 Multi-sensor fusion

Once all sensor data have been captured, our goal is to merge them to obtain a robust odometry information, minimizing possible errors. For this reason, ROAMFREE has been used. We have created a node called `fusion`, written in Python language, that subscribes to all topics from which it is possible to estimate the robot location, and publishes the robot estimated pose, using the ROAMFREE library.

4.4.1 Coordinate frames and sensor parameters

Firstly we had to provide the right sensor displacements, in terms of coordinate frames. We have considered as center of the robot the middle point of the rear wheel axis, which corresponds to the reference frame of the encoders.

Indeed, the velocity of the wheels provide information on the velocity of the axis center. As for the coordinate frames, we have adopted ROS conventions, which consider the x axis directed as the robot, and pointing to the robot front; the z axis is orthogonal to the plane and pointing upwards; the y axis is placed according to the right-hand rule. The odometry information provided by the laser scanners also adopts this convention.

Figure 4.2 shows the location of the reference frames. Using the same terminology adopted by ROAMFREE (Chapter 2), S_i indicates the reference frame of the i-th sensor, O represents the odometry frame, and $\Gamma^O_{S_i}$ represents the rototranslation from O to S_i. The two rototranslations $\Gamma^O_{S_L}$ and $\Gamma^O_{S_R}$ have been measured manually:

- as for $\Gamma^O_{S_L}$, we have set $x^O_L = 0.765m$, $y^O_L = 0.22m$, $q^O_{zL} = 0.59$, the other coordinates equal to 0;
- as for $\Gamma^O_{S_R}$, we have set $x^O_R = 0.755m$, $y^O_R = -0.235m$, $q^O_{zR} = -0.67$, the other coordinates equal to 0.

q^O_{zi} has been computed considering that, on a plane, $q^O_{zi} = \sin(\theta/2)$, where θ is the counterclockwise rotation of frame S_i around the z axis with respect to reference frame O.

In order for ROAMFREE to compute linear and angular velocities of the robot, based on the velocity of the wheels, it is necessary to provide other two parameters: the wheel radius and the axis length, that is, the distance between the wheels. These parameters have been measured manually, obtaining values 0.16 m for the wheel radius and 0.5 m for the axis length.

4.4.2 Pose tracking

Node `fusion` subscribes to topics `wspeed` (corresponding to the angular velocity computed with the encoders), `odometry_left` and `odometry_right` (the odometry topics provided by the two laser scan matchers). At start up, sensors are added to the fusion engine. The logical sensor that corresponds to the encoders is `DifferentialDriveOdometer`, which assumes that the kinematic model of the robot is the standard differential drive. Each laser scanner, on the contrary, is considered as a pair of logical sensors, one that measures linear velocity and the other that measures angular velocity. When a sensor is added, it is required to define which, among them, has to be the master sensor. The master sensor, as said in Chapter 2, is the main sensor used to build the hypergraph. It must be the fastest sensor (i.e., the one with the highest frequency) and it must provide sufficient information to predict the next robot pose. The `DifferentialDriveOdometer` satisfies

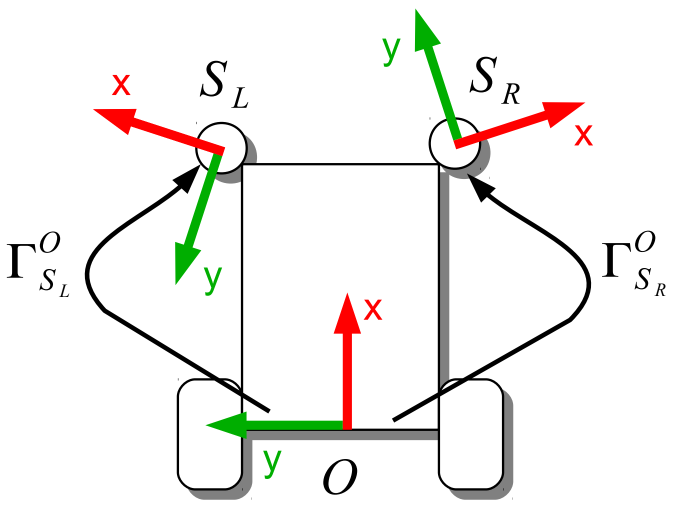


Figure 4.2: The reference frames on the robot

all these requirements, because it produces data at the frequency of 50 Hz (while laser scanners run at 10 Hz), and, from the measurement of the wheel angular velocities and the last estimated pose, it is possible to predict the new pose.

When a sensor measurement is added, the associated covariances have to be specified. The laser scan matchers already provide covariance matrices (see Section 4.2.2), so we have adopted that. Concerning the encoders, we have set an arbitrary covariance matrix with this form:

$$\begin{bmatrix} V(w_1) & C(w_1,w_2) & C(w_1,y) & C(w_1,z) & C(w_1,\alpha) & C(w_1,\beta) \\ C(w_2,w_1) & V(w_2) & C(w_2,y) & C(w_2,z) & C(w_2,\alpha) & C(w_2,\beta) \\ C(y,w_1) & C(y,w_2) & V(y) & C(y,z) & C(y,\alpha) & C(y,\beta) \\ C(z,w_1) & C(z,w_2) & C(z,y) & V(z) & C(z,\alpha) & C(z,\beta) \\ C(\alpha,w_1) & C(\alpha,w_2) & C(\alpha,y) & C(\alpha,z) & V(\alpha) & C(\alpha,\beta) \\ C(\beta,w_1) & C(\beta,w_2) & C(\beta,y) & C(\beta,z) & C(\beta,\alpha) & V(\beta) \end{bmatrix} \tag{4.9}$$

where the top-left 2x2 block is the additive noise on the wheel velocity, while the remaining parts constraint the degrees of freedom which are not fixed by the kinematics: y, z, roll (α) and pitch (β). Indeed, the differential drive kinematics provides information only on x and yaw, leaving the other space coordinates free.

Covariances $V(w_1)$ and $V(w_2)$ take into account the encoder resolution, the floor friction and other physical phenomena that have not been modelled.

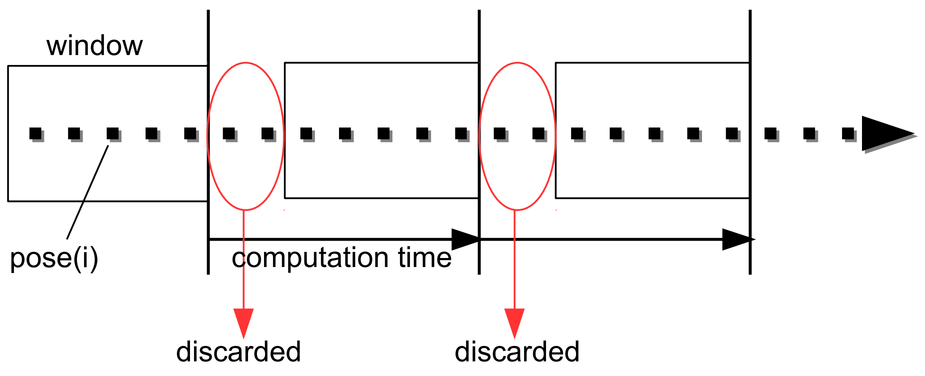


Figure 4.3: An instance of the "narrow window" problem: the algorithm uses the last 5 poses, but 7 poses have been added since the last computation start, so 2 poses are discarded.

We have assigned values on the order of 10^{-2} to them. The covariance between the two wheel velocities, namely $C(w_1, w_2)$, has been set to zero, since the wheels are independently controlled, and so we can assume that disturbances on the two measures are not correlated. $V(y)$, $V(z)$, $V(\alpha)$ and $V(\beta)$ are set to very low values, in order to constraint the motion on the xy plane. We have treated separately the special case in which the wheels are not moving. Indeed, when the robot is steady, laser scan matchers become more and more uncertain on the robot pose, resulting in divergent covariances. In order to ensure that the fusion engine considers only the encoder information in this particular situation, we have forced $V(w_1)$ and $V(w_2)$ to be very low (on the order of 10^{-6}) if left and right wheel velocities are zero, respectively.

Before running the fusion engine, we must select the time window size and choose the solver method. The time window size is the maximum number of poses that are considered in the hypergraph. This is a critical parameter, because, in order to improve the estimation, the number of poses has to be the largest possible; however, having many nodes in the hypergraph results in a higher complexity for the solver. In other words, the frequency at which the engine can run is proportional to the window size. On the other hand, if the window is too narrow it is possible that some poses are discarded from the pose window before being processed: since the window shift is faster than the solver computation, the window could "leave behind" poses, thus losing information (see Figure 4.3).

The effectiveness and performance of the fusion engine relies also on the solver; in general, the more the iterations of the algorithm are, the better the solution is. The solver methods to choose from in the current ROAMFREE implementation are Gauss-Newton and Levenberg-Marquardt [48]; the number of iterations is customizable for both. While the two algorithms have proven to have similar performances for this problem, the number of iterations is a critical parameter: increasing it in general improves the solution, but at the same time it can drastically reduce the allowed frequency for the ROS fusion node. Moreover, the computational power of the CPU plays an important role in this part.

We have found empirically a good trade-off using the following parameters:

- window size: 50 poses
- solver method: Gauss-Newton
- number of iterations: 3

With these parameters, we have succeeded in running the whole ROS fusion node at about 20 Hz, while obtaining satisfying localization solutions.

The resulting pose is formulated as a transform between the fixed frame W (the *world* frame) and the robot frame O at current time t, namely $\Gamma_O^W(t)$. The initial displacement, $\Gamma_O^W(0)$, can be set arbitrarily by means of a ROS parameter: at start, the `fusion` node searches for initial pose values (x_0, y_0, z_0, q_{w0}, q_{x0}, q_{y0}, q_{z0}) on the parameter server; if it finds them, it retrieves them, otherwise it initializes the pose at (0,0,0,1,0,0,0), meaning that the world frame is placed at the starting pose of the robot. In practice, without providing an initial pose for the robot, at start $W \equiv O$. In Figure 4.4 you can see a graphical visualization of the main ROAMFREE reference frames, obtained with *rviz*. `left_hokuyo_link` and `right_hokuyo_link` correspond to S_L and S_R respectively, while *roamfree* corresponds to O.

In Figure 4.5 the software architecture for the multi-sensor fusion part is shown. The resulting pose is communicated to the other nodes in two ways: it is published as a `tf` transform and as a message on a topic named `pose`.

4.4.3 Parameter calibration

The five logical sensors are configured with parameters that we have measured empirically (Section 4.4.1). For every parameter, however, we can specify whether it is fixed or it needs to be calibrated it during the pose-tracking process. Parameter calibration aims at adjusting measurements

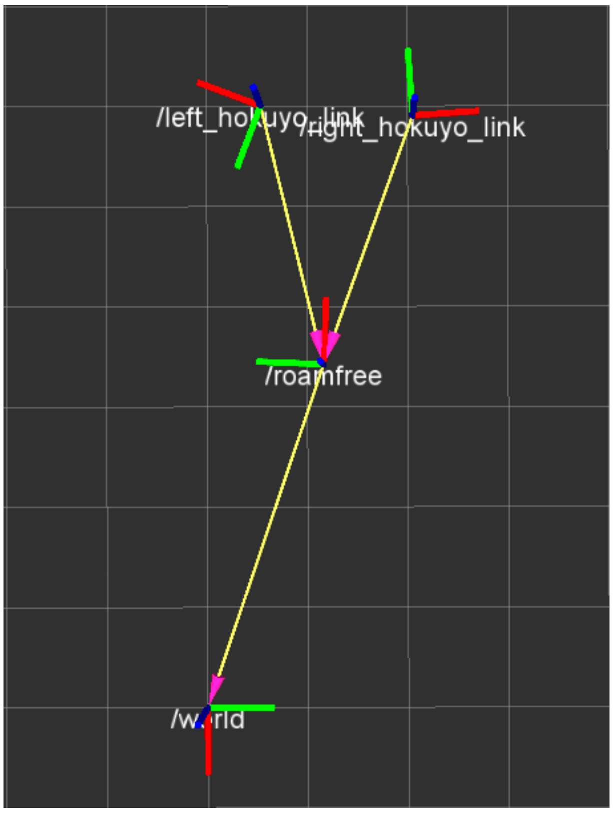


Figure 4.4: The ROAMFREE reference frames visualized in rviz

that have been obtained manually (for example, with tools like yardsticks or protractors). Empirical measures are often imperfect; moreover, measuring with sufficient precision quantities like angles is not an easy task. For this reason we have tried to use the calibration feature of ROAMFREE to estimate the uncertain parameters: the displacements of the two laser scanners, the radius of the rear wheels and the length of the wheel axis.

Online calibration has proven to be very demanding in terms of resources; thus it reduces the speed of pose estimation, making it difficult to keep track of the robot location. Since in this case parameters do not change significantly over time, we have considered doing calibration in one session, and then to use the returned parameter values for pose-tracking, keeping them fixed.

As for the calibration session, we used a set of OptiTrack cameras for motion capture [49]. Such system is able to track with high accuracy the position of special markers in a 3D space, with respect to a fixed point in the environment (in this case, a point on the laboratory's floor). We placed a marker on the wheelchair; the OptiTrack software, running on a dedicated

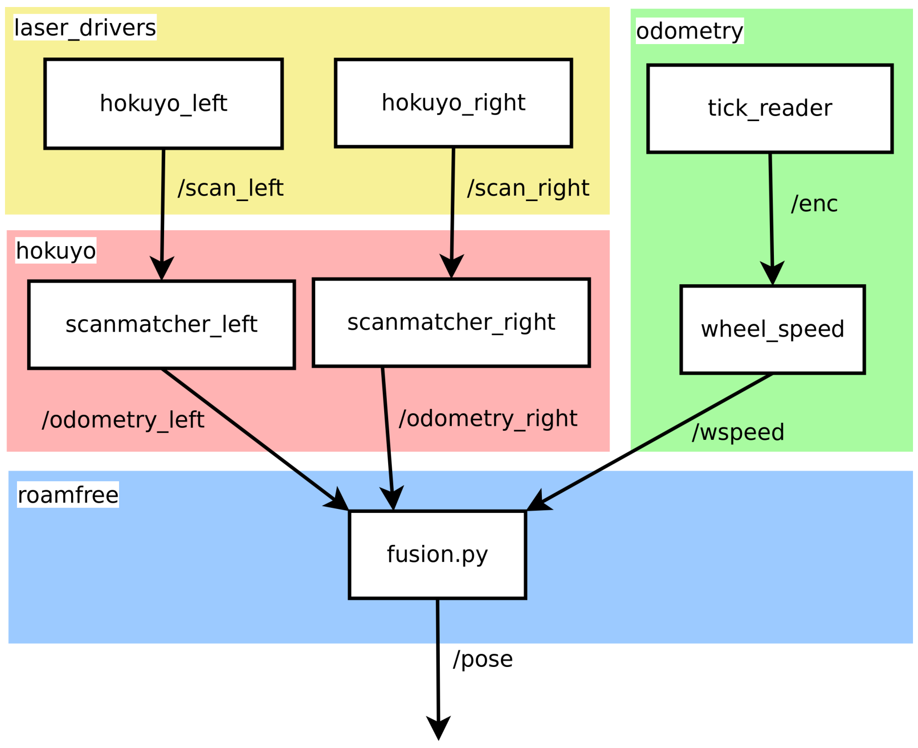


Figure 4.5: The multi-sensor fusion and pose-tracking architecture

computer, sent pose information to the wheelchair's PC, where a special node, called `mocap_node`, converted it into appropriate ROS messages. The first message was taken by node `fusion` as starting pose; in that case, frame W coincided with the OptiTrack's fixed reference frame.

The OptiTrack system was then added as logical sensor to ROAMFREE; in particular, the correspondent type of measurement was `AbsolutePosition`. We then moved the wheelchair for a few minutes, performing a random trajectory in which the robot could experience many possible movements (going forward and backward, turning left and right); we recorded sensor data by means of *rosbag*, a ROS tool that can store execution information (e.g. nodes and topics running, messages with the appropriate time stamps, etc.), allowing to reproduce it later offline. We then replayed the sample trajectory, running ROAMFREE with parameter calibration and the following configuration:

- window size: 1000 poses
- solver method: Gauss-Newton
- number of iterations: 300

We used a large number of poses for the window because we wanted the solution to be found considering all the trajectory. Moreover, we adopted a large number of iterations in order to be quite sure that the algorithm reached convergence.

The parameter values we obtained were far from reality (for instance, wheels were considered a lot smaller than the real ones, and the distance of the laser scanners from the odometric center along the x axis was significantly shorter). This is not a surprising fact, since the estimation process solves the problem in an analytical way, without taking into account the real world; thus, it finds a feasible optimal solution in terms of error minimization, which does not always correspond to the real case. Indeed, even with those unrealistic parameter values, the wheelchair position was tracked quite well [1].

4.5 Motor board and velocity control

In order to send commands to the wheelchair, as seen in Chapter 3, we needed to communicate with the motor board. Since the only type of command we can send to the board is a position for the on-board joystick, and not a velocity command, a module that translates velocity commands into joystick positions is needed. Furthermore, given that the wheelchair is not equipped with any hardware velocity control system, a PID controller has been implemented via software. The controller receives setpoints from the control devices or the autonomous navigation module, and adjusts them on the basis of the current velocity.

4.5.1 Motor board interface

The motor board has an interface which is similar to that of the odometry board, since it also communicates with the computer via a serial link. Unlike the odometry board, however, messages can be not only received, but also sent. Indeed the board sends to the computer the current position of the on-board joystick, in order to manage it via software, and at the same time receives command messages about the desired position of the on-board joystick. It is therefore important that reading and writing on the motor board happen simultaneously: a read must not wait for a write to finish,

[1]In the end we have obtained better performances using the parameters that we had measured empirically, so in the final implementation we have used those, leaving to future studies a more specific research on parameter calibration for LURCH.

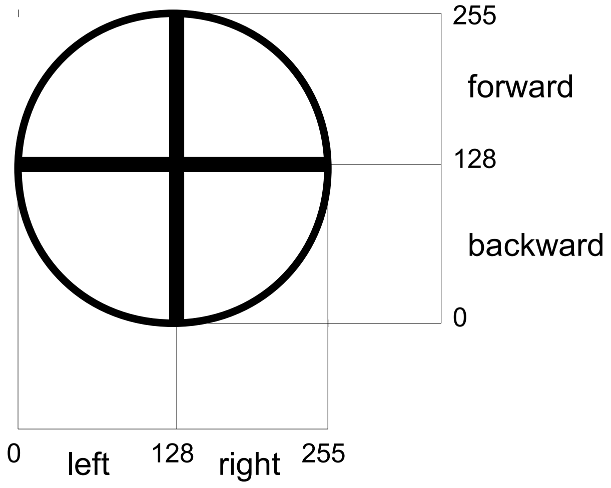


Figure 4.6: The correspondence between joystick positions and command values

and vice versa. This is an important issue, both for security and for the correct functioning of the robot.

The old implementation solved this problem with a *polling* mechanism, which periodically controlled if the buffer of incoming messages was empty; if not, a message from the buffer was read. At the end of the reading part, command messages coming from the control software were written on the board. This approach has been abandoned for two reasons: first of all, read and write within an iteration step were still sequential; secondly, commands sent from the joystick were not read as soon as they were available, but on a periodical basis. For this reason we have opted for an approach that uses two threads, one (called *main*) that sends commands to the board, and another (*receive*) which reads messages coming from it. In more detail, thread *main* subscribes to topic `command`, which contains command messages, and sends the correspondent commands to the motor board; thread *receive* listens constantly to the serial port, reads messages from the motor board as soon as they arrive, parses them and publishes them on a topic called `joystick`.

Messages received by the motor board and sent to it have the following format:

```
[MODE][FWRW][RXLX][CHECKSUM]
```

where `MODE` corresponds to the operating mode of the wheelchair, as seen in Chapter 3 (*driven by user*, *driven by PC*, *driving disabled*); ?FWRW and RXLX correspond to the joystick position coordinates (forward-backward axis and left-right axis respectively); CHECKSUM is a number used to verify message integrity, and is computed as the sum of the 3 preceding fields. Each of these fields is encoded in 1 Byte, meaning that the number of obtainable informative data is equal to $2^8 = 256$. Field MODE uses only 3 of these combinations, as there are 3 possible operating modes for the wheelchair (0 for *driven by user*, 1 for *driven by PC*, 2 for *driving disabled*); the other fields use the entire range of possibilities, namely [0, 255]. As for the two position fields, 128 means steady position (correspondent to the axis center), while 0 and 255 correspond to the speed extremities (see Figure 4.6).

Values read from the board are converted into values between -1 and +1, in order to have a more intuitive notation. The type used for ROS messages is the same for both directions (topics `command` and `joystick`). We have called it `Motion`. It reproduces the same format of the raw messages (`mode`, `fwrw`, `rxlx`), with the exception of the two position fields, which are floating point values between -1 and +1, and the presence of an header, in order to track the number of messages received and their respective time stamps. Command messages sent to the motor board must have the `mode` field equal to 1, otherwise its content is discarded.

4.5.2 PID Controllers

A fundamental part of robot software is represented by speed control. In order to make the robot execute the commands given by the user or by the motion planner, it is necessary to have a mechanism that translates a velocity setpoint into an input signal for the motors such that, in a short time, the system velocity becomes equal to the setpoint. Such mechanism, for instance, makes the robot brake when an obstacle is near.

In Figure 4.7 a generic control system is shown. The setpoint $\bar{y}$ is compared to the measured output y. Their difference $e = \bar{y} - y$ is given as input to the controller, which in turn produces a control variable u for the real system. d represents disturbances on the system, while n is the measurement error, which causes measure y to be different, in general, from the real output y_{real}. In order for the controller to work properly, a mathematical model for the system is generally needed.

PIDs [50] [51], or Proportional-Integral-Derivative controllers, are the most widespread linear controllers in industrial settings. The main advantage of using a PID regulator is that a model of the real system is not required; the control variable u is simply computed as the sum of three

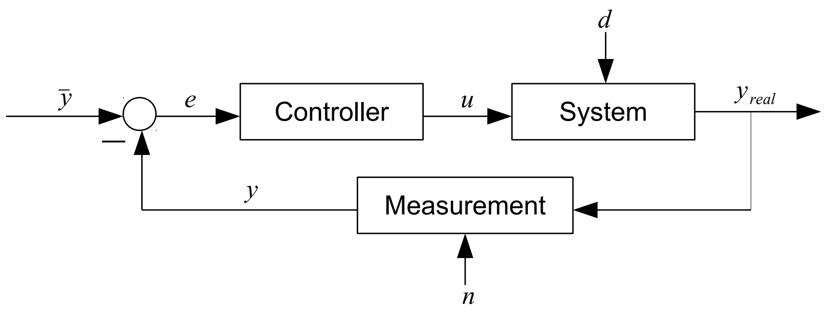


Figure 4.7: A generic control system

components:

$$u(t) = u_P(t) + u_I(t) + u_D(t). \tag{4.10}$$

The first one is proportional to the error e:

$$u_P(t) = K_P e(t). \tag{4.11}$$

In most cases, the proportional component alone is not sufficient to have an efficient control. This is because if K_P is too low, the process will respond slowly, while increasing K_P may lead the output of the system to oscillate.

The integral component contains information on the past of the error, and it is defined as:

$$u_I(t) = K_I \int_{t_0}^{t} e(\tau) d\tau. \tag{4.12}$$

This component is required in order to drive the error to zero. In many cases, the so-called PI controller, composed only of proportional and integral parts, is sufficient to have solid control.

The derivative component causes u to increase if y is increasing rapidly, making the control system more responsive. The derivative response is proportional to the rate of change of the process variable:

$$u_D(t) = K_D \frac{de(t)}{dt}. \tag{4.13}$$

The implementation of a PID controller is thus reduced to the tuning of the three constants K_P, K_I and K_D. However, these three components are not independent in general. In fact, K_P influences integral and derivative actions in the following way:

$$K_I = K_P / T_I; \tag{4.14}$$

$$K_D = K_P \cdot T_D; \tag{4.15}$$

where T_I and T_D are called *integral time* and *derivative time* respectively. Tuning K_P, T_I and T_D is generally easier, because the three quantities are independent from each other.

We have created a ROS node that implements a PID controller. The node takes a velocity setpoint (the $\bar{y}$ we mentioned above) and a velocity feedback (the measured output y), and produces a control variable for the motor subsystem. In more details, the node subscribes to two topics: `setpoint`, from which to read velocity setpoints, and `speed`, which contains measured velocity messages. It publishes the output of the PID as a message of type `Motion` (see Section 4.5.1), that is basically a position for the joystick expressed as numbers between -1 and +1. Indeed with the current configuration the wheelchair is not able to respond to velocity commands given in m/s and rad/s, therefore this node adjusts the "virtual" joystick position (as seen by the motor board) in order to obtain the desired velocity. Since velocity has a linear and an angular component, two instances of this node are present, one for each variable type. The two nodes publish their results on separated topics, called `control_linear` and `control_angular` respectively.

In order to speed up the tuning process, the node has been made fully configurable, meaning that the three constants K_P, T_I and T_D have been represented as ROS parameters that can be modified at runtime, without having to recompile the source. This allows to immediately see the effects of the tuned parameters on the robot behaviour.

Downstream of the two PIDs, a node named `cmd_mapping` has been introduced. Its task is merging the two control variables, by subscribing to topics `control_linear` and `control_angular`, and periodically publishing an appropriate `Motion` message on topic `command`, which in turn is read by node `command_manager`. In this way, command messages are sent by the motor board synchronously.

4.5.3 Velocity feedback

As for the velocity feedback, we can provide it in two possible ways. The first one is using the velocity estimated by ROAMFREE. The advantage of this solution is that the obtained measure would be more robust, since it involves two types of sensors and considers their respective error models. However, in order to retrieve a velocity estimate, some executions of the estimation process are required, and the time employed in computing the estimate would delay the response of the PID.

On the contrary, the control system must be as reactive as possible for security reasons, thus the alternative is to use the sensor data directly from the odometry board, because it has the highest output rate, and a very low variance compared with the estimate provided by the scan matchers.

In Section 4.3.2, we have shown how the angular velocities of the wheels are computed. Node `wheel_speed` subscribes to encoder data (topic `enc`) and publishes the correspondent wheel velocities on topic `wspeed`. We have chosen to reuse this module for speed calculation, by creating another node that subscribes to topic `wspeed` and provides the velocity of the odometric center of the wheelchair. We have named this node `chair_speed`.

In order to compute the velocity, the kinematic model must be used. We have chosen to adopt an ideal Differential Drive model. First of all, the wheel linear velocities are computed in the following way:

$$v_L = \omega_L \cdot r; \tag{4.16}$$

$$v_R = \omega_R \cdot r; \tag{4.17}$$

where r is the wheel radius. Given the wheel velocities, the linear speed of the odometric center of the wheelchair, which is located at the center of the wheel axis, is computed as the average between them:

$$v_O = \frac{v_L + v_R}{2}. \tag{4.18}$$

The corresponding angular velocity is computed as follows:

$$\omega_O = \frac{v_R - v_L}{d}; \tag{4.19}$$

where d is the distance between the wheels, that is, the axis length.

The presence of the free front wheels could lower the accuracy of the model. Indeed, the velocity of the wheelchair also depends on their initial position. However, the approximation we have introduced has revealed to be sufficiently good for the purposes of this thesis.

The angular and the linear velocity values are published on the topic `speed`, which is read by the PID node. Even if a generic ROS message for velocity data already exists (`Twist`), that message type does not include any header, so we have created a custom one, called `Speed`, having the following fields:

- `header`: contains a sequence number for the message and its time stamp. The time stamp is important to associate the velocity measure to the encoder data that have generated it;
- `linear`: is the linear velocity of the odometric center of the robot along the x axis, in m/s;

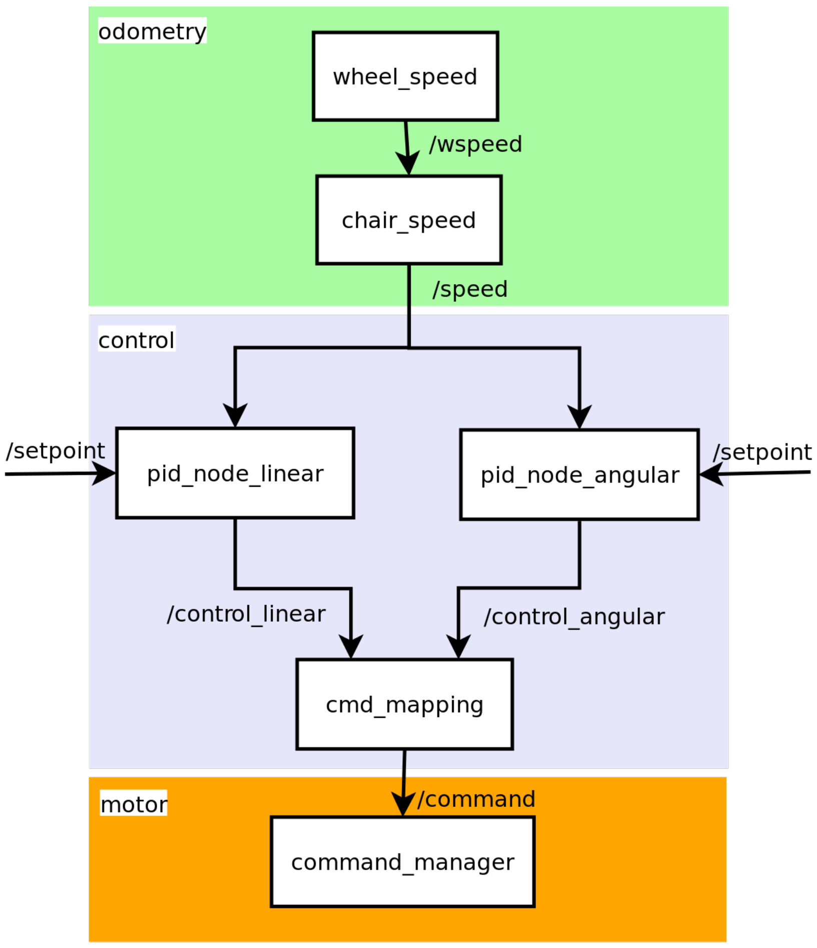


Figure 4.8: The motor interface and control architecture

- `angular`: is the angular velocity of the odometric center of the robot around the z axis, in rad/s.

Since the resolution of the encoders is quite low (about 0.28 m/s, see Chapter 3), the obtained velocity profile is not a smooth function of time, but rather it contains sharp variations that could alter the stability of the control system. For this reason, we have implemented a moving average, in order to give as feedback for the system the average of a pre-defined number of samples. This number is configurable; in the current implementation it is 5, therefore, at every iteration step of the PID controller, the last 5 samples of velocity measures are averaged and given as input to the controller. Figure 4.8 shows the software architecture for the motor interface and control subsystems.

4.5.4 PID tuning

The tuning of the parameters K_P, T_I and T_D has been done by using a graphical tool included in MATLAB. This tool, named *pidtool* [52], takes as input the transfer function between setpoints and measured output of the system in closed loop, and automatically tunes the parameters in accordance with adjustable options such as the response time and the transient behaviour, while visualizing the effect on the step-response function.

The transfer function has been estimated by using another MATLAB tool called *ident*, which, given a list of setpoints and their respective output in closed loop, computes an appropriate transfer function (whose number of poles and zeroes is customizable) that relates them. For this purpose, we recorded with *rosbag* the execution of a trajectory in which the wheelchair mainly moved forwards and backwards, in order to tune the PID for the linear velocity. We then stored the recorded setpoint and velocity values, that is, messages published on topics `setpoint` and `speed`, and gave them as input to the *ident* tool. We tuned the number of poles and zeroes in order to have a transfer function that better interpolated the samples in our data set.

The resulting function was then used with *pidtool* for tuning. The tool estimated the three PID parameters for the current settings, allowing to adjust them, either directly or by means of some options. The main options available are:

- Response Time: regulates the responsiveness of the system. A graphic slider allows to choose the appropriate value, which can make the closed-loop response of the control system faster or slower.
- Transient Behaviour: makes the controller more aggressive at disturbance rejection or more robust against uncertainty.

The obtained parameters were then set in the PID configuration. The same experiment has been repeated for the angular velocity (in that case, the data set was produced performing many rotations in both directions). However, with the parameters obtained through the MATLAB tools, the robot behaviour was not very good in terms of reactivity and robustness (there were still oscillations around the equilibrium). Our hypothesis is that the low accuracy of the encoders had introduced a measurement error that had made the measured output velocity slightly different from the real one. In order to improve the control system behaviour we have made some modifications to the estimated parameters, basing on the general assumptions that are summarize in Table 4.1 from [51]. The tuning has been done at run-

Parameter increased	**Response speed**	**Response stability**
K_P	Increases	Decreases
T_I	Decreases	Increases
T_D	Increases	Increases (only in ideal conditions, with zero noise)

Table 4.1: General effects of the PID parameters on the control system

Parameter	**Linear velocity**	**Angular velocity**
K_P	1	-0.4
T_I	0.59	0.29802
T_D	0.1475	0

Table 4.2: The current configuration of the PID

time, setting parameters by means of ROS command `rosparam set`. The current configuration is shown in Table 4.2.

The proportional gain K_P for the angular velocity controller is negative because the direction in which the angle around z increases is opposite to the command given to the motor board. In other words, a positive command value on the left-right axis makes the wheelchair turn to the right, thus producing a negative angular velocity.

We clarify that the control system, and LURCH software in general, has been tested with only one gear (*4*); obviously, changing the gear requires to use other parameters for the two PIDs.

4.6 Command devices and collision avoidance

Commands can be given to the wheelchair in different ways. As seen in Section 4.5.1, the position of the on-board joystick is retrieved from the motor board and published on a dedicated topic, allowing to reuse it via software. This lets the system modify given commands in accordance to the state of the robot, for instance the presence of obstacles on the way. The same principle can be applied to an arbitrary number of interfaces.

4.6.1 Joypad interface

In order to introduce a joypad in our architecture, the first task was adding a driver for it. Fortunately, ROS already provides a node that deals with some joypad models, and the Logitech F710 is among them. The node, named `joy_node`, is responsible for reading joystick commands and translating them into ROS messages. The resulting message is published on a topic named `joy`, and contains an array in which each element corresponds to a button or stick on the joypad. Digital button values can be either 0 or 1, while analog stick values are decimal, and range between -1 and +1. When a digital button is pressed, the correspondent value becomes 1, otherwise it is 0. On the other hand, if an analog stick is moved two elements are involved: one for the forward-backward axis and the other for the left-right axis. Values are 0 when the handle is placed in the central position. Messages are not published periodically; they are published only if the state of the joypad has changed (that is, when a button has been pressed or released, or when an analog stick has been moved).

We have created a node, named `joypad_node`, which maps joypad states to commands for the wheelchair. Since the analog sticks have a shape and usability that are similar to those of the on-board joystick, they are the most suitable input methods for driving the wheelchair. Moreover, in `joy_node`, analog sticks are given values between -1 and +1, exactly as in our `Motion` message type. For these reasons, an analog stick has been chosen for driving; in particular, the left one. However, since `joy_node` publishes messages only when the joypad state changes, simply translating joypad values into `Motion` messages would not be sufficient. In fact, it would be very difficult to keep the same command for a period of time, and the wheelchair actuators would not respond visibly. Hence, we have decided to store the last joypad state in our node, and publish it periodically (50 Hz, like the output frequency of the motor board). Messages are published on a topic called `joypad`.

4.6.2 Collision avoidance

Commands given through the on-board joystick and the joypad are read by a node called `obstacle_avoidance`. This node has three main tasks:

1. managing the mutual exclusion between input devices;
2. scaling commands in order to limit velocity;
3. using information on obstacles to adjust commands if assisted drive mode is on.

The first task of this node is managing the mutual exclusion between the input devices. The selection of the command device is done automatically, without making the user choose the preferred input modality manually. Following the choice that had been made for the previous software architecture [2], it has been decided to give maximum priority to the on-board joystick. This entails that, if the joystick and the joypad are both sending commands, only joystick commands will be executed. In order to check if a device is sending commands, the correspondent values are checked: basically, if a handle (joypad or joystick) is placed in central position (`fwrw` and `rxlx` near zero) we assume that it is not sending commands.

Applying scale factors allows to limit the maximum velocity of the wheelchair. We have chosen to use gear *4* for the wheelchair low-level controller, which is the second highest gear, because it is a good balance between braking power and security (with that configuration the wheelchair is physically unable to run at maximum velocity). Moreover, with gear *4*, the motors provide enough power to transport a person of average weight. In all our tests, we have preferred to limit the velocity via software, in order to increase security. Hence command values are divided by a scale factor (in the current implementation, both forward-backward and left-right commands are divided by 1.5, which means that velocity is reduced by 33%).

To react to the presence of obstacles, a node that translates laser readings into higher-level information on obstacles has been introduced. This node is called `obstacle_detector`. It subscribes to topics `scan_left` and `scan_right`, so that it can access to all laser data. The scanned area has been divided in three partitions: left, right, and front. Within each partition, the distance from the closest obstacle detected is considered. As you can see in Figure 4.9, front and side partitions overlap, so that, if an obstacle is detected on the front and on one side at the same time, movements in both directions can be blocked. Of course, frontal area is covered by both laser scanners; this increases robustness against obstacles that are located in the main movement direction.

The minimum distance from obstacles that are detected in the three partitions is then published as a custom message named `Obstacle`. This message has four fields:

- `header`: sequence number and time stamp of the message;
- `front`: minimum distance (m) from obstacles detected on the front;
- `left`: minimum distance (m) from obstacles detected on the left;
- `right`: minimum distance (m) from obstacles detected on the right.

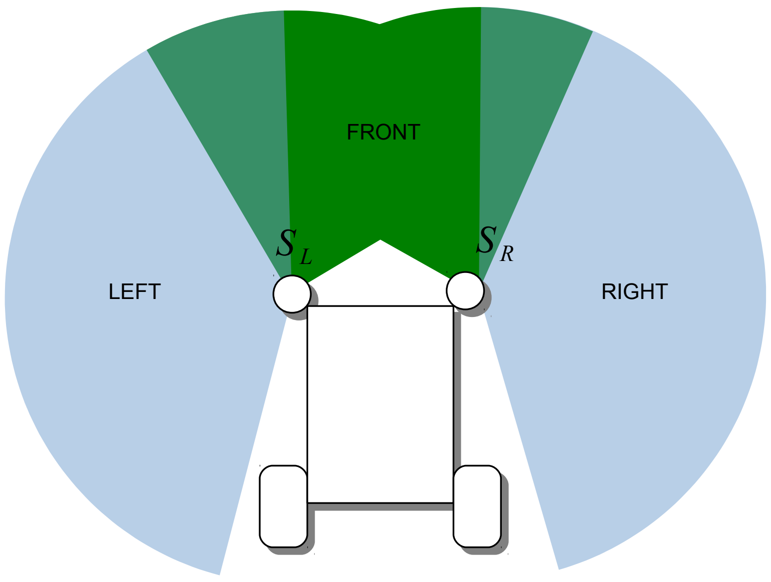


Figure 4.9: A qualitative representation of the laser scan partitions for obstacle detection

`Obstacle` messages are published on topic `obstacle`, which is read by node `obstacle_avoidance`. This node modifies the received commands in accordance with the distance values; in particular, it applies a distance threshold within which, if an obstacle is detected, the wheelchair must stop. In order to make the wheelchair slow down when an obstacle is approaching on its way, another threshold has been added, within which the wheelchair must limit its velocity, so that it can stop in time. Currently, the wheelchair slows down when an obstacle is located within 1.10 m, and stops when an obstacle is at 0.6 m or nearer. Since the wheelchair angular velocity is generally low, only one threshold has been set for rotations: left and/or right rotations are blocked if an obstacle is detected at 0.5 m or less on the respective sides.

If the robot is in assisted drive state, meaning that it must avoid collisions while it is driven by the user, the resulting command is translated into a setpoint for the PID controllers, so that the robot adjusts its speed properly; on the other hand, if obstacle avoidance features are off, commands are published directly on topic `command` and sent to the motor board, in order to leave the complete control to the user. The management of the robot states is treated in Section 4.8.

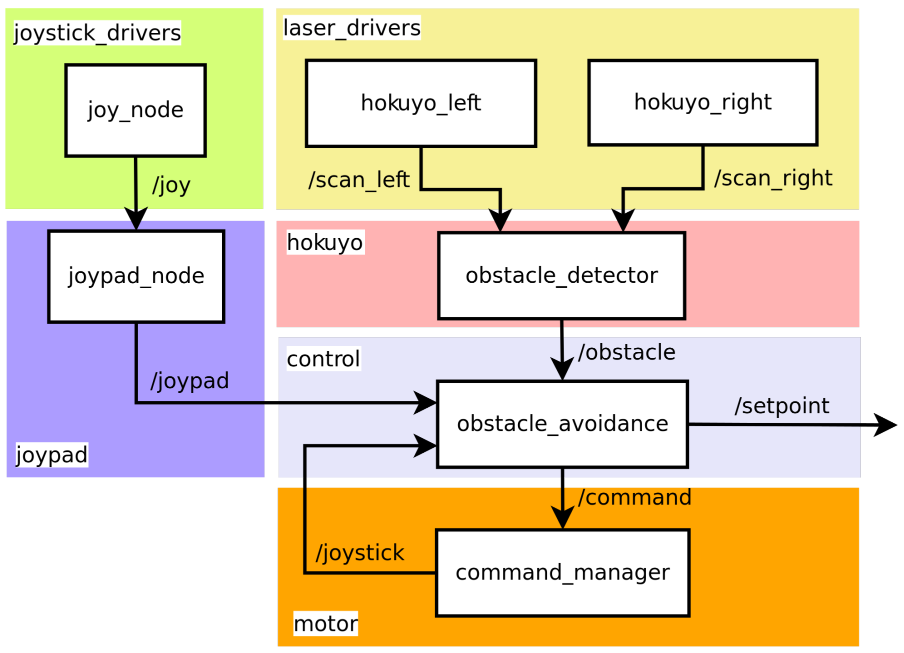


Figure 4.10: The collision avoidance architecture

Figure 4.10 shows the architecture for obstacle avoidance features.

4.7 Autonomous drive

The autonomous driving feature has been added in order to test the effectiveness of the localization and control subsystems, while providing a first view of the possibilities offered by the ROS framework on an autonomous wheelchair. For this reason, we have adopted the Navigation Stack and configured it for our case. However, the current architecture allows to substitute the motion and path planner with any other implementation without having to modify existing nodes.

4.7.1 Path planner and motion planner

As seen in Chapter 2, the path planner and the motion planner provided by ROS are included in a package called `move_base`. This package provides a global planner and a local planner. The global planner is basically the equivalent of a path planner: it searches for a path from the robot starting position to a goal position that has been published on topic `goal`, considering visible and mapped obstacles.

Once the path has been computed, the local planner simulates many possible trajectories, which take into account the environment and the current speed of the robot. This is done by creating, locally around the robot, a value function represented as a grid map built with laser scanners. This value function encodes the costs of traversing the grid cells; basically, the cost of a cell is maximum (255) when the cell is occupied by an obstacle, and minimum (0) if it is far from obstacles. The trajectory that minimizes the following cost function is chosen:

$$cost = P \cdot pdist + G \cdot gdist + O \cdot occdist; \tag{4.20}$$

where *pdist* is the distance to the planned path from the endpoint of the trajectory in meters, *gdist* is the distance to the goal from the endpoint of the trajectory in meters, and *occdist* is the maximum obstacle cost along the trajectory. P, G and O are scale factors that determine the weight of each term. We have chosen: $P = 0.6$, $G = 0.8$, and $O = 0.01$. With these values the robot gives, in general, high importance to obstacles; on the contrary, increasing P and G, or decreasing O, results in a more aggressive behaviour, allowing the wheelchair to pass closer to obstacles and reach the goal faster.

The output of `move_base` is a `Twist` message, named `cmd_vel`, which is composed of a linear and an angular velocity setpoint for the controller (in our case, the PID). This setpoint is computed considering the acceleration limits of the robot and some velocity bounds which are configurable. The current speed of the robot is communicated by means of an `Odometry` message. ROS type `Odometry` is composed of both `Pose` and `Twist`; so we have created a node, called `odometry_node`, that is responsible for merging messages coming from `fusion` and `chair_speed` and providing a proper `Odometry` message. This node might also be reused in future for other planners that adopt ROS conventions. Since trajectories are evaluated periodically while the robot is moving, the motion planner is able to react to dynamic obstacles.

4.7.2 Mapping

Although `move_base` could be run in the ROAMFREE *world* frame, the odometry drift, caused by the absence of an absolute position sensor, makes it difficult to reach the goal position successfully. Indeed, in every odometry system, the localization error accumulates over time. In order to correct that error, a map-based algorithm, called *AMCL*, has been included. The basic operating principles of this method have been explained in Chapter 2.

AMCL subscribes to the following topics:

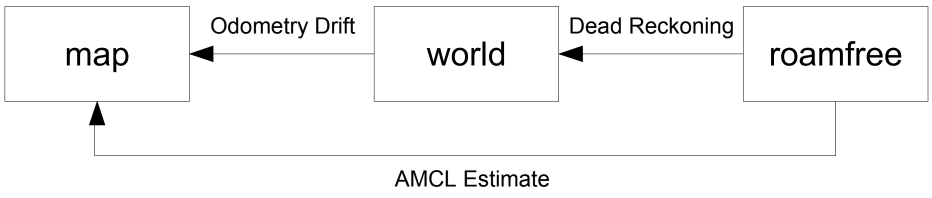


Figure 4.11: AMCL localization

- `scan`: a topic providing laser scans. Since there cannot be more than one instance of such topic, scans belonging to the two laser scanners, `hokuyo_left` and `hokuyo_right`, have been merged in a single topic by means of a node, named `whole_scan`. The two scans are distinguished thanks to the field `frame_id`, which allows to associate each scan to its correct reference frame.

- `tf`: the list of reference frames and transforms between them. In particular, the transform between the odometry frame and the robot base frame, and that between the robot base frame and the sensor frames must be provided. In our case, ROAMFREE handles this task.

- `initialpose`: an initial pose estimate, from which to initialize the particle filter.

- `map`: a map of the environment.

Given a map, laser data, and the roto-translation between the odometry frame and the robot base frame, AMCL estimates the position of the robot with respect to the map frame. In our case, the odometry frame corresponds to frame W (*world*), and the base frame corresponds to frame O (*roamfree*) (Figure 4.11). The algorithm adjusts the roto-translation between frames *map* and *world*, in order to compensate the odometry error.

Maps are obtained by means of the `gmapping` package, along with laser measurements. To build a map with `gmapping`, the wheelchair is driven in the environment with the joypad to collect data on static obstacles and boundaries, while recording the output of a laser scanner with *rosbag*. Given the recorded data, the output of `gmapping` is an image where:

- black pixels indicate that the correspondent areas are certainly occupied;

- white pixels indicate areas that are certainly free;

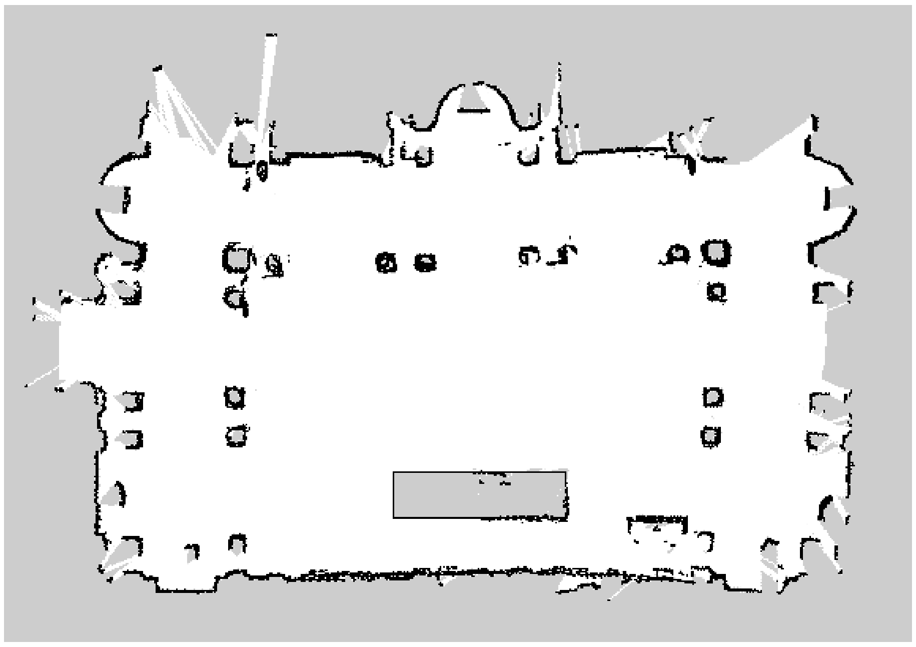


Figure 4.12: An example of map generated by gmapping *for LURCH*

- grey pixels indicate unknown areas; the shade reflects the probability of the areas of being occupied.

The map is then saved and provided at runtime by a ROS node named `map_server`. An example of map obtained with this method is shown in Figure 4.12.

AMCL requires a good initialization; indeed, if the initial pose estimate is too far from the real pose of the robot, the algorithm could take too much time to converge. A way to set the initial pose of the robot is via *rviz*. In the graphical program, the initial pose can be set by placing a vector on a point of the map. Accuracy in setting the initial pose estimate is not fundamental, since the algorithm tends to correct the robot pose according to the laser data it acquires while moving. The goal for the planner can be set in a similar way. These operations can also be done on board by using the touch screen.

In figure below, the robot internal representation of an environment is shown. The red rectangle is the robot footprint, that is the area physically occupied by the robot; we have slightly increased its size with respect to the real one, in order to have a security range. White points represent the actual laser data; red cells represent obstacles. Obstacles are inflated by the inscribed radius of the robot (orange cells). For the robot to avoid collision,

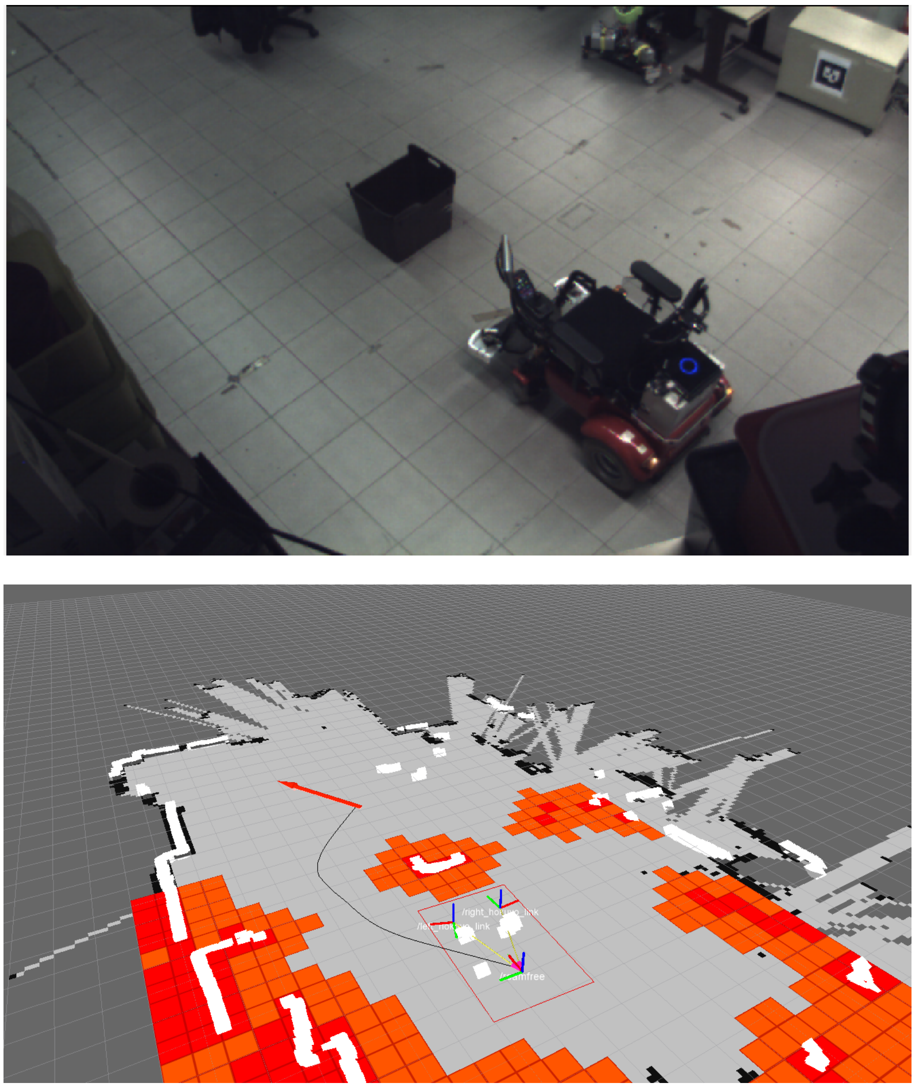


Figure 4.13: A real case (above) compared with the robot internal representation (below)

the center point of the robot must never overlap with a cell that contains an inflated obstacle. The red vector indicates a goal position and orientation, while the black line represents the path computed by the global planner.

4.7.3 Security measures

In order to handle cases in which the wheelchair becomes uncontrollable, or an obstacle is not added in the cost map in time as a consequence of some malfunction, we have added an additional collision avoidance layer. The cor-

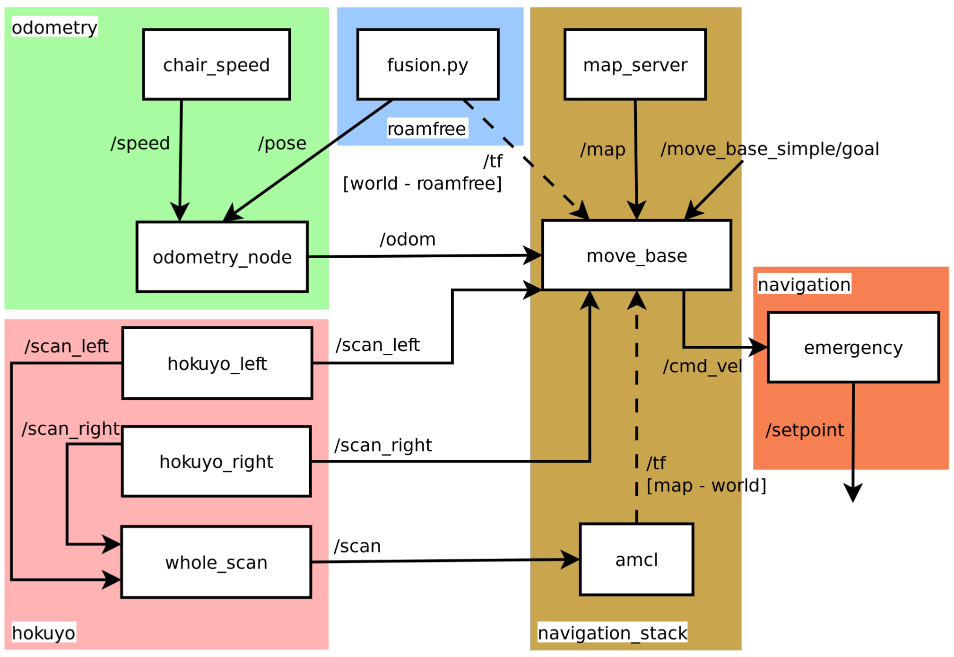


Figure 4.14: The autonomous drive architecture

responding node, named `emergency`, is logically placed between `move_base` and the PID controllers; in this way, every velocity command sent by the autonomous navigation layer must be filtered by the node before being converted in a setpoint for the control system. The main task of this node is subscribing to the output of `move_base` (topic `cmd_vel`) and to topic `obstacle`. If the distance from an obstacle is below a certain threshold, which is configurable, a 0 velocity setpoint is sent to the controllers, making the wheelchair stop if it was moving; otherwise, the setpoint becomes equal to the velocity command. This node can be reused in future to implement further security measures that operate on the planner output.

4.8 State management

With many possible operating modes, a system that keeps track of the robot state is needed, which carries out all the required actions when the state changes. We have adopted a package previously developed by Politecnico di Milano, called `heartbeat`, that implements a finite state machine designed for robots. The states defined by such package are: MANUAL, ASSISTED, AUTO, SAFE, and HALT. These states correspond to the modes in which the robot can operate: when the robot is in MANUAL state, it is driven

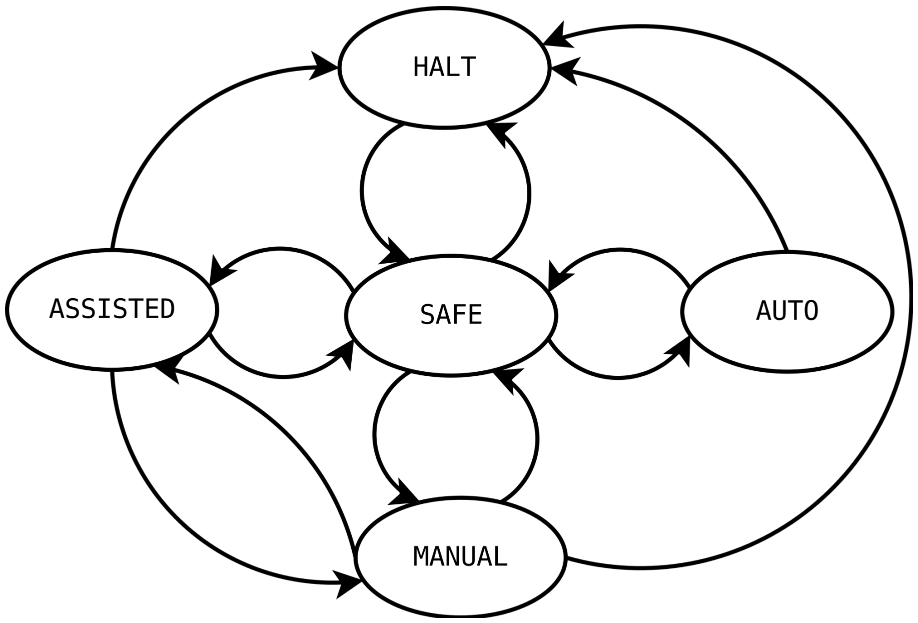


Figure 4.15: The finite state machine implemented on LURCH

without active intervention of the control software; in ASSISTED state, software features for user's assistance, like collision avoidance, are activated; in AUTO state, autonomous drive features are activated. SAFE and HALT are security states: in SAFE state the robot stays still, but can be reactivated via software; in HALT state, the robot stays still as in SAFE state, but it can only be reactivated by physically operating on the robot hardware. The transition function is defined in the following way:

- from HALT only transition to SAFE is allowed;
- from MANUAL every transition except the one to AUTO is allowed;
- from SAFE every transition is allowed;
- from ASSISTED every transition except the one to AUTO is allowed;
- from AUTO only transitions to SAFE and to HALT are allowed.

In Figure 4.15 the transition function is represented.

The package uses a client-server paradigm: a server node keeps track of the current state, and manages the state changes. Every node can retrieve the current state, or ask for a change.

We have used the finite state machine with LURCH in the following way.

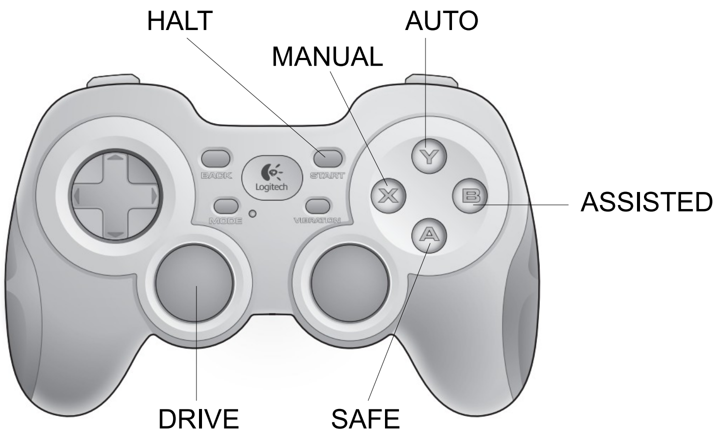


Figure 4.16: Joypad controls

- MANUAL: when the robot is in this state, it can be driven with the on-board joystick or the joypad alike; all commands are sent directly to the motor board without passing through the PID controllers.
- ASSISTED: the commands given with the joystick or the joypad are filtered by the obstacle avoidance node and are sent to the PID controllers.
- AUTO: the robot moves autonomously toward goals set by the user.
- SAFE: the robot cannot move without selecting one of the states above.
- HALT: all messages sent to the motor board are assigned mode 2. This makes the motor board go into *driving disabled* operating mode. The only way to reactivate the robot from this state is by pushing the dedicated physical button located on the wheelchair (Chapter 3).

Whenever the state changes, variables related to the previous state are reinitialized. In particular:

- setpoints are zeroed;
- if previous state was AUTO, the current goal is cancelled.

States can be set by using the joypad; every state is assigned a specific button (see Figure 4.16). Figure 4.17 shows the overall software architecture.

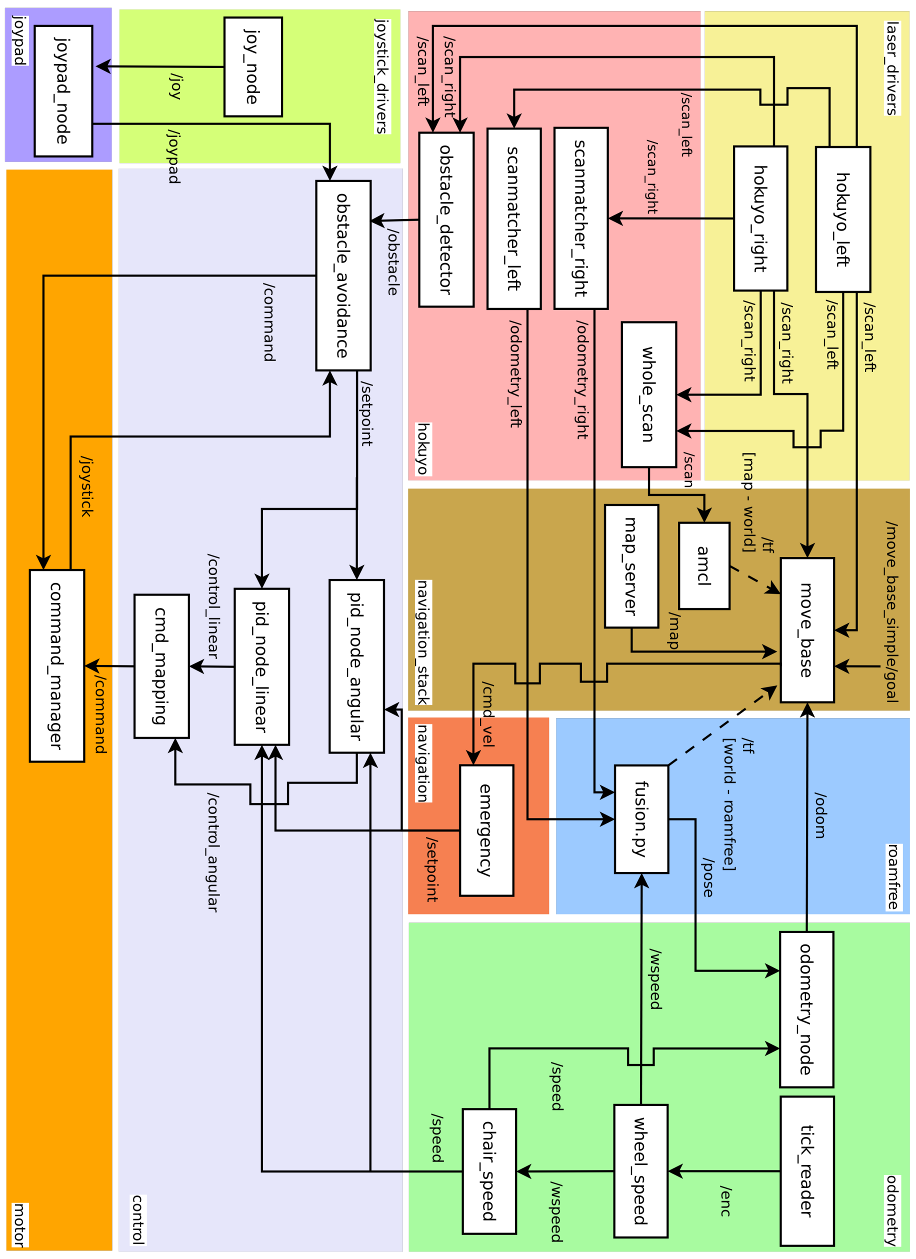


Figure 4.17: The overall software architecture

Chapter 5

Experimental Results

In this chapter we test the localization performances and the autonomous behaviour of the robot, and we analyse the obtained results.

5.1 Localization

In order to test the localization performances of the multi-sensor fusion engine, we used the same OptiTrack cameras we employed for calibration of the ROAMFREE parameters as a reference(Chapter 4). The position of the robot was initialized with the absolute position of the robot given by the motion capture system in $t = 0$. Then we analysed the differences between the trajectory estimated by ROAMFREE and the ground truth, obtained while driving the wheelchair along random paths by means of the joypad. A graph showing a comparison between localization data provided by the two sources for an example path of 43 seconds is presented in Figure 5.1. Figure 5.2 shows the evolution of the error, computed as Euclidean distance between the two measurements at each time stamp. The average error is 0.1367 m, while the maximum error is 0.3248 m.

The example shows that ROAMFREE estimates tend to diverge immediately from the motion capture output. A possible cause could be the approximation introduced by treating the kinematic model of the wheelchair as an ideal differential drive system; a more sophisticated model which, for instance, takes into account the position and orientation of the front wheels could enhance the velocity estimate.

The fact that near the end of the path the estimation error is lower is a coincidence: in general, the odometry error tends to increase. Indeed, the odometry error at time t depends not only on the noise on current sensor

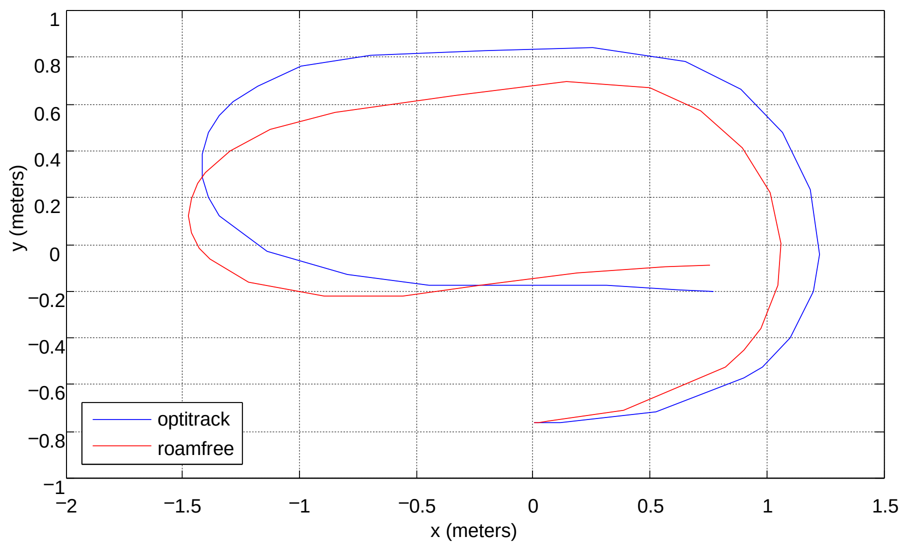


Figure 5.1: Comparison between ROAMFREE pose estimation and the absolute poses provided by OptiTrack cameras

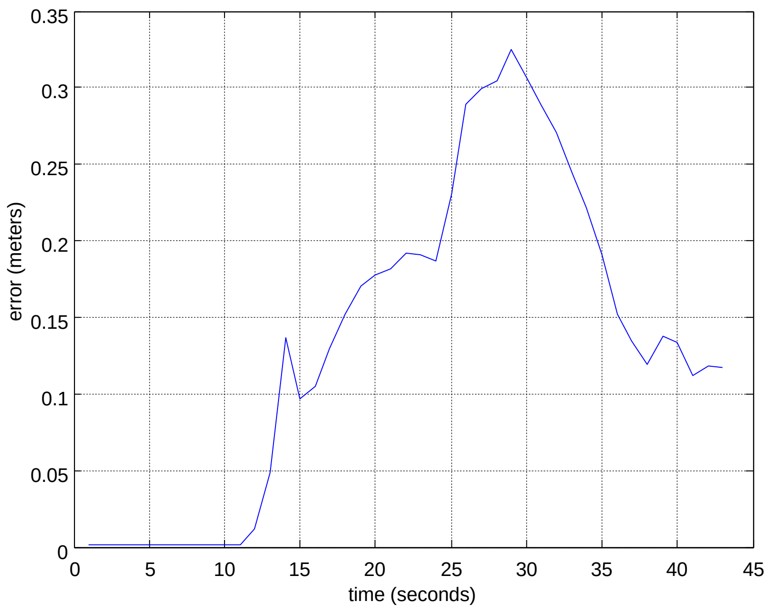


Figure 5.2: Localization error with respect to time

readings, but also on the error at time $t-1$. By using multi-sensor fusion, ROAMFREE is able to reduce the error introduced by sensor noise; however, in order to reduce the overall positioning error, an absolute position reference is needed. To this end, AMCL has been included. This algorithm has proven to reduce the error introduced by odometry, by adjusting the roto-translation between a fixed frame *map* and the odometry reference frame (which in our case is frame *world*). This is done by comparing the laser scans with a static map of the environment. We have noted that the effectiveness of this algorithm increases when the environment includes static features with distinguishable shapes (e.g., walls, nooks, pillars, furniture), while open areas make the algorithm more uncertain on the robot position. It is also important that, in every moment, laser scanners detect some obstacles within their range; note, however, that this issue is critical only for AMCL, while ROAMFREE is able to compensate the lack of information given by the scan-matchers with the odometry data obtained through the encoders.

In most cases it is necessary to drive the robot around for 1-2 minutes before having the sample distribution of AMCL converge to the real pose; however, the convergence time may vary, depending on the shape of the room, the accuracy of the initial pose estimate, and the trajectory performed (in general, moving the robot near walls and objects that are noticeable on the map can speed up convergence).

5.2 Autonomous navigation

The autonomous drive system allows to plan paths and follow them. As seen in the previous chapter, goals can be easily assigned through the *rviz* graphical interface, by selecting a specific point on the map. Indeed, when a goal point is selected, a message on topic `/move_base_simple/goal` is published, and subsequently read by `move_base`.

5.2.1 Static obstacles

One of the main requirements for LURCH is collision avoidance. In order to test the reliability of our system against obstacles, we have done several laboratory experiments. The map used for our experiments, obtained with `gmapping`, is shown in Figure 5.3.

We placed a basket in the middle of the room and we asked the robot to reach a goal located beyond it (Figure 5.4). We repeated the test 7 times, starting approximately from the same position and giving the same goal

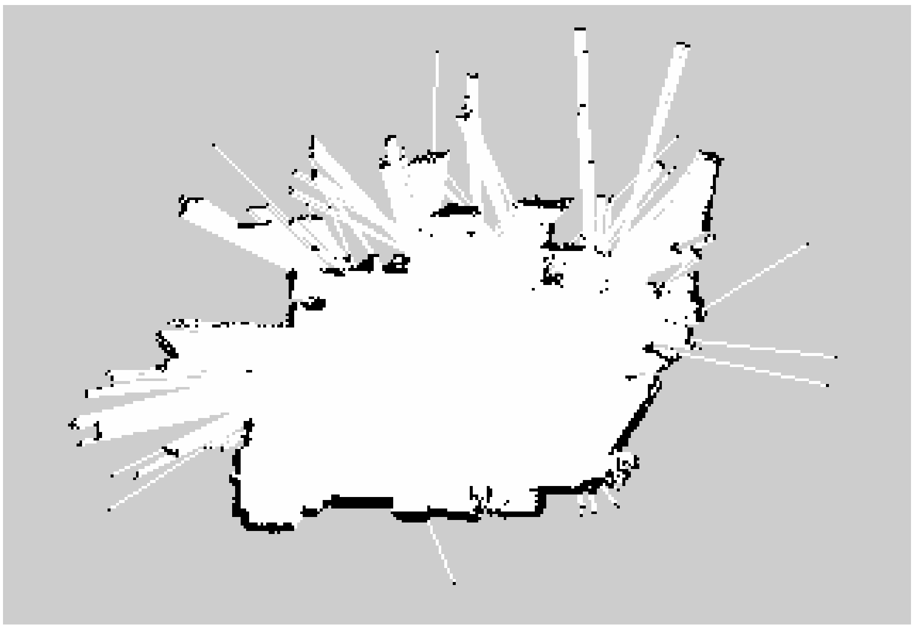


Figure 5.3: Map of the laboratory area used for experiments with LURCH

through *rviz*. The robot used both ROAMFREE and AMCL for localization.

The resulted trajectories are shown in Figures 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, and 5.11. We considered a goal succeeded if it was located within a range of 40 cm from the robot position. The blue line represents the global plan, while the red line is the actual robot trajectory (positions are given in meters). An arrow indicates the robot starting position and orientation. Table 5.1 shows the time required for reaching the goal in each test.

In all tests, the robot has successfully planned a path and reached the goal. However, in some cases, like in tests 2, 4, and 6 (Figures 5.6, 5.8 and 5.10), it had some difficulties in passing between the obstacle and the wall beside it, even if the space was physically sufficient. In these cases, the robot did not immediately proceed trough the passage; instead, it turned in place in search of free space, and tried other trajectories (this is the standard behaviour adopted by `move_base` when a clear path is not found). This behaviour is probably related to the configuration of the navigation parameters, in particular *occdist*, *gdist* and *pdist* factors for trajectory evaluation (Chapter 4). Indeed, the general result we have obtained is having a "cautious" robot, which rarely hits obstacles, but as a consequence needs more space around them.

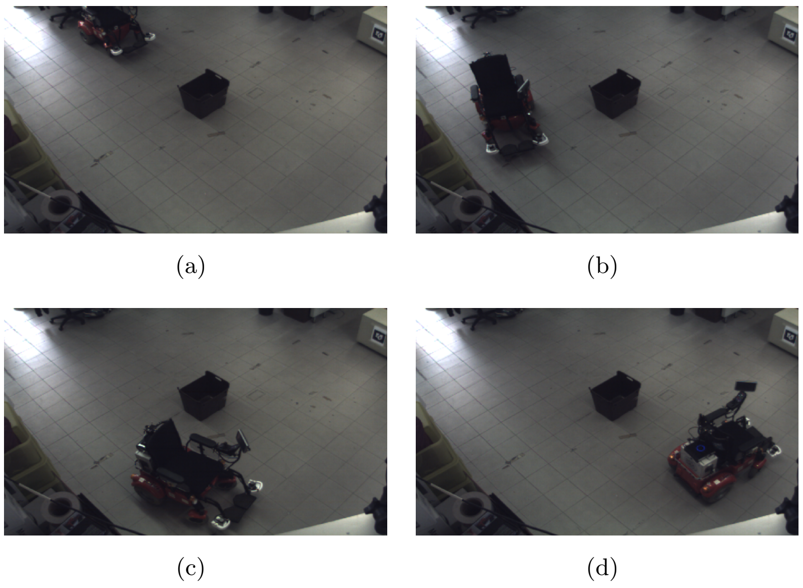


Figure 5.4: An example of trajectory followed by LURCH to avoid a static obstacle

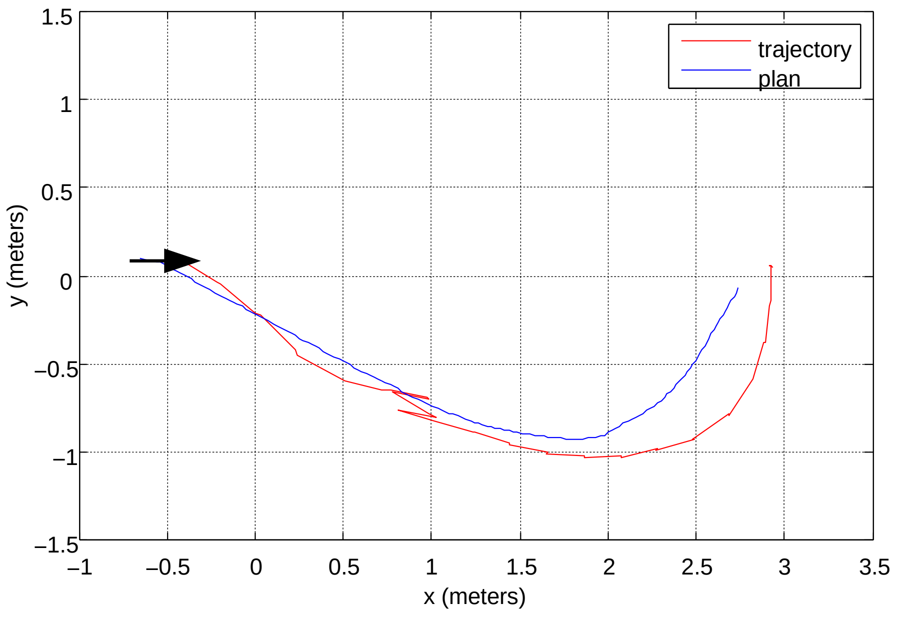


Figure 5.5: Static obstacle avoidance: test 1

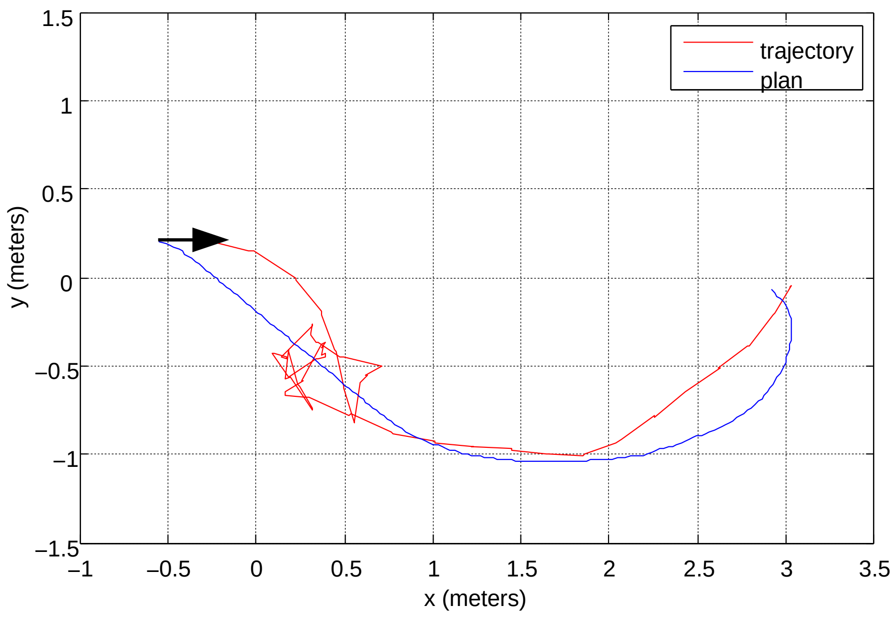


Figure 5.6: Static obstacle avoidance: test 2

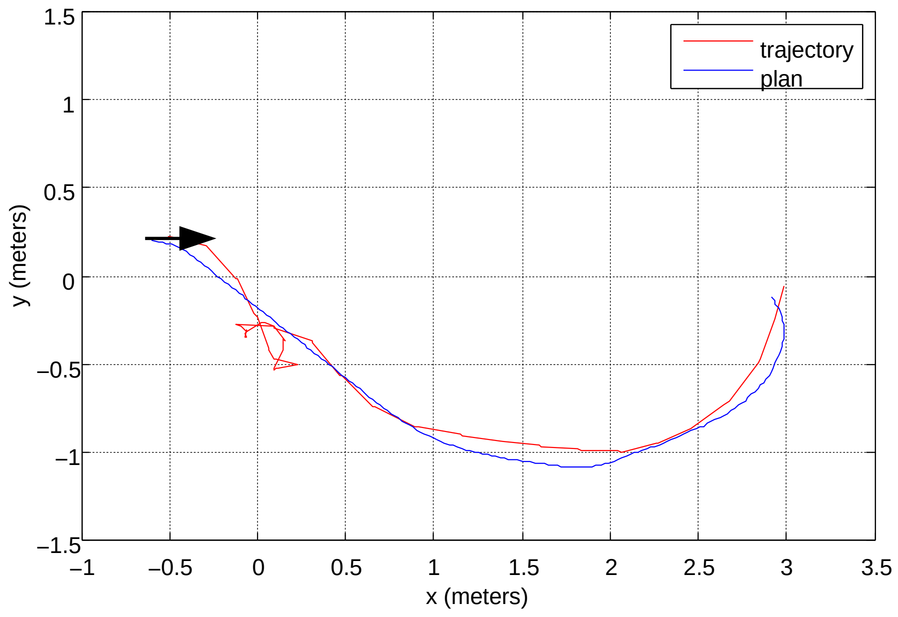


Figure 5.7: Static obstacle avoidance: test 3

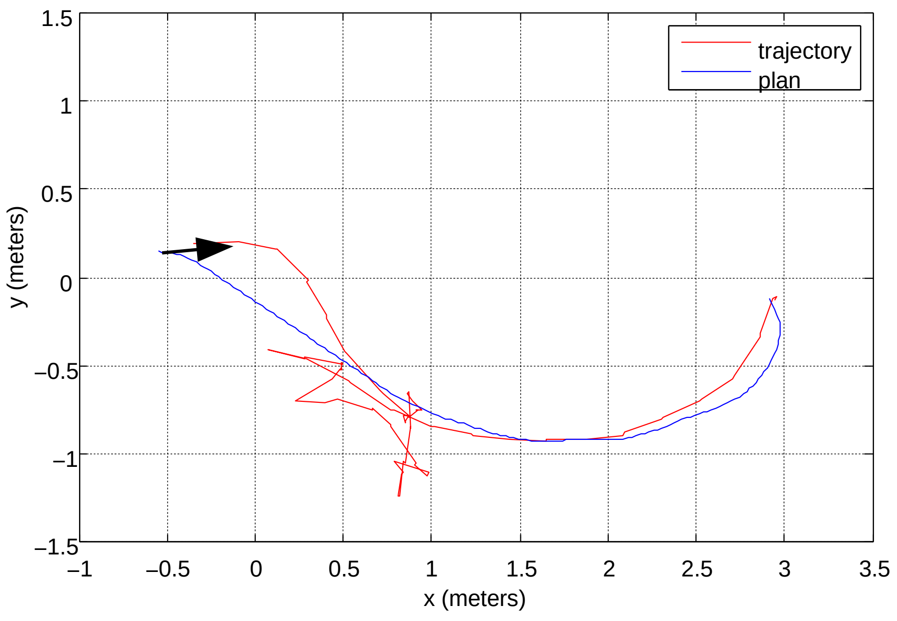


Figure 5.8: Static obstacle avoidance: test 4

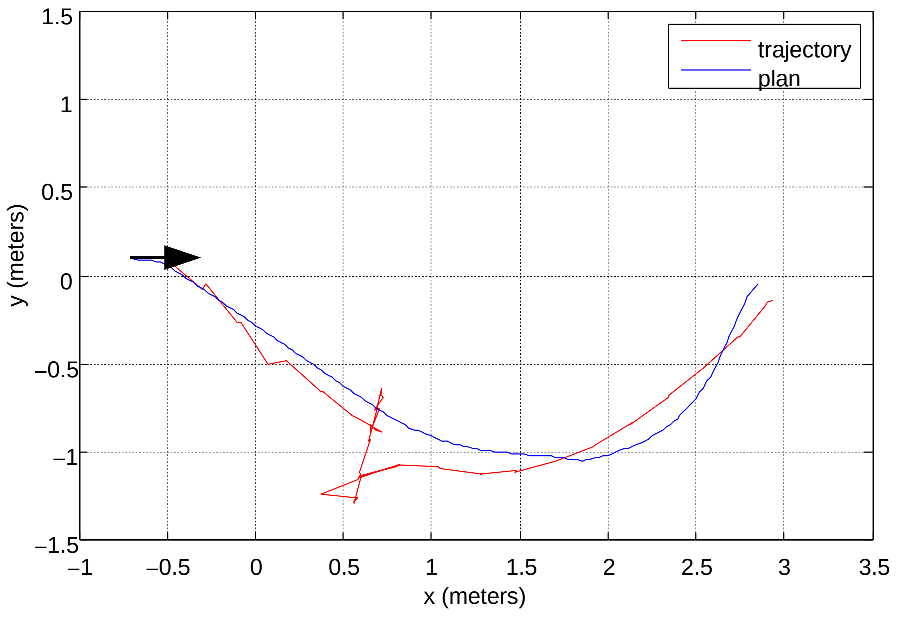


Figure 5.9: Static obstacle avoidance: test 5

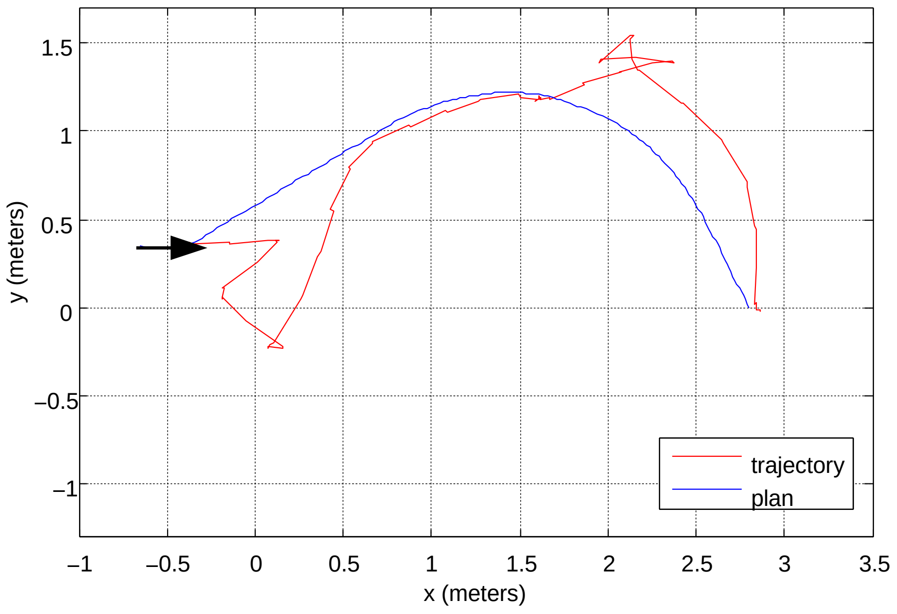


Figure 5.10: Static obstacle avoidance: test 6

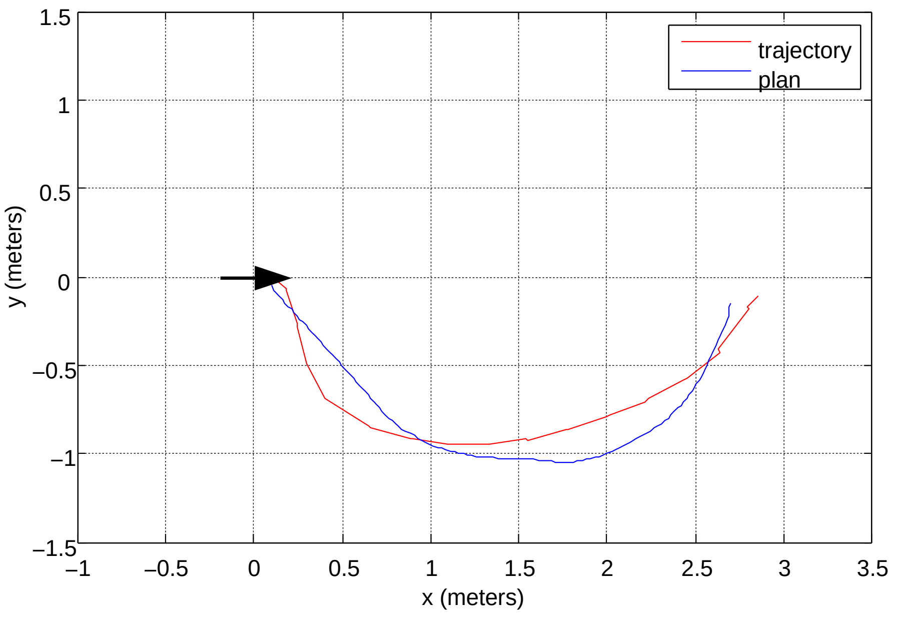


Figure 5.11: Static obstacle avoidance: test 7

Test	Time
Test 1	39.329 s
Test 2	85.674 s
Test 3	55.693 s
Test 4	91.318 s
Test 5	69.617 s
Test 6	85.980 s
Test 7	18.757 s

Table 5.1: Static obstacle avoidance: time required to reach the goal

The considerations we made above also hold for the following test case. We placed some objects in a row, in order to simulate a short corridor. Then we placed a goal on the other side of the row. The robot had to follow the corridor till the end, and then perform an U-shaped trajectory to reach the goal (Figure 5.12).

The resulting trajectories are shown in Figures 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, and 5.19. The respective times employed for reaching the goal are shown in Table 5.2.

As it can be seen from the diagrams, LURCH succeeded in reaching the goal also in this more complex scenario. The plan has always been computed correctly. Nevertheless, we can clearly see that in tests 4, 5, and 7, represented in Figures 5.16, 5.17, and 5.19, the robot had to move around for a long time before finding a clear way to reach the goal; it took 74.676 s, 155.781 s and 74.489 s to reach the goal, respectively. In tests 1, 2, and 6 the robot managed to follow the path smoothly; in tests 5 and 7, it made many attempts before completing the turn; in tests 3 and 4, it had some difficulties with the wall encountered right after the turn.

In conclusion, the performance of autonomous navigation with static obstacles is acceptable and satisfactory in terms of both path planning and path following. The planner is able to plan paths that avoid obstacles, and generate appropriate velocity commands for the robot. Movements are sufficiently fluid, indicating that the generated commands are consistent with the acceleration limits of the robot, and that the PID controllers that we have implemented work properly. However, some optimizations are required, in order to allow the robot to pass through tight passages, like narrow corridors or doorways.

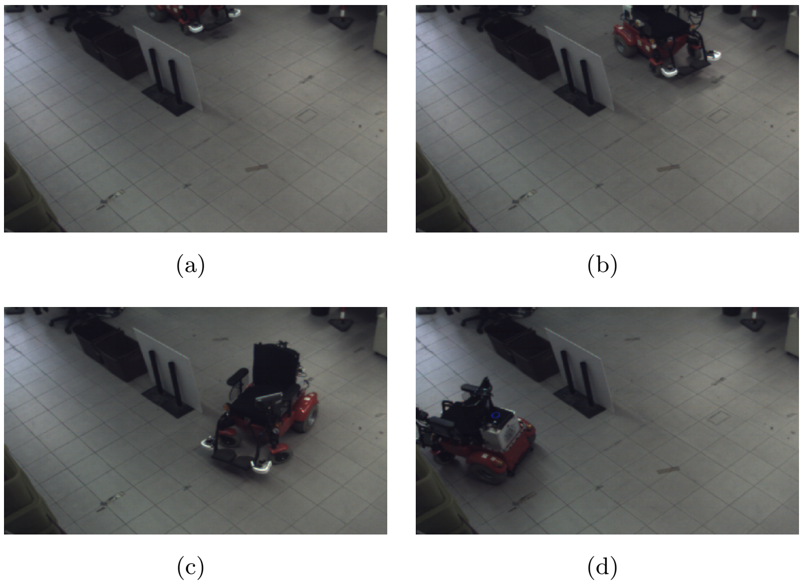


Figure 5.12: LURCH following an U-shaped path

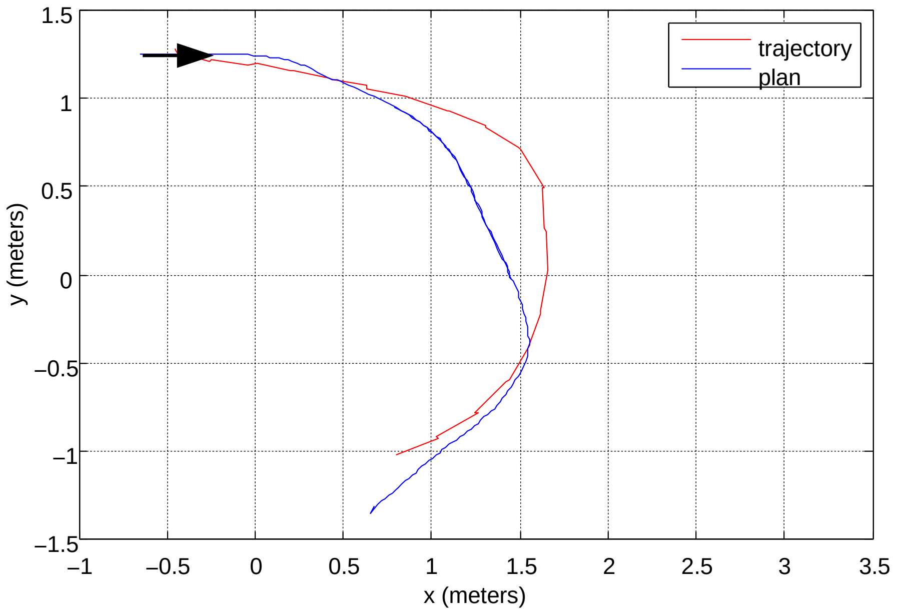


Figure 5.13: U-shaped path: test 1

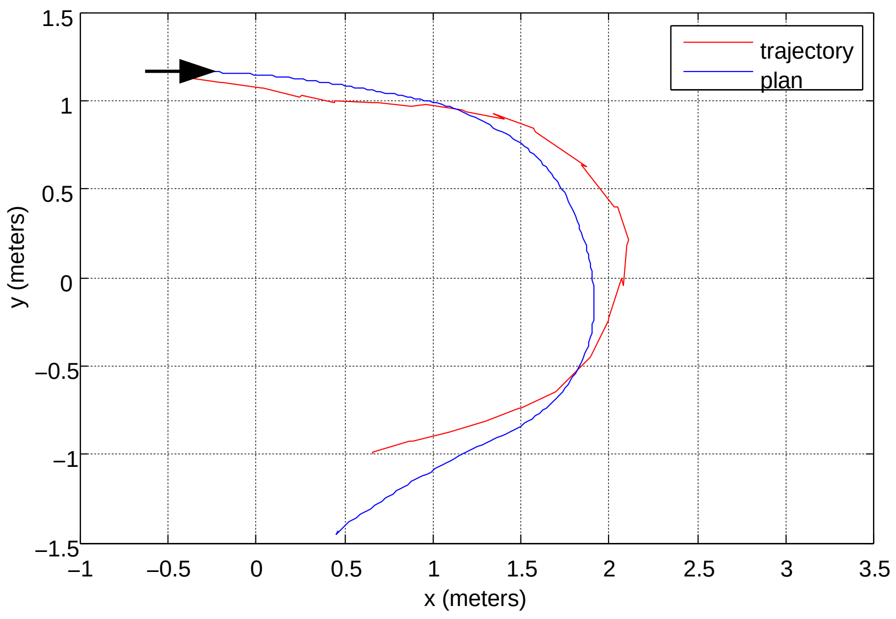


Figure 5.14: U-shaped path: test 2

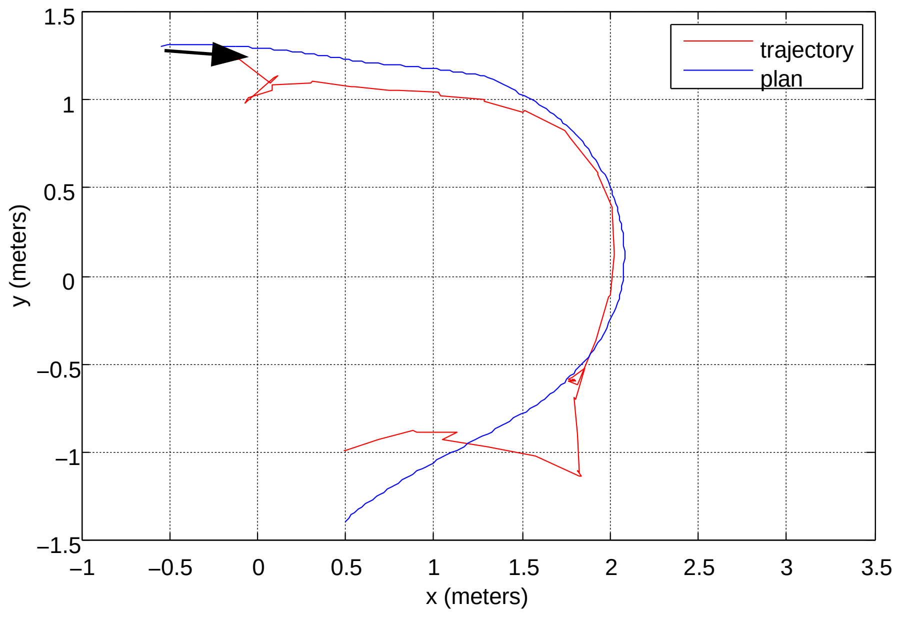


Figure 5.15: U-shaped path: test 3

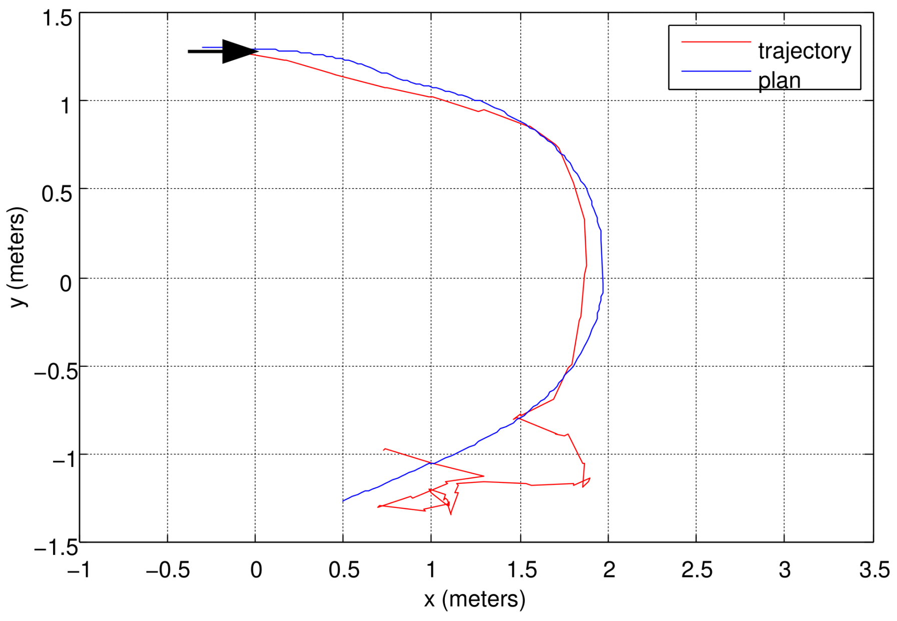


Figure 5.16: U-shaped path: test 4

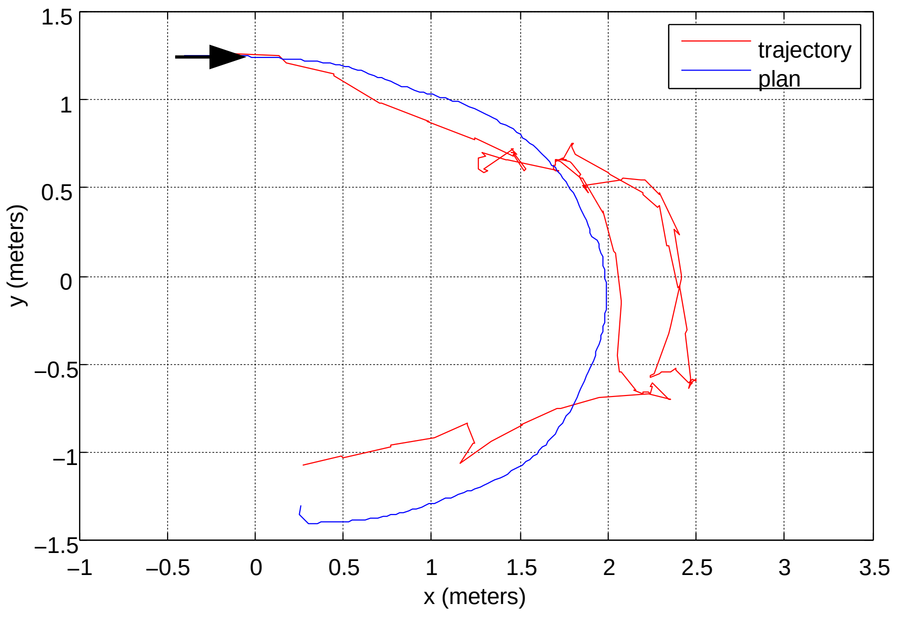


Figure 5.17: U-shaped path: test 5

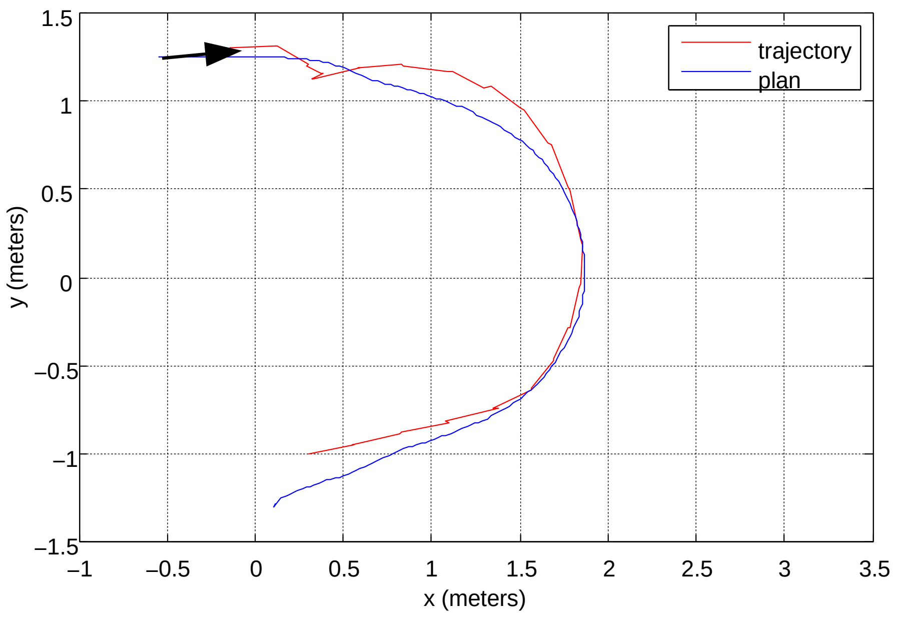


Figure 5.18: U-shaped path: test 6

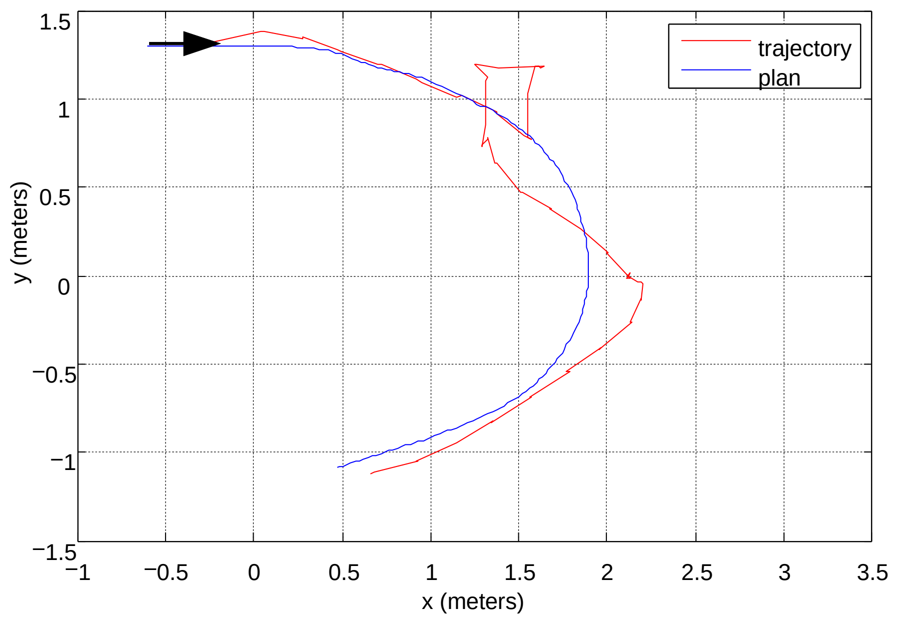


Figure 5.19: U-shaped path: test 7

Test	Time
Test 1	18.550 s
Test 2	16.710 s
Test 3	46.390 s
Test 4	74.676 s
Test 5	155.781 s
Test 6	25.376 s
Test 7	74.489 s

Table 5.2: U-shaped path: time required to reach the goal

5.2.2 Dynamic obstacles

In `move_base` the local planner chooses the most suitable action in accordance with the obstacles it sees in a cost map that is constantly updated. We have set an update frequency for the cost map equal to 10 Hz, which corresponds to the frequency at which the Hokuyo laser scanners perform a complete scan. This rate is high enough to allow the robot to notice unexpected obstacles, and react to them by avoiding collisions. Since the cost map is updated very fast, the reaction time mainly depends on the reactivity of the PID controllers. In order to test the reactivity of the robot against dynamic obstacles, we have performed several experiments in which obstacles were added suddenly along the path that the robot was trying to follow. An example is shown in Figure 5.20. First (Figure 5.20(a)), the robot was given a goal located on the other side of the room, computed a path to reach it and started following it; a person approached rapidly (Figure 5.20(b)); the robot stopped immediately (Figure 5.20(c)) and then continued by avoiding the new obstacle (Figure 5.20(d)). The correspondent plan and trajectory are shown in Figure 5.21. As it can be seen, initially the robot tried a more straight path than the one computed by the global planner, because the local planner had found a trajectory that resulted in positions closer to the goal. However, the unexpected obstacle made the robot change its trajectory, since the previous one was no more convenient (positions close to the goal were no longer free). As a consequence, the velocity commands with the highest values became the ones leading to the positions computed by the global planner; this made the robot avoid the obstacle while still following the global plan. If the person had appeared along the global plan, and no way to approach it or reaching the goal had been found in the local costmap, the global planner would have re-planned a new path considering the person

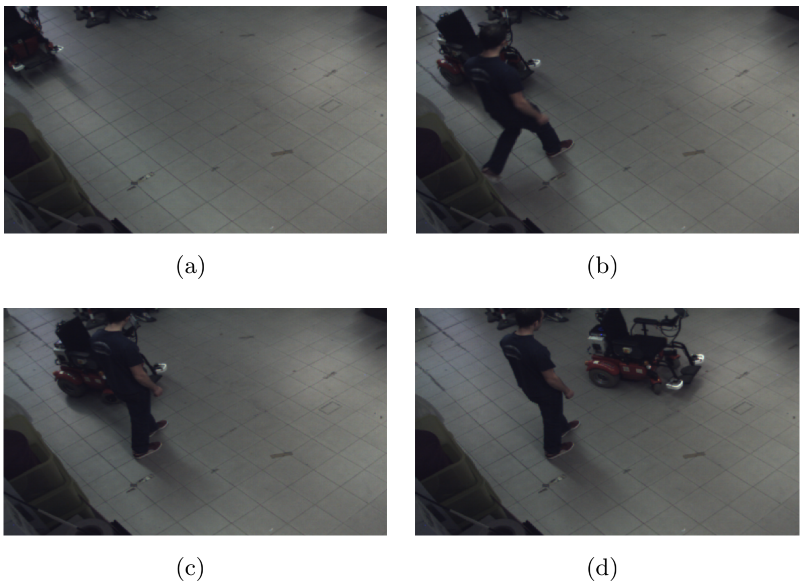


Figure 5.20: LURCH avoiding a person appearing on its way

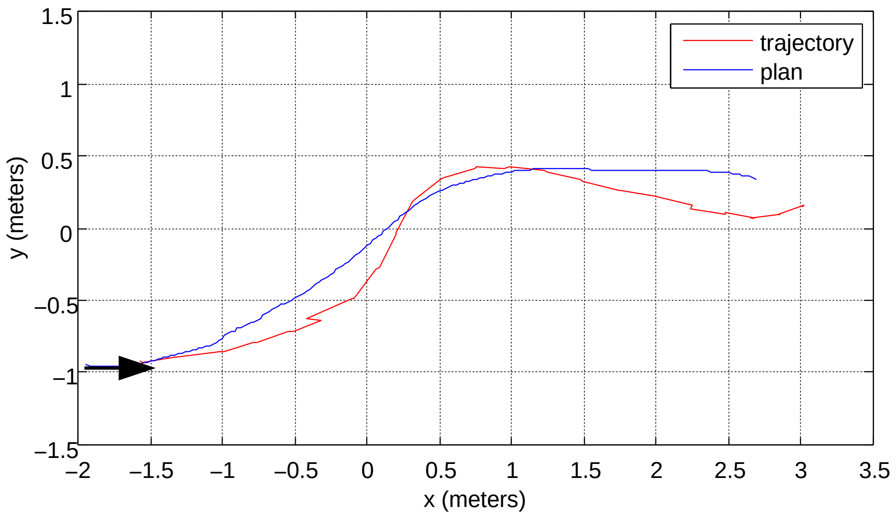


Figure 5.21: Dynamic obstacle avoidance: test

as a static obstacle. In general, the robot has proven to be very responsive against dynamic obstacles. In the following section we discuss the behaviour of LURCH in crowded areas with multiple obstacles moving unpredictably.

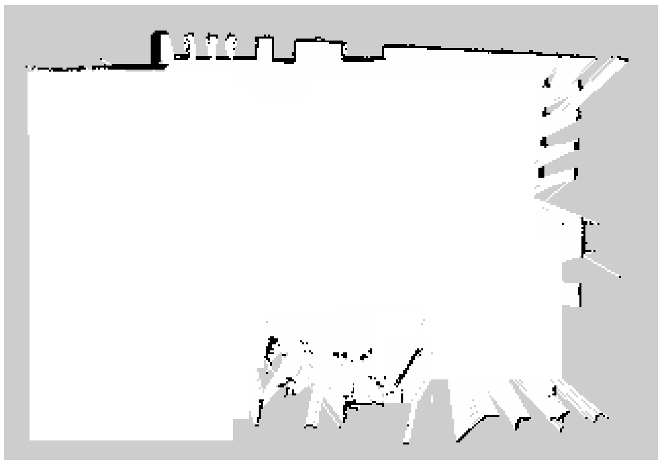


Figure 5.22: The map obtained with gmapping *for the* Open Day

5.3 Public demonstrations

The autonomous navigation features of the robot have been presented to the public at the *Open Day* of Politecnico di Milano (February 15th, 2014), an event that counted more than 15.000 visitors overall [1]. In that occasion we tested the robot behaviour in a crowded environment, planning paths and avoiding people. The map of the area had been previously captured via `gmapping`, and is shown in Figure 5.22.

LURCH proved to be sufficiently safe for people and objects around it, since it avoided collisions even in unpredictable situations, for instance when distracted people walked very close to it. The robot was able to plan paths and reach goals successfully even when surrounded by dozens of people. Dynamic obstacles were detected and collisions were avoided in time. As for the localization performance, since the area was large and with many obstacles interfering with the laser scanners, the robot encountered some difficulties in estimating its pose with AMCL, having to rely mostly on odometry data provided by encoders, but could correct its estimation when approaching walls. Figure 5.23 shows some situations in which LURCH, surrounded by people, moved autonomously at the event.

[1]Source: `milano.corriere.it`

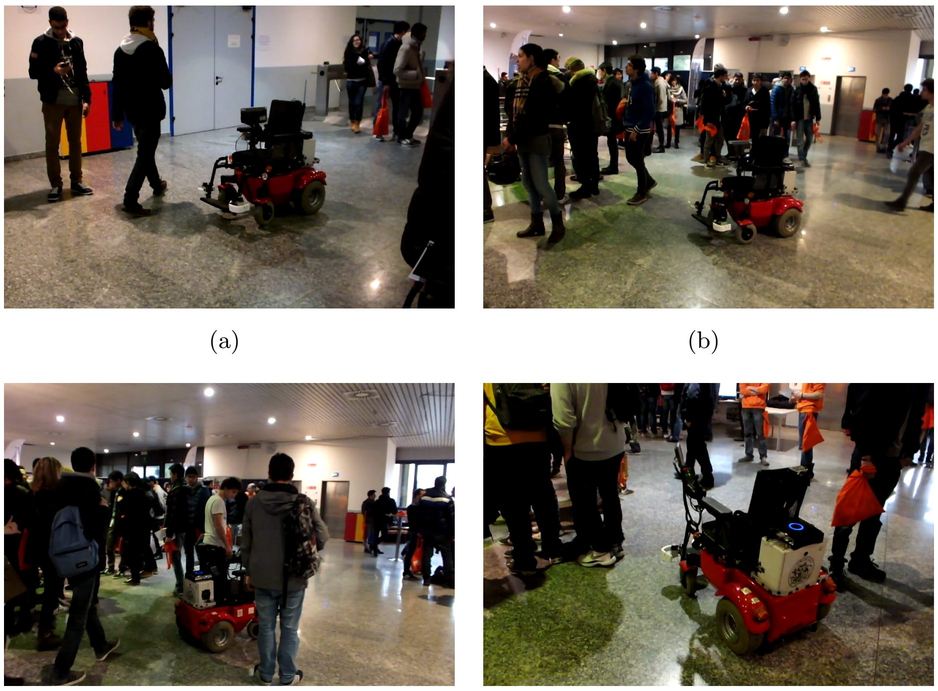

(a) (b) (c) (d)

Figure 5.23: LURCH moving autonomously during the Open Day

Chapter 6

Conclusions and future work

In this chapter we discuss strengths and weaknesses of our system, and we propose some possible enhancements, based on the results we have obtained in Chapter 5.

6.1 Conclusions

The result of this work has been a ROS package for a robotic wheelchair that includes an innovative multi-sensor fusion engine for localization and many features typically present in a mobile robot, among which path planning and motion planning. Thanks to the use of the `heartbeat` package, selecting operating modes at runtime is possible. Many parts of the package are configurable, from sensor parameters to PID gains.

With this new software system, LURCH has shown satisfying performances in both assisted and autonomous drive. Moreover, using ROAM-FREE for odometry measurements and AMCL to correct positioning errors has proven to be a good compromise when no absolute position sensor is present.

6.2 Future work

Despite the good results obtained, some features have to be improved. In particular, a more accurate kinematic model of the wheelchair should result in a better performance of both localization and control. That model could require encoders also on the front wheels, in order to take into account their position and orientation when computing the robot's twist. Since the resolution of the encoders is critical when calculating wheel velocities, also changing the rear encoders with more accurate sensors is a way to improve precision in speed and pose estimation.

Localization can also be improved with the addition of an absolute position sensor. In Chapter 3 we mention the presence of a camera on LURCH, which was previously used along with visual markers to detect the robot position in known environments. Such system could substitute AMCL as reference for localization, and some preliminary experiments have already been done in this direction.

A limit of the new software is that currently it cannot detect obstacles located behind the robot. For this reason, sonars, which are already mounted on the wheelchair, have to be added to the software architecture. Furthermore, the robot can only perceive the environment as a plane: the third dimension has not been taken into account. This can be considered a good approximation if the wheelchair is moving on flat floors, but it is no more sufficient if we want the robot to deal with slopes; moreover, collisions with obstacles that do not intersect the scanned plane cannot be prevented.

In this work we focused on the implementation of robotic features, without concentrating on aspects of human-robot interaction. For instance, we have not taken advantage of the on-board touch screen. A graphical user interface can be designed on the basis of user's disabilities. Other command devices previously experimented on LURCH, such as voice recognition and brain-computer interfaces, can also be re-introduced; indeed the architecture is built in a way that allows to interface many different types of devices.

Bibliography

[1] B. Pitzer, M. Styer, C. Bersch, C. DuHadway, and J. Becker, *Towards Perceptual Shared Autonomy for Robotic Mobile Manipulation*, IEEE International Conference on Robotics and Automation, 2011.

[2] S. Ceriani, *Sviluppo di una carrozzina autonoma d'ausilio ai disabili motori*, Master's thesis, Politecnico di Milano, 2008.

[3] M. Dalli, *Sviluppo di un sistema di controllo basato su odometria per una carrozzina robotica*, Master's thesis, Politecnico di Milano, 2008.

[4] S. Rönnbäck, *On Methods for Assisted Mobile Robots*, Doctoral Thesis, Luleå University of Technology, 2006.

[5] R.C. Simpson, *Smart wheelchairs: A literature review*, Journal of Rehabilitation Research and Development, 2005.

[6] D.L. Jaffe, *A Case Study: The Ultrasonic Head Controlled Wheelchair and Interface*, OnCenter, Issue No.2, March 1990.

[7] R. Madarasz, L. Heiny, R. Cromp, and N. Mazur, *The Design of an Autonomous Vehicle for the Disabled*, IEEE Journal of Robotics and Automation, 1986.

[8] J. Connell and P. Viola, *Cooperative control of a semi-autonomous mobile robot*, Proceedings of the IEEE International Conference on Robotics and Automation, 1990.

[9] A. Pruski and G. Bourhis, *The VAHM project: a cooperation between an autonomous mobile platform and a disabled person*, IEEE International Conference on Robotics and Automation, 1992.

[10] G. Bourhis, K. Moumen, P. Pino, S. Rohmer, and A. Pruski, *Assisted navigation for a powered wheelchair*, International conference on systems, man and cybernetics, 1993.

[11] O. Horn, A. Courcelle, *Interpretation of Ultrasonic Readings for Autonomous Robot Localization*, Journal of Intelligent and Robotic Systems, March 2004.

[12] S.P. Levine, D.A. Bell, L.A. Jaros, R.C. Simpson, Y. Koren, and J. Borenstein, *The NavChair Assistive Wheelchair Navigation System*, IEEE Transactions on Rehabilitation Engineering, 1999.

[13] D.P. Miller and M.G. Slack, *Design and testing of a low-cost robotic wheelchair prototype*, Autonomous Robots, 1995.

[14] G. Del Castilloa, S. Skaara, A. Cardenasb, and L. Fehr, *A sonar approach to obstacle detection for a vision-based autonomous wheelchair*, Robotics and Autonomous Systems, 2006.

[15] N.I. Katevas, N.M. Sgouros, S.G. Tzafestas, G. Papakonstantinou, P. Beattie, J.M. Bishop, P. Tsanakas, and D. Koutsouris, *The autonomous mobile robot SENARIO: a sensor aided intelligent navigation system for powered wheelchairs*, IEEE Robotics and Automation Magazine, 1997.

[16] U. Borgolte, H. Hoyer, C. Bühler, H. Heck, and R. Hoelper, *Architectural Concepts of a Semi-autonomous Wheelchair*, Journal of Intelligent and Robotic Systems, 1998.

[17] A. Lankenau, O. Meyer, and B. Krieg-Bruckner, *Safety in robotics: the Bremen Autonomous Wheelchair*, Advanced Motion Control, pages 524-529, 1998.

[18] A. Lankenau and T. Röfer, *A versatile and safe mobility assistant*, IEEE Robotics and Automation Magazine, 2001.

[19] A. Lankenau, T. Röfer, and B. Krieg-Brückner, *Self-Localization in Large-Scale Environments for the Bremen Autonomous Wheelchair*, in C. Freksa, W. Brauer, C. Habel, and K.F. Wender, editors, *Spatial Cognition III: Routes and Navigation, Human Memoryand Learning, Spatial Representation and Spatial Learning*, volume 2685 of *Lecture Notes in Computer Science*, pages 34-61, Springer, 2003.

[20] E. Prassler, J. Scholz, and P. Fiorini, *A robotics wheelchair for crowded public environment*, IEEE Robotics and Automation Magazine, 2001.

[21] Y. Kuno, N. Shimada, and Y. Shirai, *Look where you're going [robotic wheelchair]*, IEEE Robotics and Automation Magazine, 2003.

[22] R. Simpson, D. Poirot, and F. Baxter, *The Hephaestus Smart Wheelchair system*, IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2002.

[23] M. Mazo, *An integral system for assisted mobility*, IEEE Robotics and Automation Magazine, 2001.

[24] H. Seki, S. Kobayashi, Y. Kamiya, M. Hikizu, and H. Nomura, *Autonomous/semi-autonomous navigation system of a wheelchair by active ultrasonic beacons*, IEEE International Conference on Robotics and Automation, 2000.

[25] R. Simpson, E. Lopresti, S. Hayashi, I. Nourbakhsh, and D. Miller, *The smart wheelchair component system*, Journal of Rehabilitation Research and Development, 2004.

[26] C. Mandel, K. Huebner, and T. Vierhuff, *Towards an autonomous wheelchair: Cognitive aspects in service robotics*, Proceedings of Towards Autonomous Robotic Systems, 2005.

[27] B. Rebsamen, E. Burdet, Cuntai Guan, Chee Leong Teo, Qiang Zeng, M. Ang, and C. Laugier, *Controlling a wheelchair using a BCI with low information transfer rate*, pages 1003-1008, June 2007.

[28] F. Doshi and N. Roy, *Spoken language interaction with model uncertainty: an adaptive human-robot interaction system*, Connection Science, December 2008.

[29] I. Iturrate, M. Antelis, J. Minguez, and A. Kuebler, *Non-Invasive Brain-Actuated Wheelchair based on a P300 Neurophysiological Protocol and Automated Navigation*, IEEE Transactions on Robotics, 2009.

[30] X. Huo, J. Wang, and M. Ghovanloo, *Wireless control of powered wheelchairs with tongue motion using tongue drive assistive technology*, 30th Annual International IEEE EMBS Conference, 2008.

[31] S. Nakajima, *Concept of a Novel Four-wheel-type Mobile Robot for Rough Terrain, RT-Mover*, The 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 11-15, 2009.

[32] L. Montesano, M. Díaz, S. Bhaskar, and J. Minguez, *Towards an Intelligent Wheelchair System for Users With Cerebral Palsy*, IEEE Transactions on Neural Systems and Rehabilitation Engineering, 2009.

[33] *Intelligent Wheelchair Research Group*, accessed March 13, 2014, `http://userpage.fu-berlin.de/~latotzky/wheelchair/`

[34] T. How, R. H Wang, and A. Mihailidis, *Evaluation of an intelligent wheelchair system for older adults with cognitive impairments*, Journal of NeuroEngineering and Rehabilitation, 2013.

[35] R. Vandone, *Controllo di una carrozzina con biosensori a basso costo*, Thesis, Politecnico di Milano, 2009.

[36] P. Meriggi, *Processing and Communication Systems for Device Communities*, PhD thesis, Università degli Studi di Brescia, 2005.

[37] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, *ROS: an open-source Robot Operating System*, IEEE International Conference on Robotics and Automation, 2009.

[38] *ROS Documentation*, accessed March 6, 2014, `http://wiki.ros.org/`

[39] R.B. Rusu, *ROS - Robot Operating System, Tutorial Slides*, November 1, 2010.

[40] G. Grisetti, C. Stachniss, and W. Burgard, *Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling*, IEEE Transactions on Robotics, Volume 23, pages 34-46, 2007.

[41] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*, The MIT Press, 2005.

[42] D. Cucci, M. Matteucci, *A Flexible Framework for Mobile Robot Pose Estimation and Multi-Sensor Self-Calibration*, Informatics in Control, Automation and Robotics (ICINCO), 2013.

[43] L. Kneip, F. Tâche, G. Caprari, and R. Siegwart, *Characterization of the compact Hokuyo URG-04LX 2D laser range scanner*, IEEE International Conference on Robotics and Automation, 2009.

[44] *Zotac website*, accessed March 14, 2014, `http://www.zotac.com`.

[45] *Xenarc website*, accessed March 14, 2014, `http://www.xenarc.com`.

[46] *Logitech Gaming Gear*, accessed March 14, 2014 `http://gaming.logitech.com`.

[47] *C++ Reference*, 2000-2014, accessed March 19, 2014, `http://www.cplusplus.com`

[48] P. E. Gill and W. Murray, *Algorithms for the solution of the nonlinear least-squares problem*, SIAM Journal on Numerical Analysis, 1978.

[49] *Motion Capture Systems by OptiTrack*, 2014, accessed March 21, 2014, `http://www.naturalpoint.com/optitrack/`

[50] National Instruments, *PID Theory Explained*, 2011.

[51] M. Veronesi, *Regolazione PID. Tecniche di taratura, schemi di controllo, valutazione delle prestazioni*, Franco Angeli Editore, 2011.

[52] The Mathworks, Inc., *MATLAB Documentation Center*, 2014, accessed March 24, 2014, `http://www.mathworks.it/it/help/index.html`