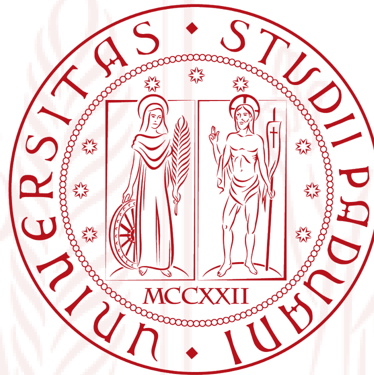
UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE



Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea

Potential field navigation for telepresence robots driven by BCI

Advisor: Prof. Emanuele Menegatti
Co-Advisors: Dott. Marco Carraro
Dott. Andrea Cimolato

Student: Fabio Brea

3 Ottobre 2016
Anno Accademico 2015/2016

Hard work beats talent
when talent doesn't work hard.

TIM NOTKE

Abstract

In this work, we present an open-source system that will help elderly and people with severe physical disabilities to use a telepresence robot, by means of brain signals. Driving a mobile device, such a robot, via Brain Computer Interface (BCI) can improve the quality of life by allowing patients to join relatives and friends, located in different rooms, in their daily activities. In order to lighten the burden of driving the robot, we use a shared control approach, so that the user needs only to concentrate on the final destination, while the robot takes care of obstacle detection and avoidance. In particular, we propose a semi-autonomous system where the autonomous navigation uses a potential field approach. The system can be seen as a Deterministic Finite Automaton where data from different sensors are merged together, in order to create a dynamic map used by the navigation.
We have developed this system using ROS (Robot Operating System), and tested it using a Hokuyo URG-04LX-UG01 and a second-generation Kinect on a TurtleBot, modified to be a telepresence robot. To prove the system portability, we successfully tested our work also with a Pioneer 3-AT which uses a Sick LMS-100 and a first-generation Kinect as sensors. In order to provide the best benefit to impaired patients, another contribution of this work is the implementation of a new interface able to stream video/audio data from the robot. The interface exploits the GStreamer framework. Finally, we tested the system with a user giving commands through the BCI.

Contents

Abstract **v**

1 Introduction **1**
1.1 State of the Art 2
1.2 Thesis structure 4

2 Materials and Methods **5**
2.1 Robot Operating System (ROS) 5
2.2 Point Cloud Library (PCL) 7
2.3 OpenCV 8
2.4 TurtleBot 9
2.4.1 iClebo Kobuki 9
2.4.2 IAS-Lab Turtlebot 10
2.5 Pioneer 3-AT 11
2.5.1 IAS-Lab Pioneer 12
2.6 Hokuyo URG-04LX-UG01 12
2.7 Sick LMS-100 14
2.8 Kinect 15
2.9 Kinect v2 17

3 Autonomous Navigation **21**
3.1 Potential field path planning 23
3.2 Dynamic Navigation 25

4 The telepresence apparatus **29**
4.1 GStreamer 29
4.1.1 Elements 30
4.1.2 Pads 31
4.1.3 Bins 31
4.1.4 Communication 32
4.2 Brain Computer Interface 34

4.2.1 The BCI loop 35
4.2.1.1 Subject - Acquisition Block 36
4.2.1.2 Processing - Machine Learning Block 38
4.2.1.3 Online Feedback - AT Devices Block 38

5 Implementation Details **41**
5.1 Navigation with Hokuyo laser scan 41
5.2 Parameters tuning and map visualization 44
5.3 Navigation with Kinect v2 52
5.4 Occupancy Grids merging 59
5.5 Navigation with Bumpers 65
5.6 Semi-Autonomous Navigation 67
5.7 Deterministic Finite Automaton 69
5.8 The Telepresence Apparatus 75
5.9 Semi-Autonomous Navigation BCI Driven 77
5.9.1 Experimental design 77
5.9.2 Experimental protocol 77
5.9.3 Data Analysis 80
5.9.4 Decision Making Algorithm 80

6 Conclusions **81**
6.1 Future Works 83

Bibliography **85**

Acknowledgements **91**

Chapter 1

Introduction

The quality of life of a person is negatively impacted when he or she cannot participate in everyday activities with family and friends, which is often the case for people with special needs, e.g. elderly or people with severe disabilities, who are full time residents at medical and healthcare facilities or are constrained at bed. Isolation can lead to feelings of overall sadness which bring to additional health issues [1]. Telepresence robots may be a benefit as they provide a way to make these people able to engage themselves again in social activities. The term telepresence refers to those technologies that allow a person to experience being in a location (or even have an effect on it) without being physically present there. This technology has also a great impact on the lives of the other family members: it allows them to keep an eye on the person with special needs without having to stay in the same room, thus also improving their quality of life. Examples of recent projects that have developed such a kind of systems are: Giraff [2], Double 2 [3] and Amy A1 [4] (see Figure 1.1).

The problem is that these systems are not easily controllable by those people with special needs. Teleoperating a robot requires a skill that it is not simple to obtain by those people, as they have reduced, if any, mobility. In particular, using a controller to drive a mobile device could be out of their possibilities.

The final goal of this work is to present an open-source system that will help elderly and people with severe physical disabilities to use a telepresence robot, without the need of using their hands. The tool that we will use in order to achieve this result will be a Brain-Computer Interface (BCI). BCI-driven robots can be guided through brain signals, so that users with an active mind can use them without needing to move. However, controlling such an application through an uncertain channel as a BCI can be complicated and exhaustive, because the commands that can be delivered are limited. A shared control approach can help in this scenario. The cooperation between the user and the robot allows the former to only focus the attention on his/her final destination, while the latter will deal with low level

(a) Giraff (b) Double 2 (c) Amy A1

Figure 1.1: Examples of telepresence robots.
(Sources: http://www.aal-europe.eu/projects/excite/,
http://www.doublerobotics.com/, http://telepresencerobots.com/)

navigation problems, such as obstacle detection and avoidance.
In order to create such a system, we will start by implementing a potential field based navigation. We will integrate data from different sensors, keeping the system as modular as possible, so that we could add and remove sensors if the need arises. Then, we will have to create the shared control system, thus we will allow the user to give simple commands, from a remote laptop, to make the robot rotate in the desired direction. By using this semi-autonomous approach we will give to the BCI user a feeling of being able to fully control the robot. After adding a bidirectional audio/video connection, the user will be finally able to join and interact with relatives and friends that are in different rooms, thus improving their quality of life and, somehow, their independence.

1.1 State of the Art

In the last years, much work has been done in developing autonomous navigation by mobile robots. There are three general way to approach the problem of mobile navigation: teleoperated, autonomous or semi-autonomous.
The teleoperated approach relies on the human operator's skill to fully control a mobile robot, so these system are relatively inexpensive, with respect to the sensors needed. These system are often used with telepresence robot. For example in [5] the authors developed a teleoperated telepresence system, designed specially for elderly care. In this work they do not address the problem of the senior ability

to effectively teleoperate the robot. This is usually a major problem, also considering that they must be able to react in time to avoid possible obstacles that are in the robot path.
Instead, the autonomous approach focuses on making the robot navigate through the environment with very little, if any, human control. Such systems needs multiple sensors and advanced algorithms, which can be economically expensive. A fundamental component in this type of navigation is the localization of the robot in an environment, as well as building a map of it. The problem of placing a robot in an unknown environment and making it incrementally build a consisted map of this environmente while simultaneously determining its location within this map is a famous problem called Simultaneous Localization and Mapping (SLAM) [6]. In [7] the authors propose the Adaptive Monte Carlo Localization (AMCL) approach, which uses a particle filter to track the pose of a robot in a known map, as a mean to localize the robot. On the other hand, Gmapping [8] proposes to solve the SLAM problem by using Rao-Blackwellized particle filters [9] on a grid to learn the map by taking into account the movement of the robot and the most recent observation, thus creating a map of the environment. Gmapping and AMCL can be seen as the starting point for many other works. Among the many, in [10] the authors developed a system for elder care, making the robot navigate autonomously with both static and dynamic obstacle avoidance. This system is particularly interesting because of the marked similarities between it and the one presented in this work. Among the tasks their robot can accomplish, auto-docking and the automatic coverage of known maps are notable.
When it comes to autonomous navigation, different techniques can be applied. For example, in [11] the authors developed a navigation system using three different algorithms: genetic algorithm, artificial neural network and A*. Another technique used in autonomous navigation is based on the concept of potential field, that is also the method used in this work. In [12] the authors make the robot navigate through the environment using the potential field method and the ant-colony paradigm to optimize the results. All the methods presented so far required a known map of the environment. As we have seen, however, such detailed information is rarely available and often hard to retrieve. Instead, the algorithm in our work relies on a dynamic map, thus the computational effort of the planner is lower and we do not have to localize the robot, because we can rely on the user will.
This is the principle of the semi-autonomous approach: the robot is able to navigate on its own, but the user can monitor and command the robot as needed. The work on which our project is based rely on a semi-autonomous approach. In [13] the authors created a shared control system that could be driven by the BCI. In our work we extend this solution by adding the possibility to use data from multiple sensors, specially from the Kinect v2, the Deterministic Finite Automaton

approach, that allows to obtain different behaviours of the robot to the need, and the video/audio streaming through GStreamer.

1.2 Thesis structure

The remainder of the thesis is organized as follows:

- In **Chapter 2**, we will describe methods and libraries used in this work, as well as the framework on which we based our system: ROS. Then we will give information about the robots used to test our algorithms and about the sensors that are mounted on them.
- **Chapter 3** describes the basis of autonomous navigation. It will focus on potential field based navigation, particularly on dynamic navigation, that is the method that we will use to make the robot move and avoid obstacles.
- In **Chapter 4**, we will describe the methodologies applied to create the telepresence apparatus. We will discuss about GStreamer as the framework to link the robot vision to the laptop of the end-user, and about the BCI as the tool to help the robot to reach a goal, giving the user intentions.
- In **Chapter 5**, we will describe the core part of our work, pointing out the issues occurred and how we overcome them.
- Finally, in **Chapter 6**, we will report our conclusions and possible future works.

Chapter 2

Materials and Methods

In this chapter we will describe materials and libraries used in this work. The algorithms are written in *C++* and tested under Ubuntu 14.04 Trusty Tahr.

2.1 Robot Operating System (ROS)

Figure 2.1: ROS Logo. Source: http://ros.org/

ROS (Robot Operating System)[14] is a framework for robot software development, providing libraries and tools to help software developers create robot applications. ROS is considered a meta-operating system, because it provides the services that are expected from an operating system: hardware abstraction, low-level device control, message passing between processes and package management. At the same time it can be considered a middleware, because it provides methods to achieve inter-process communication.
ROS was designed to be as distributed and modular as possible, so that developers could reuse code already written with minimum effort and could devote themselves to solve new problems. One of the goals of ROS is, in fact, to guarantee a software platform compatible with most robots, no matter how diverse. Thanks to its distributed nature a wide collection of algorithm ready to use is available to the

end-user. To understand ROS modularity, think of a system controlling a movable robot: one process (called node) controls a sensor, another node uses sensor data to create a map, a different node calculate the shortest path, another node moves the robot, and so on.
ROS is based on a graph architecture. From the computational point of view this graph is the peer-to-peer network of ROS processes that are processing data together. The main computational graph concepts in ROS are:

- **Nodes:** A node in ROS is a running process that performs computation. Nodes are the main entity in the graph. They can communicate with each other using streaming **topics**, RPC **services** or the **Parameter Server**. A node is written with the use of a ROS client library, such as *roscpp* or *rospy*.

- **Messages:** A message is a simple data structure, comprising typed fields. Standard primitive types are supported, but it's possible to define own messages. Nodes communicate with each other by publishing messages to topics.

- **Topics:** A topic is a named bus over which nodes exchange messages. Topics are part of publishers/subscribers pattern. A node that is interested in certain data will subscribe to the appropriate topic, while, a node that generates data will publish to an appropriate topic. There can be multiple concurrent publishers and subscribers to a topic. This permits to decouple the production of information from its consumption.

- **Services:** A service is an alternative way of communication that does not use topics and allows to create RPC request/reply interactions. Services are defined by a pair of message structures: one for the request and one for the reply.

- **Master:** The ROS Master provides name registration and look-up to the rest of the computation graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.

- **Parameter Server:** The Parameter Server runs inside of the ROS Master and is used to store and retrieve parameters at run-time.

- **Bags:** A bag is a file format for storing ROS message data. ROS permits, through bags, to record one or more topics and play them back afterward.

An example of the ROS graph is visible in Figure 2.2.
The ROS distribution used in this work is *ROS Indigo Igloo*.

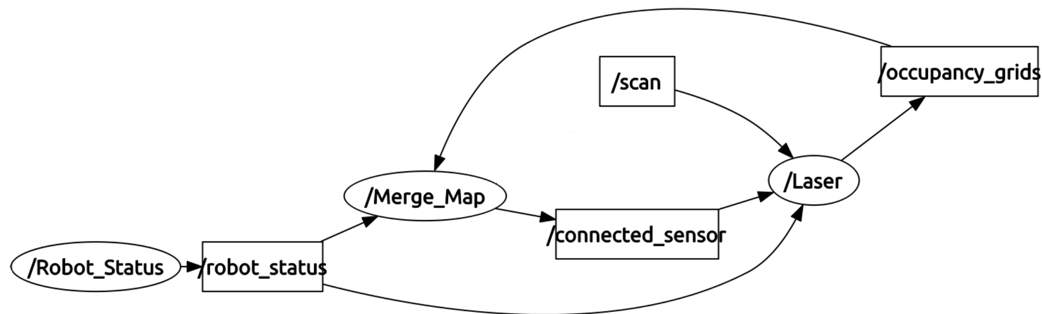


Figure 2.2: ROS graph: nodes are represented as ellipses, topics are represented as rectangles. An outgoing edge means that the node is a publisher for the topic, while an incoming edge means that the node is subscribed to the topic.

2.2 Point Cloud Library (PCL)

Figure 2.3: PCL logo. Source: http://pointclouds.org/

With the advent of low cost 3D sensors such as the Kinect sensor, 3D perception has gained more importance in robotics, as well as other fields. The upcoming need to handle 3D data efficiently has been satisfied by the Point Cloud Library.
The Point Cloud Library (PCL)[15] is a standalone, large scale, open project that efficiently processes 2D/3D image and point cloud. Point clouds are a way to intuitively represent and manipulate the information provided by 3D sensors, such as time-of-flight cameras and laser scan, in which the space is sampled in a finite set of points in a 3D frame of reference. PCL provides a number of data structures to easily represent the points of the sampled space. PCL also provides numerous state of the art algorithms for filtering, feature estimation, surface reconstruction, point cloud registration, model fitting and segmentation. Another reason to use PCL is that it is well integrated in ROS. Indeed, the PCL interface for ROS provides the means required to communicate PCL data structures through the message-based communication system created by ROS. In order to do so, a set of conversion functions are also provided to convert from native PCL data types to ROS messages.

PCL has been used to manipulate point clouds from the Kinect v2 sensor. In detail, PCL has been used for:

- Acquiring point clouds.
- Filtering point cloud by distance through ConditionAnd and PassThrough.
- Downsampling through VoxelGrid.
- Removing noise through Statistical Outlier Removal and Radius Outlier Removal.
- Transforming a point cloud into a Laser Scan.
- Performing the ICP algorithm.

2.3 OpenCV

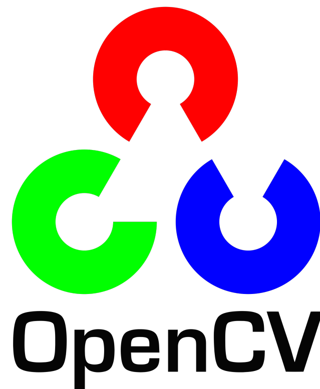


Figure 2.4: OpenCV logo. Source: http://opencv.org/

OpenCV (Open Source Computer Vision)[16] is a computer vision library designed for computational efficiency and with a strong focus on real-time applications. One of OpenCV's goals is to provide a simple to use computer vision infrastructure that helps people build fairly sophisticated vision applications quickly. OpenCV provides many algorithms that span many areas in vision, including factory product inspection, medical imaging, security, user interface, camera calibration, stereo vision and robotics.
As for PCL, OpenCV is completely integrated into ROS, which also provides image type conversions between OpenCV and ROS formats.
In this work we will use OpenCV to:

- Load and save images;
- Visualize merged laser scans from different sensors;
- Find the obstacles that have the greatest impact on navigation.

2.4 TurtleBot

TurtleBot[17] is a low-cost mobile robot platform useful to get familiar with ROS and robots in general. It is an open source platform, so many applications are already available. The default kit is composed of a Kobuki robot as base, a laptop with ROS and a Xbox Kinect as sensor. With this simple setup the TurtleBot is able to handle vision, localization, communication and mobility. It can also be modified for other purposes, in fact the TurtleBot used in this work has been modified to be a telepresence robot (more about the modification in subsubsection 2.4.2).

2.4.1 iClebo Kobuki

TurtleBot's base is an iClebo Kobuki base (see Figure 2.5). It is a mobile research base with sensors, motors and power sources and is used exclusively indoor. This robot is specially designed for education and research on state of the art robotics, thanks to its customizable structure. Kobuki has highly reliable odometry and long battery duration. Its battery can also provide power to a laptop as well as to additional sensors and actuators.
The specifications of the Kobuki are:

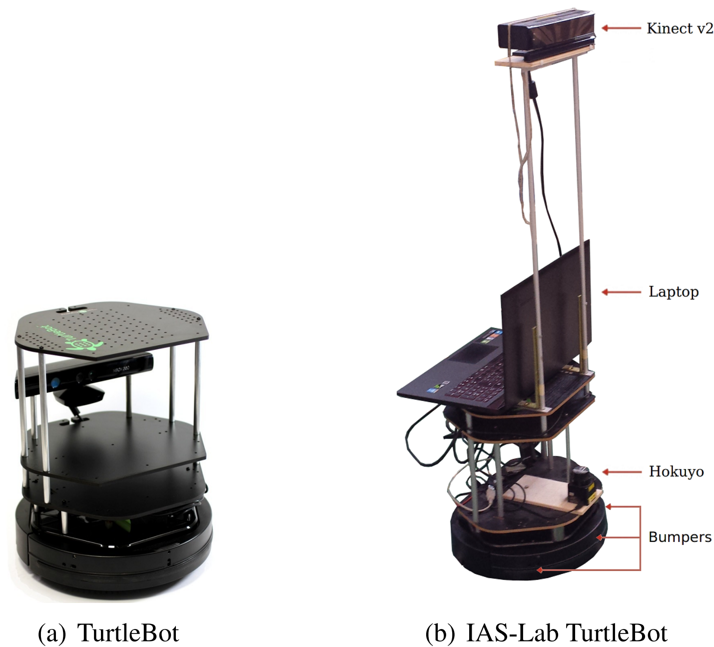
- It is round-shaped with a radius of 354mm.
- Its maximum payload is 5Kg.
- Its maximum speed is 0.7m/s, with maximum turning speed of 180$\deg$/s.
- The odometry is built in 3-axis gyrometer and has a high-resolution wheel encoder.
- The basic battery allows the robot to operate for 3 hours (7 hours with big-sized battery).
- Bumpers sensors (left, right and center).

Figure 2.5: iClebo Kobuki. Source: http://dabit.industries/products/iclebo-kobuki

2.4.2 IAS-Lab Turtlebot

The TurtleBot used in this work has been modified to be a telepresence robot (see Figure 2.6(b)). In detail, these are the modification that have been made:

- The Kinect has been removed.
- The middle platform has been raised.
- An Hokuyo scanning range finder URG-04LX-UG01 has been mounted on the bottom platform of the TurtleBot, at an height of 15 cm from the floor.
- Two rods have been mounted on the top of the TurtleBot with a little shelf on the extremity, so that a screen can be placed in front of the TurtleBot, to make it useful as a telepresence apparatus.
- A Kinect v2 has been placed on the little shelf, at an height 112.2 cm from the floor.



(a) TurtleBot (b) IAS-Lab TurtleBot

Figure 2.6: TurtleBot comparison. Source: http://sites.hevra.haifa.ac.il/

2.5 Pioneer 3-AT

The Pioneer 3-AT[18] is a highly versatile four-wheel drive robotic platform. It is a small four-motor skid-steer robot ideal for all-terrain operations or laboratory experimentation. The default kit is composed of one battery, an emergency stop switch, wheel encoders and a micro-controller with ARCOS firmware. It also comes with the Pioneer SDK advanced mobile robotics software development package. The core software mainly consists of the Advanced Robot Interface for Applications (ARIA). ARIA is a C++ library that provides a framework for controlling and receiving data from all MobileRobots platforms, as well as from most accessories. It includes open source infrastructures and utilities useful for writing robot control software, support for network sockets, and an extensible framework for client-server network programming.
The Pioneer 3-AT can be used in ROS through the package RosAria, that creates a bridge between the two.
The specifications of the Pioneer 3-AT are:

- it is 508mm long, 497mm wide and 277mm high.
- its maximum payload is 12Kg on tile/floor, 10Kg on grass/dirt and 5Kg on asphalt.

Figure 2.7: Pioneer 3-AT. Source: http://www.cyberbotics.com/

- its maximum speed is 0.7m/s, with maximum turning speed of 140deg/s.
- it supports up to three batteries that allows the robot to operate for 2/3 hours.
- it has 6 sonar sensors facing outward placed at 20deg intervals plus one on each side.

2.5.1 IAS-Lab Pioneer

The Pioneer 3-AT used in this work has been modified to be a telepresence robot (see Figure 2.8). In detail, these are the modification that have been made:

- A metal apparatus has been mounted on the top of the robot. This apparatus provides a shelf on which a laptop can be placed and two rods that can be used to attach a screen to the Pioneer 3-AT, in order to make it useful as a telepresence apparatus. Between the rods there are three horizontal bars that keep the rods together and can be used to mount other accessories.
- A Sick LMS-100 laser scanner has been mounted in front of the robot.
- Two Kinects have been placed on the metal bars on the top of the robot.

The Pioneer 3-AT will be used in order to prove the portability of the algorithms created.

2.6 Hokuyo URG-04LX-UG01

Hokuyo URG-04LX-UG01[19] is a laser sensor for area scanning. The light source of the sensor is infrared laser of wavelength 785nm with laser class 1 safety.

Figure 2.8: IAS-Lab Pioneer 3-AT.

The Hokuyo laser scanner uses a rotating mirror to sweep the low-power infrared laser beam in a circle. Scan area is 240°, with pitch angle of 0.36°, and the sensor outputs the distance of 683 measurements. Scans are performed at a frequency of 10Hz. The detection distance is in the range of 20-5600mm, with accuracy of ±30mm within 1000mm and ±3% of measurement otherwise.
Because using time of flight information to calculate distances would require expensive hardware capable of gigahertz level timing, the laser is subject to amplitude modulation and the resulting phase difference of the reflected beam is used to calculate the distance. In this way it is possible to obtain stable measurements with minimum influence from objects color and reflectance, as well as a significant decrease of the cost of the device.
The scanner is not typically suitable to outdoor work due to infrared interference from other light sources.
Hokuyo laser scanner is very well integrated in the ROS environment. The package used in this work, that allows to use the laser scan, is urg_node.
These are the specification for the node:

- **Published topic:**

   - **/scan:** sensor_msgs/LaserScan
- **Subscribed topic:**
   - **none**

Figure 2.9: Hokuyo URG-04LX-UG01. Source: http://www.hokuyo-aut.jp

2.7 Sick LMS-100

The Sick LMS-100[20] is a laser sensor that can be used for area monitoring, object measurement, object detection and to determine positions. It operates by measuring the time of flight of laser (class 1 safety) light pulses: a pulsed laser beam is emitted and it is then reflected as it meets an object. The distance is calculated based on the time between the transmission and the reception of the impulse. The pulsed laser beam is deflected by an internal rotating mirror so that a fan-shaped scan is made of the surrounding environment.
The angular resolution can be adjusted as needed. With an angular resolution of 0.25° the maximum scanning angle is 270° and the sensor outputs 1081 measurements, while with an angular resolution of 0.50°, the maximum scanning angle is also 270°, but the sensor outputs 541 measurements. Scans are performed at a frequency of 25/50Hz. The maximum detection distance is 20m, but with objects with low reflectivity, like cardboard and matt black, it decreases to 18m. The accuracy of the measurements is ±30mm.

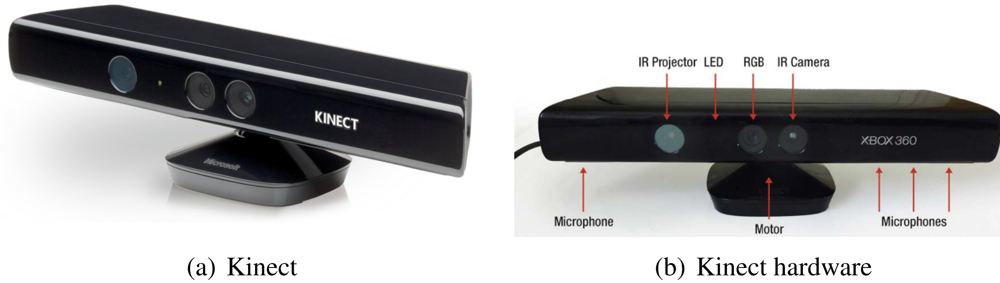
Figure 2.10: Sick LMS-100. Source: http://wll.kr/

The sensor is best suited for indoor use as it can be dazzled by sunlight, causing it to give erroneous readings. It can be connected to the laptop via an ethernet cable. Due to the use of time of flight technology, which requires expensive hardware, the Sick laser scanner has a higher price than the Hokuyo laser scanner.
The Sick lasers are implemented in ROS through the package lms1xx. These are the specification for the node:

- **Published topic:**
  - **/scan:** sensor_msgs/LaserScan
- **Subscribed topic:**
  - **none**

2.8 Kinect

The Microsoft Kinect[21] consists of a depth sensor, an RGB camera and a multi-array microphone. It is also equipped with a motorized pivot that allows to tilt the sensor vertically in the range of -27° - +27°.
The Kinect can extract depth information by a triangulation process called structured light. The depth sensor uses an infrared laser projector, which creates a grid of points by diffraction, and combines it with a monochrome CMOS sensor, which in turn interprets the infrared light captured. The Kinect then compares the original pattern and its deformed reflected image and obtains a disparity map on the basis of variations between both patterns.



(a) Kinect (b) Kinect hardware

Figure 2.11: Kinect. Sources: http://www.engadget.com/ , http://web.uvic.ca/

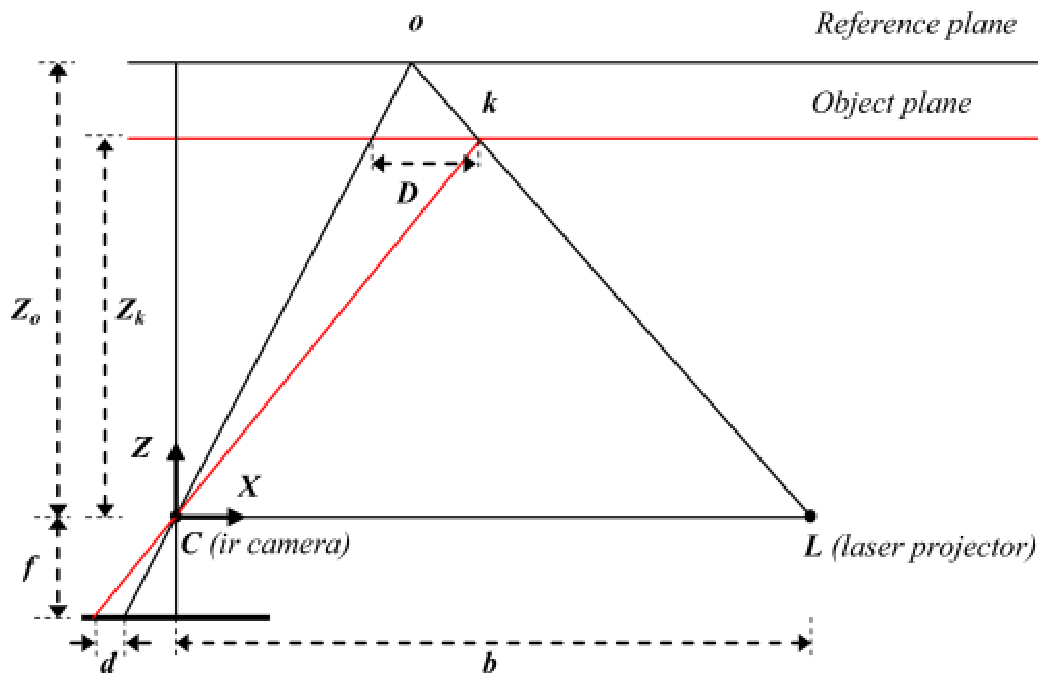


Figure 2.12: Depth calculation. Source: http://www.mdpi.com/

Figure 2.12 illustrates the relation between the distance of an object point k and the measured disparity d. The Z axis is orthogonal to the image plane towards the object, the X axis is perpendicular to the Z axis in the direction of the baseline b between the infrared camera center and the laser projector, and the Y axis is orthogonal to X and Z making a right handed coordinate system. Assume that an object is at a distance Z_0 from the sensor and the disparity measured is d. Using triangles similarity we have:

$$\frac{D}{b} = \frac{Z_0 - Z_k}{Z_0} \tag{2.8.1}$$

and:

$$\frac{d}{f} = \frac{D}{Z_k} \tag{2.8.2}$$

where Z_k denotes the depth of the point k, b is the base length, f is the focal length of the infrared camera, D is the displacement of the point k, and d is the observed disparity. Substituting D from Equation 2.8.2 into Equation 2.8.1 yields:

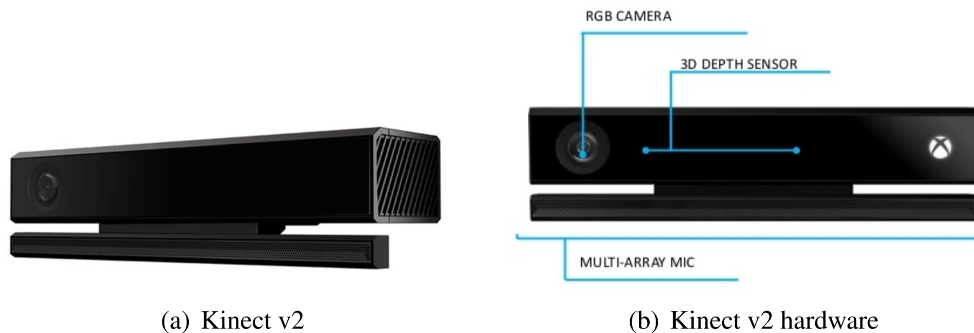
$$Z_k = \frac{Z_0}{1 + \frac{Z_0}{fb}d} \tag{2.8.3}$$

The resulting depth image has a resolution of 640x480 pixels. The depth sensor deliver images at 30Hz, has a field of view of 57° in the horizontal and 43° in the vertical direction in a range between 0.4m and 4.0m. The acquisition of depth data can be corrupted in presence of infrared light that can disturb the view of the projected pattern, so the sensor can not be used outside. The depth sensor also has problems with black objects, due to light absorption.
The Kinect sensor has been implemented in ROS, in the package *openni*, that creates a bridge between the driver for Kinect(*libfreenect*) and ROS.
These are the specification of the topics, that openni publishes or subscribes, used in this work:

- **Published topic:**
  - **/camera/depth/points:** sensor_msgs/PointCloud2
- **Subscribed topic:**
  - **none**

2.9 Kinect v2

The Microsoft Kinect v2 (or Kinect One) [22] is equipped with an RGB camera and a multi-array microphone. The tilting motor has been removed in this version. The Kinect v2 has three infrared light sources each generating a modulated wave with different amplitude. In order to capture reflected waves, Kinect v2 also has an infrared camera.
The infrared sensor in the Kinect v2 is a state of the art 512x424 CMOS array of differential pixels. The differential pixel distinguishes the time-of-flight sensor from a classic camera sensor. Each pixel has two photo diodes (A, B) that are controlled by the same clock signal that controls wave modulation. This clock signal drives the diodes such that if A is turned on, B is turned off, and vice versa.



(a) Kinect v2 (b) Kinect v2 hardware

Figure 2.13: Kinect v2. Sources: http://123kinect.com/ , http://ignatiuz.com/

The photo diodes then convert captured light into current which can be measured. The $(A - B)$ differential signal provides a pixel output whose value depends on both the returning light level and the time it arrives with respect to the pixel clock. The pixel output leads to a useful set of output images:

- $(A + B)$ gives a regular gray-scale image illuminated by ambient lighting (ambient image).
- $(A - B)$ gives phase information after an arctangent calculation (depth image).
- $\sqrt{(\sum(A - B)^2)}$ gives a gray-scale image that is independent of room lighting (active image).

Chip optical and electrical parameters determine the quality of the resulting image. Multiphase captures cancel linearity errors and simple temperature compensation ensures that accuracy is within specifications. Reasons to prefer time-of-flight system include the following:

- One depth sample per pixel: $X - Y$ resolution is determined by chip dimensions.
- Depth resolution is a function of transmitted light power, receiver sensitivity, modulation contrast and lens f-number
- Higher frequency: the phase to distance ratio scales directly with the modulation frequency resulting in finer resolution.
- Complexity is in the circuit design. The overall system, particularly the mechanical aspects, is simplified.

The operation principle, in a time-of-light system, for measuring the distance is based on measuring the time it takes for light wave to travel from emitter to object and back to sensor. With a single light pulse, letting d be the distance, we have:

$$d = \frac{t_r - t_e}{2} \cdot c, \tag{2.9.1}$$

where t_e and t_r represent respectively time for light pulse emitting and receiving and c is speed of light in air. Although simple, measuring distance through Equation 2.9.1 is not very practical for scene capturing devices like Kinect. Hence, the distance is calculated using the phase difference of the emitted light wave and the detected light wave reflected from the object. Let the transmitted wave

$$T(t) = \sin(wt) \tag{2.9.2}$$

have modulation frequency $w = 2\pi f$. The distance travelled by the modulated wave is $2d$ which produces phase shift ϕ. The received wave

$$R(t) = \beta \sin(wt - \phi) \tag{2.9.3}$$

has an amplitude factor β, but is not needed for measuring distance. Phase shift depends on time difference:

$$\phi = w(t_r - t_e) \tag{2.9.4}$$

Substituting Equation 2.9.4 in Equation 2.9.1 yields:

$$d = \frac{\phi c}{2w} \tag{2.9.5}$$

The phase shift ϕ can be estimated using different phase-delayed versions of the reference signal and processing the results using a low-pass filter. Kinect sensor uses three different phase-shifts of 0°, 120° and 240°.
Since measuring the distance is based on phase shift of the modulated wave, the maximum distance depends on the wavelength of the modulated wave. Knowing that phase wraps around 2π, the longer the wavelengths, the longer the maximum measured distance, but shorter wavelengths gives better resolution. In order to enable good resolution and also measuring longer distances, Kinect v2 uses three different frequencies of 120 MHz, 60 MHz and 16MHz. This allows the Kinect v2 to sense depth at 8m, but for more reliability it is advised to stay in the range $0.5 - 4.5$m.
The Kinect v2 delivers images at a frame rate of 30Hz, with a resolution of 1920x1080, but lower quality images are also available. Its field of view is 70° in the horizontal and 60° in the vertical direction. The Kinect v2 also requires an

USB 3.0 to work, due to the big quantity of data to transmit to the computer. The sensor can also be used outdoors in overcast and direct sunlight situations, with the opportune parameters settings. Light absorption problems with black objects are not solved yet.
The Kinect v2 sensor has also been implemented in ROS, in the package iai_kinect2, that creates a bridge between the driver for Kinect v2 (libfreenect2) and ROS.
These are the specification of the topics, that iai_kinect2 publishes or subscribes, used in this work:

- **Published topic:**
  - **/kinect2_head/depth/image:** sensor_msgs/Image
  - **/kinect2_head/depth_ir/points:** sensor_msgs/PointCloud2
- **Subscribed topic:**
  - **none**

Chapter 3

Autonomous Navigation

Navigation[23] is of great importance for any mobile system, since most robotic tasks require the travelling between different positions avoiding collisions. Given partial knowledge about the environment and a goal position, navigation encompasses the ability of a robot to act based on its knowledge and sensor values so as to reach its goal as efficiently and as reliably as possible.
There are two key competences required for mobile navigation: *path planning* and *obstacle avoidance*.
Given a map and a goal location, the goal of path planning is to find a trajectory that will cause the robot to reach the goal location when executed. In order to find this trajectory, as well as to navigate safely and efficiently, the robot must also be able to localize itself.
Path planning is formally done in a representation called **configuration space**. Suppose that a robot arm has k degrees of freedom. Every state or configuration of the robot can be described with k real values: $q_1, ..., q_k$. The k-values can be regarded as points in a k-dimensional space called the configuration space C of the robot. This description is useful because it allows us to describe a complex 3D shape of a robot with a single k-dimensional point, but this also means that we must inflate each obstacle by the size of the robot's radius to compensate (assuming that the robot is holonomic).
The problem to find a path in the physical space from an initial position to a goal position, avoiding all collisions with obstacles, is a difficult one, particularly as k grows large, but in the configuration space is straightforward. In fact, defining the **configuration space obstacle** O as the subspace of C where the robot bumps into something, we can compute the free space F = C - O, where the robot can move safely. The result of these simplifications is that the configuration space looks as a 2D version of the physical space.
The second competence required for mobile navigation is equally important, but occupies the opposite, tactical extreme. Indeed, given real-time sensor readings,

the goal of obstacle avoidance is to modulate the trajectory of the robot in order to avoid collisions.
Although the two competences seem very different from each other, they are strictly connected. The robot, during execution, must react to unforeseen event, like obstacles, in such a way that it will still be able to reach the goal. Without avoiding the obstacle, the planning effort will not pay off, because the robot will never physically reach its goal. On the other hand, without planning, the reacting effort cannot guide the overall robot behavior to reach a faraway target and then the robot will never reach its goal.
Regarding path planning, several methods for navigating in a known environment have been proposed. These methods change depending on how the operating area is defined. In fact, the robot's environment representation can range from a continuous geometric description to a decomposition-based geometric map or even a topological map. Path planners differ in how they use the discretized map, derived from a transformation from the continuous environmental model. There are two general strategies:

- **Graph-based:** the operating area is described as a connectivity graph in free space. The graph construction process is often performed offline using different methos, e.g, *Visibility graph*[24], *Voronoi diagram*[25], *Exact/ Approximate cell decomposition* [26, 27]. After the creation, the best path has to be found. This is performed by algorithms like *best-first search* (BFS)[28], *Dijkstra's algorithm*[29], *A* / D**[30, 31], *Rapidly Exploring Random Trees* (RRTs)[32].

- **Potential field:** a mathematical function is imposed directly on the free space[33]. The robot can be seen as a ball rolling downhill. The goal is to create a field with a gradient which is attractive toward the desired position (global minimum) and repellent from the obstacles (local maxima) in the area[34]. This strategy will be used to create a planner in this work.

Concerning obstacle avoidance, the approach changes depending on the existence of a global map and on the robot's precise knowledge of its location relative to the map.
For example, the *Bug algorithm* uses the robot's sensors to avoid obstacles, following the contour of each obstacle in the path between its current position and its goal.
Other methods scan the environment and compare it to the expected result of the scan. If a set of measured points deviates too much from the expected position, they are added to a list of potential obstacles and different counter-measures are taken to avoid them.

Finally, the method which inspired the algorithm used in this work creates a Virtual Force Field (VFF), using the idea that obstacles / targets exert repulsive / attractive forces on the robot, in order to avoid obstacles and reach the targets.

3.1 Potential field path planning

Navigation based on potential field is a common reactive approach for path planning. The basic idea is to guide the robot through the environment by defining the goal as attractive forces, and the obstacles as repulsive ones. In Figure 3.1 we can see that an analogy can be made to animals trying to reach their goal, while avoiding obstacles.

(a) The fish act as if they are repelled by a uniform force emanating from the shark.

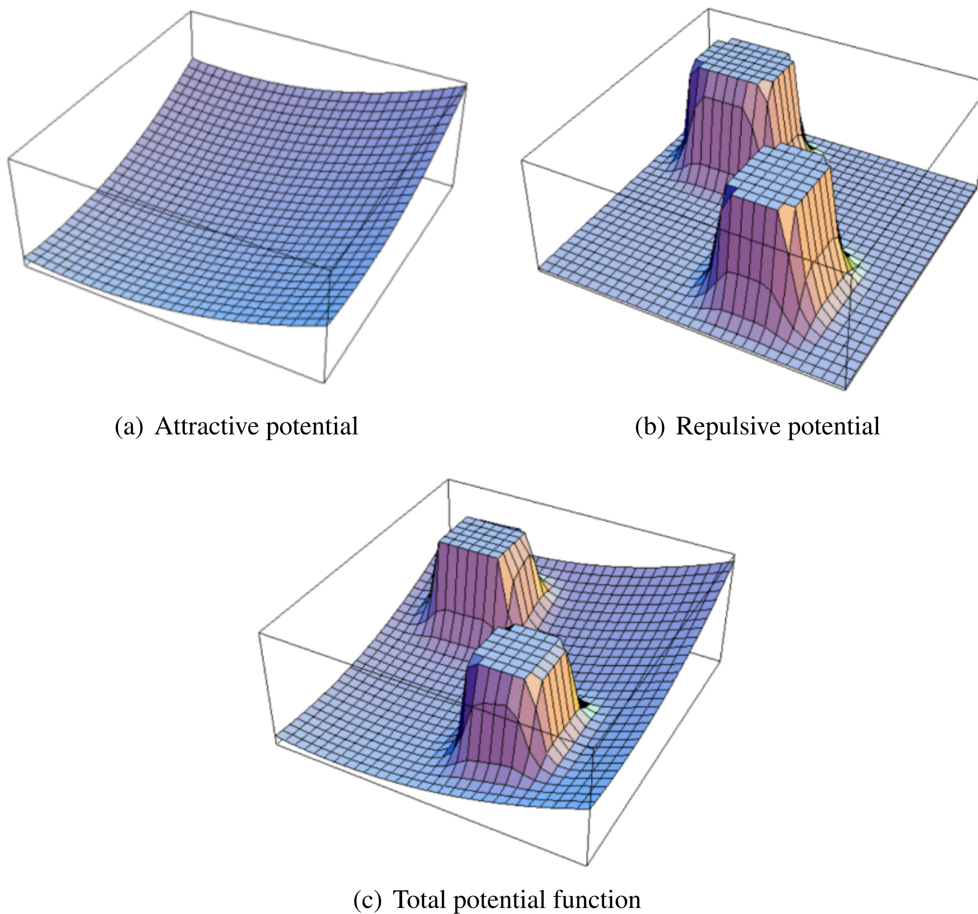
(b) The fish act as if they are attracted by a uniform force emanating from the light.

Figure 3.1: Animals analogy with potential field.
Sources: http://ulaulaman.tumblr.com/ , http://en.wikipedia.org/

In Figure 3.1(a), the fish keep away from the sharks as if they are repelled by a uniform force field surrounding the predators. In Figure 3.1(b), the fish are attracted by the light in a similar way.
The potential field method attempts to formalize this kind of behavior. This approach treats the robot as a point, at position $q = (x, y)$, under the influence of an artificial potential field $U(q)$. The goal acts as an attractive force on the robot while the obstacles act as repulsive forces. The potential field acting on the robot (see Figure 3.2(c)) can be computed as the sum of the attractive field of the goal (see Figure 3.2(a)) and the repulsive fields of the obstacles (see Figure 3.2(b)):

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{3.1.1}$$

It is important to note, though, that this is also a control law for the robot. Assuming the robot can localize its position inside the map taking into account the potential field, it can always determine its next required action based on the field.



(a) Attractive potential

(b) Repulsive potential

(c) Total potential function

Figure 3.2: Superposition of potential fields. Source: http://voronoi.sbp.ri.cmu.edu/

If we assume $U(q)$ to be a differentiable potential field function, we can define the related artificial force $F(q)$ acting at position q as:

$$F(q) = F_{att}(q) + F_{rep}(q) = -\nabla U_{att}(q) - \nabla U_{rep}(q) = -\nabla U(q), \tag{3.1.2}$$

where $\nabla U(q)$ denotes the gradient vector of U at position q. The vector $F_{rep}(q)$

points from the robot toward a point far from the obstacles and $F_{att}(q)$ points from the robot toward the goal.
The attractive potential U_{att} should converge to zero for q close to the goal. Such a potential can be dimensioned as follow:

$$U_{att}(q) = \frac{1}{2}k_{att} \cdot \|q - q_{goal}\|^2, \tag{3.1.3}$$

where k_{att} is a positive scaling factor and $\|q - q_{goal}\|$ is the distance between the robot and the goal. This attractive potential is differentiable and the corresponding attractive force $F_{att}(q)$ is given by:

$$F_{att}(q) = -k_{att} \cdot (q - q_{goal}), \tag{3.1.4}$$

which converges linearly toward zero as the robot reaches the goal. The repulsive potential U_{rep}, instead, should be zero if the robot is far from the object and should gradually take larger values as the robot gets closer to the obstacle. One example of such a repulsive field is:

$$U_{rep}(q) = \begin{cases} \frac{1}{2}k_{rep}\left(\frac{1}{\|q-q_{obs}\|} - \frac{1}{d_0}\right)^2 & \text{if}\|q - q_{obs}\| \leq d_0 \\ 0 & \text{if}\|q - q_{obs}\| > d_0, \end{cases} \tag{3.1.5}$$

where k_{rep} is a scaling factor and d_0 is the distance of influence of the object. The repulsive potential function is positive (or zero) and tends to infinity as the robot gets closer to the object. The corresponding repulsive force F_{rep} is:

$$F_{rep}(q) = \begin{cases} k_{rep}\left(\frac{1}{\|q-q_{obs}\|} - \frac{1}{d_0}\right)\frac{q-q_{obs}}{\|q-q_{obs}\|^3} & \text{if}\|q - q_{obs}\| \leq d_0 \\ 0 & \text{if}\|q - q_{obs}\| > d_0, \end{cases} \tag{3.1.6}$$

Under ideal conditions, by moving along $F(q) = F_{att}(q) + F_{rep}(q)$ and by setting the robot's velocity vector as proportional to this force, the robot can be smoothly guided in the direction of the goal while staying away from the obstacles.
There are some limitations with this approach. One is local minima, that appears to depend on the obstacle shape and size and which could also sacrifice completeness (the ability to reach a goal if a path exists). Another problem is the tendency to cause unstable motion in tight environment, resulting in oscillations between the obstacles.

3.2 Dynamic Navigation

Autonomy requires systems that are not only capable of controlling their motion in response to sensor inputs (e.g. when avoiding obstacles), but that are also capable to react to unexpected events, like qualitative changes in the environment

or strong perturbations. The navigation path of an intelligent system should not only contain trace paths, but also contain information for real-time tracking control. The dynamic approach to autonomous robotics was developed, in part, in response to this conceptual shift.
The main ideas behind dynamical navigation[35] are:

- Behaviors, that is all processes occurring along the stream from sensing to acting, are generated by ascribing values in time to behavioral variables. These variables are chosen such that tasks can be expressed as values of these variables.
- The time courses of the behavioral variables are obtained as attractor solutions of dynamical systems, the behavioral dynamics, formulated to express the task constraints through attractive or repulsive forces.
- Sensory information or information from other behavioral modules (dynamical systems of other behavioral variables) determine location, strength and range of attractive or repulsive contributions to the behavioral dynamics.

An example of behavioral variable is a robot's heading direction ϕ_h, measured over the behavioral dimension ϕ relative to a world fixed reference $\phi_0 = 0$. If the task is to drive to a target which is in direction ϕ_t, the behavior to be performed is called *target acquisition*.
The dynamic navigation solution in this work uses the concepts of potential field planning in an obstacle avoidance approach, creating a Virtual Force Field [36, 37].
Let the heading direction of the robot, ϕ_h, be the fixed world reference frame, repulsive obstacles are defined around each direction in which obstructions are sensed.
The obstacle/repellor *i* is characterized by:

- λ_i: the **strength** of the obstacle, that can be made a function of the distance to the obstacles, so that closer repellors are stronger.
- ϕ_i: the **direction** in which the obstacle is located.
- σ_i: is the **weight** of the obstacle. This parameter defines the basin of attraction of the obstacle in a way that only if the behavioral state lies within this range, the robot is affected by the obstacle. The weight can be made a function of the robot's diameter and the distance - directly proportional to the former, inversely proportional to the latter.

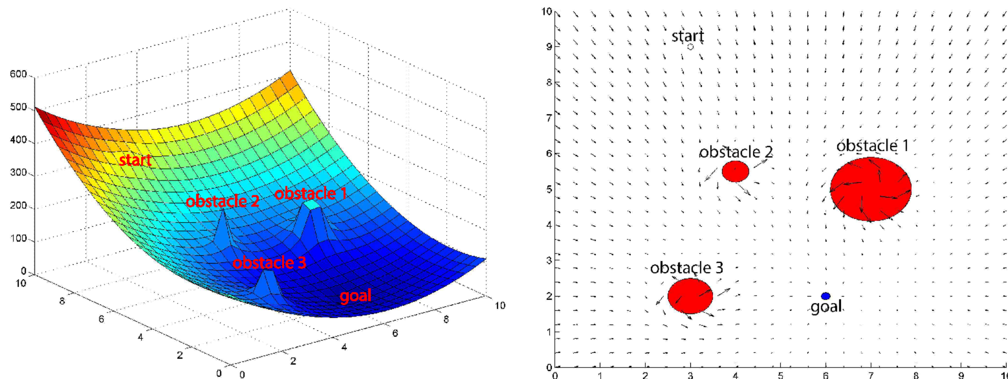


Figure 3.3: Dynamic Navigation in a Virtual Force Field. The obstacles are seen as peaks and the target is seen as a trough.
Source: U. Orozco-Rosas, O. Montiel, R. Sepúlveda, "Pseudo-bacterial Potential Field Based Path Planner for Autonomous Mobile Robot Navigation".

Attractors and repellors are characterized in the same way. The only difference between them is that λ_i is positive for attractors and negative for repellors.
Defining attractors and repellors in this way, creates a field of potential like the one we can see in Figure 3.3.
Letting $\dot{\phi_h}$ be the next direction that the robot have to follow to avoid the obstacles, the dynamical system can be expressed as:

$$\dot{\phi_h} = \sum_i \lambda_i(\phi_i - \phi_h)e^{\frac{(\phi_i - \phi_h)^2}{2\sigma_i^2}}. \tag{3.2.1}$$

As we can see, the sensing of the obstacles, as well as correctly dimensioning the three parameters, is of extreme importance to have a safe and successful autonomous navigation.

Chapter 4

The telepresence apparatus

The robot in this thesis will be used as a telepresence mobile apparatus. Such a robot could enable impaired end-users, who are constrained to remain in bed because of their severe degree of paralysis, to participate in various activities together with family and friends.
In this chapter we will discuss about the methodologies applied to create the telepresence apparatus: Gstreamer, an open-source library for streaming audio and video in a network, is here used to enable the communication between the end-user. Furthermore, the BCI will be used by the user as a tool to help the robot to reach a destination.

4.1 GStreamer

Figure 4.1: GStreamer logo. Source: http://gstreamer.freedesktop.org/

GStreamer [38] is a multimedia framework based on pipelines to create multimedia streaming applications such as video editors or media players. The core framework is written in C programming language with the type system based on GObject. The main idea of GStreamer is to link together various elements in a

graph-based way to obtain a stream that meets some desired requirements (see Figure 4.2). These elements are included in the framework or written by third party programmers. GStreamer handles managements of these elements, data flow and negotiation of the formats. Also, it is not restricted to handle multimedia formats only, but can handle any type of data stream. The pipeline design is made to have little overhead above what the applied filters induce. This makes GStreamer a good framework for designing even high-end audio applications which put high demands on latency.

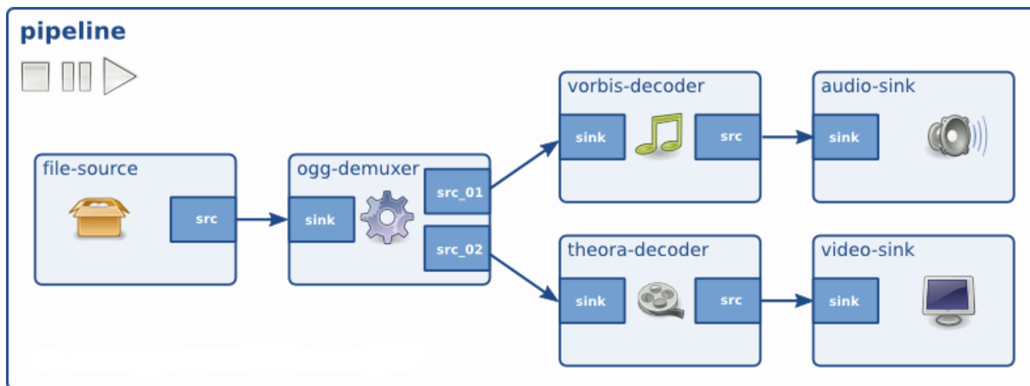


Figure 4.2: A typical GStreamer pipeline.
Source: http://gstreamer.freedesktop.org/

In particular, in this work we will use GStreamer to send video from the Kinect v2/Webcam and audio from the microphone to another laptop, using the network as medium.
We will now explain the basic concepts and objects that are necessary to build such a GStreamer pipeline: elements, pads, bins/pipelines, and communication.

4.1.1 Elements

An element is the most important class of objects in GStreamer. It is usual to create a chain of elements linked together and let data flow though this chain of elements. Every element is part of a plugin and has a specific function, which can be the reading of data from a file, decoding of this data or outputting this data to the sound card. Elements can be used as black boxes by the application programmer: given an input, the element will do something with it and will output something else. For a decoder element, for example, given encoded data as input, the element will output the decoded data. Several elements can be chained together, creating a pipeline that can do specific task, for example media playback or capture.
Elements can be broadly classified in the following categories:

- **Sources**: these elements generate data for use by a pipeline, for example by reading from disk or from a sound card. Source elements do not accept data in input, but only generate data. Figure 4.3(a) illustrates how this kind of element only has a source pad, an output always on the right, which can only generate data.

- **Filters, convertors, demuxers, muxers and codecs**: these elements have both input and outputs pads, meaning that they can receive a data stream, process it and provide it for other elements in the pipeline, for example a videoflip element gets a video in input and outputs a flipped and rotated version of it. Filter-like elements can have any number of source or sink pads (see Figures 4.3(b) - 4.3(c)).

- **Sinks**: these elements are in the role of consumers in the pipeline. They receive data streams and perform an action, usually to output the stream to a sound card, or display the video on a screen. They do not provide data for other elements in a pipeline (see Figure 4.3(d)). Disk writing, soundcard playback, video output can be implemented by sink elements.

4.1.2 Pads

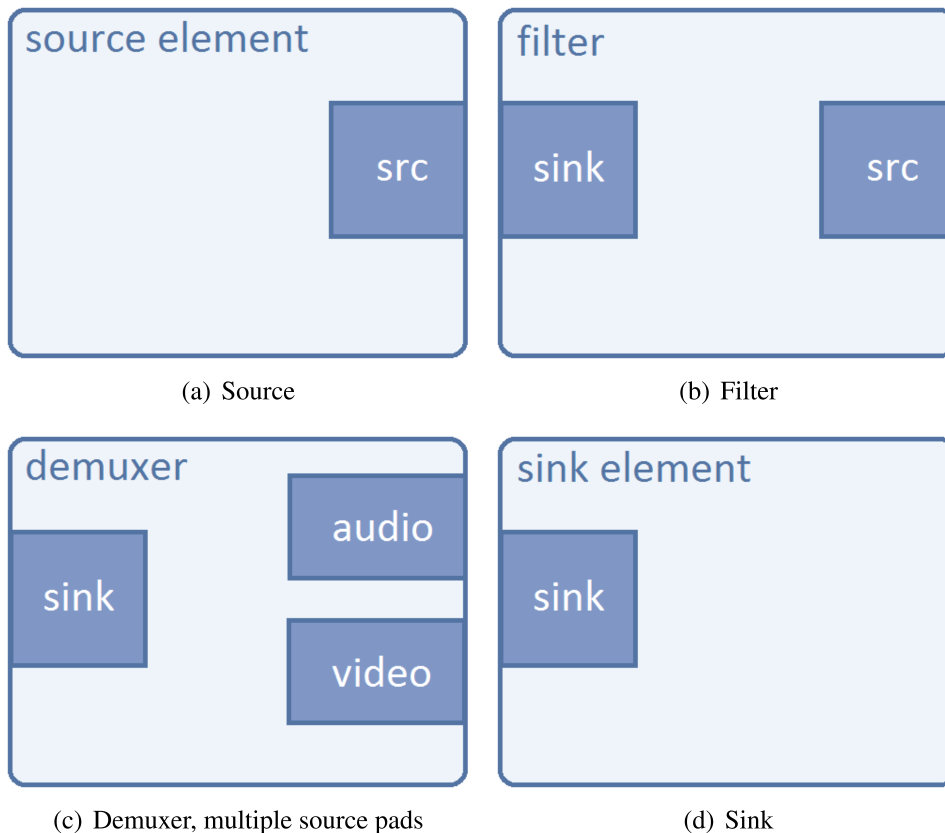
Pads are objects associated with elements through which data flows in or out of an element. Data streams from one element's source pad to another element's sink pad. Pads have specific data handling capabilities: a pad can restrict the type of data that flows through it. Links are allowed between two elements only when the data types of their pads are compatible. Data types are negotiated between pads using a process called caps negotiation.
A pad type is defined by two properties:

- **Direction**: GStreamer defines two pad directions: source pads and sink pads. Elements can receive data only on their sink pads and generate data only on their source pads.

- **Availability**: a pad can have any of three availability: always (it always exists), sometimes (it exists only in certain cases and can disappear randomly), and on request (it appear only if explicitly requested by the applications).

4.1.3 Bins

A bin (see Figure 4.4) is a container for a collection of elements. Since it is an element itself, a bin can be handled in the same way as any other element. Once an



(a) Source (b) Filter

(c) Demuxer, multiple source pads (d) Sink

Figure 4.3: Different categories for an element in GStreamer.
Source: http://gstreamer.freedesktop.org/

element is in a bin, there is no need to deal with it individually anymore, because the bin will manage the elements contained in it. It will perform state changes on the elements as well as collect, synchronize and forward bus messages. A specialize type of bin that every application needs to have is the pipeline.
A pipeline is a top-level bin that provides a bus for the application and manages the synchronization for its children. When a pipeline is set to PAUSED or PLAYING state, data flow will start and media processing will take place. One started, pipelines will run in a separate thread until stopped or the end of the data stream is reached.

4.1.4 Communication

GStreamer provides several mechanisms for communication and data exchange between the application and the pipeline:

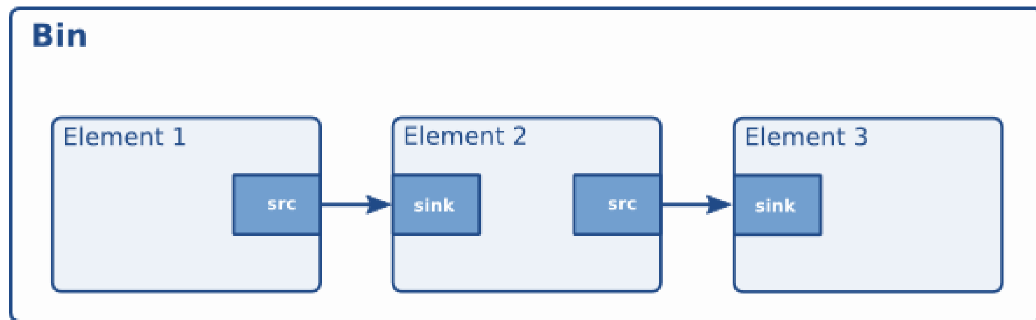


Figure 4.4: A bin.
Source: http://gstreamer.freedesktop.org/

- **Buffers**: are objects for passing streaming data between elements in the pipeline. Since streaming is always done from sources to sinks, these objects always travel downstream.

- **Events**: are objects sent between elements or from the application to elements. Since, in some cases, communication needs to be performed both upstream and downstream, events can travel in both directions. In addition, downstream events can also be synchronized with the data flow.

- **Messages**: are objects posted by elements on the pipeline's message bus, where they will be held for collection by the application. Messages are used to transmit information such as errors, tags, state changes, buffering state, redirects etc. from elements to the application in a thread-safe way. They are usually handled asynchronously by the application from the application's main thread, but can be also intercepted synchronously from the streaming thread context of the element posting the message.

- **Queries**: are objects used to query an element about specific information. They differ from events because they are always synchronously answered. Queries can be used by the elements or the application to query information about the current state of an element or of the whole pipeline. They can travel both upstream and downstream, but upstream queries are more common. Example of a query is a query to find out the information about the duration of a video stream.

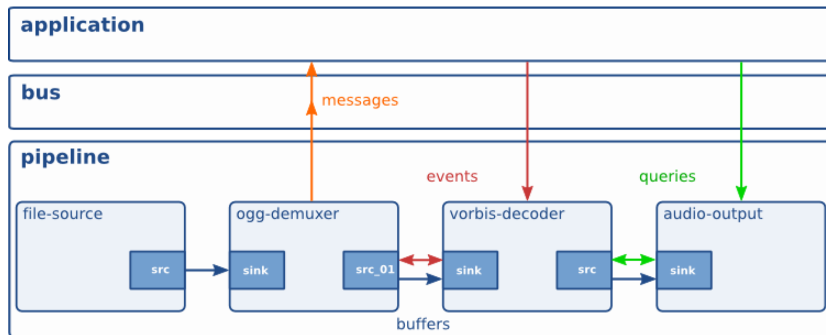


Figure 4.5: A GStreamer pipeline with different communication flows.
Source: http://gstreamer.freedesktop.org/

4.2 Brain Computer Interface

The idea of interfacing minds with machines, along with the ability to control them using high-level commands, has long captured the human imagination. Over the last years, it has been possible to actually bypass the conventional neural pathways, such as muscles or speech, connecting directly the human brain to a computer. The capability to use brain signals as new communication and control channel is called Brain Computer Interface (BCI). This framework [39, 13] monitors the users encephalography (EEG) activity and translates their intentions without activating any muscle or peripheral nerve.
The immediate goal of BCI research is to provide communication abilities to severely disabled people who are totally paralyzed or "locked in" by neurological neuromuscolar disorders, such as amyotrophic lateral sclerosis, brain stem stroke, or spinal cord injury. In the latter case, people cannot move their arms, legs, or eyes, and usually depend on an artificial respirator, because any voluntary control of muscles is lost. BCI in this panorama results an effective and reliable tool to create an alternative interface, so that they will be able to communicate with the environment again.
BCI systems can be classed as **exogenous** or **endogenous**, depending on the nature of the input signal. Exogenous BCI systems depend on neuron activity evoked by external stimuli and do not require intensive training. Such stimuli include Steady State Visual Evoked Potentials (VEPs) or P300 evoked potentials. The control can be activated with a single EEG channel by analyzing the changes in the signal right after a visual or auditory stimulus.
In contrast, endogenous systems do not rely on an external stimulus, in fact they depend on the user's ability to control their electrophysiological activity, such as the EEG amplitude in a specific frequency band on a specific area of the cerebral

cortex. These systems include those based on slow cortical potentials (SCPs) and on Motor Imagery (sensorimotor rhythms). In particular, those based on Motor Imagery (MI) exploit the correlation between specific EEG oscillations and the imagination of movements.
MI is a cognitive process generated when the subject imagines movement performance without actually executing it. It is a dynamic state during which the representation of a specific motor action is internally activated without any motor output. In other words MI requires the conscious activation of brain regions that are also involved in movement preparation and execution, accompanied by a voluntary inhibition of the actual movement. The imagination of different types of movements (e.g. right hand, left hand or feet), results in an amplitude suppression (event-related desynchronization ERD) or in an amplitude enhancement (event-related synchronization ERS) of Rolandic mu rhythm (7-13 Hz) and of the central beta rhythm (13-30 Hz) recorded over the sensorimotor cortex of the user [40]. BCIs based on the SensoriMotor-Rhythms (SMR) are then able to classify, thanks to this variations in the EEG oscillations, the mental state of the subject, in order to drive their output. The task that is given to the user can change, depending on the control signal that has to be extracted and on the application of interest, but it has to maintain some standard characteristics:

- be **simple**, because even unhealthy user must be able to perform it multiple times.
- generate **significant brain signals**, to ensure **repeatability** and **accurate interpretation** by the classifier.
- The paradigm used must be such as to involve brain processes **easy** to **activate** and to **control** and **fast** in the **exhaustion**.

It is important that the task is well recognized, given these significant signals, because an unmotivated user can quickly lose interest, if the system is not rewarding. The lose of interest leads to a worsening of the signal generation resulting in lower performance, demotivating the user even more.

4.2.1 The BCI loop

Independently to the type used, every BCI has loop-based architecture as the one we can see in Figure 4.6. The BCI loop is composed by three main different blocks: **subject - signal acquisition**, **signal processing - machine learning**, and **online feedback - AT devices**. Each block can have an independent implementation, even with respect to the same BCI paradigm. Given the modularity of the loop, we will now explain each block in detail.

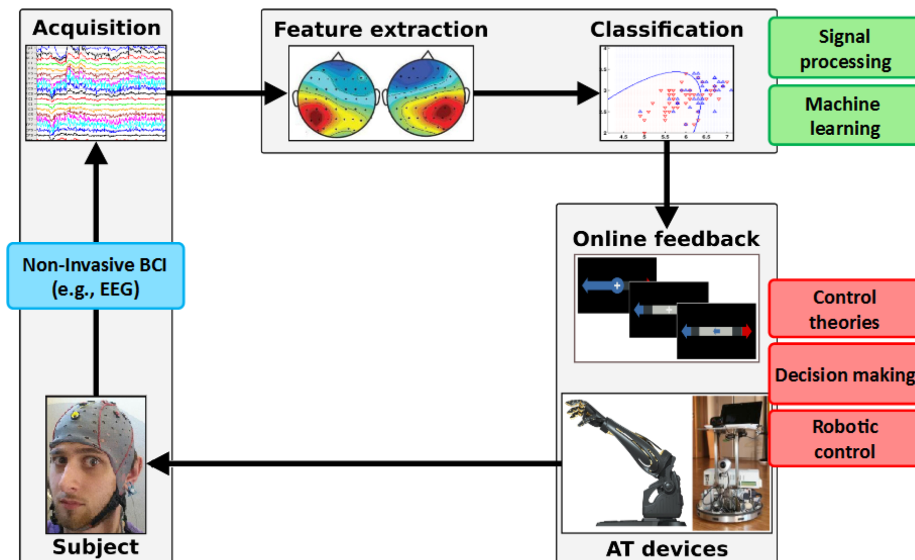


Figure 4.6: The BCI loop

4.2.1.1 Subject - Acquisition Block

The subject represents the starting and the arrival point of the loop. In fact it is the subject, or user, that generates the input of the loop. This input must be a repeatable signal that can be classified (e.g. EEG signals) and then used to drive the framework. Thanks to the online feedback block, at the same time, the user can adapt to the interface in order to achieve the desired task.
We cannot talk about BCI without putting the subject in the loop. Every BCI application have to be tested and validated with users included in it.
Regarding the acquisition of the signals, there are several types of Brain Computer Interfaces that are reported. Those types differ in how much the acquisition device is invasive into the human body [41]:

- **Invasive:** these acquisition devices are directly implanted into the brain tissue from the cerebral cortex. This invasive signal acquisition requires surgery to implant the sensors. Electrodes are implanted by opening the skull through a proper surgical procedure called craniotomy and then placing them on the cortex. The signals acquired are called electro-corticogram (ECoG) or Invasive EEG. ECoG recording techniques combine excellent signal quality, very good spatial resolution, and a higher frequency range thanks to the employments of micro-electrodes array. These techniques are used to provide functionality to paralyzed people. Invasive BCIs are also used to restore vision by connecting the brain with external cameras and to restore the use of limbs by using brain controlled robotic arms and legs.

Due to surgery requirement these BCIs do not provide a practical instrument, even if they can grant fast and meaningful information. Ethical, financial, and other considerations make neurosurgery impractical except for some users who need a BCI to communicate. Moreover it is still unclear if these devices can continuously provide strong signals over the years, mostly because they are prone to scar-tissue build-up, causing the signal degeneration. Invasive techniques, due to their drawbacks, are almost exclusively investigated in animal models or in patients who undergo neurosurgery for other reasons, such as treatment of epilepsy.

- **Non Invasive:** They represent the least invasive and lowest cost devices. They are also considered to be more practical, because they don't need surgery operation and they only need to wear a EEG cap. This leads to a lower signal clarity, poor spatial resolution, and a lower frequency signals due to the tissue of the cranium that deflects and deforms signals. Non Invasive BCIs use different techniques to record these signals:
  - Surface **Electroencephalogram** (sEEG) uses electrodes to record electrical activity.
  - **Magnetoencephalogram** (MEG) records magnetic fields produced by electrical currents using very sensitive magnetometers.
  - **Positron Emission Tomography** (PET), **functional Magnetic Resonance Imaging** (fMRI), and **optical imaging** record brain metabolic activity, reflected in changes in blood flow.

  Most BCIs rely on the sEEG acquisition method because it appears to be an adequate alternative for its good time resolution and relative simplicity. The main source of the EEG is the synchronous activity of thousands of cortical neurons. Therefore measuring EEG is a simple non invasive way to monitor electrical brain activity, but it does not provide detailed information on the activity of single neurons (or small brain areas) and is characterized by small signal amplitudes and noisy measurements.

  On the other hand, magnetic field and blood flow based techniques require sophisticated devices that can be operated only in special facilities. Moreover, techniques for measuring blood flow have long latencies and thus are less appropriate for real-time interaction, typical of BCI applications for robotic device control. This mean that, because of its low cost, portability and lack of risk, sEEG is the ideal modality to bring BCI technology to a large population.

- **Partially Invasive:** this kind of devices are implanted inside the skull, but that rest outside the brain rather within the grey matter. Signal strength

using this type of BCI is weaker than fully invasive signals. They produce better resolution signals than non-invasive BCIs where the bone tissue of the cranium deflects and deforms signals and have less risk of scar tissue formation in the brain when compared to fully invasive BCIs.

4.2.1.2 Processing - Machine Learning Block

Due to the low magnitude of the raw EEG signals, about 100 μV, a first step of signal amplification has to be applied. The acquired analogical signals are then, thanks to a wired/wireless amplifier, magnified up to 10000 times and digitalized. The resulting signals have to be processed for noise extraction and correction. EEG recordings typically contains unwanted signals as interference from electronic equipment, Electromyography (EMG) activity evoked by muscular contraction or ocular artifacts, due to eye movement or blinking. Those components may bias the analysis of the EEG resulting in algorithm performance reduction. The goal of pre-processing is to improve signal quality by improving the signal-to-noise ratio (SNR). Lower SNR means that the brain patterns are occluded in the rest of the signal, so relevant pattern are hard to detect. Higher SNR, on the other hand, simplifies the BCI's detection and classification task. The signal is spatially and frequency filtered in order to achieve the higher SNR as possible.
The brain patterns used in BCIs are characterized by certain features or properties, as amplitudes and frequency of EEG rhythms, that need to be extracted after the signal processing. These features depend on the paradigm that is established to be used. It is important to extract a set of meaningful features, in order to avoid overfitting problems in machine learning and to reduce the computational time.
The next important step in the BCI loop is the classification of the signals and the subsequent task identification using these selected features. Classifier are usually implemented through Machine Learning techniques as Artificial Neural Networks (ANN) [42], Support Vector Machines (SVM) [43], Bayesian Networks [44], Deep Learning [45], etc. The output of the classifier is expressed as a probability that the input feature set refers to one class with respect to the other.

4.2.1.3 Online Feedback - AT Devices Block

After the features classification, the user must be able to get a feedback indicating the current mental state recognized by the embedded classifier. This feedback can be tactile or auditory, but most often it is a visual, graphical representation of a decision making algorithm output. The decision making algorithm employs the classifier output probabilities for interpreting the user intention and deliver the desired command. The feedback is a necessary block in the BCI loop, because the user must be able to understand the state of the BCI system and adapt to

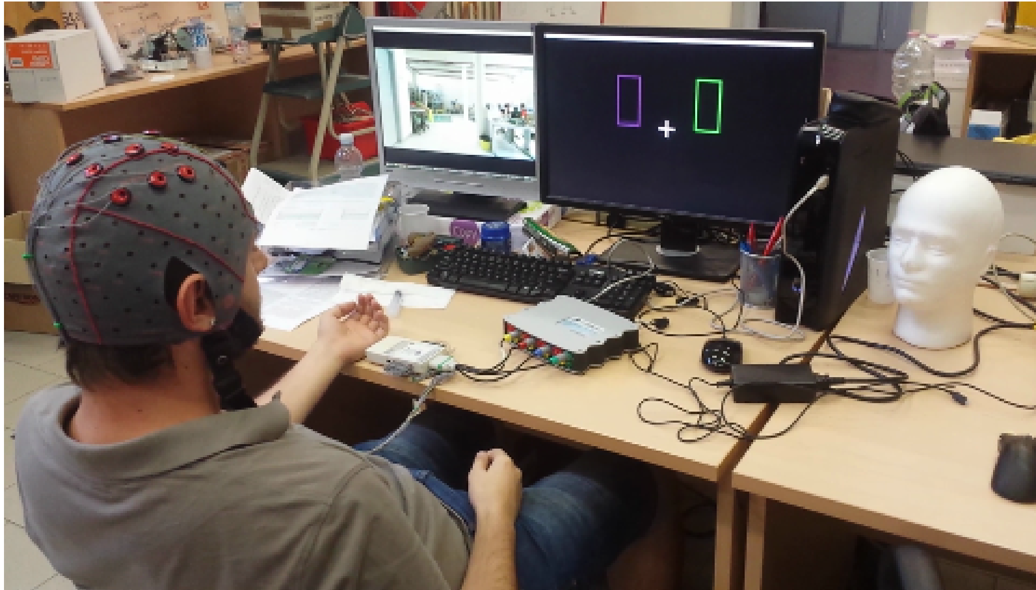


Figure 4.7: A BCI user with the feedback from BCI on the right monitor and the telepresence robot vision on the left monitor

it, improving the final performance. With a feedback the user can have a finer control of the system. Thanks to the feedback the user can understand if he/she is performing the right task and, if not, change aptitude to deliver the correct one. The user can also use the feedback in order to not deliver any mental command by continuously balance the mental tasks.

The feedback component is really important in the BCI loop also because the final applications are more demanding to the user than the simply control over BCI, so it must be as intuitive as possible. In fact, the user must split his/her attention between the BCI feedback and the application control (dual task). Moreover in case of particular applications, e.g. telepresence robot control (see Figure 4.7), the mental task must be performed with certain temporal precision, requiring intense attention of the subject. For this reason, in the last years many applications for assistive technology focus their attention on the creation of frameworks able to relieve the workload on the user, in the so called shared control. In a shared control system there is a cooperation between the Assistive Technology (AT) devices, e.g. a mobile robot, and the end-users in order to complete a task. Usually the AT device will take care of low level task, as the detection and avoiding of the obstacles, instead the end user will have high level task, as giving direction or stopping the AT device.

If the feedback is the representation of the state of the system, the AT devices are the final actuators of the decision making algorithm. In other words, AT devices

are what allow the user to interface himself/herself to the real world. BCI as a proof-of-concept has already been demonstrated with several AT devices: driving a robot (telepresence) or a wheelchair [46], playing video-games [47], operating prosthetic devices [48], navigating in virtual realities [49], moving cursors [50], internet browsing [51], supporting spelling device [52], etc.

(a) Playing video-games (b) Internet browsing

(c) Driving a wheelchair

Figure 4.8: AT devices BCI driven

Chapter 5

Implementation Details

In this section we will show how the dynamic autonomous system has been implemented, pointing out the difficulties found and how the system evolved to overcome them. Since the type of navigation that we use in this work heavily relies on sensor data acquired in real-time in order to keep the robot from hitting obstacles, the first part of the section will be dedicated to the analysis and merging of data from different sensors into a consistent map. Then we will show how manual control, using a keyboard, has been implemented, in order to help the robot to decide at crossroads. Finally, we will transform the system into a deterministic finite automaton (DFA).

5.1 Navigation with Hokuyo laser scan

We started by implementing the dynamic navigation algorithm using only the data derived from the Hokuyo laser scan. The reasons for this choice are:

- Laser scans are easier to use than point clouds because they carry less data (only a one dimensional slice of the world).
- Laser scan data can be used almost as is.
- Hokuyo laser scanner has a wide scan area (240°), with a long detection range (up to 5.6m).

The first thing we did was the acquisition of the laser scan data through the topic **/scan**. In this topic messages of type *sensor_msgs/LaserScan* are published by the Hokuyo sensor. These messages contain these information:

- std_msgs/Header **header** contains a timestamp, that is the acquisition time of the first ray in the scan, and the frame_id of the scan.

- float32 **angle_min** is the starting angle of the scan, measured in radians. This angle is -2.09 rad (-120°) for the Hokuyo.
- float32 **angle_max** is the ending angle of the scan, measured in radians. This angle is 2.09 rad (120°) for the Hokuyo.
- float32 **angle_increment** is the angular distance between measurements, in radians. This measure is the pitch angle and with our laser scanner is 0.0061 rad (0.36°).
- float32 **time_increment** is the time between measurements, in seconds.
- float32 **scan_time** is the time between scans, in seconds.
- float32 **range_min** is the minimum range value, calculated in meters. For the Hokuyo it is 0.20m.
- float32 **range_max** is the maximum range value, calculated in meters. For the Hokuyo it is 5.60m.
- float32[] **ranges** is an array that contains the measured distances, in meters.
- float32[] **intensities** is an array that contains the measured intensities. Not all laser scanners calculate intensities and in this case the vector must be empty, like with the Hokuyo laser scanner.

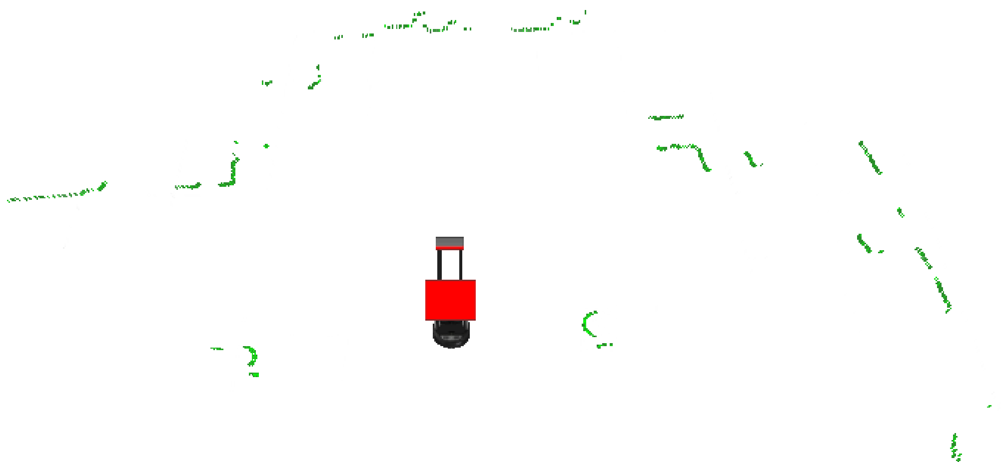


Figure 5.1: Visualization in RViz of a laser scan

In Figure 5.1 we can see the RViz visualization of a laser scan taken from the Hokuyo sensor.
The next step was transforming the laser scan into a useful data type. In order to do that, we have created a custom message type, **dyn_msgs_Obstacle**, with this definition:

- float32[] **strength**
- float32[] **direction**
- float32[] **weight**

The custom message is composed by three arrays that specify the three components for the dynamic navigation. It is assumed that the obstacle i will have its strength, direction and weight defined at the index i of the array.
Once we had created our message type, we were ready to transform a *sensor_ msgs/ LaserScan* into a *dyn_ msgs/ Obstacle*. In order to do so, after receiving the message through its topic, we saved constant parameters like *angle_min*, *angle_max*, *angle_increment*, *range_min* and *range_max* and then we iterated through the *ranges* array to get the positions of the obstacles.
In order to get useful information, we had to discard all those measures that were *NAN*, meaning that the ray had not found an obstacle, or greater than *maximum_range* or lower than *minimum_range*. The valid obstacles were saved in the *dyn_msgs/Obstacle*'s three array, using these formulas:

$$\begin{aligned} \text{strength}[i] &= \frac{1}{\text{ranges}[i]} \\ \text{direction}[i] &= \text{angle_min} + \text{angle_increment} \cdot i \\ \text{weight}[i] &= \frac{\text{robot_diameter}}{\text{ranges}[i]} \end{aligned} \tag{5.1.1}$$

These formulas have been created by taking into account that strength must be inversely proportional to the distance from the obstacle. Weight must be directly proportional to the robot dimension and inversely proportional to the distance from the obstacle. The direction of the obstacle was calculated keeping in mind the discarded obstacles, in order to get the right direction.
In this first implementation of the algorithm, the *dyn_msgs/Obstacle* message was being sent directly to the node *Navigation* that controlled the robot movement. This node took the message from the apposite topic and then used the data in the message in order to calculate the new direction to follow, through the equation 3.2.1. The angle ϕ_h was fixed to 0, because the robot did not move on a prefixed map and so its heading did not influence the calculation to reach a target.

Initially, the linear velocity was kept constant and low enough to better understand if there were errors during the test, specially in the next direction calculation. The message, of type *geometry_msgs/Twist*, was then published on the topic */cmd_vel_mux/input/navi*.
The robot in this work will not have a target. It will navigate through the environment keeping a straight direction until an obstacle is found. In this work we propose a semi-autonomous navigation, thus, the responsibility to stop the robot when the goal is achieved is upon the user.
A problem that we found during the test phase of the Navigation node was that the robot moved intermittently after publishing a message. This was probably due to how the TurtleBot package was designed since, with other robots, the velocity remains constant to the value of the last message. In order to get the same behavior, no matter the velocity control implementation, a new node has been created.

This node is an intermediate step between the Navigation node and the publishing of the new velocity to the robot. In its first implementation, it only took the message from the topic on which the Navigation node published and it continuously published it on the */cmd_vel_mux/input/navi* topic, until a new message arrived.

This behavior could also be achieved in a single node, but we wanted to differentiate the tasks of the nodes: calculation of the new velocity on one node and maintaining the velocity on the other. The implementation of this node allowed the robot to perform smooth movements, so we could dedicate ourselves to improve the dynamic system.

5.2 Parameters tuning and map visualization

Using the Equation set 5.1.1, different tests were executed on the real robot, pointing out some problems. The most important was that the strength and weight parameters were not tuned correctly for a real robot. Obstacles were taken into account only when the robot was very close to them and the corrective measure adopted was excessive, resulting in a sudden change of direction that could even make the robot fall on the ground.
The first thing we tried to do was to add some scaling factors to the formulas (in the Equation set 5.1.1 the direction is bound to be correct), leading to:

$$\begin{aligned}
\text{strength}[i] &= \frac{\text{max_strength}}{\text{ranges}[i]} \\
\text{direction}[i] &= \text{angle_min} + \text{angle_increment} \cdot i \\
\text{weight}[i] &= \frac{\text{robot_diameter} \cdot \text{max_weight}}{\text{ranges}[i]}
\end{aligned} \tag{5.2.1}$$

We performed different tests in order to tune the *max_ strength* and *max_ weight* parameters, but the results did not change much. Upon increasing max_ strength the robot would turn more violently. This led to excessive corrective measure, as the robot took into account only very close obstacles. Upon increasing max_ weight the robot would turn by an angle wider than necessary. This also increased the maximum distance within which an obstacle was considered. On the other hand, upon decreasing said parameters, the robot would collide with the obstacles. So, we needed to find a way to visualize the effects of increasing and decreasing those parameters, as well as to tune them better.
Another issue linked to the visualization problem was to merge data from different sensors: even if we sent a *dyn_msgs/ Obstacle* message to the same topic of the laser, the obstacles were not saved anywhere and the same obstacle would be taken into account twice in the summation. This behavior could be desired, but we decided to use a more general approach: Occupancy Grid Mapping.
One of the contributions of this work is the implementation of occupancy grid mapping with dynamic navigation.
Occupancy grid maps are spatial representations of the robot's environment. The grid representation is based on a multidimensional (2D or 3D) tessellation of space into cells, where each cell stores a probabilistic estimate of its state. In our case the state represents whether or not an obstacle is in the cell (binary occupancy grid). There are several reasons to prefer an occupancy grid mapping approach, but the most important for us is the easier **interpretation** and **integration** of data.
The system was reviewed with the introduction of occupancy grid mapping. One of the first important changes was the sending of a *nav_msgs/ OccupancyGrid* message to the topic in charge of navigation, instead of a *dyn_msgs/ Obstacle*.
The specifications for a *nav_msgs/OccupancyGrid* message are:

- std_msgs/Header **header** contains a timestamp, that is the acquisition time of the map, and the frame_id of the sensor.
- nav_msgs/MapMetaData **info** contains the basic information about the characteristics of the occupancy grid, like resolution (m/cell), width (cells) and height (cells).
- int8[] **data** is the map data, in row-major order. Probabilities are in the range [0,100] and unknown is -1.

The parameters defining an occupancy grid (resolution, width and height) have been dimensioned so that the resolution could take into account the sensors imprecision and width and height could cover the maximum range of the sensors. The parameters are settable in a yaml file, but their default values are:

- **Resolution**: 0.06 m/cell; this way a distance measured with the Hokuyo sensor, that has an accuracy of $\pm 3cm$, will be wrong by a cell at most.

- **Width/Height**: 200 cells; so that we could cover an area of 12m x 12m.

The reason why the size of the map is more than twice the maximum range of the laser (5.6m), is that the robot will be at the center of the map. This is useful because this way we can use the data measured behind the robot, since the Hokuyo has a scan area of 240°.
The idea behind the implementation of the occupancy grid is to merge data from different sensors, so the system design was changed accordingly, as we can see in Figure 5.2, where the final implementation of the system is represented. Instead of sending the messages directly to the topic to which the node that moves the robot is subscribed, the message is sent to another topic (*/occupancy_grids*). The node that merges the data takes these messages and it publishes a merged occupancy grid to the */merged_occupancy_grid* topic that is used to draw the map and to make the robot move.
This way if new sensors were to be added to the robot, it is sufficient to create an occupancy grid from the sensor data and publish the message on the */occupancy_grids* topic. No further modifications are required in order to use the algorithm.

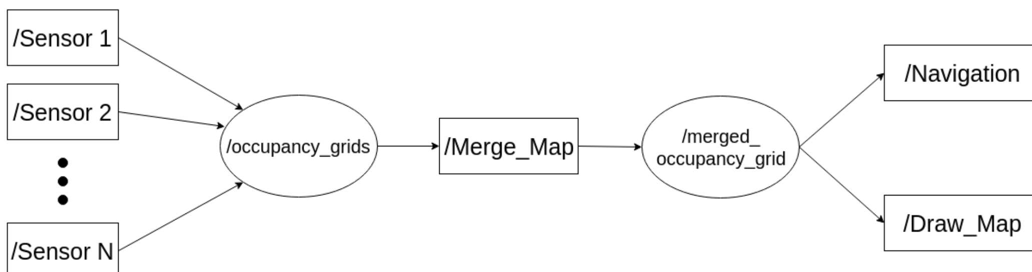


Figure 5.2: Part of the designed system. Nodes are represented as rectangles and topics as ovals.

With the occupancy grids implementation, the *dyn_msgs/Obstacle* message has been changed so that the variables are no longer arrays, but one-dimensional variables. It has also been incorporated in a new type of message that we created, the *dyn_msgs/DynamicOccupancyGrid* message, with the following definition:

- std_msgs/Header **header** contains a timestamp, that is the moment in which we started merging the map, and the frame_id of the merger.

- nav_msgs/MapMetaData **info** contains the basic information about the characteristics of an occupancy grid, like resolution (m/cell), width (cells) and height (cells).
- int8[] **obstacle** is the map data, in row-major order. Probabilities are in the range [0,100], -1 if the value is unknown.
- bool **near_obstacle** true if an obstacle is within a security range in the front of the robot.
- dyn_msgs/Obstacle[] **data** strength, direction and weight of the obstacles. The data array and the obstacle array have matching indices.

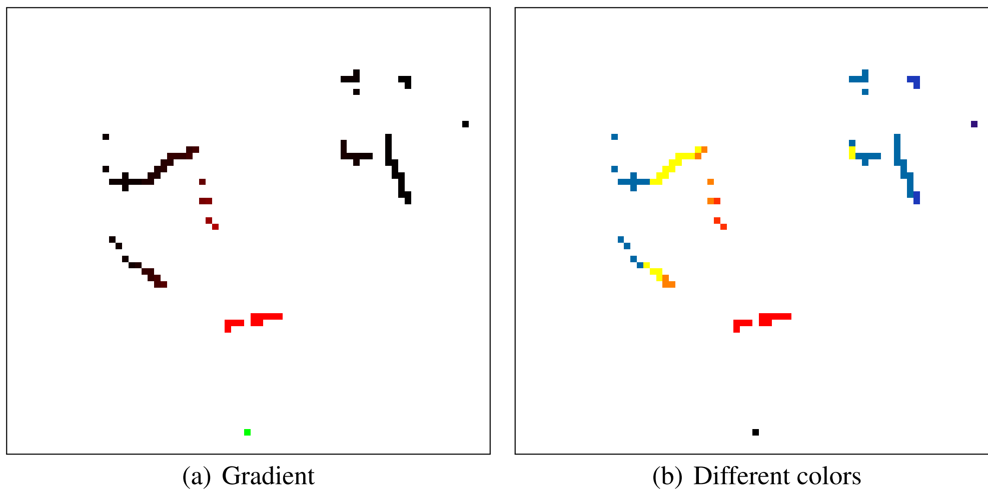
The Navigation node, after receiving the *dyn_msgs/DynamicOccupancyGrid*, has to iterate through the obstacle array, searching for cells with value > 0. Then, in order to calculate the next direction, it takes the strength, direction and weight at the indices corresponding to those cell in the same index of those elements. The near_obstacle variable is useful when the robot is near to an obstacle and it has to stop, in order to avoid the obstacle more safely.
The node that merges the occupancy grids publishes the merged occupancy grid, through the *dyn_ msgs/ DynamicOccupancyGrid* message, on the */merged_ occupancy_ grid* topic.
The Draw_Map node subscribes to this topic and uses the map message and OpenCV utilities to draw the map of the robot field of view. In this way we can control which obstacles have more influence on the navigation. To do so, we colored the different obstacles using the following parameter as a reference:

$$\text{obs_force} = \text{obs_strength} \cdot e^{\frac{(\text{obs_direction})^2}{2\cdot(\text{obs_weight})^2}} \tag{5.2.2}$$

This parameter defines the obstacle influence in the calculation of the new direction and, as we can see, it is part of Equation 3.2.1. Two ways to color the obstacles have been tried:

- **Gradient**: the maximum force has been calculated and the obstacles with the maximum force has been given the red color. The other obstacles has been colored using a shade of red, where higher force has more red component. The robot is the green point on the map. As we can see in Figure 5.3(a), the map is not understandable for colors other than red and this approach has been set aside.
- **Different colors**: seven different colors have been used. In descending force order we have: red, dark orange, orange, yellow, light blue, cyan and dark blue. Each color corresponds to a different percentage range of the

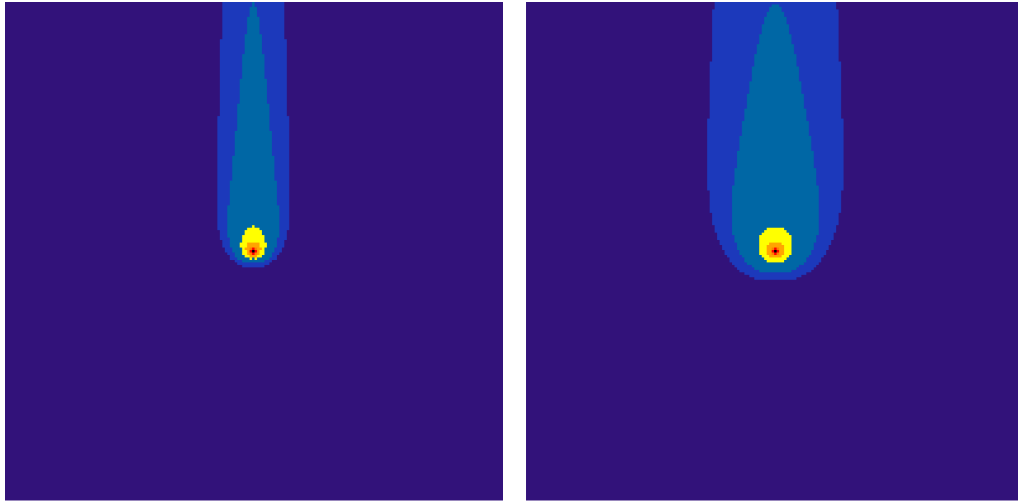
maximum force. The robot is the black point on the map. In Figure 5.3(b) we can see that now we can understand more clearly which obstacles have major and minor impact to the navigation.



(a) Gradient (b) Different colors

Figure 5.3: Dynamic Occupancy Grid visualization. The maps were cropped in the area behind the robot because there were no obstacles.

With the visualization help and the occupancy grid approach we could dedicate ourselves to parameters tuning. In order to do that, we created a test node that simulated a sensor. This node simply sent a *nav_msgs/OccupancyGrid* message to the */occupancy_grids* topic. The contents of this message could be changed as needed. For the sake of understanding how the changes in the *max_force* and *max_weight* parameters (in the equation set 5.2.1) affected the systems, we sent an occupancy grid with every cell occupied by an obstacle. The results can be seen in Figure 5.4.
The Figure gives a comparison between the implementation of Equation 5.1.1 and Equation 5.2.1. As we can see in Figure 5.4(b), increasing the parameters permits to consider more obstacles, but the main problem is that the closest obstacles still monopolize the navigation. This means that using $\frac{1}{\text{ranges}[i]}$ to calculate strength and weight does not allow the robot to fully utilize its sensory data to calculate the new direction. We decided to achieve the inverse proportionality needed, with respect to the range parameter, using a gaussian, so that the obstacle force would decrease less quickly. We centered it at a cell distance from the robot, in order to get the maximum strength at an immediately near obstacle. The corresponding new equations are:



(a) Implementation of the equation set 5.1.1

(b) Implementation of the equation set 5.2.1 (with increased strength and weight)

Figure 5.4: Influence of the obstacles in two different implementations.

$$\begin{aligned} \text{strength}[i] &= \text{strength_param} \cdot \exp^{-(\text{ranges}[i]-\text{resolution})} \\ \text{direction}[i] &= \text{angle_min} + \text{angle_increment} \cdot i \\ \text{weight}[i] &= \text{weight_param} \cdot \text{robot_diameter} \cdot \exp^{-(\text{ranges}[i]-\text{resolution})} \end{aligned} \tag{5.2.3}$$

The implementation of these equations in our system led to Figure 5.5.
As we can see, we obtained a more predictive system, where the obstacles with the highest impact are in front of the robot. The obstacles behind the robot are not almost taken into account, while those at the sides are more prominent than before.
We tried these parameters on the real robot, achieving a smoother and safer navigation. The robot was able to consider farther obstacles, so it could adjust its trajectory more gently and could reach farther goals. Although many problems have been solved using the new equations, new issues arose. In open spaces the robot navigated avoiding all the obstacles, but in tight places, as corridors, the robot started to swing to the left and to the right. This was due to the fact that the two walls were not taken into account both at the same time, so the robot would see the wall to its right, it would turn in order to avoid it and then it would see the wall to its left. We tuned the *weight_param* in order to give more prominence to the side obstacles. Even though the corridors problem was solved, the robot would not pass through the doors, because higher weigth_param made it too narrow to pass.

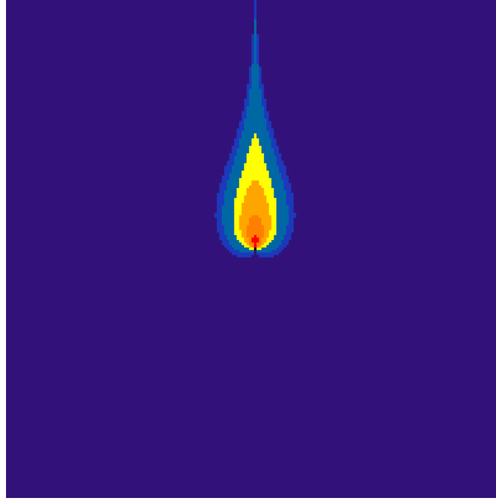


Figure 5.5: Implementation of the equation set 5.2.3

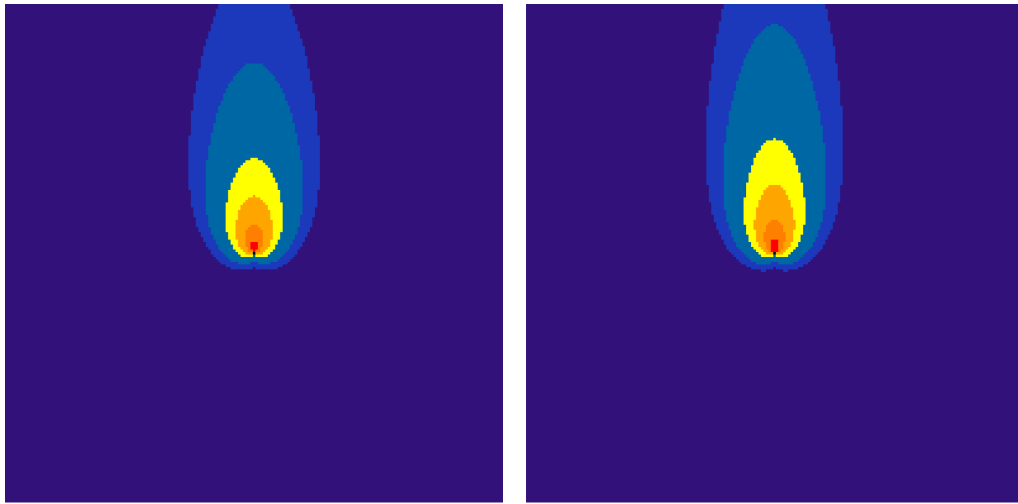
In order to overcome these problems, we tried to change approach modifying the equations, as suggested in [36]. We also introduced a normalizing factor (number_of_obstacles) in the strength equation, as well as the value of the occupancy grid cell (map[i]), that will be useful when we will use a probabilistic approach and not a binary one. The new equations became [1]:

$$\begin{aligned}
\text{strength}[i] &= \text{map[i]} \cdot \text{strength_param} \cdot \left(\frac{1}{\text{num_of_obstacles}}\right) \cdot \exp^{-\frac{\text{ranges}[i]-\text{resolution}}{\text{decay_param}}} \\
\text{direction}[i] &= \text{angle_min} + \text{angle_increment} \cdot i \\
\text{weight}[i] &= \text{weight_param} \cdot \arctan\left[\tan\left(\frac{\text{angle_resolution}}{2}\right) + \right. \\
&\quad \left. + \frac{\text{robot_diameter}}{\text{robot_diameter} + \text{ranges[i]}}\right]
\end{aligned} \tag{5.2.4}$$

In Figure 5.6 we can see the results of these new equations.
With this implementation we could finally give meaning to the parameters on which the equations were based:

- The **strength_param** defined how much strength an obstacle had, that is, how much it influenced the final direction of the robot. Greater strength would make the robot turn more violently, while lower one would make the robot turn more gently, causing the robot to pass more close to the obstacles.

[1] In these equations the *resolution* is the distance in meters between two adjacent cells, while the *angle_resolution* is the minimum angle representable in the map. Both are fixed parameters.



(a) Implementation of the equation set 5.2.4

(b) Implementation of the equation set 5.2.4 (with increased decay param)

Figure 5.6: Influence of the obstacles in the final implementation.

- The **decay_param** defined the area of influence in front of the robot. If the decay_param was high, farther obstacles would be taken into account for the calculation of the new direction (see Figures 5.6(a)-5.6(b)).

- The **weight_param** defined the area of influence on the side of the robot. Higher weight_param meant that more obstacles on the side would be considered.

We tuned the parameters so that we could keep the weight parameter low enough to pass through doors, but high enough to consider two opposite walls in a corridor. The decay parameter was increased a bit in order to keep into account farther obstacles, while not too far from the robot. The strength parameter was kept as in the previous implementation. The final implementation can be seen in Figure 5.6(b). These three parameter can be tuned in a yaml file and, as a consequence, it is possible to adapt the navigation to different situations and different robots, in case the default parameters do not work.
During the test phase the robot could navigate autonomously in the environment, so we got a working system using only the laser. The navigation, however, was far from being safe: indeed there were multiple problems to the navigation that we could not overcome. The Hokuyo laser scan cannot yaw, so there were obstacles that could not be seen from it. Two common examples of these obstacles are tables(too high) and bases of the chairs (too low to be detected).

5.3 Navigation with Kinect v2

Once we got a working system, we tried to add a new sensor in our system: the Microsoft Kinect v2. The first step was to choose which data message would be of interest for our application, because the *iai_kinect2* package publishes multiple topics with different messages. Since we had to measure distances, in order to find obstacles in the navigation, we used a topic in which is published a depth image (/kinect2_head/depth/image) and a topic in which is published a point cloud(/kinect2_head/depth_ir/points).

Figure 5.7: A depth map from the Kinect v2 sensor.

We firstly tried to use the depth image, but we discovered that the images taken from the Kinect v2 have a lot of noise, specially in the corners (see Figure 5.7). We searched for state of the art algorithms to remove noise from those images, but we did not find anything useful. Most of the tips that were given were to use the disparity image in order to recreate the 3D image. So we decided to use directly the point cloud that the Kinect v2 gives us after a filtration phase.
In Figure 5.8 we can see a point cloud visualized in RViz, using intensity to color points. As we can see, there are a lot of flying pixels, specially near the camera; we have to filter them in order to get a scene in which the object are visible.
The first thing to do was to get the point cloud in a data type that could be handled

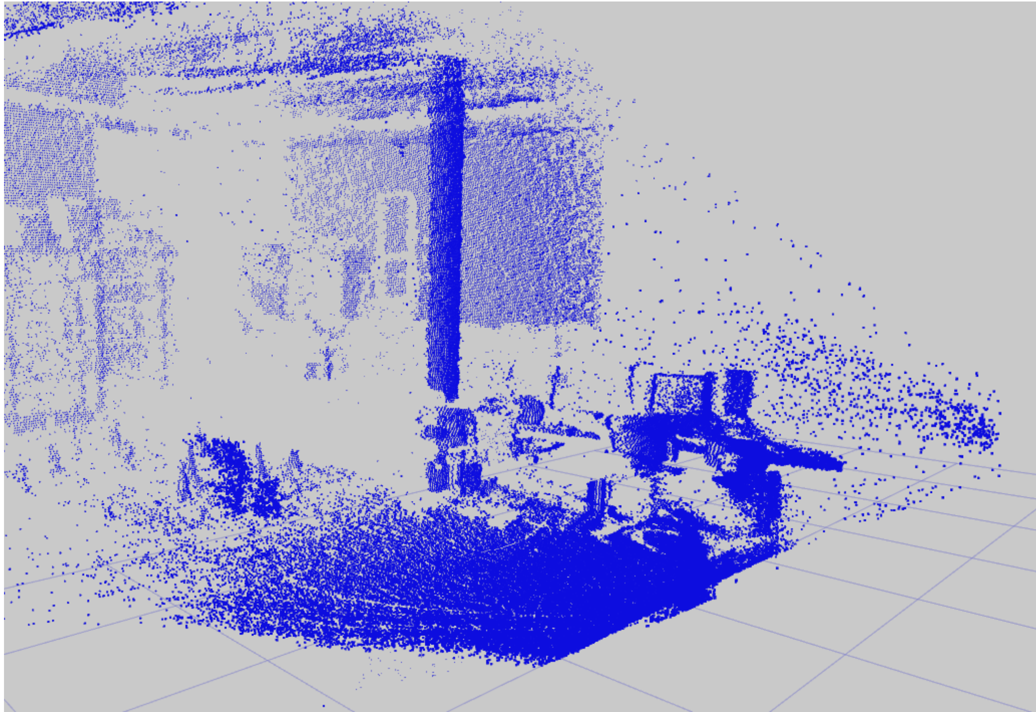


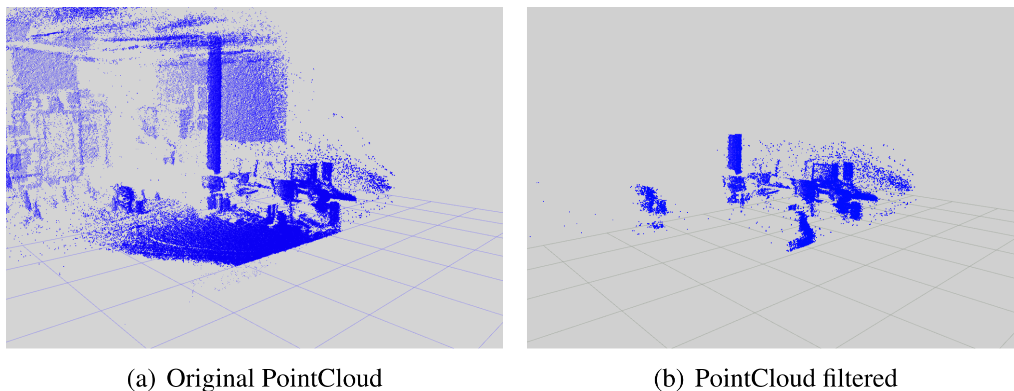
Figure 5.8: Visualization in RViz of a point cloud taken from the Kinect v2 sensor.

efficiently. We used PCL which is well integrated in ROS and provides a method that converts a ROS message to a PCL point cloud: *pcl:: fromROSMsg(sensor_ msgs/ PointCloud2, pcl:: PointCloud<T>).*

Once we acquired the point cloud, the next step was to lighten it, so that the computational time to filter it would be lowered. The time constraint was due to the fact that the robot needed the filtered data in time to avoid the obstacles that the laser could not detect.

Firstly, we decided to exclude parts of the point cloud where there could not be obstacles that could harm the robot, that is all the points above a certain threshold. To be lighter on computation, we decided not to use any algorithm to remove the ground plane, but to put a threshold to the minimum height for the point cloud. We also decided to cut in the depth direction, using the specification range of the Kinect v2. In order to do that, we used a PCL filter called Conditional Removal. This filter removes from the cloud all those points that do not satisfy one or more given conditions that are specified by the user. We imposed conditions in every direction, so that we could limit the point cloud to specific bounds. At first, we thought that with a threshold to reject points with a low depth we could remove most of the noise in the cloud, but this way we also removed obstacles near the robot and this was not a part of the intended behavior. So we removed the depth

constraint, keeping only an upper bound.
Besides filtering the point cloud with those thresholds, we used a VoxelGrid filter to downsample the number of points. With this approach, all of the points in a voxel would be replaced with their centroid. A custom approach of selecting only a point every x points has also been tried, but we did not find a significant improvement in the performance and the remaining point cloud did not represent the surfaces in an accurate way, unlike the VoxelGrid filter. The result of these two firsts phases of filtering can be seen in Figure 5.9.



(a) Original PointCloud (b) PointCloud filtered

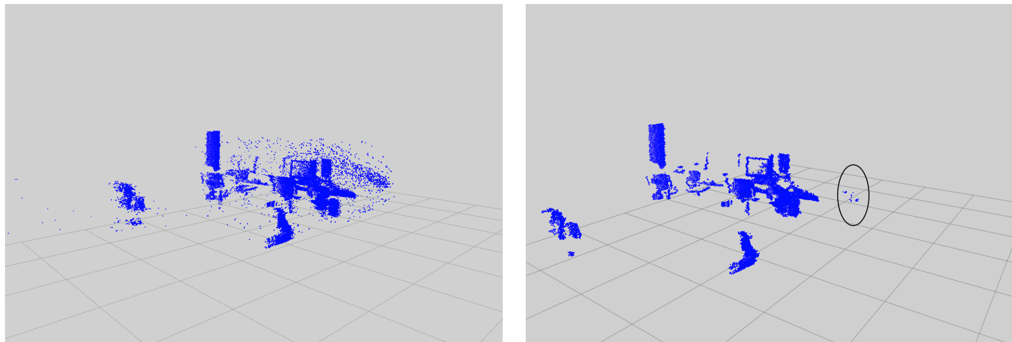
Figure 5.9: Comparison between the point cloud before and after the Conditional Removal and Voxel Grid filtering.

As we can see in the Figure 5.9, although a lot of points have been removed and the scene keeps only the important objects, there is still a lot of noise. In order to remove that, we relied on another filter, the Statistical Outlier Removal (SOR). This filter uses only two parameters to remove outliers:

- **MeanK** is the number of nearest neighbors to use for mean distance estimation. This can be set with the method pcl:: StatisticalOutlierRemoval <PointT>:: setMeanK(int nr_k).

- **StddevMulThresh** is the standard deviation multiplier for the distance threshold calculation. This can be set with the method pcl:: StatisticalOutlierRemoval <PointT>:: setStddevMulThresh(double stddev_mult).

Using these two parameters, a distance threshold can be calculated with this formula: $meanK + stddev_mult \cdot stddev$. Points with their average distance from their neighbors below this threshold will be classified as inliers, otherwise they will be outliers. In Figure 5.10 we can see the best results achieved after a session of parameters tuning.



(a) PointCloud after Conditional Removal and Voxel Grid

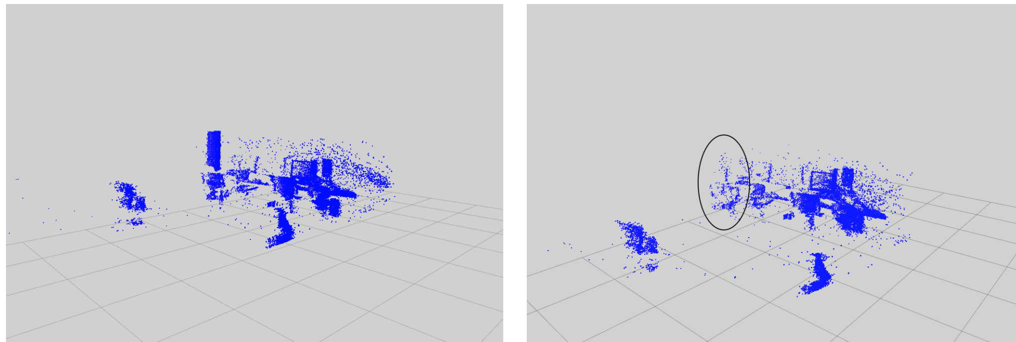
(b) PointCloud filtered with SOR

Figure 5.10: Comparison between the point cloud after the firsts filters and after Statistical Outlier Removal.

Although we obtained good results from filtering with SOR, small clusters of spurious points were still present in the point cloud, like we can see in Figure 5.10(b) on the right of the image. Then we decided to substitute SOR with another filter: Radius Outlier Removal (ROR).
This filter removes all of the points in the cloud that do not have at least some number of neighbors within a certain range. With this algorithm we could define threshold loosely enough to keep the significant objects in the cloud, but strictly enough to remove the clusters of points. In fact, this algorithm allows to set these two parameters to remove outliers:

- **Radius Search** is the sphere radius that will determine which points are neighbors. This can be set through the method pcl:: RadiusOutlierRemoval <PointT>:: setRadiusSearch(double radius).
- **Minimum Neighbors** is the number of neighbors that need to be present in the sphere centered at the considered point. This can be set through the method pcl:: RadiusOutlierRemoval <PointT>:: setMinNeighborsInRadius(int min_pts).

Trying to find the best parameters for ROR, we understood that, in order to remove the clusters of points, we needed to use strict parameters for the filter. We also found out that Radius Outlier Removal left small clusters of noise, no matter the parameters used. Acknowledging these two facts, in Figure 5.11 we can see the result after filtering with ROR.
As we can see, we obtained a point cloud without big clusters of spurious points, but flying pixels are still present in the image. It should also be noted that ROR



(a) PointCloud after Conditional Removal and Voxel Grid

(b) PointCloud filtered with ROR

Figure 5.11: Comparison between the point cloud after the firsts filters and after Radius Outlier Removal.

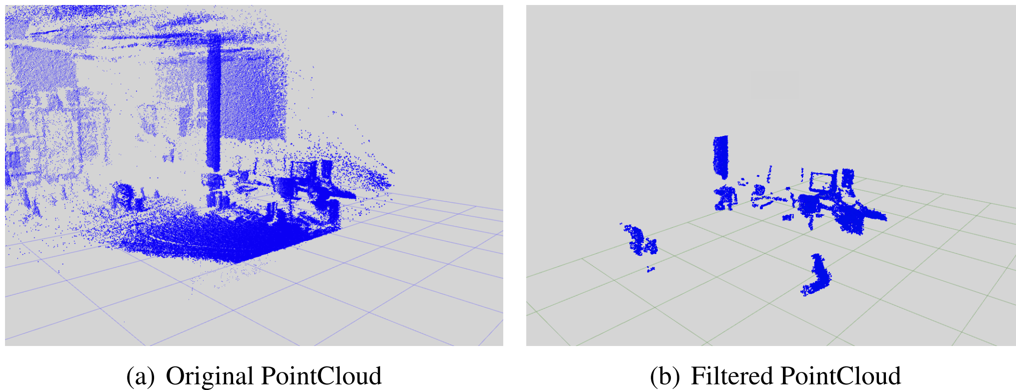
filters a bit too much in the farthest part of the point cloud. We can see in Figure 5.11(b) that the pillar is almost completely removed by the filtering.
In order to get a useful filtered point cloud we then differentiated the way we filter the cloud. Since we needed a cloud without any kind of noise near the robot to have a safe navigation, but we could accept a not perfect filtering far from the robot, we decided to split the point cloud in two, using depth as a threshold. The filter that we used to achieve that is the PassThrough filter. With this tool we could set the axis that we wanted to filter with the method pcl:: PassThrough <PointT>:: setFilterFieldName(const std::string & field_name) and then the range of the point cloud we wanted to keep with the method pcl:: PassThrough <PointT>:: setFilterLimits(const float & limit_min, const float & limit_max). After having extract the near cloud, we could reverse the limits set with the method setFilterLimitsNegative(const bool limit_negative), in this way we could use the filter to extract also the far cloud.
The final filtering pipeline is then:

1. Conditional Removal
2. Voxel Grid
3. PassThrough
4. Radius Outlier Removal:
    - Near Cloud: strong filtering.
    - Far Cloud: no filtering.

5. Statistical Outlier Removal:

    - Near Cloud: filtering with a lower StdDevMultThresh.
    - Far Cloud: filtering with an higher StdDevMultThresh.

The two clouds are finally merged together. In Figure 5.12(b) we can see that we got a point cloud with all the relevant obstacles visible and without noise.



(a) Original PointCloud (b) Filtered PointCloud

Figure 5.12: Comparison between the point cloud before and after the filtering.

The node that filters the point cloud then transform it back into a ROS message, using the PCL method: pcl:: toROSMsg (pcl::PointCloud<T>, sensor_msgs /PointCloud2), and publishes it to a topic. The topic publishes at an average rate of 8.8Hz and, considering that the Kinect v2 publishes on the input topic with a rate of 10Hz, the filtering operation is quick enough for our purposes.
The next important step was to get the obstacles in a way that we could represent them in an Occupancy Grid. In order to do that, the package **pointcloud_ to_ laserscan** has been used. Like the name says, this package requires a point cloud as an input and returns a laser scan. The package uses nodelets, that are efficient nodes, to do this work. Different parameters can be set, like the minimum and maximum height (of the point cloud) to scan to create the laser scan, or the laser scan parameters, angle_min, angle_max, angle_increment, scan_time, range_min and range_max (for these parameters we used the Hokuyo's ones). Another parameter that can be set is the concurrency level, that affects the number of point clouds queued for processing and the number of threads used. Thanks to how the package was done (nodelets), with the help of multithreading, this node publishes the laser scans keeping the same rate as the input topic.
Initially we did not get the expected results. In RViz a line, if any, was displayed in the vertical direction. We found out that the reason for this strange behavior was attributable to a frame problem. In Figure 5.13 we can see that the frame

used by the Kinect v2 and the frame used by the Hokuyo laser scan have not the same reference system and then all the calculations are wrong.

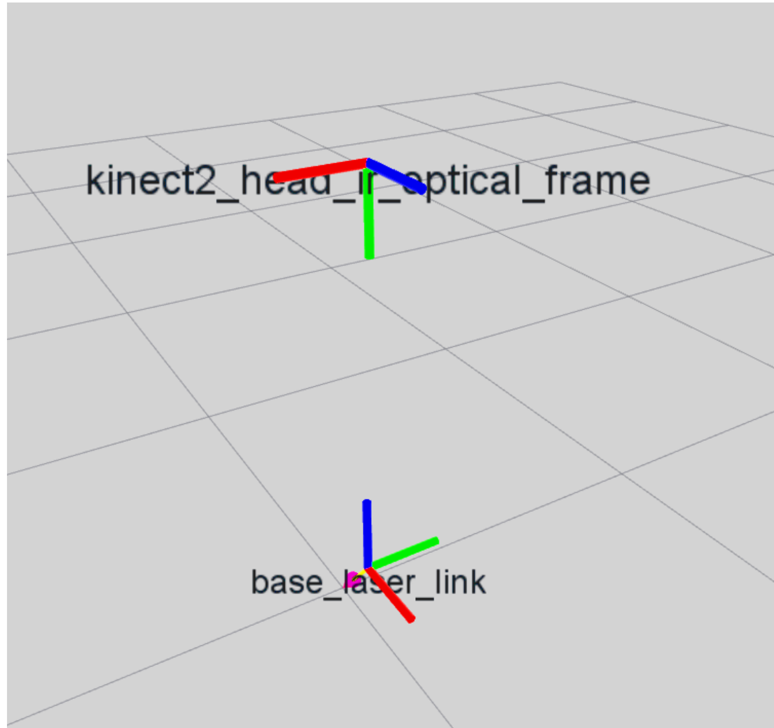


Figure 5.13: The Kinect v2 frame and the base frame have not the same reference system.

To overcome this problem, we used the tf2 package to acquire the transform between the kinect2_head_ir _optical_frame and the base_laser_link frames and, after the filtering, we applied the transform to the ROS message using a tf method. The package then transforms the point cloud into a laser scan correctly, like we can see in Figure 5.14.
Finally, we used the implemented Laser class to transform the laser scan into an occupancy grid.
Then we tried to make the robot navigate using only the Kinect v2 sensor. The results were good: the Kinect found obstacles that the laser could not find, like chairs and tables, but the navigation was not as stable as it was with the laser. In a small hallway, in which laser-based navigation was stable, navigation with the Kinect was not, because of the limited field of view of the sensor. The robot started to swing like when the strength parameter was not dimensioned correctly. In order to overcome these problems we started working on merging data from different sensors.

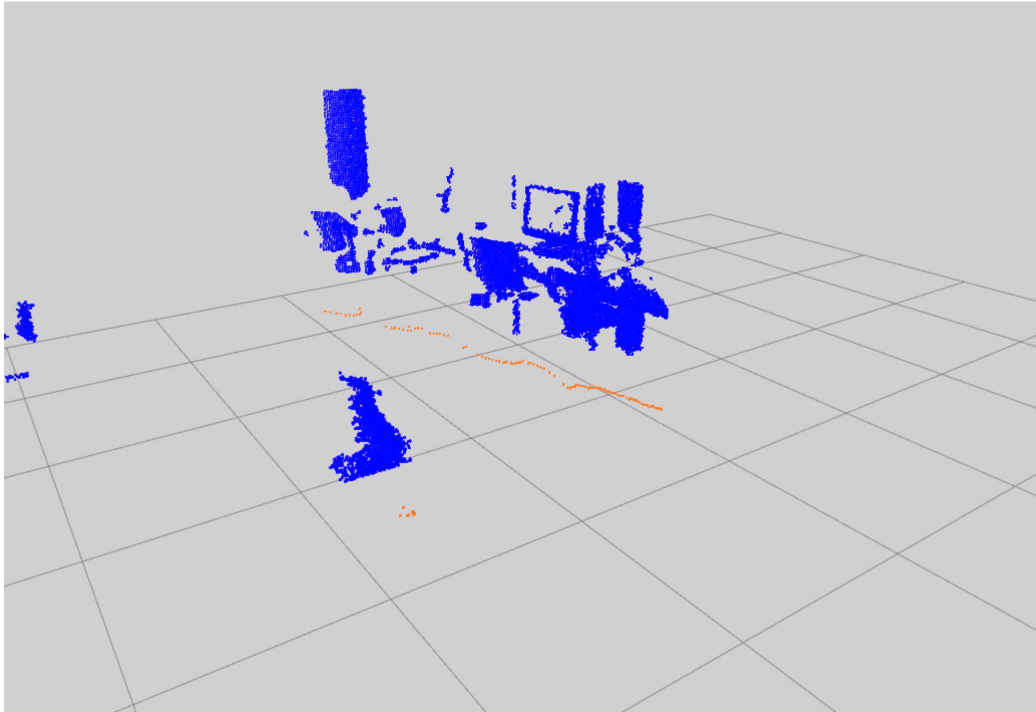
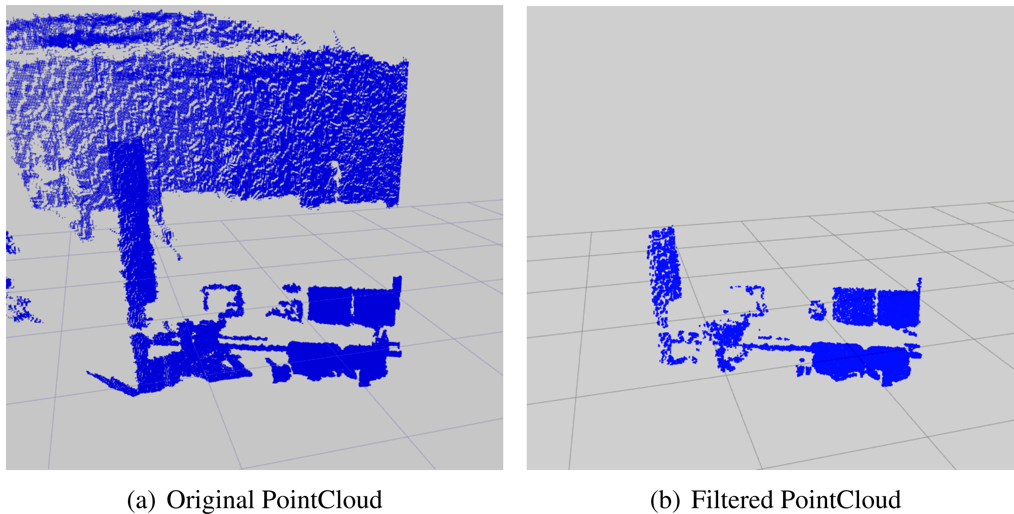


Figure 5.14: The laser scan resulting from the filtered cloud.

Before that, we also tried our algorithm on the Microsoft Kinect sensor (first version). We acquired the point cloud from the topic */camera/depth/points* and we kept the same filtering as the Kinect v2. As we can see in Figure 5.15, the first version of the Kinect gives a point cloud with little, if any, noise, but it has a limited field of view, in addiction to the other problems that we discussed in subsection 2.8. This is what makes the Kinect v2 more suitable to this application. However, the Kinect could be used together with the Kinect v2 sensor, in order to cover the blind spots that both have, also considering that the Kinect does not need to use an USB 3.0 and its publishing rate is 30Hz (that become 29Hz once filtered).

5.4 Occupancy Grids merging

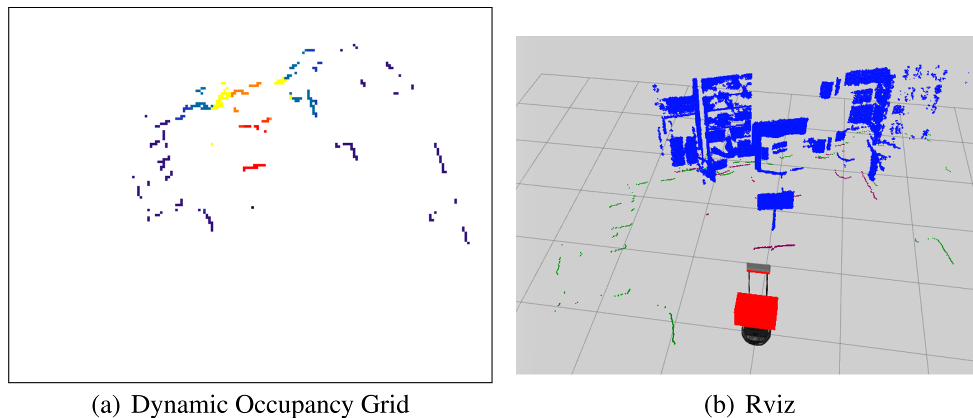
The next step in designing our system was to deal with the merging problem. With the use of occupancy grids, the merging of the messages was straightforward.
Each message was distinguished from another using the frame_id as the key. If a message with an already acquired frame_id was received, it would be discarded. Other way, if a message with a new frame_id arrived, the obstacles in the new map would be summed to the obstacle of the saved map by adding the value of



(a) Original PointCloud (b) Filtered PointCloud

Figure 5.15: Comparison between the point cloud before and after the filtering for the Kinect sensor.

each cell to the value of the corresponding cell of the saved map.
In the first implementation of the merging algorithm we used a binary occupancy grid, where 0 indicated the absence of an obstacle and 100 its presence. In case of the same obstacle being detected by two or more sensors, the value would not change and would stay constant to 100 (certain probability).
When the number of new frame_ids acquired matched the number_of_sensors parameter, defined by the user in a yaml file, the merged map would be transformed into a dynamic occupancy grid and then sent through a *dyn_msgs/ DynamicOccupancyGrid* message to the topic (/merged_ occupancy_ grid) that would be used by the navigation node and the draw map node.
A problem associated with this approach is that the faster sensor has to wait for the slower one. Indeed the map of the faster sensor will be saved when the first message arrives and then all the other messages from this sensor will be discarded until the first message of the slower sensor will be available. This way the scene could change and the merged map would not be consistent with reality, making the navigation unstable. To overcome this problem we saved the new map in a vector with its sensor frame_id associated and then we updated that map at the arrival of every new message. When the vector size matched the number_of_sensor parameter, the maps would be merged together and then sent, as before.
Another problem with this algorithm is that the number_of_sensors parameter has to be set from a yaml file and it is static. This way, if a node crashes the robot cannot continue to navigate through the environment, even though the other sensors are active. In order to overcome this problem, a change in the system



(a) Dynamic Occupancy Grid (b) Rviz

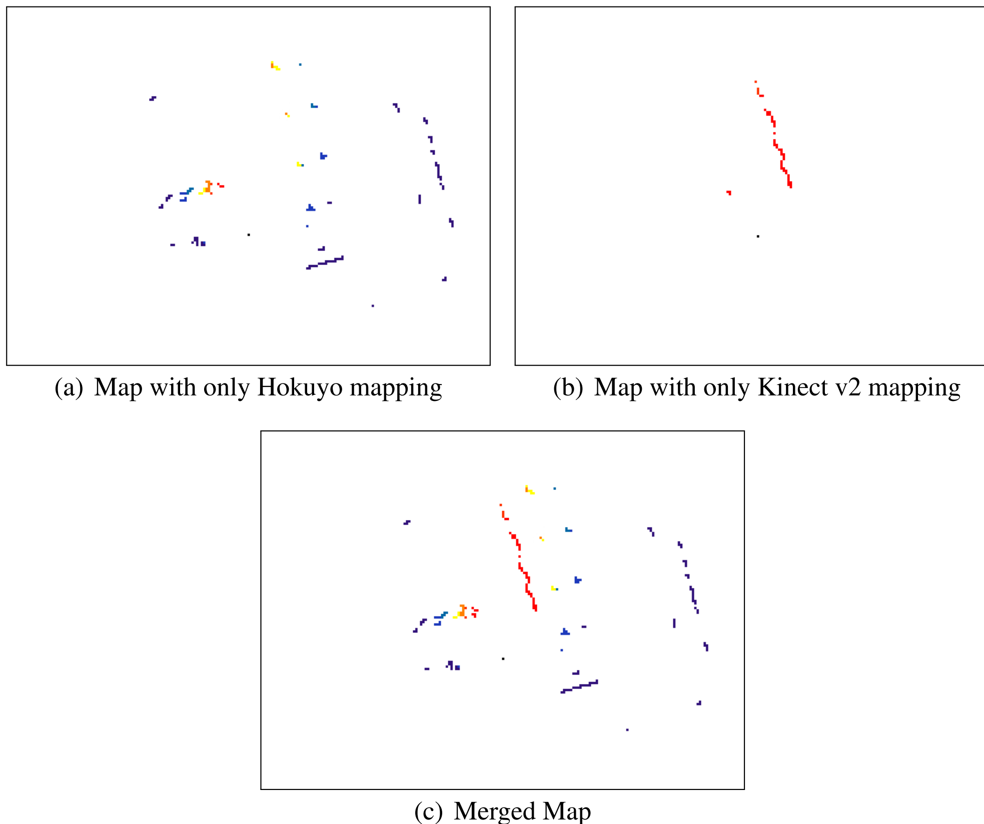
Figure 5.16: Merged map from different sensors. Comparison between this work Dynamic Occupancy Grid and Rviz visualization

design has been made. Every sensor node has been subscribed to a topic: */connected_sensors*. The merging node, instead, has been made the publisher for that topic. This way, through the ROS method *getNumSubscribers*, the number of active sensor will be updated every time an occupancy grid is received and the number_of_sensors variable can be discarded. The publisher also has to publish a message in the topic to update the number of subscribers, so we made it publish an *std_msgs/Empty* (empty to lower network load) and we made a void callback for every sensor, to speed up the updating process. In Figure 5.17 we can see the map visualized when the Hokuyo laser scan is active (Figure 5.17(a)), when the Kinect v2 is active (Figure 5.17(b)) and when both of them are active (Figure 5.17(c)).

During the test phase of the system, another problem appeared: on the same objects, e.g. a wall, the two laser flows did not match (see Figure 5.18). This was still true when the robot faced the wall in the other direction. In particular the Kinect v2 scan was shifted to the left in both cases with respect to the Hokuyo laser scan and it seemed shifted by a fixed distance, meaning that the transform between the Kinect and the Laser frame was wrong. Since we could not know beforehand how much the measurements were wrong, in order to change them in the TurtleBot package, we created a node that could calculate the shift between the two laser flows.

In the design of the algorithm we have taken into account that it could not be possible for the robot to calculate the static shift between the two frames during the navigation, because there could not be matching obstacles. Then we designed this algorithm to be run offline.

The first thing to do was to take some training images of a known common object. We created our dynamic map with one sensor at a time, from different angles, and



(a) Map with only Hokuyo mapping (b) Map with only Kinect v2 mapping

(c) Merged Map

Figure 5.17: Comparison between the Hokuyo, the Kinect v2 and the merged Occupancy Grids

we saved the images. In order to get significative images, we limited the Hokuyo laser scan to the same field of view of the Kinect v2, in this way all the useless obstacles will be avoided in the matching algorithm.

Then we loaded those images two by two (laser and Kinect shooting the same scene) and we scanned the image to find those pixels that were not white. When such a pixel was found, a correspondent point in a point cloud would be created. We also had to pay attention to the frames, indeed the Z axis in OpenCV corresponds to the Y axis in PCL. Once we created our two point clouds, we could use the ICP algorithm to calculate the transformation between the two. We tuned the algorithm parameters so that we could obtain a fitness score as low as possible, meaning that the transformation found was the most accurate possible. We associated a weight of 1 - fitness_score to each match, so that we could do a weighted average over the ten matches. Matches with fitness_score > 1 have been excluded from the calculation because the transformation found was not accurate enough.

Once we obtained the transformations and their weights, we averaged them. Every transformation was composed by two components, translation and rotation. The average between translations was straightforward, but computing the average between rotations was not an easy task since the rotation was saved as a quaternion.
Quaternions encode only orthogonal transformations, but the average of several orthogonal transformations is not, in general, orthogonal, so it is not representable by a quaternion. But if the transformation are near in the 3d space, like in this case, averaging is possible [53]. Let

$$Q = [a_1 q_1 \; a_2 q_2 \cdots a_n q_n] \tag{5.4.1}$$

be a 4xn matrix where a_i is the weight of the i-th quaternion and q_i is the i-th quaternion to be averaged. The normalized eigenvector corresponding to the largest eigenvalue of QQ^T is the weighted average. Since QQ^T is self-adjoint and at least positive semidefinite, fast and robust methods to solve the eigenproblem are available.
We used the *Jacobi SVD (Singular Value Decomposition)* algorithm from the Eigen library. This algorithm calculates a factorization of a real or complex matrix. The resulting factorization is in the form $U\Sigma V^T$ where the column of U are a set of orthonormal eigenvectors of $Q * Q^T$. In particular, the first column is the eigenvector associated with the largest eigenvalue, so it is the weighted average that we needed.
Once we had the two averages, we reconverted the transformation to the openCV

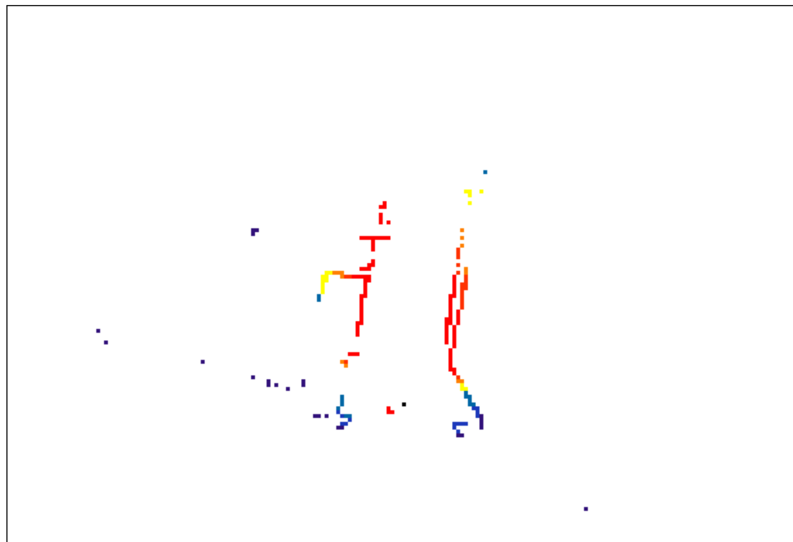


Figure 5.18: Laser and Kinect v2 laser scans do not merge correctly

frame and we saved them in a configuration file. Then, when we acquired the transform from Kinect v2 and Hokuyo, we applied it (if no transform was available, the identity transform would be applied). The results were good and as we can see from Figure 5.19, the two laser flows now merge perfectly, making the scene more understandable.

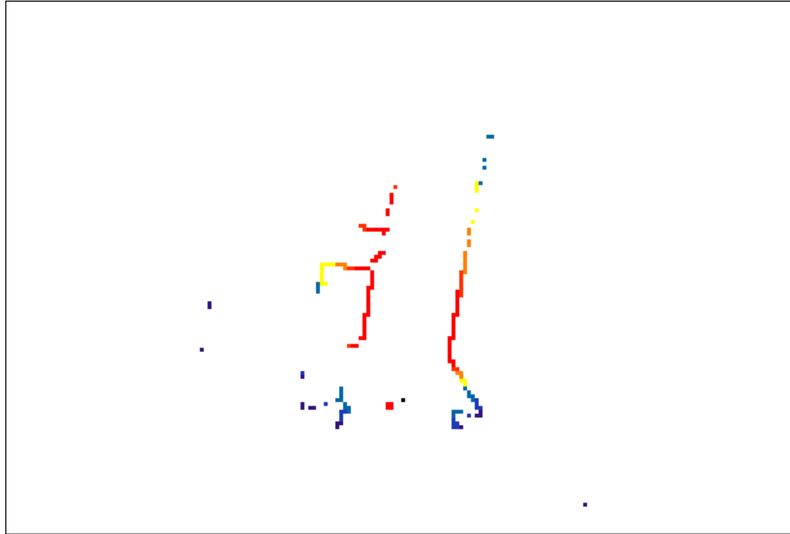


Figure 5.19: Laser and Kinect v2 laser scans merge correctly

The last thing we have done with the occupancy grids merging has been to transform the occupancy grids from binary (the obstacle is in that cell or it is not) to probabilistic grids (the obstacle is in that cell with probability p). To do that, we created a yaml file in which the user can set the weight of the different sensors, using this format:

- publisherX:
  - id: "publisher"
  - ranges: $[r_1, r_2, \cdots, r_n]$
  - costs: $[c_1, c_2, \cdots, c_n]$

Where X must be a number in ascending order (the first sensor must be 0, then 1 and so on), "publisher" is the name of the frame_id of the sensor, r_i is a range, meaning every distance between r_{i-1} and r_i (if $i = 1$, 0 is the lower bound), and c_i is the corresponding weight for the sensor in that range. Every obstacle with distance $> r_n$ will have weight 0.

Having defined the sensor in this way, when the merge node creates the merged map, it calculates the distance of the obstacle and it uses a percentage of the total weight of the active sensors at that distance to give a value for the cell.
This way we can give more weight to the sensors that we know to be more reliable at certain distances, like the Hokuyo is in the firsts 0.50m, where the Kinect v2 can not detect anything.
During the test phase of the algorithm, the robot could move autonomously through the environment without notable problems. The robot could avoid all the visible obstacles, like chairs, tables and humans. It also navigated in a smooth way, without dangerous oscillations or sudden changes of direction.
At this point we decided to change the linear velocity, that up to this point was kept constant. We used Equation 5.4.2 to get a velocity that changed according to the obstacles presence:

$$\begin{aligned} \text{velocity} &= \text{max_velocity} \cdot \cos\left(\sum_i \lambda_i(\phi_i - \phi_h) e^{\frac{(\phi_i-\phi_h)^2}{2\sigma_i^2}}\right) \\ &= \text{max_velocity} \cdot \cos(\dot{\phi_h}) \end{aligned} \tag{5.4.2}$$

This way, when the new direction was almost the same as the current heading of the robot, it could go at maximum speed, instead if it had to turn 90°, the linear velocity was zero. We tested the system with different values for *max_velocity* (e.g. values in a range of 0.08-0.12 m/s) and the system gave the best results with max_velocity = 0.10 m/s, where the robot could avoid all the visible obstacles keeping a fast velocity. The only problem that occurred is that it could not see obstacles that were lower than the height of the Hokuyo, if they were near (otherwise the Kinect v2 could have detected them). Obstacles in its blind spot, that is the space not scanned between the laser scan and the lowest angle of the Kinect v2 field of view, were invisible as well.

5.5 Navigation with Bumpers

Another contribution of this work is the introduction of the bumper sensors feedback in the algorithm loop. The bumpers, located at the front and/or back of the robot, are meant to absorb the impact of a collision. The Kobuki base had three bumper sensors placed on the front side of the robot, in particular at the left, at the front and at the right (see Figure 5.20). The bumpers could be triggered by a collision, or by a wheel drop sensor for fall detection.
There were two topics, published by the TurtleBot package that allowed us to get bumpers information: */mobile_base/events/bumpers* and */mobile_ base/ sensors/*

Figure 5.20: Bumpers on the Kobuki base.

bumper_ pointcloud. On the former were published *kubuki_ msgs/ BumperEvents* messages, that had this specifications:

- uint8 **bumper** defined which bumper has changed its state (LEFT, CENTER, RIGHT).
- uint8 **state** defined what happened to the bumper, its current state (RELEASED, PRESSED).

On the latter, were published *sensor_ msgs/ PointCloud2* messages that contained a point cloud with a point representation of the direction in which the corresponding bumper had been pressed, or no points if the bumper was released.
Both topics were published only when the event occurred and not continuously, so the bumpers worked in a different way with respect to the other sensors. Firstly, we decided which topic to use. We chose to use the */mobile_ base/ sensors/ bumper_ pointcloud* topic because we wanted to separate the type of robot used as much as possible from our implementation and the first topic used messages typical of a Kobuki base.
Unlike the Kinect v2 topic, we only had to scan through the message to get the information needed, without filtering. We created the corresponding occupancy grid map and sent it to the merging node, as shown in Figure 2.2. We modified the Merge_Map node so that we could use this sensor without having to block the rest of the system. Indeed, if we used the node like it was designed, the robot would go straight until it would hit an obstacle and only at this point the map would be visualized, because it would wait for all the sensors to acquire their data before publishing. So, to overcome the fact that the topic would be published

only at the time of the event, we made an exception to include the bumpers without counting them as an active sensor. In this way we were able to keep our working autonomous navigation, but we could also use this event sensor.
During the test phase, we noticed that the single point published by the bumpers in the direction of the obstacle hit was not enough to make the robot turn to another direction. This was due to the fact that we had all of the other sensors obstacles in the map, so a single point could not have enough strength to make the robot turn. Hence we modified the occupancy grid map so that the obstacle could have more strength, adding 8 more points to the adjacent cells of the point created by the bumper. This way the robot turned in the right way after hitting the obstacle, that in our case was the underside of a chair. A problem that occurred with this implementation was that, even though the new direction was correct, the robot still tried to go ahead, taking the chair with it. To avoid this behavior, we used the *near_obstacle* variable of the *dyn_msgs/DynamicOccupancyGrid* message so the robot could stop moving straight and could only turn on the spot. In this way the obstacles that were hit were overcome, and so the low obstacle problem was solved.

5.6 Semi-Autonomous Navigation

Semi-autonomous navigation is a type of navigation where the robot can go through the environment autonomously, using only its sensors. The user can prompt it to move in a direction that it will follow. In order to achieve this result, we used the keyboard to guide the robot.
The node could not subscribe to a topic that could relay to the keyboard strokes, so we made a listener for the user. The first thing to do was to get the keystrokes. To do that, we overrode the behavior of the keyboard, so we could get the characters from the terminal without visualizing them in output. A string message, instead of an ASCII value, would be displayed after pressing a key. In its first implementation, the Keyboard node could take in input only the arrow keys and the **q** character which would shutdown the node.
The next step was to transform the acquired characters into an occupancy grid. We created the map using the unknown value of an occupancy grid, -1, to define an attractor (see subsection 3.1). This way we could use the designed algorithm without doing other adjustments except adding a new condition to the Merge_Map node. The new attractor, for left and right arrows, was not created on the side of the robot but shifted ahead, so that it could be weighted more in the summation. As with bumpers, we added attractors to the adjacent cells to obtain a stronger attractor. We also introduced a publishing time for the messages, that is the period of time during which the node must continuously publish the messages in order to

make the robot turn correctly.

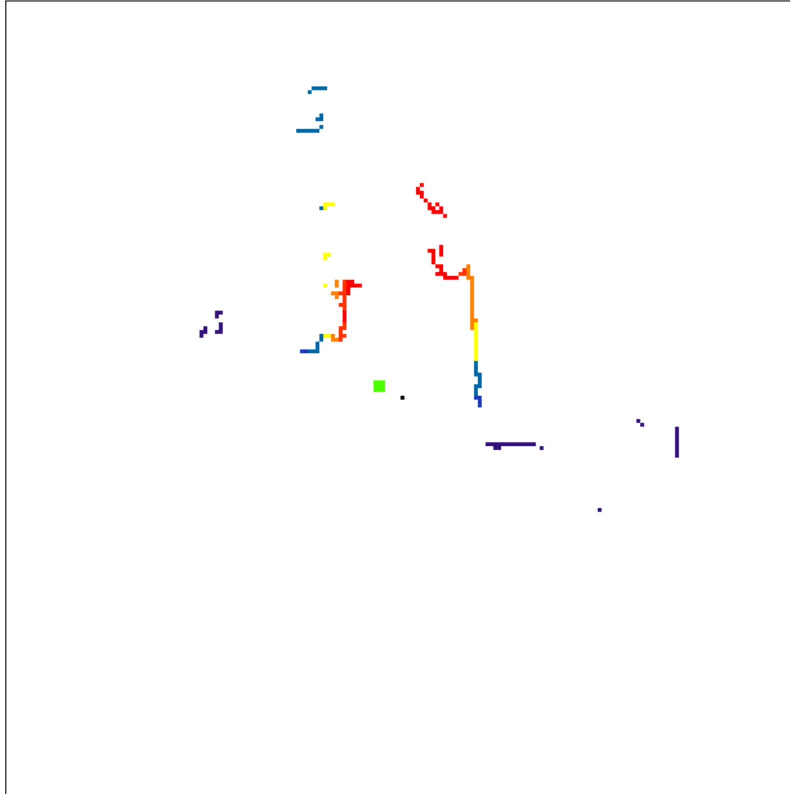


Figure 5.21: The dynamic map after pressing the left arrow, with the attractors in green.

In Figure 5.21 we can see the resulting map, with the attractors colored in green. We tested the algorithm and adjusted the publishing time so that the robot could turn 90° when a key was pressed. One problem we had was that we had to press the key twice to make the robot actually turn, because it seemed like the attractors had no strength.
After getting a working system, we improved it by allowing the user to give commands from another computer. To do that, we created a program that could send keyboard strokes using UDP. We were not interested in using a connection based protocol, instead we wanted the message to arrive as soon as possible. In case the input was not received by the robot, the user could always resend it, so UDP was the natural choice for this kind of transmission. The only required inputs were the IP and port of the computer connected to the robot. For the receiving part, every key received at the right port could be used as if the input was taken from the keyboard. The only parameter to set was a connection type variable in a yaml file, that could be *LOCAL* or *REMOTE*.

We tried the algorithm giving the key strokes from another computer and it worked as well as the local input case, without notable delays.

5.7 Deterministic Finite Automaton

With the introduction of the keyboard and the bumper sensors, we noticed that we had to add exceptions to the way the Merge_Map node worked. This was not what we wanted to do, because the idea was to add new sensors by making them publish an occupancy_grid to the */occupancy_grids* topic, without any other modifications.
To overcome this problem, we thought of modifying the system in order to use the keyboard and the bumpers as the events they are, and not as sensors. What we wanted to achieve was to keep a system like the one already designed, where the events could publish an occupancy grid on the right topic and the Merge_Map node could use the event information to give a priority command to the robot.
The first thing we did was choosing a convention for the occupancy grids, so we distinguished between frame_ids by renaming them respectively */sensor/name* for sensors and */event/name* for events. Thus, the merging node could receive in the same topic messages from both events and sensors and then it could put them together to get the new map. Using this simple solution we had some problems as the robot seemed stuck after a keyboard strokes, as when we added the exception to the node. Then another keyboard strokes was needed to make the robot turn. Besides, we did not achieved the priority command objective yet.
To fix this problem and achieve the desired behavior, a major modification to the system was needed. The robot was modified to be interpretable as a Deterministic Finite Automaton (DFA) [54], so we could give it commands more easily and define different behaviors as needed.
These are the states that have been created for the robot:

- **Ready**: the system is ready to navigate and the user has some seconds to stop the robot before it starts moving.
- **Running**: the robot is navigating through the environment.
- **Paused**: the system has been paused by the user. The robot stops moving, but it keeps its sensors active. This state can be used when the robot reaches a goal, but it is expected it will have to move again.
- **Stopped**: the system has been stopped by the user. The robot stops moving and it shuts down its sensors. This state can be used when the robot reaches a goal and remaining power is a concern.

- **Manual**: the user has given a command to the robot. This can be used to change the robot status or to make the robot turn another way.
- **Collided**: the robot has collided with something, so some countermeasures are needed.

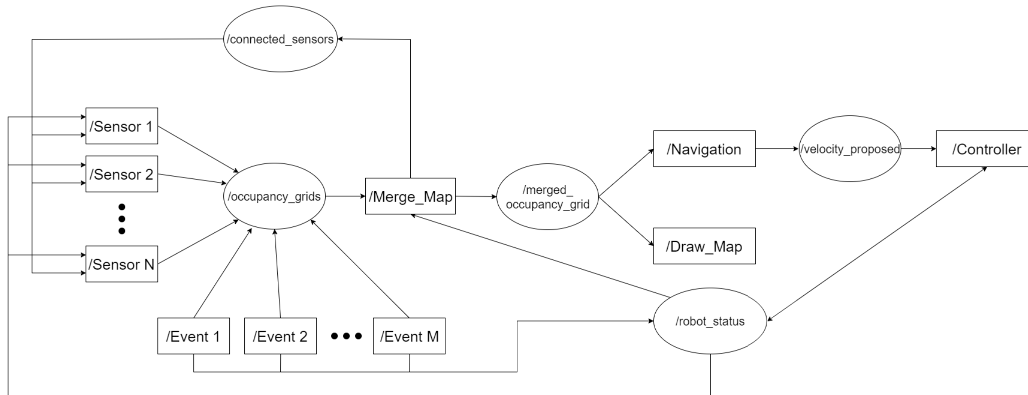


Figure 5.22: The system after the transformation into Deterministic Finite Automaton

The entire system with the state implementation is shown in Figure 5.22. The */robot_status* topic can be published only by an event or by the Controller node, through a *std_msgs/Int32* message (this way more statuses can be implemented in future). In particular, the Keyboard node has been changed so that the user could give more hints to the robot: *r* changes the state to **ready**; *p* changes the state to **pause**; *q* changes the state to **stopped**. Before changing the status of the robot into the correct one, the Keyboard node changes it to **manual**, so that the user, or someone else that is checking the robot through the monitor, can be aware that the state of the system has changed deliberately and not by some internal errors. If the user prompted an arrow key to the robot the status changes to **manual** and, after some seconds, during which the robot will turn, the node will change the robot status to **running**.
The other event that can change the status of the robot in our system is a collision, detected by the bumpers. So, before sending an occupancy grid to the topic, the Bumpers node changes the status to **collided** and, after publishing the messages for an adequate period of time, it changes it back to **ready**.
The Merge_Map node has been modified so that, when the status changes, all the saved maps are dropped and the new occupancy grids that will be saved depends on the frame_id. In particular, if the status is **manual** or **collided**, only messages from events will be saved, while in any other case but **stopped**, messages from sensors will be saved. This way, when an arrow key is pressed, only an attractor

will be put in the resulting map, without all the other obstacles. This is not dangerous for the robot, because the robot will rotate on the spot and no obstacles should be present.
The last node that was modified with the state introduction, is the Controller node, which takes the velocity and uses it to move the robot. This node has obtained a crucial role in the system through this modification; it is responsible of interacting with the robot in different ways, depending on the current state:

- **Ready**: the node will start a five seconds timer before changing the status to **running**.
- **Running/Manual**: the node will publish the linear and angular velocity proposed by the arriving messages.
- **Paused**: the node will keep the robot still.
- **Stopped**: the node will stop the robot and it will shut itself down.
- **Collided**: the node will make the robot reverse and turn with the appropriate angular velocity.

This way we could achieve a different behavior depending on the state, that is something that we could not do before with only the dynamic navigation. Another problem we had was that not all the nodes received the message to update the robot status while this was moving. This was due to the workload of the nodes that continuously received messages from sensors, so a single message on another topic could be lost.
In order to overcome this problem, we had to change the system. We created a node whose only task was to keep the states of the robot updated. Then, changes of the robot status were published to the topic *change_robot_status*. The new node acquired the message and continuously published the updated state to the *robot_status* topic. This way we could be sure that all the subscribed nodes would receive at least one message with the current status of the robot.
In Figure 5.23 we can see a working implementation of the system. We tested it on the real robot and all the messages, those sent using the keyboard and those sent from the bumpers, were correctly received and the status was correctly updated by all the nodes.
Thanks to the implementation of the Deterministic Finite Automaton, we could improve the performance of the system and fix some undesired behaviors. The first thing we did was to average the messages that arrived from the Navigation node. This way, when a sudden change of direction occurred, the robot would turn more gently and get a smoother navigation. The angular velocities were saved in a

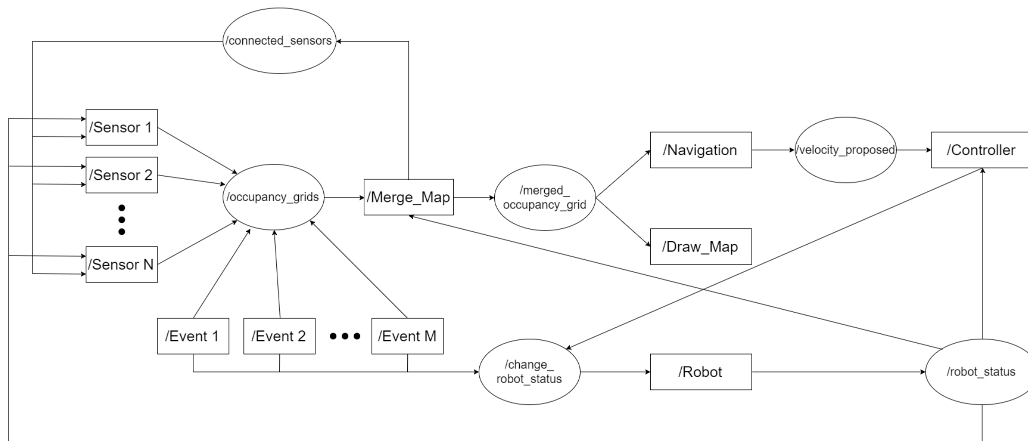


Figure 5.23: Deterministic Finite Automaton working system

double-ended queue, keeping a fixed size, so that when a new message was saved, the oldest one was dropped. We tested this feature in order to tune the maximum number of angular velocities to save. We noticed that as the number of the double-ended queue grew bigger, the robot would turn in a smoother way, but it would also be less reactive. We decided to set the maximum size to two, because the reactiveness of the system was an important factor in a dynamic environment.

Another problem that occurred during the tests was that, if the robot was perfectly perpendicular to a wall, the summation of the obstacles influences would be zero. Then the robot would go straight till the near parameter (in the dyn_ msgs/ DynamicOccupancyGrid message) would stop the robot. To overcome this problem, we made the obstacles on the right have a little more strength than those on the left. The strength increase did not unbalance the system, that kept the same behavior during the navigation, and when the robot was perfectly perpendicular to a wall, it would slightly turn and it could avoid it safely. The resulting influence of the obstacles can be seen in Figure 5.24.

Then we fixed a problem that occurred during the merging of the occupancy grids of the Hokuyo and the Kinect v2. After a rotation larger than a certain threshold, the Kinect v2 scan would be a little late with respect to the Hokuyo one, even if we tried to filter the point cloud as quickly as possible. The resulting occupancy grid would make the robot turn more than necessary, so we changed the system, so that, after a rotation bigger than 30° (after parameters tuning), the robot status would be set to **paused** for a second. This way the slower topics could synchronize with the current view of the robot.

This solution was not general enough for the purpose of our application, so, once again, we modified our system. The main problem was that the synchronization of the messages was left to the processing speed and to the unitary queue of each

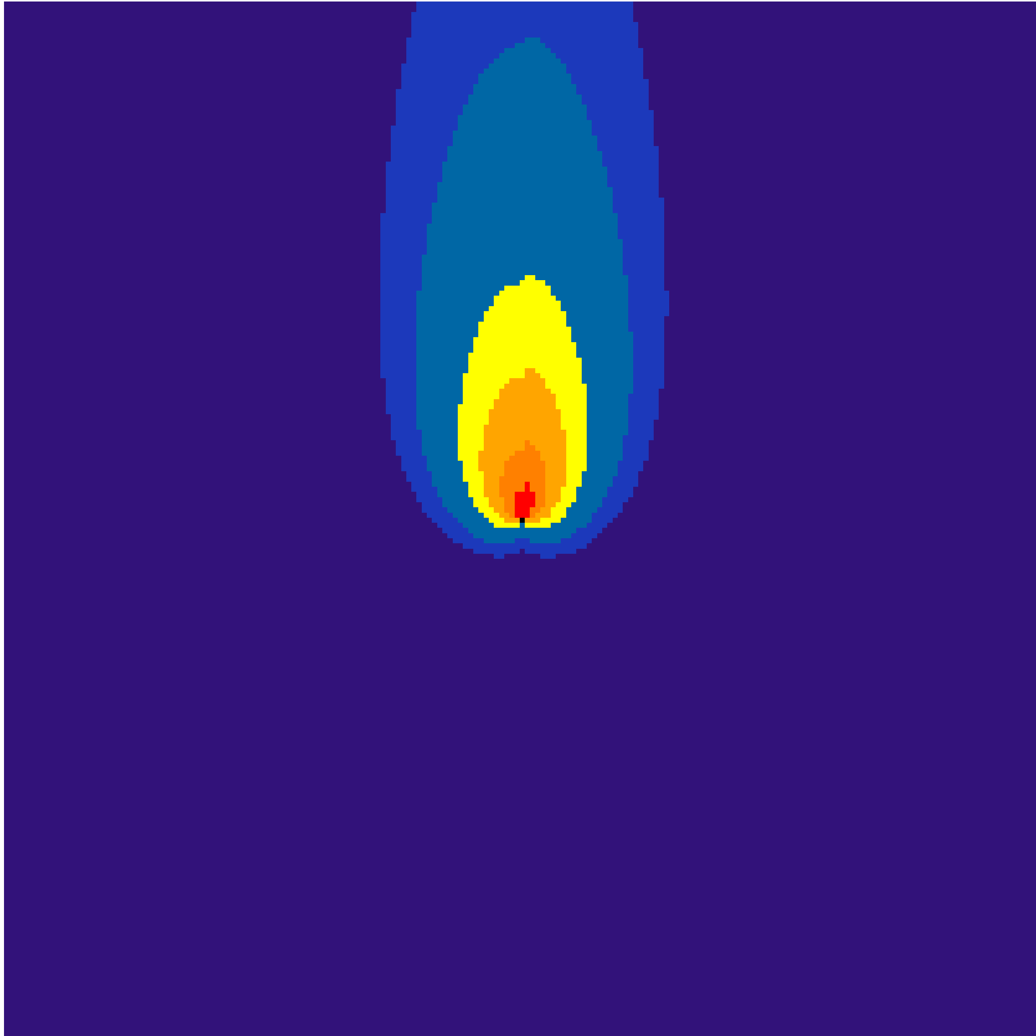


Figure 5.24: The final dynamic map

sensor topic (so that ROS could drop older messages). In addition, the robot had slow speed, so the maps could seem as if they were synchronized, but they were not. This could be seen, as we said, when a rapid rotation occurred. In this case the slower message (Kinect) were not synchronized to the faster one (Hokuyo) and the merged map was not consistent.
The first thing we tried to do was to use the ROS package **message_filter** that took in messages and output them at a later time, based on the conditions that filter needs to meet. In particular, the time synchronizer filter outputs messages with the same timestamp, if the exact time policy is used, or near in time timestamp, if the approximate time policy is used. The one big problem with the use of this package is that the messages must be published in different topics and this was against the

design principle of our algorithm, where the *occupancy_grid* topic should have been the same for all the connected sensors. So we could not use this package in our application.
The next, and last, thing that we tried, was to use the odometry of the robot in order to get consistent merged map. The idea behind this modification was to save the position where the robot was when the map was acquired, alongside with the corresponding occupancy grid. Then, when the Merge node would merge all the occupancy grids, it would calculated the transformation between the robot current position and the position where the occupancy grid was acquired. Then the transformation would be applied to the old map and the resulting roto-translated version could be merged with the other roto-translated occupancy grids.
In order to do that we modified the system so that each sensor/event could send a *dyn_msgs/OccupancyGridWithPose* message, that had this definition:

- nav_msgs/OccupancyGrid **grid** is the occupancy grid that was published before this modification.

- geometry_msgs/Pose **pose** is the current position and orientation of the robot (with respect to the robot bring up position).

This way each sensor/event needed to have access to the odometry of the robot. Instead of subscribing each node to the */odom* topic, that could be resource demanding for those nodes that already had to do a lot of processing work, we created another node, *Get_Pose*, that provided a ROS service: giving the current odometry to the requesting node (see Figure 5.25). This node subscribed to the */odom* topic and its only duty was to keep the odometry updated.

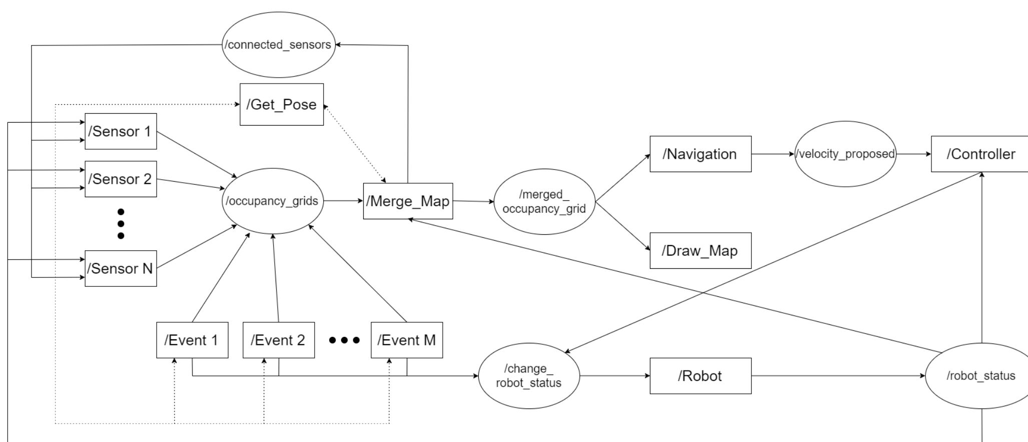


Figure 5.25: The final system of this work

A problem that occurred was that the map frame and the odometry frame were not the same, so we had to transform the roto-translation between two different positions into the right frame before we could use it on the acquired map. Once we overcome this problem, the maps were correctly merged and there was no need to stop the robot after a rotation anymore.
Another benefit of this improvement was that we could change once again the behavior of the bumper node, so that, once the bumpers were pressed, the node would continuously publish the obstacle with the position where it was detected. Thus, after increasing the publication time, the bumped obstacle would stay in the merged occupancy grid for an adequate time, so that the robot could dodge it more safely.

5.8 The Telepresence Apparatus

The last module implemented in this work, is the telepresence module which enables a video stream through laptops far from each other. We used GStreamer 1.0 to do that, because it was easy to use and to implement and in this way we did not have to rely on third party software.
Initially we implemented the laptop web-cam video stream, because one goal of our application was to show, from the robot side, the end-user video of the application. We created the pipeline so that we could take the video from the web-cam device as the source. Then we added a filter, to accept only images of a certain format, a convertor, to transform the video into the right format to send it over the network, an encoder, to transform the video into rtp packets, and finally we sent it to the other laptop via a UDP connection.
From the client side, the UDP socket was the source for the pipeline. Then we continued the pipeline by adding a filter to the arriving packets, to keep only those that had the right format, a decoder and a converter, in order to visualize the images with the sink element. We also added the possibility of save the streamed video, by redirecting the stream to another sink. At this point we tuned the parameters of the various elements of the two pipelines, so that we could get a video of good quality without noticeable delay in local streaming (delay over the network depends on the network quality and workload).
The next step was to stream video from the Kinect v2 to another laptop. The client side did not need to be changed because it worked with every video stream, so we needed to implement only the server side. Initially, we tried to get the Kinect v2 video stream as we did with the web-cam, but this led to a problem: if the Kinect v2 was used in this way, we could not use it in ROS, making the robot navigation less safe. So we changed approach and we relied on ROS to get the stream that we needed. We created a ROS node that was subscribed to the

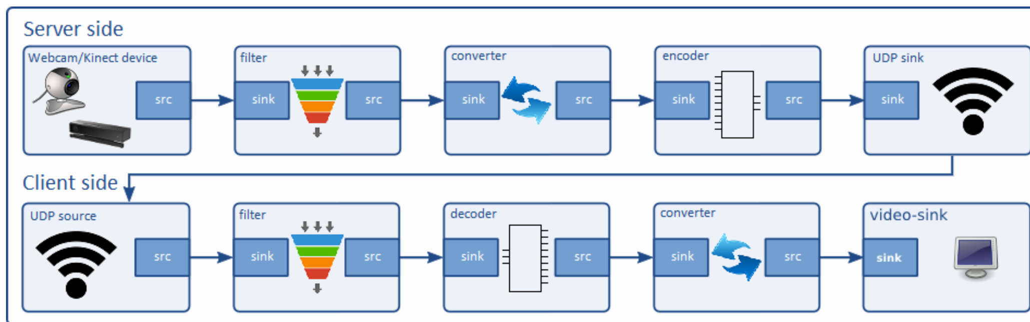


Figure 5.26: The designed GStreamer pipeline used in this work.

Kinect v2 topic */kinect2_head/rgb_highres/image* and we put the arriving images in a buffer that would be the source of our application. The resulting stream was a little slow, because the streamed images had high resolution (1920x1080), so we used images with a lower resolution (960x540), that were provided by the topic */kinect2_head/rgb_lowres/image*. The workload on the network was reduced substantially and the stream was fast enough for the application sake, without having lost too much quality (see Figure 5.27).

Figure 5.27: A scene captured from the Kinect v2 mounted on the top of the robot

The server audio and the client audio were designed in a similar way to the video one. The source for the server was the microphone of the laptop. The sound was then converted, encoded and sent via UDP, the sink of the server pipeline. The source for the client was the UDP stream. The packets were then decoded and amplified, before being played back. With speakers loud enough, or earphones, the audio streaming could be sent between laptops without noticeable delay and with quality high enough for the application purpose.

5.9 Semi-Autonomous Navigation BCI Driven

The last step, after having designed a robot that could navigate in a semi- autonomous way in the environment and after having built our telepresence apparatus, was the integration of the BCI framework.
In this work we used a SMR based BCI that adopts non-invasive EEG acquisition method. The already existing code for SMR-BCI was firstly adapted to permit the link with our system standards. We added to the system keyboard commands to change the robot status, but allowing, at the same time, the possibility to send direction via BCI. The BCI assignment was, in fact, to derive from the EEG activity the corresponding MI task and translate it in a specific direction command, in this work left or right. Likewise we did for the semi-autonomous navigation, we created an UDP socket in order to send the commands to the robot laptop. After the set-up of the communication between the different frameworks and devices, we started the experimental phase.

5.9.1 Experimental design

A male volunteer sat comfortably on a chair, which was positioned in front of a monitor. The subject wore a EEG cap in order to facilitate the EEG electrodes positioning and preventing their movement. A set of 16 active electrodes were attached to the cap. To achieve adequate signal quality, the skin areas that were contacted by the electrodes had to be carefully prepared with a conductive gel. The EEG channels were placed over the sensorimotor cortex: Fz, FC3, FC1, FCz, FC2, FC4, C3, C1, Cz, C2, C4, CP3, CP1, CPz, CP2, and CP4 according to the international 10–20 system with reference on the right ear and ground on AFz (see Figure 5.28). Each electrode was labeled with a letter to identify the lobe and a number to identify hemisphere location. The recording was done using a 16-channel g.USBamp (g.tec medical engineering, Schiedelberg, Austria) system at 512 Hz. The recording system was directly connected to the remote desktop that could send commands to the laptop attached to the robot.

5.9.2 Experimental protocol

Before being able to use a SMR BCI, the user had to go through a number of steps to learn to voluntarily modulate the EEG oscillatory waves by performing MI tasks. Furthermore, the BCI system had to learn what the user-specific patterns were and had to train a specific classifier based on these features.
During the experiment the user was positioned in front of a monitor. The BCI application window is composed of two fundamental element running on black background. A white "+" symbol, called fixation point, positioned at the center.

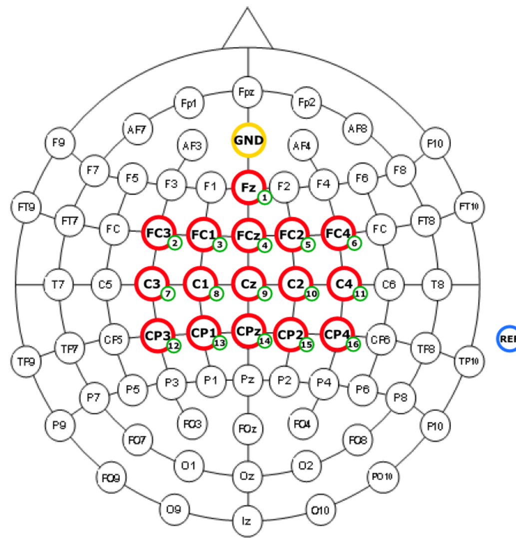


Figure 5.28: Electrodes layout used for the implemented BCI interface.

On the higher part of the window, instead, there were some colored bars, each corresponding to a specific MI task.
Each trial was composed of 4 parts. The first phase, of duration of 2 seconds, called "Fixation", displayed only the "+" sign. The second phase, called "Cue", of duration of about 1 second, suggested the user which motor task he had to accomplish, by transforming the fixation cross into a dyed circle of the same color of the bar corresponding to the MI. Afterwards "Continuous Feedback" (CF) phase was active until one of the bars was filled. The level of the bars represented the status of the BCI system. The "Boom" happened when a bar was finally filled and the corresponding command was sent.
The experiment was divided in three phases:

- **Offline** was the first phase of the experiment. The screen layout presented three different colored bars. Each bar corresponded respectively to right hand, both feet and left hand MI. The user was instructed to perform the MI task from the cue comparison for all the duration of the CF, till the bar was filled, and to relax afterwards. In this part, a positive feedback was given to the user: the bar corresponding to the current task was filled automatically. This kind of feedback was necessary first to data collection for offline calibration and second to motivate the user in the task execution: user rewarding was a important aspect in this type of applications where algorithm performance are inevitably linked with user concentration. The user had to go through three offline session for classifier training. Each session was composed of 45 trials, 15 for each MI task. Then the EEG data

was analyzed and a Gaussian classifier was trained for each pair of MI tasks that the user had rehearsed.

- In the **Online** phase, the Gaussian classifier which showed the highest separability (e.g., right hand versus left hand or right hand versus feet), was employed for feature classification [13]. The user in this experiment showed left hand and both feet MI as the tasks with better performances. Equivalently to the offline phase, the screen layout presented the "+" sign at the center and only the two bars on the top, corresponding to the selected task (see Figure 5.29). This time, during the CF the real feedback was displayed and bars level depended on a decision making algorithm and user performance. Three online sessions were recorded. Each session was composed of 30 trials, 15 for each MI. Online data was then used to refine the offline training classifier.

- **Navigation** was the last phase and the user should be by then in complete control of the BCI system. The layout on the screen was the same as in the online phase, but no cue appeared. During this phase, since the purpose of this phase was the semi-autonomous navigation, the user had to voluntarily deliver a command or he had to balance the tasks so that no command was delivered. Each delivered command was then redirected to the navigation laptop. In this phase it was required additional effort from the the volunteer, because he had to split his attention between the BCI and the Kinect v2 feedback.

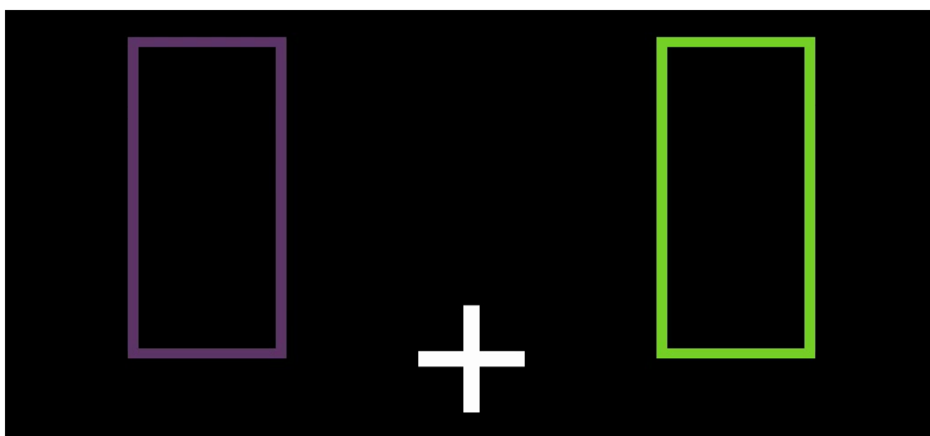


Figure 5.29: The BCI visual feedback during online and navigation phases

5.9.3 Data Analysis

We will see now more in detail the steps between the signals acquisition and the creation of the classifier.
The signals were initially bandpass filtered between 0.1 Hz and 100 Hz and a notch filter was set at the power line frequency of 50 Hz. Each channel was then spatially filtered with a Laplacian derivation before estimating its power spectral density (PSD) in the band 4–48 Hz with 2 Hz resolution over the last second. The PSD was computed every 62.5 ms (i.e., 16 times per second) using the Welch method with five overlapped (25%) Hanning windows of 500 ms. Each PSD matrix was then converted in a feature vector of 16 channels x 23 frequencies. Canonical Vector Analysis (CVA) was finally applied by projecting all data on a canonical space of dimension k-1, where k was the class cardinality, in order to select the subset of the most discriminative features as input of the Gaussian classifier embedded in the BCI.

5.9.4 Decision Making Algorithm

Every 16 times per seconds a new output probability was generated from the classifier. These probabilities were then fed to a decision making algorithm implemented through a probability integration framework. The accumulation framework worked through an exponential smoothing that integrated the new classifier output probability with the old value according to the formula:

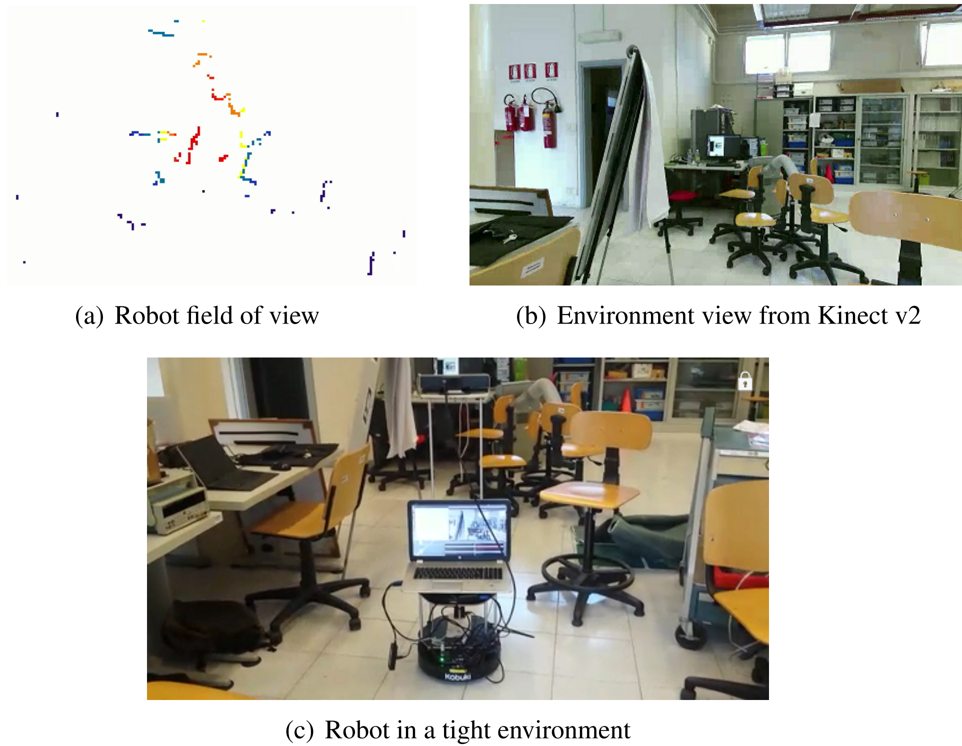
$$D(y_t) = \alpha \cdot D(y_{t-1}) + (1 - \alpha) \cdot p(y_t|xt) \tag{5.9.1}$$

where $D(y_t)$ was the aggregated probability distribution, $D(y_{t-1})$ was the previous aggregated distribution and α the integration parameter. Thus, probabilities were integrated until a class reached a certainty threshold about the user's intent to deliver a command in order, for instance, to change the robot's direction. At that moment the mental command was delivered and the probabilities were reset to a uniform distribution. This decision making strategy allowed to obtain a smooth and predictable feedback, thus helping user training by avoiding confusing and frustrating fluctuations. Another benefit of this accumulation framework was that the user could manifest the intention of not delivering any mental command by continuously balance the mental tasks and never reaching the certainty threshold of command delivery.

Chapter 6

Conclusions

In this work we implemented a semi-autonomous system, based on a potential-field navigation, that will be used on a telepresence robot. This system has been created as a Deterministic Finite Automata, which allows the system to merge data from different sensors and events as well as giving the possibility to obtain specific behaviours from the robot, depending on its status. Moreover, the DFA ensures the scalability of the system. Indeed, new sensors can be added with no modifications.
By testing the system on different robots, we tuned a set of parameters that could also be used to make a robot navigate in tight environments (see Figure 6.1(c)). During the tests the robot could navigate safely in the environment without collisions with obstacles that were in its field of view. The introduction of single-shot sensors as the bumpers in the algorithm control loop ensure the robot to avoid also low obstacles by a trial-and-error approach. It is important that the robot can navigate safely in the environment, so that the application end-user can only focus on giving high level commands to the robot, such as to make it rotate at crossroads.
Even though the TurtleBot was used for most of the project, the algorithm has also been tested with the Pioneer 3-AT, using the SICK LMS-100 laser scan as the only sensor (see Figure 6.2). The only parameters that needed to be changed were robot intrinsic parameter, as its diameter and the name of the topics were its sensors published. The final results were a smooth and collision-free navigation, as with the TurtleBot, therefore demonstrating the portability of the system.
The system was finally tested with the BCI framework, to prove that the system could also be used by people with severe disabilities (even though the volunteer was healthy), thus achieving the final objective of the project. The subject, after some training sessions with the BCI, was able to make the robot turn using only his will (see Figure 6.3). Through the user-machine interaction, we were able to make the robot reach areas that it could not be able to reach by using only its sensor and the autonomous navigation algorithm.



(a) Robot field of view

(b) Environment view from Kinect v2

(c) Robot in a tight environment

Figure 6.1: Three different views of the robot.

Figure 6.2: The IAS-Lab Pioneer 3-AT during the navigation.

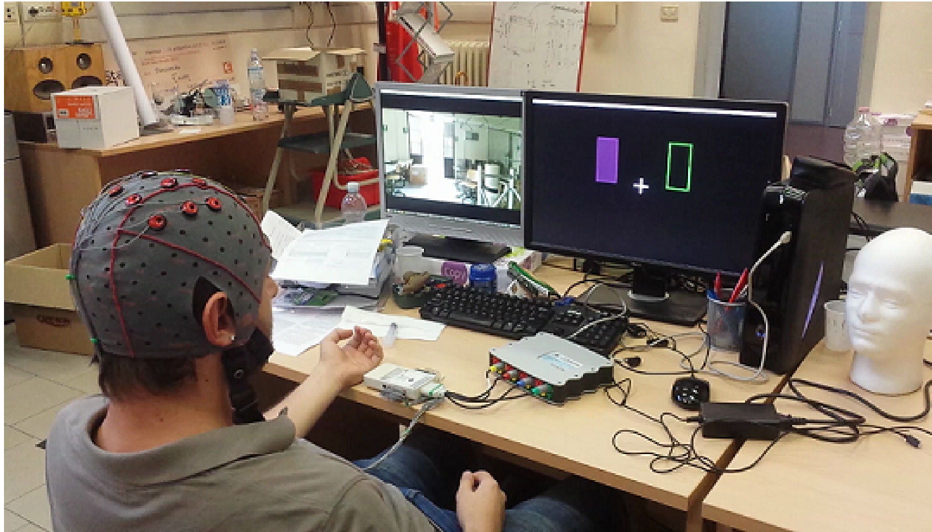


Figure 6.3: The BCI subject during the experiment, delivering a left command.

6.1 Future Works

Some possible avenues for improvement include:

- Implementation of other kind of sensor in the control loop, as the ultrasonic sensors. This kind of sensors will enhance the robot perception to glass walls.
- Extending the field of view of the robot, to provide better information to the user about the robot position, by adding another Kinect v2.
- Improving the shared control system, so that the user could make the robot turn less with a single command, or making the command dependent to the robot surrounding, e.g. if the user deliver a left command, but the robot is near a wall, it will turn at the next crossroad.
- Relieving the workload on the end-user by adding some attractors in the environment.
- Adding a third command to the BCI framework, corresponding to a relax state, where the user does not want to deliver any command.
- Adding a **FOLLOWING** state to the system, so we could exploit people detection and tracking algorithms, as described in [55, 56], in order to relieve even more the workload on the end-user.

Bibliography

[1] J. M. Findlay and I. D. Gilchrist, *Active vision: The psychology of looking and seeing*. No. 37, Oxford University Press, 2003.

[2] J. González-Jiménez, C. Galindo, and J. Ruiz-Sarmiento, “Technical improvements of the giraff telepresence robot based on users’ evaluation,” in *2012 IEEE RO-MAN: The 21st IEEE International Symposium on Robot and Human Interactive Communication*, pp. 827–832, IEEE, 2012.

[3] J. Yamaguchi, C. Parone, D. Di Federico, G. Felzani, and P. B. Zobel, “The telepresence robot for social participation: how much assistance is required?,” *International Journal of Integrated Care*, vol. 15, no. 7, 2015.

[4] M. E. Foster, A. Gaschler, M. Giuliani, A. Isard, M. Pateraki, and R. Petrick, “Two people walk into a bar: Dynamic multi-party social interaction with a robot agent,” in *Proceedings of the 14th ACM international conference on Multimodal interaction*, pp. 3–10, ACM, 2012.

[5] T.-C. Tsai, Y.-L. Hsu, A.-I. Ma, T. King, and C.-H. Wu, “Developing a telepresence robot for interpersonal communication with the elderly in a home environment,” *Telemedicine and e-Health*, vol. 13, no. 4, pp. 407–424, 2007.

[6] H. Durrant-Whyte and T. Bailey, “1simultaneous localisation and mapping (slam): Part i the essential algorithms,” 2006.

[7] D. Fox, W. Burgard, F. Dellaert, and S. Thrun, “Monte carlo localization: Efficient position estimation for mobile robots,” *AAAI/IAAI*, vol. 1999, pp. 343–349, 1999.

[8] G. Grisetti, C. Stachniss, and W. Burgard, “Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling,” in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pp. 2432–2437, IEEE, 2005.

[9] A. Doucet, N. De Freitas, K. Murphy, and S. Russell, “Rao-blackwellised particle filtering for dynamic bayesian networks,” in *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pp. 176–183, Morgan Kaufmann Publishers Inc., 2000.

[10] M. Carraro, M. Antonello, L. Tonin, and E. Menegatti, “An open source robotic platform for ambient assisted living,” *Artificial Intelligence and Robotics (AIRO)*, 2015.

[11] R. Kala, A. Shukla, R. Tiwari, S. Rungta, and R. Janghel, “Mobile robot navigation control in moving obstacle environment using genetic algorithm, artificial neural networks and a* algorithm,” in *Computer Science and Information Engineering, 2009 WRI World Congress on*, vol. 4, pp. 705–713, IEEE, 2009.

[12] M. P. Garcia, O. Montiel, O. Castillo, R. Sepúlveda, and P. Melin, “Path planning for autonomous mobile robot navigation with ant colony optimization and fuzzy cost function evaluation,” *Applied Soft Computing*, vol. 9, no. 3, pp. 1102–1110, 2009.

[13] R. Leeb, L. Tonin, M. Rohm, L. Desideri, T. Carlson, and J. d. R. Millan, “Towards independence: a bci telepresence robot for people with severe motor disabilities,” *Proceedings of the IEEE*, vol. 103, no. 6, pp. 969–982, 2015.

[14] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009.

[15] R. B. Rusu and S. Cousins, “3d is here: Point cloud library (pcl),” in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 1–4, IEEE, 2011.

[16] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. " O’Reilly Media, Inc.", 2008.

[17] clearpathrobotics, “Turtlebot.” Available at `https://www.clearpathrobotics.com/turtlebot-2-open-source-robot/`, June 2016.

[18] mobilerobots, “Pioneer 3-at.” Available at `http://www.mobilerobots.com/Libraries/Downloads/Pioneer3AT-P3AT-RevA.sflb.ashx`, July 2016.

[19] robotshop, “Hokuyo urg-04lx-ug01 scanning laser rangefinder.” Available at http://www.robotshop.com/en/hokuyo-urg-04lx-ug01-scanning-laser-rangefinder.html, June 2016.

[20] Sick, “Lms100-10000 | lms1xx.” Available at https://www.sick.com/media/pdf/1/41/841/dataSheet_LMS100-10000_1041113_en.pdf, Sept. 2016.

[21] K. Khoshelham and S. O. Elberink, “Accuracy and resolution of kinect depth data for indoor mapping applications,” *Sensors*, vol. 12, no. 2, pp. 1437–1454, 2012.

[22] J. Sell and P. O’Connor, “The xbox one system on a chip and kinect sensor,” *IEEE Micro*, vol. 2, no. 34, pp. 44–53, 2014.

[23] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to autonomous mobile robots*. MIT press, 2011.

[24] E. Welzl, “Constructing the visibility graph for n-line segments in o (n 2) time,” *Information Processing Letters*, vol. 20, no. 4, pp. 167–171, 1985.

[25] H. Choset, S. Walker, K. Eiamsa-Ard, and J. Burdick, “Sensor-based exploration: Incremental construction of the hierarchical generalized voronoi graph,” *The International Journal of Robotics Research*, vol. 19, no. 2, pp. 126–148, 2000.

[26] N. H. Sleumer and N. Tschichold-Gürman, “Exact cell decomposition of arrangements used for path planning in robotics,” *Institute of Theoretical Computer Science, ETH Zürich, Technical Reports*, vol. 329, 1999.

[27] J.-C. Latombe, *Robot motion planning*, vol. 124. Springer Science & Business Media, 2012.

[28] R. E. Korf, “Depth-first iterative-deepening: An optimal admissible tree search,” *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[29] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[30] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[31] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Transactions on Robotics*, vol. 21, no. 3, pp. 354–363, 2005.

[32] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[33] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," *The international journal of robotics research*, vol. 5, no. 1, pp. 90–98, 1986.

[34] T. Hellström, "Robot navigation with potential fields," *Department of Computing Science, Umea University, Tech. Rep*, 2011.

[35] G. Han, W. Fu, and W. Wang, "The study of intelligent vehicle navigation path based on behavior coordination of particle swarm," *Computational intelligence and neuroscience*, vol. 2016, p. 1, 2016.

[36] E. Bicho and G. Schöner, "The dynamic approach to autonomous robotics demonstrated on a low-level vehicle platform," *Robotics and autonomous systems*, vol. 21, no. 1, pp. 23–35, 1997.

[37] A. Steinhage and R. Schoner, "The dynamic approach to autonomous robot navigation," in *Industrial Electronics, 1997. ISIE'97., Proceedings of the IEEE International Symposium on*, vol. 1, pp. SS7–S12, IEEE, 1997.

[38] W. Taymans, S. Baker, A. Wingo, R. S. Bultje, and S. Kost, "Gstreamer application development manual (1.2. 3)," *Publicado en la Web*, 2013.

[39] S. Moghimi, A. Kushki, A. Marie Guerguerian, and T. Chau, "A review of eeg-based brain-computer interfaces as access pathways for individuals with severe disabilities," *Assistive Technology*, vol. 25, no. 2, pp. 99–110, 2013.

[40] R. Leeb, S. Perdikis, L. Tonin, A. Biasiucci, M. Tavella, M. Creatura, A. Molina, A. Al-Khodairy, T. Carlson, and J. dR Millán, "Transferring brain–computer interfaces beyond the laboratory: successful application control for motor-disabled users," *Artificial intelligence in medicine*, vol. 59, no. 2, pp. 121–132, 2013.

[41] H. Anupama, N. Cauvery, and G. Lingaraju, "Brain computer interface and its types-a study," *International Journal of Advances in Engineering & Technology*, vol. 3, no. 2, p. 739, 2012.

[42] P. Forslund, "A neural network based brain-computer interface for classification of movement related eeg," 2003.

[43] T. N. Lal, M. Schroder, T. Hinterberger, J. Weston, M. Bogdan, N. Birbaumer, and B. Scholkopf, “Support vector channel selection in bci,” *IEEE transactions on biomedical engineering*, vol. 51, no. 6, pp. 1003–1010, 2004.

[44] W. D. Penny, S. J. Roberts, E. A. Curran, M. J. Stokes, *et al.*, “Eeg-based communication: a pattern recognition approach,” *IEEE Transactions on Rehabilitation Engineering*, vol. 8, no. 2, pp. 214–215, 2000.

[45] S. Jirayucharoensak, S. Pan-Ngum, and P. Israsena, “Eeg-based emotion recognition using deep learning network with principal component based covariate shift adaptation,” *The Scientific World Journal*, vol. 2014, 2014.

[46] G. Vanacker, J. del R Millán, E. Lew, P. W. Ferrez, F. G. Moles, J. Philips, H. Van Brussel, and M. Nuttin, “Context-based filtering for assisted brain-actuated wheelchair driving,” *Computational intelligence and neuroscience*, vol. 2007, pp. 3–3, 2007.

[47] A. Lécuyer, F. Lotte, R. B. Reilly, R. Leeb, M. Hirose, M. Slater, *et al.*, “Brain-computer interfaces, virtual reality, and videogames.,” *IEEE Computer*, vol. 41, no. 10, pp. 66–72, 2008.

[48] M. Velliste, S. Perel, M. C. Spalding, A. S. Whitford, and A. B. Schwartz, “Cortical control of a prosthetic arm for self-feeding,” *Nature*, vol. 453, no. 7198, pp. 1098–1101, 2008.

[49] Q. Zhao, L. Zhang, and A. Cichocki, “Eeg-based asynchronous bci control of a car in 3d virtual reality environments,” *Chinese Science Bulletin*, vol. 54, no. 1, pp. 78–87, 2009.

[50] G. E. Fabiani, D. J. McFarland, J. R. Wolpaw, and G. Pfurtscheller, “Conversion of eeg activity into cursor movement by a brain-computer interface (bci),” *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 12, no. 3, pp. 331–338, 2004.

[51] M. Bensch, A. A. Karim, J. Mellinger, T. Hinterberger, M. Tangermann, M. Bogdan, W. Rosenstiel, and N. Birbaumer, “Nessi: an eeg-controlled web browser for severely paralyzed patients,” *Computational intelligence and neuroscience*, vol. 2007, 2007.

[52] J. d. R. Millán, R. Rupp, G. R. Müller-Putz, R. Murray-Smith, C. Giugliemma, M. Tangermann, C. Vidaurre, F. Cincotti, A. Kübler, R. Leeb, *et al.*, “Combining brain–computer interfaces and assistive technologies: state-of-the-art and challenges,” 2010.

[53] F. L. Markley, Y. Cheng, J. L. Crassidis, and Y. Oshman, "Averaging quaternions," *Journal of Guidance, Control, and Dynamics*, vol. 30, no. 4, pp. 1193–1197, 2007.

[54] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Automata theory, languages, and computation," *International Edition*, vol. 24, 2006.

[55] M. Carraro, M. Munaro, and E. Menegatti, "Cost-efficient rgb-d smart camera for people detection and tracking," *Journal of Electronic Imaging*, vol. 25, no. 4, pp. 041007–041007, 2016.

[56] M. Carraro, M. Munaro, and E. Menegatti, "A powerful and cost-efficient human perception system for camera networks and mobile robotics," in *Intelligent Autonomous Systems 14*, Springer International Publishing, 2016.

Acknowledgements

I would like to thank several individuals for their support, motivation, and inspiration while writing this thesis. Only through their constant financial, intellectual and emotional support could this work have been completed.

I would like to acknowledge Prof. Emanuele Menegatti for giving me the chance to work in a research lab and for the passion for robotics transmitted through his classes.

Huge thanks to my co-advisors Marco Carraro, for his support and day-by-day guidance in the development of the work presented in this thesis, and Andrea Cimolato, for helping me using the BCI and for last days hard work. I would also like to thanks all those people in IAS-Lab that helped me during all this project.

To my friends and roommates, thank you for listening, offering me advice, and supporting me through this entire process. Special thanks to my university friends for sharing those stressful and difficult moments with me, and for all those gaming night with the BraWarteam[...]. Thank you: Alberto, Andrea, Davide, Debora, Elisabetta, Emanuela, Enrico, Federica, Francesco, Giacomo, Leonardo, Manuel, Marco, Mario, Nicola, Silvia, Simone and all of the other people.

Also, I would like to thank my family for all their love and support. My family has always been there for me, to help me through personal hardships and push me to succeed. They have taught me the value of an education and the benefit of hard work. Mom, Dad, Davide, Stefano, Valeria, Grandma, and Grandpa, thank you.

Last but not least, I would like to thank my girlfriend, Francesca, for being such a sweet, caring, and understanding person. She has been my motivation to finish this work and pushed me to keep going. Her love has been inspiring. I am where I am today only because of her love and support.