POLYTECHNIC UNIVERSITY OF TURIN

Master of Science in Mechatronic Engineering

Master's Degree Thesis

A ROS-based Platform for Autonomous Vehicles: Foundations and Perception

Academic Advisor
Massimo Violante

Candidates
Sebastiano Barrera
Vincenzo Comito

Internship Tutor
Stefano Moccia

A.A. 2017/2018

Contents

Chapter 1

Introduction

1 Abstract

Autonomous driving (AD) and Advanced Driver-Assistance Systems (ADAS) are at the forefront of today's technical prospective achievements in the field of automotive engineering, promising disruptive changes in the role of transportation in today's society, and in the world's economy in general. This endeavor is fueled by the production of new advances in research, and their swift and smooth transfer to the industry. The present work aims at helping with the latter: it is a software and hardware platform based on ROS (Robot Operating System) that aims to allow researchers, students and companies in the field to share algorithms and data in the most painless way. It provides a functionality baseline that includes sensor support, localization, SLAM, object and lane detection, and basic navigation capabilities. Moreover, modern practices in distributed systems engineering are applied to allow users to easily monitor the system, diagnose problems, transfer and deploy the system's components.

2 Overview

Autonomous driving (AD) and Advanced Driver-Assistance Systems (ADAS) are at the forefront of today's technical achievements in the field of automotive engineering, promising disruptive changes in the role of transportation in today's society, and in the world's economy in general.

Generally speaking, these systems are implemented by augmenting standard vehicles (typically cars, trucks) with a number of sensors that continuously scan and collect real-time data about the surrounding environment, and (in the AD case) with actuators that allow the system to effectively drive the car by applying commands to vehicle's powertrain. The effect of the commands is then observed by the sensors, closing the control loop. The result is that the modern, ADAS-enabled

vehicle can be seen as a fully-fledged "robot on wheels", with a similar theoretical background, similar physical and logical organization, and a similar suite of on-board hardware and software components.

The practical development of such systems is fueled by the research advances in the field of robotics, and their swift and smooth transfer to the industry. Similarly, companies operating in the field can provide valuable feedback to the academia in the form of data, extensive testing, and optimization of both the software and the hardware.

The present work aims at helping with this very "transfer". In this thesis, we propose a software and hardware platform built on top of ROS (Robot Operating System) that researchers, students and companies can use to share and run programs and data, reducing as much as possible the friction typically encountered when putting a system together out of heterogeneous pieces.

This is achieved, in summary, by:

- building on top of ROS (Robot Operating System), which allows access to the largest ecosystem and community currently existing in the field of robotics, both in industry and academia;
- applying modern practices in distributed systems engineering for easy transfer, monitoring, deployment of software components.
- providing a functionality baseline that includes sensor support, localization, SLAM, object detection, and basic navigation capabilities.

Figure 1.1 offers an high-level overview of the perception and processing system. The sensors and algorithms involved are described in details in the following chapters.

The present study is organized as follow: After a short introduction on the topic of autonomous driving and the current state of the art, we go through the various sensors and devices available on our platform in chapter 2. We then discuss the topic of localization in chapter 3, where we show how the vehicle tracks its position on the road. In chapter 4 the techniques for detecting objects and road users through camera and lidar are analyzed. The results of these chapters are used in chapter 5 for path planning. Chapter 6 and chapter 7 are dedicated to modern software engineering paradigms and support infrastructures that helped us during the development and testing process. Finally, we review the obtained results in chapter 8, suggesting some future works based on ideas we have not had the chance to try out in chapter 9.

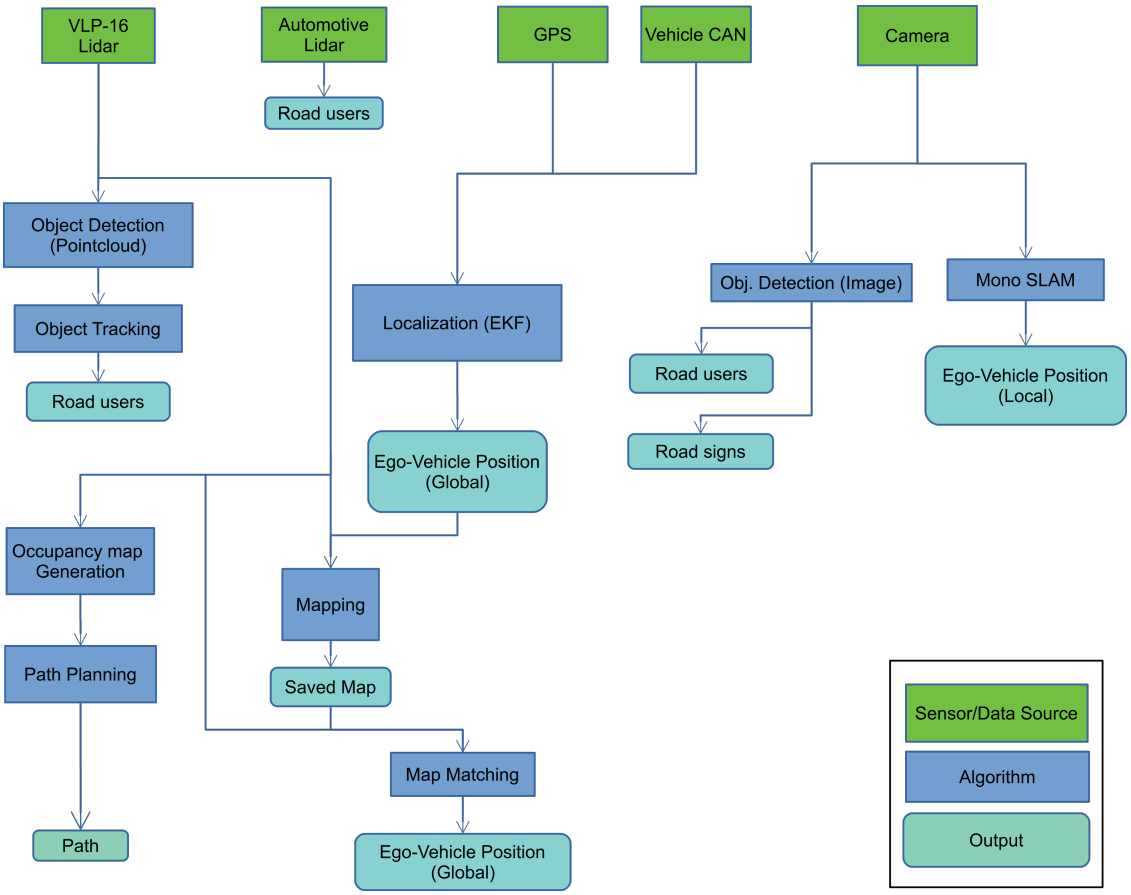


Figure 1.1: Functional schema of the perception functionalities.

3 Objective Software

The work for this thesis was carried out with the blessing and support of Objective Software GmbH[1] and Objective Software Italia s.r.l.

Objective Software operates since 2006 in the field of autonomous vehicles, driver-assistance and teleoperated driving systems, in conjunction with companies such as BMW and Bosch.

In the context of this work, it is worth noting that Objective is the provider of the research vehicle and sensors that constitute the hardware part of the platform we are proposing. The vehicle is further described at section 2.

[1] www.objective.de

4 State of the Art

Self-driving cars are a hot topic, both because they are an innovative use of artificial intelligence and because they predicted to have a deep impact on society, affecting in many ways the consumer market, the safety of road users, and the jobs of thousands of people. If a few years ago autonomous driving sounded like a far-fetched vision of technological companies, today traditional car manufactures are also making their bet on it, with ambitious plans but less optimistic predictions. No one seems to agree on when self-driving cars will hit the roads: predictions range from twenty to a few years, while the voice of some enthusiastic experts assert they are already here. Such confusion is also due to the definition of self-driving vehicle, which is not as clear as a consumer may think. Even if there is no standard definition and no approval requirement as of today, the definitions offered by the SAE International [1] provide some insight[54]. These are comprised of six levels of automations:

Level 0 No assistance, only warnings.

Level 1 The ADAS may act on steering or gas, not at the same time.

Level 2 The gas pedal and steering wheel are controlled by the vehicle in some circumstances, but the driver needs to pay the same level of attention and be ready to intervene.

Level 3 The vehicle performs all the tasks (including lane/road changing, stops etc...). The driver still needs to be focused as in previous level.

Level 4 The driver does not need to pay attention when the system has control in some conditions.

Level 5 The system is capable of controlling the vehicle in all conditions, driver is not required.

Most of today's premium cars' offerings may be described by level 1 or 2. **Tesla** is about to reach level 3. Companies like **Waymo** are publicly against such partial systems[2], alleging that the benefits are pointless, if not harmful, as long as the driver still needs to pay attention. Indeed, recent incidents suggest that it is too much to ask to a driver to distrust a semi-automated system whose use is encouraged and advertised at the same time: any human driver will eventually decrease their level of attention, which unfortunately has proved to be fatal in some cases. Thus, an *everything or nothing* approach seem the only viable way car companies should focus on.

[2] https://www.wired.com/2017/01/human-problem-blocking-path-self-driving-cars/

The current assistance systems mostly regard a single, independent functionality:

Adaptive Cruise Control (ACC) takes care of controlling the car's speed, by electronically acting on acceleration or brake, in order to reach a set target velocity while keeping a safe distance from the next vehicle on the same lane. This way, the driver can keep the foot off the gas pedal and only has to control the steering wheel. The system is usually implemented with the use of the front radar.

Lane Keeping Assistant (LKA) instead directly controls the steering wheel with the aim of maintaining the vehicle centered in the current lane. It can be seen as an automated lane departure warning system that, instead of notifying the driver, changes the vehicle's direction. These system are usually implemented with the front camera, which is able to extract lateral distance of the lane markings. Modern version of LKA also implement lane changing or highway exit.

Traffic Sign Recognition (TSR), also implemented using cameras, recognizes road signs and their meaning, reporting it to the user on the instrument panel or also by controlling the speed of the vehicle accordingly. Today, deep neural networks allow to recognize all signs with very high confidence.

These systems are more of a result of automatic control theory rather than artificial intelligence such as that which will characterize future autonomous vehicles. The driving task is complex: only in the latest years computers have been able to extrapolate semantic information from visual, lidar or radar data, and still in many cases without reaching the performance of a human. Even by having perfect perception of the environment around the vehicle, the autonomous system still has to deal with the complex system of rules, choices and behaviour that cannot be simply coded with classical methods. Thus, new achievements in artificial intelligence seem required for full level 5 automation on every scenario.

Still, subsets of the task, such as driving on highways, involve much less complexity and current systems seem to behave pretty well, with long records of proven automation with limited intervention by the driver.

We will now focus on some of the technological companies that propose different plans. The statements of traditional automakers (except General Motors and its Cruise subsidiary) are still generic nowadays and it is thus hard to define their strategy.

4.1 Tesla

As a car manufacturer, Tesla is definitely the most dedicated to technological innovation, branding its cars as the first fully electrical, autonomous vehicles. The

company states to have a fully defined plan for achieving self-driving cars that will be carried out, interestingly, not only on future vehicles but also on some of the past models. Indeed, according to the producer[1] all the vehicles already have all of the sensors and processors needed. What is missing now is software, which is periodically improved and sent to vehicle thanks to over-the-air updates. The hardware architecture proposed by Tesla is the following:

- Eight cameras around the vehicle, with different fields of view and maximal distances.
- Twelve ultrasonic sensors
- Front radar

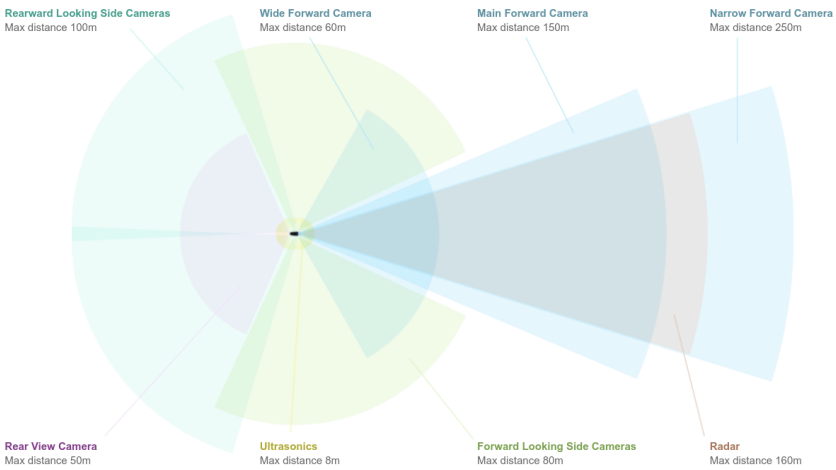


Figure 1.2: The sensors of the Autopilot system [Tesla]

As the number of cameras may suggest, they are the main sensor for autonomous driving, while radar and ultrasonics have a backup and safety role. Surprisingly, the system has no lidar, which many companies such as Ford, BMW and Audi will probably include. The thesis here is that visual information from cameras is enough for understanding every scenario, thanks to the use of neural networks.

The processing will be handled by the Drive PX2 system from NVIDIA, a Tegra System-on-Chip designed specifically for the automotive industry and deep learning. Tesla is however also partnering with AMD, so it is likely that this choice will be reviewed.

[1] *All the vehicles have the hardware needed for full self-driving capability* - Tesla - Autopilot

A promotional video (fig. 1.3) from Tesla shows the current results of Autopilot: the system recognizes people and objects (blue), it detects the full road surface along with lane markings (red and cyan), computes optical flow (green lines) and recognizes traffic signs. These results prove that Tesla's engineers successfully integrated the state of the art research in a consumer product.

Figure 1.3: A snapshot from the Autopilot promotional video.

Today, the car controls the vehicle's speed and steering angle during normal cruise, lane changing and auto parking. The company requires the driver to be focused in all cases; this is enforced by constantly checking that the driver's hands are on the wheel.

4.2 Waymo

Originally born as a Google project, **Waymo** is now a company belonging to the bigger Alphabet holding. The company does not produce cars: instead it collaborates with FCA and Jaguar, retrofitting their vehicles with cameras, lidars and radars and providing the control software. Lidars, in particular the top Velodyne HDL-64, have an important role for generating the full 3D model of the environment and objects in it.

As Waymo announced a public autonomous taxi service in Phoenix, Arizona by the end of 2018, we can safely suppose that the system is capable of analyzing high level information necessary for fully navigating an urban scenario, making it one of the first systems capable of reaching level 4 or level 5 autonomy. So far, Waymo's vehicles have travelled for more than 11 million kilometers in driverless mode.

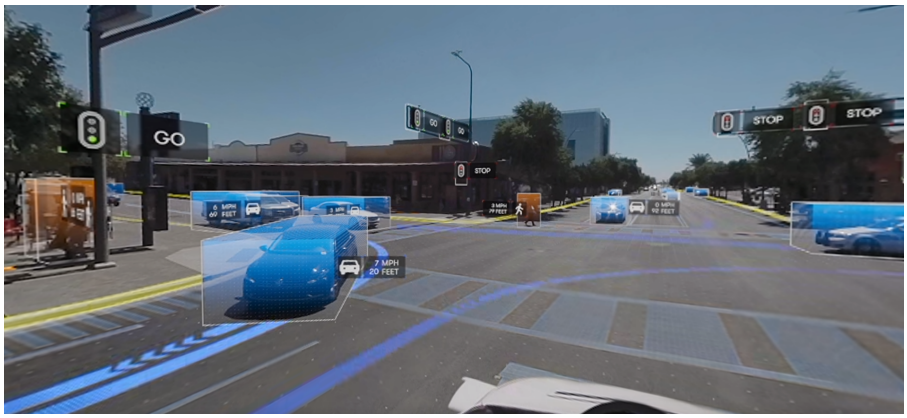


Figure 1.4: The joint use of lidars and cameras allows to find all objects and people on the road all around the vehicle. [Waymo 360° Experience: A Fully Self-Driving Journey - Waymo]

4.3 NVIDIA

Having successfully surfed the wave of deep learning innovation by providing dedicated processors, NVIDIA today produces both embedded computers and software for automotive applications intended for car manufacturers or Tier 1 suppliers. The company also proposes the use of virtual testing and learning, where the autonomous system is trained in a simulated road environment. The advantages of this system, beyond the safety and obvious simplifications, is the scalability afforded by the large number (thousands) of simulations can be ran at the same time. However, it is hard to say if it system can replace, even in part, field testing on real vehicles.

Chapter 2

Platform Description

1 ROS

ROS [39], short for Robot Operating System, is a fully open-source software platform designed for robotics, based on an unmodified GNU/Linux operating system. It provides an abstraction for easy interoperability, basic software, and a means to create, distribute, build and (partly) deploy software packages.

1.1 Introduction to ROS

The core feature of ROS' design is the emphasis on distributed programming: a ROS-based system is typically composed of processes exchanging messages following the publisher-subscriber (aka pub-sub) pattern. Such processes are called, in ROS terminology, *nodes*.

The communication between nodes, and other core system functions, are managed by a single program named the "ROS master", or "ROS core".

The publisher-subscriber pattern can be thought of as an abstract communication mechanism, where participants (nodes) announce their wish to receive certain messages ("subscribe") or their intention to send them ("publish"). Moreover, in this abstraction the messages are not explicitly targeted at nodes, but are instead routed through virtual communication channels named *topics*. A topic is nothing much more than a label that groups related messages. For example, an image processing node may subscribe to the *camera* topic, signifying its interest in the raw images captured by the camera; correspondingly, a "driver" node may announce itself as a publisher of the *camera* topic, promising that it will capture camera images directly from the device, and then encode them, envelope them in a properly formatted message, and publish them on the topic.

This abstraction layer is implemented by a protocol that also involves the ROS

master. This keeps track of every node's publications and subscriptions, and announces each new registration or deletion of publisher or subscriber to every interested node. The nodes will then open or close the corresponding connections, and carry out the communication directly, without routing the messages' data through the ROS master process. The transport protocol can be based either TCP or UDP; each node can negotiate its preference, based on reliability or performance requirements.

There are no limits to the number or name of topics at any given time. A topic is created simply as soon as the first publisher or subscriber has announced itself to the master. Subscribers and publishers to the same topic are always connected together, no matter which one is announced first. Finally, there can be any number of publishers and subscribers to any given topic; there will be one connection per publisher-subscriber couple connecting them directly (or "as directly" as the underlying network stack allows).

This design generates a few useful properties:

- **Network transparency**. Since the communication mechanism is based on top of the standard TCP/IP or UDP/IP network stack, its trivial to distribute the robot's (or vehicle's) functionality across different physical computers. As long as all nodes are connected to the same ROS master, this will inform every party of each other's network address, allowing them to reach each other without any further configuration.

- **Fault tolerance**. Nodes don't "call each other by name": messages are always sent to, or received from, a topic. A topic simply exists as long as there is at least one publisher or one subscriber for it; it will be created/destroyed automatically as needed. As a consequence, the crash of a single node or a network partition never directly causes the failure of other nodes; at worst, the flow of messages is interrupted, possibly for just a short time. More interestingly, node failures can be easily detected and monitored by the system, as the ROS master keeps track of each node's liveness, and can be readily queried by standard ROS tools. This endows the system with a useful degree of fault tolerance, to the point where a "let-it-crash" philosophy becomes practical. This design philosophy posits that it is more desirable to have much simpler internal designs for the nodes, and allow them to abort execution as soon as possible and describing the mishap simply through log messages, than it is to devise complex error recovery mechanisms on a case by case basis. The fine granularity of a single node's responsibility, and the observability of the failure to monitoring tools and other nodes allows to recover from error conditions much more effectively, and is much less error-prone to implement and maintain. This has been applied to great advantage in other distributed systems architectures, such as those based on the Erlang VM.

- **Late-binding, and standard tooling.** The ROS system is highly dynamic: late-binding is a driving design principle. In other words, objects of any kind in the system, including nodes, topics and message formats, are all simply referred to by their name. They are typically observed, destroyed, and often even brought into existence, by simply using the name in queries to the ROS master. This amounts, essentially, to a very powerful *decoupling mechanism*. This decoupling, in turn, allows for a whole suite of standardized command line tools and libraries to query and manipulate each part of the system, without relying on each other's internals. For example, `rosnode` can be used to list and query active nodes in the system, with their publications and subscriptions; `rostopic` can list and get information about topics, and publish arbitrary messages. No pre-processing or configuration is required for these tools to interface with new nodes or topics. Another notable example is the **RViz** tool, which can visualize many kinds of messages in a 3D scene, thanks to a rich set of plugins for various message types.

On top of the primitives just described, a number of conventional tools are implemented that provide basic services required by any robotic application. This can be thought of as the *basic* or *system software* of other operating systems. One of these, in particular, merits mention: the **tf** package for defining and communicating transforms between frames of reference. Using the messages and nodes included in this service, other nodes can express geometric data (state of the robot's joints, object's positions, point clouds, and anything else) in whichever frame of reference they prefer. By also communicating the rigid transforms that bind the frames of reference to each other in a tree structure, the nodes can relate geometric data gathered (or generated) independently. This becomes indispensable for robots that manipulate or navigate the environment, in order to relate its own position with the perceived landmarks.

1.2 Open Source Software

It's in a sense our duty to note here that the almost totality of the software that we have used (directly or as foundation for our own work) is Free and/or Open Source Software, available under free software licenses such as GNU's GPL and LGPL, MIT, BSD, and others. It's in this way that we stand on the shoulder of giants: our achievements are fundamentally enabled by the great deal of work and effort spent by the hundreds of contributors to the GNU/Linux OS and distributions, the ROS framework, OpenCV, PCL, and the myriad of other software packages that we have come to rely on.

Speaking about ROS in particular, it should be noted that ever since its inception, its mission has been to enable and foster *collaborative* robotics software development. As such, it fits naturally in the framework of open source software.

All of the source code for the basic software is freely available; the official documentation is maintained in a wiki that many people contribute to.

More notably, ROS-compatible software is usually organized in *packages*. These are basic containers of source code and data, that can be distributed, built, and installed in a standard way, with the use of a dedicated tool named *Catkin*. As a consequence, the overwhelming majority of ROS software is published in source code form, and built by the final users. The more popular packages, especially those considered to be part of a "standard" are also built and distributed as binary Debian packages, through the official ROS APT repository, for easier installation.

During its history, a very large number of researchers, developers, companies and hobbyists has flocked around the ROS project, giving rise to a lively community that, to this day, represents one of the main advantages in choosing ROS as the foundation for new robotics projects. Together with the emphasis on late-binding and non-command messages, this is the main factor that makes the ROS platform a fertile ground for innovation in the field of robotics, and one of the main catalysts for its application in industry.

1.3 Limitations

The current version of ROS, dubbed the 1.x series, suffers from some limitations, mostly due to some unforeseen use cases. These have emerged throughout the years due to ROS' very popularity.

The most "wished for" feature is soft and hard **real-time capabilities**. The type of robot ROS was initially designed for was a "simple" (though well built) robot with a mobile platform for indoor exploration, and robotic arms to grab objects and move them around to perform simple manipulation tasks. Nowadays though, there is great demand for a standardized software platform for unmanned aerial vehicles, submarine robots, and road vehicles (we were part of this last group). ROS seems to be a natural fit, where much of the technology can be reused with little change, but these newer application areas have much more stringent timing requirements. For example, in order for an autonomous car to safely react to a life-threatening situation, the detection of such a condition, the decision to brake and the actuation of the command all have to be completed within a fixed time interval. Missing the deadline for a single cycle may not necessarily be a disaster, but in order to be considered safe at all, the system has to guarantee action within a bounded time interval (as opposed to just, "eventually").

ROS was not designed for any kind of real-time operation. Every message is delivered under a "best-effort" policy, with no guarantees as to how late the message will be when it arrives, nor how long it will take to produce a reaction. (In all fairness, even in the current design, most messages are time-stamped, so their "staleness" at the time of arrival can be estimated, at least roughly.)

Another shortcoming of ROS 1.x is that it was designed to be run on common, PC-style computers, on the x86_64 and ARM architectures, with plenty of computing power and memory to spare. For some types of robots, instead, some prefer to run the system in part or in total on embedded systems. Typically, these are markedly under-powered and lack an operating system, but it is much easier to guarantee certain properties of the resulting hardware-software system. For example, having a single program possess full control over an embedded computer's resources makes it much easier to guarantee the satisfaction of hard real-time requirements (this ties in directly with ROS' shortcomings for real-time), or to have truly exhaustive testing that reliably ensures its correctness.

Moreover, ROS was designed to work in a (quasi-)reliable network, where the chance of partition and packet loss is low to nil, such as local cabled Ethernet networks, or close-range wireless. On a less reliable network (e.g. a multi-robot system deployed on the field, where radio interference may make wireless communications shaky), messages may well be lost undetected, lowering the chance of accomplishing the mission, or causing excessive retransmission (e.g. with TCP) and a subsequent performance degradation.

The pain points just described are at the core of the motivation for the new iteration of the ROS platform, called ROS 2. This version approaches these issues radically, by changing some core parts of ROS' design. Among these, is the adoption of Object Management Group's Data Distribution Service for real-time systems (DDS), a standard that defines a common interface for publish-subscribe pattern implementations. This replaces ROS 1's custom middleware, and enables ROS 2 to take advantage of industry-proven real-time implementations of message passing, network standards and architectures. Moreover, the APIs (for the C++ and Python programming languages) have been overhauled to enable new programming architectures that enable more efficient and reliable concurrency architectures, possibly taking advantage of the limited resources of embedded systems. The package structure, and the build and distribution systems have also been redesigned to favor collaboration between even more disparate parts of the community.

At the time of writing, ROS 2 has reached its first non-beta release ("Ardent Apalone", December 2017), but is not yet ready for production use by the larger community. The basic functionality has been implemented, but the overwhelming majority of ROS-compatible software in existence today is not compatible with ROS 2 yet. Even though migration efforts are slowly starting to take place, we decided to keep using the (still very popular and actively developed) ROS 1, so as to take advantage of the larger amount of software packages available. Nonetheless, we were interested in taking advantage in the new possibilities opened up by ROS 2's architecture, and we plan to test it as soon as the newer platform reaches a sufficient degree of maturity.

2 Vehicle

The research platform was built on a customized *BMW 5 Series*™ vehicle (F10) fig. 2.1. Such vehicle, intended for research activities, offers direct access to the internal communication buses thanks to connectors added in the trunk. The trunk is also well suited to host industrial-grade computers, so that the data can be processed directly on the vehicle and shown on a custom dashboard next to the instrumentation panel.

Figure 2.1: The BMW 5 Series Research Vehicle provided by Objective Software

During the research, we added our own hardware to the vehicle, such as cameras and LIDARs, during the data recording trips.

Being it a research vehicle, it was possible to retrieve real time data from one of the CAN networks (a typical BMW car has up to 8 separate CAN networks). This data was then integrated in the ROS system, as we will show next.

2.1 CAN Protocol

The **C**ontroller **A**rea **N**etwork protocol is an industrial standard, born in the automotive world and later extended to other industries. Developed by Bosch, it was standardized under ISO (ISO 11898 was published in 1993) and soon adopted by most automakers due to its robustness and flexibility. The protocol indeed satisfied the developers' need to have a simple, cost-effective yet fast way to allow communication between different microcontrollers on the vehicle, where more and more processing units are scattered around [1]. The protocol is fast compared to the previously existing ones (up to 1 Mbit/s, but usually limited to **500** Kbit/s), and it is differential, so it guarantees a high signal integrity. It works with only two cables and many nodes can be added over the same bus. Moreover, the protocol has no central arbiter or master, and thus no single point of failure; nodes perform

[1] A typical BMW vehicle has dozens of mCUs

the synchronization (or medium access) logic independently. This greatly improves the robustness of the system, since a faulty node in the network can simply be ignored while the other functions are still available. Consider, as an example, a short range radar module: the failure of this module will only affect parking assistance functions, while it is still possible to drive on the road.

CAN Protocol overview

The protocol[1] occupies the two bottom layer of the ISO/OSI stack, the Physical and Data Link layer. In automotive, it is rare to see higher layer protocols, since applications are usually designed to directly work with message-based protocols whose messages fit CAN frames. In other industries though, it is common to have protocols occupying higher layers of the abstractions stack, such as the CANopen protocol.

Data is transmitted over a single bus composed by two differential lines, which have symmetrical voltage level (see fig. 2.2).

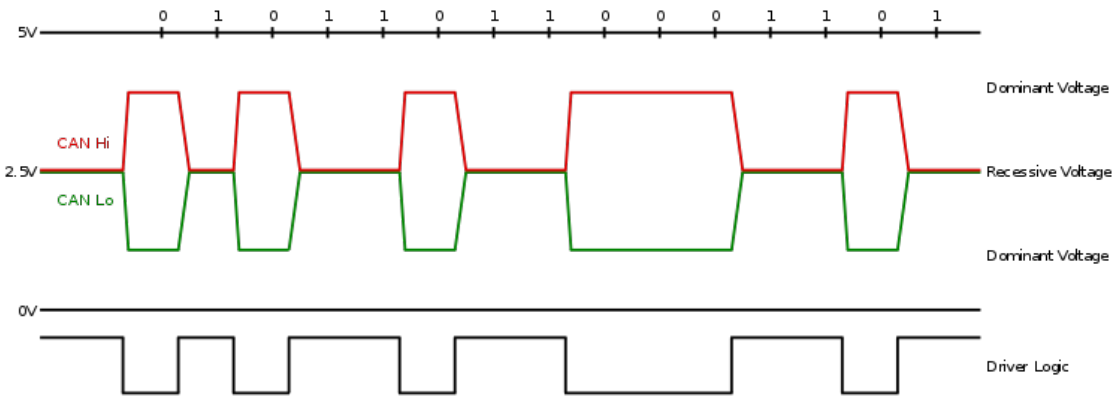


Figure 2.2: The two opposite line in the CAN network. Dominant voltage level corresponds to a 0 bit - [*Wikipedia*]

In a CAN network, nodes exchanges frames in broadcast on the same bus. Bits are written on the positive line as a low or high voltage level. These levels are respectively called **dominant** and **recessive**, due to the fact that, being the transducers open collectors, a given node can only impose a low voltage (dominant) level. The high voltage level is maintained only when no node is transmitting, or they are all transmitting a recessive bit (high impedance). For further clarifying this, let's assume two nodes, A and B, are transmitting a bit at the same time:

[1] [11] provides a more in-depth description of the protocol.

A tries to transmit a recessive bit (high voltage); B, instead, is transmitting a dominant bit (low voltage). Since the low level will be imposed on the line, the dominant bit is successfully transmitted. The A node, sensing the resulting voltage level, acknowledges that the recessive bit was not transmitted. This mechanism is also the foundation of the arbitrage method, described in the following.

On a single transmission line, not more than one node can write in any given moment. Collisions are avoided using a cooperative and priority-based mechanism: when two or more nodes transmit a frame, only the node that is sending the frame with higher priority will acquire the right to transmit, while the others will stop, wait, and retry at a later time.

Every node will continuously listen to the bus, even during transmission, and will compare the emitted signal level against the one observed on the wire. The two will differ when a recessive voltage level is emitted at the same time as a dominant level is emitted by another node.

Priority is defined by the binary content of the frame: reading the frame from most significant bit (MSB) to the least significant (LSB), the frame with lowest binary value has highest priority. The first sequence of bits of each frame is called Arbitration ID, noted as (ID) in the following text. Since the ID is transmitted first, it also determines the priority of the frame.

This means that it is possible to compare the priority of two frames at the same time as they are still being transmitted. This is how each transmitting node deduces whether its own message is the one with highest priority among the ones being transmitted at that very moment. A node will stop transmitting as soon as, while transmitting a recessive bit, a dominant bit over the network is sensed instead.

There are two ISO versions of the CAN protocol, A and B, which mainly differ in the maximum length of the frame payload of the ID.

Since the most widely adopted version in the automotive world is A, we will focus on this, but the same considerations can be translated to the other protocol version.

A CAN Frame is composed by the following fields:

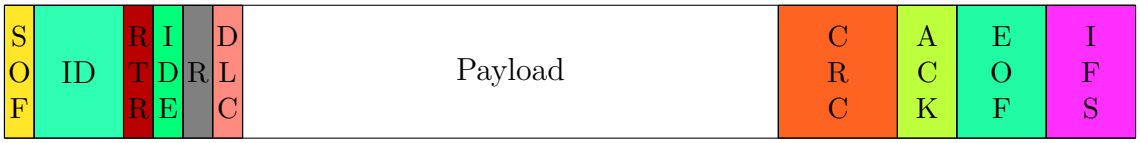
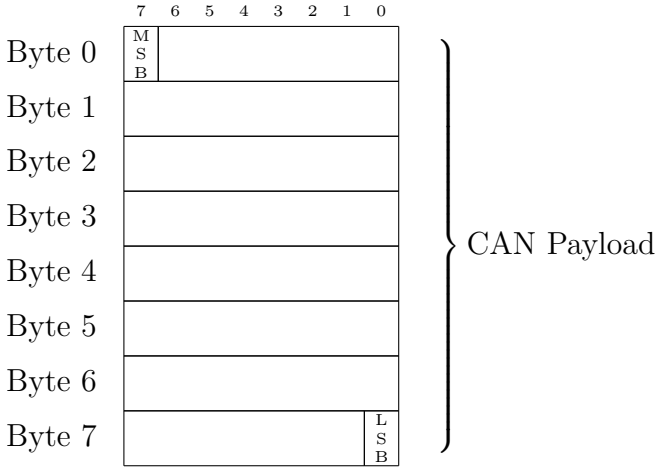


Figure 2.3: Standard (11-bit Identifier) CAN frame

Field Name	Length (bits)	Description
SOF	1	**S**tart **O**f **F**rame. Marks beginning of frame.
IDentifier	11	ID of the can frame.
RTR	1	**R**emote **T**ransmission **R**equest. Used for indicating a request
IDE	1	**ID**entifier **E**xtension. Indicates whether it is a standard or extended frame.
R	1	Reserved bit
DLC	4	**D**ata **L**ength **C**ode. Length of payload, expressed in number of bytes.
Payload	0..64	Payload of the CAN frame. Length may vary from 0 to 8 bytes
CRC	16	**C**yclic **R**edundancy **C**heck. Checksum of the previous payload content.
ACK	2	**ACK**nowledge. Indicates correct read of data from listener nodes.
EOF	7	**E**nd **O**f **F**rame. Marks end of frame.
IFS	7	**I**nter**F**rame **S**pace. Small buffer space where no transmission take place

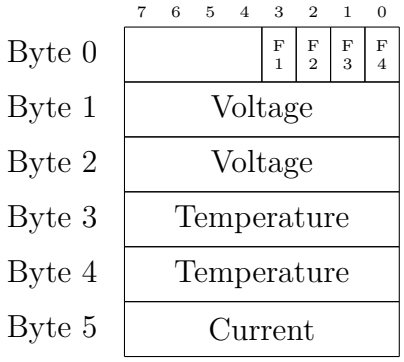
IDs are statically configured by the automaker or the component producer. Each ID is usually associated to a set of data, and usually, given a certain ID, only one node sends frames with this ID.

We will describe the payload from a CAN frame using the following convention: most significant bits appear on the top left, least significant bits appear on the bottom-right. So western style reading order reflects the order of transmission of bits. Every octet is written on a different line.



For an example on how payloads are usually structured, consider a battery sensor that transmits battery information over CAN. The battery sensor will have been

assigned a CAN ID, for instance **0x780**. Assuming that the battery information contains a voltage (16 bit), a temperature (16 bit) and a current (8 bit) field with some flags (4 bits), we may imagine the following layout:



The content of the payload is not standard and must then be interpreted by using a predefined schema. Also, the integer value expressed by a sequence of bits must somehow be converted to a real, decimal value. Usually, this is done by using a fixed-point representation: a linear relationship is defined that relates the real measurement and the binary value. To this end, an offset and a multiplicative value are predefined and used to make the conversion. For instance, the temperature value in byte $2, 3$ of the previous payload expresses a temperature in Celsius using the following formula:

$$temperature_{real} = 30°C + temperature_{hex} \cdot 0.01°C$$

This conversion can happen concretely (performed by the hardware), or only logically, by adapting the program to directly use the fixed-point representation in computation (fixed-point arithmetic).

Many formats were defined over the year to describe payload layout and content. Currently, for the CAN protocol, one of the most widely known formats is Vector™ DBC. Protocol specifications are intellectual property of the car maker, and are thus not shared, though some of them are available online thanks to the reversing attempts made by third parties [1].

Usually, measurement samples such as this one are generated periodically, for instance every $100\mu s$. Intrinsically eventful data, for instance, a button press, are instead sent at the time of the event.

By knowing the exact format of the transmitted data, it is thus possible to retrieve important vehicle information using common commercially available CAN interfaces for computers that provide the raw data

[1] http://opengarages.org/index.php/Raw_link_references_for_CAN_IDs

2.2 fa_can_ros

Within the scope of the thesis, we are mostly interested in the following vehicle data:

- Longitudinal speed
- Yaw rate
- GPS Latitude and Longitude
- Longitudinal and lateral acceleration

These and other measurements are sent by the Dynamic Stability Control (**DSC**) and Navigation microcontrollers. Thanks to the knowledge of the format, we implemented a ROS node called **fa_can_ros** that provides the following topics:

- **/vehicle/commands**
- **/vehicle/dynamics**
- /vehicle/blinklights
- /vehicle/fuel
- **/vehicle/gps**
- /vehicle/powertrain

The format of these topics is vehicle-agnostic, and thus can be also used with any other vehicle manufacturer or model with little effort, as we will show next.

The **fa_can_ros** node runs on the platform's computer, reads raw CAN data using an external CAN interface adapter fig. 2.4, performs conversions of raw data into meaningful measurements, and finally publishes these measurements on the related topics. Update rate is equal to the same frequency of a certain CAN message (e.g. 1Hz for GPS, 100Hz for acceleration etc...)

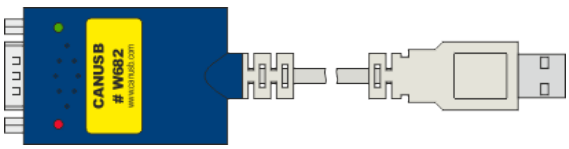


Figure 2.4: CANUSB - A CAN to Serial Converter from LAWICEL

During the implementation of **fa_can_ros** we encountered the problem of data deserialization: integer and float values are expressed in various bit lengths

(often multiple of nibble length), values must be converted using offsets and multiplicative factors, endianness must be taken into account. In order to avoid the use of error-prone conversion of the data through manually written code, we opted to solve the problem by implementing a C++ code generator and employing some metaprogramming techniques.

The code generator (written in Python) processes a CAN message description (in YAML format) and generates a C++ header and source file for each CAN message type. The header and source together define a self-contained C++ class that serializes/deserializes the raw payload from/to a data structure containing measurements values, hiding details on bit positions and order in the CAN message payload. The C++ classes can then be used in the fa_can_ros program for conversion to ROS messages using a callback dispatcher. Indeed, CAN frame are continuously read from an abstract *CanInterface*, which can be a SocketCAN[1] interface, a CANUSB driver, or a binary log of a previous recording) Each CAN frame is first converted to a C++ class instance depending on its CAN ID, selecting bits from payload (2.1) and converting measurements (2.2), then the callback dispatcher calls the corresponding function that takes care of converting data for ROS (2.3). The use of multiple steps provides multiple layer of abstraction.
See fig. 2.5 for an high-level illustration of the process.

Listing 2.1: Bitmasking of a CAN payload by the auto-generated code

```
inline static CAN_GPS_MSG fromData(const uint8_t* data) {
  CAN_GPS_MSG A;
  A.LAN_BYTES = ( (data[7] & 0b11111111UL) >> 0) << 16 |
                (data[6] & 0b11111111UL) >> 0) << 8 |
                (data[5] & 0b11110000UL) >> 4) << 0 );
  A.LON_BYTES = ( (data[2] & 0b00001111UL) >> 0) << 16 |
                (data[1] & 0b11111111UL) >> 0) << 8 |
                (data[0] & 0b11111111UL) >> 0) << 0 );
  return A;
}
```

Listing 2.2: Conversion from fixed-point to floating-point representation of the measurements

```
GPS translate(const CAN_GPS_MSG &A) {
  GPS gps;
  gps.lat = 0.0000501 * A.LAT_BYTES;
  gps.lon = 0.0000501 * A.LON_BYTES;
  return gps;
}
```

[1] https://www.kernel.org/doc/Documentation/networking/can.txt

Listing 2.3: Last step: Conversion to ROS message

```
void RosConverter::send(const GPS &obj, const std::string &msgType,
          const ros::Time &stamp) {
  navSatFixMsg.header.stamp = stamp;
  navSatFixMsg.latitude = obj.lat;
  navSatFixMsg.longitude = obj.lon;
  navsatFixPublisher.publish(navSatFixMsg);
}
```

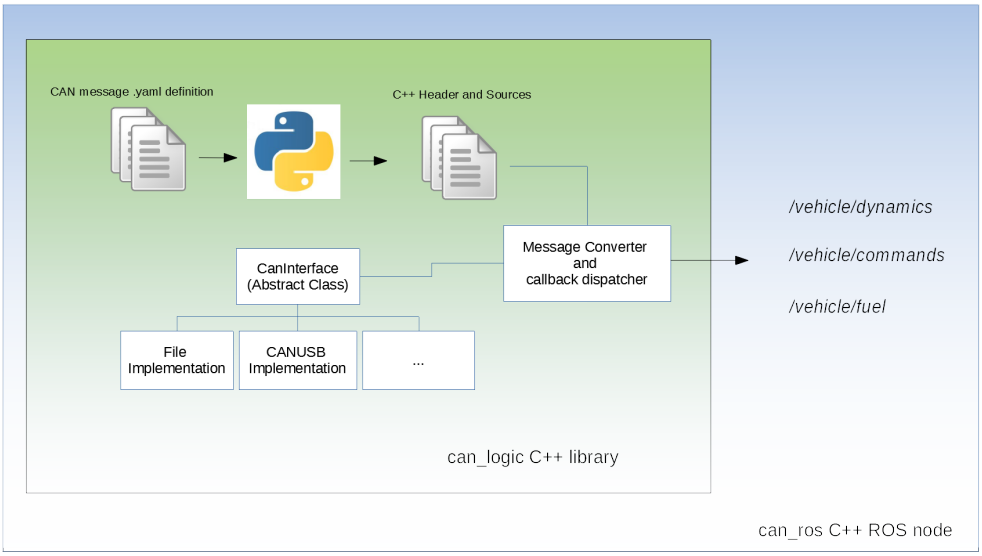


Figure 2.5: During compile time, the YAML definitions of message types are converted to C++ classes. These are then used for converting CAN messages from payload bytes. After another conversion to ROS messages, data is published over ROS topics

Since the data is available over ROS topics, it is easily used as input in the localization and perception components of the system as shown in other sections. Furthermore, this architecture can be transferred to an another vehicle in a short time by just redefining the YAML files according to the proprietary CAN format.

3 Sensors

3.1 Cameras

Image processing research has made great strides in the last years, and since the cost of cameras has gone down it has become viable to add several of them on a vehicle, capturing, if required, the full environment around the vehicle. Since cameras are the closest sensor to human vision, it is an obvious choice to use them as part of the sensory system of a self-driving car, if not the only one.

The resolution and frame rate of cameras, even commercial-grade ones, has grown considerably. Still, these sensors have many deficiencies when compared to the human eye: human vision can adapt to a vast array of scenarios and brightness conditions, focusing quite easily to both close and distant objects. Cameras cannot always guarantee this performance: brightness changes require an adaptation time (e.g. when entering and exiting a tunnel), and scenarios with high contrast can cause some invisible areas to appear too bright or too dark. Thus, other sensor types or at least redundant cameras must be considered for a robust system.

Another important characteristic of cameras is **Field-of-View** (FoV), that is, the extent of the area in front of the sensor that is projected to the output image. Usually it is expressed as a horizontal and a vertical angle, the latter being smaller since usually the images are recorded in a 'landscape' aspect ratio. Of course, the larger the horizontal angle, the farther the camera can see on the left or right side. This is especially useful in turns or in overtakes. The human eyes together have an horizontal field of view of around 210° [57], allowing to see most of the road even without rotating the neck. In order to obtain a similar result, fisheye lens or multiple cameras must be used.

Fisheye lens strongly distort the incoming light towards the focus and, when coupled with a sufficiently large sensor, they can provide up to 180° FoV. For a normal, rectilinear camera (i.e. one that presents only minor or no barrel distortions) the field of view α along one dimension (e.g. horizontal) can be calculated by knowing the size of the sensor d and the focal length f:

$$\alpha = 2 \arctan\left(\frac{d}{2f}\right)$$

Instead, the formula for fisheye lenses depends on the specific type of lens. They are distinguished by how they distorts the image: some lenses preserve angles, other surfaces. For the most common type, the equisolid, the FoV can be calculated as:

$$\alpha = 4 \arcsin\left(\frac{d}{4f}\right)$$

which turns out to be much higher for a sensor of the same size. Thus, fisheye lenses allow for a higher field of view with smaller (and possibly less expensive) sensors. However, the introduced distortions must be accounted for, as will be shown in section 3.1.

Figure 2.6: (a)

Figure 2.7: (b)

Figure 2.8: (c)

Figure 2.9: Cameras can hardly capture the details of highly illuminated and dark objects at the same time (a). For instance, the sensor can be regulated for the buildings in the back (b) or the courthouse (c). Techniques such as High Dynamic Range, however, reduce this problem. [*Wikipedia*]

The image processing techniques will be analyzed in chapter 4, section 4. In the following sections, instead, we review the sensor model.

Camera models and basic processing

Before being captured by the sensor, light is deviated by the lens system, which maps points in 3D space over the 2D surface of the sensor. It is important to know this mapping in order to interpret the camera's images and transform them into a model of the 3D environment. Specifically, we need to:

- Measure distances and angles between objects.
- Reconstruct 3D objects from multiple 2D images.
- Remove the distortion effects introduced by the lenses.

A mathematical description of each stage of projection can be obtained, from the real object's shape, to the lenses, to the sensor. Often it is possible to simplify the mathematical formulation of most stages. Knowing the general form of the equations is not enough, however. We also have to find out their parameters: for

instance, the magnitude of distortion and focal distances. Since these values are unique for each sensor and lens, the manufacturer usually do not provide them. Nonetheless, it is possible to estimate them using calibration procedures like in section 3.1.

For normal (rectilinear) lens, which only introduce minor distortions to the image (discussed later), the **pinhole camera** model is used. In this model, it is assumed that the full system of lens is replaced by a single point. This point represents the physics of a hole in the wall of a pinhole camera (or a *camera obscura*), where rays of light pass through and are projected over the sensor's surface. Since no (non-linear) distortion takes place, the point P is directly projected on point Q over the sensor depending only on its position and the focal distance (note that it is assumed that the sensor is exactly placed at the lens focus):

$$\begin{bmatrix} x_Q \\ y_Q \end{bmatrix} = -\frac{f}{z_P} \begin{bmatrix} x_P \\ y_P \end{bmatrix} \quad (2.1)$$

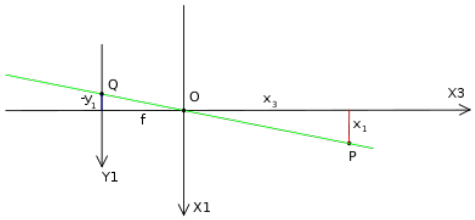


Figure 2.10: Projection formula with pinhole camera model [*Wikipedia*]

As we see, this formula is pretty simple. However, actual lens introduce notable distortions, often not immediately noticeable, but which would affect calculations. Before understanding the implications of these distortions, it is necessary to review how images are processed by the computer.

Even if many different encoding formats for images exist (e.g. JPEG, PNG), in computer vision images are processed in simple bitmap form: an image is a simple two-dimensional grid of fixed size, where each cell (named *pixel*) is associated to one or more scalar values named *channels*. Grayscale images have a single channel, an intensity value; RGB images have three, one for each elementary color (Red, Green, Blue). In the space of the image, each pixel has a coordinate pair (x, y), expressed in integer, often from the top left corner (fig. 2.11).

Cameras essentially project each 3D point to a pixel. Distortions affect the 2D coordinates of the 3D point's projection. If we later try to use the camera's images estimate the position of this 3D point, our calculations will be affected by these distortions, so they have to be taken into account. Thus, for real lenses, the previous projection formula gets more complex. We need to consider the radial distance r from the focus, and the radial and tangential distortion coefficients ($k_{1..2}$, and p_1, p_2, respectively). Also, we need to consider that the focus length on the horizontal (x) axis is slightly different from that on the vertical (y) axis:

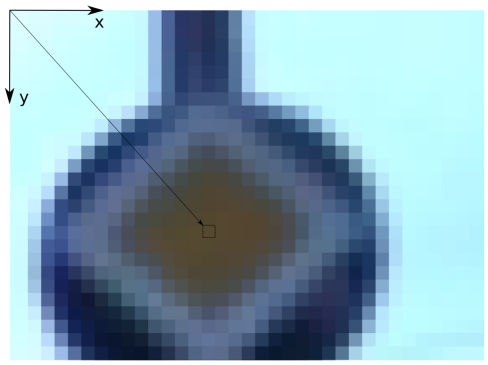


Figure 2.11: (a)

Figure 2.12

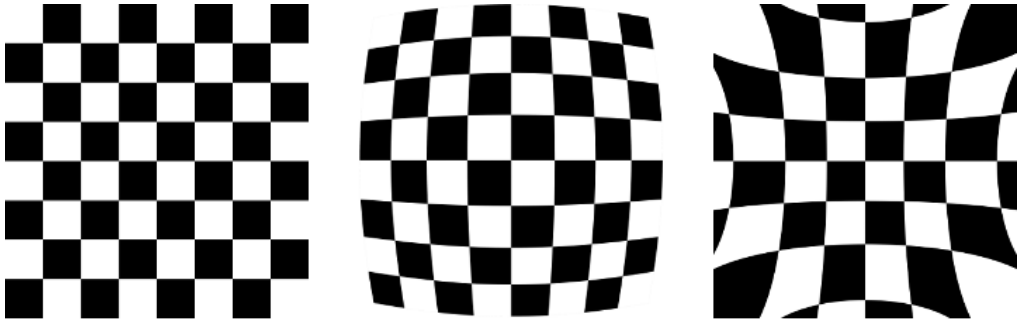


Figure 2.13: Distortion examples [*OpenCV documentation*]

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} x' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + 2p_1 x' y' + p_2(r^2+2x'^2) \\ y' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + p_1(r^2+2y'^2) + 2p_2 x' y' \end{bmatrix}$$

where

$$\begin{aligned} x' &= \frac{x_P}{z_P} \\ y' &= \frac{y_P}{z_P} \\ r^2 &= x'^2 + y'^2 \end{aligned} \tag{2.2}$$

Fisheye have of course a different model, discussed in the next section along with the calibration procedure.

The camera we use is a UI-3060CP from Imaging Development Systems GmbH (abbrev.: iDS), a wide-angle industrial camera with USB 3.0 interface:

Table 2.1 Characteristics of the UI-3060CP camera

Sensor	CMOS IMX174LQJ-C
Resolution	1936x1216
Optical size	11.345 x 7.126 mm
Focal length	4mm
Field of view	126°
Maximum frame rate	166 frames per sec

The ROS driver for this camera is already available[1]. Frame rate was limited to 7 or 12 fps since the conversion and storage process by itself is a quite CPU- and disk-intensive task.

Camera Calibration

For fisheye cameras, we use the model defined in [23]. This model was added to the recent OpenCV 3 library and it is probably not yet mature enough for production usage. Indeed, we obtained sufficient but not completely satisfying results.

The standard calibration procedure for a camera consists in obtaining multiple pictures of a known model from different positions, angles and scales. Usually the model is a checkerboard or a object that is easily recognizable and has a pattern known with millimeter accuracy. In the case of the checkerboard, we printed it on an A3 paper with a predetermined square length. The simple black and white geometry allows the OpenCV library to easily detect the checkerboard even in the presence of cluttered backgrounds. For a perfect pinhole, the corner points of the checkerboard should have a known projection over the 2D image, given by eq. (2.1) but due to non-linear distortions the corners are slightly offset from their expected positions. The calibration algorithm tries to find the distortion parameters that best justify these offsets overall on the collected images, along with linear intrinsic parameters such as focal length and image center. Once these parameters are known, it is possible to apply the inverse distortion formula in order to cancel the distortion, or as it is known, **rectify** the image. Parameters are calculated by using nonlinear least squares, with **reprojection error** as the cost function to be minimized. This error is the RMS of the 2D Euclidean distance between the detected and predicted positions of the corners, measured in pixels. Usually, in a correctly calibrated camera, this value should be less than 0.5 pixels.

[1] http://wiki.ros.org/ueye_cam

Our calibration script takes as parameter an input directory where images are stored, the size of the squares in the checkerboard pattern, and a maximum number of images to use. It is possible to specify an output directory where detected checkerboard frame are rectified, in order to evaluate qualitatively the results of the calibration.

Listing 2.4: fisheye calibration command

```
./fisheye_calibrate --pattern_size=9x6 --square_size_mm=30
   --debug_output_dir ./debug_output_imgs --max_input_images 250
   --input_dir 2017-11-16-calibration_pngs/
```

The script generates a calibration matrix. The parameters should be recomputed every time the experimental setting changes (e.g. different camera or lens), so here we report one of the results we generated during the tests.

$$K = \begin{bmatrix} 661.61057476 & 0.0 & 965.86940897 \\ 0.0 & 662.15425078 & 614.68159879 \\ 0 & 0 & 1 \end{bmatrix}$$

$$P = \begin{bmatrix} 661.61057476 & 0.0 & 965.86940897 & 0.0 \\ 0.0 & 662.15425078 & 614.68159879 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \end{bmatrix}$$

$$D = \begin{bmatrix} 0.02066096 & 0.00113071 & 0.00234002 & -0.00137322 \end{bmatrix}$$

(a) (b)

Figure 2.14: Example debug output, before and after rectification

3.2 Lidar

Among the technologies that have driven the latest ten years of Autonomous Vehicle research, lidar is for sure the most important one. Lidar sensors directly solve the problem of reconstructing 3D geometries around the vehicle, with a high speed and resolution, at the expense of higher sensor cost, which is nonetheless constantly decreasing over the years.

While they can be implemented using many different technologies (range gated sensors, RF modulation, etc.) current automotive lidar devices are based on the principle of pulsed laser Time-of-Flight: they emit a short (in the order of nanoseconds) **laser pulse** that is then reflected by an external surface back to the sensor (fig. 2.15). Travel time is accurately measured, and used to calculate the distance travelled by light. Multiple such laser emitters and receivers are mounted on a frame rotating on one axis (fig. 2.16). Thus, a 360° point cloud of the environment is generated. In order to further reduce the cost and number of moving parts of the system (and therefore its fragility), lidar manufacturers are developing solid state lidars based on **MEMS** or **phased array optics**. Such sensors will probably replace the existing ones within years, making mass-production possible and thus enabling use of lidars in retail vehicles.

Table 2.2 Comparision of lidar devices in automotive. Prices are being reduced year by year, especially when many units are produced. For instance, the VLP-16 reached a 4000$ price in 2018, half of the original price

Product	Producer	Year of release	Price at release (estimated) ($)	N. of rays
HDL-64	Velodyne	2010	80000	64
HDL-32	Velodyne	2010	30000	32
VLP-16	Velodyne	2015	8000	16
M8	Quanergy	2017	6000	8

Usually these sensors generate point cloud data, along with intensity information, in 3D space. Intensity depends on the surface material, incidence angle of the laser and lightning condition of the environment, so it may vary for the same object. Still, it is an important datum that can allow to detect lane markings.

Two lidars were used on the vehicle: a Velodyne VLP-16 (also known as PUCK) and a roadview automotive lidar. These sensors, as many other lidars, have a similar working principle and differ mostly for number of lasers, field of view, accuracy and range.

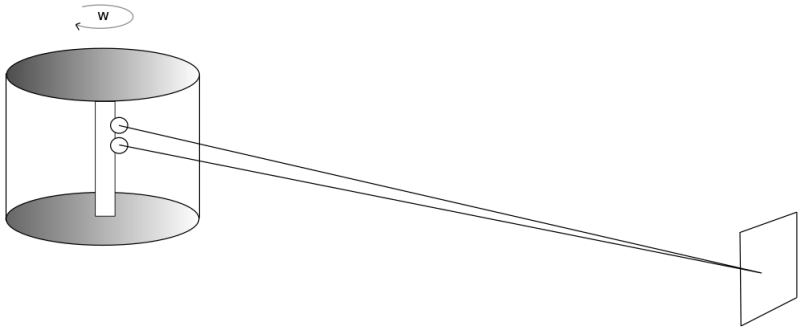


Figure 2.15: Laser diode and receiver are mounted on a rotating frame. Distance of measured point is given by $\Delta t/2 \cdot c$

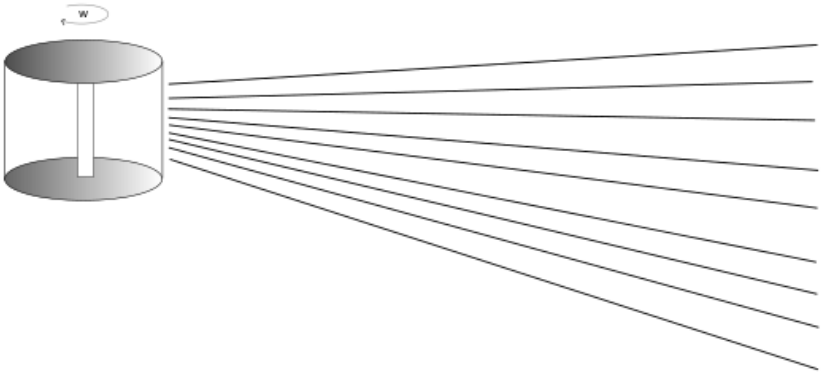


Figure 2.16: Multiple laser distributed vertically allow to obtain almost a fully spherical surface. Sensor usually provide points measurements in spherical coordinates.

3.3 Future trends

Rotating a laser accurately while performing range measurements requires high-precision manufacturing. Small errors in angle can generate high inaccuracies in the position estimation of the reflected object. Furthermore, a device that continuously spins is bound to lower durability and higher chance of failure. A normal user expects his car to last several years while preserving most of its reliability. Thus, even if the sales of current generation lidars are increasing worldwide, it is clear that they will never reach mass production for standard consumers.

Big lidar companies and new startups are focusing nowadays on building the next generation of these sensors using new principles. Generally, these sensors are called **solid state lidars** since they do not include rotating components. The complexity of the product is reduced to the single integrated circuit. Since the

cost depends mostly on the research for the optimal manufacturing process and production costs can be quite low in mass production scenarios (exactly like today happens for processors) it is expected that these new sensors will reach a cost of 1000$ or even less. Such components will probably soon be added on many medium-high price range vehicles along with radars in the near futures. Companies are betting on different technologies for achieving this results:

MEMS lidars This solution exploits the use of microelectromechanical systems (MEMS) for moving the rays instead of a classical rotating device. Infineon's lidar for instance moves a tiny mirror for directing the laser, controlling its inclinations electrically. Caption and other companies follow a similar path.

Optical Phased Array These sensors are quite similar to the old phased array radar: by accurately controlling the phase of multiple elements in a row it is possible to generate a coherent ray along only one direction, while other directions are destructed due to interference. This is performed along two directions in the sensor produced by Quanergy. Beyond the mentioned advantages, since the elements are controlled by software it is possible to steer the ray in many patters, obtaining a custom field of view or even random access on a restricted angle.

Hybrid approaches Distributing array elements along two dimensions requires a number of elements proportional to the square of the sensor's width, thus increasing costs. Some solutions [21] use OPA only for one direction, while the second direction is obtained separately using lenses that deviate the ray depending on the wavelength, which is purposely controlled.

3.4 Point Clouds

A **point cloud** is particular type of 3D computer image: while a 2D image is a set of value (pixels) densely and regularly distributed on a 2D rectangle, a point cloud is a simple set of points in $\mathbb{R}^3$, often generated by a lidar or other techniques such as stereoscopy or photogrammetry. In an image, the channels are often the Red, Green, and Blue color intensities as sensed by a CMOS or CCD sensor. In a point cloud instead we have a single intensity value, proportional to the energy of the received reflected pulse. Some 3D sensors are capable of also collecting color data with various solutions; those are called **RGB-D** sensors, where D stands for depth. Such sensors, however, are rarely used in automotive applications so they will not be treated here.

Usually, point clouds are stored in memory as an array of tuples, where each tuple represents a different point and has at least three floating point elements for the absolute position in Cartesian coordinates, plus one more for each channel. So, in most situations there is a fourth value for the point's intensity.

One of the main differences compared to camera images is the fact that, being distributed in 3D space, point clouds are intrinsically 'sparse', that is, there are many empty volumes within the boundary of a scan, while in an image we do not have missing pixel at any coordinates. Nonetheless, in the case of lidar and similar sensors, *no data means data*: if a pulse travelled from the sensor to the reflected object, we know not only that there is an object there, but we can also suppose that there are no bodies along the path of the pulse (ray). We can use this absence of data for reconstructing empty spaces. Applications of this observation in the automotive world include detecting available lanes or free parking slots.

More details on point clouds and their processing methods can be found in chapter 4 section 2.

Velodyne VLP-16

The smallest of VLP family, PUCK (fig. 2.17) offers 16 laser over 30°, making it suitable for road applications even if not as powerful as the 64 or 32 lasers version. Accuracy is around 3 cm, range up to 100 m. Being it symmetrical, it is preferably mounted on top of the car roof, close to the center. Sensor data is sent through a UDP stream over an Ethernet connection. Intensity value of each laser sweep is sent along with its azimuth value (altitude is fixed for each laser). It is thus possible to reconstruct the 3D point cloud starting from the collection of intensity values expressed in spherical coordinates. This final conversion is performed by software.

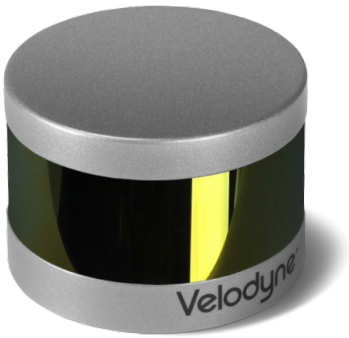


Figure 2.17: Velodyne VLP-16. A cost-effective rotating lidar for various robotic applications.

It is possible to configure the sensor rotation speed and return mode (i.e. a policy for selecting one or more among multiple reflection for the same laser channel). Throughout the thesis, the sensor was used in *strongest* mode, with 20 Hz (1200 RPM) rotation. No software was written for acquiring data from this sensor, since

the existing ROS driver `velodyne`[1] provided all the required functionality.

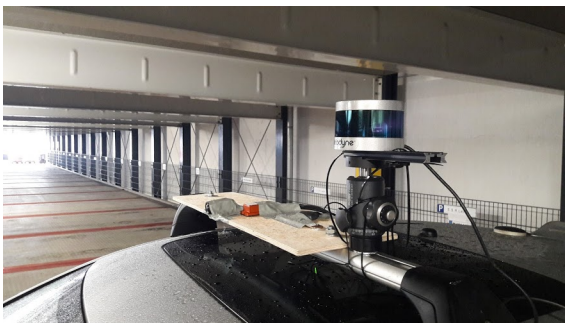


Figure 2.18: The VLP-16 mounted on top of the vehicle

Automotive Lidar

Purposely for automotive use, these kinds of lidars are mounted at bumper's height and around the vehicle (usually up to 4 devices; see fig. 2.19). These lidars have a small FoV, a reduced number of lasers and a range shorter than 100m. The data from each lidar is sent through a TCP connection, and it can be processed in real time on a computer, that joins the inputs from all the sensors under a unique interface.

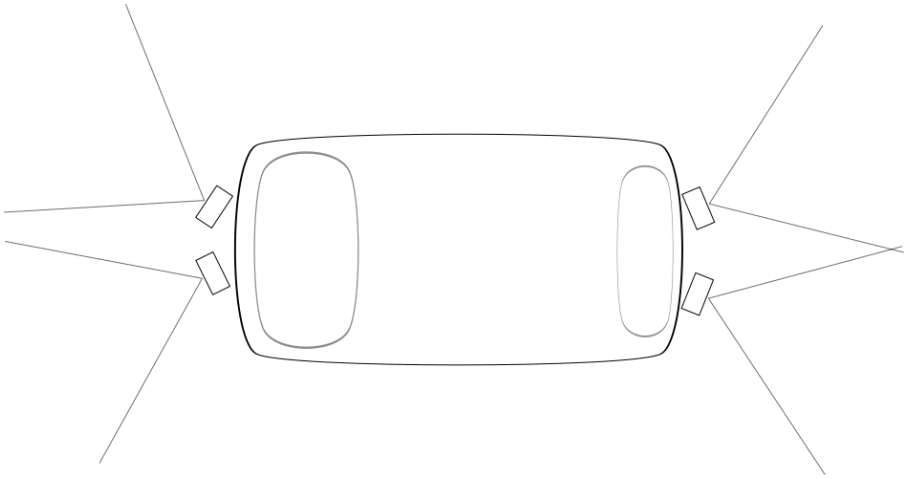


Figure 2.19: Position of the four automotive lidars around the vehicle. Almost 360°coverage is obtained by joining the data from all the sensors.

At the time of writing, there is no open-source ROS driver for this sensor, so we wrote our own solution. We used a C++ SDK that takes care of retrieving

[1] https://github.com/ros-drivers/velodyne

data from the sensors through the TCP connection and deserializes it. On top of the SDK, we implemented a ROS layer that publishes point cloud data and object detection results.

Listing 2.5: Excerpt from the automotive lidar ROS driver. This functions converts a point cloud from the SDK to ROS' PointCloud2 format

```
sensor_msgs::PointCloud2 toPointcloud2Msg(
    const NewScanData* scan) {
  sensor_msgs::PointCloud2 msg;
  msg.header.stamp = toRosTime(getStamp(scan));
  msg.header.frame_id = std::string("bumper_lidar");
  msg.height = 1;
  msg.width = scan->points.length();
  msg.data.reserve(scan->points.length()*
    sizeof(int32_t)*4);
  msg.is_bigendian = false;
  msg.point_step = sizeof(int32_t) * 4 + 1;
  msg.row_step = msg.point_step * msg.width;

  msg.fields.push_back(makePointField("x", 0,
    POINT_FIELD_FLOAT32, 1));
  msg.fields.push_back(makePointField("y", 4,
    POINT_FIELD_FLOAT32, 1));
  msg.fields.push_back(makePointField("z", 8,
    POINT_FIELD_FLOAT32, 1));
  msg.fields.push_back(makePointField("rgb", 12,
    POINT_FIELD_UINT32, 1));
  msg.fields.push_back(makePointField("intensity", 16,
    POINT_FIELD_UINT8, 1));
  msg.is_dense = true;

  for(const auto& point : scan->points) {
    pushFloat32(msg.data, point.x);
    pushFloat32(msg.data, point.y);
    pushFloat32(msg.data, point.z);

    uint32_t color = chooseColorForPoint(point);

    pushUInt32(msg.data, color);
    msg.data.push_back(point.intensity);
  }

  return msg;
```

```
}
```

Lidar error sources and problems

The accuracy of the sensor depends on **construction quality** (small angle errors of the emitted laser may be important on long distances) and **rotation measurements quality**. These kinds of errors are often reduced by the manufacturer by providing post-production calibration data unique to each unit.

Another source of error is **multipath effect** (or *multipath interference*), which may cause distance overestimation due to the laser traveling a longer distance, like in GNSS (See section 3.5 for a detailed explanation of the phenomenon).

These kinds of errors can cause mismeasurements of few centimeters. While important in applications such as geographical survey or 3D object scanning, we believe they are not so important since the behavior of all algorithms depends on sets of dozens or hundred of points, where errors can be filtered out.

However, in our particular case, we had to pay attention to errors due to **vibrations and rotation** of the VLP-16 lidar. Since it rotates at high speed (1200 RPM), when not mounted correctly it follows a precession movement, which causes a strong radial distortion on distant points. After unfortunately noticing this problem during dataset analysis, we opted for manually filtering out points with distances greater than 45m.

Lidar works with the assumption that line of sight between the sensor and the reflecting object is free. This is not always the cause. **Transparent or mirror-like surfaces** often cause unpredictable effects on the final point cloud. Also, they are most likely not be detected. The F.A.Q. on the Velodyne website[1] states that **rain and snow** are not a big problem for lidars. On our trials however, we observed how a mild-strong snow makes the VLP-16 mostly unusable, since it detects snowflakes around the vehicle. We assume then that the Velodyne statements can be applied only to 32 or 64 laser models.

Black surfaces can almost totally absorb the incident laser light. We noticed that on long distances opaque, black cars are poorly visible for the sensor.

These results tells us that a lidar-only solution seems impractical for autonomous vehicles, and that the support of cameras is probably essential, since it can always work at least as a fallback.

[1] http://velodynelidar.com/faq.html

3.5 GPS

Since localization is one of the main topics of this thesis and self-driving vehicles in general, we will review here the basic principles and related problems.

A Global Navigation Satellite System (**GNSS**) is a positioning system that exploits the use of multiple satellites, equipped with a high accuracy clock, that continuously transmit their own position along with a transmission time (**TOT**). A receiver on the ground that can sense the signal of at least four satellites, can triangulate its own position by estimating the time of flight of each satellite signal (fig. 2.20).

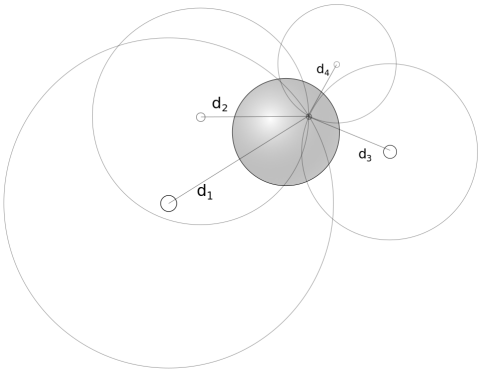


Figure 2.20: Triangulation of a GPS receiver. The ToF of each signal is estimated by the receiver.

Three signals are not enough since time is a variable too, due to the fact that the receiver cannot have an high accuracy clock. Thus, the Time of Arrival (**TOA**) is affected by the time skew t_s of the receiver.

Ignoring possible corrections, in their simplest forms the equations for calculating receiver position are the following:

$$
\begin{aligned}
c \cdot (t_{TOT,1} - t_{TOA,1} + t_s) &= \sqrt{(x_1 - x)^2 + (y_1 - y)^2 + (z_1 - z)^2} \\
c \cdot (t_{TOT,2} - t_{TOA,2} + t_s) &= \sqrt{(x_2 - x)^2 + (y_2 - y)^2 + (z_2 - z)^2} \\
c \cdot (t_{TOT,3} - t_{TOA,3} + t_s) &= \sqrt{(x_3 - x)^2 + (y_3 - y)^2 + (z_3 - z)^2} \\
c \cdot (t_{TOT,4} - t_{TOA,4} + t_s) &= \sqrt{(x_4 - x)^2 + (y_4 - y)^2 + (z_4 - z)^2}
\end{aligned}
$$

Taking into account clock errors, receiver can solve by using least squares or other optimization methods.

GNSS Technologies, performances and source of errors

The ubiquitous GNSS technology, GPS, is in wide use since 1995. Since then, GLONASS (Russian), BeiDou (Chinese) and the most recent Galileo (European) were launched. The operating principles are almost identical, so that nowadays one receiver can easily support all four systems at the same time. All of them offer encrypted and public frequencies, with precision varying from 0.01 to 10 m. Unfortunately, the accuracy of a GNSS can get much worse in many cases. Indeed, many error sources may affect the evaluation of distance from the receiver to one satellite. The most importance sources will be listed here:

- **Propagation in troposphere and ionosphere**
  Speed of light is not constant. Weather conditions affect the travel time of light through troposphere and ionosphere, thus introducing an error in the previous equations.

- **Multipath effect**
  This error affects the actual traveled distance of the signal, and is the most important in urban scenario. GPS signal may bounce between buildings before reaching the receiver, which, even if it will try to reject low SNR signal, will perceive a fake distance between it and the satellite. This can introduce errors directly proportional to the reflected path.

- **Satellite position inaccuracies**
  The position of satellites, as accurate as it can be, is still affected by error.

Since the receiver uses a least-squares technique, the availability of more than one GNSS signal improves accuracy. However, even with the new Galileo system, accuracy cannot be better than one meter due to the previously listed problem. Further improvements may be obtained through the use of new techniques that exploit the use of a ground station.

Differential GPS Errors due to signal propagation in the troposphere and ionosphere and inaccuracies in Satellite ephemeris are highly correlated for close points on the surface of the earth. This means that two GNSS receivers will be equally affected by approximately the same error when they are close. The idea of Differential GPS (**DGPS**) is to use one or multiple fixed reference stations whose position is known with centimeter level accuracy. The reference station, knowing its position, is able to evaluate the difference between actual and GNSS perceived position and can thus provide correction information to other GNSS receivers. Data is usually locally broadcast using a radio link. A mobile GNSS receiver applies the correction obtained from the DGPS station, thus obtaining an accuracy of dozens of centimeters.

Real Time Kinematics Centimeter level accuracy can finally be obtained thanks to Real Time Kinematics (**RTK**) sensors. Such sensors use more advanced electronics and software for obtaining the position not only by the data within the GNSS signal, but also from the phase-shift of the carrier wave. This means that the receiver can theoretically differentiate distance that are a fraction of the carrier wavelength. For instance, for the Galileo E1 signal (1575.42Hz), the wavelength is 19cm. The software complexity relies in finding out the right distance among the possible multiple of the wavelength. This is possible using statistical method that take the original GNSS signal as prior and, as in DGPS, correction from the base station.

GNSS in automotive

While these technologies were indeed successfully proven and used in research or land survey, their application in the automotive world seems impractical since that would require the support of ground stations every few kilometers. Moreover, for an autonomous vehicle, it is more important to know its position with relation to the road that its absolute latitude and longitude. Thus localization techniques based on odometry and HD maps are more promising.

Thus, RTK GNSS are used for research for ground truth data or during the creation of HD map, since the high infrastructure costs are not high for a few units.

The series GPS in a car can be fairly inaccurate, with error usually around 1 to 20 meters. In cities, accuracy further decreases due to strong multipath effects caused by buildings.

GPS sensors on the platform

As stated previously, we can retrieve GPS data from the vehicle using the CAN network. Unfortunately, no accuracy nor SNR information is available. Ground truth data is provided by an AHRS system (MTi-G-710 fig. 2.21). This sensor, other than receiving a GPS signal at 100Hz, uses an internal Kalman filter for improving position estimation. We used the Automotive software scenario for the internal Kalman filter. A ROS package for this sensor is already existing and provided by ETH Zurich[1]. This package also provides accuracy data and raw accelerometer values.

Both the MTi-G-710 and its antenna were mounted on the vehicle's roof, close to the center in order to reduce centrifugal effects.

[1] https://github.com/ethz-asl/ethzasl_xsens_driver

Figure 2.21: XSens MTi-G-710

3.6 Wheel Odometry

We do not use a specific sensor for wheel odometry, but it should be noted that the longitudinal speed coming from CAN data is expressed with the original vehicle wheels rotation as the zero value. We have to remind that, by itself, wheel odometry is not a good method of localization because, being a form of dead-reckoning, its accuracy decreases with distance, especially along the lateral axis. Moreover, cars in realistic scenarios are strongly affected by slip, though the use of all-four wheel speed sensor help averaging out this source of error [6].

See [6] and [25] for a report of odometry performance. Albeit error strongly varies depending on path, the error in our first-hand experience has been of a few dozens meters after a few kilometers when using just dead-reckoning.

However, when embedded within a Kalman filtering framework, odometry can be a quite good source of data, as we demonstrate in section 6.

The advantage of this technique is that it can be implemented on all cars, since all that is needed is the data from the ABS speed sensor. Such sensors are already high quality, since they are designed for stability control and anti-lock braking applications.

In its most basic version, an example of odometry data integration is reported in listing 2.6.

Listing 2.6: Simple integration (dead reckoning) of odometry data for calculating 2D pose

```
def integrate_vel(velocity_m__s, yaw_radian, dt_s):
    mag = velocity_m__s * dt_s
    return np.array([mag*np.cos(yaw_radian),
                     mag*np.sin(yaw_radian)])
```

```
# callback called when new odometry msg is available
def callback(self, msg):
    dt_s = (msg.header.stamp -
      self.last_msg_stamp).to_nsec() / 1.0e9
    self.yaw += np.deg2rad(msg.yaw_rate_deg__s) * dt_s
    self.pos += integrate_vel(msg.velocity_m__s,
                  self.yaw, dt_s)
```

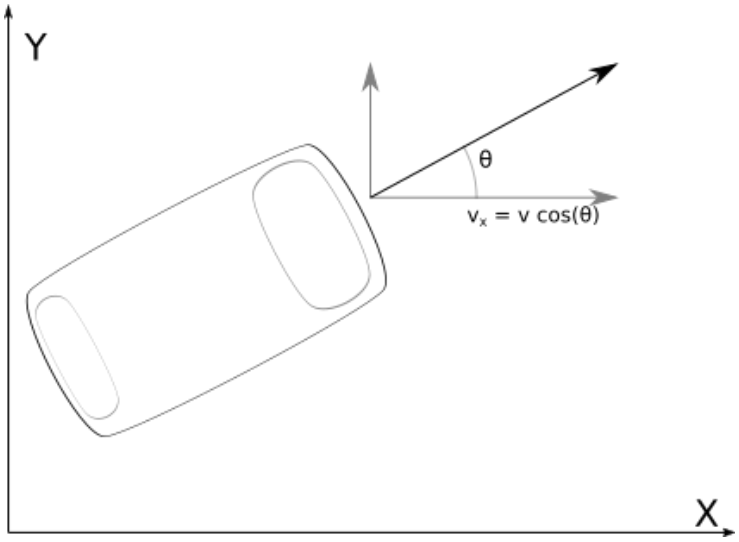


Figure 2.22: Dead reckoning of the vehicle, with speed and yaw rate given by wheel rotations

3.7 Summary of the platform

In fig. 2.23 we report the components involved and their connections. The xsens MTi-G-710 IMU, its GPS antenna, the VLP-16 and the iDS uEye USB camera were mounted on the front carrier rack (see section 3.7). In order to reduce the effects of centrifugal forces, which would cause noise on IMU readings, it would be optimal to position the IMU close to the center of gravity of the vehicle. It was not possible to do this, but the position is still quite aligned to the center. The front camera position allows to have a full front view, partially obstructed by the car's front hood. The camera is centered horizontally and parallel to the ground. The VLP-16 is also parallel to the ground and positioned a bit on the left.

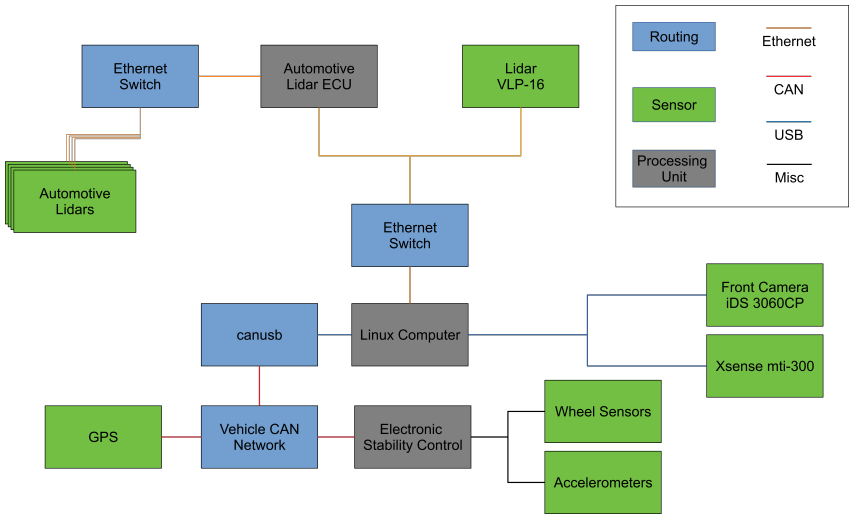


Figure 2.23: Major components of the system

(a)

Figure 2.24: Camera, GPS antenna and xsens IMU mounted on top of the vehicle.

Chapter 3

Localization

One of the most fundamental functions of a robot that is able to navigate its environment is the ability to observe and "learn" its surroundings, and estimate its own position and orientation. The first is commonly known as *mapping*; the latter as *localization*. These two basic problems can be solved separately and independently, or come together to be solved in lockstep, in the formulation known as *SLAM* (Simultaneous Localization And Mapping).

In this thesis, we have investigated options for both mapping and localization. We provide a few notes on the theoretical underpinnings of the problem and the analyzed solutions, and then provide a description of the evaluated options. In the end, we present the final results, and our final choice of software.

1 Problem and approach

Just like most complex organisms, a robot will need to observe and study its environment, and it will do so by processing information through dedicated measuring devices called sensors. Out of this information, the robot will then estimate its own position and orientation relative to a map of the surrounding environment. The problem can be mathematically expressed as the search for:

$$p(x_t|u_t, ..., u_0; , x_{t-1}, ..., x_0; m_t) = p(x_t|u_t, x_{t-1}, m_t) \tag{3.1}$$

This task is valuable in and of itself, and it is a fundamental input for many other functions in the system. Depending on the particular functions that need to be performed, a certain degree of accuracy will be required out of this resulting estimate.

It's worth noting that much of the motivation for this formulation of the problem lies on the fact that there is no "perfect" absolute positioning device that yields a pose estimate that is sufficiently accurate, recent, and reliable in every possible environment, for all of the typical functions of an autonomous vehicle. Typical

examples of such demanding functions are collision avoidance with other users of the road, maneuvers such as lane change and parking, and autonomous navigation in indoor (or roofed) environments.

A modern GPS device will come close to such a device, but still not enough. Moreover, all sensors will have a certain degree of *sensor noise*, usually explicitly declared and quantified by the manufacturer. Another phenomena that robots often have to deal with is *sensor aliasing*: this is the general situation where two different ground truths are read as the same by a given sensor; in other words, whenever two places "look just the same" (to the "eyes" of that sensor). Movement also generates another source of noise, named *effector noise*, due to the fact that the command coming from the computer is applied with a certain degree of uncertainty, and that the environment itself will respond to the movement in unexpected (unmodeled) ways (e.g. slippery or uneven road, different terrain types...).

Since localization is still of the utmost importance, the general strategy adopted by most robots is **data fusion**: readings from a variety of sensors, possibly based on different physical phenomena, are all included in the same mathematical framework with the objective of producing an estimate that is more accurate than any single input. In the case of a vehicle-like robot such as our research car, the most popular types of sensors are GPS navigation devices, cameras and LIDARs. (Details on our particular setup can be read at section 3.)

The problem can therefore be further split in two "phases": first the readings from each sensor are all transformed into a localization estimate; then, these estimates are all collected and integrated in a single estimate that constitutes the final *belief* about the robot's own position and orientation.

Regarding the first half, different input data types call for different approaches. We've tried out both algorithms that understand images (from cameras) and point clouds (arrays of points coming from LIDARs or Time-of-Flight cameras). GPS readings also need to be converted from a geographical frame of reference (e.g. a latitude-longitude coordinate) into a local one. This conversion is also used to "anchor" the final result to a geographical location (i.e. generally, position tracking is performed, but the global localization problem is also solved to a degree).

Finally, the final integration of all sensor-specific estimates is performed by a Kalman filter implementation dedicated to this particular task.

Considering the diverse tasks that use the localization estimate as input, different estimates are produced each with a different error profile. We have a *local* estimate, which is based on local measurements only (speed, acceleration, rate of turning), is only valid in the near surroundings of the vehicle and accumulates error over the long distance, but never presents any discontinuity across time steps; and then we have a *global* estimate which also integrates GPS readings, which causes it to converge automatically to more correct estimates on a geographical scale, at the cost of the occasional apparent "jump" (time discontinuity) on the map.

A third estimate comes from SLAM, a slightly different problem solved by different algorithms. This estimate is anchored to a 3D map of the environment produced simultaneously by the same algorithm. This one is a more adequate input for cases where e.g. the vehicle needs to interact with the environment or closely watch obstacles. This family of algorithms is explained in section 3.

2 Theory notes: Bayesian algorithms and Kalman filtering

Reading the problem description presented in the previous section, it is evident that the core of the issue is that the robot's "view" of the world is always incomplete, and subject to error-generating phenomena such as sensor noise and aliasing. The mathematical tool for dealing with such partial knowledge is probability theory, which means that our final resulting estimate will also have a margin of uncertainty. In other words, the robot will have (and update) a *belief* about its own position and orientation, which will hopefully be precise enough for its mission.

The most general tool for iteratively updating a belief based on new information (or measurement) is the *Bayes filter*[1], also known as *recursive Bayesian filtering*. Keeping in mind that time is considered discrete throughout, this algorithm makes two basic assumptions:

- **Markov assumption**: the current state (x_t) only depends on the state at the previous time (x_{t-1}) and the most recent *control* (u_t):

$$p(x_t|u_t, ..., u_0; , x_{t-1}, ..., x_0) = p(x_t|u_t, x_{t-1}) \tag{3.2}$$

- The system is a **Hidden Markov Model**: at time t it has state x_t but we are only able to "read" measurement z_t. The system is therefore modeled in terms of a *state transition probability* ($p(x_t|x_{t-1})$) and *measurement probability* ($p(z_t|x_t)$).

Under this model, the algorithm can be formulated like the pseudo-code at algorithm 1. For better comprehension, consider the example where the system under estimation is a simple robot, commanded by velocity, where the state, command, and state transition laws are as following:

$$\mathbf{x_t} = \begin{bmatrix} x \\ y \end{bmatrix} \qquad \mathbf{u_t} = \begin{bmatrix} x' \\ y' \end{bmatrix} \qquad \mathbf{x_{t+1}} = \mathbf{x_t} + \mathbf{u_t}\,\Delta t$$

[1] Where not otherwise explicitly mentioned, the bibliographical source for the entire section is [56]

The procedure (see algorithm 1) is composed of two steps:

- The *prediction step*, where the system under estimation is simulated for one time increment, supposing that it is being subject to our control/command u_t. The computation is based on the state transfer probability $p(x_t|u_t, x_{t-1})$ that models how the state can evolve from the last estimate, under the command.
- The *update step*, where the state estimate produced at the prediction step is corrected in order to reconcile it with the new measurement z_t just acquired. The likelihood of the measurement, coming from the system model, is utilized during this step. The constant η simply scales the product to the $[0, 1]$ range, in order to convert it into a valid probability value.

This procedure is called at every time increment, supplying the last update of the system state belief $\hat{x}_{t-1}$, and the latest value of the control and measurement signals. The Bayes filter is the most general tool for keeping and updating beliefs about any system that satisfies the two aforementioned assumptions, since it is valid for any probability distribution of the system state belief.

A variant of the Bayes filter that is particularly relevant for robot localization is called **Markov localization**, and is listed at algorithm 2.

Algorithm 1 The Bayes filter, or recursive Bayesian filtering algorithm.

function BayesFilter($\hat{x}_{t-1}, u_t, z_t$)

▷ Prediction step

$\hat{x}_t^* \leftarrow \int p(x_t|u_t, x_{t-1})\, \hat{x}_{t-1}\, dx$

▷ Update step

$\hat{x}_t \leftarrow \eta\, p(z_t|x_t)\, \hat{x}_t^*$

return $\hat{x}_t$

end function

Algorithm 2 Markov Localization, a variant of the Bayes filter with a map as an additional input.

function MarkovLocalization($\hat{x}_{t-1}, u_t, z_t, m$)

▷ Prediction step

$\hat{x}_t^* \leftarrow \int p(x_t|u_t, x_{t-1}, m)\, \hat{x}_{t-1}\, dx$

▷ Update step

$\hat{x}_t \leftarrow \eta\, p(z_t|x_t, m)\, \hat{x}_t^*$

return $\hat{x}_t$

end function

The Markov Localization algorithm has one extra input m compared to the basic Bayes filter. This represents any sort of "environment memory" kept by the

Algorithm 3 EKF localization, a special case of Markov localization that can be concretely implemented by virtue of using an Extended Kalman Filter to provide for the probability estimates. It assumes that (1) state and output variables can be modeled by Gaussing variables, and (2) linearization yields a sufficiently good approximation for the state and output transfer functions.

function EKFLocalization($\mu_{t-1}, \Sigma_{t-1}, u_t, z_t$)

▷ Prediction step

$\bar{\mu}_t \leftarrow g(u_t, \mu_{t-1})$
$\bar{\Sigma}_t \leftarrow G_t \, \Sigma_{t-1} \, G_t^T + R_t$

▷ Update step

$K_t \leftarrow \bar{\Sigma}_t \, H_t^T \, (H_t \, \bar{\Sigma}_t \, H_t^T + Q_t)^{-1}$
$\mu_t \leftarrow \bar{\mu}_t + K_t \, (z_t - h(\bar{\mu}_t))$
return μ_t, Σ_t
end function

program. Examples for such a memory can be a database of locations indexed by their appearance (for vision-based location recognition), or a large-scale point cloud of the environment (for lidar-based implementations). The formulation of the algorithm given in algorithm 2 is purposely abstract: as long as the program keeps any kind of memory of past visited locations and possesses a system model (a formulation of $p(x_t|u_t, x_{t-1}, m)$ and $p(z_t|x_t, m)$) that takes it into consideration, it can be considered an implementation of the Markov Localization algorithm.

One practical, concrete design of Markov localization that is popular and useful in real robots is **EKF localization**, named after its use of an *Extended Kalman Filter*. The Extended Kalman Filter estimates the state and output of a dynamical system modeled after the following system of equations (assuming the case of discrete time):

$$\begin{aligned} x_t &= g(u_t, x_{t-1}) + \epsilon_t \\ z_t &= h(x_t) + \delta_t \end{aligned}$$

where g and h are the state and output transfer function respectively. The functions g and h do not have to necessarily be linear, but need to be known and computable point-wise, together with their first derivative. The error terms (ϵ, δ) are assumed normally distributed. The symbols u_t, x_t, and z_t represent the measurements, state, and output vectors respectively. The EKF works by assuming a Gaussian distribution for state and output posterior estimates and error terms, and approximating g and h by their first-order Taylor expansion (i.e. by linearization):

$$\begin{aligned} x_t &\sim \mathcal{N}(\mu_t, \Sigma_t) \\ z_t &\sim \mathcal{N}(\bar{\mu}_t, Q_t) \\ g(u_t, x_{t-1}) &\approx g(u_t, \mu_{t-1}) + G_t \cdot (x_{t-1} - \mu_{t-1}) \\ h(x_t) &\approx h(\bar{\mu}_t) + H_t \cdot (x_t - \mu_t) \end{aligned}$$

where G_t and H_t are the Jacobian matrix of g and h, respectively. Note how μ_t is used as the best current guess for x_t and as origin point for localization. As such, the model is approximated by the following equations:

$$\begin{aligned} x_t &= g(u_t, \mu_{t-1}) + G_t \cdot (x_{t-1} - \mu_{t-1}) + \epsilon_t \\ z_t &= h(\bar{\mu}_t) + H_t \cdot (x_t - \mu_t) + \delta_t \end{aligned}$$

The EKF *algorithm* is a special case of the Markov localization algorithm that keeps a local posterior estimate of the state as a sequence of Gaussian variable, and uses the EKF formulation to provide expressions for $p(x_t|u_t, x_{t-1})$ and $p(z_t|x_t)$:

$$\mathbf{v} = x_t - g(u_t, \mu_{t-1}) - G_t \cdot (x_{t-1} - \mu_{t-1})$$
$$p(x_t|u_t, x_{t-1}) = \sqrt{\det(2\pi R_t)}\, \exp\left(-\frac{1}{2}\mathbf{v}^T\, R_t^{-1}\, \mathbf{v}\right)$$

$$\mathbf{w} = z_t - h(\bar{\mu}_t) - H_t\,(z_{t-1} - \bar{\mu}_t)$$
$$p(z_t|x_t) = \sqrt{det(2\pi Q_t)}\, \exp\left(-\frac{1}{2}\mathbf{w}^T\, Q_t^{-1}\, \mathbf{w}\right)$$

By plugging the above equations into the abstract specification of the Markov localization algorithm, one obtains the formulation of the EKF algorithm as represented in algorithm 3. For simplicity, the term m representing environment memory in Markov localization is excluded, but the explicit full formalization can be found in more specialized texts, such as [56].

3 Theory notes: Simultaneous Localization and Mapping

The *Simultaneous Localization and Mapping* problem (SLAM) consists in progressively building a map of the environment, and contextually finding the vehicle's position in it. This is significantly more difficult than the mapping and localization problems taken separately, because of the circular dependency: mapping works from a known ego pose (e.g. fixed, or already computed by localization), while localization algorithms similarly assume a known map (e.g. predefined, or already detected by sensors with sufficient confidence).

Mathematically, one can define the SLAM problem as the estimation of

$$p(x_{1:t}, m \mid z_{1:t}, u_{1:t}) \tag{3.3}$$

Like in the previous paragraph, here $z_{1:t}$ and $u_{1:t}$ indicate the past history and current value of measurements (sensor readings) and control inputs. This formulation is named *full SLAM*, and is distinguished from *on-line SLAM*:

$$p(x_t, m \mid z_{1:t}, u_{1:t}) \tag{3.4}$$

The difference lies in the fact that the latter only seeks to estimate the current (most recent) pose, while the first attempts to keep (and possibly rewrite) the full history of the vehicle's movement. This has consequences on the structure of the algorithms dedicated to each variant. Both are important in practice, although the evaluated algorithms fall more often in the *full* category.

To get a more concrete picture, one may look at the mutual dependence relationship between the "localization" and "mapping" parts of the problem. Mapping uses localization as an initial hint about where to place a newly observed portion of the environment (e.g. new sensor readings) in the map. Localization depends on mapping for *place recognition*: the map is designed to offer a way to recognize already visited places, and on the event of a successful recognition, correct the current localization estimate. In the case of *full* SLAM, the correction is propagated to the past, adjusting the whole history of past poses as well. This is referred to in SLAM literature as the *Loop Closure* problem.

4 Visual SLAM

4.1 Cameras for visual SLAM

Thanks to the use of binocular vision, biological mechanisms, and experience, humans are able to estimate distances and track their position while moving, especially in a structured environment like a grid of roads.

Replicating such ability on robots became an important field of research since the application to robots for Mars exploration, and was eventually extended to various applications such as 3D object reconstruction and mapping. This family of algorithms can estimate distances and reconstruct geometries using multiple image of the same scene, captured from different points. Since the projection of 3D points over 2D images is well known, it is possible to apply inverse projection formulas in a probabilistic way, so that more data (e.g. multiple frames, more features) can be used to improve accuracy. For such applications, both multi-camera (especially stereo) and single-camera setups are viable options and still being improved in research. Stereo cameras directly provides two frames taken from slightly different positions (usually they face the same direction and are separated by a small distance, typically around 10 cm), while in the monocular cameras case it is necessary to use subsequent frame during movement.

In this thesis, we evaluate the use of a mono-camera system. While, in principle, performance and accuracy are lower with respect to stereo systems, it is still unclear which of these two solutions will be adopted more widely in the future. Tier 1 suppliers like Bosch already provide both products. Currently, the OEMs are divided in this choice, with BMW, Jaguar and Subaru focusing on the first, and Volkswagen and Toyota on the latter [1].

[1] statement from http://www.researchinchina.com/Htmls/Report/2016/10231.html. It should be noted that these companies are at different stages on this research

An argument in favour of monocular cameras comes from Mobileye, which indeed focuses entirely on this solution:

> *"The Mobileye mono-camera was inspired by human vision, which only uses both eyes to obtain depth perception for very short distances," says Amnon Shashua, co-founder, chairman and CTO of Mobileye. "Therefore, the added benefit of a second camera lens is only relevant for short distances. Driving-scene interpretation is based on much longer distances. All depth-perception cues for farther distances – such as perspective, shading, texture, and motion cues, that the human visual system uses in order to understand the visual world – are interpreted by a single eye. Therefore, Mobileye understood that a single-lens camera could be the primary sensor to enable autonomous driving."*
>
> [Mobileye - About]

Due to the interests in the solutions adopted by Tesla and comma.ai and the ease in setting up a monocular system, we favored this while aware of the better capabilities of stereo system in some applications.

4.2 Basics of monocular SLAM

Monocular SLAM methods generally belong to two categories: **feature-based** or **direct**. In feature-based methods, the program finds salient points (named *features*) in the image that can easily be tracked between frames. As previously said, the position of these points on the different frames are constrained by geometrical projection. The features are then added to the map in the position that minimized the *reprojection error* computed on past camera poses. In *direct* methods instead, the image's pixels are all used, and their placement in the map is estimated based on the minimization of a *photometric* error. Figure 3.1 offers a general comparison between the two approaches.

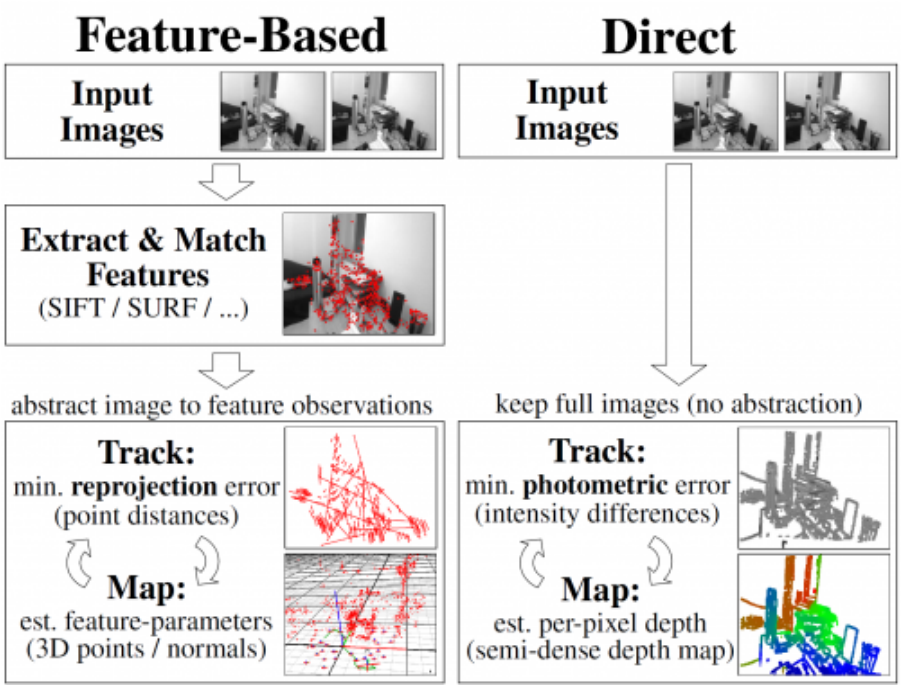


Figure 3.1: Schematic comparison of direct vs. feature-based SLAM methods. [22]

Typically, a feature-based method is composed by two processes running concurrently. This was a central innovation of the PTAM system by Klein and Murray [27], adopted by most of its successors.

The first process is for data acquisition and camera tracking, and runs the following cycle:

1. Prepare the image, by normalizing, scaling and blurring for filtering out noise and giving more robustness to the feature detector with respect to the variability of illumination conditions.

2. Detect features.

3. Match the features with one or more of the previous frames, or with the map.

4. Compute the 3D roto-traslation of the camera that most plausibly would yield the observed projected position of the features.

5. Add the new frame and pose to the map, together with relationships linking it to the frames and poses already stored. These relationships form the *pose graph*.

The estimation of the pose of a newly received frame is setup as a maximum likelihood estimation: what is sought is the roto-translation parameters that maximize

the likelihood of a correct match, which is inversely proportional to the reprojection error. Techniques such as random sample consensus (**RANSAC**) allow to quickly evaluate many roto-translation hypotheses by considering a random subset of the image features.

Once the most probable transform between the camera poses is accepted, usually the algorithm optimizes the full map periodically. During this phase, if the camera returns close to a previously visited location, the current estimate is "forced" to coincide with the previous estimate, and the correction is then propagated to the whole path. This is called *loop closure*, and it adds a strong constraint, both for the positions and rotations along the path, see fig. 3.2.

While new poses are added to the pose graph, the other process continuously optimizes it so as to minimize the global reprojection error. In order to perform this optimization, a complete formulation of this error has to be available. Different algorithms make different proposals on this regard.

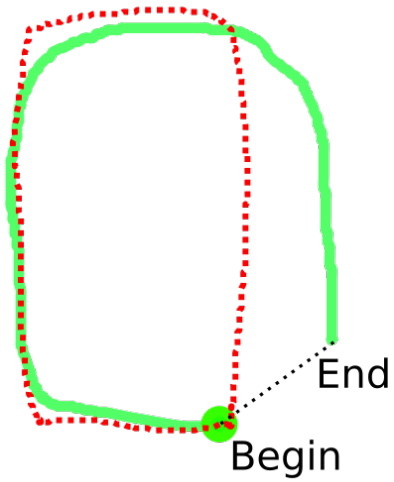


Figure 3.2: When a loop is detected (between the **Begin** and the **End** positions for instance) a constraint is added to the map, thus the global optimization algorithm reevaluate all the poses in the existing path (green) in order to obtain a new path (red) that explains the loop closure.

Feature detection/recognition

There are many algorithms available for **feature detection** and **feature matching/recognition** (the task of matching the same feature between different images). They are often evaluated in terms of computational time (usually it's in the order of ms), robustness, recall and precision. Among the most used are **SIFT**, **FAST**, **SURF** and the most recent **ORB**, on which the **ORB-SLAM** algorithm is based.

An overview on these algorithms and their performance can be found in [18]. A brief explanation on how these algorithms are implemented follows.

All these methods generally define a *descriptor*, which is a vector or bitstring that identifies the appearance of the feature, and a *distance* or *dissimilarity* function. Matching then happens by selecting the mutually "nearest" features based on the distance function; for this, several methods are available for deciding which distances to compute, and matching descriptors based on the result. For example, other than the brute-force solution which works fine in many cases, there is FLANN, a Fast Library for Approximate Nearest Neighbors (see [34]).

Features are detected and recognized in the image by applying simple statistical computations in the neighborhood of a pixel. The neighborhood has fixed size, for instance a squared patch of 17×17 pixels around a point. The operations may be performed on all the pixels, scanning the image from top-left to bottom-right, or on a subset preselected through a faster but less selective algorithm. The same operations are often performed on multiple scaled versions of the image in order to make the algorithm robust to scale variability. The operations are very different depending on the technique, but they are usually similar to a gradient computation, since we are interested in points similar to corners, where brightness changes consistently and strongly in the neighborhood (think about the corners of a traffic sign, or the tiles in a pavement). SIFT, for example, includes a convolution of the image with a 2D **difference of Gaussians** function in every point. Such convolution results in high values close to edge or corners. Only points with response (output of the function) over a pre-defined threshold are considered.

Other algorithms such as ORB, while they also work on a squared patch around a point, compute simple differences between pairs of pixels around the neighborhood. The coordinates of the pairs, with respect to the center point, are precomputed (see the original paper [48] for the machine learning techniques involved in the computation of these point coordinates).

These differences are encoded in a binary vector of fixed length, equal to the number of pairs, called the **feature vector**. This vector, when computed for the same corner in different images, will have similar values even in presence of Gaussian noise, brightness or orientation change. This makes the ORB descriptor very robust.

When matching features between two images, the feature vectors are compared using Hamming distance[1] and matched using brute force (all possible matches are tested) or with the help of a nearest neighbour search.

In comparison with other feature detector and descriptor algorithms, ORB is

[1] i.e. the number of different elements in a binary vector

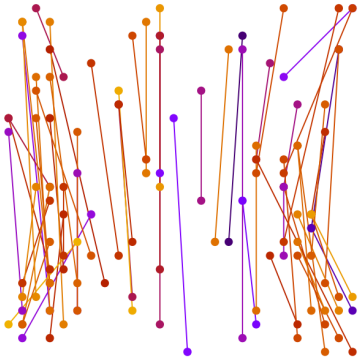


Figure 3.3: An example of precomputed pair of the ORB algorithm from the original paper. The positions of these points seems random due to the fact that it is chosen by optimization

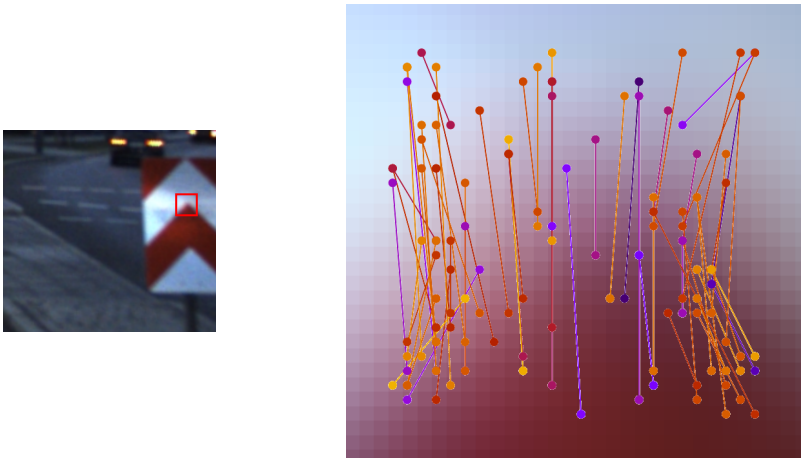


Figure 3.4: When searching features, the same computation is performed around possible salient points in the image. For each pair, in order, the brightness value of the selected pairs of pixels are subtracted. The results are encoded in a binary vector (1 when difference is ≥ 0, 0 when < 0) of the form $[1, 0, 1, 1, \ldots, 0, 0, 1]$. Beyond being easy to compute, these vectors are also easy to compare (simple Hamming distance). Note that there is a simplification in this example: the selection of the pixels pairs is performed along the orientation of the considered patch. This allow to have the same or similar result even if the image is rotated.

fast (enough for real-time tasks) due to the simplicity of the operations and has comparable performance to SIFT, plus it is available under an open-source license. The type of operations involved can also be easily performed in parallel on GPU. These features have made it popular in robotics.

Figure 3.5: Example of matching of ORB features from the OpenCV Tutorials

4.3 ORB_SLAM2

Built on top of the ORB feature detector and the g2o graph optimization algorithms, ORB_SLAM [35] is a complete SLAM algorithm for stereo or monocular cameras. The system tracks ORB features over the captured images and generates a graph of poses and observations. The graph is then used for applying a fast bundle optimization [1] that optimizes the poses and the created map.

The graph is composed by `KeyFrame` structures, which include the pose of the camera and the ORB features of that frame. While the camera is moving, a new `KeyFrame` is added to the graph every time the rototranslation between the current frame and the last `KeyFrame` is enough different (i.e. small movements are ignored, and instead incorporated in the last `KeyFrame`). It should be noted that in all visual odometry algorithms it is easy to estimate translations of the camera, unlike rotations, which pose difficulties due to the fact that projections during rotations are not linear. Although adding a new `KeyFrame` with high accuracy helps the algorithm follow fast rotations and movements with good performance, a large number of keyframes would make the overall algorithm slower and raise memory usage, since the optimizer has to go through every pose and we would have more data to store (such as ORB features). Thus, ORB_SLAM introduces a parallel thread that periodically removes redundant keyframes, which are those that do not affect the results of the bundle adjustment algorithm. This makes the system suitable for long distances as well.

ORB_SLAM also has a feature not usually found in other SLAM algorithms: it supports localization in a pregenerated map thanks to the matching of ORB features. When going through the same road, the algorithm is capable of noticing

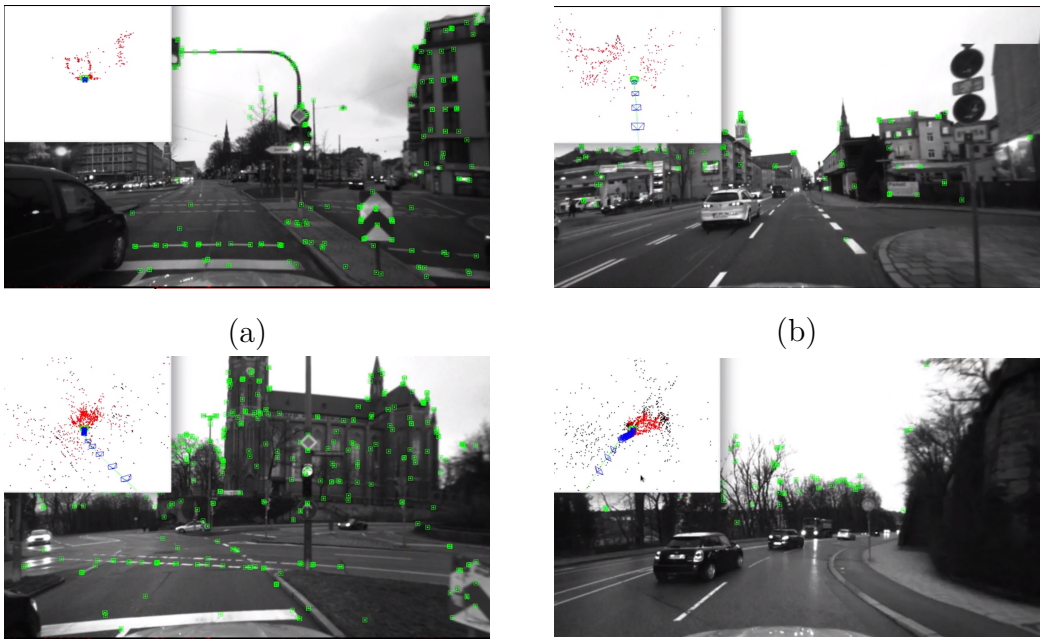
[1] **Bundle optimization** is a task very similar to SLAM, since it consists in jointly reconstructing a 3D map together with camera poses from a set of images. However, while the two topics shares a lot in common, SLAM is distinguished because it is usually performed incrementally (as new measurements arrive), because it is not limited to projective geometry but can incorporate different models. [7]

the same elements in the image and thus can point out the viewpoint in the map. This is possible thanks to the bag of words approach: An index of occurrences of every ORB feature allows to match the ORB features from the current image to any stored keyframe. This is the same mechanism that enables loop closure.

Optimization is performed using the g2o library, which implements a variant of Levenberg–Marquardt applied to graphs (see appendix 3 for a general description of the algorithm).

We used ORB_SLAM2[36], an updated version from the same authors with some improvements and support for stereo or RGB-D cameras (not used here). The algorithm is applied using only the rectified images from the central front camera. Our target was to reproduce similar results to the ones reported in the paper over the KITTI[1] dataset, so we positioned and tuned our camera in the most similar way. After rectification, we crop the image in order to maintain most of the horizontal field of view, while cropping out the hood of the vehicle and the sky. It is still important to keep distant objects from the background in the image, since ORB_SLAM2 is capable of using them for improving rotation estimate. This is due to distant points not changing following a translation of the camera.

[1] http://www.cvlibs.net/datasets/kitti/



(a) (b)

(c) (d)

Figure 3.6: Sequence from one of the attached test video where ORB_SLAM2 is running. In the top left corner of every image, the 3D viewer shows the current position of the camera (green), the previous keyframes (blue) and the distribution of past and current feature points.

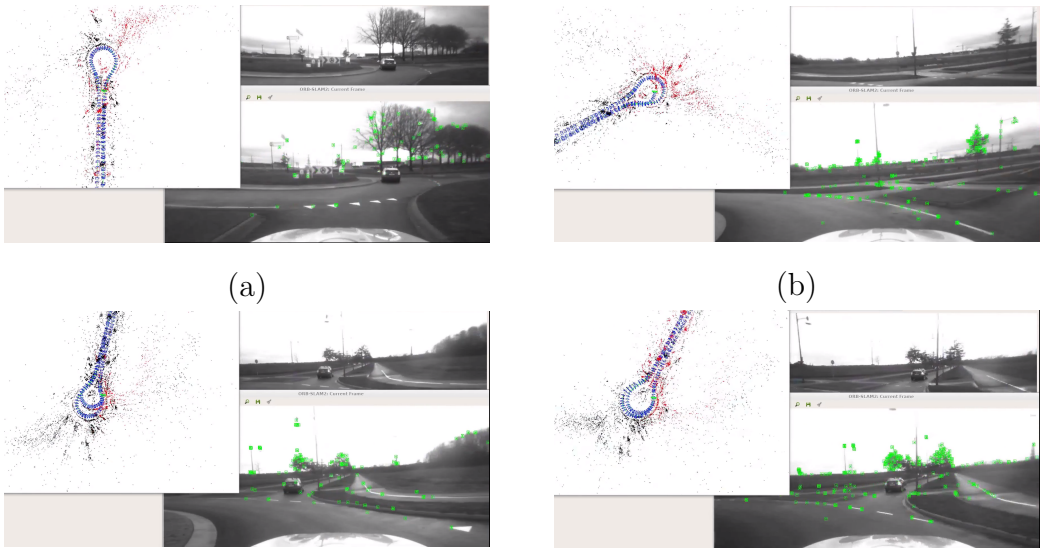
As we can see in fig. 3.6, ORB features (indicated by green squares) are almost uniformly distributed on the whole image. This is enforced by searching for ORB features in every cell of a grid in which the image is divided. This distribution helps find points distributed along the **Z** axis of the camera (**X** axis of the car) which is really important because coplanar points do not add enough geometric constraints during rotations. The more points are distributed in 3D space, the more accurately we can track the pose change of the camera.

The marked features are the features that were successfully tracked in previous frames. Usually these features are robustly detected and they come from corners (for instance, in windows, lane markings, signs). Usually, the number of matched features is between 50 and 300. If not enough features are tracked, then the system will not be able to follow the camera movement any longer, causing a "camera lost" situation. This can happen on empty environments, where all surfaces appear uniform; for instance, a parking lot with no markings. It is a known limitation that affects all vision-based localization systems and that highlights the fact that other valid sources of odometry are needed for a proper implementation.

It is important to note how some features will actually have the effect of decreasing the accuracy. For instance, let's assume that some points are tracked on a

close, moving vehicle. If the vehicle is moving, the points on the vehicle will change their position with respect to the static points of the scenario (for instance, points belonging to buildings). RANSAC, as explained earlier, should recognize these points as outliers, making the system robust to these kind of problems. However, this is not always the case, and the points of the vehicle will affect the evaluation of the camera movement. As suggested in section 2, hinting ORB_SLAM to ignore these points would improve overall performance.

We observed that ORB_SLAM's capability to optimize the pose and the path using the previously mapped points works as expected in many situations. For instance in sequence of fig. 3.7 we see the car going through a roundabout already visited (a), but this time the system wrongly computes the camera movement assuming that the radius of the round is much shorter (b, c). However, while exiting the roundabout, the system recognizes the previously known position and realigns the whole path around the roundabout, improving it.



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

Figure 3.7: Roundabout.

5 Lidar SLAM and localization

Thanks to the high accuracy of the generated point clouds, lidars are a viable and proved sensor to use as base for localization and mapping. Most solutions are based on comparing multiple scans and estimating their relative pose, in a way similar to Structure-from-Motion algorithms. At a basic level, most methods project the 3D

points from the lidar's scan to the horizontal plane, in order to simplify operations and make them more intuitive to debug. In robotics, it is indeed often the case that the robot moves on the horizontal plane. For cars, however, this solution may not always work well, except in the short range where the vehicle does stay approximately on a plane, i.e. such methods are fine on straight roads, but will cause problems on hills, ramps etc.

We will analyze the most common methods and explain why we chose the **ndt_mapping** package for mapping.

It must be highlighted that these methods can both work standalone (e.g. without Kalman filtering and, in some cases, without odometry data) and within other localization systems for sensor fusion (See section 6).

5.1 Iterative Closest Point

This is one of the most basic methods for lidar SLAM. It works in 3D space and it performs quite well in structured and dense environments. As a SLAM algorithm, **ICP** [3] tries to solve the following problem: given two poses and the point clouds generated from the two pose as points of view, what is the transform (i.e. the rototranslation) between them? By solving this problem multiple times during vehicle movement, the algorithm can be used for reconstructing the vehicle's path. Usually, the accuracy of the path suffers due to the accumulated error between each pose calculation, but this problem is solved using bundle adjustment techniques (see Loop Closure at section 4.2).

The algorithm can also be used as a constituting step of a larger localization solution. For instance, assuming that we have a highly accurate estimation of pose (thanks, for instance, to the use of an RTK GPS; see section 3.5), we can build a map by accumulating multiple point clouds and then aligning them. Such procedure of aligning multiple scan is also called **registration** in literature.

Many improvements to the original ICP algorithm were suggested, and they do obtain better results in some cases. We will however shortly describe the classical algorithm, which, being really common, is included in the PCL (2.1) library. Intuitively, the algorithm works by "overlaying" two point clouds using successive approximations (fig. 3.8). For a human, this operation is quite easy, and so it is for the algorithm as long as the point clouds are "similar enough". Indeed, the original paper proves convergence, but only to a local minimum. So it is the duty of the developer to make sure the two point clouds were generated from close positions, in order to give a good starting point to the algorithm.

Given the new point cloud P and the model point cloud X, the ICP algorithm finds for each point in P the closest point in X. A correspondence is assumed between these closest points, and a measure of misalignment is calculated as the root mean square of the distances between the points. This is also used as the

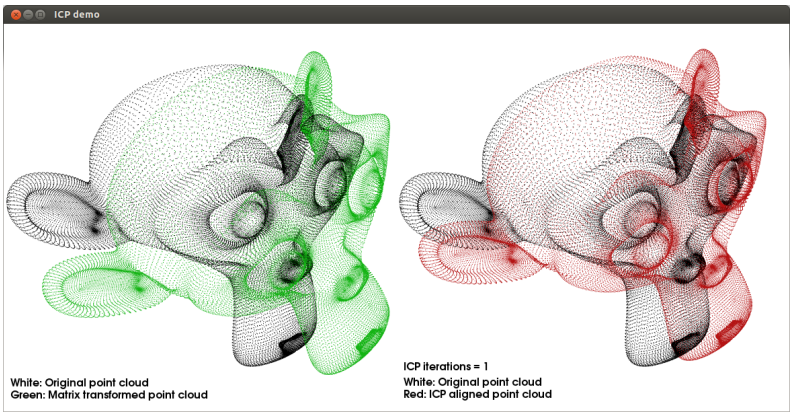


Figure 3.8: Alignment of two point clouds using the ICP algorithm. The misalignment between the model (black) and the data (green, then red) is reduced iteration after iteration converging to a local minimum. It is not possible to determine if the algorithm converged to the global minimum. [*pointclouds.org*]

objective function. The algorithm calculates the transformation matrix to be applied to P that reduces the misalignment. The procedure is repeated until a certain preconfigured tolerance is reached.

5.2 Normal Distribution Transform

Like the ICP algorithm, the Normal Distribution Transform (**NDT**) [4] computes a geometrical transformation between a given point cloud and an existing model, which is often a pregenerated map or a point cloud collection generated during SLAM like in fig. 3.9.

The algorithm however introduces the idea of simplifying the model point cloud by transforming it into a set of normal distributions in 2D space.

First, the space is divided in a fixed size 2D grid. Given a point cloud, each point belongs to a cell in the grid depending on its (x, y) coordinates. Then, for every cell containing at least three points, a 2D normal distribution is computed. The distribution is simply computed by fitting the 2D coordinates of the points within the cell:

$$\text{mean:} \quad \mathbf{q} = \frac{1}{n} \sum_i x_i$$

$$\text{covariance matrix:} \quad \mathbf{\Sigma} = \frac{1}{n} \sum_i (x_i - \mathbf{q}) (x_i - \mathbf{q})^T$$

At this point, we have a sum of normal distributions, which, being the normal distribution differentiable, implies we have a piecewise differentiable function.

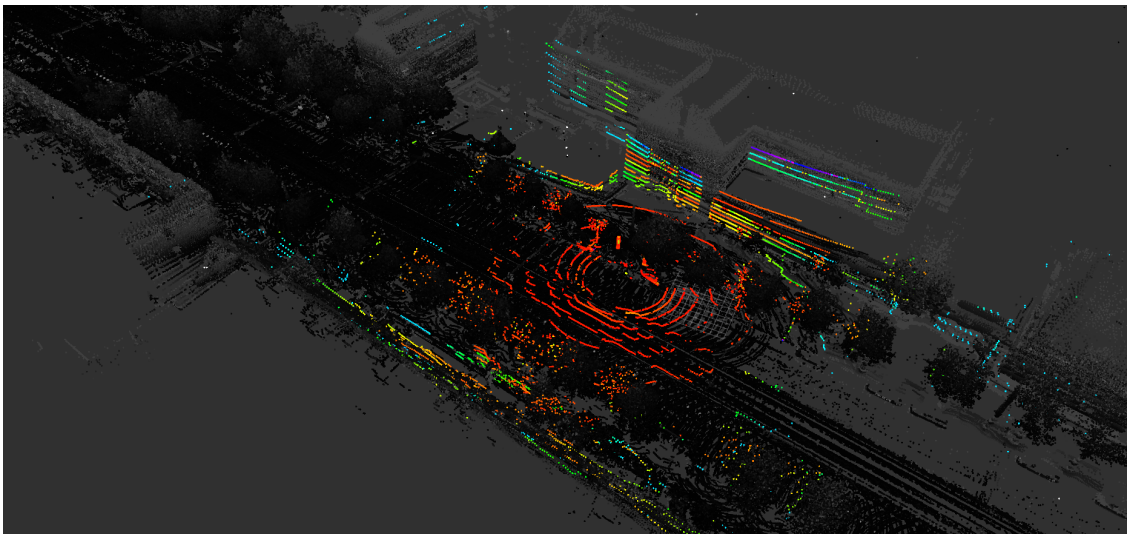


Figure 3.9: The normal distribution transform allows to track the position of the vehicle by matching the point cloud coming from the sensor (colored points) against an existing point cloud map (grayscale) using the ndt_matching implementation from Autoware. The map is generated using the same process described in section 7.3

Given a new point cloud, it is possible to choose the optimal rotation and translation parameters in order to match the cloud with the model by using the standard **Newton's method** (See appendix 1). More in detail, the parameters $\mathbf{p} = (\phi, t_x, t_y)^T$ define a transform from a 2D point (x, y) in reference frame A to a point (x', y') in reference frame B:

$$T : \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} cos(\phi) & -sin(\phi) \\ sin(\phi) & cos(\phi) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

Optimal values for the parameters $\mathbf{p}$ are sought that align a new scan to the model, summarized by a grid of normal transforms. Starting from an initial estimate $\mathbf{p} \leftarrow \mathbf{p_0} = (\phi_0, t_{x,0}, t_{y,0})$ we calculate a *score* of the match between the model and the scan:

$$\text{score}\,(\mathbf{p}) = \sum_i \exp\left(\frac{-\left(x_i' - \mathbf{q}\right)^T \Sigma_i^{-1} \left(x_i' - \mathbf{q}\right)}{2}\right)$$

In other words, the "goodness" of matching is computed as the sum of the probabilities yielded by the grid-of-Gaussians model, computed for each point of the new point cloud and summed. The value of $\mathbf{p}$ that maximize the score is

sought by using successive approximations. As said before, this is possible since the objective function is differentiable. The exact expressions of the Jacobian and Hessian matrix, used in the Newton's method, can be found in the original paper.

6 ROS implementation: robot_localization

Localization is an essential and ubiquitous task in robotics, as most robots need to localize themselves or parts of them (such as a manipulator, even if the robot does not move around an environment). As such, ROS naturally provides many high-quality implementations of localization and SLAM algorithms, catering to a number of different use cases. Among these, some notable options are:

- `amcl`, [47] which is included in the standard ROS navigation stack [46], and provides a probabilistic 2D localization system based on the KLD-sampling Monte Carlo localization approach [15];
- `robot_localization`, [33] which is a generalized Extended Kalman Filter that supports a wide range of sensor inputs;
- *Google Cartographer*, [19] a real-time SLAM system by Google, mainly intended for LIDAR (point cloud) and inertial sensors input, both in 2D and 3D;
- *ORB-SLAM* [36], a feature-based visual SLAM system, which we discussed more in detail in section 4.3;
- *LSD-SLAM* [14], a direct (full image) visual SLAM system.

The first choice we made was whether to use a full SLAM system (see section 3) or a more traditional filtering-based localization solution. We decided for the latter, mainly on the grounds that we easily get two different types of estimates out of the algorithm: (1) a *local* estimate, akin to the result of performing odometry, that is *continuous* and can be considered accurate in the immediate surroundings of the vehicle, but accumulates an error if measured against the car's starting location; (2) a *global* estimate, which may be subject to discontinuities (i.e. the vehicles seems to "teleport" itself a few feet away every once in a while), but is generally accurate with regards to a global (geographic scale) map.

We then chose the implementation based on a few criteria, the most important being:

- **Compatibility with many sensor types**. Our experimental vehicle is equipped with many sensors, including cameras, LIDARs, inertial platforms, GPS, speed meters, some being part of the vehicle's standard equipment, and

some we installed ourselves. In theory, localization can be aided by utilizing the data from all those sensors, so we were interested in implementations compatible with them.

- **Implementation quality**. ROS is not just made out of concrete components (such as software and collaboration infrastructure on the Internet), but also conventions. A node that does not adhere to ROS' conventions poses many costs and challenges: it is harder to integrate with the rest of the system, harder to make tools for, may require modifications to its source code (and therefore integration in our build system). These costs are non-essential, and are ideally avoided by using software with high implementation standards.
- **Documentation quality**. Implementation quality would be largely useless without good documentation that guides the user through the configuration and tuning of the ROS nodes.

The project that best matches our requirements that we know of is `robot_localization`. The software includes two complete, high-quality, well-documented implementations of filtering-based localization; one is based on an Extended Kalman Filter (ROS node `ekf_localization_node`), the other on an Unscented Kalman Filter (`ukf_localization_node`). Both nodes have identical interfaces with regards to ROS parameters, topics and services, so they can be swapped with almost no additional work (other than adapting the part of the configuration that is specific on the type of filter).

Importantly, the package also includes tools to convert the estimate from the vehicle's linear frame of reference to the WGS84 geographic frame of reference (which is significantly non-linear over a large enough scale) and vice versa. In particular, a node named `navsat_transform_node` allowed us to easily bring the geographical coordinates provided by GPS into a form compatible with the other inputs, and use it alongside other sensors.

We evaluated our results by visualizing the location estimate overlaid over a geographical map by using the Mapviz visualization tool [31]. Mapviz uses another method for making localization estimates compatible with geographical data, from the `swri_transform_util` package. Both are published as open source by the Southwest Research Institute.

`robot_localization` is very strictly compliant to REP-103 [42] and REP-105 [43], ROS' standardized convention for units of measure and the communication of odometry through messages and frame of reference (in the sense of TF frames, see section 1.1), respectively. For the purpose of localization, the most important convention is that the TF tree has root in a frame named `map`, and that it contains the chain of transforms `map` → `odom` → `base_link`, where:

- `map` is the fixed frame, conventionally rigidly attached to "the world". Any map of the global environment will be expressed in this frame.

- `odom` is also a world-fixed frame, but it differs from map in order to represent the accumulated odometry error. The discontinuities ("teleports") of the car's trajectory will also be visible in the `map` → `odom` vector.

- `base_link` is the vehicle-fixed frame. It represents the current location of the robot.

Somewhat non-intuitively, even though we have two localization estimates independently computed (global and local) that relate `map` and `odom` to the same `base_link` frame, it is not possible for `base_link` to be represented in both frames of reference (i.e. the TF has to be a tree, and a node can not have two parents). Instead, there is a *map* → *odom* transform, computed from the *difference* between the two estimates.

Another useful topic covered by the REP-105 specification is the convention for representing geographical data (such as GPS). `navsat_transform_node` requires GPS messages to follow this convention in order to translate them into the same form as other inputs.

6.1 Configuration

The state estimation nodes from `robot_localization` are both configured in the same way. Different message types can be used as input, in arbitrary combinations, as long as they follow REP-103 and REP-105 [38]. We made use of the data extracted from the car's CAN bus by our `fa_can_ros` package (see section 2.2):

- GPS

- Accelerometer

- Wheel odometry

The difference between the two state estimation nodes that we run as part of the localization service is whether or not they use the GPS feed as input. The local estimator (odometry) does not use the GPS, and mostly integrates the velocity and acceleration feed, while the global estimator does. Since the GPS input is relatively precise, it is configured with correspondingly smaller values in the error covariance matrix. As a consequence, the global estimator will follow the GPS' lead quite close, and be subject to the "jumps" that were already mentioned.

This can be directly seen in the launch files for the localization service [1]. Each input must be associated to boolean mask (given as a ROS parameter), specifying

[1] Further details about our concept of services are at chapter 6

which of the state variables to which the particular input can contribute. The complete list of state variables estimated by `robot_localization` is:

$$[x, y, z, \alpha, \beta, \gamma, \dot{x}, \dot{y}, \dot{z}, \dot{\alpha}, \dot{\beta}, \dot{\gamma}, \ddot{x}, \ddot{y}, \ddot{z}] \tag{3.5}$$

The dot indicates a derivative, so the variables are, in the order: position, heading (as Euler angles), linear velocity, angular velocity and acceleration along the three axes.

Each input is linked to a state estimation node through a collection of configuration parameters. Other than the ROS topic, a boolean mask needs to be provided that matches the state variables vector. It will have a true value whenever the corresponding value of the state variable is to be considered. For instance, the configuration for the wheel odometry input is the following:

$$\begin{bmatrix} true & false & false \\ false & false & false \\ true & true & false \\ false & false & true \\ false & false & false \end{bmatrix} \tag{3.6}$$

That is, we only use the following components from the wheel input: $[x, \dot{x}, \dot{y}, \dot{\gamma}]$. One may ask why $\dot{y}$, i.e. lateral speed, is included, since only the longitudinal speed is measured. This is because we can consider a simplified physical model of the car, where lateral speed is practically 0.

7 Results

7.1 Localization performance

Usually the performances of a localization system are evaluated against a more accurate device, which is more expensive or that needs particular installation/setup. In our case, we used the Xsens MTi-G-710 as ground truth source. The device, beyond providing raw accelerometer and compass data, integrates a proprietary Kalman filter. In order to obtain the best accuracy from it, we configured it in the '*automotive*' scenario (id=4) with the following commands, that also enable additional outputs:

```
rosrun xsens_driver mtdevice.py -c if2000,oq400fw,rr,mf,vv,sw -x 4
```

The measure consists in comparing the error (*RMS*) of the normal GPS data and the output of the localization system against the XSens ground truth. Results differs of course depending on scenarios, boolean and covariance matrixes.

Unfortunately we were not be able to find optimal parameter values that would improve the localization on both the small (tens of meters) and big scale (kilometers). After various trials, we confirmed some of our expectations:

- The odometry is very accurate on small scales.
- The accuracy and the frequency of the accelerometer are not high enough for being an valid source of information in this system. It is thus ignored, explicitly or implicitly through the use of a covariance matrix with high entries.
- The error model of the GPS of the car is not Gaussian.

The GPS provided by the vehicle augments the original position with unknown internal methods, and, from observing the data, we can assert the behaviour is considerably different from standard GPS. The validity of a Gaussian model for this data is thus valid up to a limited degree.

We believe this problem makes it impossible to improve localization in large scales and reach a significantly better quality than GPS.

When giving more weight to the odometry instead, the system is able to also track small movements, like lane and speed changes. These settings should be used when we are more interested in the accuracy in a small scenario. For instance, the map generated in section 7.2 results from these setting.

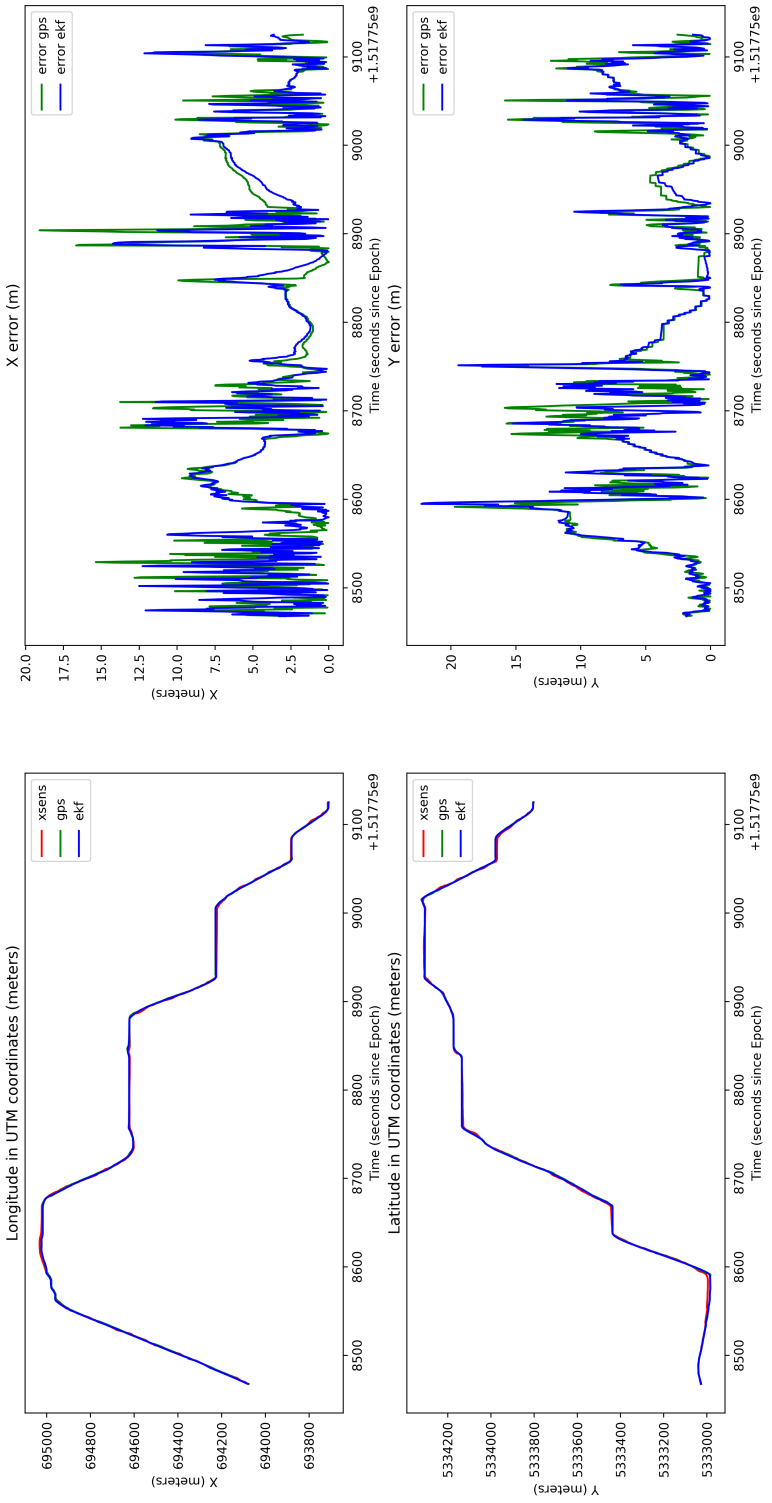
For reference, we include the path shown in the **Mapviz** tool and the related error measures for some tests.

Measures are reported in meters, in the local UTM frame. Red/Orange corresponds to XSens data, blue to EKF output, green to GPS.

Around Ostbahnhof N.2

In this trial, the EKF estimates the position with only slightly improvements. The odometry helps tracking the position during turns.

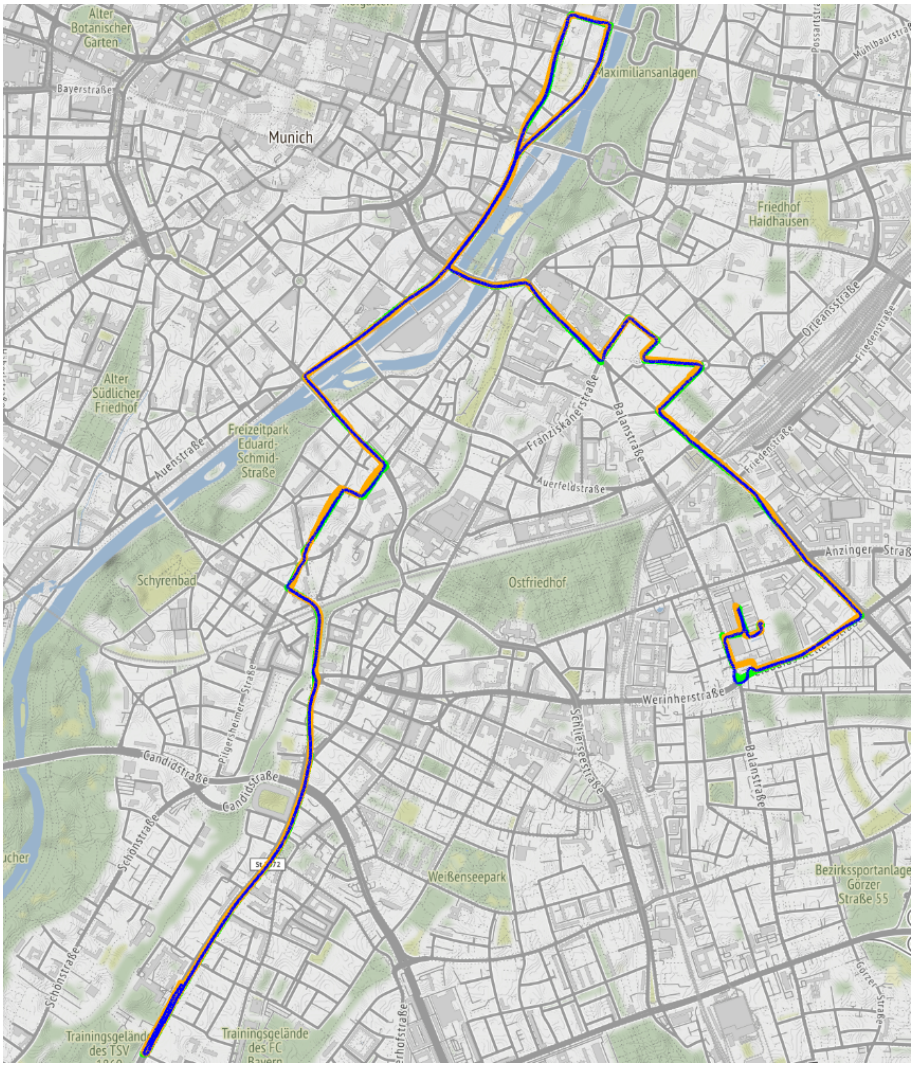
Path distance	3 485 meters
GPS X rms	5.02
EKF X rms	4.89
GPS Y rms	5.83
EKF Y rms	5.71

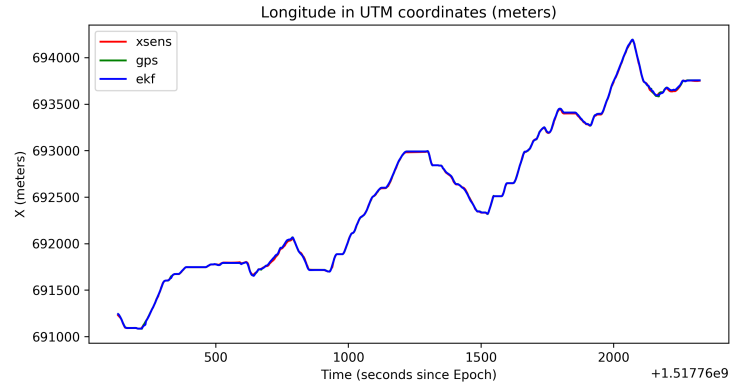

X error (m)
error gps
error ekf
X (meters)
Time (seconds since Epoch)
+1.51775e9
Y error (m)
Y (meters)
Longitude in UTM coordinates (meters)
xsens
gps
ekf
Latitude in UTM coordinates (meters)

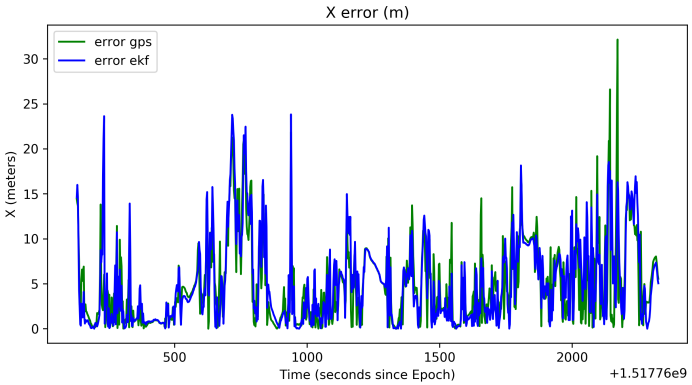
Bridge N.1

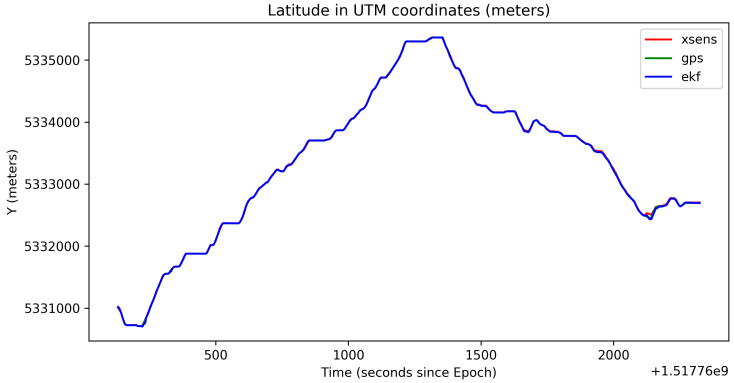
Here we can see the pathological effect of GPS causing the localization system to fail, especially around the final part of the path (section 7.1). The final error of the EKF output is even worse that the simple GPS estimate.

Path distance	12 145 meters
GPS X rms	6.63
EKF X rms	6.94
GPS Y rms	8.97
EKF Y rms	10.79




Longitude in UTM coordinates (meters)
xsens
gps
ekf
X (meters)
694000
693500
693000
692500
692000
691500
691000
500
1000
1500
2000
Time (seconds since Epoch)
+1.51776e9


X error (m)
error gps
error ekf
X (meters)
30
25
20
15
10
5
0
500
1000
1500
2000
Time (seconds since Epoch)
+1.51776e9


Latitude in UTM coordinates (meters)
xsens
gps
ekf
Y (meters)
5335000
5334000
5333000
5332000
5331000
500
1000
1500
2000
Time (seconds since Epoch)
+1.51776e9

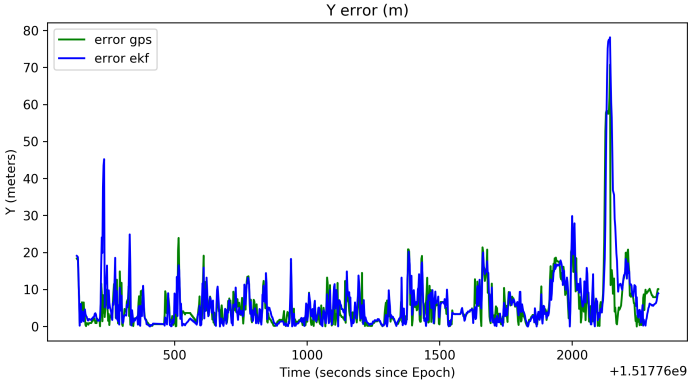

Y error (m)
error gps
error ekf
Y (meters)
80
70
60
50
40
30
20
10
0
500
1000
1500
2000
Time (seconds since Epoch)
+1.51776e9

Figure 3.10: Example of the GPS causing the localization to fail: at point 1, the GPS (green) tries to track the straight path, while the Xsens (orange) correctly follows the right turn. The EKF filter output (blue) initially follows the odometry data, detecting the right turn, but eventually the system becomes unstable due to the wrong GPS estimate, causing oscillations. A correct state is only recovered around point 2

7.2 Generation of the Neue Balan map

As stated in section 6, it is possible to achieve good localization results just by using odometry and GPS data. The system, in this configuration, is able to record point clouds during a trip, so that each point cloud scan is associated to a position measured with high accuracy. The collected scan are then processed offline by an optimization algorithm [1] that locally registers point clouds. The result is an high resolution 3D map of the Neue Balan Campus (Munich), which hosts the main headquarters of Objective Software.

[1] https://github.com/CPFL/Autoware/tree/master/ros/src/computing/perception/localization/packages/lidar_localizer/nodes/ndt_mapping

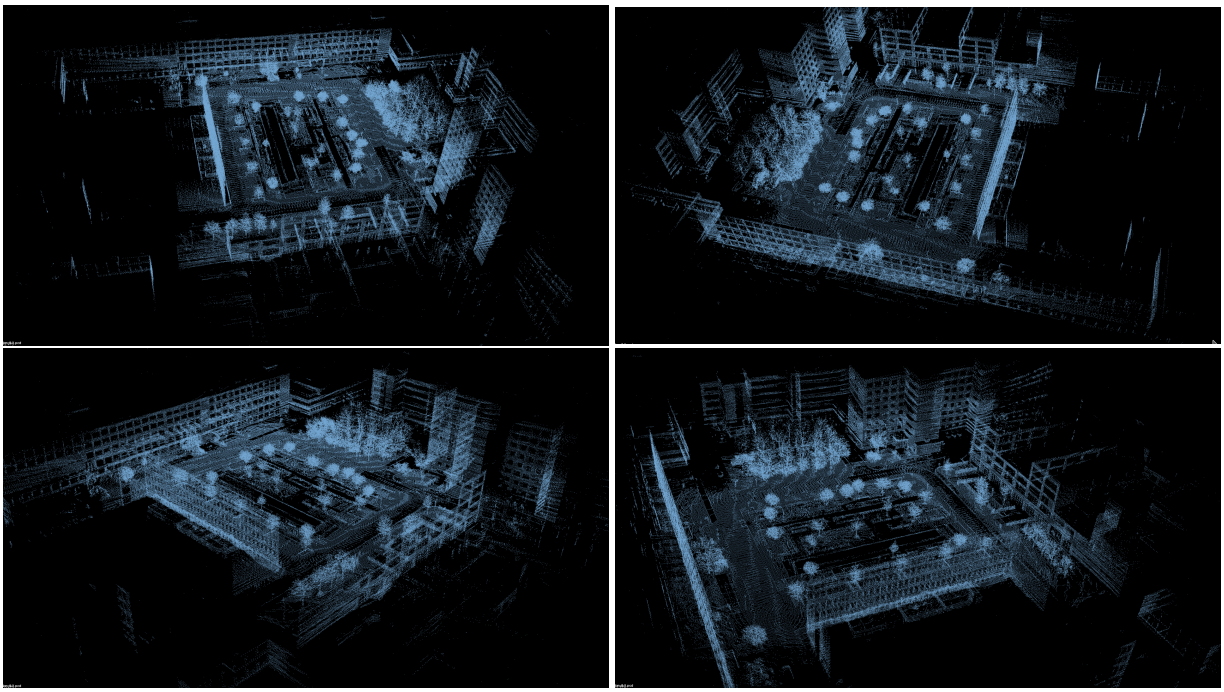


Figure 3.11: 3D point cloud model of the Neue Balan campus in Munich, from four different point of view. The system is able to maintain alignment between buildings and roads. Of course, the laser scanner has some difficulties scanning higher and distant points. These kind of model can be used for many scopes as explained earlier.

Figure 3.12: A promotional photo of the campus, for reference [*http://www.neuebalan.de/*]

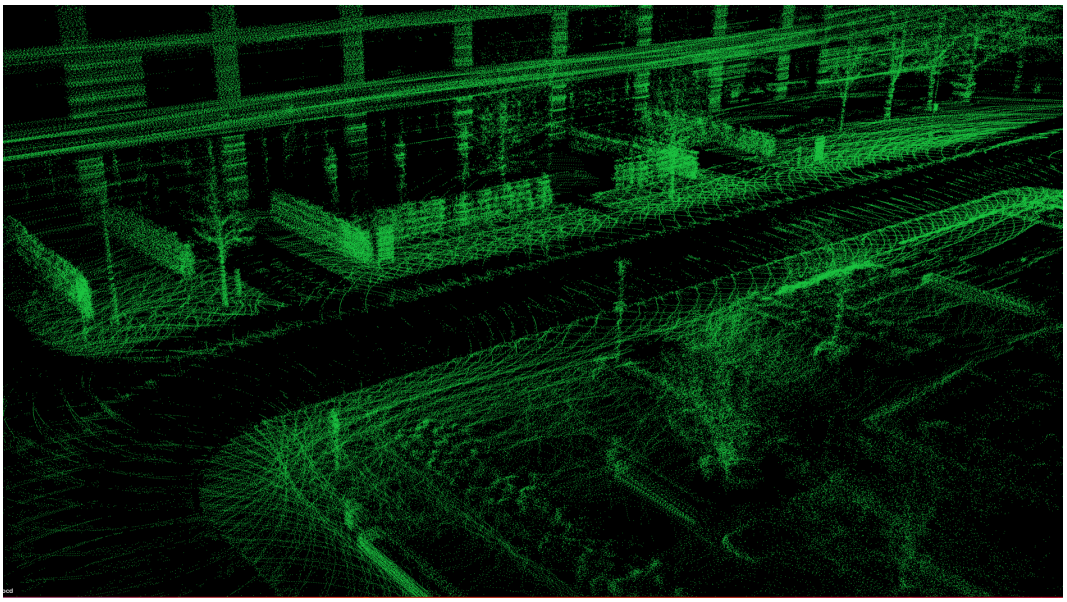


Figure 3.13: The level of details are high enough for distinguishing trees, poles, pavements

7.3 Generation of Urban Scenarios map

The generation works in urban scenarios as well, even if the computation of long point cloud recording is computational intensive and requires some hours even for a few kilometers.

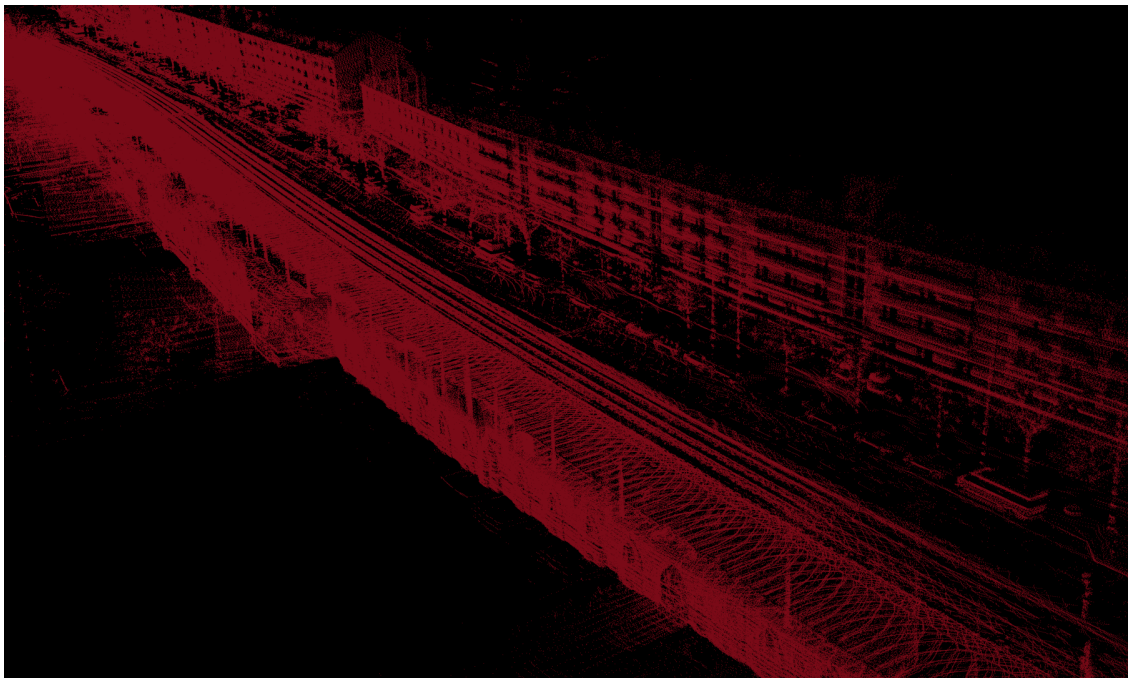


Figure 3.14: 3D map of a road using the same configuration. Also vehicles are well captured.

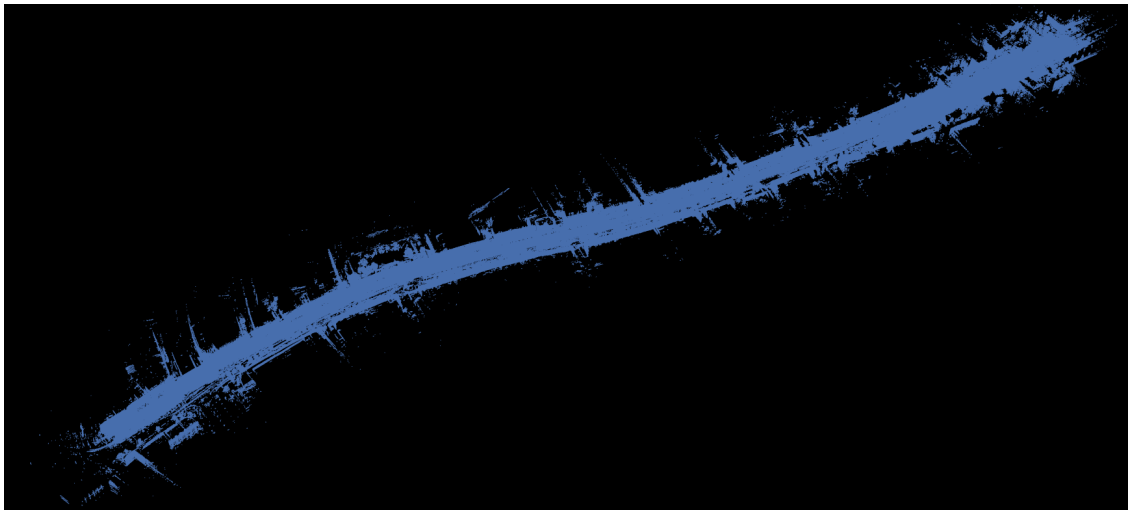


Figure 3.15: Birdeye view of Grünwalder Strasse, from dataset **Bridge N.1**

For reference, we add some comparision between the point cloud map (top) and the 3D model from **Google Maps** (bottom).

Figure 3.16: *Grödner Str., 81547 München, Germany* - (48.104655, 11.569883)

Figure 3.17: *Grünwalder Str. 31, 81547 München, Germany* - (48.107557, 11.573141)

Chapter 4

Detection

1 Camera Detection

1.1 Objects

Object recognition from camera images was an important topic well before the idea of self-driving vehicles. The task was once performed through the use of local image features and classical machine learning algorithms. Today these techniques are superseded by deep neural networks (appendix B section 2) and were not used within the thesis, so we will not discuss them here.

Deep neural networks for object recognition and detection

The following is a short overview of how neural networks are applied for the task of object recognition and detection. A short introduction on neural network and common terminology is provided in appendix B.

The capability of deep neural network in recognizing object are the results of the convolutional structure that can extract features from low to increasingly higher level. The first layers of the network are able to detect line patterns. In later stages, simple shapes are detected, then compositions of those shapes and so on. As stated before, the relative position of all of these features is meaningful thanks to the convolutional stages. It is possible to know what each neuron recognizes by using optimization methods that optimize for a single neuron, which generate images like the ones in fig. 4.1.

In complex networks able to detect tens of categories of objects it is not always so easy to distinguish such features for the human eye in a given network. This makes neural network known for poor introspection and control over errors, thus the visualization of these features is also an important topic in research.

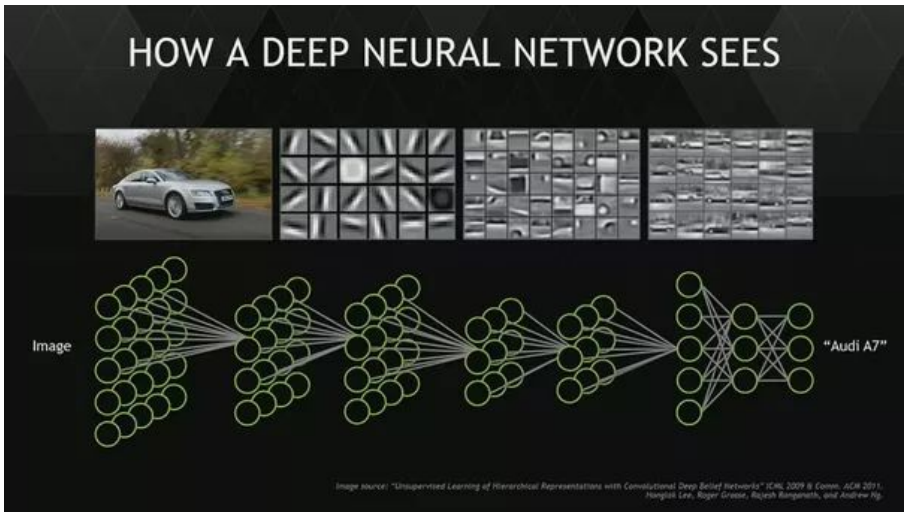


Figure 4.1: Deep neural network are able to recognize more and more complex feature at each layer. Simplifying a bit, the network first recognizes elements like lines or circles, then wheels and windows, finally vehicles [nVidia - based on [29]]

The output layer of the network and final processing functions are changed in order to obtain different types of output. By default, the outputs of the last layer are real values, hardly meaningful.

In the recognition case, one simple application is to assign an image to one category of a list of predefined possibilities. For instance, deciding if in a camera image there is a car, a truck, a bike or a person.

In order to obtain a percentage of confidence, indicating the probability that an image belongs to one of these categories, the **softmax function** is applied to the output layer. Indeed, this function will normalize the output to an array of real values within the range (0, 1), and the sum of these values will be equal to 1. Thus, these values can be interpreted as the probability for the image to belong to one of the categories.

The expression for calculating this probability is for a neuron output z_j is then:

$$P(j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

For instance, given the four outputs from the neural network $[2.2, -4.1, 3.92, 4.1]$ after applying softmax we will have $[0.0753451, 0.000138357, 0.420767, 0.50375]$, whose sum is equal to 1. If our categories are car, truck, bike, person as stated before, then we will affirm with 0.50375 confidence that the image shows a person.

Now, suppose we have an image that is not focused on a single object but on a more varied scenario, where multiple objects are present. We are interested in detecting all the objects, their category and their position in the image. As usual, the position of an object in an image can be defined with a bounding box, that is, a box centered around the object, often encoded with a tuple of four values: $(center_x, center_y, width, height)$

This kind of network is useful for instance when applied to multiple patches of the image, some of which may contain an object of interest. This approach is called *region proposal*. It is possible to scan all the regions one by one, from top-left to bottom-right.

Other algorithms are able to directly identify both objects and their position within the image. These approaches have better performance both in recognition and timing, since they do not need to repeat inference at different scale and position within the image.

The YOLO detector is one of these.

1.2 YOLO

The image recognition capability was built on the second iteration of the YOLO neural network, YOLOv2[40]. As of today, this neural network achieves almost the best performance when compared to existing solutions and yet is very light, so much that it can run on smartphone.

The innovation of this network relies in its simple pipeline and learning approach: No region proposal, no complex functions applied to the output, but just a network that outputs both predicted class and bounding box directly, so that it is possible to train it end-to-end[1]. As reported in the original paper [41], the use of a single network with no additional steps in the pipeline has many advantages: the full system is faster to execute and train and contextual informations are implicitly used since there is no region proposal and convolutional properties are maintained on all scales.

The network works by dividing the input image in a grid and in each cell of the grid a class is associated. Also, in each cell a variable number of bounding boxes (usually 2) are predicted. These bounding box are characterized by the (x, y) coordinates, relative to the cell origin, the (w, h) dimension, relative to the full normalized image dimension, and finally the confidence, defined as $Pr(Object) \cdot IOU_{pred}^{truth}$, that is the

[1] That is, the input and output of the system corresponds to the input and output of the full system

probability that an object exists in the cell times the IOU[1] between the prediction and the ground truth. The training jointly optimizes for these output values, eventually picking only classes with higher confidence. The architecture fig. 4.2 of the network is also simple, composed by convolutional and maxpool layers.

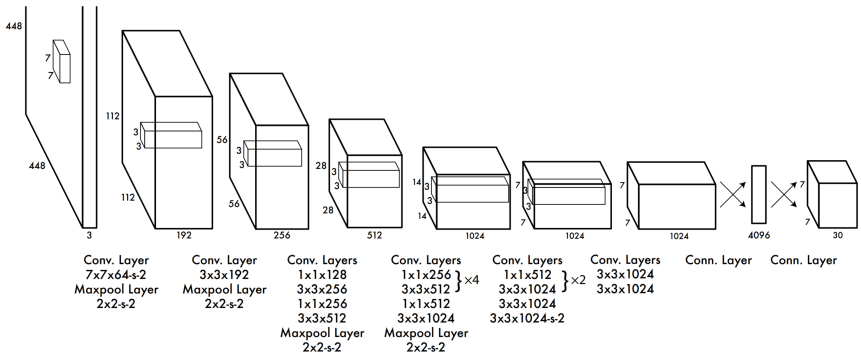


Figure 4.2: architecture of the original YOLO network. YOLOv2 adopts a similar network, but empirically add some steps in order to improve accuracy while maintaining realtime performances on high end hardware

The results of YOLO on the standard datasets are impressive, both for accuracy and performance. On our images fig. 4.3, the network shows good qualitative performance even with no further tailoring from our side. The recognized objects are published on the *image_recognition* topic as an array of bounding boxes containing the corresponding class ID and confidence probability.

[1]The **Intersection over Union** is a common measure of the quality of prediction of the bounding box, it is defined as $\frac{Area(Overlap)}{Area(Union)}$ of two bounding boxes. A good predictor should have a bounding box close to the ground truth and with similar size, thus with IOU close to **1**

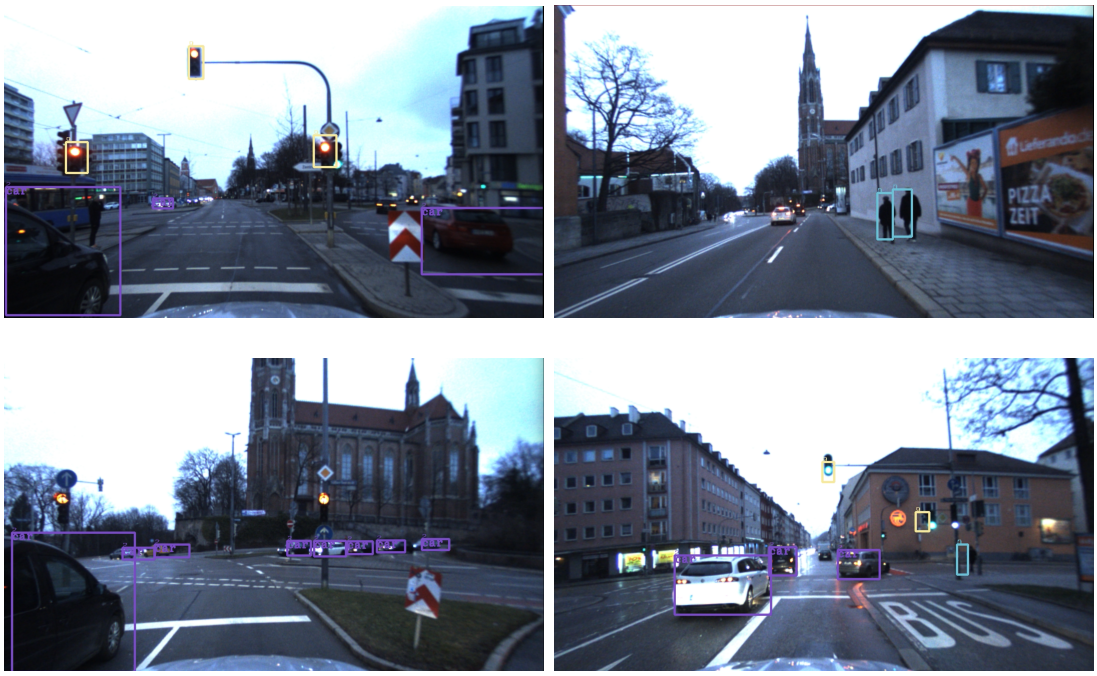


Figure 4.3: YOLOv2 applied on our images. Generally, the detector is able to find car, people and traffic lights in the sequence of images, but fails to detect all objects in a single frame.

2 Lidar Detection and Recognition

There is a growing and recent literature on the problem of detecting and recognizing objects from a point cloud. The problem is quite close to the corresponding 2D image case and, not surprisingly, many algorithm are a transposition from 2D to 3D space. In image processing 3D data is processed through the use of **histograms**, **features**, **euclidean metrics** and recently **neural networks**. Some basic solution for detection generates 2D image from 3D data. While this works, it does not fully exploit the information provided by a laser scanner, and thus achieves lower performance than direct 3D analysis. Tracking of recognized objects can still be based on Kalman filtering, also here with the necessary modification for handling 3D translations and rotations.

2.1 Processing point clouds with PCL

The **P**oint **C**loud **L**ibrary [49] is a collection of algorithms and utilities created for processing point clouds. It provides functionalities for loading and saving data, extract features and statistics, filter noise, detecting surfaces, objects and register

multiple point clouds. Released under a BSD License, is the de facto standard library for processing point clouds the same way OpenCV is for 2D images.

The available algorithms are usually used in a pipeline within a C++ application: The original point cloud obtained from a sensor is filtered, features and keypoints are detected, recognition may be performed on statistics or other metric properties. It also offers visualization tools. The library is enriched by new state of the art algorithms every year.

2.2 Clustering

Objects in most 3D environments, especially urban scenarios, have well defined boundaries. Even without evaluating futures, it is easy to notice how the shape of cars, people, buildings stand out on the road. It is thus possible to collect the points of distinct objects from a point cloud in different clusters. After this process, we still do not know if the cluster of points corresponds to an house or a car or any other class of objects, but we have a well defined object that can be assumed to be a rigid body. In the next lidar scan, the object will move or rotate, but its structure should be mostly the same.

This is a required step for some object recognition algorithms, which are basically classification algorithm, that tell if a set of points looks like a class of object. These algorithm take as input a single cluster at time, and they output the probability the object belong to a class. Usually this method is called **region proposal**. We will look further in these methods in the following sections.

A few notes on data structures and search algorithms

A simple laserscanner like the VLP-16 generates hundreds of points at every scan. For realtime applications it is necessary to use advanced data structures in order to reduce computational complexity. Structures such as KdTree and Octree allow algorithms that normally executes in $O(n^2)$ to reduce their complexity to $O(n \cdot log(n))$.

It should be noted that many of these techniques are used in machine learning in general, where the dimensionality of data can be greater than 3.

KdTrees KdTree are a data-structure that divides points in a K dimensional tree (where $K = 3$ in this case). It allows so quickly solve some geometrical problems where it is necessary to evaluate the euclidean distance between points.

A KdTree is built by splitting a node of three in two along **one** dimension **at time**. The dimensions are alternated ciclically at every level of the tree (for instance: X axis, Y axis, Z axis and then again X axis). Usually the median point is chosen as middle point. This way, the final tree will be balanced.

The structure allow quick search in a neighbours of a point. For instance, in order to select the points within a certain radius, it is enough to go bottom-top from one of the tree leaves until a point with distance greater than the radius is found.

Octree Like KdTrees, octrees split points spatially by assigning them to nodes in a tree. Each node usually represents a cube, divided in eight "sub-cubes" of equal size, associated to lower nodes, and so on, recursively. The minimum leaf side length is chosen by the programmer. Octree are not usually balanced, and their usefulness relies in quickly navigating between empty and filled space. They are often used in filtering and compression, since they can simplify a large point cloud while maintaining the geometrical information with some losses. They are used in navigation algorithms, where they work similarly to occupancy maps in 2D space.

Filtering Point clouds generated by a lidar are subject to many outlier points, sometime due to sensor inaccuracies or error phenomena, some time due to the presence of actual small bodies in the environment (rain, snow, leaves, paper etc...). Since for the automotive cases we care about modeling large and structured bodies such as people or trees, we filter out outliers using various techniques.

One of the most simple and often the least aggressive consists in just computing the normal distribution of each point's neighborhood in 3D space. Standard deviation and mean are computed, and points outside of a fraction multiple of the standard deviation, (e.g. 1.5x), are removed.

Other filtering techniques, not always suitable depending on the final scope, consists in algorithms based on intensity or on a model reconstruction. For instance, surface can be reconstructed by computing the angle between multiple points. Then points too far from the surface may be removed.

2.3 Euclidean Clustering

The basic Euclidean clustering algorithm, as the name suggest, is based on Euclidean metrics. It works by processing all the points one by one and assign them to a cluster when their distances are within a certain radius. The algorithm fig. 4.4 executes this process:

1. Create an empty queue Q of points
2. For each point in the original point cloud, add the point to the queue Q and:
   (a) Extract a point P from the Q. Add it to cluster C_i
   (b) Add all the points within a certain radius from P to the queue Q
   (c) Go to item 2a until the Q is empty. When empty, cluster C_i is completed

3. Repeat for each remaining point in the original point cloud

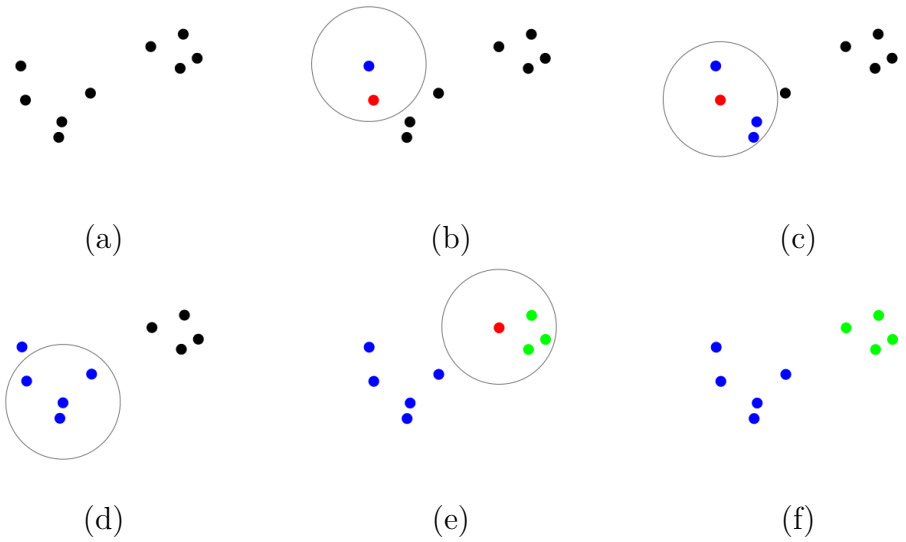


Figure 4.4: Sequence of Euclidean Clustering steps:
A non assigned point is selected (a), points within a radius are added to the queue Q (b), then the same operation is repeat with all the points in Q (c) until the queue is empty (d). At this point, one cluster is completed and the process is restarted for the next cluster (e) until all the points are assigned (f)

The computational complexity of the method is reduced by using a Kdtree, allowing neighbour search.

Once the algorithm is completed, a list of cluster is generated. This algorithm is avaiable in PCL, see pcl::EuclideanClusterExtraction

Euclidean Clustering ROS node In the Autoware[24] framework a ROS layer was added on top of the PCL library. The node **euclidean_clustering** node takes a *sensor_msgs/PointCloud2* as input and output information for each cluster (point clouds, convex hull and bounding box).

The output of this node can then be used as input for object recognition and tracking nodes.

2.4 Tracking with Kalman filtering

Note: for a general overview of Kalman filtering, read section 2

Euclidean clustering is performed at every lidar scan. Given two point clouds A and B and the two set of clusters C_A and C_B generated by **euclidean_clustering**

it is not directly possible to define a correspondence between clusters of C_A and C_B. Comparing the distance between the centroids of the clusters may not be enough (think about two cars moving close to each other). Also, sometime we want to know not only the position but also the velocity of a certain cluster of objects. The way we can obtain this information is to apply a Kalman filter to each cluster. Each cluster will have its own state (position and velocity) and the information is updated with the output of every scan. Since most bodies move horizontally, a bidimensional Kalman filter is used.

When a new set of cluster is generated, the problem of assignment arise: we need to compare each cluster to each previously tracked cluster for position and velocity compatibility. This comparison may result in a match or a rejection. In case of match, the state of the Kalman filter is updated with the new measurements. This gives us knowledge of the speed and direction of other cars relatively to the ego-vehicle.

Assignment is performed in a probabilistic way, using the **Hungarian Algorithm**. By defining a matrix of matching likelihood, the Hungarian Algorithm choose the most probable assignments.

Autoware[24] includes some plugins for **RViz** useful for visualizing the detected objects fig. 4.6, fig. 4.7. The results visualized with these plugins are shown in the attached video.

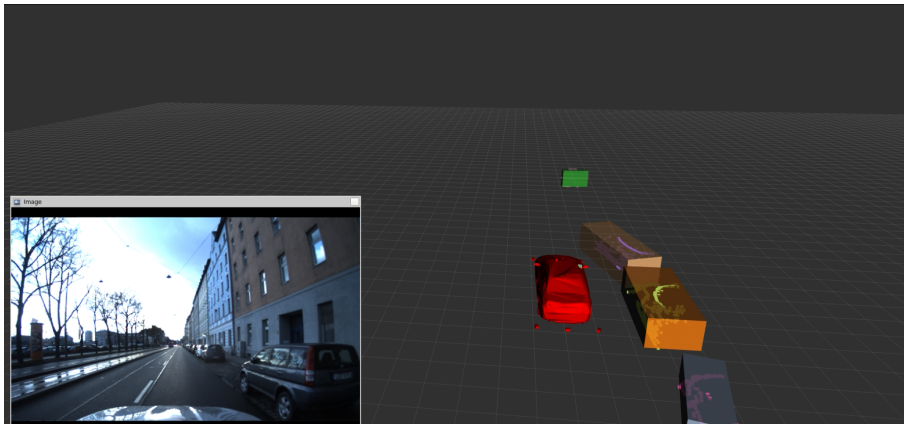


Figure 4.6: The set of points with the same color belongs to the same cluster. A bounding box around the cluster is generated, and that is used for estimating the full space occupied by the vehicle, with some extra margin. Thanks to the kalman filter, it is possible to know the relative speed of each detected box.

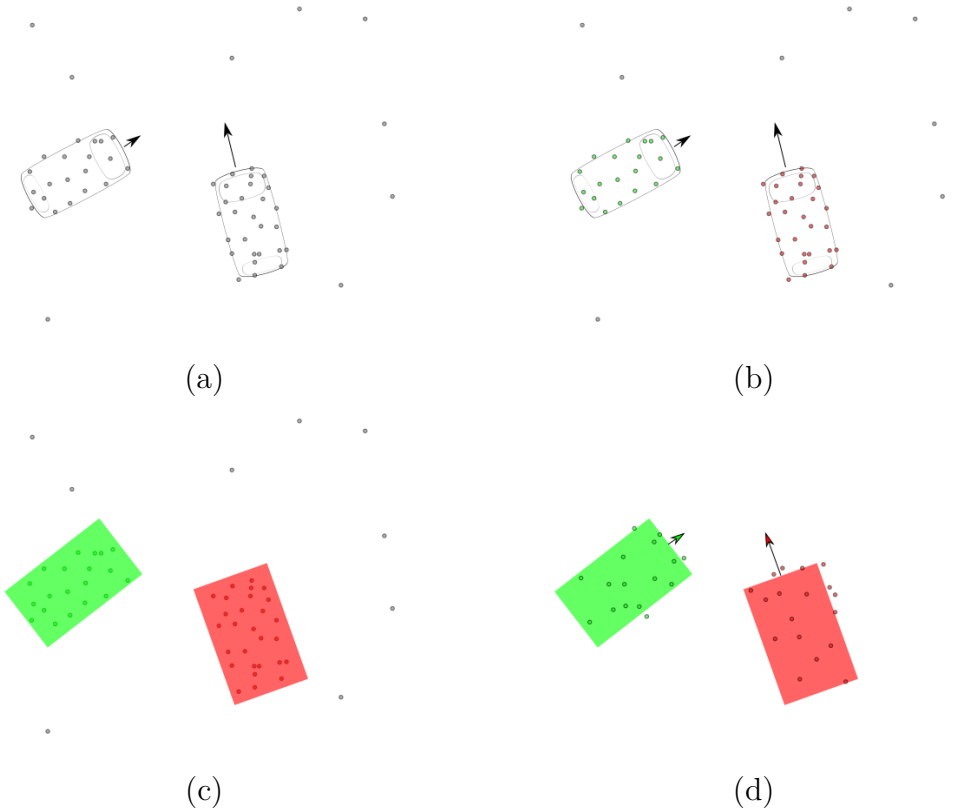


Figure 4.5: Kalman Filter tracking.
After clustering is performed (a and b), instances of the filter are initialized (c). State is then updated based on the data of new point clusters

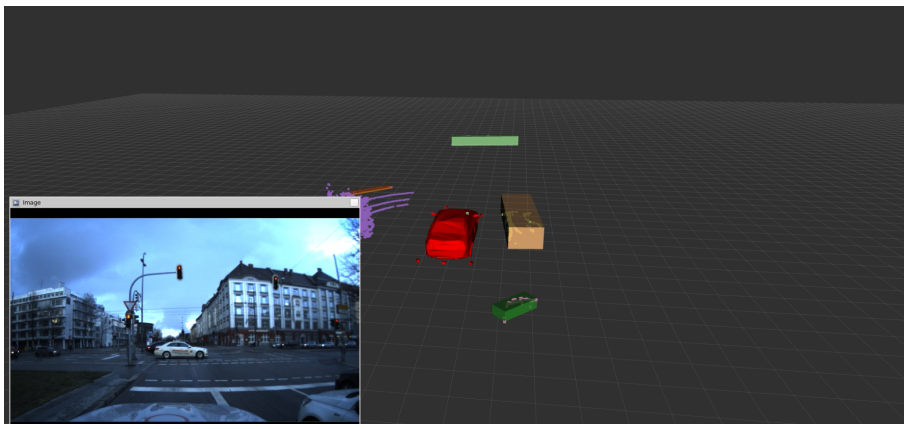


Figure 4.7: The amount of laser available is enough for detecting close objects, but not all the vehicle across a medium intersection. In this image, only the taxi is detected, while vehicles behind it are too far or hidden.

3 Recognition

As stated previously, there are many recognition algorithms, most of them inspired by the corresponding 2D case. We skipped the investigation on algorithms that reduce 3D point clouds to 2D projections, since the density of our point cloud is low. Still, we cite some major works here since they belong to the state of the art.

The algorithms listed here can be applied on all point cloud data. Also note that the Automotive laser scanner solution offers already an object detection capability in its SDK, thus we focused on the VLP-16 laser scanner.

3.1 Projection methods

Due to the fact that many recognition algorithms already exist for 2D images and that they have good performance, it makes sense to apply these algorithms to point cloud data, where the full cloud or just a region is projected to a surface.

In [2] point cloud data is projected to the 2D ground plane, thus obtaining a top-bottom view that it is usually called "bird's eye view". Then a third image, composed by height, intensity and density channel is created. Note however that it is still possible to apply algorithms designed for RGB data. In the **BirdNet** case, the initial **VGG-16** deep neural net layers are used for extracting features from the image. From here, the **Faster R-CNN** is used for region proposal and finally recognition. Detected object may fall in the **car**, **cyclist** and **pedestrian** category. Such method achieves a detection rate from 50.00% to 75.52% [1] depending on the scenario. Instead, [51] projects the point cloud along different point of view. Instead of using a neural network, an SVM is used for classification. [9] mixes the two above projection method in the same architecture.

3.2 Automotive Lidar recognition capabilities

The automotive lidar software, included in the SDK, beyond providing the reconstructed point cloud from multiple sensors, also analyzes the point cloud recognizing peoples, vehicle, bikes etc... It also tracks their position and velocity, probably also with the use of a kalman filter.

We used the C++ API for integrating this data with the rest of the ROS system, so that this can be used as an alternative to the recognition capabilities previously stated.

[1] See kitti dataset benchmark for detection rate definition and other metrics

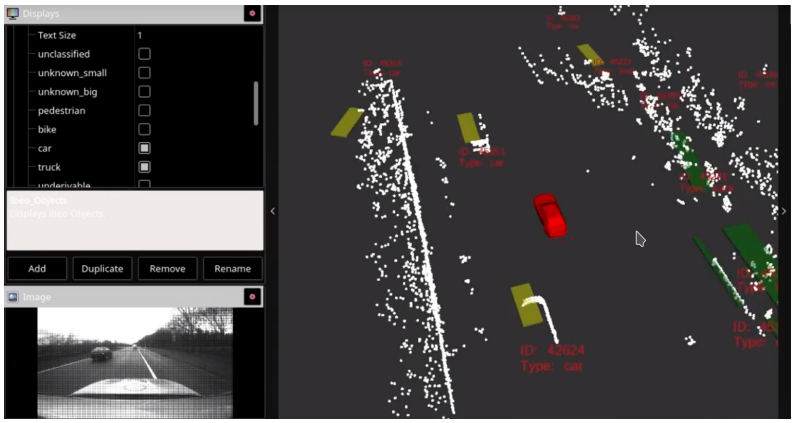


Figure 4.8: ROS visualization of automotive lidar recognition system. Cars appear in yellow bounding boxes

3.3 Lane recognition

The vision for autonomous vehicles in their final, commercially available form sees them capable of performing complex maneuvers and navigating traffic while integrating well with the existing road infrastructure in its usual form. Among these capabilities are reading road markings, changing lane, and exiting from a highway.

The basic environmental information that is needed in order to reach this goal is the recognition of lane markings, in particular of lines.

In this paper, we propose just a basic version of this capability, implemented using simple techniques. Our algorithm will be limited to segments; other road markings, such as text or curved lines, are not recognized.

Our method is constituted by a pipeline of three phases. First, the frontal camera image is submitted to an image segmentation algorithm that recognizes the region of the image where the road is seen. Then, on the selected region of the image, Canny edge detection is applied, after which a Probabilistic Hough Transform [32] recognizes the "most linear" edges.

The image segmentation algorithm, named KittiSeg, is based on deep learning. It's based on the Fast R-CNN algorithm, and can be jointly trained end-to-end to perform the three tasks of classification, detection, and segmentation. It features a high speed of prediction, and has reached a top ranking in the KITTI challenge. [55]

While KittiSeg can be trained from user-provided labeled data, we directly adopted the authors' published model, so to avoid labeling the data manually. KittiSeg's Python code is clean and well-written, and it was easy to adapt it to our purpose. Thanks to libraries commonly used in Python's ecosystem, we could easily transform the bundled demo classification program into a ROS node, so that it feeds the images received from a ROS topic to the algorithm. The result of classification,

a region of interest, is output through another ROS topic as a grayscale image.

The remaining parts of the pipeline, which are Canny edge detection and Probabilistic Hough Transform, are ubiquitous in Computer Vision, and imported directly from OpenCV.

The segmentation and recognition performance is promising, although it has shown some failures on our dataset, and shows much margin for improvement fig. 4.9.

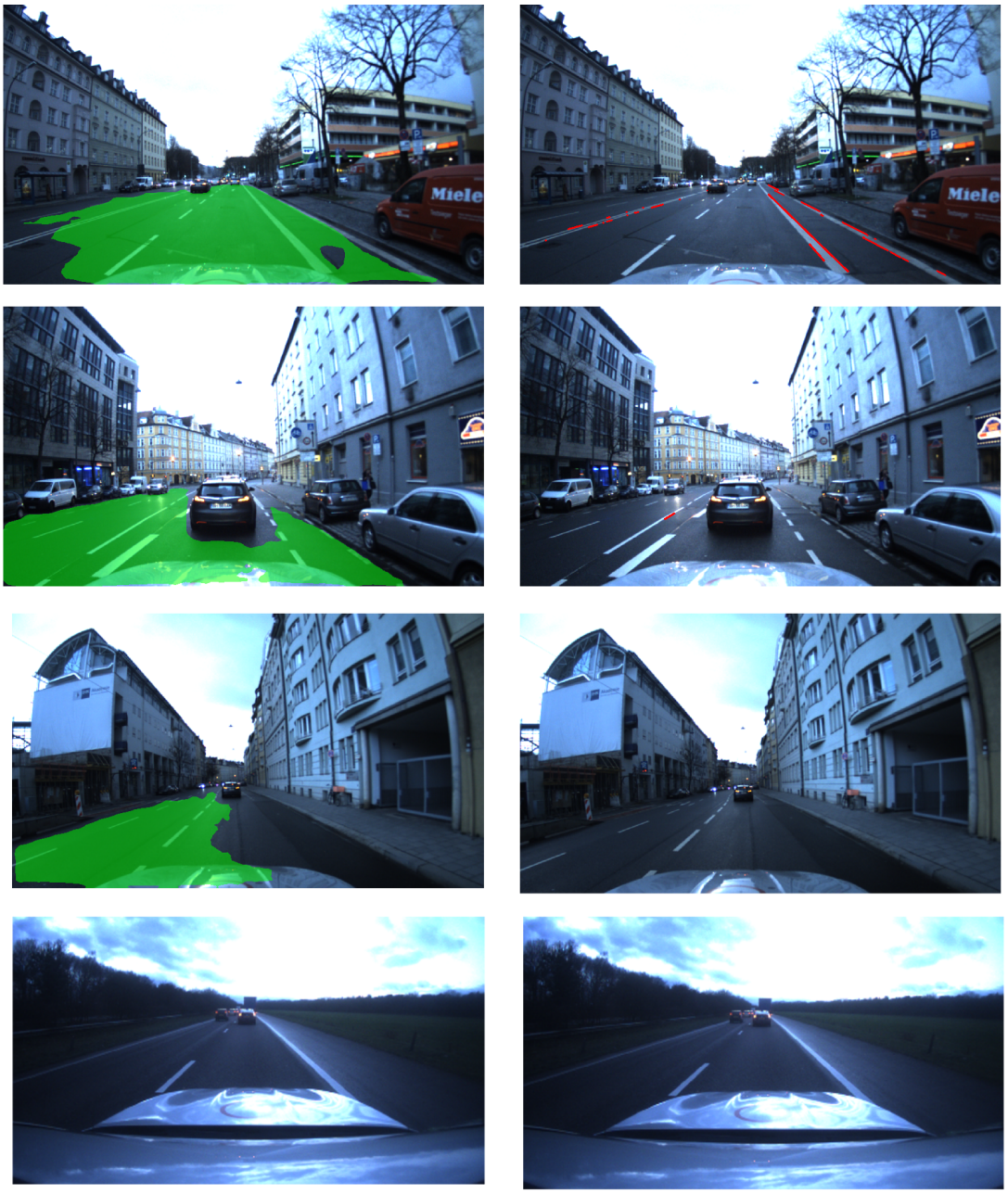


Figure 4.9: Results of lane detection. The green area is the region marked as "road" by KittiSeg. Red segments are the detected road markings. The results have not always been up to the expectation for a production autonomous driving system. In these examples, failures usually happen because the edge detection and Hough transform are not picking up features in the image, or due to bad segmentation.

Chapter 5

Navigation

Provided that a sufficiently accurate map of the environment is available, and that all road users are correctly classified and recognized, the next step in an autonomous car is to evaluate which trajectory to follow. Within this thesis we have deployed a simple model not suitable for actual road, but just for open space not subject to road rules: we still do not consider the meaning of traffic sign (thought they are detected), we do not follow lanes. Such behaviour will be the subject of following work that can exploit the basic work here described. Indeed the possible actions that a vehicle can perform on the road are a subset of all the possible action on a free space.

We highlight that the problem discussed here is about the generation of local trajectory, of length of meters. The road planning as done by commercial GPS navigator that computes the streets toward a goal destination is a different problem.

1 Cost and Occupancy maps

The vehicle is not able to move freely: viable path, collision with static object and dynamic road users as well must be considered when planning a trajectory. The data generated by sensor and the high level detectors/filters in the pipelines previously described must be used for avoiding obstacles. A **navigation algorithm** uses an high level view of the environment in order to plan a tracjectory that satisfy some contraints.

In section 2.2 we already described a data structure useful for this problem. Octree provide a compressed and fast-accessible way to track free and occupied space, where for free space we intend a volume of space that can be occupied by the ego-vehicle without intersecting other bodies. Such solution was implemented in the **Octomap** library [20], which provide, amongst other features, a dynamic 3D map that can be directly integrated in ROS and updated via the standard point

cloud interface. the map also takes care of probabilistically filtering invalid points by performing a continuous integration of the received data.

For the automotive case, where vehicle move mostly horizontally and height is not usually a problem (except for tunnels and indoor parking areas), a 2D occupancy map is often enough for solving most navigation problem. A simple occupancy map is made by a grid, usually with fixed cell dimension, that keeps track of occupied and free space in metric space. Usually the dimension of each cell is little enough to take into account all possible collision. As we will see, the grid dimension also affects the turning capacity and the available path. A really basic map of this kind would be implemented using a two dimensional boolean array.

However, for none-or-little computing efforts, an occupancy map can be replaced with a cost map, which assign a value proportional to the 'traversability' to each cell. Such value would be use for soft-constraints or favour some trajectory against others. Non traversable cell would still be represented with an extreme value (such as a value higher than 250). ROS already provides a costmap system, called **costmap_2d** [1], built upon a plugin architecture. Within the same package, some plugins that process a given point cloud are avaiable. However we choosed to not use them, and instead building our own based on the high-level information generated by the detection pipeline.

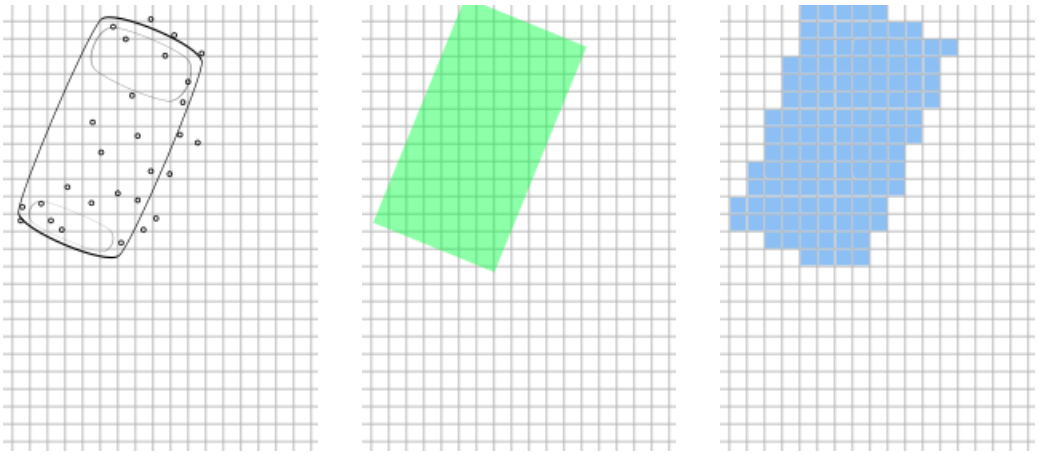
Our costmap subtracts from the free space the space occupied by other road users and static objects. An actual implementation should keep track of the road surface, maybe detected from a camera, in order to compute all the available free space.

The detection pipeline generates bounding boxes of pose (x, y, θ) and dimension (w, h). Our C++ costmap plugin use this data for marking the cells within the bounding box as non-traversable (fig. 5.1). This approach is similar to a rasterization of vectorial geometries into pixels.

2 A*

We used the **A*** (*A star*) path planning algorithm provided by Autoware[24] for navigation. Even if this algorithm is optimal, other algorithm should be considered in the automotive case: algorithms such as **RTT** [30] or **D*** [52] compute non optimal but heuristically optimized path in shorter time, moreover, modified versions of these allow to recompute the path with small modifications with fewer steps when a new obstacle appear. Replacing the A* star with these algorithm will be part of future works.

[1] http://wiki.ros.org/costmap_2d



(a) (b) (c)

Figure 5.1: Generation of costmap from a point cloud. After processing (a) and obtaining the bounding box (b), we mark all the corresponding cell withing the bounding box (c)

The **A*** algorithm works on graphs, thought it is often easier to think about it as an algorithm that works on a 2D grid. The algorithm is well known so we will resume it shortly: it performs a modified depth-first search from a *start* cell to an *end*. The search is guided by a heuristic, that assigns a priority to each unexplored cell. The heuristic is often the simple metric distance from the current cell to the *end* cell.

The bearing of the vehicle during traversal is not considered in the simple A*. But the algorithm implemented in Autoware can be seen as three dimensional: each cell is identified by a (x, y, θ) tuple. This way, the generated path has also a yaw variable associated to each position. The heuristic algorithm is also slightly modified in order to take into account reverse and turns.

In order to adjust the parameters and perform tests on ideal data, we added a functionality for importing a costmap from an image fig. 5.2.

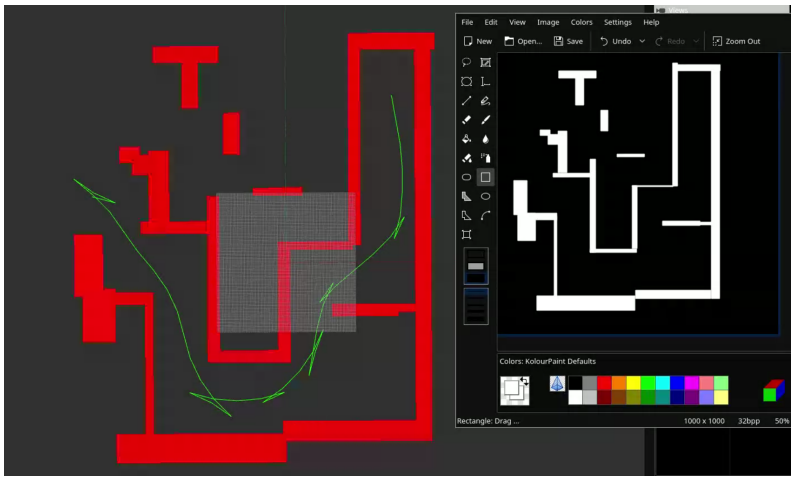


Figure 5.2: A tool for importing a costmap from an image. This allowed us to debug and improve the existing navigation algorithm

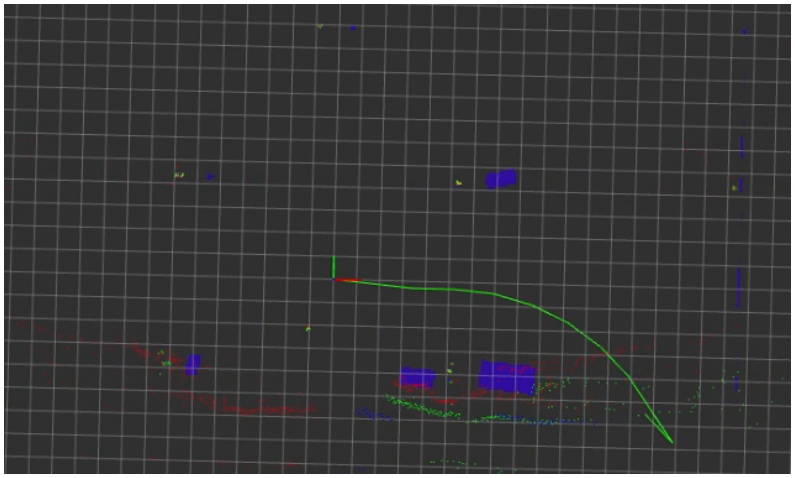


Figure 5.3: Path generated by the A* algorithm (green). Costmap cells are marked blue as described in section 1.

Chapter 6

Deployment and Continuous Integration

One of the main goals of this project is to provide a development environment that can foster innovation and collaboration between people coming from diverse backgrounds, businesses, and research institutions.

The architecture of ROS goes a long way towards that goal, especially in comparison with monolithic (i.e. non distributed) systems that specify APIs as abstractions on programming language primitives (e.g. object-oriented interfaces) rather than system primitives (message passing and inter-process communication in general).

Nonetheless, the effort required to start collaborating is still non negligible, and can be a significant source of friction, especially since it requires, unreasonably, a decent familiarity with system administration and systems programming to all students and researchers.

The following is a non-exhaustive list of sources of friction that lead to this kind of frustrating experience:

- **Build systems.** C++ libraries are packaged and/or imported into a project in a variety of ways. CMake is a very flexible build system generator that helps manage this complexity, but also introduces some of its own. The Catkin tool extends CMake and implements the standardized procedures for building and bundling ROS packages. In order to avoid losing any of CMake's large array of functionality, Catkin makes no attempt at hiding the underlying CMake machinery, and regularly reports CMake's output as part of its own. However, this makes it complex to interpret and troubleshoot error conditions. As a consequence, the developer will need to have a good understanding of CMake, Catkin, and the general way they interact, no matter how far removed this may seem from their scientific or business endeavor.

- **Dependency management at the OS level.** Even when limiting oneself

to binary distributions of ROS packages, there is still a significant chance that the installed packages will have conflicting dependencies. This mainly derives from dynamic linking, an inherently complex mechanism that is highly dependent on the system's configuration. We have observed this first-hand when trying to install different programs that depend on different versions of the OpenCV library, or keeping compatibility with CUDA drivers for use by deep learning programs.

- **Process management and task definition.** All non-trivial ROS systems are composed of a number of separate, communicating process. In the typical case, a single task (e.g. localization, or object recognition) is performed by a small number of mutually communicating processes, forming a sort of "node cluster" that is always started, stopped, and monitored as a group. The standard *roslaunch* utility is very useful to describe these node groups in a standard format based on ROS concepts, but it does not offer any support for isolating the group and turning it into a self-contained unit with a well-specified interface.

- **Run-time platform idiosyncrasies**. Typically, ROS nodes are in equal numbers written in C++ and Python. These two programming languages have radically different requirements and provisions for building, packaging, and distributing software. The situation with C++ has been described above; Python also has its own set of problems, such as dealing with the version 2 vs. version 3 schism, OS packages vs. PyPI packages, native libraries installation, and more. The situation can get even more complicated when the number of platforms increases. For example, the Common Lisp, Julia and JavaScript programming languages can all be considered viable and valuable platforms for writing ROS nodes, and they were all wildly different from each other.

1 Similarities with the microservices paradigm

After an even cursory analysis of these issues it appears evident that these same pain points are shared with virtually any distributed system implemented with native programming languages and/or different language runtime platforms.

This is exactly the case for a large number of web- and cloud-based services. In particular, the popular *microservices* paradigm [1] appears particularly relevant.

It has no formal definition, but for our purposes a generic introduction is sufficient, and it can be considered a variant of the earlier, standardized Service-Oriented Architecture (SOA; see [17]). Its most basic tenet consists in always

[1] Throughout this text, the term *microservices* can also include the earlier *Service-Oriented Architectures* (SOA), as the differences between the two are irrelevant to our purposes here.

factoring the application in a number of small, self-contained *services*, communicating with each other through light-weight protocols. This is the same essential design decision that ROS is based on.

This, in turn, generates a number of similarities. To name a few, they both: advocate for building systems out of processes and inter-process communication primitives; employ external tools for system management, monitoring, reporting, and introspection; rely on protocols to allow (and encourage) the use of multiple programming languages in order to exploit the strengths of each; implement decoupling mechanism in order to be agnostic to the physical location of each process, and therefore achieve the maximum flexibility and horizontal scalability.

Implementers of the microservices pattern generally have to solve the same issues we have listed for ROS (although their motive mainly revolves around difficulties with deployment). In this project, we have exploited the similarity of the problems faced by the two communities, and then borrowed and adapted some of the tools and practices that have become popular for microservice architectures.

To provide an overview of the solutions we became interested in, this is a generic list of technologies and strategies that are typically adopted in a microservices architecture:

- Isolating dependencies by **containerization**. In order for the architecture to be effective, services necessarily have to be independent from the configuration of their run-time environment. This is generally achieved through *containers*, a technology offered by several OSs (most notoriously Linux, but also most BSDs, under the term *jails*) where a single process (or process group) is isolated to a subset of the system's CPU, storage, and network resources, and bundled with a complete operating system installation in the form of a file system *image*. The contained program runs isolated in an instance of the image, which contains all of its dependencies. Since an image usually supports just one or very few programs, it is relatively easy to avoid conflict and maintain internal consistency. A process (or process tree) running with configured resource limitations and with an image as its isolated file system is named *container*, and is usually managed by a dedicated tool such as Docker [12] or rkt [44].

- **Service discovery**. Much of the flexibility gained with a microservice architecture would be lost if the location of each microservice had to be manually and explicitly configured in each of its clients. Instead, the system's components are entirely decoupled by offering a way for each service to locate the servers it requires, and checking for their availability. A single server, whose location is explicitly configured once for the whole system, generally stores and distributes the service's availability and location data. Popular examples of such solutions are ZooKeeper [1], Consul [10], and etcd [58].

- **Standardized service behavior**. Each service should be designed to seek its necessary servers as soon as possible, and degrade gracefully when optional runtime dependencies are not met. Every service should use the same standard logging and monitoring interfaces. This way, the whole system can be managed, monitored and diagnosed effectively with a small set of standard tools. This also reduces the effort necessary to add new services to the system.

- **Automated deployment**. Deployment, meaning the operation of setting up a service on production machines and keeping it running, should be standardized and automated to the highest degree possible. This in turn allows for **Continuous Delivery**, where a releasable artifact is built at each version control commit, with all unit tests automatically re-run and reported, reducing the number of regression that risk emerging in production, and minimizing the difference between the condition of the system in production and in development.

In order to enjoy the same benefits with our robotics platform, we analyzed each of the listed strategies, and attempted to translate them to a ROS-based distributed system. The following subsections offer a recap of our analyses.

2 Containerization solution

We factored our system in a number of **containers**, each assigned to a single "macro-task" (also named *service*; e.g. localization, object recognition, motion planning, etc.).

We evaluated both rkt and Docker as the container management engine. The differences that we deemed relevant to our use case are the following:

- Docker is designed as a client/server architecture, with a system daemon doing the heavy lifting, while rkt is a standalone program managing the container as single child process. In this regard, rkt behaves very similarly to the UNIX shell and init systems, and as a consequence can be composed effectively with existing management tools and policies.

- Docker offers process management out of the box. It allows to easily start, stop and monitor containers and collect their logs without requiring any extra set up. rkt instead allows for a more decoupled architecture, by allowing an external process manager to manage the rkt process and therefore, indirectly, the container.

- While both Docker and rkt offer virtual private inter-container networking, Docker also comes with the popular Docker Compose tool, which allows a

collection of containers to be specified, configured and monitored as a group with a simple and effective command-line interface. rkt again follows the principle of decoupling, and delegates these functions to already existing service discovery and process monitoring systems.

- rkt supports Docker images, but not the other way around. Moreover, rkt allows much greater flexibility in building images, while Docker imposes general principles by offering a purposely limited `docker build` command.

- rkt implements much better security defaults than Docker, for example requiring each image to be digitally signed.

To recap, the advantage of rkt is the greater flexibility afforded by its design choice of exposing a whole container as a single UNIX process to the rest of the OS. This allows to compose a variety of simple tools and strategies to build the desired behavior, but also leads to a slightly steeper learning curve, and makes the work for deployment more complicated (albeit not more complex).

Since Docker already provides all of the functionality we required, and considering how important a smooth user experience is for our goals, we opted for Docker.

By simply running the ROS master into its own container, we immediately ticked several of the other checkboxes in our wish list:

- Thanks to the Docker's inter-container networks, each container has a host name that can be used from within all other containers. By also using Docker's support for environment variables, containerized ROS nodes can connect to the ROS master by following the the URI we supply to their `ROS_MASTER_URI` environment variable. Docker's internal DNS server will route the connection to the correct container, and the ROS master will then direct service discovery, revealing the current location of each node. Under this aspect, Adding new containers does not require any additional configuration effort.

- Standardized service behavior is mostly already provided by ROS. Each service is designed so that it will perform whatever fraction of its functionality is feasible with the current state of the system, and adapt to the availability of dependencies. For example, a localization service will always start up, regardless of the availability of sensor data streams; when those data streams become available (for example, after starting up a sensor-specific "driver" service), the localization service is supposed to pick up the change and start using the available data. This is relatively easy to achieve thanks to the way ROS topics work: as explained in section 1.1, ROS nodes bind their publishers and subscribers to the *topic*'s name, while the ROS master protocol

tracks connections and their endpoints. The underlying ROS protocols will cause connections to be opened and closed automatically, as necessary. Message format compatibility is also automatically ensured by the ROS protocol. Note that ROS maintains this property at the node level, but we make an effort to keep the same property at the service level.

- Deployment is naturally facilitated, as it is Docker's first and foremost use. We created one source code repository for each task, each dedicated to creating one image based on a specific Dockerfile. We then distribute images from an image registry we set up on a private server.

Docker Compose provides one more useful piece of functionality: it allows to script the configuration of a group of inter-connected containers, and then allows to start, stop, monitor, and manage the group of containers. Docker Compose also supports adapting the running system to a changed configuration, reusing the containers whose configuration that hasn't changed, and rebuilds containers only as needed.

It is important to notice that this creates a new level of modularization. The container is the basic unit of software distribution, deployment and run-time composition, while the ROS package is the basic unit of *source code* distribution, build, and assembly.

Development is also relatively unhindered by the use of Docker, as it is entirely possible to install ROS directly on the development machine (outside of Docker's jurisdiction), and use the containerized ROS master. By doing so, every ROS node started on the development machine (i.e. not running in a container) will participate in the same ROS network as the containers, in a fully transparent way and no additional configuration (other than allowing the host to enter the containers' network).

One disadvantage of containerization, especially in relatively constrained environments, is waste of disk space. In order to ensure proper isolation and the self-sufficient nature of containers, many images will contain copies of the same software. Every one of our images, for example, contains a copy of the basic ROS distribution, while many of them include copies of important libraries such as OpenCV and PCL. Our set up featured large amounts of storage, so we never reached a critical shortage. Fortunately, such a shortage can be mitigated by leveraging the fact that Docker images are *layered*, meaning that they are a union mount of many read-only file systems, each being called a *layer*. Layers are produced as part of the image building process; each layer extends another layer (or a bare file system) with the files produced by a single shell command from the Dockerfile, and is labeled with a cryptographic digest. If two Dockerfiles have the same basic image and share a sequence of commands at their beginning, they will usually produce identical layers. Docker can use the cryptographic digest to detect this condition, and avoid duplicates by sharing layers between images.

3 Continuous Integration

In keeping with our idea of borrowing successful practices from mature software development communities, we also adopted a form of *Continuous Integration*. This consists in repeating the build process so that a releasable software artifact with the latest changes in the code base is always available and subjected to testing [8].

In our project, the artifact is the container image. As such, the build process is defined by the corresponding Dockerfile, which in turn runs a complete installation and/or build of all dependencies. The testing happens on different levels:

- **Unit tests**, where single programming language units, such as classes or functions, are tested in isolation with fake ("mock") objects covering the role of dependencies. This is based on programming language-specific tools such as Google Test (for C++) and py.test (for Python).
- **Node tests**, where a small group of ROS nodes are started, and the resulting network gets tested as a group. In this form, what is tested is the interaction between nodes and their individual behavior under certain environmental conditions signaled by ROS messages. The ROS-standard tool `rostest` offers support for node-level tests.
- **System tests** which is similar to node-level tests, with whole containers as test subjects. At this level, it is made sure that the container under test behaves compatibly with the container management system: the service must detect its dependencies at run-time and scale gracefully when some or all of them aren't available.

The process is automatized by using the *drone.io* [13] platform. This is a program that can be installed on-premises and can run Docker-based scripts describing the build and test process, and run it for each commit pushed on a Git repo. It has a web-based interface, which makes it easy to access it remotely from any kind of device. The flexibility and simplicity of its architecture was the main reason for choosing it. Other evaluated options were Phabricator's built-in application ("*Harbormaster*"), CircleCI, TravisCI, Jenkins; the impossibility of installing the program on-premises, complex usage, or relative inflexibility with regard to the organization of source code have driven us away from them.

3.1 User experience

Thanks to the properties of our solution, both newcomers and experienced researchers and engineers can get up and running in short time, enduring an amount of effort that is independent from the complexity of the system at the time.

A hypothetical engineer seeking to add a new capability to the system will first decide on how many new services to create as part of their solution. Typically, a single research topic will result in the implementation of a single service. Then, having installed ROS and Docker on their development machine, and after adding our image registry to their configuration, they will:

- Download or update the images of the services they need, by simply using the command `docker pull` *`image-name`*.
- Create a new service-specific repo, by cloning the dedicated template repository.
- Set the `ROS_MASTER_URI` environment variable so that it points to the containerized ROS master.
- Carry out the development on their machine, without caring much (if at all) for Docker or the containerized parts of the system.

It's highly recommended to use the Continuous Integration infrastructure and practice system-level testing as often as is practical, in order to make sure that a valid, working, and compatible service container can be built and run at all times. For this reason, it is important to keep build times to a minimum. To this end, the Dockerfile has to be kept up to date and it should install the fewest possible dependencies.

Deployment on the vehicle computer is identical. Some services, which can be considered "driver services", are dedicated to translating the interfaces of the car's control hardware to ROS interfaces, and viceversa. This allows to use the full flexibility afforded by ROS to control the car. Refer to section 2 for a detailed explanation of the system's interface with the vehicle.

As a side note, one may notice that each release of ROS is only supported on a handful of Linux distributions, and of each, only a handful of versions (typically just one). For example, the version of ROS that we were using (the latest Long Term Support at the time the project began) is "Kinetic Kame", and it is only supported on Ubuntu 15.10 and 16.04 [26]. Although other ports exist (for macOS, Gentoo, and the OpenEmbedded/Yocto platform), they are experimental, and therefore do not enjoy the same level of support. This strongly limits the compatible OS types and versions that a developer may run. We ran into this problem practically, and we managed to solve it following two types of strategies: the first is using hardware-level virtualization ("full VMs") to create a VM with the exact version of the OS that ROS supports installed; the second was to create a Linux container that acts as a "sandbox" with a supported GNU/Linux distribution installed. The full VM strategy is the only option if the developer wants to use a non-Linux OS (Windows, typically). The container strategy works on any OS based on a relatively recent

Linux kernel, and is more flexible, albeit potentially less secure. Note that the containers used for this purpose are not managed by Docker, but by a different tool such as LXC or `systemd-nspawn`; here, the goal is to isolate an instance of a full OS as if it was running in a virtual machine, but avoiding the virtualization overhead by sharing the running Linux kernel. Docker's goal, instead, is to create single-process, ephemeral containers, eliminate the concern of the process' run-time dependencies, and give it a uniform and accessible management interface.

Chapter 7

Infrastructure

In order to pursue the project's goals, we have set up a number of pieces of infrastructure that have proved to be very useful, to the point of being necessary for our activities, and even more so for future developments. By infrastructure, we mean here any set of shared resources and services that any (current or future) developer can benefit from to aid with their activities.

These resources and services have been located on a dedicated physical server, owned and generously offered by one of the developers (Vincenzo Comito), providing plenty of computing power and storage.

The following sections detail two sets of benefits that we gathered from the use of this infrastructure: the first is *project management*, which allowed us to collaborate effectively; the other is *data collection*, which allows any developer to repeat our experiments, augment them, or try out new approaches without the physical availability of our test vehicle.

1 Project Management

Right after the initial investigation of the literature, the need for instating some project management policies and tools immediately became evident. The project was concurrently developed by the two authors of this thesis, and it presents a significant degree of architectural complexity, in the sense that it is composed by many modules, each with a number of interfaces with different scopes. We could not have collaborated without effective processes and tools.

In order to implement the architectural design detailed in chapter 6, we made extensive use of **Phabricator**, an open-source web-based platform for software development and collaboration. Its main feature is the tight integration of its many constituent *applications*, each dedicated to a particular facet of the software development activity, and the possibility of installing the software on-premises.

We used Phabricator to implement a methodology loosely inspired by Scrum

[50]. We did not use any commercial cloud services, preferring instead to host everything on a private physical server, exposed to the Internet, and assigned to a global host name through a DDNS service. This was facilitated by Phabricator's "monolith-like" behavior with regards to installation and configuration, and the relative lack of required maintenance.

Then:

- We created many source code repositories, all managed by Git and Phabricator's *Diffusion* application. Generally speaking, one repository corresponds to a single ROS source package, while service-specific repositories builds a whole service together with its dependencies, by the definition given in chapter 6.
- We split the entire work into self-contained **tasks** with a well-defined requirement set and "Definition-of-Done" (completion requirements). A few "top-level", principal tasks were recursively split into its constituting sub-tasks, down to the level where the concrete work required was clear enough to be carried out independently by a single person. Tasks are therefore placed in a dependency tree, and only its leaves are assigned to a single developer. We used Phabricator's (*Maniphest* fig. 7.1) application to keep a shared view of the planning.
- We proceeded in *sprints*, which are time intervals of about two weeks, at the beginning of which we met to plan the activity to be carried out during that time period. During those meetings we often estimated the time necessary to complete a given task, prioritized the pending work, and made sure to plan in advance to meet any extra requirement (such as a trip to the office in Munich, the availability of a certain piece of hardware, etc.). Maniphest also supports this methodology alongside many others, by offering the possibility to arbitrarily define milestones, and assign tasks, deadlines, and other attributes to them.
- Whenever we had to create some new code from scratch, we followed the best practices for the platform and programming language. We made sure to give each package a uniform structure, we used code formatting, and enforced a naming convention extensively in order to improve readability. Importantly, we tried to perform **code review** whenever the size of the change would warrant it. This allowed us both to always be aware of the state of the module, reduce the chance for accidental mistakes, and agree on the effect of the change on the module's design. Code review were supported by Phabricator's *Differential* application and the client-side command-line client `arc`.
- During our activities, including development, field testing, literature review, and market research (for, e.g. sensors and software), we produced an extensive set of notes, organized in the form of a Wiki, by using Phabricator's *Phriction*.

It's worth noting that undertaking, in parallel, the effort of implementing the basic perception and localization tasks allowed us to constantly exercise, verify, and improve our decisions regarding the system's structure and our project management policy. The process presented here is only the result of this progressive improvement, which we tweaked and tuned during the entire duration of the project.

An interesting consequence of using a globally reachable server is that it is potentially possible to participate to development and organizational activities from any location. Adding to this our usage of containers, we have reason to believe that it is possible to quickly bring an entire, functional development environment on any computer that is connected on the Internet and runs a relatively modern GNU/Linux OS, although we did not test this solution extensively first-hand.

In conclusion, following this methodology allowed us to implement the system with the structure we initially envisioned (detailed in chapter 6), and to ultimately reap its benefits. Moreover, all other members of the company (in particular other students) that plan to contribute to this project will find a wealth of (loosely) organized information, code, and data, which will hopefully facilitate their activities, including new developments or troubleshooting of the existing work.

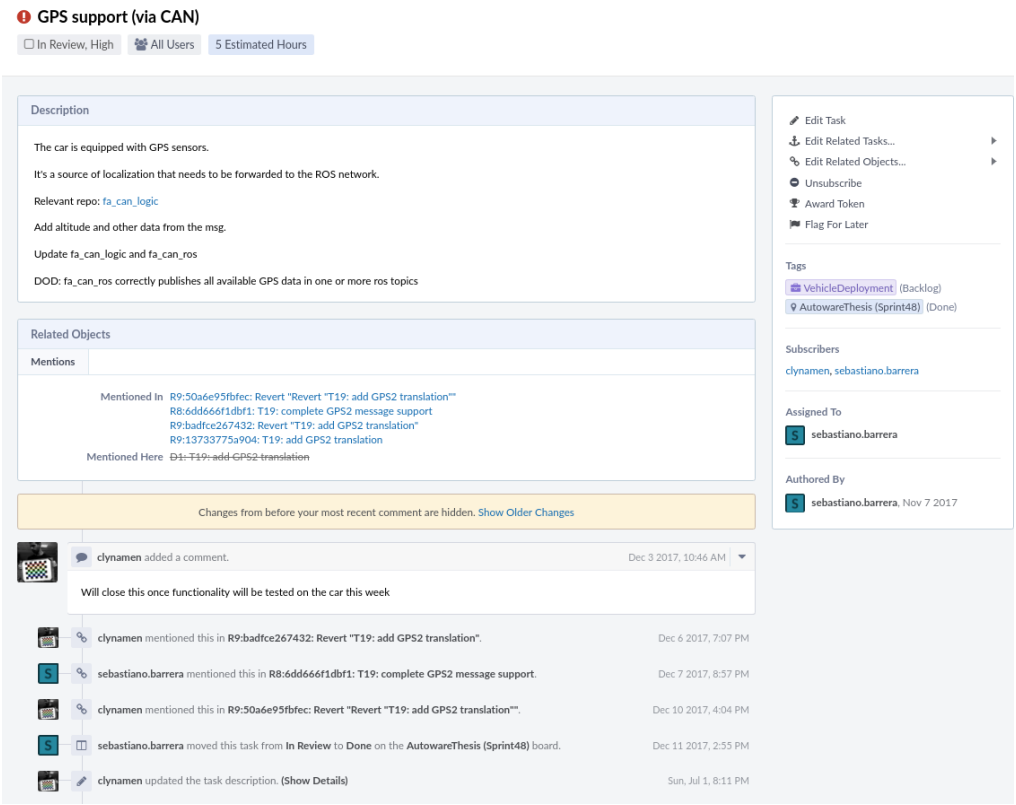


Figure 7.1: An example of a Task within Phabricator's **Maniphest**. The related git commits are automatically linked, along with tracking of the review process

2 Data collection

As stated numerous times, one of the most important objectives for this project is allowing many developers to more easily experiment in the field of autonomous driving and ADAS applications more broadly.

One of the most effective resources that facilitates these activities is a rich collection of data, recorded live from a vehicle, such that new experiments (or replication of old ones) can be performed as though they were available at the time of the recording.

The datasets include all sensors, and the outputs of all of the main nodes. Cameras, LIDAR scanners, localization inputs and estimates (both raw and filtered), object recognition the vehicle's dynamics, and more.

The data collection mechanisms directly descend from the decoupling afforded by the design choice (discussed in section 1.1) of adopting message passing as the principal, most pervasive way of composing the system. Once all inter-process communication is reified into concrete, simple *messages*, those can be trivially labeled, stored and recovered.

The practical side of the collection process is fairly simple, and amounts to not much more than using the standard `rosbag record` tool to record most of the messages flowing through the ROS network while the system is running. The tool will collect the messages by subscribing to any set of topics, much like any other nodes, and store them into a *bag file* in a time-stamped and serialized form, alongside with all *connection headers*. Connection headers are simple data structures describing the topic to the degree necessary to subscribe to it and decode its messages. `rosbag play` is the tool that performs the inverse function: it reads the bag file, extracts indices and connection headers, and publishes the stored messages on the same topics as it subscribed during the record phase, and at the same relative times.

Interestingly, `rosbag play` is an important part of *simulated time*. In the general case, ROS nodes are supposed to get the current time from a dedicated topic (`/clock`), rather than the OS. This way, the current time can not only be directly translated from the OS, but also provided by another external program. This allows the user to replay a bag file slower or faster than its natural speed, or skipping forward or backwards along the time line. This mode has to be explicitly enabled by setting the parameter `use_sim_time` to `true`, which is honored by all ROS client libraries. `rosbag play` can then publish `/clock` and direct the passage of time on the system.

Several ROS bags have been recorded and stored on the server, with a standardized file name based on the date and time of recording. We did not limit ourselves to simply producing the records. An additional web-based service was implemented where any authorized user can connect and view an interactive web page associated to each recorded bag, summarizing the bag's contents. This includes a few useful features: other than the date and time of recording, the page includes: a

table listing the topics with their message types (linked to the corresponding on-line documentation), number of message and average frequency; the recorded videos, streamed and directly viewable from the web browser; plots of the localization estimate, overlaid on a geographical map (the map is fetched from OpenStreetMap). This is just one more way to facilitate the browsing of these relatively big datasets.

One important property of ROS' implementation of message passing is that, from the point of view of any ROS node, messages representing "fresh" computation results coming from another running node are entirely indistinguishable from messages recovered from a bag file from `rosbag play`. This creates a lot of opportunities for reusing our datasets for a very large range of experiments and set ups.

To make a few examples, the following are all possibilities opened by the collected data sets. A student looking to investigate a new implementation for object recognition that combines images from the camera and point clouds from laser scanners can replay those inputs from a ROS bag from the server, and compare their results with the current implementation of object recognition. A researcher investigating innovative applications of machine learning can use the data from any ROS bag to extract labeled datasets to be used as the training set of a supervised learning application (such as those in the category of deep learning), replacing solutions of other kinds (for example, traditional computer vision) that are more difficult to develop.

Chapter 8

Conclusions

Though the main aim of the work was fulfilled, having reached the core target that we proposed at the beginning of the thesis, we regard these conclusions as a milestone step instead of a closure of the project, which we hope to continue along with new students.

The proposed distributed architecture, based on ROS and similar to the popular microservice pattern, has been proved to be versatile and well suited for research and prototyping, and, component after component, evolved in a full sensorial systems similar to the one found in many self-driving car platforms, albeit very far from them with regards to performance and accuracy.

The features of the developed system are:

- Support for accessing main vehicle data (section 2)
- Customizable, portable sensorial system that can work with a variety of sensors and allow to add new ones (section 3)
- Fusion system for localization (chapter 3)
- Object detection through camera (section 1) and lidar (section 2)
- Mapping capabilites (section 4 and section 5)
- Basic support for navigation (chapter 5)
- Fast system development and deploy, by simplifing access to resource and libraries (chapter 6)

We have to keep in mind that it was possible to integrate all these functions in a comparatively short time thanks to the availability of many, high quality open

source packages, respectively cited throughout the previous sections. In this sense, this thesis can also be regarded as a survey of the capabilities and limits offered by open software at this date.

Due to the many aspects and components involved, we have to draw different conclusions for the specific results of every subcomponent in the dedicated section. Some of these fully met our expectations: we observed it was possible to accurately generate small but composable maps of urban scenarios based on lidar, that are also able to detect objects close to the vehicle, providing useful spatial data that can be used for avoiding them.

The camera detection system based on neural networks seems promising, capable of quickly processing images on medium-end hardware in real-time and low frame rate, but not enough reliable on a per-frame basis.

Other components such as the localization system suggest that sensor fusion works well only on small scales, where variations of the position of the car are correctly estimated with accuracy greater than GPS, for instance when changing lane or during short accelerations.

Beyond specific improvements that regard the limitations throughout described, wesuggest some specific future work in the next section.

Chapter 9

Next steps, future work

One of the aim of this thesis was to develop a basic perception system for an autonomous vehicle that can be easily extended by updating or adding new components, thanks to the availability of both raw and already processed high level data. Even if the system requires many improvements, we think the system meets the proposed requirements.

As a result of using ROS as framework and of the orthogonal, loosely coupled design of the system's components, it is possible to change a fully functionality or just a piece in the pipeline, making it feasible to try new algorithms and quickly prototype new methods. It is both company and our aim to exploit these capabilities in the future within novel projects or thesis research of new students. For instance, the obstacle avoidance is being already reviewed by two master students for their thesis. They will replace the existing component with a new one that predict future poses of road agents, extrapolating them from their situational behaviour.

The field of research is so broad that many improvements come in mind. However we report here the ones that we believe are the most interesting and that should be prioritized over others.

1 Real-Time applications and ROS2

The versatility of the system comes with a tradeoff: it runs on a linux computer with variable performances, practically with no memory limits and timing requirements. There is no need to say that such a system is not good for basic vehicle control, let alone for a refinished product for a normal user. The automotive industry must comply to high safety requirements and currently system meets none of them. For our scope, this is not a problem since we are more interested in research

than in developing a product, nonetheless real-time capabilities will be required for controlling the vehicle and implementing autonomous driving. While moving the whole system to real-time computers or microcontroller and maintaining the same versatility at the same time seems impossible, the use of the future ROS2 framework looks like a promising compromise.

ROS2 [45] is a full redesign of the current ROS system, focused on solving some problems and review part of the system architecture, that currently limit the use of ROS in industry and in multi-robot system. The main characteristics of **ROS2** will be:

- Distributed communication system (there is no central master)
- Direct communication with microcontrollers
- Quality-of-service support
- **Real-time support**

Before describing what ROS2 will offer in term of RT support, we have to remember the limitations that a modern computer imposes versus a simple microcontroller. One limitations is related to hardware: A computer CPU is way more complicated and cores fight among them and other units for accessing some resources, for instance the PCI bus, memories. Non-modificable firmware are not managed by the operating system and they can introduce random delays [1]. By default, common user operating system (ROS works only on linux and partially on windows), are not realtime: The kernel has always top priority and cannot be preempted, memory allocations are hidden, timer are accurate up to a point. For linux, it is possible to configure the kernel in real-time mode thanks to the **CONFIG_PREEMPT_RT**[2] patch, which solves the above problems. Still, it is never possible to know latencies in advance and the RT system designer can only assume worse-case limits, usually measured statistically.

ROS2 exploits the CONFIG_PREEMPT_RT patch, allowing to use RT-threads and configure the system in order to use prefixed, static structures and memories. This would allow a developer to design, for instance, a control system with some guarantees in term of deadlines and performance. However, due to the overall indeterminism that would still affect the system, ROS2 should be regarded as a soft-RT more than a proper hard-RT system.

[1] https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions

[2] https://mirrors.edge.kernel.org/pub/linux/kernel/projects/rt/

In future we would like to move the current system to ROS2 for implementing actual controls on the vehicle, such as cruise control or lane keeping. As of today, the first version of ROS2 was already released but the most interesting functionalities are still work in progress.

2 Visual odometry and image segmentation

Image segmentation consists in labeling each pixel of an image indicating it belongs to a certain class of object, like the road surface or a vehicle. The technique is relatively new and using deep learning it is possible to apply it on high resolution image.

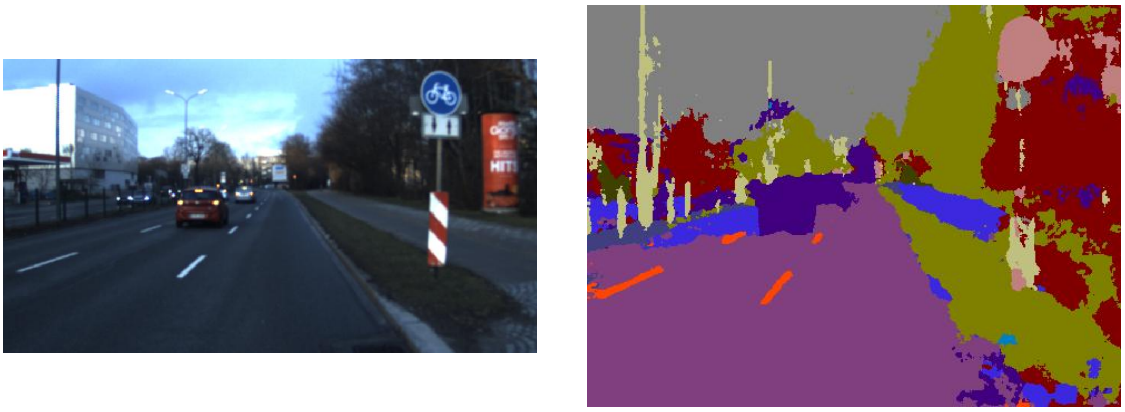


Figure 9.1: Example of image segmentation applied to our images. Road surfaces, vehicles, trees or grass, and other elements have a different color indicating they belong to different classes

As we suggested in section 4.3 an error source in visual slam system is to include in calculations of the camera movement some features belonging to moving object such as the other vehicle on the road. Using image segmentation, it would actually be possible to exclude such points, creating a non uniform region of interests (ROI).

Figure 9.2: Mock-up of the image segmentation ROI: cars are excluded. This remove 'noise' in the map

3 Correlative scan matching for lidar SLAM

The NDT technique presented in section 5.2 has been surpassed by new algorithm such as the one described in [37]. The Real-Time correlative scan matching of Olson when applied on GPU has better performance and lower execution times. The idea behind consists in searching in a multi-resolution, discretized (x, y, θ) space for the maximum likelihood pose that can explain the current lidar observation against the map known so far (pre-generated accumulating points).

The map is not made by a point cloud, but it is a rasterization of the log probabilities of lidar observation. The rasterization of course, has higher value close to the scanned points. The map can be displayed as a 2D grayscale image. The search in every discretized point of the space is handed to the GPU, capable of computing in parallel the correlation for each possible pose. The search is repeat at multiple resolution, where low-probable solution at lower resolution are discarded in order to save time. This trick and the use of GPU allow the matcher to execute in short time on modern hardware, thus the 'Real-Time' of the name.

Appendix A

Common Mathematical Methods

1 Newton's algorithm

Also known as **Netwon-Raphson method**, this algorithm can be applied to differentiable functions for obtaining approximations of the roots. For a real-valued one parameter function $f(x)$, the algorithm consists in iteratively evaluating:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Until $f(x_n)$ reaches a value sufficiently close to 0. In the first iteration, $X_n = X_0$ where X_0 is a sufficiently close approximation of one root.

More importantly for today research, the same algorithm can be applied to the first derivative of the function $f\prime$, thus allowing to evaluate local minima (or maxima). So the algorithm is used as one of the most common and simple optimization techniques:

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

Of course, it is up to the user to make sure the first approximation x_0 is close enough to a global minimum via other methods. See [5] for further details

For higher dimensions, the algorithm can be extended by using the gradient $\nabla f(x)$ and the hessian matrix $\mathbf{H}f(x)$:

$$x_{n+1} = x_n - [\mathbf{H}f(x_n)]^{-1} \nabla(x_n)$$

2 Gradient Descent

As the name may suggest, **gradient descent** methods consists in following the direction of the function gradient in order to iteratively descending to a local minimum. If the starting point is enough close to a global minimum, then the algorithm

is a good (but not fast in comparision to many other algorithms) method for finding the minimum of a function.

The method uses only the gradient, that is, the first derivative of the function, thus it is often easy to apply to all functions, also high dimensional ones. Especially for the machine learning and deep learning scope, many improvements to the original gradient descent were proposed, for instance **AdaGrad**, **Adam**, **RMSProp**.

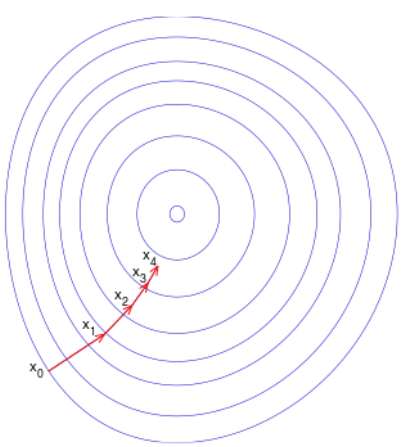


Figure A.1: Gradient descent iteratively follows the gradient in the direction to a local minimum - [*Wikipedia*]

Given a multidimensional function $f(x)$ and its gradient $\nabla f(x)$, at every iteration the gradient is subtracted to the current x_n:

$$x_{n+1} = x_n - \gamma \nabla f(x)$$

where γ is a parameter that can be varied at every iteration. For higher γ in principle the algorithm should approach faster to the minimum, however it may also induce the algorithm to go back and forward around the minimum, where smaller step would be needed instead. There are various technique for choosing γ at every iteration.

3 Levenberg–Marquardt

Levenberg–Marquardt is a method for non-linear least squares, that is, a method for function fitting with the exception that the proposed approximating function is non-linear.

Like in nonlinear least squares, given a set of N samples $(y_k,\ x_k)$ and our approximating function $f(x, \beta)$, dependent on parameters β, we want to optimize the parameters of $f(x, \beta)$ so that the sum of squared error C is minimum:

$$C = \sum_{k=1}^{N} y_k - f(x_k, \beta)$$

The Levenberg-Marquardt search for a minimum by both using the second and first derivative of the objective function, where the latter is weighted by λ :

$$\left[\mathbf{J}^T\mathbf{J} + \lambda diag(\mathbf{J}^T\mathbf{J})\right] \delta\beta = \mathbf{J}^T \left[y - f(\beta)\right]$$

Note that:

- We are not optimizing over x, but over β
- $\delta\beta = \beta_{n+1} - \beta$
- We are not optimizing $f(x)$, but C
- $(y_k,\ x_k)$ are given

It is interesting to acknowledge that the method is actually an interpolation between the Newton (due to the second derivative) and the gradient descent (first derivative) method. The interpolation is regulated by λ. Indeed by setting $\lambda = 0$ we have:

$$\begin{aligned} \mathbf{J}^T\mathbf{J}\delta\beta &= \mathbf{J}^T \left[y - f(\beta)\right] \\ \mathbf{J}^T\mathbf{J} \left[\beta_{n+1} - \beta_n\right] &= \mathbf{J}^T \left[y - f(\beta)\right] \\ \beta_{n+1} &= \beta_n - \mathbf{J}^T\mathbf{J}\mathbf{J}^T \left[y - f(\beta)\right] \end{aligned}$$

$\mathbf{J}^T\mathbf{J}$ is the Hessian of first order linearization of the objective function. Thus the similarity to the Newton algorithm.

See http://people.duke.edu/~hpgavin/ce281/lm.pdf for more details about the method.

Appendix B

Deep Learning

Due to the intensive use of deep learning in detection and recognition system of today self-driving vehicle, we add here the details that can be used as a short reference. For a more in-deep explanation of the topic the reader can refer to [16]

1 Neural Networks

Neural networks are machine learnings methods, usually represented by a graph of node of various, layered shape. Node in the network (**neurons**) represents functions, called **activations**, that usually are single-value non-linear functions. Edges in the network instead represents the input-output connections between these node: the output of a node is used as input of a node in the next layer. All the inputs are weighted by a value that is associated to the edge. The full set of these values is called **weights** of the network. These are the values that are optimized during the training phase of the network. The structure of the network is purposely designed for a defined problem domain and in the latest years it is possible to see more and more complex structures, with more layers and memory elements (See RNN).

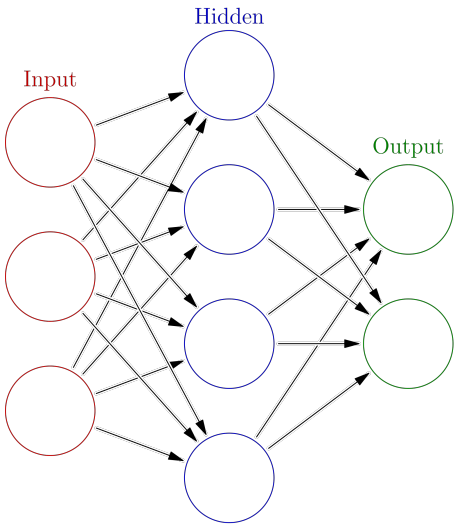


Figure B.1: Structure of a simple neural network. Hidden layers in today networks are much more complex.

As any machine learning methods, input data is an array of values. The **input layer** is the first layer of the network and represents this array. The output layers generate the output values of the network. In some cases these are the final results of the machine learning algorithm, in other there is a further function such as softmax, that normalize the output or transform it a probability value.

The earliest forms of neural networks were **fully connected**, that is, every neuron was connected to all the neurons of the previous and next layer. This seems natural, since eventually some connections will be ignored after the training phase when the weight assigned to the edge is close to 0. However, it was later demonstrated the advantage of define a pattern of connections between the layers in order to exploit the locality of some informations. This is very important for visual applications where it is important to recognize lines, patterns, then shapes and higher and higher level features in increasing layers of the network. This will be detailed in appendix 2

Common activation functions are the *sigmoid*, the *hyperbolic tangent* (*tanh*) and the *rectified linear unit* (*ReLU*), which is the most preferred nowadays since it is the most simple to compute and also it cause the tendency to make models simpler.

The operation that starts from input neurons to the final output is called **feed-forward**.

At each neuron, the following operations are carried out:

$$o_j(t) = f_{nl}\left(\sum_i o_i \cdot w_{ik}\right)$$

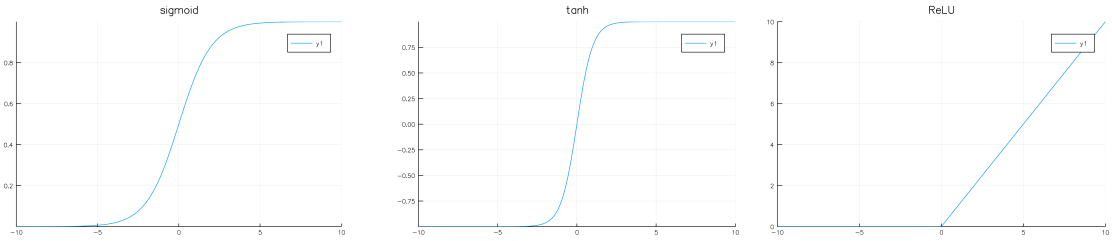


Figure B.2: Three common activation functions

That is, we multiply the output of each neuron o_i of the previous layer by the weight of the corresponding edge w_{ik}. The weighted average is then passed to the non linear function f_{nl}. Weighting averages between each layer are carried out in form of matrix multiplications. Activation function calculations are quite simple and independent among neurons. Thus all the operations can be performed in parallel at each stage. This is why **GPU**s and the ad-hoc designed **TPU**s (Tensorial Processing Unit) performance in the field are times higher than CPUs. These processors are also much faster in the **training phase**. During this phase, which can take days even on high-spec hardware, the weights are optimized in order to obtain predictions that match values or categories of a training phase (in the supervised learning case). One major topics in academic research is finding the various method for making optimization faster and robust to overspecialization.

Fortunately the calculations for the feedforward step can be often easily performed by standard hardware in milliseconds. Thus the cost of neural network is only related to the design and training phases.

Standard fully connected neural networks were applied, in the early years of their use, for control systems involving non-linear functions. Recognition capabilities were proven with **deep neural network**, and **convolutional neural network** specifically[28], some years after and then rediscovered lately thanks to the new availability of massive parallel processors.

2 Deep Neural Network

The number of layer in a network is called **depth**. When multiple hidden layers are used we usually call the network *deep*. The operations performed in the hidden layers of a deep neural network are more complex of usual: Not only locality of data is exploited using convolutional layers, usually also these operations are performed:

- **pooling** consists in replacing a set of nearby outputs with a unique value. In max pooling the maximum output of a neighbourhood is used. Weighted averages are also possible but less common.

- **dropout** of outputs, forcing them to zero. This is a simple regularization techniques.

- **concatenation** of multiple outputs from different layers into a single input set.

For image applications, it is useful to represents neural network layer as three dimensional boxes. For instance, the input layer of a neural network that works on a 2D image is usually a $Width \cdot Height \cdot 3$ box, where each value is the value of a Red, Green or Blue pixel. A neuron in the first convolutional layer, for instance, would be connected to a patch of 3x3 pixel in the image (fig. B.3)

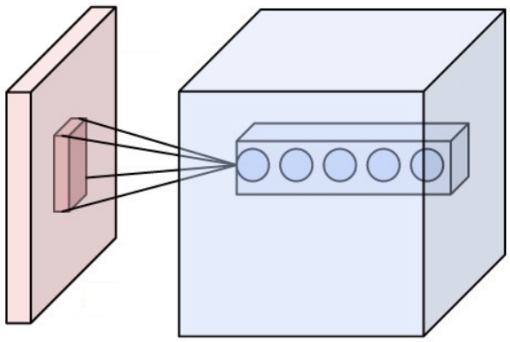


Figure B.3: Example of convolutional layer. Each neuron of the blue layer has as input the output of a path of the previous layer. Note that also the blue layer has three dimension. Along the width and height dimension, all the neurons have the same patch but different weights. [*Wikipedia*]

The operations above are mixed together by the designer of the networks, with experience and guidance of trial-and-error approaches. The sequence of these operations, along with the size of each layer, is called **architecture** of the network. One of the earliest architecture [28] is composed by convolutional, subsampling and fully-connected layers fig. B.4. It obtained a fairly good result in recognizing hand-written digits. More recent architectures may be more complex fig. B.5, such as Google's inception [53], proposed for image recognition task.

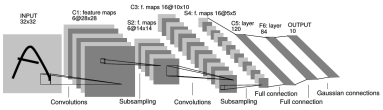


Figure B.4: (a)

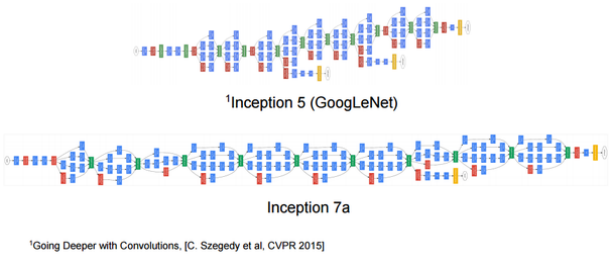


Figure B.5: (b)

Figure B.6: Example of two deep convolutional neural network. LeNet (a) appears really simple today. Some architectures as inception (b) can be far more complex.

Appendix C

Geodetics and projections

We report here a few notions about geodetics due to the use of GPS and interests in localizing the vehicle in Earth global coordinates. There are many ways to describe the earth shape, position around the surface, distances and other measures. Each of them has better properties for a certain scope. For coordinates in automotive localization it is often common to switch between **Latitude/Longitude** representation and the **UTM** coordinate system.

1 Latitude/Longitude

The classical spherical geometry representation is a direct method for describing points on earth, usually approximated by an ellipsoid. Any position is given by the tuple $(lat, lon, height)$. However, these values are actually meaningful only when associated to a **geodetic datum**, that is a given ellipsoid defined by the length of the semi major axis a, the flattening f and the origin. As of today, especially for any consumer, the datum is never specified since it is implicitly **WGS84**. This datum has origin in the Earth's center of mass, and axis, flattening parameters given by approximated measures of Earth performed through various techniques around 1980. Being it used in GPS, it became the de-facto international standard.

Other datum exists, albeit less used. The same location on earth may differ a lot depending on the datum. For instance in Spain some positions differs up to 100 meters in the WGS84 and the older **ED50** system.

The use of a spherical system is not always well versed for vehicle localization, since it is much more convenient to use a Cartesian system that makes possible to evaluate distances, angles and the conversion with local sensor frame of references. It is thus necessary to also use a **projection**.

2 UTM

The **U**niversal **T**ransverse **M**ercator is quite similar to the better known **Mercator** projection, which has the good property of conserving angles. The original projection distorts size however (for instance, the size of lands around the poles). UTM instead has less distortions by splitting the whole Earth's surface in sixty zones along longitude and 20 bands along latitude (with some exception, it is not a proper grid, see fig. C.1)

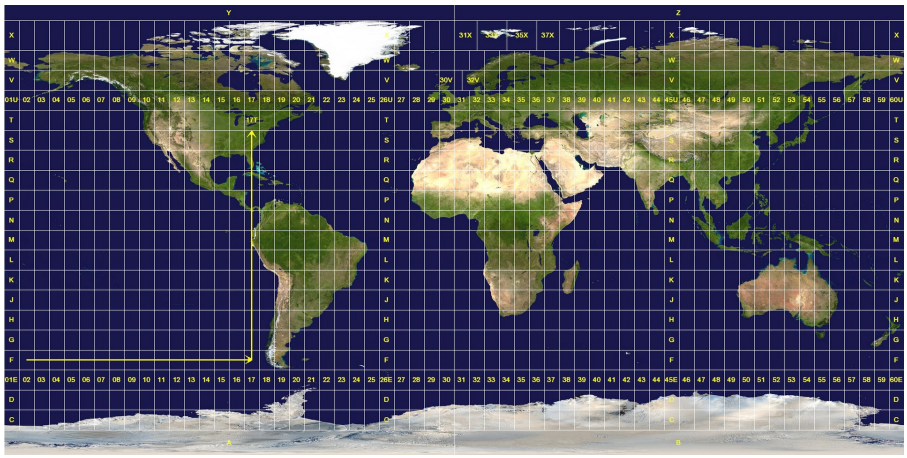


Figure C.1: Full grid of UTM - [*Wikipedia*]

Locally, each cell of the grid maps a corresponding region of varying area (bigger close to the equator). In every cell, coordinates are defined in **easting** and **northing**, measured in meters. The easting start at 500 000 meters at the center meridian of the UTM zone, northing starts at 0 at the equator for norther bands and at 10 000 000 meters south of the equator for souther bands. Coordinates are **always positive** and are expressed by including the number of the zone, the letter of the band. For instance, Munich is at **32U 691 650E 5 334 754N**.

Withing a UTM cell, distortion (when ignoring the height change) are low enough to easily measure distances with normal euclidean metrics. Still, one should pose particular attention at the **grid convergence** problem (also known as meridian convergence): The north direction in a point over the UTM zone surface does not always point to the true north direction. Indeed, the vertical line at the meridian of the zone correctly matches it, but on the east or west the angle between the true north and the vertical line progressively change. Given the latitude ϕ and longitude λ of a point, the longitude λ_0 of the UTM zone meridian, the angle difference is $\gamma = arctan\left(tan\left(\lambda - \lambda_0\right) \cdot sin(\phi)\right)$

Acknowledgements

We thank **Objective Software** for its support, provided both in form of resources and guidance, giving us the chance to test all of our ideas. The company also supported and helped us with technical and managerial advices since the proposal till the review of documents and presentation material, sharing the enthusiasm for the little advances obtained now and then.

Sebastiano Barrera, Vincenzo Giovanni Comito

Writing this note gives closure to the long, intense experience that this thesis proved to be. It's been an invaluable source of experience, learning, and growth. For this, many people deserve my gratitude.

I would like to thank **Team DIANA**, the robotics team I used to proudly be a part of, for the great learning opportunities, the hard work, the trips, and the friendships.

Finally, I thank my father, my grandmother, my entire family, past and present, and of course my friends, for helping me along the road and for bearing with me during the many times when I thought this work would never end.

Sebastiano Barrera

I would like to thank prof. Giancarlo Genta for his **Team DIANA** and all its past, present and future students. Being part of the team was an invaluable experience, it shaped me personally and professionally, allowing me to work with extraordinary colleagues and friends. I hope the Polytechnic will continue its long tradition of supporting these initiatives.

I thank my family for the continuous support and encouragement, and especially my mother for the patience and her ability to understand me and accept all my decisions.

Vincenzo Giovanni Comito

Bibliography

[1] *Apache ZooKeeper*. URL: https://zookeeper.apache.org/.

[2] J. Beltran et al. «BirdNet: a 3D Object Detection Framework from LiDAR information». In: *ArXiv e-prints* (May 2018). arXiv: 1805.01195 [cs.CV].

[3] P. J. Besl and N. D. McKay. «A method for registration of 3-D shapes». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2 (Feb. 1992), pp. 239–256. ISSN: 0162-8828. DOI: 10.1109/34.121791.

[4] P. Biber and W. Strasser. «The normal distributions transform: a new approach to laser scan matching». In: *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*. Vol. 3. Oct. 2003, 2743–2748 vol.3. DOI: 10.1109/IROS.2003.1249285.

[5] J.F. Bonnans et al. *Numerical Optimization: Theoretical and Practical Aspects*. Universitext. Springer Berlin Heidelberg, 2013. ISBN: 9783662050781. URL: https://books.google.it/books?id=1ffvCAAAQBAJ.

[6] P. Bonnifait et al. «Data fusion of four ABS sensors and GPS for an enhanced localization of car-like vehicles». In: *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*. Vol. 2. 2001, 1597–1602 vol.2. DOI: 10.1109/ROBOT.2001.932839.

[7] Cesar Cadena et al. «Simultaneous Localization And Mapping: Present, Future, and the Robust-Perception Age». In: *CoRR* abs/1606.05830 (2016). arXiv: 1606.05830. URL: http://arxiv.org/abs/1606.05830.

[8] L. Chen. «Continuous Delivery: Huge Benefits, but Challenges Too». In: *IEEE Software* 32.2 (Mar. 2015), pp. 50–54. ISSN: 0740-7459. DOI: 10.1109/MS.2015.27.

[9] X. Chen et al. «Multi-View 3D Object Detection Network for Autonomous Driving». In: *ArXiv e-prints* (Nov. 2016). arXiv: 1611.07759 [cs.CV].

[10] *Consul by HashiCorp*. URL: https://www.consul.io/.

[11] Texas Instruments - Steve Corrigan. *Introduction to the Controller Area Network (CAN)*. URL: http://www.ti.com/lit/an/sloa101b/sloa101b.pdf.

[12] *Docker - Build, Ship, and Run Any App, Anywhere*. URL: https://www.docker.com/.

[13] *Drone · Continuous Deliver*. URL: https://drone.io/.

[14] J. Engel, T. Schöps, and D. Cremers. «LSD-SLAM: Large-Scale Direct Monocular SLAM». In: *European Conference on Computer Vision (ECCV)*. Sept. 2014.

[15] Dieter Fox. «KLD-sampling: Adaptive particle filters». In: *Advances in neural information processing systems*. 2002, pp. 713–720.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[17] The Open Group. *Open Group SOA Source Book*. http://www.opengroup.org/soa/source-book/intro/. Van Haren, 2009.

[18] Oguzhan Guclu and Ahmet Burak Can. «A Comparison of Feature Detectors and Descriptors in RGB-D SLAM Methods». In: *ICIAR*. 2015.

[19] Wolfgang Hess et al. «Real-time loop closure in 2D LIDAR SLAM». In: *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1271–1278.

[20] Armin Hornung et al. «OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees». In: *Autonomous Robots* (2013). Software available at http://octomap.github.com. DOI: 10.1007/s10514-012-9321-0. URL: http://octomap.github.com.

[21] Jared Hulme et al. «Fully integrated hybrid silicon two dimensional beam scanner». In: 23 (Mar. 2015).

[22] Prof. Dr. Daniel Cremers Jakob Engel. *LSD-SLAM: Large-Scale Direct Monocular SLAM*. Computer Vision Group, Technische Universität München. URL: https://vision.in.tum.de/research/vslam/lsdslam.

[23] J. Kannala and S. S. Brandt. «A generic camera model and calibration method for conventional, wide-angle, and fish-eye lenses». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.8 (Aug. 2006), pp. 1335–1340. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2006.153.

[24] S. Kato et al. «An Open Approach to Autonomous Vehicles». In: *IEEE Micro* 35.6 (Nov. 2015), pp. 60–68. ISSN: 0272-1732. DOI: 10.1109/MM.2015.133.

[25] Chang Sup Kim et al. «Improving odometry accuracy for car-like vehicles by using tire radii measurements». In: *30th Annual Conference of IEEE Industrial Electronics Society, 2004. IECON 2004*. Vol. 3. Nov. 2004, 2546–2551 Vol. 3. DOI: 10.1109/IECON.2004.1432203.

[26] *kinetic/Installation - ROS Wiki*. URL: http://wiki.ros.org/kinetic/Installation.

[27] Georg Klein and David Murray. «Parallel Tracking and Mapping for Small AR Workspaces». In: *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*. Nara, Japan, Nov. 2007.

[28] Y. LeCun et al. «Gradient-Based Learning Applied to Document Recognition». In: *Intelligent Signal Processing*. IEEE Press, 2001, pp. 306–351.

[29] Honglak Lee et al. «Unsupervised Learning of Hierarchical Representations with Convolutional Deep Belief Networks». In: *Commun. ACM* 54.10 (Oct. 2011), pp. 95–103. ISSN: 0001-0782. DOI: 10.1145/2001269.2001295. URL: http://doi.acm.org/10.1145/2001269.2001295.

[30] Steven M. Lavalle. «Rapidly-Exploring Random Trees: A New Tool for Path Planning». In: (May 1999).

[31] *mapviz - ROS wiki*. URL: http://wiki.ros.org/mapviz.

[32] J. Matas, C. Galambos, and J. Kittler. «Robust Detection of Lines Using the Progressive Probabilistic Hough Transform». In: *Computer Vision and Image Understanding* 78.1 (2000), pp. 119–137. ISSN: 1077-3142. DOI: https://doi.org/10.1006/cviu.1999.0831. URL: http://www.sciencedirect.com/science/article/pii/S1077314299908317.

[33] T. Moore and D. Stouch. «A Generalized Extended Kalman Filter Implementation for the Robot Operating System». In: *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer, July 2014.

[34] Marius Muja and David G Lowe. «Fast approximate nearest neighbors with automatic algorithm configuration.» In: ().

[35] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós. «ORB-SLAM: A Versatile and Accurate Monocular SLAM System». In: *IEEE Transactions on Robotics* 31.5 (Oct. 2015), pp. 1147–1163. ISSN: 1552-3098. DOI: 10.1109/TRO.2015.2463671.

[36] R. Mur-Artal and J. D. Tardós. «ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras». In: *IEEE Transactions on Robotics* 33.5 (Oct. 2017), pp. 1255–1262. ISSN: 1552-3098. DOI: 10.1109/TRO.2017.2705103.

[37] E. B. Olson. «Real-time correlative scan matching». In: *2009 IEEE International Conference on Robotics and Automation*. May 2009, pp. 4387–4393. DOI: 10.1109/ROBOT.2009.5152375.

[38] *Preparing Your Data for Use with robot_localization — robot_localization 2.5.2 documentation*. URL: http://docs.ros.org/melodic/api/robot_localization/html/preparing_sensor_data.html.

[39] Morgan Quigley et al. «ROS: an open-source Robot Operating System». In: *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*. Kobe, Japan, May 2009.

[40] Joseph Redmon and Ali Farhadi. «YOLO9000: Better, Faster, Stronger». In: *CoRR* abs/1612.08242 (2016). arXiv: 1612.08242. URL: http://arxiv.org/abs/1612.08242.

[41] Joseph Redmon et al. «You Only Look Once: Unified, Real-Time Object Detection». In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640.

[42] *REP 103 – Standard Units of Measure and Coordinate Conventions (ROS.org)*. URL: http://www.ros.org/reps/rep-0103.html.

[43] *REP 105 – Coordinate Frames for Mobile Platforms (ROS.org)*. URL: http://www.ros.org/reps/rep-0105.html.

[44] *rkt, a security-minded, standards-based container engine - CoreOS*. URL: https://www.docker.com/.

[45] *ROS 2.0 Design*. URL: http://design.ros2.org/.

[46] *ROS navigation stack*. URL: http://wiki.ros.org/navigation.

[47] *ROS navigation stack: AMCL*. URL: http://wiki.ros.org/amcl.

[48] E. Rublee et al. «ORB: An efficient alternative to SIFT or SURF». In: *2011 International Conference on Computer Vision*. Nov. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[49] Radu Bogdan Rusu and Steve Cousins. «3D is here: Point Cloud Library (PCL)». In: *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China, May 2011.

[50] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004. ISBN: 978-0-7356-1993-7.

[51] Shuran Song and Jianxiong Xiao. «Sliding Shapes for 3D Object Detection in Depth Images». In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, 2014, pp. 634–651. ISBN: 978-3-319-10599-4.

[52] Anthony Stentz. «Optimal and Efficient Path Planning for Partially Known Environments». In: *Intelligent Unmanned Ground Vehicles: Autonomous Navigation Research at Carnegie Mellon*. Ed. by Martial H. Hebert, Charles Thorpe, and Anthony Stentz. Boston, MA: Springer US, 1997, pp. 203–220. ISBN: 978-1-4615-6325-9. DOI: 10.1007/978-1-4615-6325-9_11. URL: https://doi.org/10.1007/978-1-4615-6325-9_11.

[53] Christian Szegedy et al. «Going Deeper with Convolutions». In: *CoRR* abs/1409.4842 (2014). arXiv: 1409.4842. URL: http://arxiv.org/abs/1409.4842.

[54] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. DOI: 10.4271/j3016_201609. URL: https://doi.org/10.4271/j3016_201609.

[55] Marvin Teichmann et al. «MultiNet: Real-time Joint Semantic Reasoning for Autonomous Driving». In: *CoRR* abs/1612.07695 (2016). arXiv: 1612.07695. URL: http://arxiv.org/abs/1612.07695.

[56] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN: 0262201623.

[57] H.M. Traquair. *An Introduction to Clinical Perimetry*. Mosby, 1946. URL: https://books.google.it/books?id=9ECsAAAAIAAJ.

[58] *Using etcd*. URL: https://coreos.com/etcd/.