



**Politecnico
di Torino**

Politecnico di Torino

Corso di Laurea Magistrale in Mechatronic Engineering

A.a. 2022/2023

Sessione di Laurea Marzo 2023

Analysis of algorithms for autonomous driving of a telescopic handler

Relatori:

Prof. Somà Aurelio

Ing. Mocera Francesco

Candidato:

Campana Davide

ABSTRACT

Autonomous vehicles have started revolutionizing the automotive industry about a decade ago, and are now starting to revolutionize the construction and agricultural fields.

Construction and agriculture presents different challenges with respect to the automotive counterpart, some of them given by the environment, others given by the nature of the vehicle at study.

The aim of this thesis is to analyze the sensors and the sequence of algorithms that are needed for AV applications, and apply them to the vehicle at study, the Merlo telescopic handler, in order to understand and highlight the challenges posed by this kind of vehicle in an agricultural environment.

The analysis has been carried out on four different fields: path planning, Lidar data clustering, obstacle representation, navigation.

Particular attention has been given to the clustering algorithms, in which both well established solutions and a very recent algorithm have been analyzed and implemented.

The clustering and navigation algorithms have been tested using two different datasets, one belonging to the construction field and the other to the urban environment: the lack of freely available agricultural Lidar datasets is still a difficult point in the development of AV for agriculture.

The challenges posed by the arm of the telescopic handler have been analyzed and visualized using the same datasets as above.

Summary

INTRODUCTION	5
CHAPTER 1 – OFFLINE PATH PLANNING	14
1.1 Introduction to path planning	14
1.2 Working scenario and motivations	16
1.3 Path planning algorithms	18
1.3.1 Grid-based search algorithms	18
1.3.2 Dijkstra algorithm	21
1.3.3 Sample-based algorithms	24
1.3.4 RRT algorithm	25
1.3.5 RRT* algorithm	26
CHAPTER 2 – SENSORS AND METHODOLOGIES FOR AUTONOMOUS NAVIGATION	31
2.1 Obstacle perception	31
2.1.1 Ultrasound sensors	32
2.1.2 Radar	33
2.1.3 Lidar	35
2.1.4 Cameras for obstacle detection	39
2.1.5 Cameras for object classification and sensor fusion	40
2.2 Localization	43
2.2.1 GPS RTK	43
2.2.2 SLAM	45
CHAPTER 3 – LIDAR DATA CLUSTERING	49
3.1 Lidar sensor	49
3.1.1 Choice of sensor	49
3.1.2 Motivations for clustering	50
3.2 Classic clustering algorithms	51
3.2.1 K-means	51
3.2.2 DBSCAN	55
3.3 FLIC (FAST LIDAR IMAGE CLUSTERING)	59
3.3.1 GROUND REMOVAL	60
3.3.2 BINARY CONNECTIVITY MATRICES	61
3.3.3 RECOGNIZING CLUSTERS IN BINARY CONNECTION IMAGE	64
3.3.4 FLIC RESULTS AND COMMENTS	66
3.3.5 MAP CONNECTIONS AND MINIMUM THRESHOLD	67
CHAPTER 4 – OBSTACLE REPRESENTATION	71

4.1 FROM CLUSTERS TO CUBOIDAL MODELS.....	71
4.2 OCCUPANCY MAP REPRESENTATION	74
4.3 FROM CUBOIDAL MODELS TO OBJECT LIST REPRESENTATION.....	76
4.4.1 Structure of dynamicCapsuleList	77
4.4 COLLISION CHECKING	79
4.4 LIMITATIONS.....	80
4.5 TELESCOPIC HANDLER: THE TELESCOPIC ARM ISSUE	80
4.5.1 Simple model of the arm	80
4.4.2 Obstruction handling.....	82
CHAPTER 5 - NAVIGATION	86
5.1 MODEL PREDICTIVE CONTROL	86
5.2 REFERENCE PATH.....	90
5.2.1 Frenet Coordinate system	91
5.2.2 Frenet trajectory generation	93
5.2.2 Cinematic constraints	96
5.3 NAVIGATION ALGORITHM ANALYSIS AND SIMILARITIES WITH MPC.....	98
5.3.1 Initialization of capsule list	98
5.3.2 Alternative trajectories generation	98
5.3.3 Terminal states cost function	99
5.3.4 Lidar data processing.....	100
5.3.4 Trajectories collision checking.....	100
5.4 TRACKING	102
5.4.1 Data-association problem - Hungarian algorithm	104
5.4.2 Prediction problem - Kalman Filter	107
6. CONCLUSIONS	113
7. APPENDIX.....	117
7.1 FLIC IMPLEMENTATION	117
7.1.1 main	117
7.1.2 connection matrix standard	118
7.1.3 connection matrix with one Map Connection.....	119
7.2 TELESCOPIC HANDLER OCCLUSION SIMULATION	121
7.3 LOCAL NAVIGATION.....	123

INTRODUCTION

The telescopic handler

The telescopic handler is a machine equipped with four wheels of the same size, and in this it differs from the tractor. It's a machine that uses a hydraulic pump in order to push pressurized oil in several hydraulic cylinders, in order to push and extend its distinctive component, which is the telescopic arm. The telescopic handler has commands in order to maneuver the arm and its attached tools in several configurations, making it able to lift, bring down, extend and retract the load that is being maneuvered. The versatility of the machine makes it capable of working in several different environments with different mission goals, for many of which, though, the base constituted by the wheels alone is insufficient in order to provide the necessary stability. To account for this, four extendable legs are also powered by the hydraulic pump: these legs have the task to stabilize the telescopic handler in a static configuration (this means that the vehicle won't be able to move while the legs are extended) and are composed of the leg and of the hydraulic cylinder that makes them extend.



Figure 1-Merlo Roto telescopic handler

The telescopic handler is not to be confused with the forklift: the latter is a machine, usually smaller in size with respect to the handler, that is only able to lift loads in a vertical way, and whose lifting device, which are the forks, have limited mobility with respect to the telescopic arm. In fact, the forks of the forklift don't move longitudinally, and this makes so that they never leave the base of the forklift.

The telescopic handler is constituted of the wheels, the main body, the cabin and the arm. Usually the arm is placed, in its resting position, on the right of the cabin, allowing good visibility for the operator when handling loads. Inside the cabin there are commands in order to drive the machine and to move the arm, and the design is studied in order to be able to do both at the same time; the arm in the newer models is controlled by a joystick, and the signal transmission to the control unit is electronical. This allows for the latest models to be able to control remotely, within certain limits, the arm maneuvers. For example, when the basket lifter is attached to the arm, and operators are on it, they can control the machine without having another person in the cabin.

The arm of the telescopic handler is attached to the right of the cabin with a big hinge: this can be modeled and viewed as a rotational joint, with its working angle included between 0° (resting position) and 80° (largely varying between models and manufacturers) [1].

The arm of the telescopic handler is composed in an onion-like structure: multiple steel parallelepipeds, the smaller one closer to the tip, slide one into the other during the extension and retraction phases. This can be modeled by a series of multiple prismatic junctions. Hydraulic cylinders placed inside the arm are the actors of this extension movement. The smaller of the parallelepipeds ends with an almost perpendicular component, called nose (highlighted in red in figure 1). The nose constitutes the link between the extendable part of the arm and the end effector of the arm, which is called the “zattera” and badly translates into english. The end effector is attached to the nose with a revolute joint, that allows it to change its angular orientation within some degrees. The “zattera” then constitutes the attach point for lots of different tools, that increase the versatility and use case scenarios of the handler. Most popular tools are forks, multifunction claws, earthmoving bucket, pallet forks, straw bale picks, basket lifter. The multitude of the tools that are attached to the telescopic handler makes it clear that the mission it can be used into belong to the most different environments: agricultural, construction, quarries, civil works.

The telescopic handlers' names contain in them the maximum liftable capability and arm extension; for example the Merlo Turbo Farmer 44.7 is able to lift 4400 kilograms at a height of 7 meters, or the Merlo Roto 50.30 is able to lift 5000 kilograms at a height of 30 meters. As it can be seen, the range of products varies a lot, depending on the use cases.

Some models of telescopic handler are capable of rotating their cabin and arm at a full 360° ; in the Merlo franchise, these go by the name Roto, and are composed of the same components of the “normal” telescopic handler, plus a big central pin that allows for the upper part of the machine to freely rotate. This allows for an additional degree of freedom and even more use cases.

One of the peculiar distinctive traits of the Merlo telescopic handler in particular is the feature of having four steering wheels, and the hydrostatic transmission. The latter one is the feature that allows the former to exist. Hydrostatic transmission has the advantage of transmitting high powers at virtually any distances, using flexible tubing; this is what allows it to create out of axis transmissions and without any particular alignment constraints [2]. This also implies the advantage of having a good peak load absorption, making the telescopic handler a robust vehicle. The power over weight ratio is very high, and this allows for compact actuators situated far away from the big engine-pump body. The sacrificed factor in the hydrostatic transmission is the efficiency, as it is lower than the dry counterpart. The hydrostatic transmission also provides a challenge in deriving the transfer function from the control unit to the wheel, and this aspect is of importance in an autonomous driving context. There are several ways to perform system identification of this kind of transmission, one of the newest being the Set Membership approach (Abuabiah, Regruto, Set-membership identification of a dry-clutch transmission model, 2017); this work was not carried out in this thesis but it will be of necessity in order to the creation of a future prototype.

Autonomous driving in the agricultural field

Autonomous driving solutions in the agricultural field has seen a growth in research in recent years, some examples of recently published papers on the topic are [3], [4]. The usage of autonomous vehicles in this particular field present a number of advantages that make it so that lots of companies are investing in reseach and development so to create autonomous versions of their robots [5], [6]. The usecases are widely distributed across the whole spectrum of agricultural activities.

Tractors maufacturers, for example John Deer, focused their energy towards the development of autonomous driving tractors, based on their classical man-driven vehicles (figure 2).



Figure 2- source: John Deere

These kinds of tractor autonomous solutions are aimed at big open spaces, open fields, and long and continuous working regimens. The activities carried out by these tractors are mostly ploughing, fertilizing and harrowing, so mostly soil working activities carried out over long distances. Intensive working of big quantities of soil require driving tractors for lots of hours at a time, resulting in stressful human conditions and even medical conditions to the back [7], therefore putting at risk the health conditions of the workers, potentially leading to chonical conditions. The introduction of self-driven tractors could be a turning point in the health related hazards in the “big fields” sector. Another point to keep in mind is the increased productivity of the soil, since the self driven tractors don’t need breaks and can work at nighttime in some cases.

Another state of the art development field for agricultural robots is automated weeding. Weeds constitute a problem in orchard scenarios as they are an infesting factor, causing in the less serious cases trouble in picking up the fruits produced by the plants (that is, nenttheless, a revenue limiting factor), and in the worse scenarios they can be the source of pathogens for the tree. Industries such as Aigro has worked towards creating solutions for automated weeding, since weeding is an activity that needs to be done several times in the arch of the year. The robot Aigro UP removes weeds using the tine harrow situated behind the traction-wheels; the machine works at slow speeds, as is common in these scenarios, but the speed factor is not quite important in

these kind of applications, where the productivity factor is given by the overall orchard situation over the year, and not by the speed at which the weeds are removed in a single session. This solution works on battery packs, and this is a common occurrence in small agricultural robots, while is not as common for the big tractors; the electrical solution, in its actual state of the art state allows the working of a 5 to 10 hectare field at a time.



Figure 3- AigroUP - source: Aigro

Another common field of interest in the context of agriculture robotics is vineyards care. Vineyards care is a sector more aimed towards high quality products rather than mass production, and its challenges are different from the ones that have to be accomplished by a self-driving tractor. The quality of the final product makes it so that a single solutions for all tasks does not exists, and several activities are needed in order to arrive at the grape picking time. Actions to be performed include thinning out excessive foliage, spraying pesticides, checking for signs of dehydration or malnutrition, fertilizing etc. Some of these tasks can be performed by normal rover robots, other need robots that can encapsule the whole file. An example of thinning and foliage control is given by the Vitibot Bakus bot, which has a particular bridge shaped form that allows it to navigate around the whole file, shown in figure 4. The vineyard control robots come in different sizes depending on the type of cultivation, the width of the file and the steepness of the terrain. This kind of robot is an example of extremely specialized robot, which puts it on the opposite side of the spectrum of the general purpose tractors.



Figure 4 - Vitibot's Baku - source: Vitibot

Another agricultural related job that is being taken on by robots in certain parts of the world is weed control over railways. This activity is similar to orchard weeding operations, but it differs from it mainly for the size of the working area. In the case of the orchard the working area is a closely delimited space, often marked with some kind of fence or barrier, usually in a rectangular shape. In the case of railway weed control, the working area is the narrow line that coasts the whole perimeter of the rails, and therefore is more similar to a line, and can extend for hundreds of kilometres. The collaboration between Vitirover and MDP [8], has created in France a solar powered rover capable of cutting weeds and navigating through challenging environments, in order to keep the grass level around railways under control. The solar panels that the rover is equipped with make it so that it can work continuously for several days or even weeks, without the need for human intervention. This kind of robot presents other differences with respect to the orchard counterpart than autonomy, mainly related to the navigation setting. As it can be seen in figure 5, the Vitirover solution navigates having as a reference guide a red line that runs along the railway. This is a great simplification that allows for navigation algorithms to be simple line following algorithms, something that general purpose weeding don't have, as they need to be able to work in a less-structured environment.



Figure 5 - Vitirover weeding along a railway - source: Vitirover

Working mission and considered scenarios

The mission that is tackled in this thesis works groups under the open field activities. In the previous section it was presented that several open field activities are carried out by tractors, and related to soil working. Although soil working can be considered one of the activities feasible by the telescopic handler, it is not one of those at its core activities. The peculiarity of the telescopic handler is, of course, the ability of performing movimentation of loads with good dexterity. An open field activity that requires good dexterity is hay ball movimentation. Hay balls movimentation is currently carried out manually, but it has the potential to become an automated task; it is a repetitive task and hayballs are relatively simple objects to handle.

The draft of the ISO 18497 [9] proposes some classifications and scenarios in which autonomous vehicles for agriculture will need to fall into. In particular it proposes, in the nomenclature section:

perception system: system that gathers and processes information about the environment in which the machine is operating

warning zone: area where if an *obstacle* is within and no action is taken, then the *obstacle* might enter the *hazard zone*

hazard zone: area which is a subset of the *warning zone* and where if an *obstacle* is within that area, then the potential for injury can exist

autonomous mode: mode of machine operation in which a machine performs *functions* related to its defined tasks without *operator interaction*

autonomous operating zone: designated area in which machines operate in *autonomous mode*

active state: machine system state in which *partially automated, semi-autonomous or autonomous functions* are provided

safe state: operating state of a system with acceptable level of risk for operator or bystander even when the control system fails or partly fails

use case: specific situation in which a product is used; includes machine type(s), operating conditions, functional ranges, system boundaries and operational design domain

Using the proposed nomenclature, we can define the **use case** of the autonomous vehicle studied in this work as open field autonomous work. The functional ranges refer to the extent to which the machine can still performed its assigned task. Here we will consider an almost completely controlled environment, in the sense that some sort of map of the environment is given to the machine, and localization is available; in this sense the functional ranges of the machine will be limited to obstacle avoidance of non-alive obstacles, and in the case of presence of alive obstacles, such as humans or animals, the machine will exit the active state (state in which the machine performs the work it is assigned) and enter the safe state, see above for definition. Since the presence of humans mandates the entering of the safe state, the scenario in which the machine operates is not a cooperative scenario but a fully autonomous one. In this sense, the definition of hazard zone needs to be stretched to injuries provoked to the machine or its surroundings (comprising obstacles), because injuries to people are not considered. Therefore we can speak of hazard zone in the general sense of zone that leads to some form of damage to objects. The

warning zone meaning remains unchanged. Both the hazard zone and the warning zone will be tuned and adjusted by enhancing or reducing the safety boundaries around the machine, in the context of the navigation algorithm and obstacle representation presented in chapters 4 and 5.

High level scheme of the considered solution

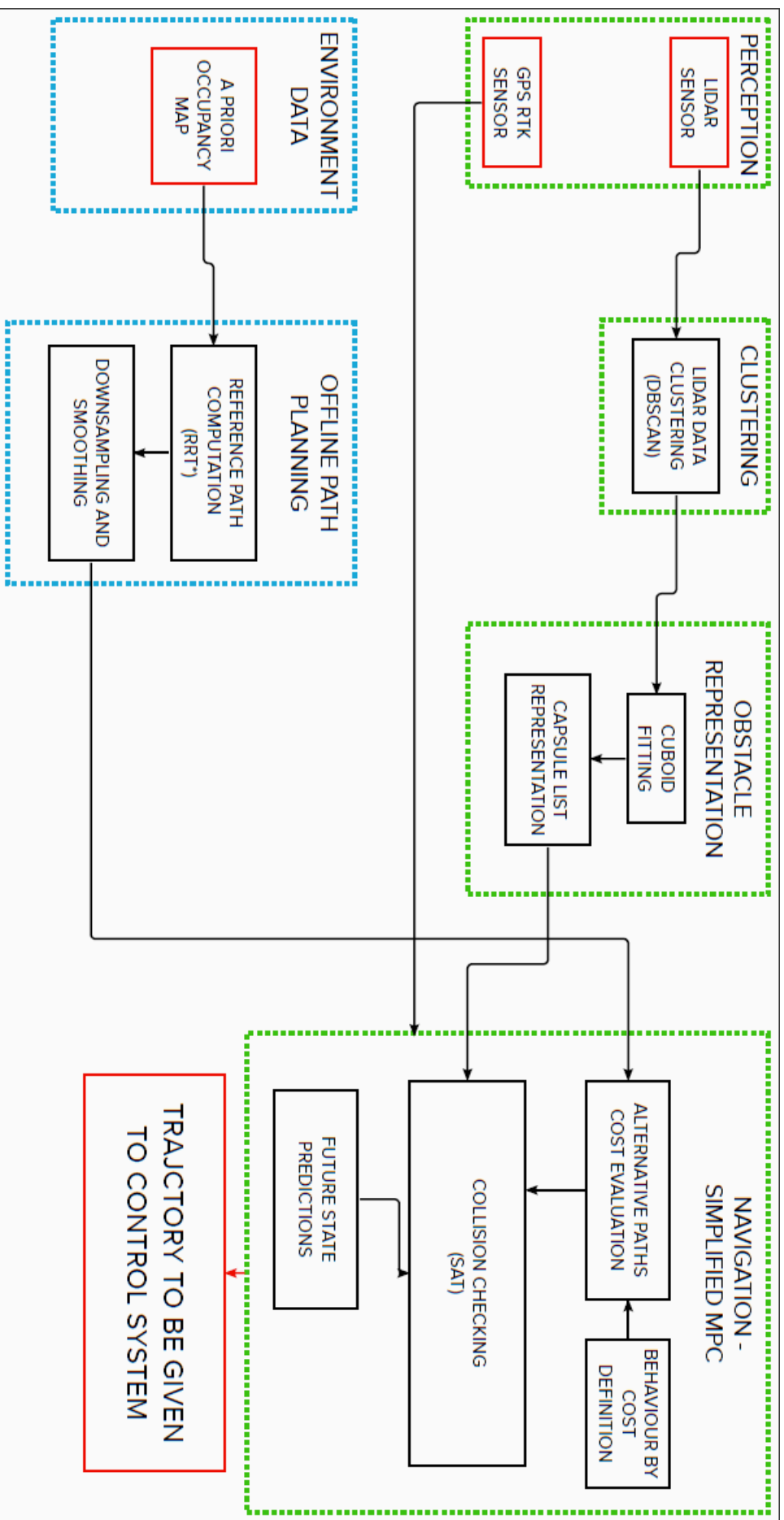
Autonomous driving is a name that labels an incredible amount of research that crosses lots of field in computer science, mathematics, automation engineering, sensor fusion, sensor technology, although most of it is done in the automotive domain. The focus of this thesis work is to perform simulations of some of the blocks that compose the chain of algorithms that build an AD system. In certain blocks, state-of-the art algorithms will be compared to well-established solutions; the whole process will be analyzed, where it makes sense to do so, highlighting the additional challenges posed by the fact that the physical vehicle at study is a telescopic handler and not a transportation vehicle.

In the high level scheme presented in the next page, the covered blocks are presented, divided into thematic groups, with colors having the following meaning:

- The blocks containing data that belongs to the input and output elements of the algorithm are highlighted in red
- The blocks containing actions that are performed online (from the moment the machine enters active state to where it returns to safe state) are highlighted in green
- The blocks containing actions that are performed offline (before the machine enters active state; these blocks may be done on board or remotely) are highlighted in blue

The blocks not implemented in this work, but that are cited more or less extensively are, at least:

- A machine learning or deep learning algorithm that, given the input provided by the cameras, determines the presence of people or animals in the warning zone and consequently toggles the active state / safe state switch. Examples of popular algorithms in literature are [10]
- A finite state machine that, based on the environment, the perceived obstacles, the weather ecc... tunes the parameters of the different algorithms such as clustering, navigation... in order to obtain the best performance in each working environment
- A finite state machine that determines current mission, waypoints and initiates safe state mode if the conditions of the environment/weather or the data coming from the sensors aren't acceptable for navigation. This also accounts for the presence of people / animals detected by the machine learning / deep learning algorithms.



INTRODUCTION - BIBLIOGRAPHY

- [1] : https://it.frwiki.wiki/wiki/Chariot_t%C3%A9lescope
- [2] : [https://www.ilprogettista industriale.it/trasmissioni-idrostatiche](https://www.ilprogettistaindustriale.it/trasmissioni-idrostatiche)
- [3] : Joong-hee Han et. al., Performance Evaluation of Autonomous Driving Control Algorithm for a Crawler-Type Agricultural Vehicle Based on Low-Cost Multi-Sensor Fusion Positioning, 2020
- [4] : Abhishesh, P., et al. "Multipurpose agricultural robot platform: Conceptual design of control system software for autonomous driving and agricultural operations using programmable logic controller." *International Journal of Mechanical and Mechatronics Engineering* 11.3 (2017): 507-511
- [5] : John Deere, John Deere Reveals Fully Autonomous Tractor at CES 2022
- [6] : www.aigro.nl, Aigro ROBOT
- [7] : Massimo Bovenzi, Alberto Betta, Low-back disorders in agricultural tractor drivers exposed to whole-body vibration and postural stress, 1994
- [8] : <https://www.vitirover.fr/>, 2022
- [9] : DIS_ISO_18487_Machine_Design_Principle
- [10] : Joseph Redmon, Ali Farhadi, *YOLO9000: Better, Faster, Stronger*, 2016

CHAPTER 1 – OFFLINE PATH PLANNING

1.1 Introduction to path planning

Path planning is a general problem that arises in various areas of automation engineering; for example how to move the end effector of a robot manipulator is a famous example of a path planning problem: The robot must be able to move through an environment without entering contact with obstacles and performing the assigned task.

In our case, the problem of path planning consists of finding an obstacle free path that the ego vehicle can follow in order to arrive to the objective area. The components of the path planning problem are:

- the ego vehicle, which in our case is the telescopic handler
- the starting point
- The objective point; usually, instead of an objective point an objective area is considered instead: this helps with the feasibility of the problem and it's similar to a real use case scenario, where it is not needed to achieve perfect positioning in one point
- The obstacles: the obstacles are considered to be static; in the mission scenario considered examples of obstacles could be a building, a fence, a wall. Note that in this part of the algorithm moving obstacles such as people or vehicles are not accounted for.

In the consider scenarios a map of the whole working area is considered to be given; this assumption implies that the telescopic handler will not be completely autonomous in the sense that some sort of human guidance will be needed; for example if the vehicle is supposed to work in a factory a map of the factory will have to be processed by a human and transformed in a representation of the factory that clearly shows the various kind of obstacles that are present.

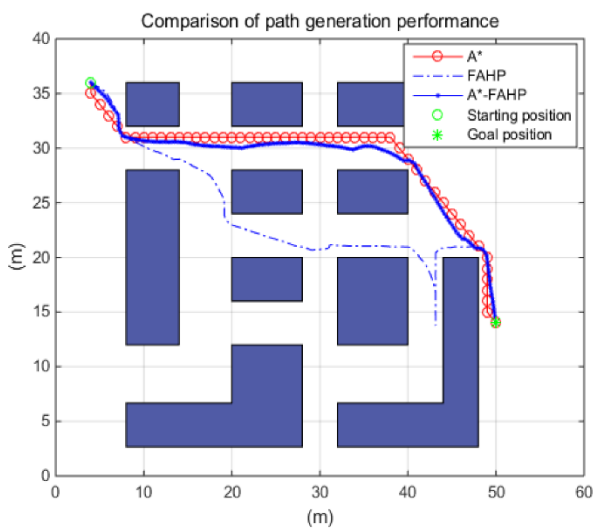
This aspect can be seen as a limiting factor of the capabilities of the machine come but it also the advantage of being able to retain the machine from going in a particular area where instead it would go if a map of the area were not to be available. In this sense this is a conservative approach to autonomous driving.

Apart from the basic elements written above, path planning comes in different flavors, because there are different missions that can be achieved by an autonomous vehicle; for some of these missions the exact shape of the path, provided that it is an obstacle free one, is not important; examples of this category are the missions in which the vehicle has to transport some amount of material from one place to another like in a construction site: it is not important the path taken to move the material as long as it doesn't impacts with anything.

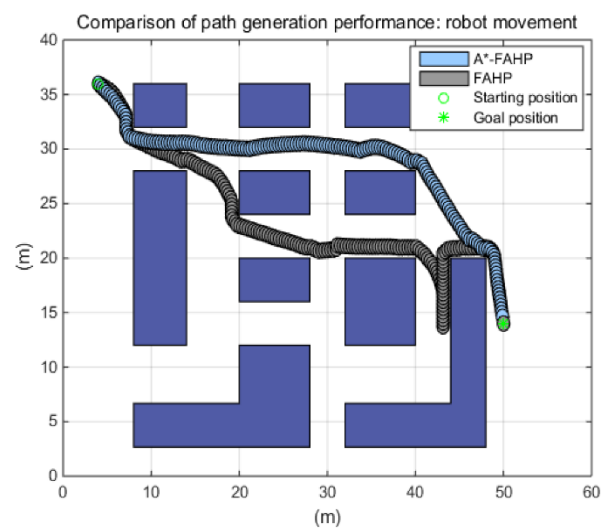
But there can be cases in which the in particular path taken is important: an example of this are the various soil preparation activities such as plowing watering or even planting. In this case is the

past must satisfy specific requirements such as shape working area coverage etc; this kind of requirements are often difficult to incorporate in a path planning algorithm solver and are another example of an activity that would be preferentially done offline by a human worker. On the other hand if the requirements are not particularly stringent and the machine is only supposed to follow some specific points in the working area it is possible to introduce the concept of waypoints: a waypoint is the fine as an intermediate objective point from which the path necessarily needs to pass by; again as in the case of the objective point a certain tolerance can be accounted for when defining the waypoints.

In Figure 1 some example of path planning problem are given (this is an example concerning indoor mobile robots):



(a) Longitudinal/lateral movement of the robot.



(b) Trajectory of the robot.

Figure 6- path planning exmples

In Figure 1.b an important aspect of the path planning problem emerges: it is not a problem with just one solution; in fact this is a problem that, if feasible, has an infinite number of solutions; but some solutions will be better than others and it will be important to define metrics to evaluate this concept of “goodness” of a solution.

1.2 Working scenario and motivations

In order to understand the motivations behind the choice of the path planning algorithm, some considerations about the working scenario have to be done.

The development of the thesis was done around a target mission, that is to move a simple load from one point to the other. The exact means on how to handle the load are irrelevant; the machine is supposed to start with the load already loaded in a previous operation. The chosen load has then been chosen to be a hayball, which can be taken by the machine via various tools, and the chosen working scenario is that of the outsides of a farm; that is, some sort of field where agricultural work needs to be done.

The first big hypothesis underlying the farm scenario is the “visible sky hypothesis”: the chosen environment is an open one, as the sky is assumed to be visible at all times, and this means that verticality of the machine or obstacles that do not start from the ground up will not be accounted for; this is a limiting hypothesis for a number of reasons:

- Stocking areas of different kinds of material often include some vertical and nonconvex structure that extends some meters above the ground
- There may be present temporary structures such as tensile structures that leave the ground free but constitute an obstacle some meters above
- In real farms, the material can often be stocked inside, or has to be moved inside the farm.
- Telescopic handlers often work staying outside of a closed working area and then moving the material inside it using the arm, this is impossible to do without considering vertical encumbrance

As it is noticeable, the “visible sky hypothesis” presents various limitations that affect the performance of the handler and also its potential. One of the strengths of the telescopic handler is its versatility, and the ability to work in an unstructured environment; assuming that all loads and obstacles are on the ground can be seen as quite unrealistic in most cases.

The hypothesis was assumed in order to simplify the first study and algorithms, but it is worth mentioning that use cases where it holds well exist, and can be considered important enough to justify it by themselves. In the following, some remarkable examples are given:

- Moving big quantities of excavated material in a wide environment such as a cave
- “Human machine cooperation”: since the sensors mounted on the machine can work at nighttime, a possible case is the one where a human operator gathers the goods to be transported in an open area near the farm (covering small distances) and leaves the machine to cover big distances over and over during nighttime.
- Ploughing: this is a classical open field activity in which autonomous driving has been shown to be very efficient at, and seldom requires entering in closed areas

After the first part of the development of the path planning algorithm, the problems deriving from the “visible sky hypothesis” will be partially solved, see chapters 4 and 5.

The second big hypothesis underlying the farm scenario is the “static hypothesis”: this refers to the staticity of big structures in a farm or field. While it is true that some changes can occur over time, it is reasonable to think that from the moment the path planning algorithm develops a possible path for the vehicle to the moment the vehicle has started moving, no big changes should have happened in the working area topography. This hypothesis is easy to find in agricultural scenarios, and the field scenario is one of them; instead, common scenarios where this hypothesis wouldn't work too well are construction sites, where big changes in the topology of the place can happen rather quickly (i.e.: a tower crane can block a previously available way by putting some heavy pallet).

Assuming staticity of the working environment allows us to compute a possible path only once (from here the name offline path planning), but this shouldn't be confused with the absence of moving actors in the scene: moving people, vehicles, loads ecc... are all allowed to be in the working area of the machine as long as their dimensions do not alter topologically the map of the place. This means, we are assuming that possible obstacles that were not considered during the offline path planning process can be avoided, and thus treated as obstacles rather than map features.

For example, if a cow happens to walk in the designated path of the handler, it is assumed that there is enough space for it to avoid it and return on the designated path, after some time.

In figure 2, the green pentagon-shaped obstacle causes a topological change of the environment, that renders the previously computed path (in bold blue) unfeasible. A new path is needed, represented by the dot stretch line.

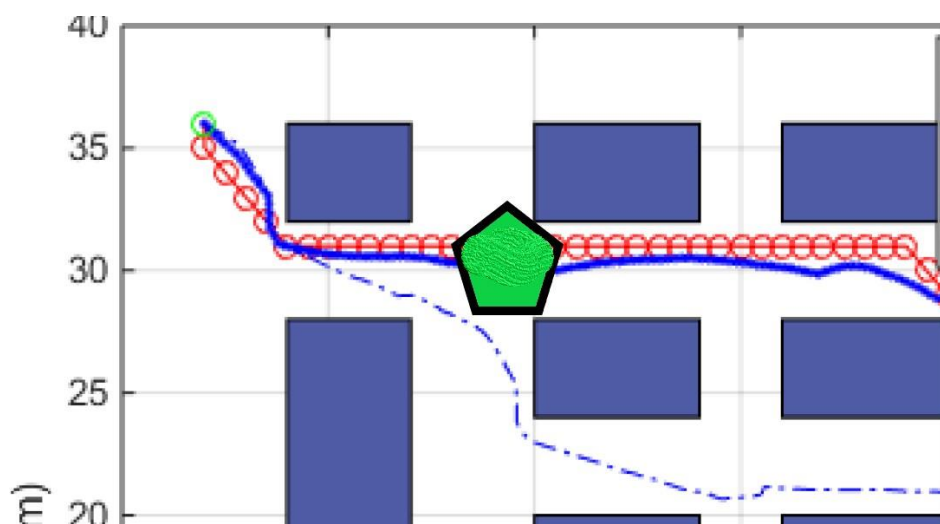


Figure 7- Topological change in the environment

With the basic hypothesis in mind, we'll now take a look at the various kinds of algorithms that are used to solve this problem.

1.3 Path planning algorithms

The first notion to introduce in order to talk about path planning algorithms is the notion of free configuration space. Strictly speaking, the free configuration space can be defined as the set of all feasible configurations, and this accounts for all constraints of the vehicle (or robot), usually kinematics and size ones [1].

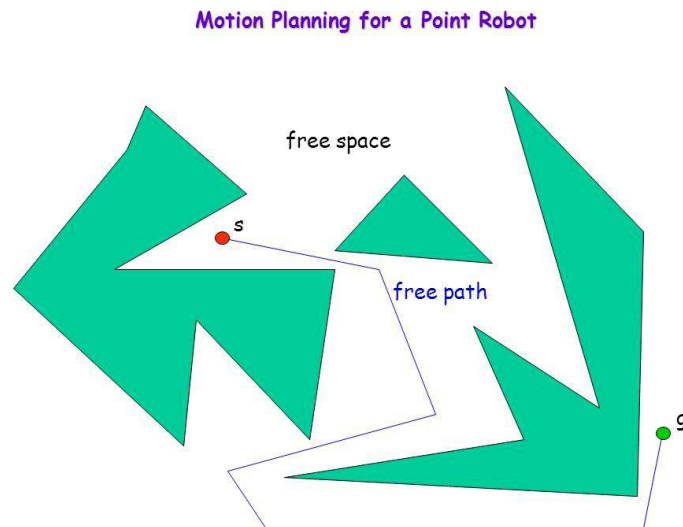


Figure 8- free configuration space

This kind of algorithm is said to be a global algorithm, as it searches for a path that connects the initial point to the final; a local algorithm, instead, solves a problem only in the immediate vicinity of the current position of the robot. This latter category is also often referred to as path following algorithms, more on chapter 5.

In this chapter we focus on global algorithms, and they are usually divided into three classes:

- grid-based search algorithms
- visibility graph algorithms
- sample-based algorithms

1.3.1 Grid-based search algorithms

This kind of algorithms work by discretizing the free space (called F from now on) into elementary cells. The elementary cells will contain or not contain an obstacle, and therefore can be seen as free cells or occupied cells. After the discretization, the problem is translated into a connectivity graph problem, where the goal is to find a path that connects starting point to end point passing only in free cells.

So the grid-based search algorithms divide the path finding problem in two: a discretization problem (also known as cell decomposition problem) and a connectivity problem.

The cell size is important: they can't be too big or with a complicated shape, and they can't have holes; cell decompositions must also satisfy these constraints: [2].

- finding a path that connects two points inside of a cell must be an easy task: convex cells are to be preferred as each point inside them can be simply connected by a line (from the definition of convex set)
- adjacent information between cells must be easily retainable
- given two points of the space, finding which cells contain them must be an easy task.

If the composition satisfies all properties allocated above, the problem reduces to the resolution of a graph. Most popular methods for this objective are Dijkstra and A* algorithms, more on these later.

One of the possible ways to decompose a space is the vertical decomposition, here described: let P be the set of vertices that enclose the space of configuration with obstacles C_{obs} . for each point belonging to P Extend when possible, vertical segments towards up and down directions inside of C_{free} until we reach again C_{obs} . This divides the space in lines and triangles, as shown in figure 5, And there are four possible cases as shown in figure 4.

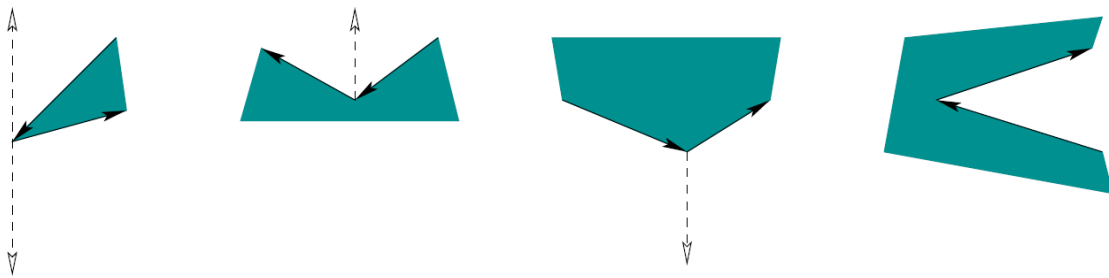


Figure 9- 1) bidirectional extension 2) towards up 3) towards down 4) impossible extension

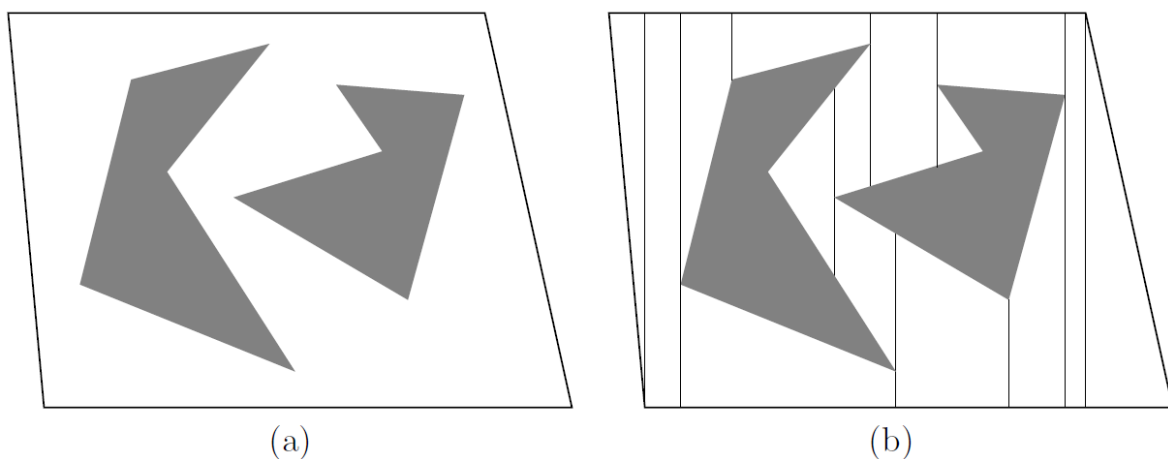


Figure 10- vertical decomposition of a generic space

Once a composition has been created, it is necessary to define the road map. For each cell C_i , let q_i be its designated sampling point: there are various ways to define a sampling point, usually the centroid of the cell is preferred.

Let $G(V,E)$ be the topologic graph defined in the following way: for each cell C_i , define a vertex q_i such that each cell has one independently from its size. sampling point inside of our 2D cell, start a line that connects it to a sampling point in the 1D cell that are on its boundary. Edges created this way represent paths that unify between them all the adjacent cells. Graph created in this way satisfy accessibility conditioned from the moment that every point can be reached via line, thanks to the convexity of cells; It also satisfies connectivity conditions from the moment that G has been created by the composition that intrinsically maintains connectivity of C_{free} . An example of road map can be seen in Figure 6.

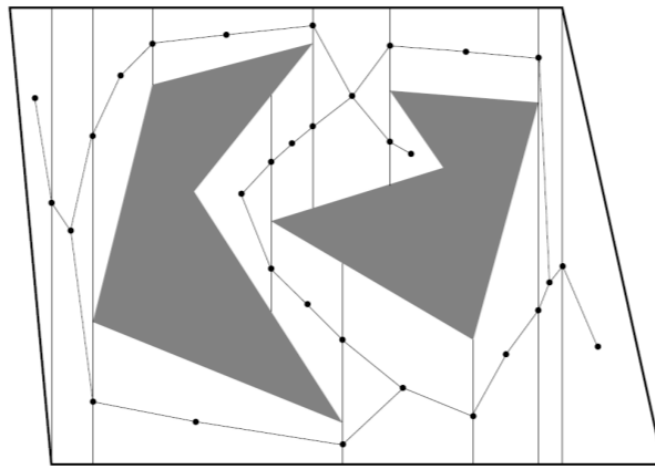


Figure 11 - example of roadmap

Once the road map has been obtained, the last step to perform is to solve the graph; in order to do this, Most popular algorithms are Dijkstra and A*. In Figure 7 an example of solved road map can be seen:

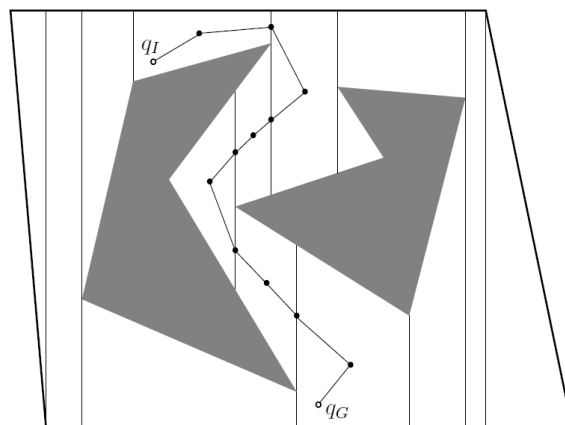


Figure 12- example of solved roadmap

Grid based search algorithms are quite simple in their working, But their main drawback is the computational cost. In fact, supposing that n vertex are present, the complexity cost is $O(n^2)$; In the case of a particularly complex model, computation time will be extended too much.

1.3.2 Dijkstra algorithm

This is a popular algorithm used to solve graphs, and it can be seen as the basis on which A^* is built. It is not an algorithm that can by itself find a path, it is the algorithm used to solve the graph problem obtained for example in the first part of the grid based algorithms, that is the class decomposition.

The Dijkstra algorithm visits the nodes of the graph in a similar way as a search in width or in depth. At each instant the set N of nodes of the graph is divided in three parts: the set of already visited nodes V , the set of frontier nodes F , and unknown nodes that are still to be examined [3].

A value d_z is given to each node z , initially put equal to 1.

At each step the algorithm takes from the set F any node z with d_z minimum, It moves it from F to V , and moves all the successors of z unknown in F . For each successor w of z values d_w are updated. Update is done according to:

$$d_w \leftarrow \min\{d_w, d_z + pa\}$$

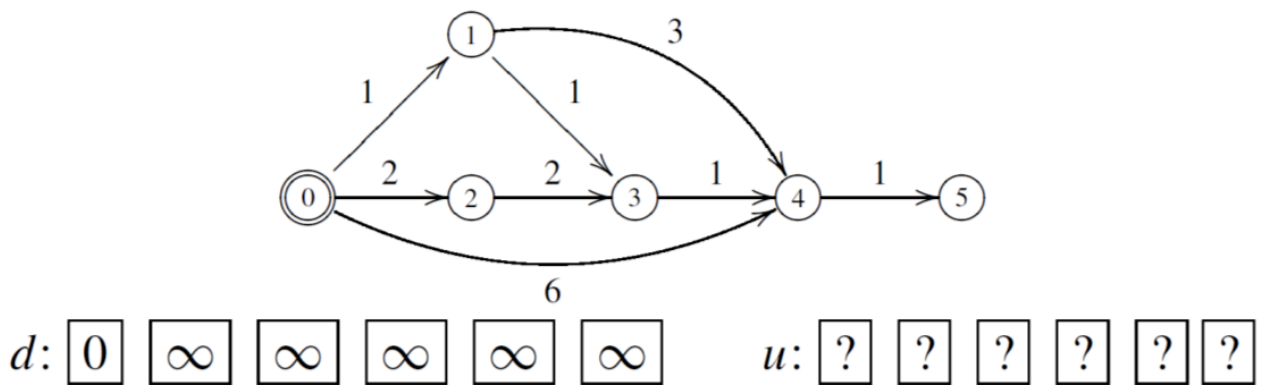
where a is the arch that connects z to w ; if the value of d_w has been modified, then u_w is put equal to z .

The basic idea of this algorithm is the following: if we know that arriving to z costs d_z , arriving to w can't cost more than arriving to z and moving along and arch until w .

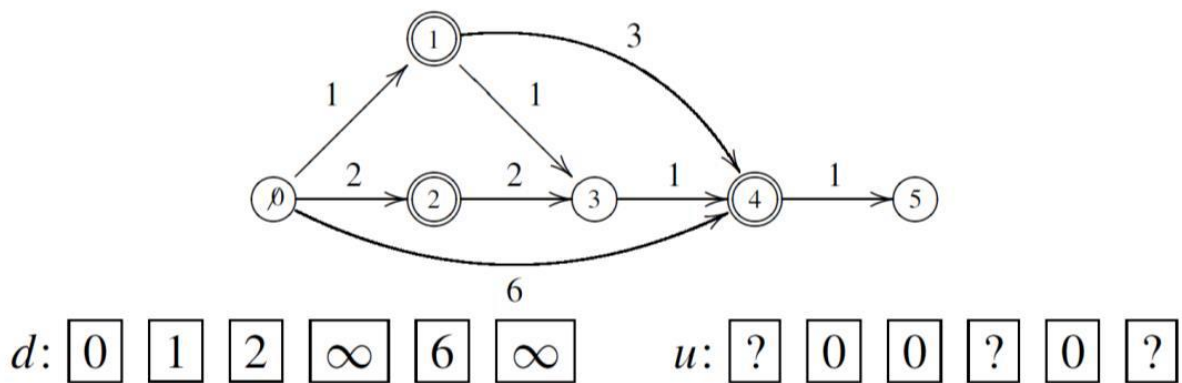
The algorithm starts with $V = \emptyset$, $F = \{x\}$, $d_x = 0$ and continues until y is not visited or until $F = \emptyset$: In this case, y is not reachable from x along an oriented arch.

At the end of the algorithm d_z contains for each node z the weight of a minimum path from x to z ; furthermore, vector u allows to reconstruct the tree of minimum paths with origin in x .

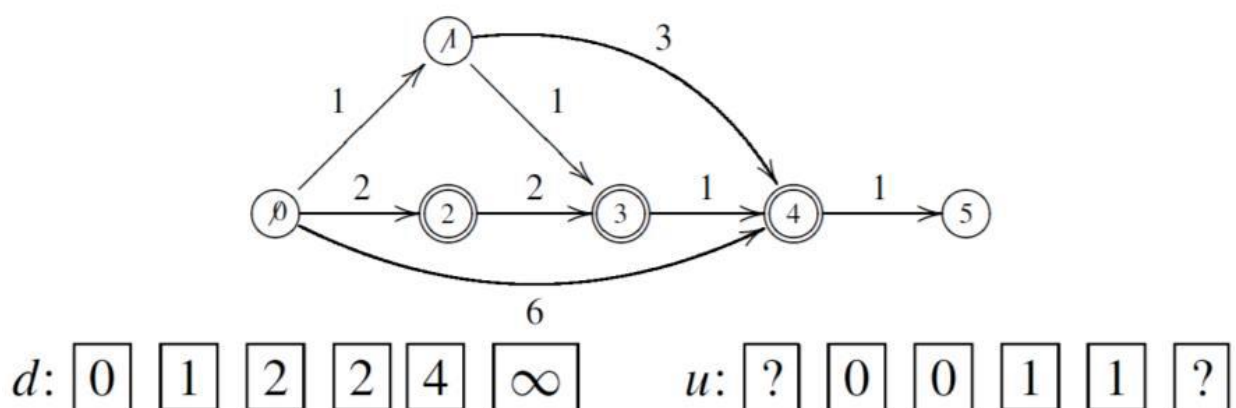
In order to better understand the algorithm, a simple example is reported: the objective is to reach node 5 starting from node 0:



First step: node 0 is visited, while 1,2,4, are put in frontier F (double circled in figure 9):



Now we have to take the frontier node with minimum distance from 0, and it can be seen how Dijkstra algorithm solves problem in an amplitude way: node 3 and 4 are quite distant from 0 in terms of path weight, and thus the closest node is 1. 1 is visited, and this updates d and u:



Once again, instead of choosing nodes initially added to the frontier, we proceed with the node that presents minimum d , that is node 2. The effect is simply to move node 2 in the set of visited nodes, as the for the only successor of node 2, that is node 3, we already know a better path (with smaller weight) than the one that would pass in 2.

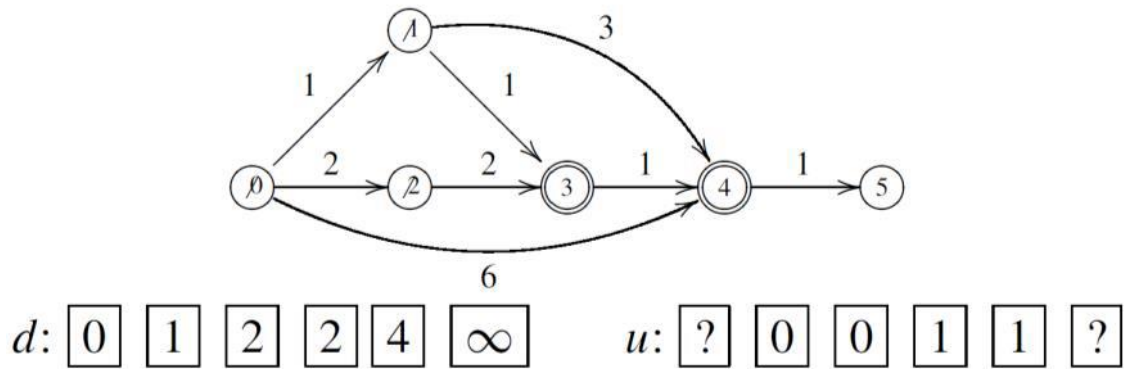


Figure 16 - third step, visiting 2

Now the minimum d node is node 3, and visiting it updates d and u :

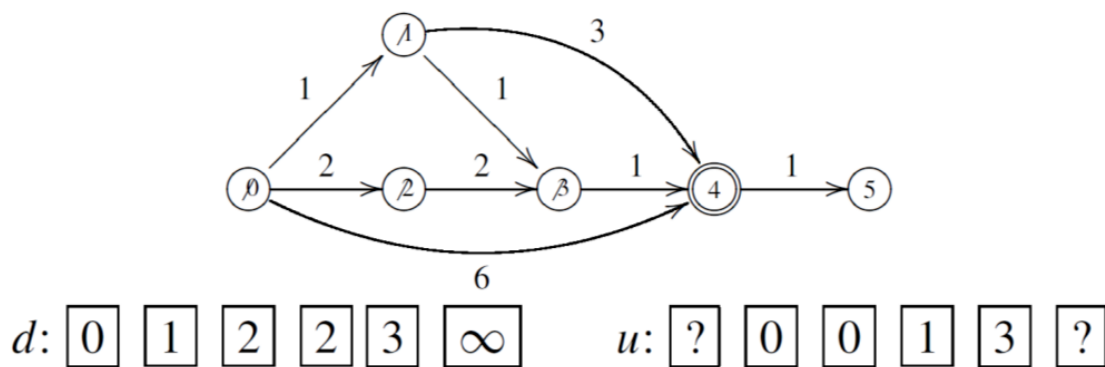


Figure 17 - step 4, visiting 3

Lastly, node 4 is visited:

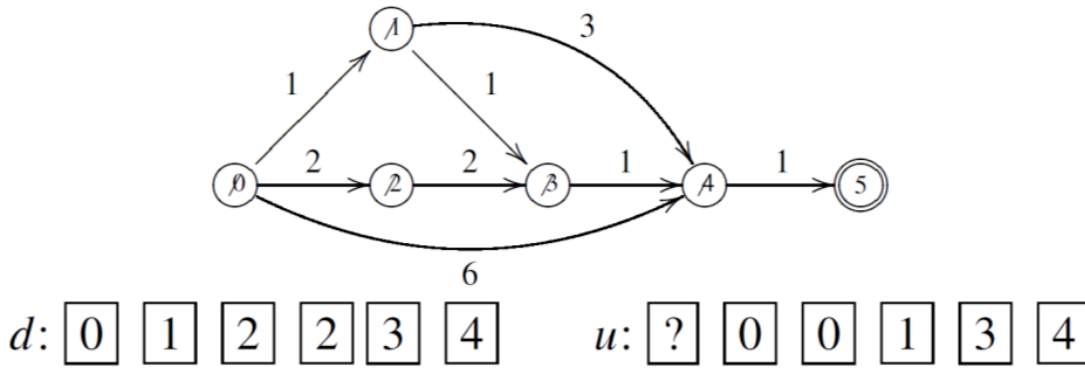


Figure 18 - step 5, visiting 4

The following step will find a path between 0 and 5, of weight 4: following back the pointers of u the path can be seen: 0,1,3,4,5.

It can be proved that Dijkstra algorithm is complete: that means that it will find an existing solution or it will be able to return an error if a solution is not found. An algorithm that is not complete will continue to work forever.

1.3.3 Sample-based algorithms

This class of algorithms presents a great advantage with respect to the grid based one; that is, reduced computational cost and high speed; differently from what happens in grid based, where the whole configuration space is explored, here the problem is analyzed in a stochastic fashion, and that is the main contributor to reducing computational cost.

The two main algorithms of this family are Probabilistic Roadmap (PRM) and Rapidly-exploring Random Tree (RRT). Both algorithms can be seen as the succession of two phases, namely:

- **Learning phase:** here the C_{free} space is analyzed and nodes are stochastically chosen and connected by archs. The set of chosen nodes constitutes a graph
- **Querying phase:** here the best path between all possible paths in the graph previously found are analyzed; graph solving algorithms are needed, such as Dijkstra.

The differences between PRM and RRT are in the learning phase, while the querying phase is common in both.

The PRM algorithm has not been analyzed, while RRT (more specifically, one of its variants) has been analyzed and implemented in the final project. The choice was carried out looking at various papers on autonomous robots in which many flavours of RRT were developed, making easy to include many types of constraints, some examples of these papers are [4] , [5] , [6] , [7] .

1.3.4 RRT algorithm

The main idea behind RRT is quite simple: it aims at creating a graph in the learning phase which will then be analyzed in the querying phase. In order to create such graph, starting at the first point of the graph (which is known to be in the free space as our robot can't start its movement inside of an obstacle), a number of random points are randomly generated in a neighborhood of the first point, and they are connected to it.

Each time a new vertex is created, tho, a check is made that such vertex lies outside of an obstacle, and inside of the free space region. Also, the link between a vertex and the previous point with which the vertex will be connected to must be obstacle free; this means that all the points that are on such link will have to be tested too.

Once a new vertex is successfully added to the graph, the algorithm starts all over on this new vertex. It stops when a node is generated within the objective region, or when a iteration limit is reached.

It is worth writing the algorithm in a list- fashion:

Learning phase of an RRT algorithm:

1. Creation of the graph $G(V,E)$ as an empty set; the starting point is added to it as the first node, called q_{start} , which is obviously belonging to C_{free} .
2. Random generation of a node q_{rand} , coming from C_{free} , following some generation parameters (more on this later).
3. Identification between existing nodes of G of the closest node to q_{rand} : this node will be called $q_{nearest}$.
4. Computation of the new node q_{new} , using a so called *steering function* (more on this later)
5. Validation of the path between $q_{nearest}$ and q_{new} : the link between them must belong to C_{free} , so it doesn't have to collide with anything.
6. Once validation is successful, adding q_{new} to V , and the arch that connects $q_{nearest}$ and q_{new} to E .
7. Repetition of the previous steps until a point q_{new} that belongs to the objective region is added to the graph, or the number of iterations reaches a pre set number n .
8. End of algorithm, with the resulting solution $G(V,E)$.

The steps presented so far are quite popular and standardized, and written in a number of papers such as [8]. The pseudo code for the algorithm is given in figure 14:

```

1:  $V \leftarrow \{q_{start}\}; E \leftarrow \emptyset;$ 
2: for  $i = 1, \dots, \text{maxNodes}$  do
3:    $q_{rand} \leftarrow \text{SampleFree};$ 
4:    $q_{nearest} \leftarrow \text{Nearest}(G = (V, E)), q_{rand}$ 
5:    $q_{new} \leftarrow \text{Steer}(q_{nearest}, q_{rand})$ 
6:   if  $\text{CollisionFree}(q_{nearest}, q_{new})$  then
7:      $V \leftarrow V \cup \{q_{new}\}; E \leftarrow E \cup \{(q_{nearest}, q_{new})\}$ 
8:   end if
9: end for
10: return  $G = (V, E);$ 

```

Figure 19 - pseudo code for RRT

1.3.5 RRT* algorithm

The first major variant of the standard RRT algorithm is the so called RRT* algorithm, which is in short “an optimized version of RRT”.

The RRT* algorithm is based on a simpler and version of the original RRT algorithm, which appears to be a worse choice. Such simpler algorithm is called RRG (Rapidly-exploring Random Graph).

The main difference between RRT and RRG is the following: each time a new node q_{new} and the consequent arch, are added to the graph $G(V, E)$, it will also be connected to a neighborhood of the $q_{nearest}$ node (as long as all the archs that will be created are collision free ones).

A rule to define the concept of neighborhood of the $q_{nearest}$ node is needed: a distance is therefore introduced:

$$r(card(V)) = \min \left\{ \gamma_{RRG} \left(\frac{\log(card(V))}{card(V)} \right)^{\frac{1}{d}}, \eta \right\}$$

where:

- The parameter η is the same parameter used in the *steering function* of step 4 of the original RRT algorithm. It is a reasonable distance from $q_{nearest}$, and in the RRG case it constitutes an upper limit to r .
- d is the dimension of the configuration space of C_{free} : in case of a 2D map, $d=2$.
- γ_{RRG} : it is a multiplying factor that depends on the dimension of configuration space and the Lebesgue measure of C_{free} . It is defined by:

$$\gamma_{RRG} > \gamma_{RRG}^* = 2 \left(1 + \frac{1}{d}\right)^{\frac{1}{d}} \left[\frac{\mu(C_{free})}{\zeta d} \right]^{\frac{1}{d}}$$

- V is the set of vertices of the graph $G(V,E)$

Comparing RRG and RRT: the end result of RRG is a graph with same vertices as the one resulting from the RRT algorithm, but with much more archs and lines that connect them; in this case, there is the possibility that some closed cycles can exist.

In the following figure 15 the RRG algorithm is reported in pseudo code form:

```

1  $V \leftarrow \{x_{init}\}; E \leftarrow \emptyset;$ 
2 for  $i = 1, \dots, n$  do
3    $x_{rand} \leftarrow \text{SampleFree}_i;$ 
4    $x_{nearest} \leftarrow \text{Nearest}(G = (V, E), x_{rand});$ 
5    $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand});$ 
6   if  $\text{ObstacleFree}(x_{nearest}, x_{new})$  then
7      $X_{near} \leftarrow \text{Near}(G = (V, E), x_{new}, \min\{\gamma_{RRG}(\log(\text{card}(V))/\text{card}(V))^{1/d}, \eta\});$ 
8      $V \leftarrow V \cup \{x_{new}\}; E \leftarrow E \cup \{(x_{nearest}, x_{new}), (x_{new}, x_{nearest})\};$ 
9     foreach  $x_{near} \in X_{near}$  do
10    if  $\text{CollisionFree}(x_{near}, x_{new})$  then  $E \leftarrow E \cup \{(x_{near}, x_{new}), (x_{new}, x_{near})\}$ 
11 return  $G = (V, E);$ 
```

Figure 20- RRG algorithm pseudo code

At a first glance it is unclear why to use RRG instead of RRT: not only it is slower, the fact that loops can exist makes the resolution of the graph more difficult. Infact, RRG is by itself a worse version of RRT, but here comes into play the peculiar aspect of the RRT* algorithm: the *rewiring* phase.

The rewiring phase is run each time a new node is connected to all nodes in its neighborhood: the new node q_{new} is reconnected to the graph not in a path that minimizes the cost of the local segment that links it to the graph, but in order to minimize the overall cost of the path that starts from the first node. Once rewiring is run for the node q_{new} , it is then run also for all nodes belonging to the neighborhood of q_{new} .

The process described so far guarantees the computational efficiency of a graph tree structure and a solution which is asymptotically optimal.

It is useful to write the total RRT* algorithm in a list fashion:

1. Creation of the graph $G(V,E)$ as an empty set; the starting point is added to it as the first node, called q_{start} , which is obviously belonging to C_{free} .
2. Random generation of a node q_{rand} , coming from C_{free} , following some generation parameters (more on this later).
3. Identification between existing nodes of G of the closest node to q_{rand} : this node will be called $q_{nearest}$.
4. Computation of the new node q_{new} , using a so called *steering function* (more on this later).
5. Validation of the path between $q_{nearest}$ and q_{new} : the link between them must belong to C_{free} , so it doesn't have to collide with anything.
6. Computation of the set X_{near} containing all nodes that lie within a distance r from q_{new} .
7. Adding of q_{new} to V .
8. Creating auxiliary variables for the rewiring phase:
 - q_{min} : initially set equal to $q_{nearest}$, it is the node prior to q_{new} and the one that minimizes the cost of the overall path from q_{start} to q_{min} to q_{new} .
 - c_{min} : it is the cost of the overall path from q_{start} to q_{min} to q_{new} .
9. Rewiring phase: the algorithm searches inside the set X_{near} the node that, if considered as prior to q_{new} , can minimize the overall cost of the path; if a node is found, the algorithm updates the variables q_{min} and c_{min} .
10. After q_{new} has been connected to the graph in the best possible way, the algorithm evaluates if the nodes belonging to X_{near} can be connected to the graph with a minor cost considering q_{new} as prior node.

In the following figure 16, the pseudo code of RRT* is provided:

```

1  $V \leftarrow \{x_{init}\}; E \leftarrow \emptyset;$ 
2 for  $i = 1, \dots, n$  do
3    $x_{rand} \leftarrow \text{SampleFree}_i;$ 
4    $x_{nearest} \leftarrow \text{Nearest}(G = (V, E), x_{rand});$ 
5    $x_{new} \leftarrow \text{Steer}(x_{nearest}, x_{rand});$ 
6   if  $\text{ObstacleFree}(x_{nearest}, x_{new})$  then
7      $X_{near} \leftarrow \text{Near}(G = (V, E), x_{new}, \min\{\gamma_{RRT^*}(\log(\text{card}(V))/\text{card}(V))^{1/d}, \eta\});$ 
8      $V \leftarrow V \cup \{x_{new}\};$ 
9      $x_{min} \leftarrow x_{nearest}; c_{min} \leftarrow \text{Cost}(x_{nearest}) + c(\text{Line}(x_{nearest}, x_{new}));$ 
10    foreach  $x_{near} \in X_{near}$  do // Connect along a minimum-cost path
11      if  $\text{CollisionFree}(x_{near}, x_{new}) \wedge \text{Cost}(x_{near}) + c(\text{Line}(x_{near}, x_{new})) < c_{min}$  then
12         $x_{min} \leftarrow x_{near}; c_{min} \leftarrow \text{Cost}(x_{near}) + c(\text{Line}(x_{near}, x_{new}))$ 
13     $E \leftarrow E \cup \{(x_{min}, x_{new})\};$ 
14    foreach  $x_{near} \in X_{near}$  do // Rewire the tree
15      if  $\text{CollisionFree}(x_{new}, x_{near}) \wedge \text{Cost}(x_{new}) + c(\text{Line}(x_{new}, x_{near})) < \text{Cost}(x_{near})$ 
16        then  $x_{parent} \leftarrow \text{Parent}(x_{near});$ 
17         $E \leftarrow (E \setminus \{(x_{parent}, x_{near})\}) \cup \{(x_{new}, x_{near})\}$ 
18 return  $G = (V, E);$ 

```

Figure 21 - pseudo code for RRT*

In figure 17 an example of the rewiring phase is given: at first the new node q_{new} was linked to the upper part of the graph, while the two nodes in its surroundings were connected to the bottom part of it. During the rewiring phase, it is seen that a smaller overall cost is achieved if the two nodes are connected to the new node q_{new} instead.

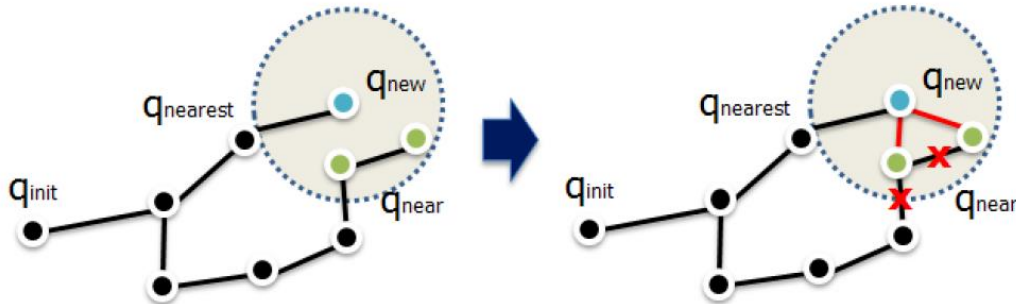


Figure 22- rewiring example

In Figure 18 a result from running the RRT* algorithm has been reported. The implementation has been written by [9] (trova matlab implementation). The run time on an i7 dual core processor varies between 7 and 9 seconds, depending on the random creation of the trees. Still, it can be considered adequate for a one-time path planning before the navigation starts.

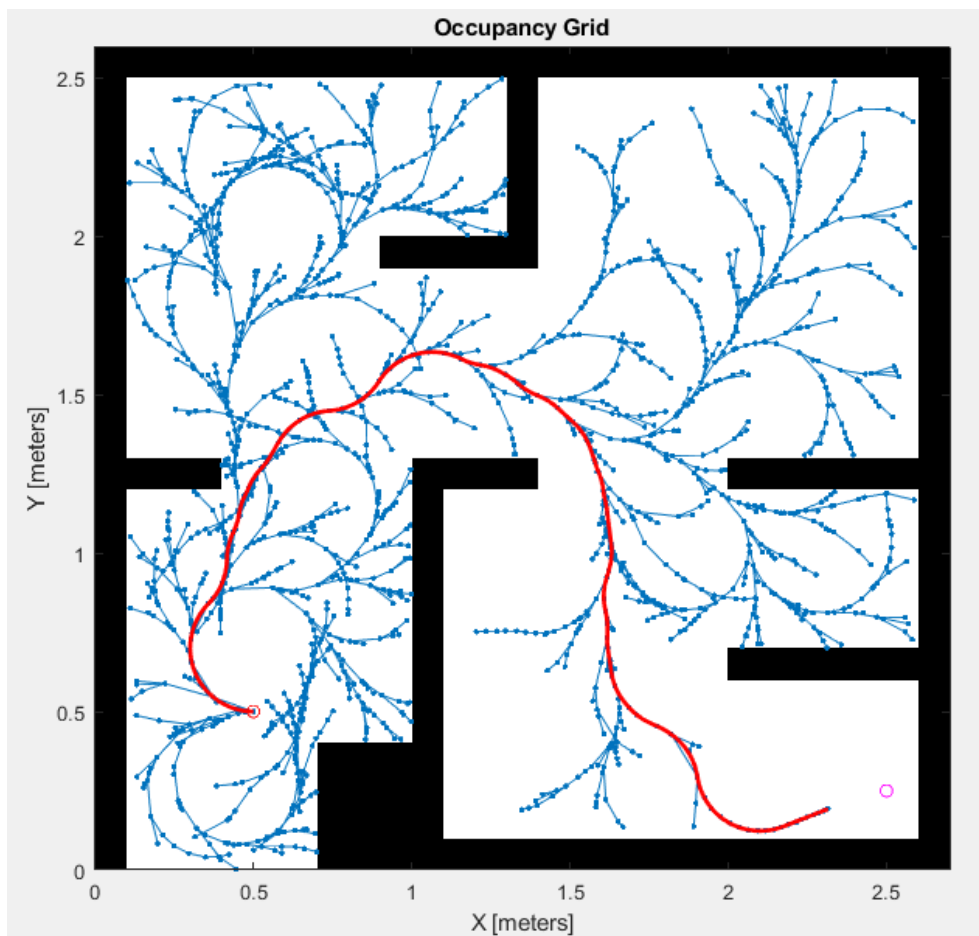


Figure 23 - Result of RRT* implementation in Matlab

CHAPTER 1 - BIBLIOGRAPHY

- [1] : Biolè D., “Analisi di algoritmi di path-planning e obstacle avoidance in uso nei sistemi AGV”, 2021
- [2] : Steven M. LaValle, “Planning Algorithms”, 2006
- [3] : S. Vigna, “L’algoritmo di Dijkstra”, 2006
- [4] : Hyunki Kwon et al., Trajectory Planner CDT-RRT* for Car-Like Mobile Robots toward Narrow and Cluttered Environments, 2021
- [5] : Dustin J. Webb, Kinodynamic RRT*: Optimal Motion Planning for Systems with Linear Differential Constraints, 2012
- [6] : Bragaglia Matteo et al., Poli-RRT*: optimal RRT-based planning for constrained and feedback linearisable vehicle dynamics
- [7] : Feraco Stefano et al., A local trajectory planning and control method for autonomous vehicles based on the RRT algorithm, 2021
- [8] : S. Karaman et E. Frazzoli , Sampling-based algorithms for optimal motion planning, 2011
- [9] : MATLAB RRT* implementation by Adam Munawar

CHAPTER 2 – SENSORS AND METHODOLOGIES FOR AUTONOMOUS NAVIGATION

2.1 Obstacle perception

The algorithms and methods explained in chapter 1 belong to what can be defined as the offline part of the mission. That is, they exploit all available information on the map at the moment of starting the mission. We can call this information “a priori knowledge”, and the path created on the basis of the a priori knowledge can be called “reference path”; it will be the goal of the control algorithm to try and follow as closely as possible the reference path.

In certain fields of robotics the a priori knowledge is all that is needed in order to succeed the mission; a valuable example of application are welding robots: welding robots usually operate under extremely controlled conditions, and the path of the end effector can be wholly computed before the beginning of welding operations, as it is extremely unlikely that new and unexpected obstacles will enter the working zone of the welding robot. Thus, in this case and in general in the case of CNC machining, the reference path will be only computed once and followed without intermediate modifications.

Unfortunately, for the case study of an autonomous telescopic handler, it is unlikely that the working environment will be isolated from the external world once the reference path is computed. It is worth noting that, especially at this time and progress of autonomous vehicles for construction and agriculture, human-machine cooperation is still strongly discouraged; the machines have not yet reached a degree of safeness such that working while humans are present is considered a feasible activity. In fact, work of an autonomous vehicle is suggested to operate in an environment isolated from humans, and the machine should reach a safe state whenever human presence is detected. The laws are actually still quite behind in this particular field, but some general ideas have been found in the drafts of ISO norms such as [1]. However, even if the machine is suggested to operate in non human presence, human interaction still has to be taken into account, and it usually means a stopping of the activity of some sort.

In this chapter we don't focus on human interaction, but on the more general topic of obstacle detection. Without considering the case where the obstacle is human, obstacle detection is of paramount importance in this application of autonomous navigation, as the usual working environments of the telescopic handler can be constantly entered by vehicles and animals. Non moving objects are also a problem: in fact, every change to the environment that is not present in the a priori knowledge must be taken into account. This includes both interacting with moving objects and also avoiding still objects that are left in the way of the reference path.

The first step towards obstacle identification and avoidance is the perception step, for which a number of sensors and methodologies have been developed.

2.1.1 Ultrasound sensors

Ultrasound sensors are a family of devices that use ultrasonic waves to perform different kinds of measurements; in an obstacle detection framework, ultrasound sensors are used to measure point to point distance.

An ultrasonic sensor is usually composed of a transmitter and a receiver: the transmitter is made of a transducer that converts electrical energy into ultrasound, while the receiver receives the sounds and converts it into an electrical energy; different kind of measurements are then performed on this electrical quantity to reconstruct the distance of the perceived obstacle.

More in detail, the most popular technique for measuring point to point distances is called *sonomicrometry*: it consists of transmitting and receiving discrete bursts of ultrasound between a transmitter and a receiver. The transmitter emits a short burst of ultrasounds which travel through the air until an object is encountered, which makes the burst bounce back to the receiver; when the burst is emitted a timer is started and when the receiver senses an incoming burst the timer is stopped. Assuming that the speed at which the ultrasonic wave travels through the medium is known, then the traveled distance can be simply computed via:

$$d = \frac{\text{time} * c_s}{2}$$

where c_s is the medium's speed of sound.

The use cases of ultrasonic sensors are quite vast and diverse: they can be used for measuring liquid levels, for measuring the presence or absence of certain gasses in an environment; these cases are relevant because they take advantage of one key characteristic of ultrasonic sensors: their performance does not depend on light conditions. They work equally well during daytime and nighttime, and the surface texture of the particular obstacle is not important.

One of the major fields of application of ultrasonic sensors is parking assistance, where they have been employed for years. This is because they are cheap sensors, and their accuracy is usually quite good: 1% to 3% of the detected range is standard, while under controlled conditions it can be brought down to 0.1% to 0.2%. [2].

Their range varies from sensor to sensor, but most popular sensors work in a 1 cm to 1000 cm range; thus, we can classify them as short range sensors.

They are mainly used for gross detections, where orientation and shape of the object are not important, as they are only able to give a single measurement per object.



Figure 1- HC_SR04 a popular DIY ultrasonic sensor

2.1.2 Radar

Radar indicates a technology that measures distances and direction of an object using radio waves. The name Radar is an acronym and stands for Radio Detection and Ranging.

The principle at the heart of radar is called backscattering, and it is not too different from the basic principle of the ultrasonic sensor. Instead of sound waves, radio waves are used. More in detail, when a radio wave hits an object of size larger than its wavelength, parts of it are bounced away from the object. In the case the object size is smaller than the wavelength, other phenomena occur, such as diffusion or diffraction.

The backscattered radio wave is then received by an antenna, that computes the distance based on the same formula used in the ultrasonic sensor, that is:

$$d = \frac{\text{time} * c_r}{2}$$

where c_r is the medium's speed of propagation of the radio wave, about 300m/μs.

Is it also possible to compute the azimuth angle of the detected object.

A radar system is composed of:

- An antenna, which acts both as a transmitter and as a receiver.
- A waveguide, which is a hollow metal pipe that is used to confine and carry radio waves. A typical radar waveguide application is reported in figure 2.
- A duplexer, which is a device used to decouple the transmitted wave from the received one.
- A timing device that is some sort of very accurate and precise clock.

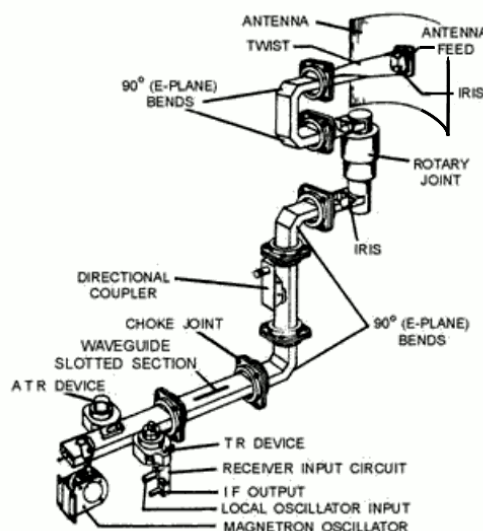


Figure 2 - waveguide for a rotary radar system

The radar transmitter produces waves that are strongly directional, and are parallel to the horizontal plane on which the radar is mounted on. This is also referred to as the azimuth plane. Due to the strong directionality of the antennas that act both as transmitter or receiver, if a 360° field of view is required, the antenna will have to be mounted on a rotating joint, such as the one shown in figure 2.

The radar technology comes in different flavors for different applications, and thus the range of distances and accuracies varies a lot, but all applications have the same kind of problems: multipath fading, constructive and destructive interference, internal and external radio noise in the transmitter or in the receiver... This array of uncertainties makes it mandatory to elaborate the radar signals in a stochastic way; in this sense, as more widely in the telecommunication field, probability concepts such as false positives (false detections) and false negatives (missed detections) are introduced. The aleatory nature of radar measurements thus implies that some sort of filter, algorithm, and probabilistic criteria are needed in order to correctly handle the results.

In the autonomous vehicle field, radars have been widely used both in experimental and in commercial vehicles (one notable example is given by Tesla's lineup). In this application, the dimensions of the sensor are quite smaller with respect to the long range military counterparts (figure 3a), and sensors are usually compact and flat (figure 3b).

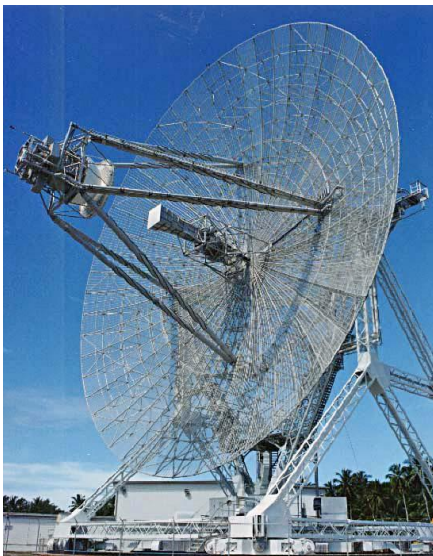


Figure 3a: military long range radar



Figure 3b: automotive grade radar

Working frequencies for automotive radars are in the range of 75 – 110 GHz , and in the IEEE radio spectrum frequency bands they are classified as W, with wavelengths in the order of 2.7 mm to 4 mm [3].

The range of a commercially available radar system for automotive lies in between 1m – 2m and 100m. A single sensor can give multiple detections in a single scan, making it possible to not only evaluate an object distance but also its shape, if a suitable clustering algorithm is coupled with the

measurement. One of the good features about Radar is the fact that it is resilient with respect to adverse weather conditions: Radar works equally well in all weather conditions such as fog, rain, and snow, and dust. Another upside of Radar is that it can determine relative traffic speed or the velocity of a moving object accurately using the Doppler frequency shift, and this feature is unique to Radar. The main downside to this technology is that it is not angularly accurate, and scans performed using a Radar present distorted objects, whose shape can vary considerably from the actual one; in later years, high definition Radar have mitigated this problem. An example of Radar distortion and a comparison with the Lidar sensor (see next paragraph) is given in Figure 4.

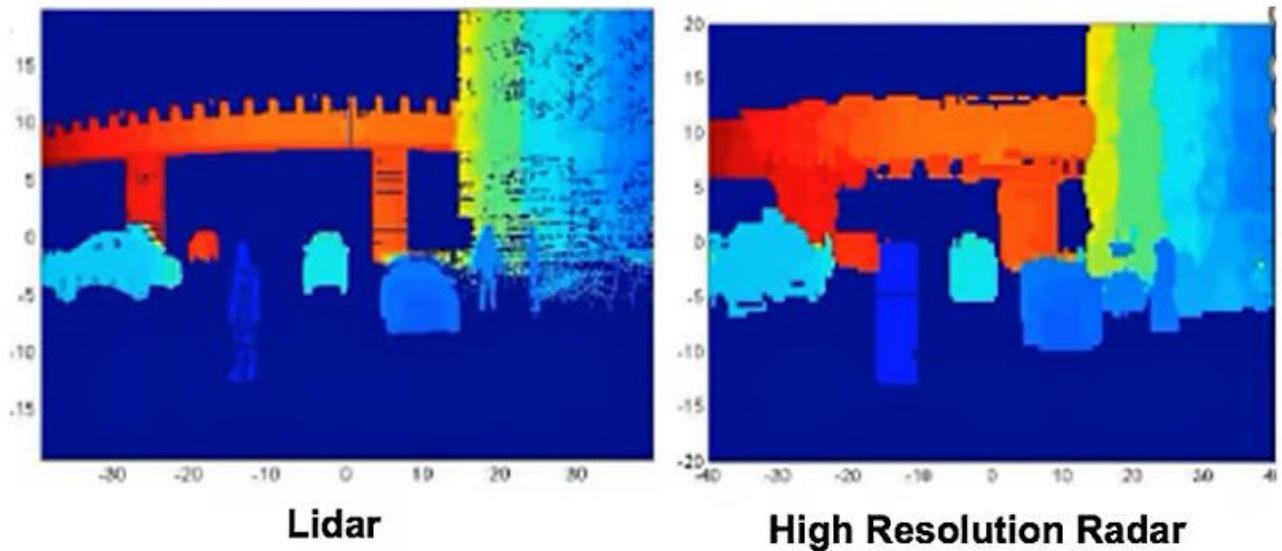


Figure 4: comparison between Lidar and high resolution Radar scan. The details of the bridge highlight the radar distortions

2.1.3 Lidar

Lidar is an acronym and stands for Light Detection And Ranging, reminiscent of the Radar acronym.

The concept at the heart of Lidar is very simple: Lidar sensors perform scans by emitting laser pulses, which then bounce back and are read from a receiver. Again, the formula used to measure the distance to the obstacle is:

$$d = \frac{time * c_l}{2}$$

where c_l is the speed of light.

Lidar can be used to measure objects of all sorts of materials: stones, water, chemical substances, clouds, terrestrial objects and even single molecules, thanks to the laser beam being extremely narrow. Lidar is widely used in aerospace applications, to map terrain from aircrafts and in atmospheric research: in these kinds of applications high energy Lidars are employed.

For autonomous vehicles applications another kind of Lidar technology is used: micropulse. Micropulse Lidar uses less power with respect to the high energy counterpart, and is often “eye

safe”, meaning that it can be operated in human populated environments without the need for eye protection. This is of course crucial in applications such as automotive but also agricultural and construction.

Lidar sensors can be divided into categories based on their scanning mechanism:

- *Spinning Lidar*: they employ a single laser beam that is directed towards a spinning mirror: the mirror redirects the beam towards the area to be scanned, and receives back the return beam. Since a mirror usually rotates around one axis, it can only scan radially. In order for it to scan across a 2D field of view, one option is to add a second mirror that redirects the laser beam perpendicular to the plane of the first mirror. Another option is to use one single mirror and two laser beams. This technology allows for a sensor with 360° of coverage, making it ideal to be mounted on top of vehicles for a complete view of the surroundings.
- *Solid state Lidar*: it is also called phased array Lidar system. This solution comes from the world of radars, and it employs an array of microscopic antennas, in the order of 10^6 antennas. The principle is to time the flash of antennas (in Lidar case: lasers) so that the incoming radiation measured can be correlated with the flash timing. This allows for the sensor to not have any moving parts, and the big advantage is a great increase in its lifespan: 100000 hours vs 1000 hours of the spinning Lidar system. Once the technology is mass produced, it also allows to reduce costs.
- *MEMS Lidar*: MEMS is an acronym which stands for microelectromechanical mirrors. Their working principle is the same as the spinning Lidar, but in a much smaller factor: they are cost effective and lighter with respect to spinning Lidar, and can potentially become a competitor of solid state Lidar. At the current state of commercial products, they are prone to disalignment due to vibration. This latter aspect is not to be under-estimated in the agricultural and construction fields, where the vehicles are subject to lots of vibration sources, both external (uneven and difficult terrain) and internal (engine and on-board high power appliances).
- *Flash Lidar*: flash Lidar follows a different principle with respect to scanning Lidar: the entire field of view is illuminated at once with a diverging beam in a single pulse. In scanning Lidar the sensor is a single point sensor, while in flash Lidar the camera contains a 1D or 2D sensor array. This can be particularly useful in cases where the vehicle with the Lidar mounted on is moving at some speed, as flashing a whole image at once eliminates the disalignment problems that arise during scanning of a wide field of view.

A Lidar system is composed of many different components, two of which are constant throughout the various technologies:

- **Laser**: in general, the laser used for Lidar systems lies in the range of 600 nm – 1550 nm; the fields of application are very diverse and include autonomous vehicle navigation, aerial inspection, precision agriculture, forestry and land management, cartography and mapping, and military applications too. Some of the named applications can be classified in

airborne Lidar applications, meaning that the sensor is mounted on a drone or plane. In regards to autonomous terrestrial vehicles use, common wavelengths are 905 nm and 1550 nm [4].

A study performed in 2014 [5a] has highlighted advantages and disadvantages of the two wavelength technologies, with the focus on adverse environmental and weather conditions. Water is absorbed about 140 more times by the 1550nm solution, and while operating in rain and fog the degradation of the 1550nm waves is 4-5 times worse than the 905 nm counterpart. Power consumption in wet conditions is also a negative side for the 1550 nm solution, as they require more than 10 times more power than a similar 905 nm system. Their performance under good weather conditions is comparable.

Since the telescopic handler will have to operate outdoors in variable and not optimal weather conditions, the 905 nm technology appears more adequate to the mission scenarios. The Velodyne HDL 32-E, which is the sensor with which the datasets used in this thesis were recorded, is a sensor which uses 905 nm wavelength.

Photodetector/ receiver: there are two main technologies used in Lidar systems: solid state photodetectors and photomultipliers.

- Time of flight camera: a device that measures the time of flight of a single laser pulse.

Lidar sensors have been used extensively in recent years in the autonomous vehicle field, and in research projects such as the one reported in [5], including various prototypes; while Lidar system have the disadvantage of being more expensive than other mid range sensors such as radar, their precision and accuracy makes it a great tool for fast prototyping and testing of algorithms on a real robot or AV. Most autonomous car manufacturers, such as Google and Toyota are using Lidar as their prime sensor. In the context of urban navigation, the difference between a Lidar scan and a radar scan can be seen in figure 5 (Lidar scan) and figure 6 (radar scan): in the Lidar scan there is an amount of detail much greater than in the radar scan: the accuracy and angular definition makes it easy to understand what are the objects being scanned, and it is easier to see edges and shapes of the various obstacles (within certain limits).

The radar scan, on the other hand, shows less detail and establishing the exact shape and location of the obstacles may become a difficult task if only the raw data from the sensor are used. It needs to be said that state of the art radar devices for automotive use offer performance similar to Lidar devices, but the angular accuracy and distortion are still present.

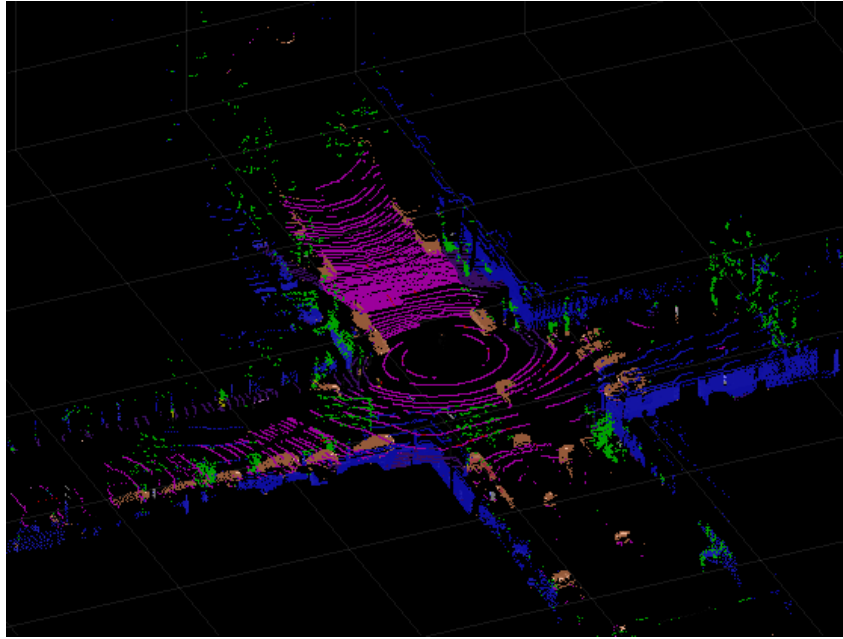


Figure 5 - Lidar scan of a crossroad, the vehicle performing the scan is situated at the center of the scene, in the black void

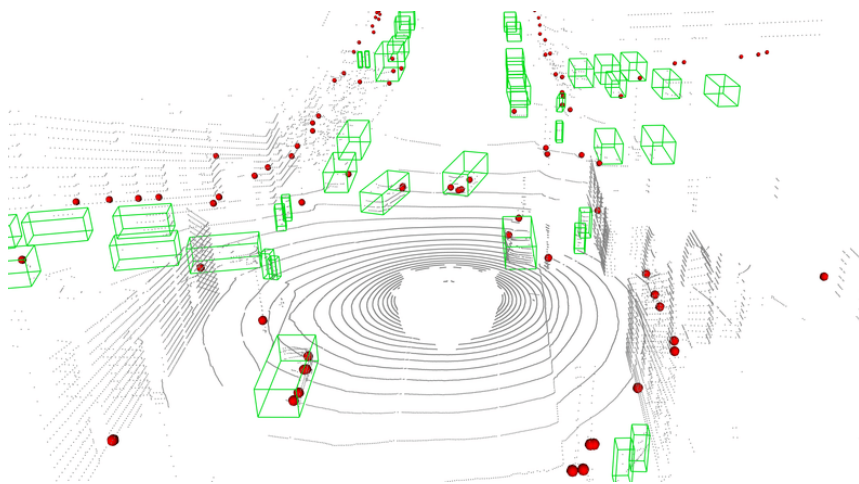


Figure 6 - radar scan of an urban environment, the scanning vehicle is placed in the center, in white

For the application studied in this thesis, a single spinning Lidar sensor has been chosen. The main reasons for this choice are:

- Greater detail of the scans with respect to radar scans imply a better identification of obstacle location, size and orientation; furthermore, better resolution and higher density make the point cloud clustering (which is the raw data outputted from the sensor) algorithms work better. This is the major reason for the choice of the sensor, as the focus of this thesis was the analysis of the algorithm chain needed for navigation. Not having to deal with the stochastic processing of the data coming from a non high end Radar sensor was a good simplification.

- One useful feature of the Lidar readings is that they are able to penetrate, with certain limits, leaves of grass in a field. This can be useful if the working area of the vehicle is an uncut field.
- Although the sensor is quite expensive, the product which it would have been used on was a high end telescopic handler. In the case of a mass production of autonomous vehicles and where cost is a more crucial factor, radar technology would have been a better choice, with additional measures such as the ones described in 2.1.5.

2.1.4 Cameras for obstacle detection

The use of cameras is widespread across all kinds of autonomous navigation applications, and their usage varies a lot.

The sensors described so far all work according to the same principle: sending some kind of wave or signal and waiting for a return signal to come. Computation of distance comes from measuring the elapsed time.

Cameras don't send any signal out, so if one wants to use them for obstacle detections other methods are to be used. An example of such methods is to use two cameras in a configuration called *stereoscopic cameras*. This kind of configuration has been used in various robotics applications, some of which concerning exploration missions (figure 7).

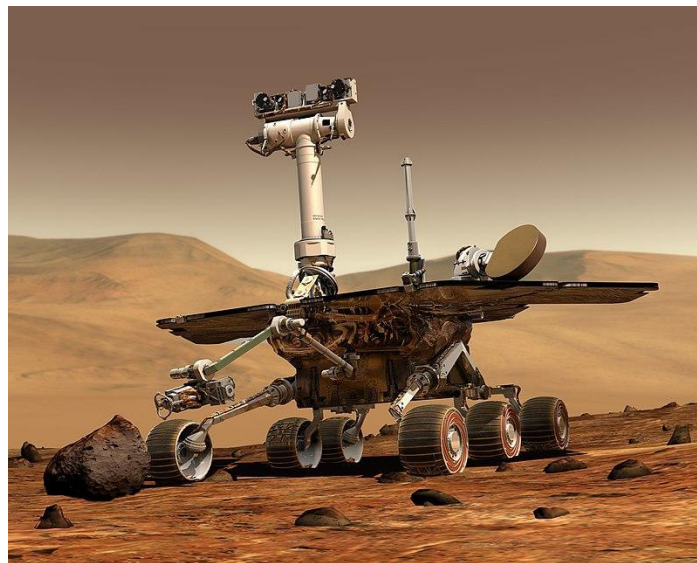


Figure 7- Mars exploration rover equipped with stereo cameras

By themselves, stereoscopic cameras provide two video signals but no information regarding the depth of the objects being recorded is provided.

Stereo cameras recordings are to be coupled with computer vision techniques in order to extrapolate depth information from the recordings. This essentially means that some kind of algorithm has to be continuously run on the camera data only to get information about depth, the so-called depth map. This is obviously an extra computational cost if compared to, for example, a Lidar system solution, where the stream of data from the sensor already contains depth information.

An alternative solution to depth mapping using stereo cameras is to use both a normal camera and an infrared camera. This configuration is the one used in the famous Kinect sensor (figure 8 and figure 9).



Figure 8 - Kinect sensor

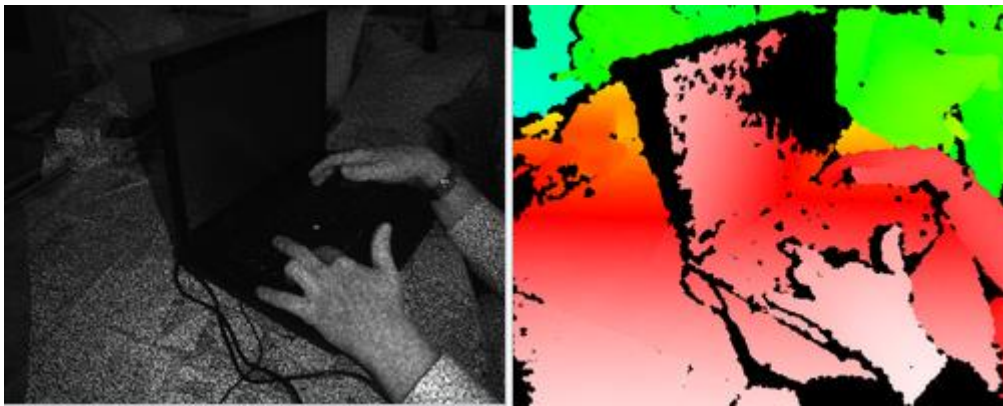


Figure 9- Output of kinect sensor. Closer objects are depicted in red, far objects are in blue

The working principle of the kinect is the following: a narrow band of light (infrared light in this case) is projected to the environment; when this band of light encounters a 3D surface, the band is deformed on this surface. The infrared camera captures the distortion of the band, and computations can be made to reconstruct the shape of the object. Usually, not only a single band of light is projected, but a whole stripe pattern; this allows to capture depth information of the whole image.

2.1.5 Cameras for object classification and sensor fusion

Obstacle detection is not the only field where cameras are used. In fact, a major application for cameras is object classification. Object classification consists in taking an image of an object as an input and getting a category as an output. Usually a number of output categories are given, each with its own probability. For example, a picture of a cat used as an input for object classification is likely to be labeled as “cat” and as other similar animals, in a well trained algorithm. In figure 10 a typical output of a well trained algorithm is shown. The problem of object detection and classification has received incredible interest in the last years, great progress has been made and the interest seems to grow each year [6].



Figure 10 - typical output for classification problem

The task of classification is tackled using the classical machine learning algorithms and methodologies such as Naïve Bayes estimators, Support Vector Machines, Feedforward Neural Network. More recently, the so-called deep learning technologies such as Convolutional Neural Networks have been recognized as working extremely well, and have already been used on commercial vehicles such as Tesla cars.

In most computer vision applications, the camera+algorithm system must be able to perform another task: edge detection; that is, the understanding of the boundaries of an object. Again in figure 10 it is possible to see edge detection at work: all the classified objects have a bounding box that surrounds them. Multiple object recognition and edge detection is a much more difficult task than object classification on its own, also because in the latter it is easier to give training and validating data to the algorithm. In figure 11 a typical road output of a commercially available vehicle is given: bounding boxes around objects show the category in which the object is recognized into, as well as its location.

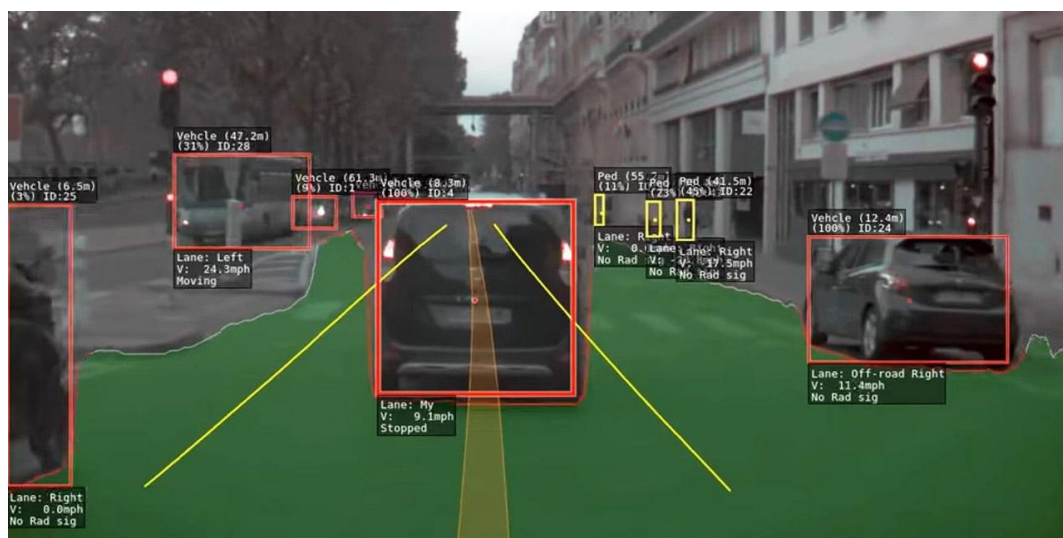


Figure 11- Output of a Tesla vehicle

In all advanced autonomous applications, a mixture of all the sensors presented above is used; Lidar, Radar ultrasonic sensors and multiple cameras with Machine Learning algorithms are often

used together by fusing their sensor data, in order to exploit the better qualities of each technology and to improve redundancy. This approach is often referred to as **Sensor Fusion**, and one of its advantages is that it is able to soften the “blind spots” of each sensor. For example, a highly reflective object may cause problems in the Lidar sensor detection, but it is correctly detected by the Radar sensor. Dirt particles can obstruct the Lidar receiver and thus cause loss of detections, but this is not a problem for the Radar receiver. Lidar sensor struggles at extremely close distances, while ultrasonic can be used to cover areas close to the machine chassis. Radar doesn't have problems in fog, rain, snow or dust conditions.

The frame shown in figure 12 comes from a Lidar and a Radar sensor mounted on a vehicle in a highway scenario (source: Matlab). The detections obtained by the Lidar and the Radar are fused together in order to improve robustness and mitigate the effects of false positives and false negatives of each sensor.

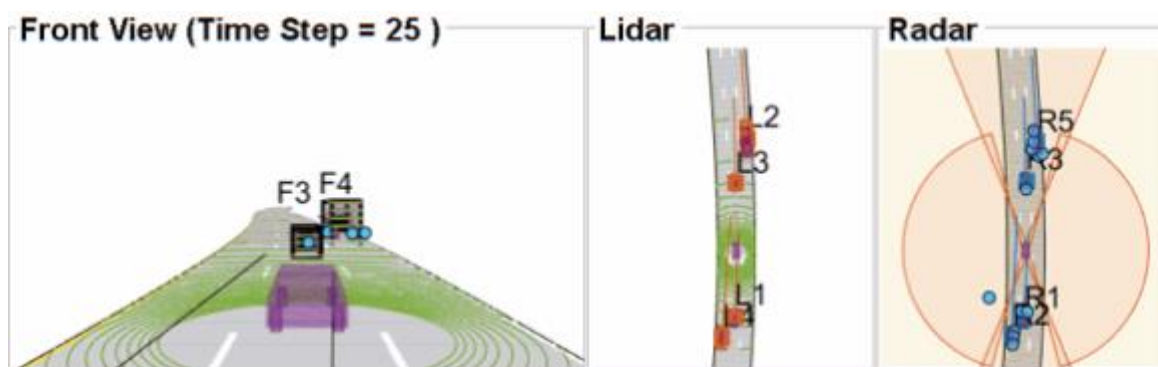


Figure 12- Multiple sensor data. Source: Matlab

2.2 Localization

In this thesis work, in the algorithm section, the hypothesis of perfect localization of the telescopic handler has been made. This fact was used, in particular, to navigate the a priori map, following a reference path. It is surely a strong assumption, but the development of localization technologies in recent years makes it a reasonable one. The approaches considered for this task are two: the usage of a GPS sensor, which relies on satellite data, and the use of SLAM techniques, which rely solely on data observed onboard.

2.2.1 GPS RTK

The GPS is a widespread technology that uses satellite networks to estimate the position of a receiver. More in detail, GPS is a type of GNSS, which stands for Global Navigation Satellite System, which works by connecting the receiver to four or more satellites belonging to the network.

In order to compute the receiver position, the distance between each satellite and the receiver must be computed, and it can be done by measuring the time the signal takes to travel from the satellite to the receiver. Such a delay is found by aligning a pseudorandom binary sequence that is generated on the satellite to a sequence generated on the receiver. The two sequences are generated as a function of atomic standard time, for example GPST, which is a time that is not subject to the corrections that other times on earth have, like leap seconds.

The sequence sent by the satellite will arrive at the receiver with some delay equal to the time it needs to travel the distance; therefore the receiver will receive a sequence which is delayed from its own internal one. The time elapsed for the radio signal to travel from the satellite to the receiver is found by aligning the two pseudorandom sequences, counting how many steps far away the two are. This process is subject to various errors like ionospheric delays and clock errors.

The distance from only one satellite is not enough to determine the receiver location; with each satellite to receiver distance, what is found is the locus of points on earth that share that distance from a satellite. Therefore, a number of satellites have to be employed, at least four, and the intersection of all the solutions from all the satellites gives the position of the receiver. These distances are referred to as pseudoranges: as there are accuracy errors in the time measured from all different satellites, the term *pseudo-ranges* is used rather than ranges for such distances, as they present the same error.

The precision of GNSS systems such as GPS and GLONASS is in the order of 1 meter in the average case. Although this is enough for directions, which is the “classic” use of commercial GPS receivers, it is not enough for navigation tasks such as the ones the considered telescopic handler will have to face. In such cases, a variant of the GPS system can be introduced: GPS RTK, which stands for GPS Real Time Kinematics.

GPS RTK technology is composed of a receiver, mounted on the vehicle, like in the normal GPS case, and a fixed base station (figure 13).

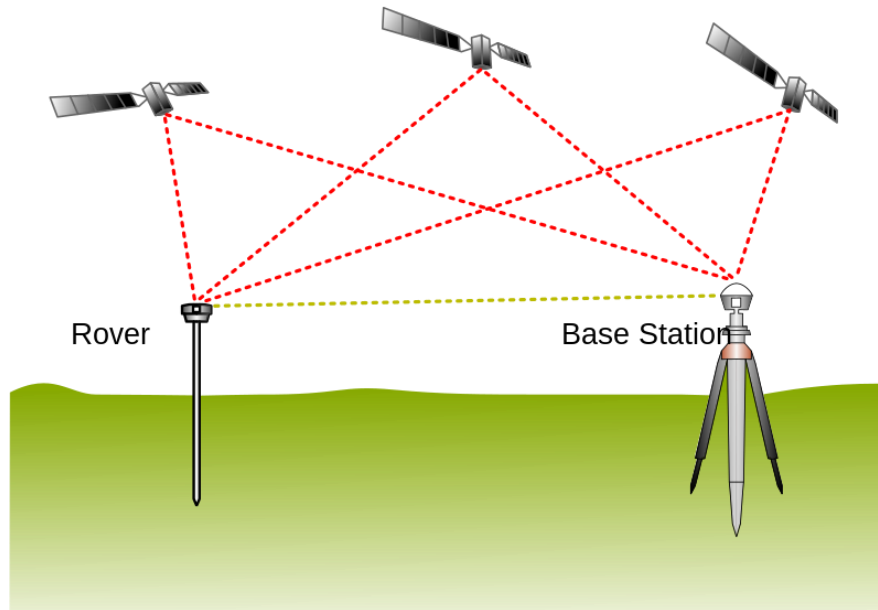


Figure 13 - GPS RTK configuration

The difference with the classic GNSS is given by the base station: the base station is a receiver and transmitter placed near the working area of the vehicle; the maximum range for precision enhancement is indicated to be about 20 km [7]; since the base station's position is fixed, it can be known with great precision, and can be continuously improved.

The mobile unit position is enhanced with several methods [8]; the first one is carrier phase measurements: the base station transmits to the mobile unit(s) the phase of the carrier wave of the signal sent by the satellite, in addition to the information content of the signal. The mobile unit confronts their phase measurement with the one provided by the base station. This correction is transmitted using several different ways, one of the most popular being UHF(ultra high frequency) radio signals.

The second one is Doppler Measurements, which can be considered as a scaled version of the time-derivative of the carrier phase measurement [8].

The absolute position accuracy of a single receiver with respect to Earth can be improved up to the accuracy of the base station, which is expected to be high as it doesn't move for very long times.

RTK has another interesting feature: in the case of multiple mobile units their relative position can be computed within millimeters margin. This could be an interesting feature to exploit in structured navigation environments, and would be of paramount importance in a fleet scenario, where multiple vehicles work cohesively to reach a certain task.

Practically speaking, an accuracy of some centimeters is a realistic estimate of the performance of an RTK system in good conditions, and it is enough to validate the hypothesis that the position of the handler is perfectly known, within certain performance requirements. The base station is not a limiting factor, as the working scenario considered is the one of a farm; since the base station needs to be within 10/15 km from the mobile receiver, this will be good enough for most farms.

Also, it is likely that the working area will always be the same, and so installing a base station and leaving it unmoved for long periods of time is a reasonable scenario.

That said, GPS RTK still requires sky visibility at all times.

2.2.2 SLAM

One of the requirements of any GNSS system is clear sky view, and again this is reasonable to expect in most agricultural applications, but as long as autonomous vehicles go there is another tool widely used in the field, and that is called SLAM (Simultaneous Localization And Mapping).

SLAM can be defined as the computational problem of defining the vehicle location in an unknown map. In order to tackle such a task, the vehicle needs to navigate the map and gather data about it. Thus, this is a problem that needs to be solved when either no map data is available, little map data is available or no position data is available, for example in the case in which the vehicle needs to be working underground or where there is no signal with satellites.

This problem is solved in different ways according to the different sensors and on board computers available; this means that the objective is an approximate solution, good enough for navigation. Due to the uncertain nature of this kind of task, the approach followed by most of the algorithms is to treat both map data and vehicle position as probabilistic variables. That is why in some basic versions of SLAM algorithms filters such as the Extended Kalman Filter are employed. The core steps of any SLAM algorithm are [9]:

- Landmark extraction: the algorithm tries to extract relevant features from sensor measurements, such as sharp corners, particular shapes etc.
- Data association: where landmarks are combined together trying to build a coherent map
- State estimation: position and orientation of the vehicle are estimated
- State update: position and orientation are updated
- Landmark update: map estimate is updated according to new measurements

Each single step presented above can be solved by different kinds of algorithms, according to what data is available.

The employed sensor determines the types of algorithms that will be used and the likelihood of a good solution to happen. For example, Lidar sensors are a great choice for SLAM applications as they provide lots of information about shapes and potentially important landmarks that are easy to recognize due to the high resolution and accuracy of the Lidar. On the other hand, visual data has also been used in SLAM algorithms, called vSLAM (visual SLAM), as cameras provide great amounts of detail. In figure 14 a map built using a Lidar sensor and a SLAM algorithm has been built:

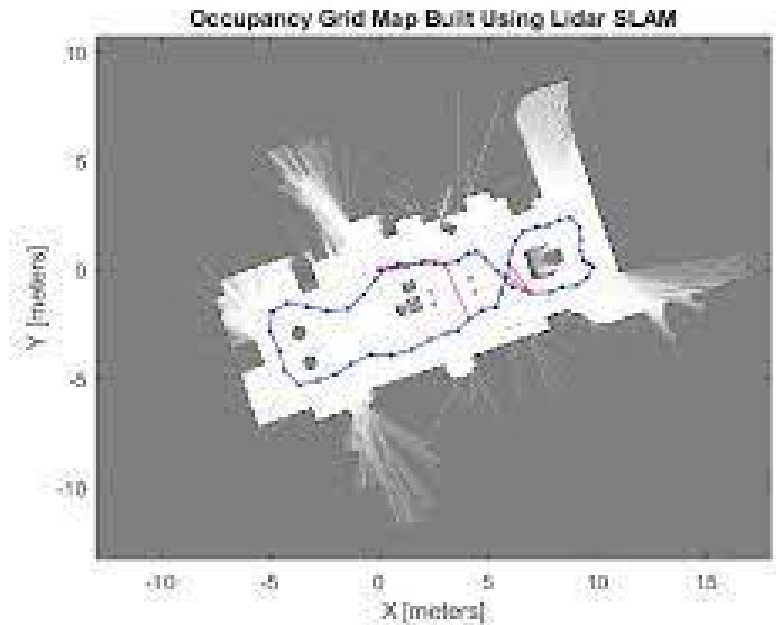


Figure 14 - An occupancy grid built using SLAM with a Lidar sensor

One of the key aspect of the SLAM problem is the so called loop closure. That is, once the vehicle gathers data of a place it already has visited, it needs to be able to understand that the map portion computed before and the one it is witnessing now are the same. A missed loop closure can lead to great misalignments form the real map, and even fail the whole process. Similarly, an accepted loop closure in a place where there shouldn't be one can have similar severe consequences. The need for loop closure comes from the unavoidable odometry errors that occur during navigation. During navigation, the vehicle needs to keep track of the direction and distance it is moving to, and to do so without the availability of a GNSS system is only possible using odometry sensors; these kind of sensors are usually encoders of some sort mounted on the wheels, that measure the number of wheel turns. But these measurements can be subject to various kind of uncertainty, like wheel slipping etc. That means, measured distances associated with certain map frames do not correspond to actual distances and position and thus lead to increasing misalignments.

In figure 15 it can be seen that a successful loop closure allows for correction of this kind of position errors.

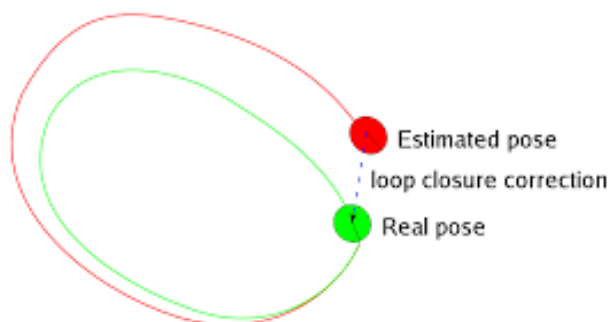


Figure 15 - discrepancies between real pose and estimated pose

In this thesis no SLAM algorithms have been used, as the use of a GPS RTK was considered a feasible option, with the hypothesis that the sky is visible at all times. The position of the vehicle in the a priori map has been considered perfectly known.

CHAPTER 2 - BIBLIOGRAPHY:

- [1] : ISO-TC23-SC19-WG8 Draft, WD ISO 18497-3, Autonomous operating zones, 2022
- [2] : senix.com/ultrasonic-sensor-accuracy
- [3] : <https://en.wikipedia.org/wiki/Radar>
- [4] : <https://velodynelidar.com/blog/guide-to-lidar-wavelengths>
- [5a] : Jacek Wojtanowski et. al., Comparison of 905 nm and 1550 nm semiconductor laser rangefinders' performance deterioration due to adverse environmental conditions, 2014
- [5] : Qingying Ge, Aijuan Li et al., Improved Bidirectional RRT * Path Planning Method for Smart Vehicle, 2021
- [6] : Zhengxia Zou, Zhenwei Shi, et al., Object Detection in 20 Years: A Survey, 2019
- [7] : Wikipedia.org, real time kinematic positioning, https://en.wikipedia.org/wiki/Real-time_kinematic_positioning
- [8] : RTK Technology - ANavS
- [9] : Søren Riisgaard and Morten Rufus Blas, "SLAM for Dummies"

CHAPTER 3 – LIDAR DATA CLUSTERING

3.1 Lidar sensor

3.1.1 Choice of sensor

The sensor chosen for this thesis work is the Velodyne HDL – 32E, shown in figure 1. It is a 360 degree field of view sensor, belonging to the spinning Lidar category. The main choice factor was the vertical resolution: lidar sensors mainly come in three different vertical resolution: 16, 32, 64 vertical channels.

The 16 channel alternative presents a smaller cost compared to the other two versions, but this comes with some disadvantages: as will be made clear in the clustering algorithm section, sparsity in the vertical direction can mean that points belonging to the same object are incorrectly classified to different objects and vice-versa. Having only 16 channels means that this kind of errors are more likely to happen and the clustering algorithms should follow a more conservative approach in order to avoid collisions, leading to worse performance.

The 64 channel alternative is the most expensive, but has a great vertical resolution. Data coming from it contain a great amount of detail and errors in clustering are less likely to happen. More data comes at a disadvantage, though: the computational cost of the clustering algorithms grows considerably, and thus the frequency of execution of the whole program is reduced. Since the algorithm will eventually have to be run on an onboard computer with limited resources to be shared among various tasks, this kind of detail can be considered excessive.

The 32 channel choice is the perfect compromise both in terms of economical and computational cost, and the clustering algorithms work well enough with its vertical resolution.



Figure 24- Velodyne HDL 32E

The specifications of the sensor are reported in table 1:

	Specifications:
Sensor:	<ul style="list-style-type: none"> • 32 Channels • Measurement Range: Up to 100 m • Range Accuracy: Up to ± 2 cm (Typical)¹ • Single and Dual Returns (Strongest, Last) • Field of View (Vertical): $+10.67^\circ$ to -30.67° (41.33°) • Angular Resolution (Vertical): 1.33° • Field of View (Horizontal): 360° • Angular Resolution (Horizontal/Azimuth): $0.1^\circ - 0.4^\circ$ • Rotation Rate: 5 Hz – 20 Hz

Table 1 - HDL 32E specifications

Although this is the sensor of choice, it is fair to say that the proposed algorithms have been tested on a number of different datasets belonging to different sensors; this has been an unavoidable necessity as visual data was not available for some HDL 32E datasets used in this work, and visual data is of great importance in understanding if the clustering algorithm is correctly working or not.

3.1.2 Motivations for clustering

Clustering can be defined as “Clustering is a fundamental data mining task that has been extensively studied. In a typical clustering problem the input is a set of points with a notion of similarity between every pair of points, and a parameter k , which specifies the desired number of clusters. The goal is to partition the points into k clusters such that points assigned to the same cluster are similar” [1].

The chosen sensor output is a $32 \times 1024 \times 3$ matrix, for a total of 98304 matrix elements that correspond to 98304 points.

The objective of an autonomous vehicle is to follow the reference path avoiding any obstacles in the way and to do so it has to check, in some manner, that the chosen trajectory for movement doesn't collide with anything. To do so, collision checking needs to be done with all the data given by the lidar sensor, but there are multiple ways to do it. The first is to use brute force and test each and every single point as it comes right out of the sensor; this method thus needs to check for almost 100000 collision at each iteration, so it is not very practical; if points are grouped into clusters, the collision checking is reduced to some tens or at most hundreds of collision checks, thus being more practical.

Another important reason is that clustered data allows for reasonings that are impossible to do on single points. A notable and important example is the orientation of an obstacle vehicle: suppose that our autonomous vehicle finds itself in an environment where other vehicles are moving, for example a road, a construction site or an agricultural field where tractors are working. Obstacle vehicles are most likely equipped with four wheels, two of which have a steering configuration, like Ackermann steering. If the orientation of an obstacle vehicle is known and its velocity is known

aswell (both in terms of direction and magnitude) it can be possible to estimate its future state for some time horizon, for example using a Kalman Filter. Knowing the future state of an obstacle vehicle is essential for the navigation task at medium and high speeds, as trajectories that are collision free in the present time interval may become unfeasible after some time; viceversa, a moving obstacle vehicle can be blocking a trajectory in the present time but may be moving out of it after some instants, making such a trajectory feasible.

These kinds of predictions are only possible some “global” information about all the points that belong to a vehicle is available; thus, it is important to be able to distinguish points that belong to a vehicle and not to something else. Doing so allows to define some object properties like speed, relative position with respect to the ego vehicle, that can be used to create prediction models and to track various actors in the scene.

One last motivation for the importance of clustering and organizing single points into larger groups is the ground segmentation problem: the first step that the navigation algorithm has to perform is to classify points that are not obstacles as belonging to the ground. This can be done in various ways, often using sensor fusion techniques and visual data classified using Neural Networks, but lidar points ground classification still contains useful informations.

3.2 Classic clustering algorithms

In this section an analysis of the most important clustering algorithms will be performed, and motivations regarding the choice of such algorithms in the navigation problem will be discussed.

3.2.1 K-means

K-means is one of the most popular clustering algorithms, and it is one of the most basic aswell [2] ; we will refer to data collected by a single lidar sensor scan as “data points”. This means that each point belongs to R^3 , as its components are either the cartesian coordinates x,y,z , or the raw sensor output range, azimuth and yaw. For the k-means algorithm we will use x,y,z coordinates for each data point.

In this algorithm the average squared Euclidean distance from the data points is minimized with respect to their closest cluster “representative”. The representative point for each cluster is a significant point for that cluster in some metric; for example the median point could be a representative for a cluster, but other choices are possible.

Each data point is called x_i , and it is a vector of dimension $n=3$.

Each representative is called c_j , with $j = 1,...,k$, and it is the representative of cluster C_j . We will be calling them cluster centers.

The squared Euclidean distance from a point x_i to any cluster center c_j is given by:

$$D_{ij} = ||x_i - c_j||_2^2$$

Since each point x_i must be best represented by one and only one center, the distance between a point x_i and its representative cluster center is given by:

$$D_i = \min_j \|x_i - c_j\|_2^2, j = 1..k$$

The objectives of k-means are finding the centers c_j so that the sum of all distances from each point to its representative is minimized and to assign each point x_i to its most representative center point. It is possible to formalize this concept by introducing a cost function to minimize, that is the sum of distances D_i for all N points, and by minimizing it; the problem becomes:

$$J^{clust} = \min_{c_j} \sum_{i=1}^N D_i, c_j = c_1..c_k$$

Substituting we get the standard explicit form of the problem to be solved:

$$J^{clust} = \min_{c_j} \sum_{i=1}^N \min_j \|x_i - c_j\|_2^2, c_j = c_1..c_k, j = 1..k \quad (1)$$

The k-means algorithm works iteratively in two steps:

- Data assignment step: for each x_i , compute the distances from the current centroids c_j , $j = 1..k$ and assign point x_i to the closest centroid. The set of points assigned to centroid c_j constitutes the cluster C_j .
- Centroid update step: given the current clusters created in the data assignment step, compute new centroids minimizing the cost function (1). The minimum for that cost function is simply given by the barycenter of the points of a cluster:

$$c_j = \frac{1}{|C_j|} \sum_{i \in C_j} x_i$$

Where $|C_j|$ indicates the number of points belonging to cluster C_j .

This iterative approach is guaranteed to converge to a result but, since the cost function is nonconvex, the solution may be a local minimum.

The algorithm keeps iterating between the two steps until no change in the cluster assignments is observed.

The starting centroids choice can be performed randomly or using more sophisticated methods, but since the k-means algorithm solves a nonconvex problem, it means that the solution found will depend on the starting point, and that some starting points will perform better than other, depending on the current situation.

There are some observations to be made:

- First, the above stated problem is nonconvex, and this means that the solutions found may be non optimal, as the solvers can end up finding local minimums. In our case scenario this means that the found centers may not be the best representative for each object and that different objects could be assigned to the same center. This is particularly true when talking about long and far from being spherical objects.
- Secondly, the algorithm demands that the number of clusters, k , is known in advance. This is a reasonable assumption in lots of clustering problems (determining if a point representing a patient is ill or sane, for example, is a problem where the number k is known and is equal to two) but is most surely not in a computer vision problem. The number of visible objects at a time may vary extensively, from one or two to tens of objects at the time. This means that it is not possible to pre-choose k ; it is still possible to find a way to use k -means for computer vision though: the k -means algorithm needs to be run multiple times with different values of k , and comparing the results.

One of the metrics that is usually used to compare results across different values of k is the mean distance between data points and their cluster centroid. A large mean distance may mean that far points probably not belonging to the cluster are being assigned to it, and this is an indicator that the number of clusters k needs to increase. However, if we chose $k = N$, the mean distance would be 0, as each point would constitute a cluster by itself. In general, increasing k will always decrease this metric, thus it can't be used by itself. What is done is to consider not the value of k that minimizes this metric, but the so called "elbow point" of the cost vs parameter function. Essentially, it is the point where the rate of decrease of the cost function sharply shifts, as is shown in figure 2.

Elbow Method for selection of optimal "K" clusters

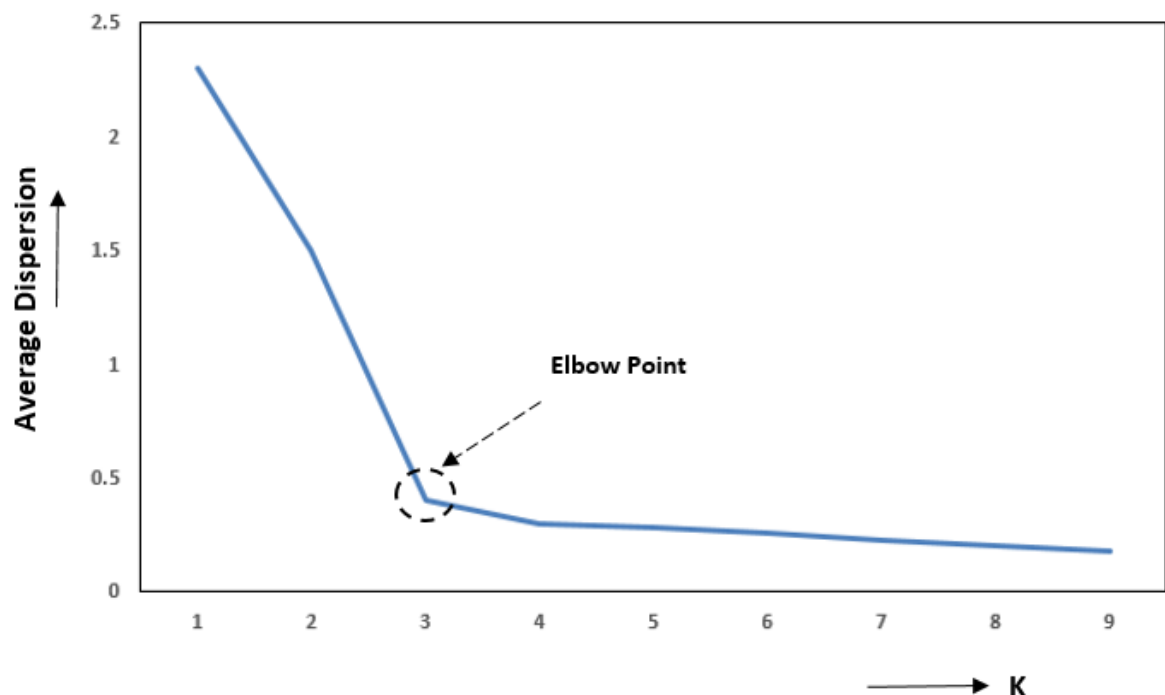


Figure 25- Elbow point for the choice of k

In figure 2 the point where the sharp shift happens is given by $k=3$, which means that the number of clusters in this case is probably going to be 3. Higher values of k don't decrease the average dispersion as much as increasing from 2 to 3.

Another observation to be made is that, as stated in equation (1), k -means can be very sensitive to outliers; this is a feature shared with most optimization problems that involve the l_2 norm, also known as the Euclidean norm. Common ways to counter the sensitivity to outliers is to add a regularization term [3], which can be an l_1 norm term or an l_∞ term. Another possibility is to get rid of the Euclidean norm and to base the whole problem on the l_1 norm, obtaining an algorithm that is referred to as k -medians.

In the following figures 3 and 4 two typical cases of failure in k -means algorithms are shown: in figure 3 two concave objects (original points on the left, in blue and in red) are classified according to k -means (on the right). The centroids are the two crosses on the right: they highlight the fact that the algorithm has tried to find the barycenter of the two halves of the whole dataset by dividing it in half; k -means has thus created two circular shaped clusters, losing the "elongated nature" of the original objects. This can be attributed to the l_2 norm term.

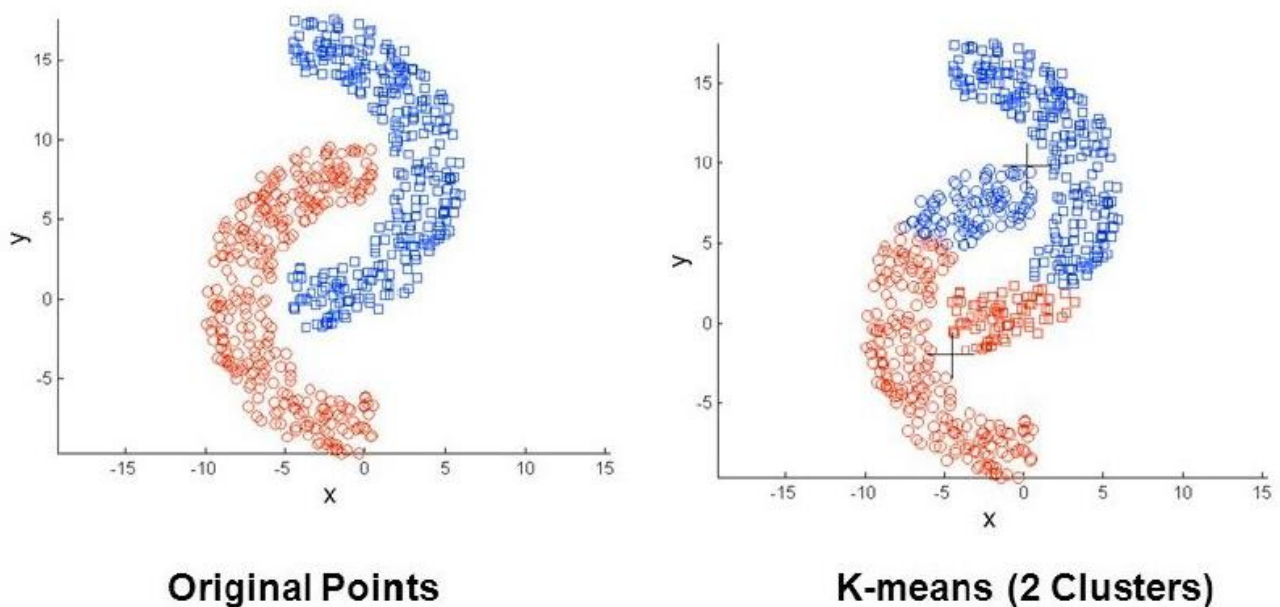


Figure 26 - poor performance in clustering problem; on the left the original points, on the right the clustered points

In figure 4 a similar problem occurs: although the number of cluster is correctly chosen equal to 3 (and this by itself is not an obvious fact to happen) the original data shown on the left are quite sparse: this leads to the two small dense circles being accorporated into one cluster, and the big circle, more sparse in nature, being split in two.

Results like these, and the fact that multiple iterations of k -means need to be run in real time without knowing the top value that should be used as a cap for k has lead to consider other algorithms to be implemented for this autonomous driving application.

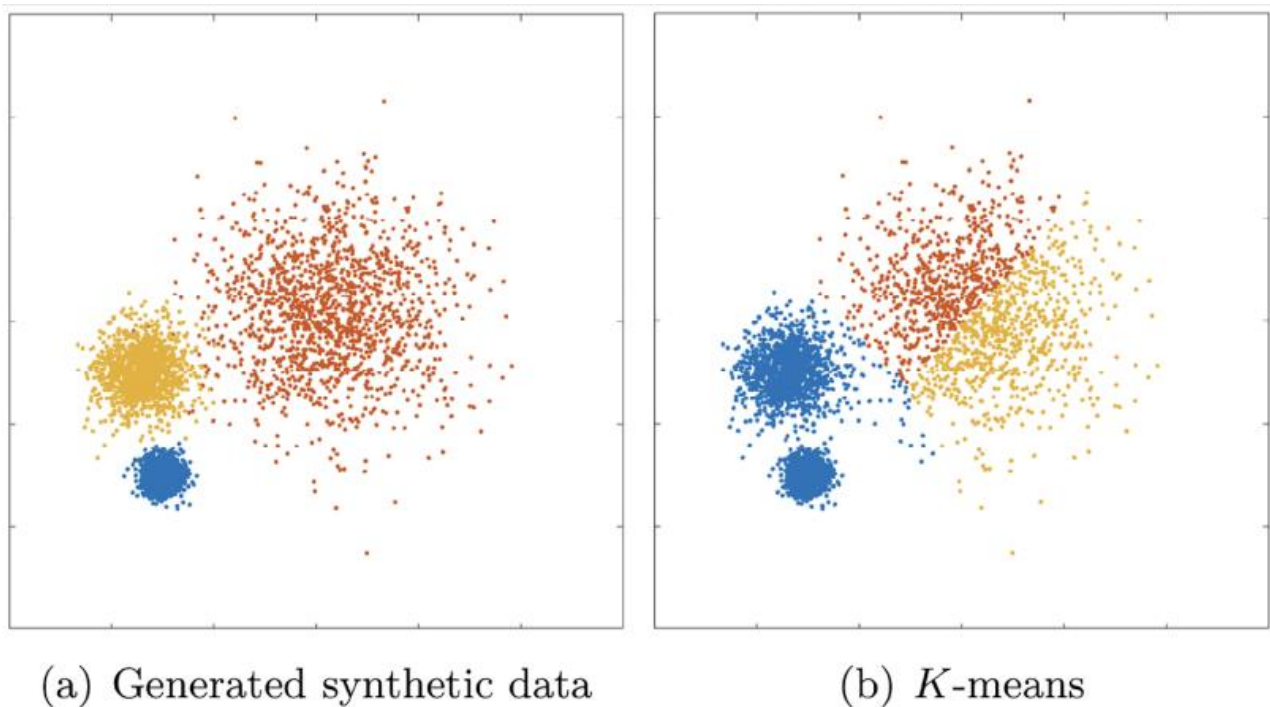


Figure 27 – another k -means clustering fail

3.2.2 DBSCAN

Recapping the problems related to k -means, the two main disadvantages found with it can be stated to be the requisite of a prior number of clusters k , and the fact that k -means works well mainly with spherical-like clusters, due to the L_2 norm being in the cost function.

DBSCAN is an algorithm introduced in [4].

Its acronym stands for “Density-Based Spatial Clustering of Applications with Noise” and its approach, as the name suggests, is based on the intuitive notion that clusters of points present some feature of density, and that points that are classifiable as noise do not present such feature. The algorithm is able to correctly identify not-spherical shapes like elongated shapes, and it does not require to know in advance the number of clusters, fixing the two main problems present in k -means and derivatives. Moreover, it is an efficient enough algorithm, and its light weight computational cost is a relevant aspect in the field of real time clustering.

As stated in [4], DBSCAN can work with any notion of distance between two points p, q , but as far as computer vision for autonomous applications go, Euclidean distance is the usual choice. The space of the point is the 3D Euclidean space but the algorithm can work in higher dimensions. The algorithm requires two parameters to be given as an input besides the set of points to cluster (such set of points will be referred to as D): Eps , which is related to maximum distance between points belonging to the same cluster; $MinPts$, which is related to minimum points between two connected points in a cluster. Both parameters will be better explained in the following definition section.

In order to understand the working principle of DBSCAN, some definitions need to be given:

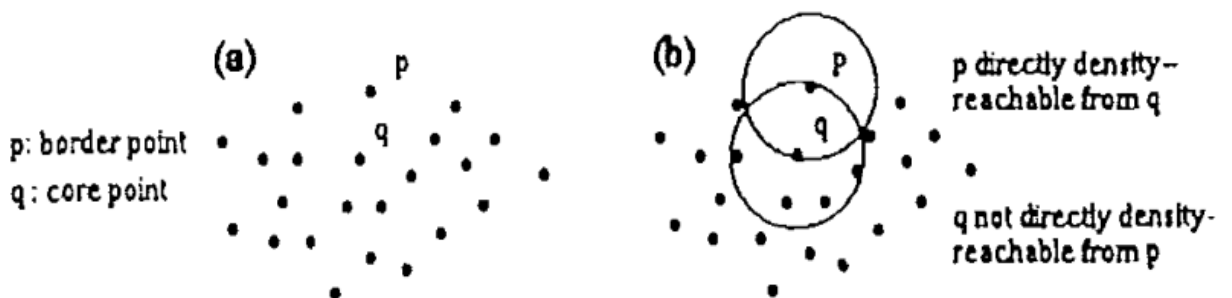
1. Eps-neighborhood of a point p: the Eps-neighborhood of a point p is denoted by $Neps(p)$ and is defined as follows:
 $Neps(p) = \{q \in D \mid dist(p, q) \leq Eps\}$, where dist indicates the Euclidean distance, in our case.

An important observation to make is that a point close to the center of a densely populated cluster (also known as *core points*) will have a number of other points in its Eps-neighborhood much greater than a point belonging to the border of the cluster (also known as *border points*). This is especially true for clusters for point like borders, where the frontier points are mostly surrounded by empty space.

2. Directly density-reachable: a point p is directly density-reachable from a point q with respect to MinPts, Eps when:
 - $p \in Neps(q)$ and
 - the number of points contained in $Neps(q)$ is $\geq MinPts$

this second condition is called “core point condition”. Each couple of points belonging to the core points set will be directly density-reachable for both points, but if one of the points is a border point then reachability will possibly be admitted only for one point.

Figure 5 shows this latter case:



3. Density-reachable: a point p is density-reachable from a point q with respect to MinPts and Eps if there is a chain of points p_1, \dots, p_n , where $p_1 = q$ and $p_n = p$ such that each point p_{i+1} is directly density-reachable from p_i . Again, this notion of reachability is not symmetric for border points but it is symmetric for core points. It arises that two points belonging to the border of a cluster can be not density-reachable. For this reason, the following definition is given.
4. Density-connected: a point p is density-connected to a point q with respect to MinPts and Eps if there is a third point, called o, such that both p and q are density-reachable from o with respect to MinPts and Eps. This definition is such that two border points belonging to the same cluster are density-connected.

Finally, the definition of a cluster can be given:

5. Cluster: the set of points density-connected maximal with respect to density-reachability. In this way, the notion of noise is simply given by the set of points not belonging to any of the clusters found by the algorithm. Formally: a cluster C is a not-empty subset of D satisfying:

- for each p, q : if $p \in C$ and q is density-reachable from p with respect to $MinPts$ and Eps , then $q \in C$.
- for each $p, q \in C$: p is density-connected to q with respect to $MinPts$ and Eps .

It is worth noting that this definition of cluster makes such that the smallest cluster that can be found by DBSCAN is composed by a minimum of $MinPts$.

The algorithm analysis of DBSCAN is now proposed. DBSCAN creates clusters starting from a random unassigned point (all points are classified as unassigned at the beginning of the algorithm and are then assigned when clusters are being created) and once a cluster has been created it moves to the next by choosing another random unassigned point and trying to form a new cluster from it. Once no new clusters are created, all remaining points are classified as noise.

Input: Points, Eps , $MinPts$

Step 1: Choice of a random point; the point is accepted as the beginning of a new cluster if the number of points lying in the Eps -neighborhood of such point are at least $MinPts$, otherwise it is discarded and labelled as noise. In figure 6, the starting point is highlighted in green, the dotted line is a circle of radius Eps and it is assumed that $MinPts$ is less than 7, so to accept this set of points as the beginning of a cluster.

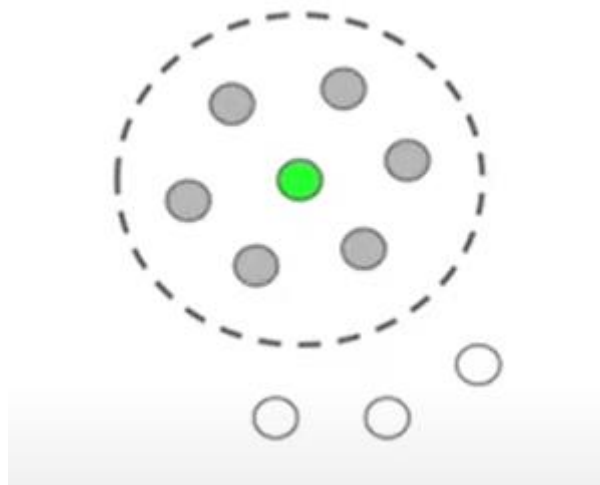


Figure 29 -starting point and Eps-neighborhood

Step 2: For all points belonging to the Eps -neighborhood of the starting point, a try to expand the cluster is made: each point's Eps -neighborhood is checked for the presence of new points. If the

newfound points satisfy the density-connected condition, they are added to the cluster. In figure 7 three points that satisfy this condition have been found and are highlighted in blue.

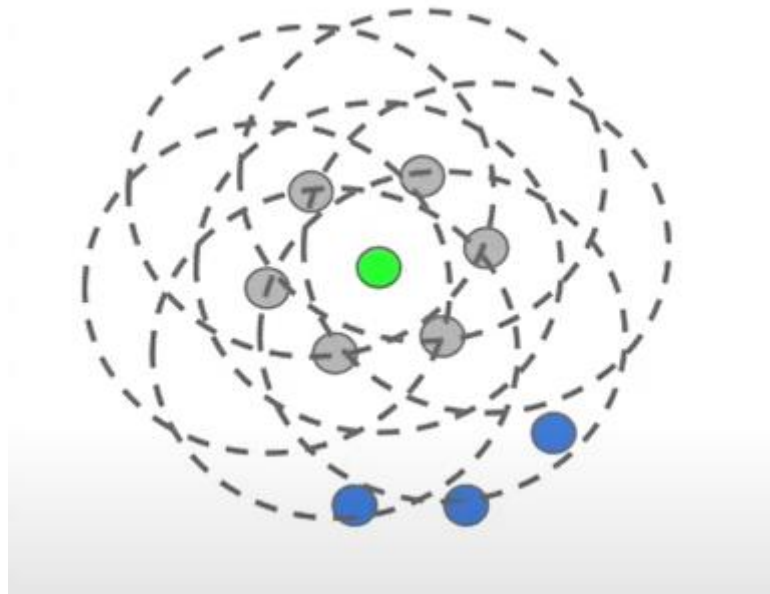


Figure 30 - growing clusters

The resulting cluster is composed of all points shown in figure 7. For each of the newfound points Step 2 is repeated until new points are not found.

Step 3: Once no new point is found, the algorithm goes on to the next random points and starts all over from Step 1. In figure 8 a new point is chosen for a new cluster attempt. Let's suppose that MinPts is chosen to be 5. In this case since the Eps-neighborhood of the new point contains less than 5 points, it is discarded as new cluster and classified as noise.

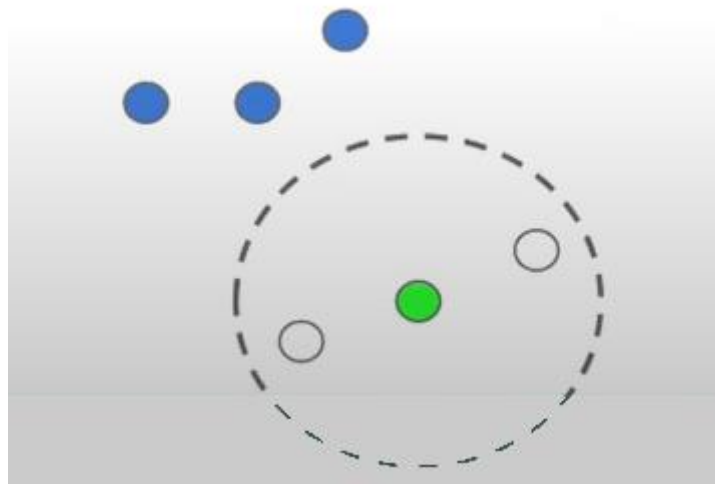


Figure 31 - new cluster attempt, the three points highlighted in blue belong to another cluster

The algorithm goes on until all points have been either classified as clusters or as noise.

It is notable that the parameters Eps and MinPts are not known at the beginning of the clustering phase, but their choice can be done via trial and error. Once chosen, the parameters can be left unchanged, differently with what happens with parameter k in the k-means algorithm. This makes

DBSCAN derived algorithms a great choice for an application such as clustering for automated driving tasks.

In this thesis work, a derived DBSCAN algorithm has been used: it is called pcSegDist, which stands for Point Cloud Segmentation based on euclidean Distance, and it is one of the algorithms contained in some of the Toolboxes for Matlab used in this work.

Using it on data coming from various datasets has shown that it is able of fast segmentation even on limited hardware such as the pc the scripts were run on, and the cluster themselves were correctly identified, once the parameters were correctly tuned; more on this will be discussed later, but an anticipation is needed to justify the next section of this chapter: although the way Matlab memorizes point cloud data and although choosing MinPts not too small so to make the algorithm avoid small clusters made of noise (making it run faster), the frequency of executions per second usually topped at 10 Hz. This by itself is an almost-acceptable performance, compared to some of the commercially available vehicle, which run at around 20Hz [5], but the fact is that clustering isn't the only real time application that needs to be done: path replanning, ground segmentation and possibly control applications such as traction control etc... although in a less-prototipal implementation is it reasonable that dedicated hardware will be used for specific tasks such as machine learning camera applications, a starting point of 10 Hz for the clustering algorithm alone is a bit on the down side.

That is why a new approach based on a recently published paper has been implemented and examined, and will be presented in the next section.

3.3 FLIC (FAST LIDAR IMAGE CLUSTERING)

FLIC (Fast Lidar Image Clustering) is a clustering algorithm specifically designed for Lidar data introduced in the paper [6].

This algorithm has been chosen mainly for its extremely high frame rate, which was declared to be 165 Hz when running the Python implementation on a single CPU i7-6820HQ CPU @ 2.70 GHz core and processing data from a 64 channel Velodyne Lidar. My implementation written in Matlab running on an old laptop with a single i7 core processor still managed to run at 40Hz-50Hz.

The main idea behind FLIC is now presented: the point cloud structure used in Matlab toolboxes is essentially a *width x height x 3* matrix, where *width* and *height* are the number of horizontal and vertical points measured by the lidar, and the three elements are x,y,z coordinates. This actually is not how the Lidar sensor outputs data. In fact, data provided by the sensor contains *width x height x Range* data, and cartesian coordinates are then computed internally by Matlab in order to create the Point Cloud structure: this involves simple calculations, which are nonetheless repeated thousands of times.

FLIC avoids the creation of a three-dimensional point cloud from the range measurements and works directly on the laser range values of the sensor.

The idea is that points measured on single objects have a distance between them significantly inferior to that of points measured on two different objects. The way that data is acquired through a Lidar sensor allows them to be represented in a set of 2D binary matrices, reframing the 3D clustering problem in a 2D connected-component-labelling task.

3.3.1 GROUND REMOVAL

The first step in the work pipeline for objects clustering is **ground removal**, which is the action of removing from the point cloud (or from a 2D representation as in this case) the points that belong to the ground. This by itself is already a non trivial task to do in general conditions, as simply removing points below a certain z-coordinate may fail to correctly remove ground points when the road surface is uneven, but since the Lidar is mounted on top of a vehicle in a way so that its spinning axis is perpendicular to the ground, it makes sense to try and remove points that belong to a plane perpendicular to the z axis.

There are various algorithms made for this specific task, the one that is here used is called **pcfitplane** [7] and is contained in various Matlab toolboxes. It accepts as input data the point cloud and a parameter that is the maximum distance of a point with respect to an inlier to the possible plane, and it gives as an output the equation of such possible plane and all points that belong to it.

Once the points belonging to the ground data are removed, an action that can be done simply by substituting their Range value with the NaN value, the rest of the data can be processed by clustering algorithms.

Why is ground plane removal necessary? It is necessary because two objects lying on the ground can be seen as connected between them by the ground itself, and this would lead to the creation of enormous clusters with little to no sense. In figure 9 a point cloud before (on the left) and after (on the right) ground removal is shown.

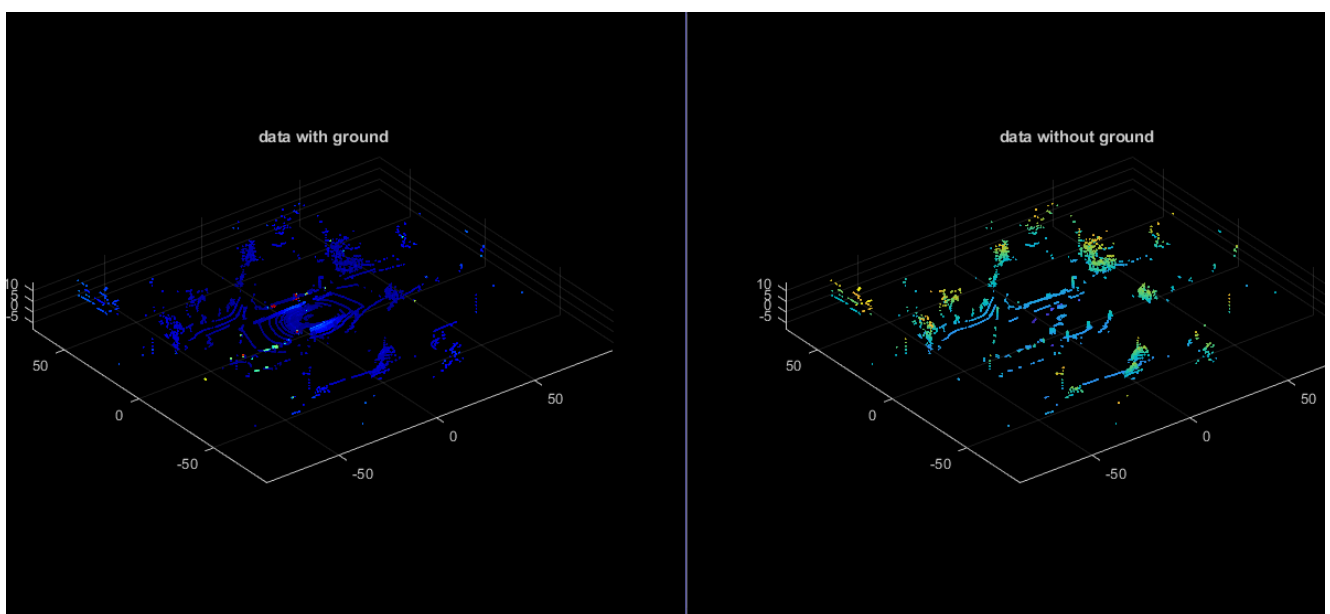


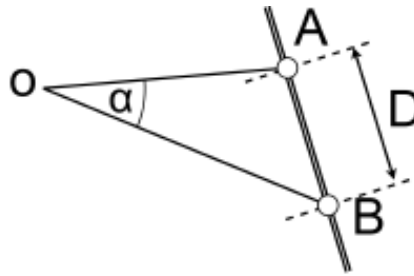
Figure 32 - point cloud data before and after ground removal

It is worth noting that the “concentric points” situated near the origin that span the flat ground around the ego vehicle (which is situated exactly at the origin) are correctly identified as ground and removed. Failing to remove these points would mean considering them as obstacle, therefore making it impossible for the navigation algorithm to find any feasible path to move to. A maximum distance of 0.8m – 1m was seen to be adequate in every dataset that was tried.

3.3.2 BINARY CONNECTIVITY MATRICES

As stated before, the speed of FLIC relies on the fact that it works on 2D connection matrices, so the first step to do is to understand what they are and how to create them.

The distance between two points read by two adjacent Lidar scans can be evaluated as shown in figure 10 (source: [6]).



(b) With the Lidar Sensor in O , the lines OA and OB show two neighbouring distance measurements. The distance between the two measurements is calculated using the spanned angle α between the points.

Figure 33 - trigonometric relations used in the creation of connectivity matrices

The angle alpha is the angle between one sensor cell measurement and the next one; it depends on the sensor, and generally it is different in the vertical span and in the horizontal span. The order of magnitude of such angles is of 0.2° for the horizontal span and of 1° for the vertical span. More considerations on these angles will be done later.

Remembering that the available data from one measurement is composed of the segments OA and OB , the formula used to find D is:

$$D = \sqrt{||OA||^2 + ||OB||^2 - 2 * ||OA|| * ||OB|| * \cos\alpha} \quad (1)$$

If we square both sides of (1) and indicate with $d1=||OA||$ and $d2=||OB||$ we can write the equation used in the code implementation:

$$D^2 = d1 * d1 + d2 * d2 - 2 * \cos\alpha * d1 * d2 \quad (2)$$

The equation (2) is worth of some comments:

- d1 is the range value, in meters, contained in one of the 1024*32 “measurements cells” provided by each lidar scan and d2 is one of the measurements contained in an adjacent cell, an horizontal one or a vertical one.
- Once a value for D has been chosen, and D^2 can be pre-computed offline, if a couple (d1,d2) produces a result on the right side greater than D^2 then the two points are considered not belonging to the same object and therefore are not considered connected. If a couple (d1,d2) produces a result that is less or equal than D^2 then the two points are considered to belong to the same object and are therefore considered connected.
- What said above means that the equation (2) is evaluated for all adjacent couples in the 1024*32 range image. Its simplicity allows for this many evaluation as: the computation of the right side only requires 4 multiplications (as the computation of $2*\cos\alpha$ can be pre-computed for speed and therefore doesn’t count as a fifth multiplication) and 2 additions. This not only makes the core condition much simple but it also means that a future possible implementation on an hardware dedicated to make those computations really efficiently is possible.
- Instead of evaluating (1), the evaluation of (2) allows for the avoidance of computing the square root, which is a costly operation to perform.

For the sake of clarity, in figure 11 an example of the range matrix is shown:

Range ✕							
32x1083 double							
	771	772	773	774	775	776	
23	NaN	NaN	NaN	NaN	NaN	NaN	
24	NaN	NaN	NaN	NaN	NaN	NaN	
25	NaN	NaN	NaN	NaN	NaN	NaN	
26	NaN	NaN	NaN	NaN	NaN	NaN	
27	NaN	NaN	NaN	NaN	NaN	NaN	
28	NaN	NaN	NaN	NaN	NaN	NaN	
29	1.7420	1.7340	1.7340	1.7160	1.7260	1.7220	
30	1.7000	1.5800	1.5900	1.5960	1.6080	1.5940	
31	1.4680	1.4620	1.4660	1.4640	1.4700	1.4900	

Figure 34- range matrix

The NaN values indicate cells where the Lidar sensor didn’t measure anything (missing return laser); the bottom lines present an “island” of values close to each other: running the (2) equation on these cells will confirm that they are close enough, thus connected.

It is now necessary to discuss how the range image will be converted to a binary connection matrix (indicated with BCM from now on). The approach used consists in the creation of three support matrices that will be created respectively for the horizontal connectivity, the vertical connectivity and a third matrix that will simply display if a cell contains a valid number or a NaN value.

Their sizes varies, and will be discussed later, but it is worth to already say, naming m and n the rows and the columns of the original range image, that the final BCM will be a $(2m-1)*(2n-1)$ matrix.

In order to create the horizontal connectivity matrix, called H from now on, the following algorithm has been run on the whole range matrix, called R from now on, as shown in the piece of code contained in figure 12.

```
H = zeros(m,n-1);
for j = 1:m
    for i = 1:n-1
        if R(j,i)*R(j,i) + R(j,i+1)*R(j,i+1) - ch * R(j,i) * R(j,i+1) < Dh
            H(j,i) = 1;
        end
    end
end
```

Figure 35 – Matlab implementaiton of horizontal connectivity matrix

As is visible, the central if statement contains the equation (2), where ch and Dh have been pre computed.

The result of this piece of code is a matrix H which contains 1 for points in R close to each other and 0 for points in R far away from each other, using (2) as threshold.

The same thing is done for the vertical connectivity matrix, as shown in figure 13:

```
V = zeros(m-1,n);
for j = 1:m-1
    for i = 1:n
        if R(j,i)*R(j,i) + R(j+1,i)*R(j+1,i) - cv * R(j,i) * R(j+1,i) < Dv
            V(j,i) = 1;
        end
    end
end
```

Figure 36- Matlab implementation of vertical connectivity matrix

The reasoning is the same behind the horizontal connectivity matrix, and the result is a matrix V .

For the third matrix, which is the one that contains the information on whether a cell in R is a valid measurement or not, the algorithm simply tests if the value contained in that cell is a valid number or the value NaN. This third matrix is referred to as RB , which is short for range binary.

The next step consists of stacking the three matrices together, in order to preserve the connectivity information, in a way that allows us to then run a 2D connected component algorithm (more on this later). In order to do this, the H , B and RB matrices need to be extended and brought to the final dimension of $(2m-1)*(2n-1)$. This “extension” process essentially consists of filling with zeros certain rows and/or columns of the matrices.

The horizontal connectivity matrix H is a $(m,n-1)$ matrix. In order to extend it, a column of zeros is added between each original column, with the first zero column added before the first original

column. Rows of zeros are added aswell. Matlab indexing makes it quite easy to create this kind of “sparse” matrices, as shown in figure 14:

```
|
HE = zeros(2*m-1,2*n-1);
HE(1:2:end,2:2:end) = H;
```

Figure 37 - creation of horizontal extended matrix

The same process is repeated, using different indexation, for the matrices V (creating matrix VE) and matrix RB (creating matrix RBE). The result is shown, using as example the process from H to HE, in figure 15. The added zero-rows and coloumns are highlighted in red.

1	0	1	0	1	0	0	0	1
1	1	0	0	0	0	0	0	0
0	1	0	0	1	0	1	0	0
			0	0	0	0	0	0
			0	0	0	1	0	0

Figure 38 - on the left, the original H matrix. On the right, the extened HE matrix

After the three extended matrices HE,VE, RB are obtained, they are stacked one on another, simply summing them. The final result is the BCM, the binary connection matrix, which is a $(2*m-1,2*n-1)$ matrix; this matrix contains the clusters of the original range image in the form of “isles” of number ones. A cluster of points close enough to each other will in fact produce H and V matrices with ones in the position of connection points. What is now to do is to recognize these isles of ones and label them as different clusters.

3.3.3 RECOGNIZING CLUSTERS IN BINARY CONNECTION IMAGE

The previously created BCM contains connection between horizontal and vertical points, but not diagonal points. Therefore, the numbers between points in the diagonal sense are of no pratical interest from the point of view of clustering.

In order to find the “isles of ones” in the BCM, a connected-component Matlab algorithm has been used, called **bwconncomp** [8]. This algorithm comes from black and white image processing, and therefore is applicable to the binary connection matrix that we produced. In fact, the BCM can be interpreted as a black and white image, as it is only composed of zeros and ones.



Figure 39- detail of BCM containing a road sign

By default `bwconncomp` searches for 2D clusters with a connectivity of 8, but in our case the useful connectivity is 4, since diagonal vicinity is not accounted for. Figure 17 shows the difference in connectivities (source: Mathworks)


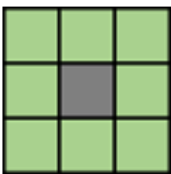
Value	Meaning	
Two-Dimensional Connectivities		
4	Pixels are connected if their edges touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal or vertical direction.	 <p>Current pixel is shown in gray.</p>
8	Pixels are connected if their edges or corners touch. Two adjoining pixels are part of the same object if they are both on and are connected along the horizontal, vertical, or diagonal direction.	 <p>Current pixel is shown in gray.</p>

Figure 40- 4 and 8 connectivities. Source: Mathworks

Running `bwconncomp` on the BCM returns an object containing various informations: the used connectivity, the number of clusters found, and the clusters themselves, indicated as lists of pixel. Pixels can be indexed as a single progressive number, counting from the top left pixel onward. This kind of indexing is not ideal for row and coloumn manipulation, as rows and columns are not explicitly indicated. To pass to the row, coloumn notation a Matlab function called `ind2sub` has been used: the pixels belonging to each cluster are now stored in row, coloumn format. There is a catch though: these are the clusters present in the BCM image, but what we need are the clusters contained in the original Range matrix `R`. A downsampling process is thus needed, in order to get back to the (m,n) matrix.

This isn't simply a dimension issue, the pixels in the BCM that connect two points in the range matrix don't have any physical meaning, and are only a mean to use `bwconncomp`, and thus do not constitute valid elements of the clusters.

The downsampling is performed for each cluster, only taking the rows and columns that corresponded to actual points in the original range matrix. The final result is an image size (m,n) whose pixels are labelled as clusters.

After that, the pixels, that are now in the sub format are re-converted in linear indexing format; this is done for the purpose of creating a coloured image to overlay to the original range image in order to see if the algorithm works well or not. For display purposes, each cluster is given a random color. The results of clustering on a lidar reading are shown in the next section.

3.3.4 FLIC RESULTS AND COMMENTS

The data used for testing this algorithm comes from a recording of 10 seconds of travel in a construction road environment, and was registered on a Velodyne HDL32E sensor. This sensor has got 32 vertical channels and 1083 horizontal channels. The raw data coming from the sensor is thus a 32x1083 matrix containing numeric values in meters, and it is the so called Range image.

A single frame of the data is displayed in figure 18: each pixel has been assigned a greyscale colour based on its value, closest points are depicted as white while far points are depicted as black, using a discretization of 255 values. The 255th value, complete black, is reserved for NaN points. Since the image is large and narrow, only a portion of it is displayed, in order to increase visibility.



Figure 41 - portion of Range image

FLIC is run on figure 18 and produces the result shown in figure 19: there are various aspects to analyze:

- The big brown cluster is a wall like structure that starts in the immediate vicinity of the ego vehicle and extends itself in the distance. FLIC correctly recognizes the whole cluster as one entity.
- Complex shapes are recognized as well: on the left, in white, a road sign with a nonconvex shape is correctly recognized as one object. Also, the pole that connects it to the ground is recognized as well.
- Foliage and trees constitute a problem: due to their sparse nature, well visible in the center part of both figures 18 and 19, the connectivity of the whole tree is lost; small clusters are created instead, but their physical meaning is irrelevant. This phenomenon is due to the sparse nature of foliage, and will be tackled in the next section.
- There are some oversegmenting issues in the big brown cluster. Particularly on the right, it is possible to see that small horizontal clusters are being created inside the brown one, and this seems to be due to some missed data recordings, as visible looking at figure 18. Nonetheless, they do constitute a problem.
- Since the ground has been removed before the creation of clusters, points belonging to ground and to the sky have the same NaN colour to them, indicating the absence of an obstacle.





Figure 42 - same portion of image clustered by FLIC – in the red circle, an oversegmented cluster

A reasonable hypothesis for this behaviour is that excessive sparsity of data leads to bad clustering performance. This phenomenon is due to either missing measurements because the laser was deviated by reflective surfaces or sparsity in the vertical direction. In fact, the vertical direction only presents 32 channels, and especially at long distances this means that each object is only hit by a few laser beams.

Another reason, as already said looking at figures 18 and 19 can be the sparsity of the object itself: trees, bushes and other kinds of foliages are by their nature sparse.

It is worth remembering that the implemented version of FLIC used, for the acknowledgement of clusters, a connection component algorithm with connectivity 4. This was chosen because the condition expressed by equation (2) is only verified for horizontal and vertical neighbors, and not for diagonal ones.

In the paper [6] these kinds of unsatisfactory results are noted and analyzed as well. A solution found by the authors is to extend the notion of neighbor to more external points than to just the four points directly connected.

3.3.5 MAP CONNECTIONS AND MINIMUM THRESHOLD

The concept of extending the checked connectivity to points more far away is called in [6] Map Connections (MC). The shape of the connectivity 4 is preserved as the neighbor points that are checked are only the points above, below and on the left and right, but the checked points are 1 or 3 points away from the first neighbor.

The code remains the same up until the creation of the BCM, as not much can be done for missing data or sparsity.

After that a second BCM is created: the difference with the first BCM is that data are checked for (2) not with their direct neighbor but with the 3rd point. This second BCM matrix is created by changing the index of the tested points, as shown as example in figure 20:

```

%extended horizontal connectivity matrix
for i = 1:4:2*m-1
    for j = 1:4:(2*n-1)-4
        if ESR(i,j)*ESR(i,j) + ESR(i,j+4)*ESR(i,j+4) - ch2 * ESR(i,j) * ESR(i,j+4) < Dh2
            EH(i,j+1:j+3) = 1;
        end
    end
end
end

```

Figure 43 - extended horizontal connectivity matrix for the 3rd neighbor matrix

Since the checked points are not the direct neighbors, the values of the cosine $ch2$ between the points is changed; the distance $DH2$ between the points is changed aswell.

The application of this algorithm leads to increased robustness and larger clusters in general.

The new obtained 3 neighbors matrix is then overlayed to the original BCM, using an OR operator.

The result is shown in figure 21:



Figure 44- FLIC with 1MC

Some of the oversegmenting issues have been solved. In particular, the green cluster in the center (highlighted with a red dotted circle in figure 21) of the image is correctly identified as a single cluster, while it was oversegmented in two different clusters in the basic version of the algorithm, as visible in figure 19. It is also possible to notice a better result in the right bottom side of the image, where the missing data present in the big light green cluster have not constituted the creation of small clusters, as opposed to what has happened with the basic version of FLIC.

The quality of clusters can be therefore said to have been increased by the adding of one Map Connection. But, it can be noted that in the segmented image in figure 21 there are still lots of small cluster, especially regarding points that are far away. This is due to the sparsity of the sensor, and is especially notable in far away objects. Although there is nothing wrong with those sparse measurements, they are not of big interest for the navigation algorithm, as they can vary quite abruptly and a planning algorithm that keeps track of far away small objects may become excessively slow. Therefore, a minimum threshold has been introduced to filter away not significant clusters. It is necessary to note that, if the threshold is correctly chosen, close objects will not be affected by this measure, as the more an object is close to the lidar, the more the number of detections and therefore points. A too large threshold can make the algorithm run very fast, but will miss small objects even at a medium to close distance to the sensor.

A threshold of minimum 10-20 points per clusters has been seen to be working well with the construction road dataset; in figure 22 an implementation with 1 MC and a threshold of 15 points is reported. The blue dotted circle in figures 21 and 22 highlights the improvement in the clustering performance obtained by introducing the threshold.



Figure 45 - FLIC with 1MC and 15 points as threshold; the absence of small clusters is highlighted in the blue circle

The clustering performance with these two improvements has improved considerably. The frequency at which the algorithm runs varies between 10 Hz and 20 Hz, so it is actually a little better with respect to the DBSCAN algorithm, but the difference is not as big as indicated in [6]; this may be because of the lack of software optimization in my Matlab implementation.

Another notable problem that has been found is that the horizontal angle between two adjacent measurements was not clearly indicated in the dataset of the HDL32E sensor; this is probably due to the fact that usually the knowledge of this angle is not of interest of the users of the sensor, as the majority of clustering algorithms use as a representation the Point Cloud data structure. An estimate based on similar sensor has identified the angle to belong to the $0.1^\circ - 0.4^\circ$ range, but of course exact knowledge should be required for the best clustering performance. In order to be able to use various Matlab toolboxes in the following chapters of this work, the chosen algorithm has been DBSCAN, used in the native Matlab implementation in the function **pcsegdist**.

CHAPTER 3 - BIBLIOGRAPHY:

- [1] : Shalmoli Gupta et. al., Local Search Methods for kMeans with Outliers, 2017
- [2] : Calafiore G., Notes about clustering in Optimization for Machine Learning course, 2021
- [3] : Calafiore G., Regularized Regression and the LASSO, 2021
- [4] : Martin Ester, Hans-Peter Kriegel, Jiirg Sander, Xiaowei Xu, A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, 1996
- [5] : thesmanian.com,"Tesla Filed Patent 'Machine learning models operating at different frequencies for autonomous vehicles'", 2020
- [6] : Frederik Hasecke, Lukas Hahn, Anton Kummert, FLIC: Fast Lidar Image Clustering, 2020
- [7] : <https://it.mathworks.com/help/vision/ref/pcfitplane.html>, 2015
- [8] : <https://it.mathworks.com/help/images/ref/bwconncomp.html>

CHAPTER 4 – OBSTACLE REPRESENTATION

4.1 FROM CLUSTERS TO CUBOIDAL MODELS

Once the points perceived by the Lidar sensor have been successfully grouped together in clusters, as explained in the previous chapter, it is necessary to extract some information from these aggregated points. As a reminder, clustering was a necessity because path planning algorithms can not deal with hundreds of thousands of points, which is the raw data outputted by the Lidar sensor, but what has been done up until this point was simply giving each individual point a label indicating which point aggregation it belonged to. The next step is to extract useful information from the clusters, and this information will be the input for the navigation algorithm. In this way, from this chapter onwards, we will no longer need to keep track of each individual point that surrounds the telescopic handler.

First, let's define what the clusters we are dealing with are. The Matlab function used to create cluster was **pcsegdist**. This function gives as output an array with size $(m \times n, 1)$, in which each element is a label number, and a label is given to each point of the point cloud. The points with label "0" are the ground points or points for which there is no measure available, such as the sky, or unclustered points. Every other point belonging to a cluster is given as label a number > 0 .

So, after using **pcsegdist**, we need to loop through every cluster; the number of clusters is simply the biggest label number. For each cluster, we therefore have the list of indices of each point. We can select the points belonging to a cluster from the original point cloud by using Matlab indexing; the result are N_c point clouds, where N_c represents the number of clusters. Once we have the clusters point clouds, we can proceed in finding useful information to represent them: for example, we might try to find a 3D shape that encapsules all the points. Although it could be possible to try and find the smallest 3D shape that fits the points (kind of trying to find an enclosing mesh) the way that was chosen is to try and fit cuboids to the cluster point clouds.

Cuboids are parallelograms with all angles equal to 90° . In the literature, there are many different proposed algorithms to achieve such a feat. A popular tool that has been used for example in [1] is Principal Component Analysis, or PCA. PCA is an algorithm commonly used in unsupervised learning problems, where the searched feature is the organization of data points. PCA essentially tries to find the direction of maximum variance, which is the direction that, if taken alone, can be used to best represent the data. PCA has been used extensively for finding patterns, especially in high dimensional problems, where its repeated application allows to find a subset of dimension with which to express data. Since the points contained in a point cloud are three dimensional points, a simpler method has been used.

Before talking about the used algorithm, though, it is important to clarify one point: in this thesis work the objective of the navigation algorithm will be to navigate the telescopic handler on a flat surface, with its arm fixed. The topology of the environment in the z-axis, therefore, is not of interest; an obstacle located above the ground will be considered the same as an obstacle located on the ground, the case where the telescopic handler passes below a bridge-like obstacle is not considered. To translate this concept in our algorithm, every point will be projected on the 2D

plane situated at $z=0$, and navigation planning will be carried out in 2D. This 2D plane projection will be the set of point on which the cuboid-fitting algorithm will work on. After having found a fitting rectangle for the base, the height of the cuboid will be given by the largest z -value contained in the cluster. The algorithm used to fit the 2D projection of point cloud into rectangles is the algorithm proposed in [2]. This algorithm does not use PCA: since the dimension of the problem is 2, the authors have used a simpler approach, but a reminiscence of the core idea of PCA is kept. In fact, the algorithm proposed in [2] iterates through all the possible directions of the rectangle; at each iteration, it is easy to find a rectangle oriented in that direction and that contains all points in the 2D projection of the cluster. Each time a candidate rectangle is created, the distance of all the points from the four sides of the rectangle is computed. These computed distances can be used to separate the points into two sets, namely P and Q .

These two sets of points are then used to solve the following optimization problem:

$$\begin{aligned} \underset{c1, c2, \vartheta}{\text{minimize}} \quad & \sum_{i \in P} (x_i * \cos\vartheta + y_i * \sin\vartheta - c1)^2 + \sum_{i \in Q} (-x_i * \sin\vartheta + y_i * \cos\vartheta - c2)^2 \\ \text{subject to:} \quad & P \cup Q = \{1, 2, \dots, m\} \\ & c1, c2 \in R \\ & 0^\circ < \theta < 90^\circ \end{aligned} \quad (1)$$

Where $c1$, $c2$ and θ are the parameters of the perpendicular lines that compose the two internal sides of the rectangle (in this context internal refers to the direction that unites the cluster with the lidar sensor, situated at the origin).

For each iteration, the squared error constituted by equation (1) is computed; we look for the direction (so, the sets P and Q) that minimize such an error. That direction and therefore those values of $c1, c2$, θ will constitute the parameters of the best fitting rectangle. In figure 1 the result of such algorithm on a random lidar measurement (source: [2]).

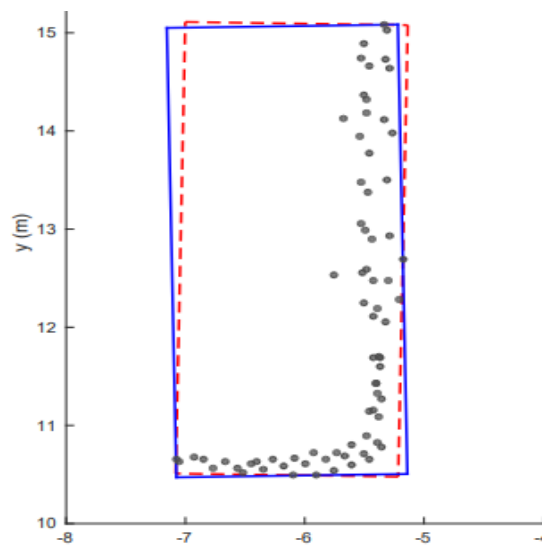


Figure 46-rectangle L shape fitting

The algorithm proposed in [2] is contained natively in the Lidar Toolbox, under the function **pcfitcuboid**. This function receives in input a Point Cloud object and gives as output a cuboidModel object. This object class can be created by specifying nine parameters in the form of a 9x1 vector. These parameters are:

- $x_{ctr}, y_{ctr}, z_{ctr}$, which these specify the center of the cuboid.
- $x_{len}, y_{len}, z_{len}$, which these parameter specify the length of the cuboid axes.
- $x_{rot}, y_{rot}, z_{rot}$, which specify the rotation angles respectively in the x,y and z axis. They are positive when the angle is clockwise. In figure 2 the cubid parameters are shown.

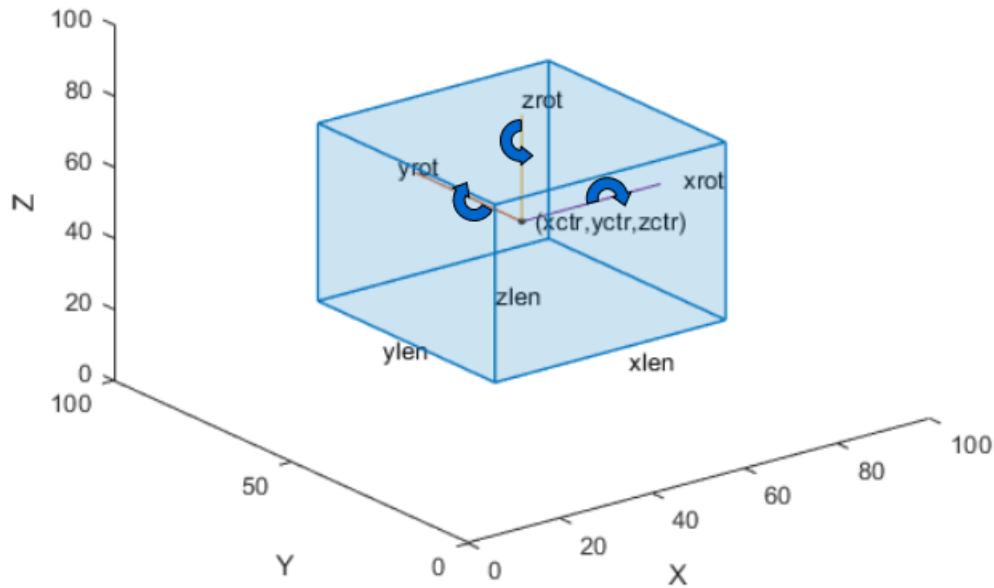


Figure 47 - cuboid model parameters

As explained before, the base of the cuboid is found by projection on the xy plane and subsequent l-shaped fitting and the height of the cuboid is put equal to the highest point in the cluster. But, since the navigation algorithm actually only works in 2D (that means it only works using the base of the cuboid), the height data is actually discarded. The choice of using cuboids and not mere rectangles thus is not essential for the navigation algorithm, and could be replaced by a 2D visualization. It has been kept mainly for the reason that a 3D visualization of the point cloud, its clusters, and the cuboids associated with them is very useful in the testing phase, as 3D data are easier to see and interpret. This allows for better understanding of the algorithms, although is not used by the navigation algorithm itself.

That said, it is a feature that would be implemented in a follow-up work, specifically in the algorithms dedicated to catching and holding the objects to be moved with the telescopic arm. Infact, even if 2D navigation is sufficient for the objectives that autonomous driving for a telescopic handler is aimed at, the movimentation aspect needs 3D representation. The idea would be to either use cuboids or cylinders as enclosure shapes for the target object, but the solution implemented in pcfitcuboid (to project to the base and give as height the highest point) may not be ideal, as its core concepts is more oriented to navigation.

4.2 OCCUPANCY MAP REPRESENTATION

Once the clusters have been enclosed into cuboids, the single point clouds are discarded. This is to speed up computations and because every information contained in the point clouds has already been passed onto the cuboids. The cuboid representation on its own, however, is not yet usable for the navigation algorithms, and needs to be translated into another representation. There are mainly two kinds of obstacle representation, and the first one are occupancy maps, or occupancy grids.

Occupancy grids were first proposed by H. Moravec and A. Elfes in 1985 [3], and represent a 2D slice of the 3D obstacles map; in this case, the 2D slice of interest is the base of the cuboids, as it already contains all the cluster points projected onto it. An occupancy grid is composed of a number of small cells, each one of them containing a binary random variable that represents the presence or absence of an obstacle. Since it is a 2D representation, it is correct to think at it as a m by n , where the dimensions depend on the width and length of the environment the robot is navigating into, and on the chosen resolution. The resolution is given in $cells/m$, and a finer resolution means a denser map. As said, the values contained into the cells are binary random variables: in this context a value of “0” is considered free space, a value of “1” is considered occupied space, but this alone does not take into account the unavoidable uncertainty linked to occupancy measures. For this reason, each cell is given a probability of occupancy, and in order to be able to report this to a single number, some estimator (such as mean estimator) is used in order to bring down the probability density function of a single cell into a scalar number between 0 and 1. The result is that the occupancy grid assumes the appearance of a grayscale image. Occupancy grid representation mainly relies on three assumptions:

- a cell is either occupied or free
- each cell is independent from the others
- the world is static, so the obstacle don't change

The first assumption seems in contrast with the probabilistic nature of the cells, as we have said that cells can assume values in between 0 and 1. In order to solve this contrast and to be able to give to an algorithm a way to easily make decisions in the grid, a threshold value is defined, called occupied threshold [4]. Cell values above the occupied threshold are considered as fully occupied, while cell values below the occupied threshold are considered fully free. This allows for tunability of the algorithms for different applications and risk scenarios.

The third assumption is about staticity of the world: once a map has been created, it is assumed that the world does not change, and that the path planning algorithm can work within it. This is a valid assumption for controlled environments, but is not valid for the applications studied in this work, where the presence of new vehicles and obstacles can't be ignored. In order to take into account the varying nature of the environment, the map itself must be updated when new data comes in. In our case, at this point of the work, such data is represented by the bases of the cuboids. The bases of the cuboids are the bases of the shapes that encapsule the obstacles, therefore they represent points in the map that are fully occupied, and can be represented by the value 1. It is worth remembering the assumption made and data available up to this point. Firstly,

it is assumed that (Chapter 2) the position of the telescopic handler is known with sufficiently high precision due to the use of RTK geolocalization, which allows precision up to the centimeter; secondly, it is assumed that a priori map is given: as said in Chapter 1 the a priori knowledge of the working environment is assumed to be known, for example using satellite data or drones readings. This a priori map, in order to be used for the offline path planning algorithms explained in Chapter 1, was already available in the form of an occupancy grid. Therefore, when the navigation starts (ie: the telescopic handler starts moving and following the offline-computed reference path) an occupancy grid is already known and present, and the handler position inside of it is known. Lastly, at each lidar scan and subsequent clustering and cuboid fitting, a number of occupied rectangles are given, referenced with respect to the lidar sensor.

A naïve approach would be to divide the rectangles into occupancy matrices, using the same resolution used in the a priori occupancy grid, and then overlay the occupied rectangles to the a priori grid, substituting the previously free cells with completely occupied cells (containing a 1). Although this is the easiest way to include the obstacles in the map, it presents some difficulties: firstly, the lidar sensor works at around 10 Hz, and it presents noise; this means that clusters and consequently occupation rectangles can vary a lot at a relatively high frequency, causing flickering in the cell values of the occupation grid; moreover, if the only criterion is to substitute cells values with 1 when an object is detected, a possible unwanted result when high noise is present is to saturate the occupancy grid with ones, creating a number of imaginary obstacles.

The solution to this flickering problem is to use some sort of filter; filtering is a widely used tool and many algorithms are available, such as Kalman filters. But, since each cell only contains a binary random variable, a particularly well fitted tool is Bernoulli Bayesian Estimators. Bernoulli Bayes allows to compute the posterior probability density of a binary random variable, using both the priori knowledge (in this case, past measurement or versions of the occupancy map) and update data (in this case, lidar data in the form of occupancy rectangles). This allows for various enhancements, as the weight for the prior and new data can be adjusted, by increasing and decreasing the variances of the prior distribution of each cell and of the lidar data distribution; for example, in the case of challenging weather conditions, the incoming lidar data can be given a higher variance, as it is possible that noise affects the sensor. The formula for computing posterior distribution is given in (3):

$$p(\theta|D) = \frac{p(D|\theta) * p(\theta)}{p(D)} \quad (3)$$

Where θ is the random variable at study, in our case the scalar value in a single cell, D is the data contained in a cell of the lidar reading occupation rectangle, and $p(\cdot)$ indicates the probability density function. Using this kind of estimator allows to solve the flickering problem and to update the occupation map using a probabilistic approach. Although occupancy maps are a widespread tool used for robotic navigation, the method used in this work is different.

4.3 FROM CUBOIDAL MODELS TO OBJECT LIST REPRESENTATION

Using occupancy map approach for the offline path planning procedure, which as explained in Chapter 1 is done before the telescopic handler starts moving, is an appropriate way to use the a priori information of a bird eye view map; but using occupancy map approach for a dynamic environment presents some issues. A part from the flickering effects presented in the previous paragraph, the big issue resides in the navigation algorithms that needs to be used in combination with the occupancy map. Such algorithms, like the presented RRT*, present a non marginable computational cost, and updating the starting occupancy map whenever encountering an obstacle means to re-run the navigation algorithm at every lidar scan; even if downsampling is considered in the lidar scans (re-running RRT* once every 2, 5 , 10 lidar scans) the time used by the navigation algorithm is still too much for being run in real time, at least for test performed on a reasonably powerful machine (dual core i7). The pure occupancy grid strategy, although simple, is computationally too heavy, especially for large environments. Therefore, the used method is now presented.

The a priori occupancy grid is searched offline, and used in order to create a *reference path* for the telescopic handler to follow (more on this in Chapter 5). The algorithm used in this offline phase does not need to be extremely efficient, as this part is computed offline (and may even be run on server computers not localized where the physical machine is). This *reference path* is an obstacle-free path, which of course is obstacle free only when considering the a priori obstacles; but even if vehicles or other obstacles can find themselves between the reference path, it is a reasonable assumption to make that it will mostly be obstacle free. This means, once the machine has gotten around an obstacle and returned to the reference path, it can continue the rest of the navigation as was originally planned (offline). This assumption makes the use of dynamic occupancy maps an overshoot: with dynamic maps the machine could, in theory, find itself in a totally different environment at each lidar scan and be able to plan a new path from zero, as RRT* is run at each scan. The mostly free reference path assumption therefore doesn't need a whole map update, but simply needs a way to avoid a finite and reasonably small number of obstacles that are on the path. The found solution is to store the obstacles into a list, called an obstacle list. This list must be some kind of data structure that allows for easy use and consultation of all of the obstacle characteristics, such as distance from the handler, height, width, length, orientation; each element of this list (each obstacle) will need to be tested for collision with the ego vehicle (it is the name given to the telescopic handler). If a collision will be detected a new path will have to be planned, otherwise the ego vehicle will continue on its default path, but the method to detect collisions will not influence the occupancy grid, that will not be updated once the vehicle starts moving. A data structure used for these kinds of purposes is present in Matlab in the Navigation Toolbox under the name `dynamicCapsuleList` [5].

4.4.1 Structure of dynamicCapsuleList

The dynamicCapsuleList object manages two lists of capsule-based collision objects in 2-D space. Collision objects are separated into two lists, ego bodies and obstacles. In our case the ego body is a rectangle representing the 2D enclosure of the base of the telescopic handler and the obstacles are the rectangles being the base of the cuboids found using pcfitcuboid. Each collision object in the two lists has three key elements:

- **ID** — Integer that identifies each object, stored in the EgoIDs property for ego bodies and the ObstacleIDs property for obstacles.
- **States** — Location and orientation of the object as an $M \times 3$ matrix, where each row is of form $[x \ y \ \theta]$ and M is the number of states along the path of the object in the world frame. The list of states assumes each state is separated by a fixed time interval. xy -positions are in meters, and θ is in radians. The default local origin is located at the center of the left semicircle of the capsule.
- **Geometry** — Size of the capsule-based object based on a specified length and radius. The radius applies to the semicircle end caps, and the length applies to the central rectangle length. To shift the capsule geometry and local origin relative to the default origin point, specify a fixed transform relative to the local frame of the capsule.

For now, regarding the state element, we just consider it to be a 1×3 matrix containing the current location of either the ego vehicle or the obstacle. In the Geometry element, there is a reference to a radius parameter, the reason being that each body stored into the dynamicCapsuleList object must be a capsule-like shape, as the one shown in figure 3:

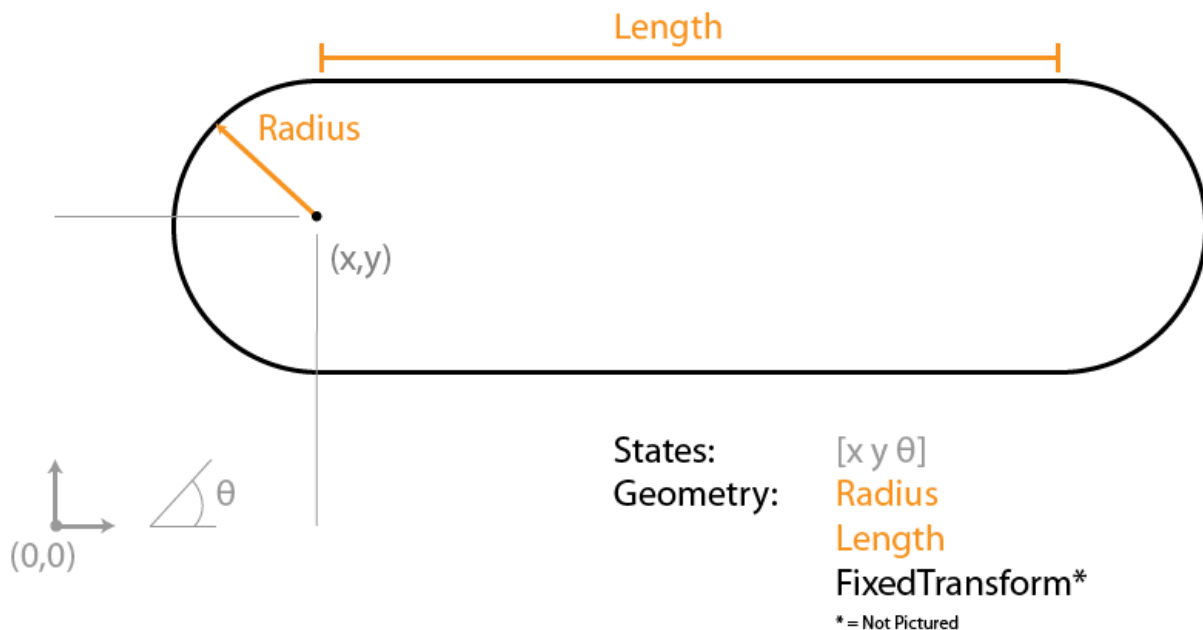


Figure 48 - 2D capsule (source: Matlab documentation)

The reason for it being a capsule like shape are to be searched into the most common application for obstacle avoidance: autonomous driving for the automotive industry. Since on-road vehicles are always of a rectangular shape and since the ego vehicle doesn't want to get extremely close to them, it makes sense to include some form of tolerance in the representation of an obstacle; in fact, looking at figure 3 it is easy to see that increasing the radius r , for a given rectangle shape of

length L and width $W < r$ will cause the navigation algorithms to keep more far away from the obstacle vehicle, both from behind (as the round capsule shape is placed in front and on the back of the vehicle) and from the sides, as increasing the radius increases the width of the capsule as well.

Up to this point in the program the obstacle are represented by cuboid objects. It is now necessary to transform each cuboid into elements of `dynamicCapsuleList`. As written above, the cuboid objects parameters are stored in a 9 by 1 vector. The necessary information for creating the capsule of an object are its x and y coordinate in the lidar reference frame, its yaw angle, and the width and length of the cuboid base, considering the length the longest of the two dimensions; since navigation is performed in 2D some data are discarded (the height, the z coordinate, the two angles of rotation around x and y axis). Actually, the fact that the length is the biggest among the two dimensions of the base rectangle is not guaranteed by default, but due to the nature of lidar scans (that cannot capture the whole cluster but only one corner and two sides, see figure 1) it is handy to have the length being the biggest. In order to do so, an if statement is added: if the yaw angle is bigger than 30° or smaller than -30° then length and width are swapped, and the yaw angle is set to 0° ; the cuboid creation function doesn't check for the longest dimension when choosing the yaw angle. It is fair to say that this measure was also adopted because the available dataset on which to perform testing were not agricultural dataset, they were city or highway datasets: in these scenarios vehicles tend to travel collectively along the same direction, with very small yaw angles especially for vehicles surrounding the ego vehicle (it is easy to visualize this fact when imagining a fleet of vehicles travelling a highway). Although this if condition is very useful in improving navigation performance in the tried datasets, it can be safely said that it would be useless when moving inside an agricultural-like environment, where the direction of moving actors can be unpredictable. The length and width are therefore extracted from the cuboid model, and the x and y coordinates too, but in order to use them for the capsuleList objects a further step is necessary, due to change of coordinates. The cuboid objects x, y and z proprieties are referred to the center of the cuboid, while the center coordinates of the capsule object are at the center of one of the semicircles of the capsules, as can be seen in figure 3. The yaw angle is also visible in figure 3; in the snippet of code shown below the necessary transformations for correctly passing between the two reference frames are shown. As another safety reason, there has been set minimum capsule dimensions for each cluster: in this way, undersegmented obstacles or small bodies can benefit from a larger safety margin; in the case of undersegmentation, having minimum dimensions makes so that if in subsequent lidar scans the cluster becomes much bigger due to correctly segmented data, the navigation algorithm will have already taken a safer road than the one that would have been chosen without this measure. The code is here reported:

```

obstacleLength=models{i}.Parameters(4);
obstacleWidth=models{i}.Parameters(5);
yaw = models{i}.Parameters(9);
if (yaw > 30 || yaw<-30)
    temp = obstacleLength;
    obstacleLength = obstacleWidth;
    obstacleWidth = temp*1.4;
    yaw = 0;
end
actorGeom(i).Geometry.Width = max(obstacleWidth,carWidth); %minimum clustercondition
actorGeom(i).Geometry.Length = max(carLen,obstacleLength);
actorPoses(i).States = [models{i}.Parameters(1)-
cos(yaw)*obstacleLength/2+obstacleWidth/2,models{i}.Parameters(2)-sin(yaw)*obstacleLength/2,yaw];

```

4.4 COLLISION CHECKING

The setting for obstacle avoidance is that all obstacles and ego body are stored in the capsule list structure; in order to validate a path for the vehicle to follow, it is necessary to check the ego vehicle for collision with every object. In the following chapter this concept will be extended in time, but for now let's limit the analysis at the static case, that means checking collision at the current time instant. There are various algorithms in literature able to check collision between general shaped complex objects, such as [6] but our case is a particularly favorable one; in fact, not only we are checking for collisions in R^2 , but we are also checking between objects that are convex; in fact, both the ego vehicle and the obstacles are represented by the capsule objects represented in figure 3, whose shape is clearly convex. This fact allows us to use tools such as the Separating Axis Theorem or SAT [7] and [8]. The SAT states that two convex objects are not in collision if there exist a line onto which the object projections do not overlap. The line on which the projections are cast on is called the separating axis, and if the objects are not colliding, the line that is perpendicular to the separating axis and that passes in between the two convex objects is called the separating line. The problem of determining whether two objects collide or not, therefore, becomes the problem of finding a separating axis. If the objects have particular geometry, their faces normals or other features direction can be used as candidate separating axis. In figure 4 the concept of separating axis and line are exemplified.

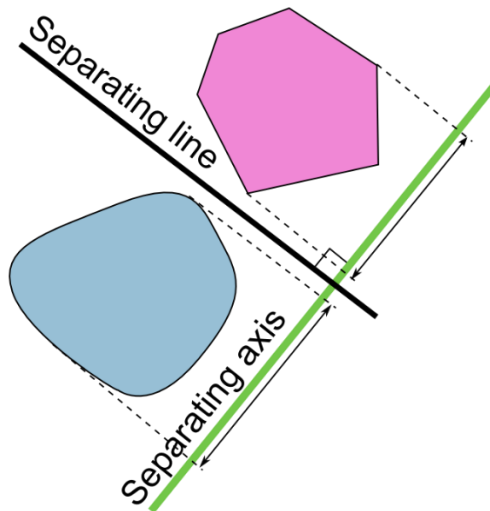


Figure 49 - SAT (source: Wikipedia)

A simple implementation of the SAT has been prototyped and tested for simple objects, but in the end it has been discarded: included in the `dynamicCapsuleList` class a method for checking collision is natively present, and can be invoked by calling the function `checkCollision`. The `checkCollision` function receives as input a `dynamicCapsuleList` object and provide several different outputs depending on the options specified. The function does not test all the bodies for collision between obstacles themselves, but it only checks for collisions between the ego body (or multiple ego bodies if present) and obstacle bodies. The method used in the current version of `checkCollision` is not specified, but in previous versions it was an implementation of the SAT [9]. It is important to remark the “static” nature of the collision checking action: the objects are checked in their current state and current geometry, but in these states the velocity, direction, or

acceleration are not specified, meaning that no future projection is done. This means that an object whose collision with the ego vehicle is imminent, or not far away in the future states, will not be detected; this by itself is not a flaw in the structure of the class but is a feature that needs to be correctly exploited, as will be made more clear in Chapter 5.

4.4 LIMITATIONS

Although the combined usage of the a-priori occupancy grid for the initial path planning and the discrete list of capsules for obstacle avoidance is convenient in terms of computation time and of simplicity of the approach it presents some drawbacks. Firstly, the “mostly free reference path” hypothesis must hold, and it is easy to imagine scenarios where this isn’t true, such as a herd of animals crossing the road. Since synthetic or real data testing was unable to be done due to hardware limitations and absence of dedicated datasets, it is unclear to what extent the obstacles on the reference path can constitute a problem. In the case of a topological change in the map, the “mostly free path” hypothesis wouldn’t hold. Another limitation is constituted by non-convex obstacles: this type of obstacles can be correctly identified and segmented, but in the cuboid phase their dimension will be inevitably increased, making them convex shapes. This by itself, in very simple scenarios, can be seen as a safety measure, as the telescopic handler won’t go near the non-convex obstacle, avoiding possible dead ends; on the other hand, this aspect may affect navigation performance overall, as some possible paths will be discarded even if they would be perfectly feasible. In general, complex obstacles or complex placed obstacles can determine the failure of the navigation algorithm, and therefore limit the usage of this kind of algorithm to open field simple applications. These kinds of limitations may be solved with the usage of dynamic occupancy maps, although real time execution speed would be an issue.

4.5 TELESCOPIC HANDLER: THE TELESCOPIC ARM ISSUE

4.5.1 Simple model of the arm

A part from the application field, there is a major difference between an automotive vehicle and a telescopic handler, and that is obviously the arm. The arm of a telescopic handler can move up and down around its hinge and can extend and retract. At the end of the arm there is a sort of end effector called nose, to which an horizontal metal component, called “zattera” is attached; the “zattera” is the docking point where various accessories can be linked, depending on the needed mission. From the navigation point of view, solving the problem of the arm is relatively easy: supposing that the vehicle is moving with its arm at rest (parallel to the ground) the easiest way to include the arm is to create a bigger capsule for the ego vehicle; this is a conservative approach not aimed at performance, but at safety. A less conservative approach is to model the handler, the arm and the “zattera” as individual and separate objects. This can be easily done with the use of the already used dynamicCapsuleList, in its 3D variant. The body of the vehicle and the “zattera” dimensions are fixed, while the arm length can vary. Therefore, the arm capsule will be able to vary its length aswell. The use of dynamicCapsuleList3D allows to use the same navigation algorithms as in the 2D case, without having to change any code except the part regarding the definition of the ego boides geometry. Creation of the three capsules is quite easy, and it involves some simple reference frame transformations. As for the main body, no changes were made from

the 2D case except now it is a 3D capsule. As for the arm, the reference frame of 3D capsules follows the same logic as 2D, see figure 3. The center of its reference frame is therefore placed where the hinge of the arm is placed. The arm capsules can be subject to a geometry transformation and to a state transformation; the geometry transformation is given by the current length of the arm, and it is a parameter already being kept track of in the current commercial machines; the state transformation is a rotation around the intrinsic y axis of the arm capsule. As for the “zattera”, its dimension can’t vary, therefore its capsule is only subject to a state transformation regarding its location: this is a translation that keeps the capsule at the end of the arm in its rotating and telescopic maneuvers. In figure 5 the obtained capsules are shown in four different moments of movement of the arm; the simulated movement is a typical for the telescopic handler: firstly the arm is aimed at a target (top two images) and then it is extended (bottom two images).

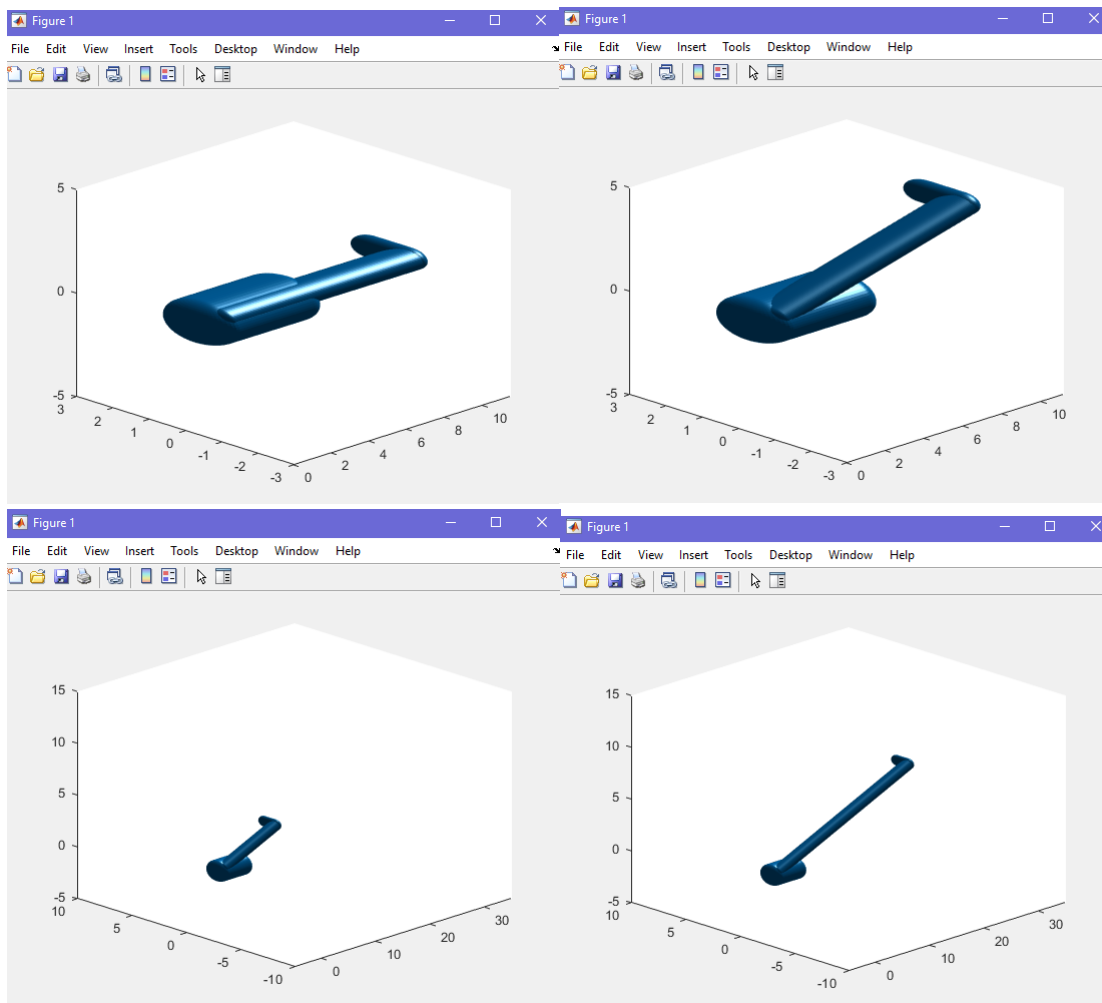


Figure 50 - top images: aiming the arm; bottom images: extending the arm

Thanks to the use of capsules, this three component model is readily usable in navigation, and can be used to navigate the telescopic handler when its arm is placed in a non-rest position. Due to the current handling of the obstacle, though, the algorithm is not able to exploit the 3D movimentations of the handler. This thesis work was aimed at pure navigation, but the next step for 3D object handling would require a different object avoidance than listing. A 3D occupancy

map is a possibility that could enable some sort of end effector movement, but this needs to be carefully thought in future work.

4.4.2 Obstruction handling

A part from being a component that can collide with the external world, the arm has another significant issue, and in order to understand it is necessary to take a look at the analyzed telescopic handler, and at the designed lidar attach point. The machine at matter is the Merlo MF 44.7, which is a telescopic handler aimed at agricultural work (figure 6). This thesis work has been developed using data recorded from a single sensor, mounted on top of a Ford Fiesta. The equivalent of this setup for the MF 44.7 would be to place the sensor on top of the roof, in the spot marked by the red dot in the left figure 6. The roof by itself constitutes an obstacle to the laser beams that are aimed at the space surrounding the base of the MF 44.7, and this alone implies the necessity of additional sensors whose job is to map the obstacles in close proximity.



Figure 51 – on the left: the Merlo MF 44.7. On the right: the inner corner points of the arm, in yellow

But there is another object that obstructs the line-of-sight of the lidar, and that is obviously the arm. If nothing is done in regard to the arm presence, the result will be that the navigation algorithm will constantly perceive a close obstacle to the right; in fact, it will perceive an obstacle so close such that all tested trajectories will result unfeasible and colliding with the arm. In order to solve this, the solution adopted is the same one adopted for the ground problem: it is worth remembering that also the ground is very close to the lidar sensor and that it too will be perceived as an obstacle preventing navigation. The solution for the ground was to try to fit an horizontal plane in the and then to remove the points that fit the plane from the point cloud, so that the whole ground area would be filled with NaN cells, that do not participate to the clustering algorithm and therefore will not constitute an obstacle. With the arm we'll do the same: all the points perceived where the arm is will be substituted by NaN values. In order to do this, we have to define the set of points that are obstructed by the arm. This can be done by defining four plane equations, choosing an internal positive normal direction for each of them and then doing the intersection of the points that satisfy each equation. Actually, since we are not planning on making maneuvers that involve avoiding obstacles specifically over or under the arm portion, we won't consider the whole vertical plane projection of the triangular region that has as vertices the

lidar itself, the posterior internal edge of the arm and the front internal edge of the arm. This means that we'll only need to have the equation of two planes (the "front plane" and the "back plane"). In order to comment the code, it is useful to remind the equation of a plane in \mathbb{R}^3 , which is given by:

$$ax + by + cz + d = 0 \quad (4)$$

Since the arm can move, rotating around its hinge and then extending, the equations of the two planes will change depending on the current state. In order to give a more realistic geometric model, the arm has been treated as a cuboid rather than a capsule; since we are not using this model for collision checking but for generating planes, the cuboid geometry best suits the task. The cuboid is created using fixed size for the section of the arm, namely 0.8 m, and using as variable parameters the inclination angle and the arm length; the cuboid then can be placed in any possible configuration in order to simulate the effect of the real arm on lidar readings.

Once the arm is in a certain position, represented by the cuboid, we first extract the corner points of the cuboid using the function *getCornerPoints*; we get the 8 points that are the vertices of the arm. Since the lidar sensor is put on the roof of the cabin, the points that are encountered by the lidar beam are the 4 internal points, shown in the right figure 6, highlighted in yellow. In order to distinguish which are the internal points in all the configurations, the following has been done: an array passing through the center of the arm cuboid, along the direction of extension, is defined. This array has the base in the pivoting point of the arm and the arrow pointing in the direction of the extension. The reference frame for lidar reading is the sensor placed on top of the roof, with the x-axis positive along the forward direction of the machine, the y-axis positive on the left direction and the z-axis pointing up. The arm on the MF 44.7 is placed on the right of the cabin, this means that the points belonging to the same quadrant as the arm are negative in y-coordinates, and the points closest to the lidar sensor are the ones whose y-coordinate is larger than the y-coordinate of the vector passing through the arm. In order to distinguish between front and back points, the criteria is that all points farer than 1.5 m are considered front points, as no arm can extend less than 1.5 m. Once internal points are divided into front and back, they can be separated into top and bottom points if necessary, but since we'll only define front and back planes we won't need it. Before explaining how the planes were found, a consideration has to be made: since the available dataset was the one of a ford fiesta, whose roof height is considerably less than the roof height of the telescopic handler, the point at which the lidar sensor is considered to be placed is not at the origin but at $[0;0;z_{\text{lidar}}]$, where z_{lidar} is set at 1.6m.

The front plane is the one passing through the lidar sensor and the two front points, while the back plane is the one passing through the lidar and the back points. Both are therefore passing through 3 points; in order to find the plane parameters in the form of (4) we wrote a simple Matlab function that solves the system of three equations obtained by substituting the 3 points in (4). We do this for both the front and the back plane, and get the equations of two planes. Both the front and the back plane are planes that are parallel to the z-axis (their normal is therefore orthogonal to the z-axis); this means that in order to find the points that are inside the arm region we need to write (4) having y explicit, and therefore the points inside the arm region, looking at the front or at the back plane, will be given by:

$$y < \frac{-ax - cz - d}{b} \quad (5)$$

Additionally, points belonging to the front plane will have their x-coordinate > 0 , while points belonging to the back plane will have their x-coordinate < 0 ; these conditions have to be kept in mind when removing the arm obstacle points, otherwise points that do not belong to the arm region will be deleted. In figure 7 the result of the code implementation of what explained up to now is reported, as top view. The arm is highlighted in red, and in this frame is not elevated. The point cloud readings are given in blue; it is easy to see the chassis of the car highlighted in the center of the image, placed at the center of the reference frame. The two planes are highlighted in the color purple, and extend from the center, intersecting the inner corner points. The region on the right of the two planes is black, as all lidar readings from that region have been replaced by NaN values. The region behind the vehicle also appears to not have any points, but this is the effect of the roof obstructing the view of the laser beams, and not part of the arm point removal algorithm.

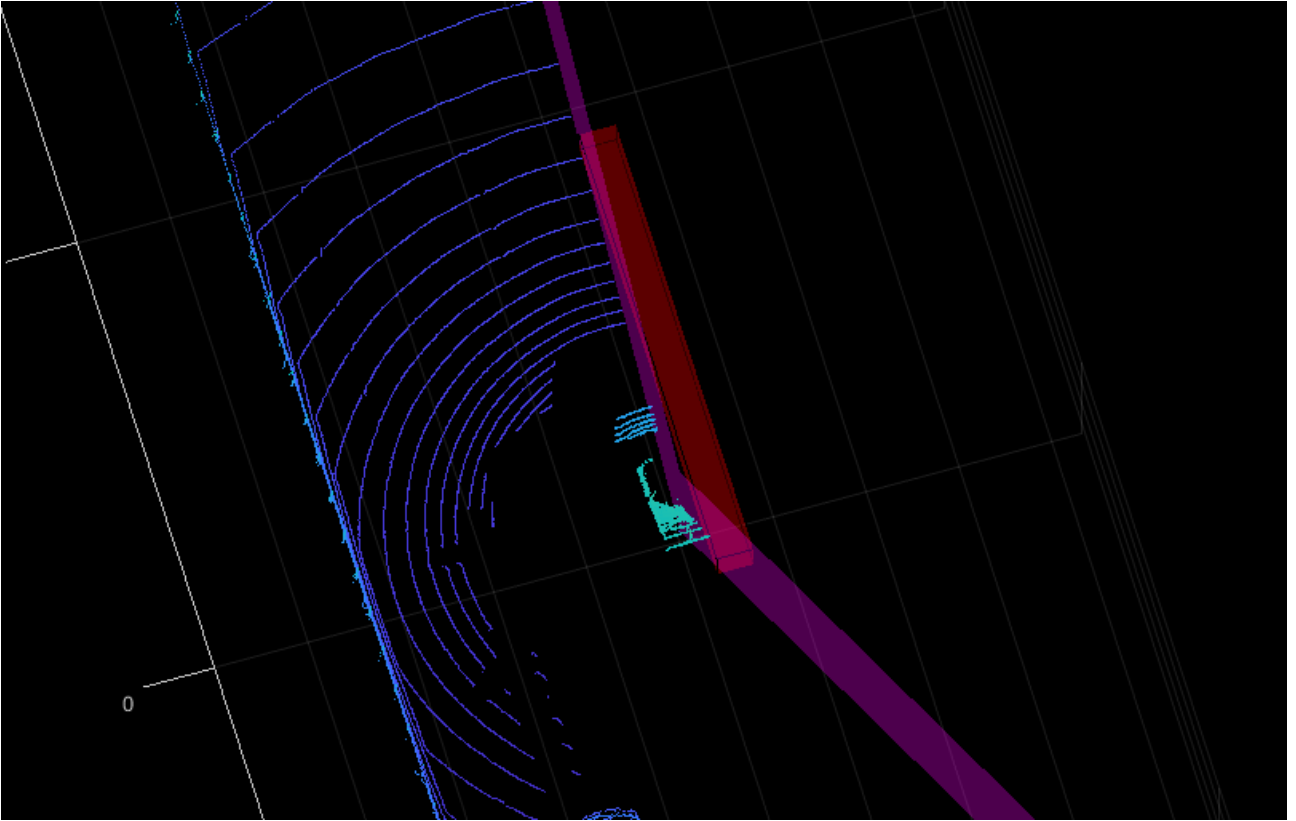


Figure 52 - result of arm occlusion point removal - top view

A legitimate question arising after seeing these result is: won't the vehicle try to navigate always on the right since no obstacles are perceived? Since the lidar can't see past the arm, it can't perceive the presence of obstacles and navigating in the arm-occluded region may have fatal consequences in people are present. But since it is necessary to eliminate arm point for the feasibility of any trajectory, it will be up to the navigation algorithms presented in Chapter 5 to take care of not going to the "forbidden region".

CHAPTER 4 - BIBLIOGRAPHY

- [1] : H. Zhao et. al., "Moving object classification using horizontal laser scan data", 2009
- [2] : Xiao Zhang et. al., Efficient L-Shape Fitting for Vehicle Detection Using Laser Scanners, 2017
- [3] : H. Moravec; A. E. Elfes (1984). "High resolution maps from wide angle sonar". Proceedings. 1985)
- [4] : Matlab documentation, occupancyMap, 2019
- [5] : Matlab documentation, dynamicCapsuleList, 2018
- [6] : Elmer G. Gilbert et. al., A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space, 1988
- [7] : Liang, C., & Liu, X. (2015). The research of collision detection algorithm based on separating axis theorem, 2015
- [8]: dyn4j.org, Separating Axis Theorem, 2010
- [9]: Matlab documentation, checkCollision, 2020

CHAPTER 5 - NAVIGATION

5.1 MODEL PREDICTIVE CONTROL

Model Predictive Control, commonly referred to as MPC, is a type of control algorithm that has been widely studied and used in recent years. Its flexibility and performance has made MPC one of the most versatile control methods for complex dynamic systems.

The use scenario of MPC doesn't reside in controlling simple systems in known conditions, but controlling either complex systems or simple systems in complex conditions; in particular, referring to the latter, MPC can efficiently handle changes in environment, perform future predictions and act consequently.

The core aspects of MPC can be understood by analyzing its name; MPC uses

- a **model** of the system (in control theory referred as *plant*), usually a dynamic model written in standard space state form. For several applications [1] it is sufficient to use a simplified model of the plant at study, capturing its main dynamics; more complex models lead to higher computation times.
- a **prediction** of the future state of the system, which is obtained by integrating the model equations over a certain period of time. This prediction feature is what allows MPC to be a flexible tool capable of applying in advance certain control inputs to the plant so to prepare to future changes
- **control** obviously refers to the fact that it is a control algorithm, but an important aspect of MPC can be understood by focusing on the control action. MPC is a great tool because, due to the model prediction action performed, it is able to both compute the control input and the state trajectory for a certain period of time. Most tools used in automation only allow for either computing a trajectory prediction of the plant state, or the control input to be applied to it, but MPC does both at the same time.

In the most general scenario we can consider a MIMO nonlinear system, that is written in the state equation form:

$$\begin{aligned}\dot{x} &= f(x, u) \\ y &= h(x, u)\end{aligned}$$

where $x \in \mathbb{R}^n$ is the state, $u \in \mathbb{R}^{n_u}$ is the command input and $y \in \mathbb{R}^{n_y}$ the output of the system. In MPC it is assumed that all states are measured with a fixed sampling time T_s and their values are provided with the same sampling to the NMPC algorithm to obtain a prediction of the plant. Naming y^{\wedge} the prediction of the y variable in the time $[t, t+T_p]$ obtained by applying an input sequence U , we can define the error between the reference r and the predicted y as:

$$\tilde{y}_p(\tau) \doteq r(\tau) - \hat{y}(\tau)$$

The working principle of MPC is the minimization of a cost function. This function, often referred to in literature as J , is a function that is defined between the current time period and a future time period, referred to as T_p , or prediction horizon. The function usually assumes the form reported in (1), where x refers to the state of the system, r refers to the reference output of the system, which can coincide partially or completely with the state of the system, R is a weight matrix that allow to give more importance to the regulation of some states rather than others.

$$J(u(t : t + T_p)) \doteq \int_t^{t+T_p} (\| \tilde{y}_p(\tau) \|_Q^2 + (\tau) \|_R^2) d\tau + \| \tilde{y}_p(t + T_p) \|_P^2 \quad (1)$$

There are three contributors to the function; the first contribution penalizes the diverging of the state of the system from the reference state r ; minimization of this component leads to a system that closely tracks the reference state; it is useful to anticipate that the reference state in our case is given by the a priori computed reference path.

The second contribution to J penalizes high command input activity, minimization of this term leads to a system that works with low intensity control inputs; this second term can equivalently be viewed as an energy conservation term, since controlling a system with minimum input activity will lead to a more energy efficient approach.

The third contribution to the cost function is a term that penalizes the final value of the difference between the reached state and the reference state; while the first and second terms are the sum of values computed at each sampling time from the current time to T_p , this latter term can be seen as a final value cost. This also means that the first two terms are computing a weighted vector norm, where each element of the vector is the value computed at a certain time instant, and the third term is simply a scalar.

As can be easily seen, the close tracking of the reference and the low energy consumption are two objectives in competition with each other. In order to choose the balancing between the two, the magnitude of the R and Q matrices needs to be tuned. Increasing the weight of Q leads to a system with better tracking and worse energy management, while increasing the weight of R leads to an energy efficient system with poor tracking performance.

The solution is obtained by solving an optimization problem that has the objective that is stated in (1), and that can be subject to a wide set of constraints. In fact, the ability of explicitly add the constraints to which the system is subject to is one of the good aspects of MPC; for example, it is possible to explicitly add input saturation constraints: this is useful in the case that the actuators that control the system have physical working limits, such as the maximum steering angle of a

wheel. By adding the constraints to the optimization problem, the solutions that will be found by solving it will already be feasible with respect to the physical system, and no additional action will be required. This is not the case for lots of control methods (such as Eigenvalue placement, LQ control), that quite often compute solutions where the control variables assume values that are not feasible for the physical actuators; in those kinds of control methods, the adopted solution is usually to slow down the system in order to need lower control actions, but this often leads to poor performance and is not scalable to more complex scenarios.

The result of the controller will be a control sequence $u(\tau)$, with $\tau=t, t+1, \dots, t+T_p$. This sequence can be interpreted as a vector of dimension of T_p , which is a function of time; this means that the minimization variable of the optimization problem is actually a function, and this makes J the function of a function, which is called a functional. Minimization of a functional is a hard task to accomplish, as the dimension of the $u(t)$ variable is very large; in order to simplify the problem from a computational viewpoint, two “tricks” are used.

Firstly, it is assumed that the variable u can only change up to a certain time instant called T_c , or control interval, which is a fraction of the prediction horizon T_p . After T_c , the input u is kept constant.

Secondly, in the time period that u can change, u is parametrized, using different parametrization. A popular one is here reported, where u is piecewise linear and continuous:

$$u(\tau) = \sum_{i=1}^m c_i \phi_i(\tau) = c\phi(\tau)$$

$$\phi_i(\tau) = \begin{cases} 1, & \tau \in [t + (i-1)T_s, t + iT_s] \\ 0, & \text{otherwise.} \end{cases}$$

The overall result gives the following formulation for the MPC problem which, since we have put ourselves in the most general case, can fall under the category of NMPC (Nonlinear MPC).

$$c^* = \arg \min_{c \in \mathbb{R}^{n_u \times m}} J(c)$$

s.t.

$$\dot{\hat{x}}(\tau) = f(\hat{x}(\tau), u(\tau)), \hat{x}(t) = x(t)$$

$$\hat{y}(\tau) = h(\hat{x}(\tau), u(\tau))$$

$$u(\tau) = c\phi(\tau)$$

$$\hat{x}(\tau) \in X_c, \hat{y}(\tau) \in Y_c, u(\tau) \in U_c$$

$$u(\tau) = u(t + T_c), \tau \in [t + T_c, t + T_p]$$

The solution of this problem is computed at each sampling time; it is important to notice that the computed solution is the optimal solution in the sense that it is the solution to a minimization problem but since this is, in general, a nonconvex optimization problem there exists the risk of local minima trapping. So the solution is an optimal one but possibly not a globally optimal one.

Another point to note is that the solution is optimal only for the current problem being solved (the time t problem), which means that at $t+1$ the solution will not be optimal anymore, and applying the solution found at a previous time instant would mean applying an open loop command.

Therefore the principle of receding horizon (RH) is applied: at any time t solve the optimization problem over the prediction horizon $[t, t+T_p]$ and apply only the first input $u^*(k)$ of the optimal sequence $U^*(k)$. At time $t+1$ repeat the optimization over the prediction horizon $[t, t+T_p+1]$ [2].

In real-world applications, the exact plant model is seldom known. This means that an approximated model f^\wedge, h^\wedge is used for control design, instead of the "true" model f, h (this holds for any method). Practical industrial experience shows that MPC tends to be inherently robust, even without any particular consideration in the design phase beyond ensuring the accuracy of dynamic models and formulating realistic specifications in terms of operational constraints and cost function weights. A necessary condition for lack of robustness is that the value function and state feedback law are discontinuous. There exists a wide range of NMPC formulations that include robustness into the formulation of the optimization problem. One can mainly distinguish between several robust approaches, such as the min-max NMPC, the H^∞ NMPC and the parametrized controller. All those techniques typically require a high computational effort and thus cannot be applied to problems where a small T_s is required. Thanks to the receding horizon strategy, standard NMPC is in general characterized by good robustness properties [1].

5.2 REFERENCE PATH

In order to be able to use any control algorithm, a reference signal must be given. In our case the reference signal to track is the path found in the offline path planning section, in Chapter 1. The reference path has been found using an algorithm such as RRT*. The found solution is composed of nodes and edges, following the graph taxonomy, but it can also be seen as a piecewise linear curve. In figure 1 the solution found by an RRT* algorithm is highlighted in red. Although it looks like a smooth curve, it is actually a piecewise linear curve, thus it presents abrupt variations of direction in the nodes. In order to create a smooth reference path, a naive approach could be to try and fit a $(N-1)$ order polynomial to the N points that constitutes the nodes of the solution found by RRT*. As explained in [3] this approach has some drawbacks:

- It is not possible to assign the initial and final velocities
- As the order of the polynomial increases, its oscillatory behavior increases as well, and this may lead to unnatural trajectories.
- Numerical accuracy for computation of polynomial coefficients decreases as the order of the polynomial increases
- The system of constraint equations is hard to solve
- The polynomial coefficients depend on all the assigned points; if even one point changes, all of the coefficients need to be recomputed.

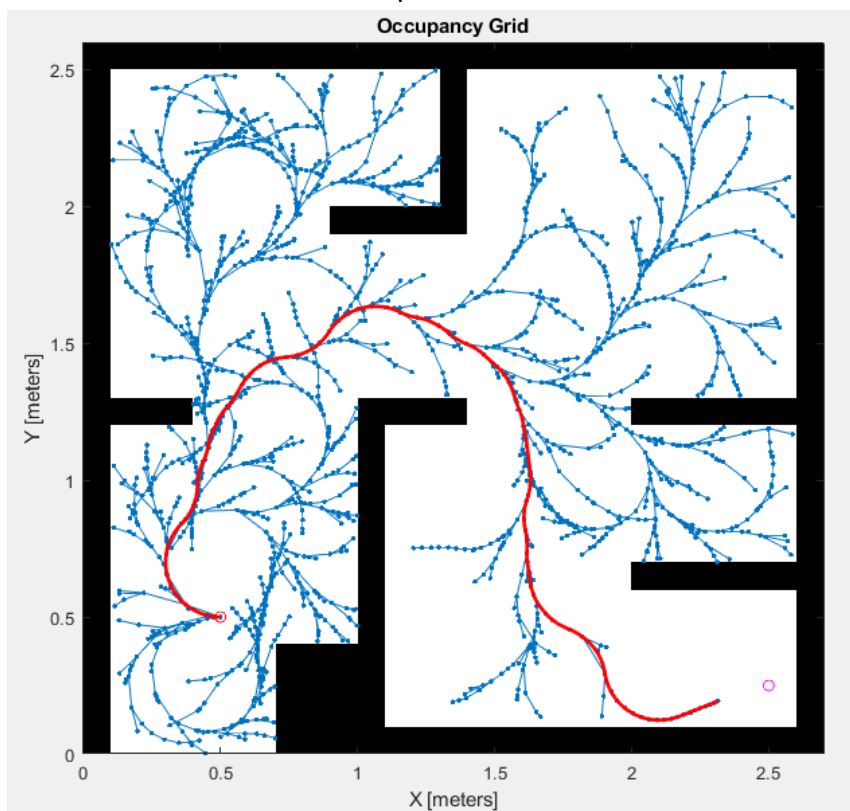


Figure 1: solution found by an RRT* algorithm in a test maze

A popular solution in robotics in order to overcome the presented drawbacks, is to use a number of low order polynomials as interpolation, continuous at the nodes. In particular, both the curve (which represents the position) and its first derivative (the velocity) must be continuous at the junction points. This implies that the minimum order polynomials that can be used are third order polynomials.

In Matlab it is possible to interpolate points using the built-in function *polyfit*.

5.2.1 Frenet Coordinate system

We firstly define a global coordinate system as an inertial coordinate system; for this purpose, we can imagine the reference system solidal with the RTK antennas installed on the working area, with its origin placed somewhere in the field, known. Since the obstacle representation, the a priori occupancy map and the navigation algorithms are all considered in \mathbb{R}^2 , the global coordinate system will be composed by an origin O and two vectors. The z-coordinate will not be considered.

Considering the solution found by the RRT* algorithm, in the global reference system we can write the coordinates of all nodes as couples (x,y).

Let's now suppose that all nodes in the RRT* solution have been connected using some continuous function, for example a single polynomial or, as explained in the previous section, a series of cubic polynomials, with continuity conditions at each node linking two cubic polynomials: this is required because the Frenet formulas apply to curves which are non degenerate, meaning that they have non-zero curvature in each point [4].

Once the curve has been defined, we can express points belonging to it as

$[x \ y \ \Theta \ k \ \delta k \ s]$, where x,y and Θ are Euclidean States expressed in the global coordinate system, with Θ in radians; "k" is the curvature of the curve in a specific point, expressed in meters; δk is the derivative of curvature with respect to arc length in meters per second and s is the arc length, computed starting from the path origin.

The path origin is placed at the first node of the curve, which corresponds to the starting position for the machine. In figure 2 the explained reference path is shown (source: Matlab).

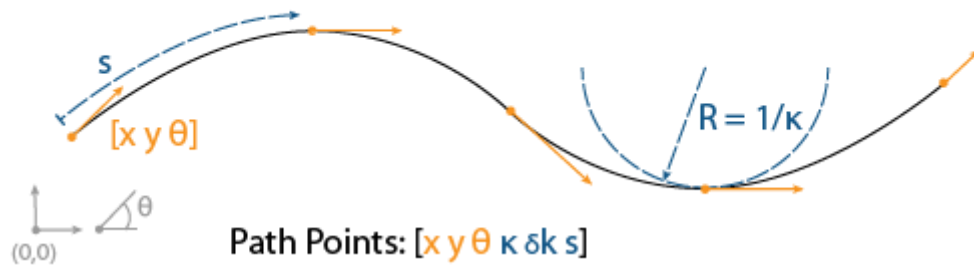


Figure 2 - reference curve for Frenet coordinate system

In Matlab we can create this reference path curve using the function *referencePathFrenet*. This function fits a set of points, called waypoints, to a smooth curve; the used method is not reported in the documentation, it could be fitting a series of cubic polynomials as explained in Siciliano book or some other method.

Once the reference path curve is given, we can express every point in R^2 in the Frenet coordinate system.

Since we are in R^2 , in order to express the coordinates of a point P, we need two coordinates. These coordinates in the Frenet coordinate system are the distance from the arc origin, defined as s , and the distance of P from the projection of P on the reference curve, defined as L .

We are interested in defining not only the position of a point P in the Frenet coordinate system, but also its velocity and acceleration. Therefore we can define the derivatives δs , $\delta^2 s$, which are the first and second derivatives of s with respect to time; we can also define δL , $\delta^2 L$, which are the first and second derivatives of L with respect to s .

We thus define the Frenet State of a point moving in R^2 as a vector of dimension 6: $[s \ \delta s \ \delta^2 s \ L \ \delta L \ \delta^2 L]$; in figure 3 the Frenet coordinates are visualized and the Frenet state is reported. The thick black line is the reference curve.

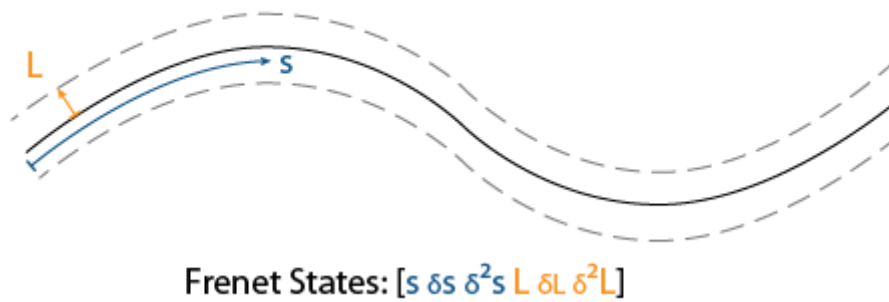


Figure 3: Frenet coordinate system - source: Matlab

Using the Frenet States we can define the position of a point by defining s and L . The other components of the state are more useful than simple cartesian velocity and acceleration, the reason are here reported:

- δs is the time derivative of arc length: since the vehicle will be traveling on the reference curve, or on an alternative curve defined using the same Frenet states, δs corresponds to the longitudinal velocity of the vehicle, which is the main component of the vehicle and the one on which we can act using wheels commands.
- $\delta^2 s$ corresponds to the longitudinal acceleration of the vehicle.
- δL is the derivative of L with respect to s , and it corresponds to the slope of the alternative trajectory with respect to the reference curve; a zero δL refers to a direction that is parallel to the reference path.
- $\delta^2 L$ is the concavity of the alternative trajectory with respect to the reference curve

The definition of the above quantities allows us to easily define alternative states for the vehicle and thus alternative trajectories, using variables that refer to an intuitive reference frame for the computation of alternative trajectories.

.see also (Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenet Frame Moritz Werling, Julius Ziegler, Soeren Kammel, and Sebastian Thrun)

5.2.2 Frenet trajectory generation

Using the Frenet states, it is possible to define points that have specific features that relate to the reference curve. If we define a parameter for a number of points we can define curves. For example, only defining the fourth parameter, L , for a set of points, is defining a curve parallel to the reference path (shown as dashed line in figure 3), that can be seen as a path as well.

We may not be able to define a state for every single point in which we want to pass, but we can define the initial state and the terminal state, and have some sort of interpolation algorithm do the linking between them.

However, we are not simply interested in creating a geometric path, but we also care to define a timing law as well. The path obtained as a “smoothed” solution of the RRT* algorithm is a pure geometric object that lies in R^2 . It can be parametrized as a function of its length but it remains a geometric object. A trajectory, however, can be defined as a “path plus a schedule” [5], or “path plus timing law”. The first thing we need to do is define what is the reference path we are going to use, in order to have the origin and axis of the Frenet coordinate system defined.

In the Matlab Navigation Toolbox, there is an object called *trajectoryGeneratorFrenet* that can be used to create trajectories, both on and off the reference path. In order to use it, we first create a *connector* object, using as argument the reference path : *connectorFrenet = trajectoryGeneratorFrenet(refPath)*.

Next, in order to define a trajectory we need: a starting point, a final point and, in the simplest case, the time needed to arrive from the former to the latter. More in detail, we’ll work using the Frenet State; so instead of a starting and terminal point we’ll define a starting and terminal state: this allows us to specify not only the waypoints of the searched trajectory but also speed, acceleration, curvature...at desired points. We’ll set the states only for the starting and ending point, and let the *connect* object fit a polynomial between the two.

In order to use the *connect* object function to create a trajectory between the starting state and the ending state, we specify the time the movement between the two needs to take, as follows:

$$[\sim, \text{trajGlobal}] = \text{connect}(\text{connector}, \text{initState}, \text{termState}, \text{time}) \quad (2)$$

At this point the *connect* function computes a feasible solution that connects the states, generating a number of intermediate Frenet States in between the two.

The solutions are found as fourth or fifth order polynomial curves, depending on the number of constraints applied: the application of a constraint is implemented by specifying a state value for an element in the Frenet state. The elements of the starting state are all known, and thus defined, while in the elements of the terminal state we have a degree of freedom of leaving one or more values undefined, inserting the value NaN.

In figure 4, the reference path is shown in red, and a set of alternative trajectories are shown in green; this set of trajectories has been obtained by setting as terminal states: *termstates = [30 0 0 L 0 0]*; where *L* is a vector whose values span between -10 and +10; thus *termstates* is a vector as well: the *connect* object can accept vectors as an argument, and returns a number of trajectories equal to the size of the vector.

If instead we impose the terminal state to be $termstate=[s \ 0 \ 0 \ NaN \ 0 \ 0]$ we are relaxing the problem on the fourth constraint, therefore obtaining as solution a fourth order polynomial instead of fifth. In many cases we don't know the value of all elements in the terminal state, a representative example is given by the terminal state generated in a overtaking maneuver: in this maneuver we may want to keep a certain final velocity, to have some lateral displacement, but we probably are not interested in the arc length s of our trajectory, as long as velocity and lateral displacement are respected.

Constraint relaxation has been shown to produce more natural-like trajectories: in figure 5(a), the terminal state has been set to $[6 \ 10 \ 0 \ 5 \ 0 \ 0]$, while in figure 5(b) it has been set to $[NaN \ 10 \ 0 \ 5 \ 0 \ 0]$.

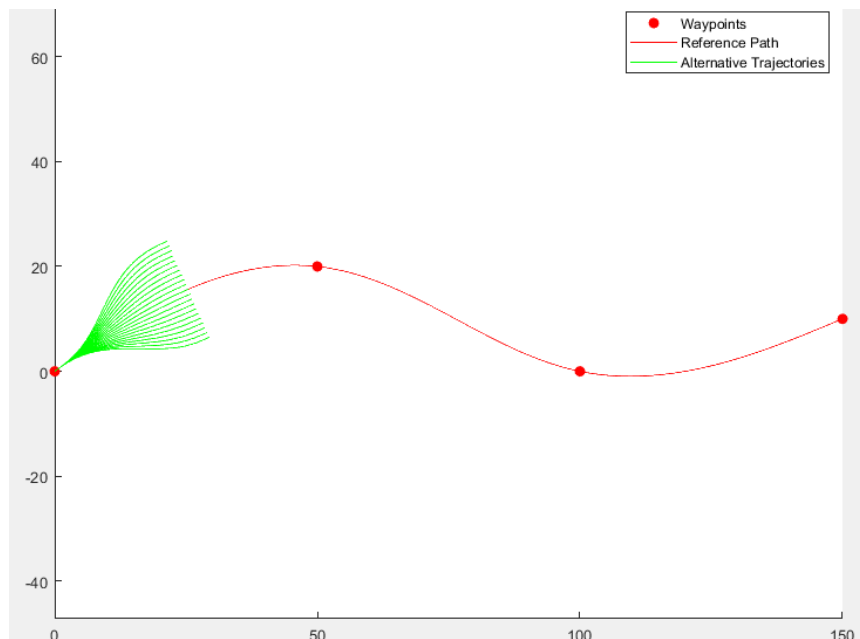


Figure 4: reference path and alternative trajectories setting all terminal state values

In figure 5(a), the imposed arc constraint makes it so that the trajectory starts in one direction, returns back to the starting point, and then proceeds forward until the terminal state is reached; this is an unnatural and dangerous behavior, and although the terminal state has been reached, the way it was reached wouldn't make sense. Relaxing the arc length constraint (figure 5(b)) respects all velocity and lateral displacement constraints while providing a natural curve, although the control on the arc length is lost. Another possible solution, which is not possible to perform using the *connect* function, would be to relax the constraint on time; this would lead to reaching a terminal state with also the arc length specified, and could be a topic of a follow up work.

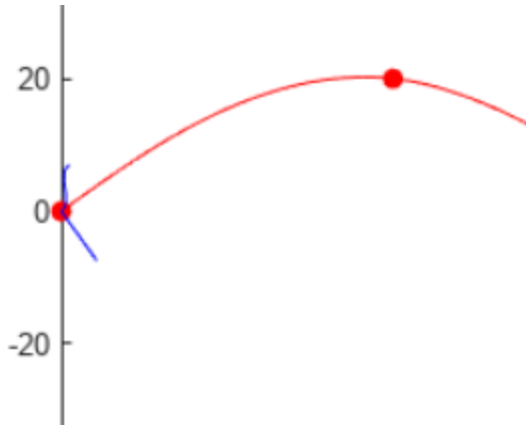


Figure 5(a): all constraints imposed

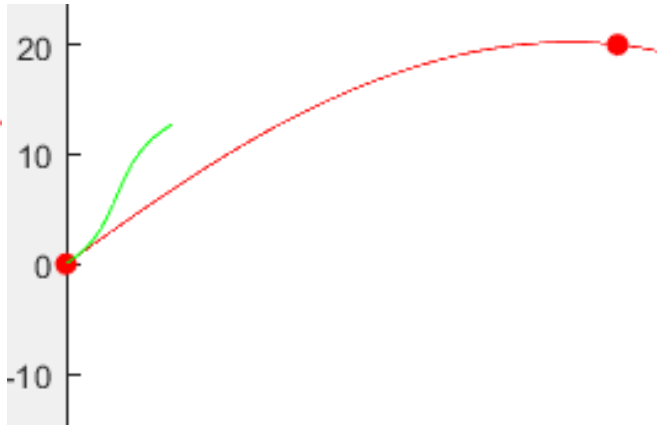


Figure 5(b): arc length constraint relaxed

The line of code reported in (2) gives as output the `trajGlobal` variable, which is a data structure composed of two fields: *Trajectory* and *Times*.

- *Trajectory* is a N-by-6 matrix, each row of which is a path point in the Frenet state (despite the name *Trajectory*, this is actually a path in \mathbb{R}^6) or in the global reference system. In the latter case, which is the default one, the state is composed by $[x \ y \ \theta \ k \ v \ a]$, where x and y are the coordinates in the global reference frame, θ is the inclination angle with respect to the x axis and v and a are, respectively, velocity and acceleration modules of the point in the θ direction.
- *Times* is a N-by-1 vector, which associates a time value to each row in the *Trajectory* matrix. The time between consecutive rows in the *Times* vector is fixed, and chosen before calling the `connect` function; this time step can be tuned and chosen in different ways, the choice that was carried out in this work is to set it at 0.1s. The reason for this is that the Lidar scans of the dataset used for testing were collected with a frequency of 10Hz; choosing as time step 0.1s means that the very next state after the current state of the vehicle will be reached when a new Lidar scan will be recorded. This concept has some connection with the idea of receding horizons presented in MPC, but will be better explained later. Other choices of the time step are possible too, being anyway smaller than 0.1s.

5.2.2 Cinematic constraints

There are cinematic constraints related to the trajectories that the telescopic handler can perform: thanks to the use of the RRT* algorithm for computing the solution of the offline path planning problem, the minimum curvature radius constraint was already considered, and the resulting reference path curve already contains it. Only looking at the geometric side of the trajectory (so, the path) we therefore are sure of the feasibility of the reference curve, but we don't have any guarantee about the feasibility of the alternative trajectories.

As will be explained in the next sections, due to the mechanism of generation of the alternative trajectories, we don't have a way of pre-imposing the cinematic constraints before computing the alternative trajectories themselves.

What will be done is, once the set of alternatives has been computed, each alternative will be tested for feasibility of the constraints. The trajectories that won't satisfy all the constraints will be discarded and not used down the line.

The minimum steering radius of the Merlo MF 44.7 is 4.090m. The maximum curvature is the reciprocal of the radius, and so is:

$$\text{maximum curvature} = 1/\text{radius} = 1/4.090\text{m} = 0.2445\text{m}^{-1}$$

Therefore, we set a parameter to the maximum curvature of the telescopic handler at study.

The maximum acceleration allowed for nominal-situation driving in highway scenarios [6] lies in between $15 \div 20 \text{ m/s}^2$. We have set a maximum acceleration of 10 m/s^2 .

The maximum permissible velocity for this autonomous vehicle application has been set to 5 km/h, or 1.39 m/s . It is important to remember that the autonomous vehicle will operate only in controlled environments, and won't navigate in streets, therefore we don't need minimum speed requirements. Recap of cinematic constraints:

- maximum curvature = 0.2445 m^{-1}
- maximum acceleration = 10 m/s^2
- maximum (longitudinal) velocity = 1.39 m/s

The procedure used in order to apply these constraints is the following: given the set of alternative trajectories to the reference path, for each trajectory the function *evaluateTrajectory* is called, passing as an argument all the trajectory points of the trajectory under testing. The trajectory points are passed as N points in the global reference frame, this means they are [x y theta k v a]. The check performed by the *evaluateTrajectory* function is then quite easy, as each point of the global trajectory passed is tested for its k, v, a parameters to be lower than requested. The *evaluateTrajectory* function returns 1 if the three cinematic requirements are all met, otherwise it returns 0.

Additionally, it is also possible to impose minimum cinematic requirements, especially in the velocity field.

5.3 NAVIGATION ALGORITHM ANALYSIS AND SIMILARITIES WITH MPC

5.3.1 Initialization of capsule list

The first thing that is done after the reference path has been defined is the creation of an empty capsule list. Since no tracking has been implemented in this work (see section 5.4 for the theoretical analysis on tracking) the empty capsule list is created at each sampling time instant. This means that no knowledge of previous obstacles or their motion is kept in memory. In order to avoid having “ghost obstacles” that are present in the capsule list but aren’t actually present the capsule list is deleted before each measurement is processed. Another measure taken in order to simplify capsule list handling is the a priori selection of a fixed number of obstacles. This choice greatly reduces the effort required to dynamically change the size of all data structures connected with the capsule list, and provides robustness to the code. The number of a priori capsules that are created has been set to 10. Testing using the two datasets found on the Matlab website has shown that 10 is an appropriate number for road applications. No data has been analyzed in the agricultural environment, though.

The 10 capsules are created and initialized in a position far away from the ego vehicle, at (-1000,-1000) in the cartesian plane. The default size of the capsules is [5,2] meters.

The parameter *maxHorizon* is chosen: this parameter is the equivalent of the prediction interval T_p of the Model Predictive Control. It is the maximum time for the navigation algorithm to perform predictions about the future states.

5.3.2 Alternative trajectories generation

After the initialization of the capsule list, all of the alternative trajectories are computed. The alternative trajectories generation happens in the same way at each iteration, and could be potentially avoided by storing them in dedicated variables, but the computational cost of computing them is negligible.

Firstly, the vector containing all the alternative terminal states is computed. The terminal states are created as zero vectors in the Frenet state, where the constraint on the longitudinal element is dropped by imposing it equal to NaN, the second term corresponding to longitudinal velocity is put equal to the chosen target velocity (since we are interested in arriving at a destination and stopping, this is put to 0) and the term on the lateral offset is put equal to 2.5m. The choice of 2.5m is arbitrary and should be tested with real world agricultural data. The result is a set of trajectories equal to the ones reported in green in Figure 4.

This approach for generating alternative trajectories can be seen as a static approach, as it is the same at each iteration. This results to be quite different from the MPC formulation that gives as a result the control command $u(t)$.

In fact, the MPC solution $u(t)$ is the solution of the MPC minimization problem, which is the minimization of the J cost function (see paragraph 5.1, equation (1)), which is nonlinear. That problem is tackled by minimization algorithms which, given the nonlinear nature of J , can end up being trapped in local minima. In any case, the computational effort required to solve at each iteration the MPC minimization problem is the main drawback of MPC.

In this work, the much simpler trajectory generation will surely bring less than optimal solution, but the approach is justified by the supposed sparsity of obstacles in the working environment of the telescopic handler (see previous Chapters).

The behavior produced as a result of this choice of terminal states is simply “try to avoid the obstacle by passing either to its left or to its right”, as it's supposed to be true that the obstacle is one that can be passed in such a way.

On field validation needs to be performed to verify that these kinds of hypotheses and solutions proposed are sufficient to correctly tackle the problem.

5.3.3 Terminal states cost function

While the alternative terminal states and trajectories are being computed, the cost of each terminal state is computed as well, and stored for later. Each terminal state is passed onto a function called *evaluate_terminal_costs* that computes the cost associated with the individual terminal state.

There can be very different ways to compute a cost function. In the more general approach considered in the MPC formulation the cost factors are the weighted norm of the difference between the state and the reference and the vector norm of the input vector u , for energy saving or input constraint purpose; both factors are weighted and are functions of time, and the cost function is defined in the interval $[t, t+Tp]$. There is also a term associated with the cost of the state reached at Tp , and that term is independent of the time.

In this work the cost function used is much simpler than the complete formulation of the MPC cost function, and only considers the latter term, the one associated with the state reached at the end of the prediction interval, which in our case is given by the terminal states. This cost function will not provide the same level of performance of the full MPC cost function, but the computational cost of computing it is very low and therefore friendly with the real time requirements.

The cost function is composed of two terms: a velocity cost and a time cost.

- The velocity cost penalizes terminal states whose speed differs from the reference desired speed; the cost is linear.
- The time cost penalizes fast trajectories. Although this may sound counterintuitive, slow trajectories are the best trajectories for a non urban autonomous vehicle, as they are

smoother than faster trajectories, since slower trajectories means lower instantaneous accelerations.

The terminal states (and subsequently their trajectories) are then sorted according to the computed costs: they will be tested for collision starting from the ones with less cost associated.

5.3.4 Lidar data processing

At this point, the current Lidar frame is processed. Since most of the point cloud processing has been already explained in the previous chapters, here only the order in which the algorithms are executed will be reported.

Firstly, the data in the point cloud belonging to the ground is removed. Secondly, the remaining points are clustered in order to recognize objects; here both DBSCAN algorithms or FLIC can be used. Thirdly, the clusters are fitted using L-shape fitting into cuboids models. The following piece of code is reported as it synthesizes the operations performed:

```
%% 4. fit cuboids
for i = 1:numClusters1
    idx = find(labels1 == i);
    models{i} = pcfitcuboid(ptCloudWithoutGround,idx);
    points1 = getCornerPoints(models{i});
    plot(models{i})
end
```

loop performed for cuboid fitting of all clusters

Once cuboid models are obtained, they are assigned to the capsules that were initialized at the beginning. The unused capsules are left in the default location of (-1000,-1000), where they do not constitute an obstacle for any trajectory.

The dimensions of the capsules are updated as well: in particular, they are assigned the maximum between their default value and the cuboid model value. This measure is a safety precaution taken in the case of under segmentation of a cluster, for example in the case some part of an obstacle is labeled as ground. Having minimum dimensions will make the navigation more cautious.

Errors will arise in the case where the number of cuboid models exceeds the number of initialized capsules.

5.3.4 Trajectories collision checking

Starting from the least cost trajectory, the cinematic constraint valid trajectories are tested for collision avoidance. The collision checking is performed by a pre-built function belonging to the capsuleList object.

The collision checking is performed exploiting a prediction of the ego vehicle, performed up to the *maxHorizon* time instant.

The model which is predicted is a pure cinematic one: the center position of the ego vehicle is computed according to the values computed in each alternative trajectory up to the *maxHorizon* time instant. This concept presents some similarities with the MPC formulation, although it is much simpler. In the MPC formulation the inputs can be the actuation inputs of the actuators of the autonomous vehicle, and this would require to know the dynamics of the whole vehicle. Here the researched inputs are simply the trajectory waypoints, and this means that the model with which the predictions are made can be a purely geometrical one. Nevertheless, simulating this simple model still allows for multiple considerations on the navigation algorithm in all of its parts.

From the point of view of the capsule list, it is considered as multiple ego vehicles are present at the same time, the multiple vehicles being the prediction of the ego vehicle across time.

In Figure 6 the predictions of the ego vehicle along the green trajectory are reported in gray.

The capsules containing red rectangles are the obstacle capsules, and the red rectangles are the cuboid models; it is worth noting that the object situated at the most right presents a capsule much bigger than its dimension (due to the cautious approach cited above) and it is clear to see how this does constitute a safety precaution, as in the absence of the big capsule one of the feasible trajectories could pass right in front of the obstacle.

In the bottom left corner a small capsule without a red rectangle inside is present: it is one of the unused initialized capsules, which has been placed at $(-10,10)$ instead of $(-1000,1000)$ for visualization purposes.

Finally, a green hollow rectangle is present in the center of the image: that is the position of the ego vehicle and the first capsule directly around it is the capsule at time t .

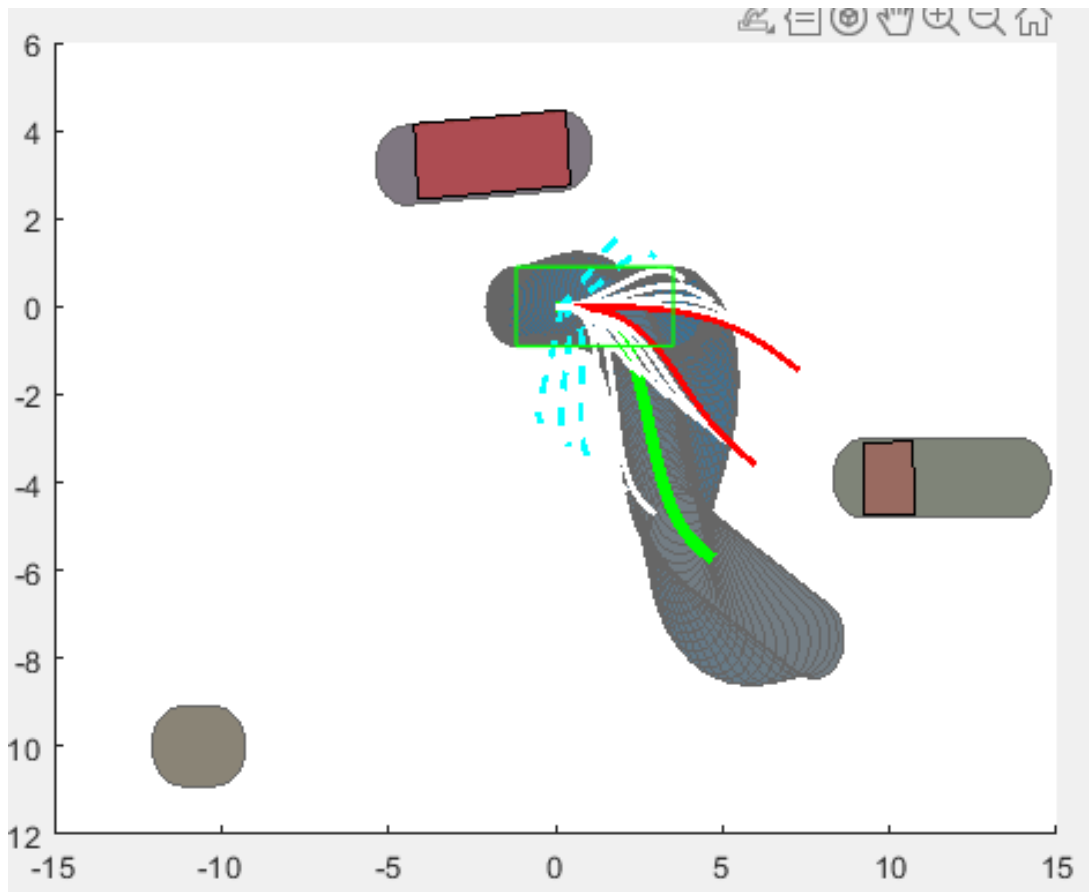


Figure 6: predictions of the ego vehicle along a candidate trajectory

About the color of the trajectories:

- the white trajectories are collision untested trajectories: in fact, once a trajectory is found as feasible, all superior cost trajectories are not checked for, in order to save computational costs
- the green trajectory is the lower cost collision free trajectory
- the red trajectories are trajectories that present a collision with an object in the *maxHorizon* time interval
- the light blue dotted trajectories do not present feasible cinematic constraints and are therefore discarded from the collision checking process

The on-path trajectory is always the first one being tested initialized with a cost of 0.

5.4 TRACKING

An important observation needs to be made about the work done so far; as a reminder the information workflow so far has been: sensor acquisition of Lidar data, ground preprocessing, obstacle clustering, obstacle list update and finally navigation. The whole stream of algorithms is performed in real time, at the highest frequency that the hardware and the software implementations allow. However, at every iteration of the algorithms, the information gathered in the previous cycles is lost; for example, at each new Lidar scan the chosen clustering algorithm is run and new clusters are created, discarding the old clusters and all information contained in

them. This means that whenever an object is detected, it is considered as a new obstacle every time. It can be said that the system works statically, as all decisions are made with respect to the current state without using the past variables, and that the notion of time is unused.

This of course is not the optimal way to handle all the gathered data, and the tool to be used in order to improve performance is tracking.

Tracking refers to the activity of predicting and following an object's motion (more generally, an object's state), as well as memorizing its history, by correlation of old and new measurements; in general, tracking is done with respect to multiple objects.

Implementation of tracking presents numerous advantages:

- It is possible to handle missing data frames: an existing track at time $t-1$ can be kept and considered as an obstacle at time t even if no new measurements have been made at time t . This makes the navigation system resilient to false negative errors
- It is possible to handle sensor uncertainty and perform filtering activity on noisy data; in the case of challenging environment conditions (an example for the Lidar sensor are reflective weather conditions like snow) the usage of tracking can mitigate the effects of "jumpy" detections
- It is possible to perform predictions on the obstacle conditions for a time interval in the future: this aspect is of crucial importance for the navigation aspect, as future predictions of the obstacles can be included in the collision checking phase of the alternative trajectories validation, and lead to the choice of better alternatives that already consider a prediction of the future environment: this will make less likely the phenomenon of frequently jumping between different trajectories, and lead to a smoother trajectory planning overall.

In order to talk about the tracking algorithms, we'll refer to the concepts of detection and track. A detection can be defined as a measurement of an obstacle performed at a present time instant, t . It is also often referred to as an observation.

A track can be defined as a past detection, or the measurement of an obstacle at a past time instant, $t-1$. The set of all detections of a past time instant $t-1$ can be called the set of tracks.

We will also define two algorithms that are commonly used in tracking, either in their original form or in a variant flavor. These algorithms are the Hungarian algorithm (also known as Kuhn-Munkres algorithm [7]) and the Kalman Filter.

- The Hungarian algorithm is a tool that can associate an object from a past frame to a present one, using some kind of score as a metric [8]. This algorithm is used to solve the so-called **data-association problem** of tracking.

- The Kalman Filter is an algorithm that can perform a prediction of the state of a dynamic system based on current state (and possibly, input) measurements, as well as perform current state estimates by correcting the previously computed prediction with the current measurement. A popular form of the Kalman filter is given by the Prediction/Correction form [9]. This algorithm is used to solve the so-called **prediction problem** of tracking.

5.4.1 Data-association problem - Hungarian algorithm

In order to tackle this problem with the Hungarian algorithm, we need to choose a score with which to compare the tracks and detections of two consequent frames. Before that, we need to define what kind of object the detections (and as a consequence the tracks) are.

Since in most computer vision solutions that use tracking the classification part is left unchanged with respect to the solutions that don't use tracking [10], we will keep the classification section of this work unchanged, and introduce the tracking after it. In our case, where the only sensor used is the Lidar sensor, the classification part is composed of the clustering algorithm.

In order to reduce computational complexity, instead of considering as detections the clusters of points, we'll consider the cuboids instead. Since we are performing navigation in 2D, the variables that define each cuboids are 5: $[x, y, \theta, w, h]$, and therefore what will be used as detection won't be a full cuboid but only the rectangular base of it. The detections are thus oriented rectangles in R^2 .

The observation "frames" are thus composed of a set of rectangles. The rectangles of the $t-1$ frame constitute the set of tracks, while the rectangles of the t frame constitute the set of detections. There are some popular scores used in the literature for comparing subsequent frames:

- Intersection over unit (IOU): this score function is defined in $[0;1]$. It is computed by measuring the overlapping portion of two rectangles, one belonging to the set of tracks and the other to the set of detections, and dividing it by the sum of their areas. Three cases are reported in Figure 6, where the tracks are colored in blue while the detections are colored in red:

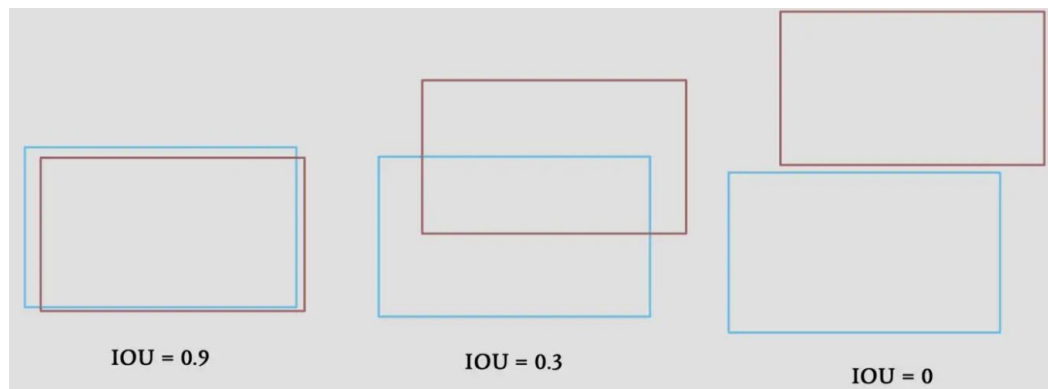


Figure 6: three

values of IOU score. (Source: thinkautonomous.medium.com)

- Shape score: this score function is defined in $[0;1]$. The score is higher for detections that have similar shape and size from previous frames. Given that all detections and tracks are, in our case, rectangles, using this score could make it difficult to distinguish between different objects.
- Convolution cost: this score works by running a Convolutional Neural Network on all the detections and tracks and, where the convolutional features remain the same it means that the object is probably the same one.

We'll consider the IOU score. In Figure 7 the time instant $t-1$ is reported as $t0$, the time instant t is reported as $t1$. In $t1$ the detections are the red rectangles, while the tracks are the blue dotted rectangles. The detections are three, two of which overlap to some extent the tracks and one of which appears to be a new object. It is possible to build a matrix by writing on the rows the $t1$ detections and as columns the $t0$ tracks, and in which each element ij on the matrix is the IOU score of the detection i with the track j :

Detection / Track	Track 1	Track 2	Track 3
Detection A	IOU = 0	IOU = 0	IOU = 0
Detection B	IOU = 0.56	IOU = 0	IOU = 0
Detection C	IOU = 0	IOU = 0.77	IOU = 0

Looking at this table it is immediate to correlate Detection B with Track 1 and Detection C with Track 2. We'll also refer to this table as a matrix, named **C**.

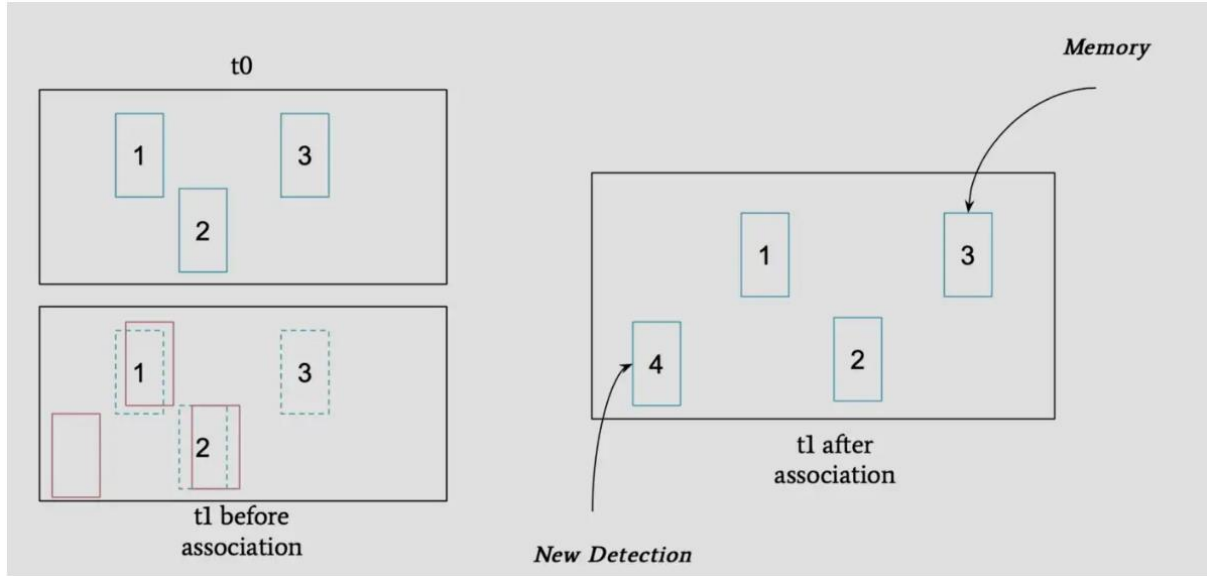


Figure 7 : example of two subsequent frames after clustering and rectangle fitting (Source: thinkautonomous.medium.com)

As far as Track 3 is concerned, in the present time instant no measurement has matched it. This could have happened either because the object that Track 3 belonged to disappeared or because of a false negative error: an unsatisfactory clustering action, a missed Lidar measurement ecc.

Using the notion of tracks, we may still keep Track 3 in memory for future time instants subsequent to t_1 , where maybe some detections will match with it; this is what allows the tracking algorithm to behave like a noise filter. It is common to insert in these implementations a threshold, where if a track doesn't receive detection updates after a number of measurements, it is dropped.

In the end (see Figure 7 on the right) we are left with four tracks, one of which comes from a new detection, Detection A.

So far the covered case was the simplest one, meaning that in the Detection/Tracks matrix there is at most one nonzero IOU score for each track (or for each detection). In the most general scenario this is not the case: the matrix C can be composed of all nonzero elements. Nevertheless, since the score function is defined between $[0;1]$, it can be said that it is a nonnegative matrix. The problem becomes the one of finding the best detection match for each track, under the hypothesis that there is at most one detection for each track. This hypothesis is strong, and can be sometimes not respected in the cases where the clustering algorithm over segments a single object: this is a case where for two detections a single track is corresponding.

The mathematical formulation of the general case is:

$$\max Tr (L * C * R) \quad (3)$$

where Tr indicates the trace of a matrix, and L, R are permutation matrices, which are the optimization variables of the problem. This is the problem that is solved by the Hungarian algorithm, and the output is the matrix M who matches the detections to the tracks by maximizing the IOU (or other) score.

Thanks to the Hungarian algorithm we can assign old and new measurements to each object, which we can denote with a number k . For each object k we can define a state x^k , which is a vector of dimension 5: $x^k = [x, y, \theta, w, h]$, where x, y are the rectangle center coordinates expressed in an inertial reference frame, for example the reference frame composed by the RTK antennas. Also, we can denote with x_t^k the state of object k at time t .

5.4.2 Prediction problem - Kalman Filter

Once we have identified an object, and we are able to assign new measurements to it, we are interested in predicting its future state and correcting the current state measurement with the prediction obtained from past measurements.

This problem is tackled using the Kalman Filter.

The Kalman Filter is an algorithm proposed by R.E. Kalman in 1960 [11] that has been widely studied, used and expanded during the decades.

In the paper, Kalman defines three different problems, namely the data smoothing problem, the filtering problem and the prediction problem. Given that the current time is t , and the time at which we are interested in knowing the values of the state of an object is $t1$, we have a data smoothing problem if $t1 < t$, a filtering problem if $t1 = t$ and a prediction problem if $t1 > t$.

In the case of tracking, we are interested in the filtering and the prediction problems.

The setting of the problem is the following: we are interested in filtering and predicting the state of an individual track k . Such state is $x^k = [x, y, \theta, w, h]$. We do not know the inputs that the track is subject to, as we can only measure the output using the Lidar. Defining each track k as a discrete system S :

$$S : \begin{cases} x(t+1) = Ax(t) + v_1(t) \\ y(t) = Cx(t) + v_2(t) \end{cases} \quad t = 1, 2, \dots$$

where $x(t) \in \mathbb{R}^n$, $y(t) \in \mathbb{R}^q$, $v_1(t) \in \mathbb{R}^n$, $v_2(t) \in \mathbb{R}^q$ and:

- v_1 (process noise) and v_2 (measurement noise) are white noises with zero mean value and known variance which are uncorrelated with each other [12]
- $A \in \mathbb{R}^{n \times n}$, $C \in \mathbb{R}^{q \times n}$ and the variance matrices of v_1 ($V_1 \in \mathbb{R}^{n \times n}$) and v_2 ($V_2 \in \mathbb{R}^{q \times q}$) are known.

Given that track/system data are collected up to N , the one step prediction is indicated by $\hat{x}(N+1|N)$, and can be visualized with the block diagram reported in Figure 8, where the top blocks are the system at study (the track in our case) and the bottom blocks, denoted by K are the Kalman filter:

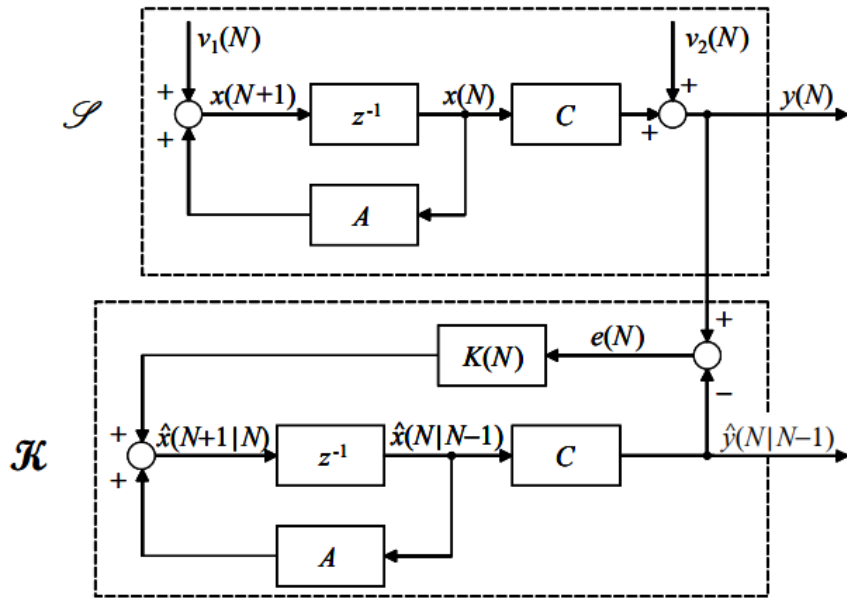


Figure 8: block diagram of the one step Kalman predictor. Source: Michele Taragna's Slides

A more popular version of the Kalman predictor is given by the predictor/corrector form (Figure 9). The form reported in Figure 9 is the one of a non-autonomous system, as the input u is present, since in our case the tracked objects can be considered as autonomous, it is sufficient to discard the u input.

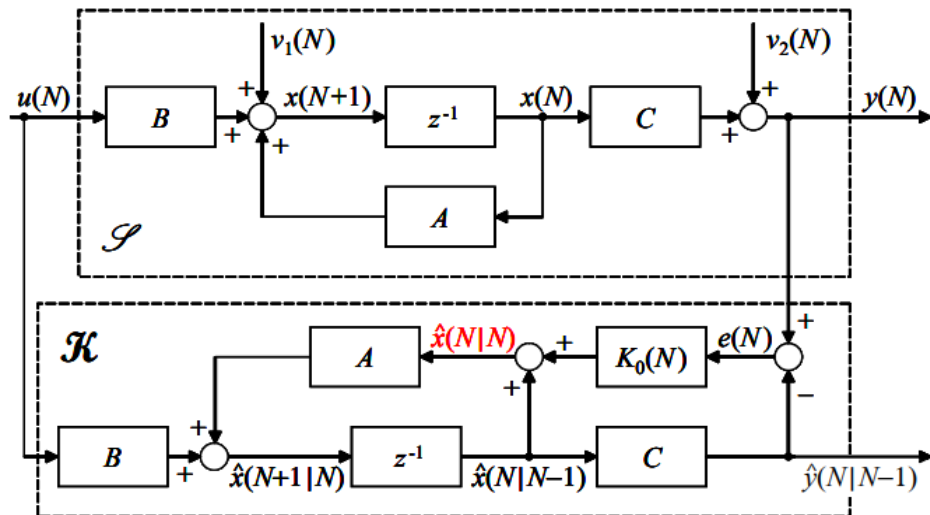


Figure 9: block diagram of the one step Kalman predictor in predictor/corrector form. Source: Michele Taragna's Slides

The predictor/corrector form of the Kalman filter contains the same prediction information as the standard one step Kalman filter (so, the $\hat{x}(N+1|N)$ prediction), plus the correction of the current measurement, $\hat{x}(N|N)$: this term, highlighted in red in Figure 9, uses both the prediction from the past $N-1$ measurements and the innovation contained in the $y(N)$ measurement in order to give a *corrected* estimate of the current state of the system S .

This gives the knowledge not only of the $N+1$ state (prediction) but also of the N state (correction), and is thus both a prediction and a filtering algorithm. This versatility makes it a quite popular implementation.

A common use scenario where the filtering action comes into use is when a GPS system loses signal for some time instants or when, due to some kind of issues, the measured position acquires an uncertainty higher than the one in the past measurements. The filtering action reduces the damages that could be made by following the new, more uncertain measurements.

In the use case presented here, the filtering action is useful to counteract the flickering effect that is often observed in most perception systems. We are referring to the effect where, passing from one frame to another, the clustered objects change much more than the physical objects do. This phenomenon is unavoidable, and is created by the relatively long chain of algorithms that lead from the Lidar data to the clustered bounding boxes; reminding the main steps, each Lidar frame has its ground removed using a plane fitting algorithm, has its objects clustered using a DBSCAN flavored clustering algorithm, then the clusters are represented by bounding boxes using L-Shape fitting; a small change in the output of one or more links of the chain and all the subsequent objects can vary considerably. For example, if the ground segmentation algorithm doesn't recognize a piece of ground as ground, two separate objects that are adjacent to that piece of ground will be grouped together and clustered as a single entity, modifying the list of objects, therefore the list of detections, having two tracks that will present no new detection and creating a new track due to the nature of the Hungarian algorithm.

This is just an example of the flickering effect that can be present due to small uncertainties and errors that get enhanced in the chain of algorithms.

Finally, a third form of the Kalman filter is presented, the multistep Kalman predictor (Figure 10). The working principle is the same as the one step Kalman predictor, but the time interval in which the system's state prediction is performed is not 1 but r . This can be referred to as a prediction interval, in which each object which is outputted by the Hungarian algorithm is predicted for r time instants.

This kind of prediction is quite useful in our use case: let's consider a case where only one object is detected, to which we can refer to as system k , and consider the set of r predictions performed by the multistep Kalman filter; this is equivalent to a set of oriented rectangles. The whole set of oriented rectangles can be added to the *dynamicCapsuleList* object that was introduced in Chapter 4.

From the point of view of the navigation algorithm, at the current time instant t , there will be r capsules present in the capsule list; the trajectory generator and the collision checker will therefore consider not only the current detected obstacle, but also the r predictions in the future. This will lead to choosing trajectories that will be feasible up to r time instants in the future. Of course, at the following time instant $t+1$ a new measurement of the detected object will bring new information on its system k and the set of predictions will change, but if the Kalman filter is working well and the object's motion doesn't change abruptly it is likely that the trajectory computed at time instant t will still be valid for some subsequent time instants.

This will allow not only for less computational resources to be used in the trajectory testing phase (and this advantage is arguably existent, since the Kalman filter itself presents non negligible computational cost), but will allow the autonomous vehicle to find itself in more cautious positions and orientation.

Also, this kind of approach is unavoidable to use in the case of particular maneuvers, such as the takeover maneuver, that require a certain amount of prediction of the environment, but are not in the scope of this thesis.

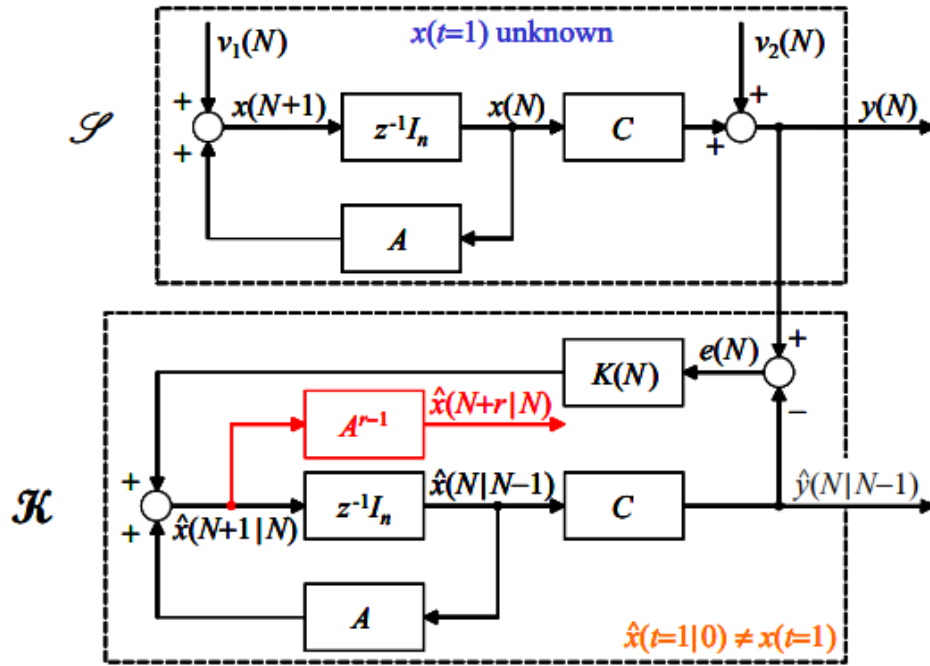


Figure 10: multistep Kalman predictor. Source: Michele Taragna's Slides

Another important aspect that is tackled by the combination of tracking and multistep Kalman prediction is the following: suppose the ego vehicle is moving along a collision free path at time t , and another external vehicle (which at time t is not on the reference path) is moving in a fashion that will intersect with the reference path at some time $t+c$. If the navigation algorithm lacks some sort of multistep prediction, the ego vehicle will keep on moving along a currently collision free path until the external vehicle will be directly on the path. This may mean two vehicles could suddenly find themselves at relatively close distance while both traveling at nonzero speed, and in this kind of situation braking or rapidly changing trajectory could be unfeasible: if a sudden obstacle appears in front of the ego vehicle while traveling at nonzero speed all the alternative trajectories computed in the Frenet space may be unfeasible, and lack of action may lead to accidents.

CHAPTER 5 - BIBLIOGRAPHY

- [1]: Fabio Faliero, NMPC Design For Autonomous Driving Applications, 2019
- [2]: Riccardo Scattolini, Introduction to MPC
- [3]: Siciliano, Robots modeling, planning and control, 2009
- [4]: [wikipedia.org,Frenet-Serret Formulas](https://en.wikipedia.org/wiki/Frenet-Serret_formulas)
- [5]: Peter Corke, summary of Robotics Trajectories
- [6]: Matlab, Highway Trajectory Planning Using Frenet Reference Path
- [7]: [wikipedia.org/Hungarian_algorithm](https://en.wikipedia.org/wiki/Hungarian_algorithm)
- [8]: <https://thinkautonomous.medium.com/computer-vision-for-tracking-8220759eee85>
- [9]: M. Taragna, slides on Estimation, Filtering and System Identification
- [10]: Joseph Redmon, Ali Farhadi,YOLOv3: An Incremental Improvement, 2018
- [11]: R.E. Kalman, A New Approach to Linear Filtering and Prediction Problems, 1960
- [12]: M. Taragna, Slides on Kalman Filter Theory, 2021

6. CONCLUSIONS

The exploratory work conducted in this thesis has analyzed some solutions involved in key aspects of autonomous vehicles. The application field of the vehicle is the agricultural one, and this has allowed for some simplifications, like the mostly-free working environment, which allows for a capsule oriented object representation, and offline path planning only done once before the starting of the mission.

About the RRT* algorithm, the possibility for including the maximum curvature constraint in the generation of the reference path allows the creation of a path that will be surely possible to track by the vehicle at study, at least from a cinematic side, eliminating one cause of tracking error from the big picture. The solutions provided by RRT* are often quite curvy and generally are not optimal in the sense of distance from the starting point to the objective area. In the main mission scenario considered (the handling of a hay ball from a defined starting point to a defined objective area) this aspect can be overlooked, especially if the vehicle is operating -as one of the starting hypothesis imposes- without human or animal presence around it; in the case of human presence, which is something that will have to be dealt with in the future implementation of autonomous driving for telescopic handler, the curved trajectory offered by RRT* could be difficult to predict and react to, and a less curvy solution should be looked for.

Another application in which this kind of geometric path is problematic is plowing, which is not the core mission of a telescopic handler but it resides in its possibilities.

Some companies solutions considered during the initial research phase of this thesis involve a semi autonomous or non autonomous offline path planning: the former being the selection of waypoints in the working area of the telescopic handler, which will be interpolated by polynomial of some order, the latter being the complete path drawing performed by the human operator; in both cases, cinematic feasibility of the solution and absence of obstacles along the path will be hand checked by the human operator before uploading the path to the machine. Although this procedure involves the expense of having a dedicated operator that provides navigation service, it can be ideal in the cases in which the exact path to be followed is important (e.g. plowing), or in which the passage in waypoints represents an important aspect of the mission.

About the sensors, the usage of a single Lidar sensor has appeared (although this hypothesis has not been validated by simulation due to lack of adequate hardware) insufficient; most autonomous vehicle reside on a variety of sensors, and use sensor fusion techniques to obtain detailed and robust information; as exposed in Chapter 2, the trend of later years it to reside more and more on camera sensors and machine learning or deep learning techniques, united with sensor that can actually measure distances such as Radar or Lidar.

Although insufficient by itself, the density and resolution of Lidar has been found to be sufficient to successfully perform clustering of obstacles most of the time, at least in the datasets used for the simulations.

The usage of the RTK GPS system has only been given as an hypothesis, and no simulations have been performed in this regard however its use for the considered case scenario seems adequate as the distances that can be covered by the RTK antennas (about 10-20 km) reside in the range of a medium to large farm.

The validity of Lidar clustering in agricultural datasets is yet to be tested due to the difficult availability of dedicated datasets, and should be the object of a followup work.

Integration with cameras and machine or deep learning techniques still seems to be a compulsory passage to be done in order to go towards the safety requirements cited in the early version of norms like the ISO 18497-3.

About the clustering algorithms, k-means has been analyzed and not implemented as the required number of clusters is a difficult aspect to deal with; elbow point selection of multiple different cluster number simulations has shown to be excessively costly and with no guarantees about the optimality of the solution.

The recently proposed FLIC (Fast Lidar Image Clustering) has been analyzed and implemented, both in its original flavor and in the 1-Map Connection variant. The results of the simulations have presented some success and some failures in clustering objects from the construction Lidar datasets, as covered in Chapter 3.

Part of the errors is to identify in the imprecision of the data about the angle of two adjacent Lidar measurements, since this angle is not an information that was available on the free datasheet of the sensor, and its value was only supposed based on similar models by Velodyne. The tests on the FLIC algorithm's clustering accuracy should be re-performed when having availability to a physical Lidar sensor or to detailed information about a sensor used in a dataset.

The execution frequency of FLIC was in the range of 5 to 10 Hz, varying from the complexity of the scene. The reasons for this variability are unknown, but an aspect to highlight is that, contrary to what was done by the authors of the FLIC paper, the implementation was written in Matlab and optimization issues could be at play. Also, since the raw azimuth, elevation, range data were not available in the dataset (the data was already given under the Point Cloud format), in order to be able to work with FLIC the cartesian data contained in the PointCloud object has been turn into azimuth, elevation, range by a custom function: this could be the source of slowness in the code execution. Again, tests using a real Lidar sensor should be performed to verify that the speed results (that lay in the order of 100 Hz) of the original paper are possible. In such a case, the implementation of FLIC in a follow up version of this project should be highly considered.

That said, with the elements available at the moment (those being the performed simulations) not much difference in speed has been observed between FLIC and Matlab's own `pcsegdist` function, which is based on DBSCAN; on the other hand, clustering accuracy performance appears to be in favor of the DBSCAN derived algorithm. The clustering speed is again in the order of 5 to 10 Hz using the same dual core i7 processor, and thus `pcsegdist` is to prefer to FLIC, at least with the current available elements of choice.

About the obstacle representation, the choice was between object list representation and grid based representation, and the former has been chosen. The reason for this choice is that the supposed environment is an open field one, with mostly free space and while the possibility of encountering an obstacle is contemplated, it is assumed that it is a rare one, and that the changes done to the a priori knowledge of the environment are limited, in the sense that the vehicle can always return on the reference path after avoiding the obstacle; the obstacle are also supposed to have a very definite shape and considerable size, that constitute an easily identifiable obstacle, such as a cow, a parked vehicle, a hayball.

Since the agricultural dataset was not available, a dataset which has similar layout in terms of identifiability of obstacles ecc. was a highway dataset present on the Matlab website. After performing some simulations, the capsule list representation was considered adequate for handling such an obstacle-sparse scenario. The performance of this kind of obstacle representation in the case of a cluttered environment is unknown, and should be the object of a followup work; an aspect to remind is that the convexity of the objects represented in an object list makes it easy to perform collision checking, using tools such as the SAT; the usage of grid based obstacle representation could be more computationally demanding.

About the navigation: for the alternative trajectory representation, the choice made was to use fixed alternatives in the Frenet space; this tool allowed us to define a fixed set of optimal states and then to produce a set of alternative trajectories from the current reference path (whether it be the actual “global” reference path or already an alternative path that is being followed due to the presence of an obstacle in a previous time instant), while keeping the computational cost associated with the trajectories generation very low, as it is simply the cost of interpolating a number N of polynomial of the 4th or 5th order in the space of states in \mathbb{R}^6 . Therefore, the solution for the trajectory to follow is to be searched in a finite number of N alternatives; this is opposed to the solution that is found in the case of MPC, as exposed in chapter 5.

In MPC, the solution of the problem is obtained by minimizing the cost function J , which is the function of a function and therefore it is a harder problem to solve with respect to what used in this work.

Moreover, the definition of the MPC cost function allows for taking into account more than just the terminal state deviation from a reference state, and allows for energy and point to point tracking considerations. The energy aspect is something that should be implemented in a future version of this work, while the point to point tracking cost should be implemented only in path-sensitive applications, like plowing. The handling of a hayball, which is the mission scenario considered here, does not require point to point tracking.

The solution provided by MPC is optimal, while the one found in our case has no guarantees of optimality, while being computationally much more efficient than MPC.

The solution provided by the code is a set of trajectories that are used for obstacle free navigation, and the vehicle model used is a very simplified one, as it is simply a cuboid.

The only physical constraint considered in this work, apart from the speed and acceleration ones which are common for most AV applications, is the maximum curvature constraint. In a follow up work, a complete or at least simplified cinematic and dynamic model of the telescopic handler should be used. The Merlo telescopic handler can move in three different driving configurations: two steering wheels, four steering wheels and crab movement. For each of these driving configurations a cinematic model should be derived; for the two steering wheels configuration the classic bicycle model could be used, while for the other two configurations custom models should be derived.

From the dynamics point of view, this work completely ignored all the dynamic aspects, and they should be inserted in the follow up work. The presence of both a cinematic and dynamic models, as well as the necessary modeling of the drivetrain and the actuators, suggest that a more model-oriented approach should be adopted.

MPC, in this regard, is a great candidate for it, as the kinematics and dynamics can be directly modeled into it, and the solution obtained could directly be the inputs to be provided to the actuators, whether they be eclectic motor torques, steering commands, ecc.

Lastly, the occupation provided by the telescopic arm has been simulated using a simple approach, with the Lidar sensor considered to be on top of the machine. These simulations have highlighted two aspects. First, the safest navigation condition is to travel with the arm at the lowest possible position. In this case, the occlusion created by it is the smallest one and objects in the medium and long distance are still in line of sight of the sensor. The case in which traveling needs to occur with the arm in a non - resting position brings us to the second point, which was already cited at the beginning of the conclusion: the usage of a single Lidar sensor is insufficient for the telescopic handler use cases. The arm constitutes an obstacle that blinds the Lidar of almost half of the plane, and does not allow for safe navigation, or even navigation at all, since turning on the right is impossible whilst the arm blocks the view. It seems therefore necessary to add perimetral sensors, perhaps Radar sensors, in order to fill the gap that can be left by occlusions.

7. APPENDIX

7.1 FLIC IMPLEMENTATION

7.1.1 main

```
%questa versione implementa il clustering robusto con 1MC
%questa versione implementa l'ignorare i clusters con un numero di elementi
%minore di Nmin
clear
close all
clc
%% 0. sensor specifications and minimum cluster number
% tipo di sensore
% velodyne HDL32E
va = 1.33;          %[deg] vertical angular resolution, specified in datasheet
ha = 0.2;           %[deg] horizontal angular resolution NOT FOUND PRECISE IN
DATASHEET
cos_vert = 2*cos(deg2rad(va));
cos_horiz = 2*cos(deg2rad(ha));
cos_vert2 = 2*cos(deg2rad(2*va));
cos_horiz2 = 2*cos(deg2rad(2*ha));
Dh = 0.8*0.8;
Dh2 = 4*Dh;
Dv = 0.8*0.8;
Dv2 = 4*Dv;
%minimum cluster number
Nmin=15;
%% 1. load data
fileName = 'lidarData_ConstructionRoad.pcap';
deviceModel = 'HDL32E';
veloReader = velodyneFileReader(fileName, deviceModel);
%% 1.bis create point cloud display
% Specify limits of point cloud display
xlimits = [-25 45]; % meters
ylimits = [-25 45];
zlimits = [-20 20];
%Create a pcplayer
lidarViewer = pcplayer(xlimits, ylimits, zlimits);
%Customize player axes labels
xlabel(lidarViewer.Axes, 'X (m)')
ylabel(lidarViewer.Axes, 'Y (m)')
zlabel(lidarViewer.Axes, 'Z (m)')
%% 2. frame by frame clustering
for frame = 1:200
    %1. simula il sensore che restituisce la Range matrix
    ptCloud = readFrame(veloReader);
    x = ptCloud.Location(:, :, 1);
    y = ptCloud.Location(:, :, 2);
    z = ptCloud.Location(:, :, 3);
    Range = xyz2range(x,y,z);
    %2. connection matrix and FLIC core
    BCM =
connection_matrix_1MC(Range,Dh,Dh2,Dv,Dv2,cos_horiz,cos_horiz2,cos_vert,cos_vert
2);
```

```

CC = bwconncomp(BCM,4);
%3. downsampling
lista_pixel_finale = {};
for idx = 1:CC.NumObjects
    if(size(CC.PixelIdxList{idx},1)>Nmin)
        %pixel --> ij
        lista_pixel = CC.PixelIdxList{idx};
        [a,b] = ind2sub(size(BCM),lista_pixel);
        %ij esteso --> ij downsampled
        lista_ij_downsampled_1 = a(((rem(a,2) == 1) + (rem(b,2)))==2);
        lista_ij_downsampled_2 = b(((rem(a,2) == 1) + (rem(b,2)))==2);
        %ij downsampled --> ij finale
        lista_ij_finale_1 = (lista_ij_downsampled_1+1)./2;
        lista_ij_finale_2 = (lista_ij_downsampled_2+1)./2;
        %ij finale --> pixel finale
        lista_pixel_finale{idx} =
sub2ind(size(Range),lista_ij_finale_1,lista_ij_finale_2);
    end
end
%rimuovo le celle vuote(quelle che contenevano SOLO un pixel pari)
lista_pixel_full=lista_pixel_finale(~cellfun('isempty',lista_pixel_finale));
%creating matrices for RGB (three different due to difficult pixel indexing)
%la dimensione cambia perché le letture lidar a volte cambiano di singoli
pixel
R_r = zeros(32,1088);
image_r = zeros(32,1088,3);
G_r = R_r;
B_r = R_r;

for idx=1:size(lista_pixel_full,2)
    R_r(lista_pixel_full{idx}) = rand;
    G_r(lista_pixel_full{idx}) = rand;
    B_r(lista_pixel_full{idx}) = rand;
end
image_r(:,:,1) = R_r;
image_r(:,:,2) = G_r;
image_r(:,:,3) = B_r;
figure(3)
imshow(image_r)
%uncomment per vedere sotto ai clusters anche il raw data
%imshow(Range,[])
%im=image('CData',image_r);
%im.AlphaData=max(image_r,[],3);

%update point cloud player
view(lidarViewer, ptCloud)
pause()
%    save('cluster_pixel','lista_pixel_full','Range');
%    return
end

```

7.1.2 connection matrix standard

```
function [BCM,H,HE] = connection_matrix(R,Dh,Dv,ch,cv)
```

```

%CONNECTION_MATRIX crea connection matrix di una matrice
% riceve in input una matrice mxn e ne restituisce una 2m-1x2n-1 con la
% connettivita in distanza Dh^2 (horizontal) e Dv^2 (vertical),
% per poter essere processata da un CCL algorithm 2d
% ch = 2*cosine horizontal angle between lidar cells
% cv = 2*cosine vertical angle between lidar cells
% importante la funzione riceve in input gia il cos moltiplicato x2 per
% evitare una moltiplicazione in piu
% importante: la distanza in input è quella al quadrato per evitare una
% operazione di radice
[m,n] = size(R);
BCM = 0;
% creazione horizontal connectivity matrix
H = zeros(m,n-1);
for j = 1:m
    for i = 1:n-1
        if R(j,i)*R(j,i) + R(j,i+1)*R(j,i+1) - ch * R(j,i) * R(j,i+1) < Dh
            H(j,i) = 1;
        end
    end
end
% creazione vertical connectivity matrix
V = zeros(m-1,n);
for j = 1:m-1
    for i = 1:n
        if R(j,i)*R(j,i) + R(j+1,i)*R(j+1,i) - cv * R(j,i) * R(j+1,i) < Dv
            V(j,i) = 1;
        end
    end
end
% creazione range binary
RB = ~isnan(R);
%creazione range binary extended
RBE = zeros(2*m-1,2*n-1);
RBE(1:2:end,1:2:end) = RB;
%creazione horizontal extended
HE = zeros(2*m-1,2*n-1);
HE(1:2:end,2:2:end) = H;
%creazione vertical extended;
VE = zeros(2*m-1,2*n-1);
VE(2:2:end,1:2:end) = V;
%creazione binary connection matrix
BCM = RBE+HE+VE;
end

```

7.1.3 connection matrix with one Map Connection

```

function [CM_1MC] = connection_matrix_1MC(R,Dh,Dh2,Dv,Dv2,ch,ch2,cv,cv2)
%CONNECTION_MATRIX crea connection matrix con 1 MC, sovrapponendo con un

```

```

%OR la conn matrix a un elemento di distanza con la connection matrix a 3
%elementi di distanza
%CAMBIAMENTI RISPETTO ALLE VERSIONI NON SPARSE E SPARSE: BISOGNA INSERIRE SIA
%2*COS(alfa) che 2*COS(2*alfa)
%E SIA D=(D)^2 CHE D = (2D)^2
% riceve in input una matrice mxn e ne restituisce una 2m-1x2n-1 con la
% connettivita in distanza Dh^2 (horizontal) e Dv^2 (vertical),
% per poter essere processata da un CCL algorithm 2d
% ch = 2*cosine horizontal angle between lidar cells
% cv = 2*cosine vertical angle between lidar cells
% importante la funzione riceve in input gia il cos moltiplicato x2 per
% evitare una moltiplicazione in piu
% importante: la distanza in input è quella al quadrato per evitare una
% operazione di radice
[m,n] = size(R);
%% 1. conn_mtx 1 elemento di distanza
% creazione horizontal connectivity matrix
H = zeros(m,n-1);
for j = 1:m
    for i = 1:n-1
        if R(j,i)*R(j,i) + R(j,i+1)*R(j,i+1) - ch * R(j,i) * R(j,i+1) < Dh
            H(j,i) = 1;
        end
    end
end
% creazione vertical connectivity matrix
V = zeros(m-1,n);
for j = 1:m-1
    for i = 1:n
        if R(j,i)*R(j,i) + R(j+1,i)*R(j+1,i) - cv * R(j,i) * R(j+1,i) < Dv
            V(j,i) = 1;
        end
    end
end
% creazione range binary
RB = ~isnan(R);
%creazione range binary extended
RBE = zeros(2*m-1,2*n-1);
RBE(1:2:end,1:2:end) = RB;
%creazione horizontal extended
HE = zeros(2*m-1,2*n-1);
HE(1:2:end,2:2:end) = H;
%creazione vertical extended;
VE = zeros(2*m-1,2*n-1);
VE(2:2:end,1:2:end) = V;
%creazione binary connection matrix
BCM = RBE+HE+VE;
%% 2. conn_mtx 3 elementi di distanza
%extended sparse range
ESR = zeros(2*m-1,2*n-1);
EH = ESR;
EV = ESR;
ESR(1:4:end,1:4:end) = R(1:2:end,1:2:end);
%binary extended sparse range

```



```

BESR = ESR;
BESR(isnan(BESR))=0;
BESR=(BESR>0);
%extended horizontal connectivity matrix
for i = 1:4:2*m-1
    for j = 1:4:(2*n-1)-4
        if ESR(i,j)*ESR(i,j) + ESR(i,j+4)*ESR(i,j+4) - ch2 * ESR(i,j) *
ESR(i,j+4) < Dh2
            EH(i,j+1:j+3) = 1;
        end
    end
end
%extended vertical connectivity matrix
for i = 1:4:(2*m-1)-4
    for j = 1:4:(2*n-1)
        if ESR(i,j)*ESR(i,j) + ESR(i+4,j)*ESR(i+4,j) - cv2 * ESR(i,j) *
ESR(i+4,j) < Dv2
            EV(i+1:i+3,j)=1;
        end
    end
end
%sparse connettivity matrix
SCM = BESR + EH + EV;
%% 3. somma OR delle due matrici
CM_1MC = BCM|SCM;

```

7.2 TELESCOPIC HANDLER OCCLUSION SIMULATION

This function receives a pointCloud object as input and returns a pointCloud object in which the points belonging to the cone occluded by the arm of the telescopic handler in the current position have been removed.

```

function pcout = simula_occlusione(currentLidar)
    roi = [-20 50 -10 10 -2 2];
    in = findPointsInROI(currentLidar,roi);
    currentLidar = select(currentLidar,in);
    z_lidar = 1.6;
    %% 1. define braccio as cuboid
    br_width = 0.8;
    br_length = 10;
    br_depth = br_width;
    braccio = cuboidModel([4,-br_width-0.1,2,br_length,br_width,br_depth,0,-10,0]);
    xlim([-10 10])
    ylim([-10 10])
    zlim([-1 10])
    %% 2. get corner points
    vertices = getCornerPoints(braccio);
    %seleziono solo i vertici interni(perchè gli altri non occludono)
    med_y = mean(vertices(:,2));
    vertices_int = vertices(vertices(:,2)>med_y,:);
    vertices_est = vertices(vertices(:,2)<med_y,:);
    %tra i vertici interni, seleziono vertici back e front, 1.5m è un buon
    discriminante perchè tutti i bracci

```

```

%al minimo sfilo sono più lunghi di 1.5m
ver_int_back=vertices_int(vertices_int(:,1)<1.5,:);
ver_int_front=vertices_int(vertices_int(:,1)>=1.5,:);
%tra i vertici esterni, seleziono vertici back e front, 1.5m è un buon
discriminante perchè tutti i bracci
%al minimo sfilo sono più lunghi di 1.5m
ver_est_back=vertices_est(vertices_est(:,1)<1.5,:);
ver_est_front=vertices_est(vertices_est(:,1)>=1.5,:);
%tra i vertici interni, seleziono i vertici up e down
med_bu = mean(ver_int_back(:,3));
med_fu = mean(ver_int_front(:,3));
ver_int_up(1,:) = ver_int_back(ver_int_back(:,3)>=med_bu,:);
ver_int_up(2,:) = ver_int_front(ver_int_front(:,3)>=med_fu,:);
ver_est_up(1,:) = ver_est_back(ver_est_back(:,3)>=med_bu,:);
ver_est_up(2,:) = ver_est_front(ver_est_front(:,3)>=med_fu,:);
ver_int_down(1,:) = ver_int_back(ver_int_back(:,3)<med_bu,:);
ver_int_down(2,:) = ver_int_front(ver_int_front(:,3)<med_fu,:);
ver_est_down(1,:) = ver_est_back(ver_int_back(:,3)<med_bu,:);
ver_est_down(2,:) = ver_est_front(ver_int_front(:,3)<med_fu,:);
%% 3. define planes between origin and vertices
%front plane
[af,bf,cf,df]=Plane_3Points([0,0,z_lidar]',ver_int_front(1,:)',ver_int_front(2,:)')';
[xf,zf] = meshgrid(0:0.1:45,0:0.01:3);
yf = 1/bf.*(-af.*xf-cf.*zf-df);
%back plane
[ab,bb,cb,db]=Plane_3Points([0,0,z_lidar]',ver_int_back(1,:)',ver_int_back(2,:)')';
[xb,zb] = meshgrid(0:-0.1:-10,0:0.01:3);
yb = 1/bb.*(-ab.*xb-cb.*zb-db);
%top plane
[at,bt,ct,dt]=Plane_3Points([0,0,z_lidar],ver_int_up(1,:),ver_int_up(2,:));
[xt,yt] = meshgrid(-5:0.1:40,-10:0.1:0);
zt = 1/ct.*(-at.*xt-bt.*yt-dt);
%% 4. remove points from point cloud
xpc = currentLidar.Location(:,1);
ypc = currentLidar.Location(:,2);
zpc = currentLidar.Location(:,3);
%removing right of front plane
for i = 1:size(xpc,1)
    if(ypc(i)<0&&xpc(i)>0)
        if(ypc(i)< 1/bf.*(-af.*xpc(i)-cf.*zpc(i)-df))
            xpc(i) = NaN;
            ypc(i) = NaN;
            zpc(i) = NaN;
        end
    end
end
%removing right of back plane
for i = 1:size(xpc,1)
    if(ypc(i)<0&&xpc(i)<0)
        if(ypc(i)< 1/bb.*(-ab.*xpc(i)-cb.*zpc(i)-db))
            xpc(i) = NaN;
            ypc(i) = NaN;
        end
    end
end

```

```

        zpc(i) = NaN;
    end
end
end
%% 5. visualize point cloud
hold off
outpc = pointCloud([xpc, ypc, zpc]);
pcshow(outpc);
plot(braccio), hold on
surf(xf, yf, zf, 'FaceColor', 'magenta', 'FaceAlpha', 0.3, 'EdgeColor', 'none');
surf(xb, yb, zb, 'FaceColor', 'magenta', 'FaceAlpha', 0.3, 'EdgeColor', 'none');
hold on
end

```

7.3 LOCAL NAVIGATION

This code performs ground removal using plane fitting, clustering using Matlab's `pcsegdist` based on DBSCAN, L-shape fitting using Matlab's cuboid fitting based on “Xiao Zhang et. al., Efficient L-Shape Fitting for Vehicle Detection Using Laser Scanners, 2017” and replanning as explained in Chapter 5. No GPS data is considered.

```

clear
close all
clc
% Set random seed to generate reproducible results.
S = rng(2018);
mappa2d=1;
%% OA.parameters and reference path
% Default car properties (in this version, these are the minimum obstacle
properties)
carLen    = 4.7; % in meters
carWidth  = 1.8; % in meters
rearAxleRatio = 0.25;
% Maximum obstacle properties
maxLen = 20; %in meters
%replanning parameter
replanRate = 2; % Hz
% Define the time intervals between current and planned states.
maxHorizon = 7; %in seconds
timeHorizons = 1:maxHorizon; % in seconds
% Define cost parameters.
latDevWeight    = 1;
timeWeight      = -1;
speedWeight     = 1;
% Reject trajectories that violate the following constraints.
maxAcceleration = 10; % in meters/second^2
maxCurvature    = 2; % 1/meters, or radians/meter
minVelocity      = 0; % in meters/second
% Desired velocity setpoint, used by the cruise control behavior and when
% evaluating the cost of trajectories.
speedLimit = 2; % in meters/second -> circa 7.2 kmh
%create path based on waypoints
waypoints = [0 0; ...

```

```

    50 -30];
refPath = referencePathFrenet(waypoints);
connector = trajectoryGeneratorFrenet(refPath);
figure(5)
show(refPath);
hold off
% Synchronize the simulator's update rate to match the trajectory generator's
% discretization interval.
sampleTime = connector.TimeResolution; % in seconds
%% 0B.initialization of caplist
%create capList for trajectory collision detection
capList = dynamicCapsuleList;
capList.MaxNumSteps = maxHorizon*10+1;
numSteps = capList.MaxNumSteps;
%create vehicle capsule
egoID = 1;
[egoID, egoGeom] = egoGeometry(capList,egoID);
egoGeom.Geometry.Length = carLen; % in meters
egoGeom.Geometry.Radius = carWidth/2; % in meters
egoGeom.Geometry.FixedTransform(1,end) = -carLen*rearAxleRatio; % in meters
updateEgoGeometry(capList,egoID,egoGeom);
%here you can choose maximum number of actors to keep track of -->
%considerarli con le dimensioni fisse è una forma di tracking
numActors = 10;
actorID = (1:numActors)';
actorGeom = repelem(egoGeom,numActors,1);
updateObstacleGeometry(capList,actorID,actorGeom);
models = cell(1,numActors);
%% 0C.Initialization of point cloud viewer
hCuboid = figure;
panel = uipanel('Parent',hCuboid,'BackgroundColor',[0 0 0]);
ax = axes('Parent',panel,'Color',[0 0 0]);
title('Fitting Bounding Boxes')
% Specify limits of point cloud display
xlimits = [-25 45]; % meters
ylimits = [-25 45];
zlimits = [-2 6];
colorLabels = [...
    0      0.4470 0.7410; ... % Unlabeled points, specified as [R,G,B]
    0.4660 0.6740 0.1880; ... % Ground points
    0.9290 0.6940 0.1250; ... % Ego points
    0.6350 0.0780 0.1840]; % Obstacle points
% Define indices for each label
colors.Unlabeled = 1;
colors.Ground    = 2;
colors.Ego       = 3;
colors.Obstacle  = 4;
%linehandles serve per visualizzare le varie traiettorie
lineHandles=[];
%this pcplayer is used if you want to see segmented ground points
% % Create a pcplayer
% lidarViewer = pcplayer(xlimits, ylimits, zlimits);
%
% % Customize player axes labels

```

```

% xlabel(lidarViewer.Axes, 'X (m)')
% ylabel(lidarViewer.Axes, 'Y (m)')
% zlabel(lidarViewer.Axes, 'Z (m)')
% Set the colormap
%colormap(lidarViewer.Axes, colorLabels)
%frame counter
frame=0;
%% 0D.load data
%if ~exist('lidarData','var')
    dataURL =
'https://ssd.mathworks.com/supportfiles/lidar/data/TrackVehiclesUsingLidarExampleData.zip';
    datasetFolder = fullfile(tempdir, 'LidarExampleDataset');
    %if ~exist(datasetFolder, 'dir')
        unzip(dataURL, datasetFolder);
    %end
    % Specify initial and final time for simulation.
    initTime = 0;
    finalTime = 35;
    [lidarData, imageData] =
loadLidarAndImageData(datasetFolder, initTime, finalTime);
%end
%%Looping through recorded data
for j=1:size(lidarData,1)
    currentLidar = lidarData{j};
    egoState = zeros(1,6); %no gps considered, only local
    %clear obstacle detections
    %% 1.generate trajectories
    %generate reference terminal state
    [termStatesref, timesSimple] = generate_terminal_simple(...
        speedLimit, 0, timeHorizons);
    %generate alternate trajectories (1 for sx, -1 for dx)
    [termStatesSx, timesSimple] = generate_terminal_simple(...
        speedLimit, 1, timeHorizons);
    %generate alternate trajectories (1 for sx, -1 for dx)
    [termStatesDx, timesSimple] = generate_terminal_simple(...
        speedLimit, -1, timeHorizons);
    %combine all terminal states and times
    allTS = [termStatesref; termStatesSx; termStatesDx];
    allDT = [timesSimple; timesSimple; timesSimple];
    numTS = [numel(allDT)];
    %Evaluate cost of all terminal states.
    costTS = evaluate_terminal_costs(allTS, allDT, speedLimit, ...
        speedWeight, timeWeight);
    % Generate trajectories.
    egoFrenetState = global2frenet(refPath, egoState);
    [~, globalTraj] = connect(connector, egoFrenetState, allTS, allDT);
    return
    % Eliminate trajectories that violate constraints.
    isValid =
evaluateTrajectory(globalTraj, maxAcceleration, maxCurvature, minVelocity);
    %% 2.separate ground points from obstacle points
    % 2.1 select data in region of interest
    roi = [-10 40 -6 6 -2 1];

```

```

in = findPointsInROI(currentLidar,roi);
ptCloudIn = select(currentLidar,in);
% 2.2 remove ego vehicle points
vehicledim = [-1 3 -1 1 -0.1 1.5];
ptCloudIn = remove_ego_points(ptCloudIn,vehicledim);
% 2.3 remove ground plane
maxDistance = 0.1;
%needed >0.3 for this specific dataset -->
% other addiotional constraints on ground segmentation and/or point cloud
% segmentation may be needed
referenceVector = [0 0 1];
[~,inliers,outliers] = pcfitplane(ptCloudIn,maxDistance,referenceVector);
ptCloudGround = select(ptCloudIn,inliers,'OutputSize','full');
ptCloudWithoutGround = select(ptCloudIn,outliers,'OutputSize','full');
scanSize = size(ptCloudIn.Location);
scanSize = scanSize(1:2);
% Initialize colormap
colormapValues = ones(scanSize, 'like', ptCloudIn.Location) *
colors.Unlabeled;
colormapValues(inliers,:) =
repmat(colorLabels(colors.Ground,:),size(inliers,1),1);
colormapValues(outliers,:) =
repmat(colorLabels(colors.Obstacle,:),size(outliers,1),1);
% Update view for ground points visualization
% view(lidarViewer, ptCloudIn.Location, colormapValues),hold on
%% 3. cluster points on euclidean distance, dividing left and right
distThreshold = 0.3; %minimum distance between two different clusters
%splitting point cloud in 2 per ridurre i problemi delle bounding boxes
%giganti
ptCloud1 =
select(ptCloudWithoutGround,ptCloudWithoutGround.Location(:,2)>0,'OutputSize','full');
ptCloud2 =
select(ptCloudWithoutGround,ptCloudWithoutGround.Location(:,2)<0,'OutputSize','full');
[labels1,numClusters1] =
pcsegdist(ptCloud1,distThreshold,'NumClusterPoints',50);
[labels2,numClusters2] =
pcsegdist(ptCloud2,distThreshold,'NumClusterPoints',50);
%perchè per ogni chiamata di pcsegdist i labels vengono resettati, facendo
così non si fa confusione
numClusters = numClusters1 + numClusters2;
labelColorIndex1 = labels1;
labelColorIndex2 = labels2;
pcshow(ptCloudIn.Location,labelColorIndex1,'Parent',ax);
pcshow(ptCloudIn.Location,labelColorIndex2,'Parent',ax);
% 4. fit cuboids
for i = 1:numClusters1
    idx = find(labels1 == i);
    models{i} = pcfitcuboid(ptCloudWithoutGround,idx);
    points1 = getCornerPoints(models{i});
    plot(models{i})
end
for i = numClusters1+1:numClusters

```

```

    idx = find(labels2 == i-numClusters1);
    models{i} = pcfitcuboid(ptCloudWithoutGround,idx);
    points2 = getCornerPoints(models{i});

    plot(models{i})
end
% plot ego vehicle model
drawcuboid('Position',[-carLen*rearAxleRatio,-
carWidth/2,0,carLen,carWidth,2],'Color','green');
for i = numClusters+1:numActors
    models{i} = [];
end
%% 5. trajectory replanning based on detections -->
% Update the collision checker with the current position
% of all actors in the scene.
for i = 1:numActors
    %states needed: x,y,yaw --> model parameters 1,2,9
    if i<=numClusters
        obstacleLength=models{i}.Parameters(4);
        obstacleWidth=models{i}.Parameters(5);
        yaw = models{i}.Parameters(9);
        if (yaw > 30||yaw<-30)
            temp = obstacleLength;
            obstacleLength = obstacleWidth;
            obstacleWidth = temp*1.4;
            yaw = 0; %safety condition
        end
        actorGeom(i).Geometry.Width = max(obstacleWidth,carWidth); %minimum
cluster condition
        actorGeom(i).Geometry.Length = max(carLen,obstacleLength);
        yaw = deg2rad(yaw);
        actorPoses(i).States = [models{i}.Parameters(1)-
cos(yaw)*obstacleLength/2+obstacleWidth/2,...
models{i}.Parameters(2)-sin(yaw)*obstacleLength/2,yaw];
    else
        actorPoses(i).States = [-10,-10,0];
        actorGeom(i).Geometry.Length = 1; % in meters
    end
end
updateObstacleGeometry(capList,actorID,actorGeom)
updateObstaclePose(capList,actorID,actorPoses);
% Determine evaluation order.
[cost, idx] = sort(costTS);
optimalTrajectory = [];
trajectoryEvaluation = nan(numel(isValid),1);
for i = 1:numel(idx)
    if isValid(idx(i))
        % Update capsule list with the ego object's candidate trajectory.
        egoPoses.States = globalTraj(idx(i)).Trajectory(:,1:3);
        updateEgoPose(capList,egoID,egoPoses);
        %UNCOMMENT QUA PER VEDERE LA CAPLIST DALL'ALTO, DURANTE IL
        %REPLANNING
        if (mappa2d)
            figure(2)

```

```

        show(capList, "TimeStep", [1:numSteps])
    end
    % Check for collisions.
    isColliding = checkCollision(capList);
    if all(~isColliding)
        % If no collisions are found, this is the optimal.
        % trajectory.
        trajectoryEvaluation(idx(i)) = 1;
        optimalTrajectory = globalTraj(idx(i)).Trajectory;
        break;
    else
        trajectoryEvaluation(idx(i)) = 0;
    end
end
end
for i = 1:numClusters
    plot(models{i})
end
% plot ego vehicle model
drawcuboid('Position', [-carLen*rearAxleRatio, -
carWidth/2, 0, carLen, carWidth, 2], 'Color', 'green');
for i = numClusters+1:numActors
    models{i} = [];
end
% Display the sampled trajectories.
lineHandles = [];
lineHandles =
exampleHelperVisualizeScene(lineHandles, globalTraj, isValid, trajectoryEvaluation)
;
    pause
end

```