Università degli Studi di Padova

Department of Information Engineering

Master's Degree in Automation Engineering

DEVELOPMENT OF PATH PLANNING ALGORITHMS FOR AGVs IN THE PRIMARY ALUMINIUM INDUSTRY

Supervisor
Prof. Angelo Cenedese

Co-supervisors
Eng. Mauro Schiavo
Dr. Nicola Lissandrini

Master's candidate
Matteo Boscarato

Academic Year 2018/2019

”Walking presupposes that
at every step the world changes in some of its aspects
and also that something changes in us.”

Italo Calvino

Acknowledgments

I thank my parents, who have offered me support and encouragement in these long five years: they have been fundamental for achieving this goal.

I thank my sister Valeria, her husband Teddy, their children Marta and Mattia for having cheered up my life of university student.

I thank my friends: from those I no longer see because of different choices of life paths, to those I continue to see and go out with on a daily basis. Your contribution was fundamental because it allowed me not to feel alone in all these years as a student.

I thank my university fellow students with whom I compared the solutions of the exercises, when I had doubts about them, or when I simply wanted to share university life with them. In particular, I thank Marco and Nicola, with whom I shared this last part of my university career: I wish you the best satisfactions in your career.

Finally, I thank Prof. Angelo Cenedese, for giving me this thesis opportunity and for his help in the drafting phase, and the company Techmo Car, in particular the Eng. Mauro Schiavo for having provided me with the necessary material to develop my thesis and for introducing me to working life in the company.

Abstract

This work, entitled *DEVELOPMENT OF PATH PLANNING ALGORITHMS FOR AGVs IN THE PRIMARY ALUMINIUM INDUSTRY*, has been carried out thanks to the collaboration between Techmo Car S.p.a. and University of Padua.
The project aims to evaluate different algorithms in the field of path planning, to automate the driving of a particular type of vehicle, *fluoride feeder vehicle*, in the aluminium industry environment.
To achieve this goal, a preliminary study was carried out concerning the world of the aluminium industry and the path planning algorithms existing in the literature. From these informations, an operational vehicle model and a scenario model have been created to be used for the simulation of different algorithms.
In particular, this thesis offers two types of algorithms for path planning: graph-based and sampling-based. The task of both is to provide a path that combines an arbitrary starting position with an arbitrary finish position, given any 2D map. The graph-based algorithms generate the path considering the map as a graph, while the sampling-based ones generate it by sampling various areas of the map. It is also considered a bonus algorithm that generates a path using wave fronts, according to a philosophy that does not belong to any of these categories. Performance evaluation is both qualitative and numerical. The quality of the algorithms is verified through graphical results of the simulations developed in the MATLAB environment while the overall length of the path, the execution time of the algorithm and steps between the starting and ending positions are considered as numerical data.
Two types of maps are used for the comparison of performance, an ideal and a real one, with the aim of demonstrating and verifying the feasibility and concreteness of the algorithms, opening then the space for a real implementation scenario.

Keywords: path planning algorithms, automated guided vehicle (AGV), aluminium industry, alumina feeder vehicle.

Contents

Listing of Figures

Listing of Tables

List of Terms

.dae File . . . . . 3D interchange file format used to exchange digital resources between multiple graphics programs as, for example, Blender and Inventor

.dwg File . . . . . databases of 2D or 3D drawings created with AutoCAD, the professional CAD software made available by Autodesk and used to create two-dimensional and/or three-dimensional drawings and projects

.urdf File . . . . . XML file format used in ROS to describe the components of the robot

Alumina . . . . . ceramic oxide of aluminium obtained from the refining of bauxite, fundamental in the industrial field due to its properties such as resistance to acids, high thermal conductivity and low electrical conductivity

Aluminium Fluoride . . . . . an important additive for the production of aluminium by electrolysis: together with cryolite, it lowers the melting point to below 1000°C and increases the conductivity of the solution

Automated Guided Vehicle . . . . . a portable robot that follows along marked long lines or wires on the floor, or uses radio waves, vision cameras, magnets, or lasers for navigation. They are most often used in industrial applications to transport heavy materials around a large industrial building, such as a factory or warehouse

Bauxite . . . . . sedimentary rock with a relatively high aluminium content and world's main source of aluminium. It consists mostly of the aluminium minerals: Gibbsite, Böhmite and Diaspore

Best First Search . . . . . type of search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule

Big-O Notation . . . . . mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity and also used to classify algorithms according to how their running time or space requirements grow as the input size grows

Binary Heap . . . . . tree data structure in which each node has at most two children, which are referred to as the left child and the right child

Bucket Sort Technique . . . . . a sorting technique that sorts the elements by first dividing the elements into several groups called *buckets*. The elements inside each bucket are sorted using any of the suitable sorting algorithms or recursively calling the same algorithm

Böhmite . . . . . an aluminium oxide hydroxide mineral and component of the aluminium ore bauxite

Campaign Plan . . . . . small portion of the earth's surface, taken as a reference for the performance of certain specific functions, essentially topographical, agricultural or related to the construction of buildings

Casthouse . . . . . building containing furnaces to which the aluminium is carried after being extracted from potcells

Caustic Soda . . . . . inorganic compound widely used as reagent in the chemical industry

Chlorofluorocarbons . . . . . chemical compounds containing chlorine, fluorine and carbon. Characterized by high chemical and thermal stability which increases with the fluorine content, they are non-flammable and not very toxic

Cryolite . . . . . a fluoride of sodium and of aluminium: in the molten state, it is the main component of the electrolytic cells because it lowers the melting temperature, making the reactivity of the systems faster

Deadlock . . . . . computer term which indicates a situation in which two or more processes or actions block each other, waiting for one to perform a certain action that serves the other and vice versa

Diaspore . . . . . an aluminium oxide-hydroxide mineral and component of the aluminium ore bauxite

Fibonacci Heap . . . . . data structure for priority queue operations, consisting of a collection of heap-ordered trees

Gazebo . . . . . A 3D graphical visualization environment for ROS

Gibbsite . . . . . one of the mineral forms of aluminium hydroxide and component of the aluminium ore bauxite

Gradient Descent Technique . . . . . a first-order iterative optimization algorithm that uses the negative of the gradient of the function at the current point for finding the minimum of a function

Incremental Search Algorithm . . . . . algorithm which reuses information from previous searches to speed up the current search and solves search problems potentially much faster than solving them repeatedly from scratch

Informed Search Algorithm . . . . . algorithm which solves the search problem: to retrieve information stored within some data structure or calculated in the search space of a problem domain. The *informed* feature stands for avaibility of heuristic information

Ladle . . . . . container used to contain molten material. Normally of 10-14t capacity, it is closed with a lid and equipped with a tube

Polycyclic Aromatic Hydrocarbons . . . hydrocarbons (organic compounds containing only carbon and hydrogen) that are composed of multiple aromatic rings. In general, they are pollutants that generate alert because some compounds have been identified as carcinogens and mutagens

Polygon Mesh . . . . . collection of vertices, edges and faces that defines the shape of a polyhedral object in 3D computer graphics and solid modeling

Potcell . . . . . rectangular steel shell lined with carbon where electrolytic process transforms refined alumina into aluminium

Potline . . . . . a serie of connected cells where the cathode of the previous cell is connected to the anode of the next cell downstream

Potroom . . . . . long building based on pillars, so raised from Campaign Plan, in which floor potlines are disposed

Random Geometric Graph . . . . . an undirected graph constructed by randomly placing N nodes in some metric space (according to a specified probability distribution) and connecting two nodes by a link if and only if their distance is in a given range

Self-Balancing Binary Search Tree . . . . node-based binary search tree that automatically keeps small its height (maximal number of levels below the root) in the face of arbitrary item insertions and deletions

Siphoning	phenomenon in which a flow overruns upwards, passing over the upper surface of a vessel by exploiting the potential gravitational energy and the principle of communicating vessels
Venturi Ejector	system that exploits the Venturi effect, i.e. the pressure of a fluid current increases with decreasing speed, to aspirate a given fluid
WBK method	method for finding approximate solutions to linear differential equations with spatially varying coefficients

List of Acronyms

$\alpha-AlO(OH)$	Diaspore
$Al(OH)_3$	Aluminium Trihydroxite
$Al(OH)_3$	Gibbsite
AlF_3	Aluminium Fluoride
AlF_3	Flux
$AlO(OH)$	Böhmite
Al_2O_3	Alumina
CF_4	Tetrafluoromethane
CO_2	Carbon Dioxide
CO	Carbon Monoxide
C_2F_6	Hexafluoroethane
FeO	Iron Oxide
HF	Hydrogen Fluoride
NaF	Sodium Fluoride
$NaOH$	Caustic Soda
Na_3AlF_6	Cryolite
SiO_2	Silica
TiO_2	Titanium Dioxide
AFSH	Aluminium Fluoride Storage and Handling

AGV . . . . . Automated Guided Vehicle

Al . . . . . Aluminium

ATV . . . . . Anode Transport Vehicle

BF . . . . . Baking Furnace

BIT* . . . . . Batch Informed Tree

CFCs . . . . . Chlorofluorocarbons

CTTV . . . . . Crucibile Transport and Tilting Vehicle

CTV . . . . . Crucibile Transport Vehicle

FMM . . . . . Fast Marching Method

GAP . . . . . Green Anode Plant

GGR . . . . . Random Geometric Graph

Na . . . . . Sodium

OSRF . . . . . Open Source Robotics Foundation

PAHs . . . . . Polycyclic Aromatic Hydrocarbons

PTM . . . . . Pot Tending Machine

R&D . . . . . Research & Develpment

ROS . . . . . Robot Operative System

RRT . . . . . Rapidly-exploring Random Tree

RS . . . . . Rodding Shop

SAIL . . . . . Sanford Artificial Intelligence Laboratory

WBK . . . . . Wentzel–Kramers–Brillouin

1 Introduction

In this chapter a brief insight of the state of art of the fields related to path planning and control is presented: in fact, a small introduction on this topic and the exposition of following works and concepts are necessary for a better comprehension of the proposed work. It continues by introducing the problem of path planning in a formal way, followed by the proposed solutions. Finally, it ends with the outline of the thesis.

1.1 State of Art

Owing to their mobility characteristics, Automated Guided Vehicle (AGV) can operate in a larger workspace and explore uncertain environments. In uncertain/cluttered operating environments, the AGV has no prior information/knowledge about the workspace. As it operates in uncertain/cluttered environments, it demands higher level of intelligence than other mobile robots.
Some of the challenges facing autonomous path planning in cluttered environment are highlighted in the literature as:

- limited online replanning algorithm.
- kinematic and dynamic constraints.
- limited sensor data about the environment.
- high computational time.
- the incomplete information about the obstacles.

In the last few years, path planning for AGVs is one of the main focus in the field of autonomous control environments. Since AGVs are widely used in many industrial and military applications, several researchers have been working on different methods to enhance the intelligence of the vehicle for different applications. Path planning has been considered as the main challenge in the field of autonomous mobile system. The fundamental requirement for the path planning is to plan the best possible collision free path from a start position to a goal position. For successful and complete path planning in cluttered environment, the AGV has to replan its path quickly in such a way to avoid any obstacles if necessary, until it completes the assigned task. There are several classes of algorithms that can solve this problem. Worthy of note are the classes:

- Graph-based.
- Artificial Potential fields.
- Visibility Graph.
- Reward-based.
- Wave Fronts.
- Sampling-based.

At the first class belongs A*, a heuristic search algorithm widely used for path-finding. It uses heuristic knowledge to search and solve to find the goal much faster than the uninformed search methods. A* is not optimised for path finding cases in dynamic environment and it can be adapted to dynamic environment by restarting once a change in the environment is encountered.
Incremental heuristic search algorithms reuse search trees from previous searches to speed up the current search and thus often find cost-minimal paths for series of similar search problems faster than by solving each search problem from scratch, which is important in domains that are only incompletely known or change dynamically. Some existing re-planning algorithms, such as D*, D* Lite Focused D* and Lifelong Planning A* (LPA*) are A* variant. LPA* is an improvement on A* that efficiently updates only the changed paths in a dynamic environment while D* Lite is based heavily on the backward LPA*. Time and space complexity are always considered with respect to some measure of the problem difficulty.
An algorithm with a different approach for path planning, related to the penultimate class, is the FMM. It is a numerical method for solving the Eikonal equation: from a practical point of view, it produces wave fronts starting from an initial position of the scenario. Then, using the gradient descent technique, it is possible to arrive at the origin from any existing position: the final result is the path that unites the initial position with the final one.
Belonging to the last class, there are for example RRT, PRM and BIT* which have proved

extremely advantageous for every type of robot. These algorithms are based on the stochastic sampling of the simulation scenario, thus creating a network of nodes and arcs. All these algorithms are fundamentally structured in two phases: the first phase consists in creating a finite number of random nodes in the scenario and in the connection of them through collision-free arcs according to logics that vary according to the particular algorithm. In the second phase the algorithm calculates the optimal path starting from a start node and a goal node. In some cases the goal node can be replaced by a goal region, or a portion of the space within which the robot must reach the end of the path.
In general, mathematical path-planning techniques assume that an AGV is a point mass which can move in any direction and is able to flawlessly reach a specified goal, with most studies using a simplified kinematic constraint to plan a feasible path. However, for real-world scenarios, these assumptions may not be valid. In order to develop an effective path-planning technique, knowledge of the relevant AGV's dynamics is very important. To model a real AGV's behaviour, the equations of motion of AGVs which rely mainly on physical characteristics, such as the vehicle and actuator specifications, are considered. Normally, AGVs are characterised by non-integrable kinematic constraints which can not be eliminated from the model equations, while the kinematic model enables understanding of the manoeuvrability properties and analysis of the behaviour of the vehicle. On the other hand, the dynamic model accounts for the reaction forces and describes the relationship between the motions and generalised forces acting on the vehicle.
Most of the researchers have been working on various techniques to represent or rebuild the environment that has both static and dynamic obstacles. Generally it is a challenging and difficult task to plan an optimal path in the environment that has unknown dynamic obstacles.
In the recent past, the different image processing approaches have emerged in order to detect, track and estimate the dynamic obstacles behaviour. Basically, a scene flow is the 3D motion field of points in the world while an optical flow is the 2D motion field of points in an image. In the computer vision community, differential optical flow methods can be classified as local and global. The former involves optimisation of a local energy and the latter determines the flow vector through minimisation of a global energy. Local methods offer robustness to noise but lack the ability to produce dense optical flow fields while global techniques produce 100% dense flow fields but are much more sensitive to noise.

1.2 Problem Context

The purpose of the aluminium industry environment is to produce aluminium from raw ore. It has several areas, each of them dedicated to very specific tasks, and different equipment, such as cranes, vehicles and machinery, which enable these tasks to be carried out.
Vehicles suitable for aluminium processing are supplied by third-party companies, such as Techmo Car S.p.A., an Italian company specialized in the production of high-end mobile and fixed equipment for the production of aluminium and metal.

One of Techmo's new goals is to make any produced vehicle an AGV: new market demands and the development of technology in the autonomous driving sector have led the company to consider this new type of vehicle.
To achieve this goal, Techmo sought collaboration with the University of Padua, particularly with the field of automation engineering.
From this collaboration, three theses are born with the task of studying and providing solutions on this precise case of autonomous guidance: a thesis (this one) focused on path planning, a second thesis focused on kinematic control and finally a third thesis focused on camera and laser vision of the vehicle.
Among the various processing areas, these thesis cover a particular area called *potroom*, in which aluminium is melted; while among the operating vehicles, they focus on the *fluoride feeder vehicle*, one of the many vehicles supplied by Techmo, which is responsible for bringing aluminium fluoride into the potroom. Therefore, all three theses share the same chapter concerning the design of the vehicle model.

1.3 Proposed Solutions

The solution proposed by this thesis in order to start making an autonomous industrial vehicle is to use graph-based or sampling-based algorithms for vehicle path planning. The basic idea is to use a 2D plan, which includes potroom and external environment, and approximate the vehicle to a simple 2D material point. In this way, it is much easier to focus on the problem of path planning.
In this context the effectiveness of different algorithms is evaluated: A*, Dijkstra's, D* Lite, Focused D* for the first category of algorithms, while RRT, RRT* and BIT* for the second category. Also the bonus algorithm FMM is evaluated, belonging at the wave fronts category.
These algorithms, following different approaches, provide an ideal path to the output, so only viable from a material point, and/or a real path, which can therefore also be travelled by a real vehicle.
The advantages of these algorithms are convenience, flexibility and robustness: the algorithms have different potentialities and demonstrate excellent results in different scenarios. Some algorithms, such as D*, allow path planning even in semi-known environments, i.e. scenarios in which the plant is known but there are mobile obstacles. This is an usual case in the aluminium industry (fixed scenario) where vehicles and workers move continuously (mobile obstacles).
A further advantage derives from the fact that the algorithms can be interfaced with an external vision system (cameras) and therefore they can improve their performance and become more usable in different situations and environments.
Other further advantages will be presented in the continuation of the thesis.

1.4 Thesis Outline

The thesis is organized as follows.
Chapter 2 is used to introduce the area of interest of the thesis on autonomous vehicle technology. In particular, the aluminium industry and the Techmo Car company are presented, with particular attention to the processing areas and involved vehicles.
Chapter 3 focuses on the most theoretical aspects of this thesis, presenting a review of the literature related to path planning algorithms.
In chapter 4, the software used to perform AGV simulations and the general procedure for obtaining a simulation model of vehicle starting from a real vehicle are presented. In order to make the simulation as coherent as possible, a testing scenario similar to the aluminium industry is created.
Chapter 5 shows the results of the tests and the analysis of the algorithms performances, considering different types of scenarios where the path planning takes place and various characteristic parameters of the algorithms. Path length, time taken by the algorithm and path shape are the features used to make a comparison.
In chapter 6 the final conclusions and possible future works are reported.

2
Context Analysis

This chapter provides an analysis of the primary aluminium industry, followed by Techmo Car company presentation. The process of aluminium processing is presented, with particular attention to the processes used and the various processing areas. Subsequently, the company Techmo Car is introduced, highlighting its role in the aluminium industry: the interest is directed to designed vehicles, from which is obtained the AGV, the main theme of this thesis.

2.1 Aluminium Production Process

Aluminium is the most abundant metallic element in the Earth's crust. It occurs in many minerals: its primary commercial source is Bauxite, a mixture of hydrated aluminium oxides and compounds of other elements such as iron.
In literature, the type of production leads to distinguish the aluminium in *primary* or *secondary*: the primary one is produced starting from the mineral bauxite, instead the secondary one is produced from re-melting of the aluminium scrap.

2.2 Primary Aluminium

Primary production is the process through which new aluminium is made: it regards the steps by which alumina is smelted to pure aluminium metal.
In the following subsections, these steps are discussed in order to understand the production cycle of aluminium.

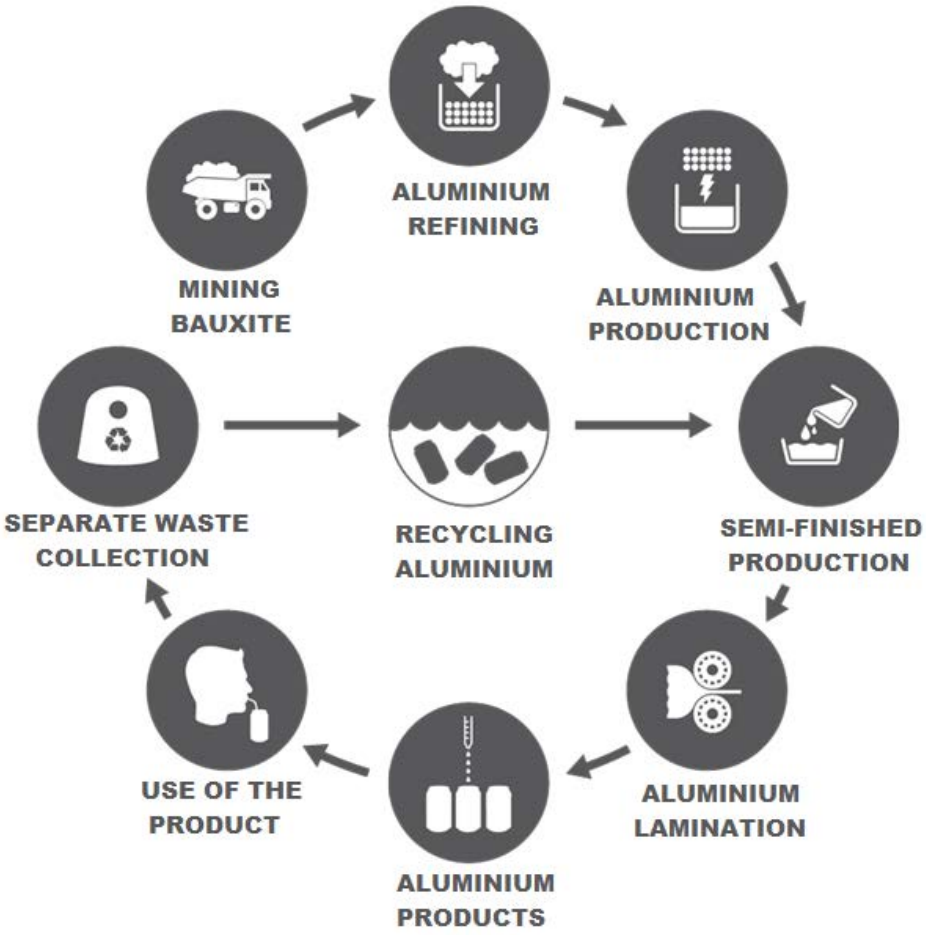


Figure 2.1: The life cycle of aluminium.

2.2.1 Mining Bauxite Ore

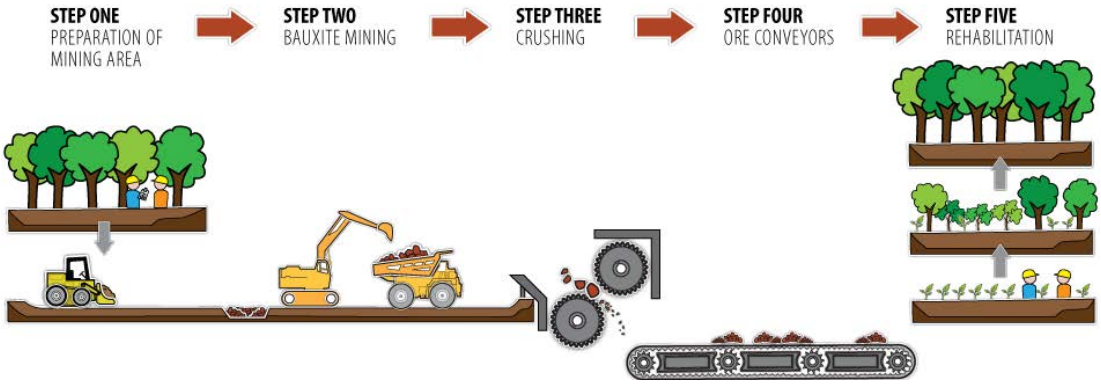


Figure 2.2: Typical bauxite mining process.

The first step of aluminium production is the mining of bauxite. The bauxite is a very common mineral and comes in the form of granular or rocky clay of various colours (pink, red, brown and grey). The name derives from Les Baux, a French town in the Pyrenees where it was first identified.
More than one hundred millions tons of bauxite are mined each year. The largest deposits are typically found in equatorial zones: Australia, Central and South America (Jamaica, Brazil, Suriname, Venezuela, Guyana), Africa (Guinea), Asia (India and China), Russia, Kazakhstan and Europe (Greece).
The material is mainly extracted in open-cast mines, in the upper layer of the ground (from 4m to 6m) and transported to crushing or washing plants to remove the overburden of several meters of rock and clay. Subsequently, it is moved to the alumina refineries.

2.2.2 Alumina Refining

Despite bauxite is crushed or washed to remove unwanted materials, it still contains inside a large number of impurities that should be removed in the refining step, to avoid the contamination of the metal during the smelting process. In refinery, from bauxite is extracted Alumina: a white powder necessary to produce primary aluminium.
The most commonly used method for alumina refining is the Bayer process [Appendix A] and it involves principally four steps: *digestion*, *clarification*, *precipitation* and *calcination*.
The ore is first ground and mixed with a hot solution of lime and caustic soda. The mixture is then pumped into high-pressure containers and heated. Then, a separation process, that dissolves the aluminium oxide by a caustic soda, results in a clarified dissolved alumina. This alumina is pumped into precipitators and aluminium crystals are added to hasten the process of crystal separation. The crystals attract other crystals and form agglomerates, which

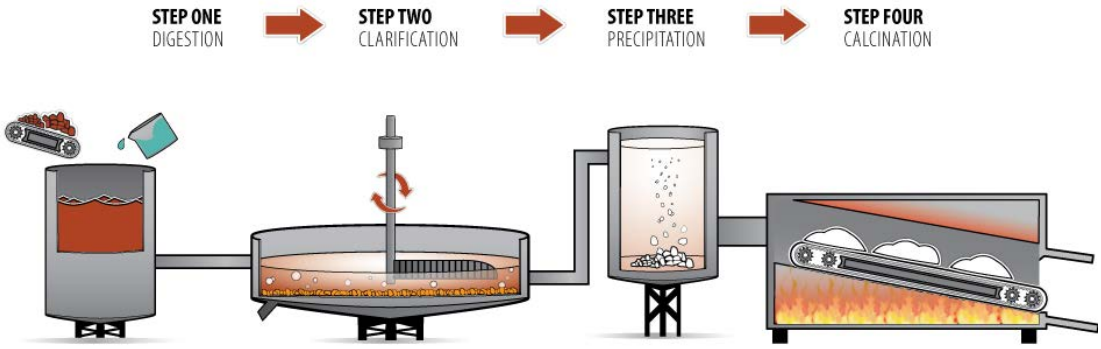


Figure 2.3: Alumina refining process.

are filtered, washed and calcined in rotary kilns or stationary fluidized-bed flash calciners at high temperatures. A dry and fine white powder of pure alumina is the result.[5]

2.2.3 Aluminium Smelting

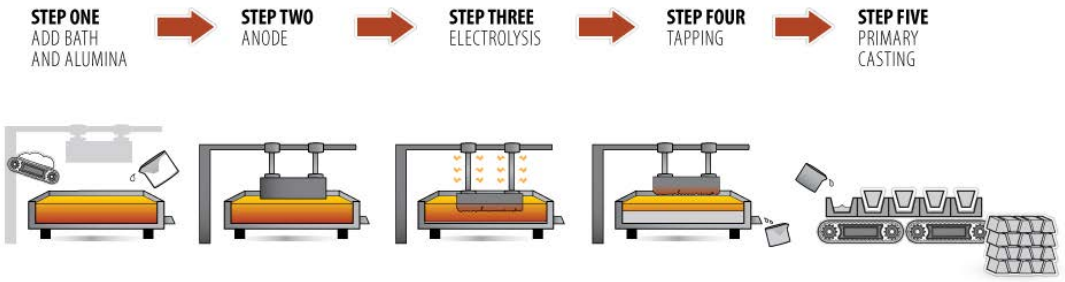


Figure 2.4: Aluminium smelting process.

Aluminium smelting is the process of extracting aluminium from alumina generally by the Hall-Héroult process.
In a modern smelter, alumina is dissolved in electrolytic-cells, *Potcells*, that are filled with a mixture of sodium, aluminium, and cryolite fluorine.[Figure 2.5] An aluminium smelter usually consists of hundreds of potcells arranged in lines, *Potlines*, divided in turn into rooms, *Potrooms*.
The Hall-Héroult smelting process [Appendix B] is the most widely used approach for smelting alumina to produce aluminium. The process requires anodes and cathodes, which are mostly made of carbon. The container accommodating the *bath*, molten solution of cryolite and alumina, is shaped as the cathode while the anode block is lowered into the bath from

above. A direct current is passed through the bath and as the aluminium ions discharge their electrical load at the cathode, the liquid metal collects at the bottom of the electrolytic-cell.

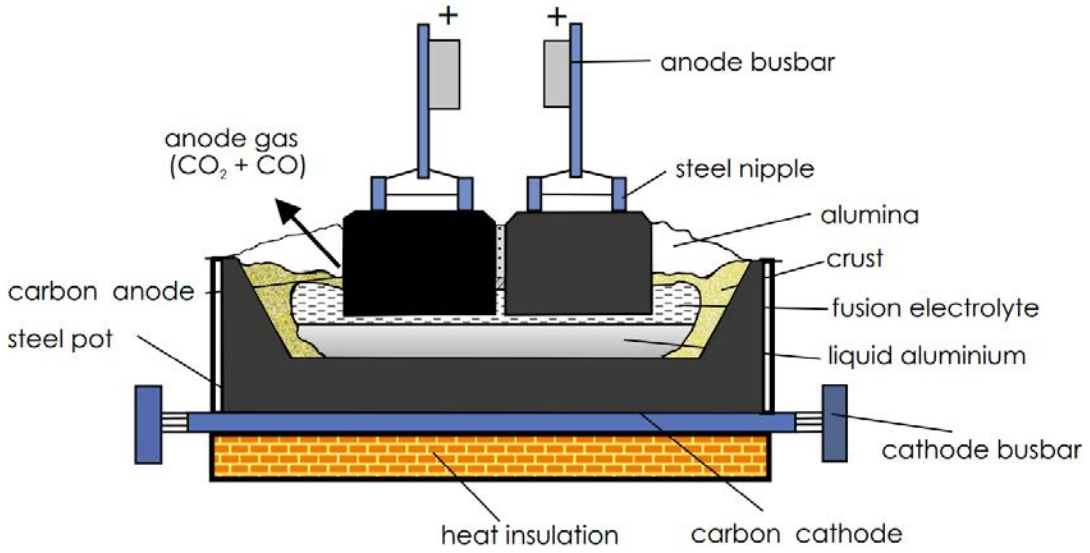


Figure 2.5: Modern electrolytic-cell for electrolysis of aluminium oxide.

Typically, the set of anodes in a cell is arranged such that only one anode must be repositioned daily. During the electrolytic process, the anode burns down to a residue providing just enough carbon to cover pins and cast iron nippel. There should always be a minimum percentage of the anode left on the bar from which it is suspended because otherwise the aluminium bath can be contaminated with metal from the bar. The used anode can be used in the construction of new anodes.
While anodes are consumed very fast, the cathodes are consumed slowly and need to be replenished only on a long-term basis.

2.2.4 Tapping and Pouring Processes

The aluminium smelting process proceeds by periodically siphoning liquid aluminium from the potcells into large crucibles by means of a vacuum vessel. The liquid substance is poured into a crucible and transported to the Casthouse where it is further processed. The activities from siphoning off the molten aluminium until placing the filled crucible in the main road are comprised in the so-called *tapping process*. Instead, the activities from placing the filled crucible in the main road until metal pouring into *holding furnaces* are comprised in the so-called *pouring process*.

2.2.5 Aluminium Fabrication

After being transported to the casthouse, the melt aluminium located inside the holding furnace can be combined with other aluminium coming from the merger of waste. This merger happens in *melting furnaces* that, as a principle, are identical to holding ones and it is made through the heat provided by the combustion of *burners* (particular gas). Subsequently, it is passed through a metal treatment system, which removes impurities such as hydrogen, metallic contaminations, and mechanical contaminations. The fluxed aluminium is then cast into shapes as specified to specific wishes of customers.
After a cooling period, the aluminium products are transferred to storage or prepared for shipment.

2.2.6 Semi-Fished Production

Depending on the desired product mix, further fabrication may include forging, casting, rolling, drawing or extruding to create different semi-finished products. In detail, the techniques for transforming molten aluminium into products for commercial use are varied and divided into:[2]

- *Drawing*: special type of cold rolling in which the starting piece is passed through a mold that gives it its final shape. It is used to produce some small diameter wires and tubes, called *drawn tubes*, used in applications ranging from aeronautics to kitchen items.

- *Continuous and semi-continuous casting*: technique for transforming molten aluminium into wire rods and subsequently, through the *drawing*, are made wires, slabs, plates or billets for further processing.

- *Lamination*: process of reducing the thickness of aluminium and can be "hot" or "cold". With "hot" rolling, a pre-heated ingot passes through the rollers, undergoing a reduction in thickness and an increase in length at each step. "Cold" rolling returns hardness and allows the desired thickness to be reached. Combined lamination processes can produce an aluminium sheet having a thickness of only 0.004mm. Normally aluminium foils have a variable thickness ranging from 0.20 to 3mm. Aluminium foil is a widely used packaging due to its impermeability to air, light and humidity, which allows the storage and transport of perishable goods.

- *Extrusion*: process in which a pre-heated aluminium billet passes through a steel matrix transforming its original volume into that of a long profiles with a constant section. Extrusion is used for the production of extruded shapes such as windows, doors, coverings, pipes, structures for trailers, railway carriages, air planes and ships.

- *Casting of Fusion*: the aluminium is melted in various forms and cast in special molds. Sand mold casting is used for small series or very complicated pieces. The automotive industry makes extensive use of cast aluminium parts such as pistons, manifolds, pumps etc.
- *Forging*: the final shape is obtained by compressing a hot metal tablet in a special mold. It is a technique to produce pieces with variable weight (from 50g to 100kg).
- *Impact Extrusion*: it is a combination of extrusion and forging. The most famous product made with this technique is the flexible tube used to hold shaving cream, toothpaste, cosmetics, etc.

2.3 Secondary Aluminium

Secondary production is the process through which aluminium is recycled from products arrived at the end of their life cycle (*old scrap*) or from the remelting of scrap from the processing of the same aluminium industry (*new scrap*).

2.3.1 Recycling Aluminium

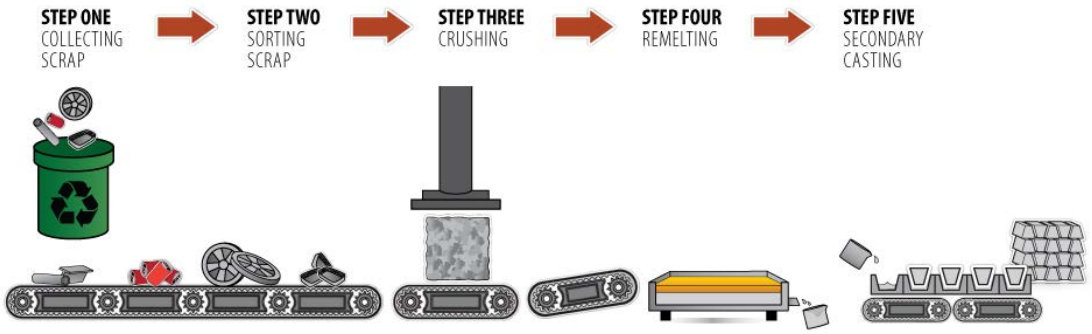


Figure 2.6: Aluminium recycling process.

The production of secondary aluminium consists of the recycling of aluminium scrap or processing waste, to obtain aluminium called *secondary* or *second fusion*.
Thanks to its total recyclability, almost a third of the aluminium used today is produced by recycling scrap without losing its quality and with consequent economic and environmental advantages for both industry and consumers. In fact, secondary aluminium production not only requires less energy (aluminium recycling requires only 5% of the energy used to produce primary aluminium) but also reduces waste and greenhouse gas emissions.
After selection, the baled aluminium scrap is sent for recycling in the foundry. In general,

the casthouse equipment used in primary smelters and secondary smelters. Here the material is subjected to a *de-slagging treatment*: it is pre-treated at about 500°C in rotary kilns, for being purged of paints or other adherent substances. Leaving the kiln after slagging, the hot aluminium shreds are transported to recycling kilns where they are melted at 800 °C until liquid aluminium is obtained, which is transformed into ingots or billets for the production of semi-finished products or new products.[3]

2.4 Typical Layout of Processing Plant

The production of primary aluminium takes place in large production lines containing hundreds of electrolytic-cells and a number of different areas, common in aluminium smelters. The smelter includes the following areas: potrooms, casthouse, carbon anode shop and others areas.[Figure 2.7]

2.4.1 Potrooms

The plant layout of a primary aluminium smelter is characterized by the typical extended parallel buildings, accommodating the electrolytic-cells, called potcells. A cell comprises rectangular steel shells lined with carbon that are filled with a mixture of sodium, aluminium, and cryolite fluorine. The cells may be lined up *end-to-end* [Figure 2.8a] or *side-by-side* [Figure 2.8b] in one or more parallel lines down the center of the potroom:

- *End-to-end layout*: it presents corridors to the two long sides of every row of cells, used for the transit of the vehicles in the miscellaneous operations. Then, for reason of layout compaction, it switched to side-by-side layout in which there is only a corridor on the short side of cells.
- *Side-by-side layout*: in this configuration, the use of ground machines was not expected because every operation had to occur through an equipped crane called Pot Tending Machine (PTM). Nevertheless, the use of some machines also spread for this type of layout, due to high cost of cranes and their failure vulnerability (a small breakage in the crane leads to whole replacement).

Each cell is reachable from two opposite sides in both layout and contains a fixed number of anode places on each side (usually, between 14 and 32 in total). A smelter roughly contains around 300 to 700 potcells, covering a total length of approximately 1km. A different number of anodes may be used per cell. An electric current is passed through the suspended anodes and cathodes in the potcells. Cells are electrically connected in series (the cathode of a cell is connected to the anode of the next cell downstream). A series of connected cells is called potline. As the cells are lined up in series, a connection between two cells caused by for instance a vehicle or a human, can lead to a short circuit (or even worse accidents). During the facility design phase, the consequences of cell alignments should be taken into account:

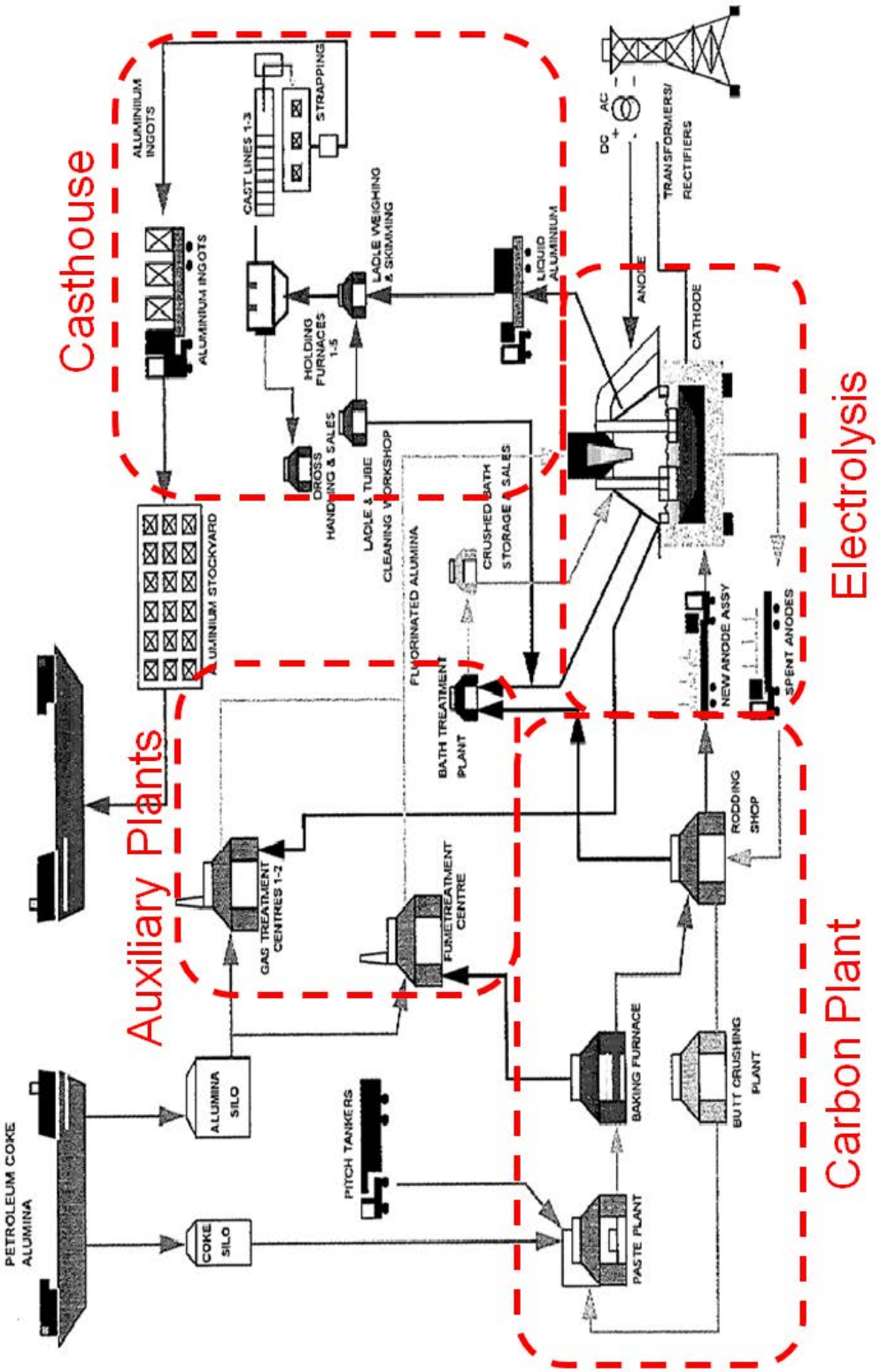


Figure 2.7: Example of aluminium plant layout.

roughly, every five years a major overhaul of a potcell takes place. During this process, the potcell will be emptied and cleaned. Usually, the potcell is completely removed from the potroom and revised elsewhere.

(a) End-to-end layout. (b) Side-by-side layout.

Figure 2.8: Layouts of system cells.

2.4.2 Casthouse

The resultant aluminium of the electrolysis process is transported from the potrooms to the casting area in crucibles. From thereon, alloys could be added, aluminium could be poured into holding furnaces prior to casting or the aluminium could be directly cast into ingots. Figure 2.9 sketches a typical casthouse in a primary aluminium smelter.[5]

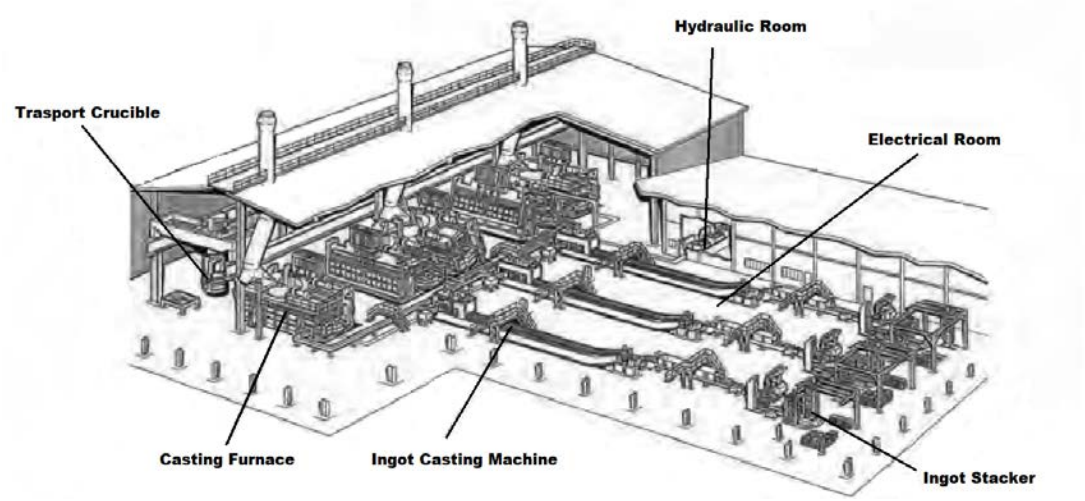


Figure 2.9: Casthouse layout for primary aluminium smelter.

2.4.3 Carbon Anode Plant

The carbon anode plant occupies a large area in the primary aluminium smelter and basically handles the manufacturing of anode blocks that go into the electrolytic-cells. Materials, such as calcined petroleum coke and pitch, are passed through various production steps before they are finally baked to form the anode block. The carbon anode plant comprises the zones: Green Anode Plant (GAP), Baking Furnace (BF), and Rodding Shop (RS).
In the GAP, the so-called green anodes are manufactured: petrol coke is crushed and ground into the required grain size distribution while used anodes are recycled by crushing and making up a coarse fraction. All fractions are mixed, adding liquid coal tar pitch as a binder. During the forming, the paste is shaped and compacted to the required size and density, after which the green anodes are obtained.
The green anodes are sent to the BF where they are calcined at a high temperature. The baking process consists of a pre-heating, heating, and cooling process. The time of baking differs from plant to plant and a baking cycle of 24 to 28 hours is required in the present production process. After the cooling period has elapsed, the block is transported to the RS.
In the rodding area, a metal rod is applied to the baked anode. The rod allows both the anode to be suspended in the potcell and the power may flow through it. *Butts* (consumed anodes) arrive in the rodding shop after which the rods are stripped of the anode block. The metal rod is prepared for re-use and the used anode is returned to the green carbon area for preprocessing. Usually, the whole carbon anode plant is capable to process all anodes required for the potlines. In large smelters, this could be more than a thousand per day.[5]

2.4.4 Other Areas

Besides the production facilities and dedicated areas as described previously, there are a number of other areas that could be identified in primary aluminium smelters. For instance, storage areas for alumina, petrol coke, green and bakes anodes, baking furnace, and rodding shop. Also, the aluminium produced and the final products are stored in dedicated areas. Finally, there is parking places dedicated to backup AGVs. Another process going on in potrooms is the filling of buckets with bath materials. After a certain amount of anode changes, this bucket is full and need to be brought to a bath cooling conveyor where it is emptied. A maintenance and administrative department complete the plant.

2.5 Techmo Car Introduction

The company Techmo Car or, simply, Techmo is a world leader in the engineering and production of high-end "tailor made" mobile and stationary equipment for the aluminium industry. In particular, areas of expertise are the primary aluminium industry in which alumina is smelted to pure aluminium metal, and the secondary aluminium industry in which aluminium scrap is recycled into aluminium that can be used again. The special designed custom-made vehicles are in operation all over the world at various kind of industries and

Figure 2.10: The Techmo Car company headquarters.

the machines are designed to perform, for example, in hot smelter environments with high magnetic fields and in the primary aluminium industry where space is often an issue.
The company was founded in 1961 by Dr. Franco Zannini and since the beginning was focused on providing original and state of the art solutions to problems related to metal production. Techmo was the first company in the world to conceive and fabricate specialized vehicles for the aluminium electrolysis in the 60's of the past century. Nowadays Techmo products are appreciated by the most demanding aluminium producers, adopting every kind of smelter reduction technologies, located in more than 40 countries. Each step of their projects is carried out through a specific engineering analysis of the production requirements. Techmo R&D programs focus on the following multiple objectives:[17]

- Improving all the environmental aspects: better comfort and safer working conditions for the operators, reduce carbon footprint of our equipment, increase recyclability of our components.
- Bringing higher efficiency to the production process while reducing production costs, running costs, and maintenance costs.
- Increasing durability in the years.

Techmo is currently investigating possibilities to strengthen their competitive position by gaining access to the emerging technology of AGVs. This rising technology can complement their current assets and capabilities and therefore lead to new market opportunities and additional after sales services. Techmo has already started some research to apply this technology in the aluminium industry. To arrive at innovative and specific solutions for their area of development, the company relied on the University of Padova: as will be seen in the following

chapters, the thesis will try to propose solutions to the problem of autonomous driving of vehicles operating in the aluminium industry.

2.6 Operative Vehicles

In addition to human work, the production of aluminium works thanks to the use of specific vehicles, without which some activities could not be carried out or would require a much longer operating time (unthinkable for the aluminium industry).
A number of vehicles and related substitutes can be identified in an aluminium smelter: depending on the work area, they can be distinguished in potroom and casthouse vehicles.
In the following subsection, the two categories of vehicles will be analysed in their characteristics and in the tasks they perform.

2.6.1 Potroom Vehicles

These vehicles concern the aluminium processing when the pure aluminium in the molten state is deposited above the cathode (in the bottom of the cell) and it is extracted by aspirating it into a ladle through an appropriate tube (*tapping tube*). In the ladle, the vacuum is created through a vacuum line of the factory or through a Venturi Ejector. Aluminium, once the ladle has been filled, is carried into the casthouse for next processing.

End-to-End Potroom Vehicles

- *Anode Changers*: provides for the replacement of the consumed anodes adjusting the distance between anode and cathode and then replacing the anode.[Figure 2.11a] According to the type of equipment, an anode changer can arrive to carry out the following functions:

    1. break the crust around the anode.
    2. grab the anodic rod and open the locking mechanism of anode on bus-bar link.
    3. remove the anode and pose it on *pallet* of exhausted anodes.
    4. clean the anodic cavity bringing on board the waste through a *bucket*.
    5. take a new anode from pallet of new anodes.
    6. fix the new anode to the right height.

- *Crustbreakers*: break the crust, which is mainly formed on the long side of the cell, to encourage the removal of the anode. They are equipped with a hydraulic hammer that is positioned in a controlled way by the operator. It can also break the crust on the short side of the cell or between anode-to-anode.[Figure 2.11b]

- *Alumina Feeders*: the task of this vehicle is to fill the cell tanks of alumina. It is added to the potcell at frequent intervals by point feeders to keep the alumina content at a constant level while metal is separated.
  These machines have a container of 7 or 9m^3 which is filled to fall from a silo with a bellows inlet. In the potroom, the operator pairs the final part of cochlea, of which the vehicle is endowed, with the loading "mouth" of cell tank.
  The feeders can be also equipped with an impact hammer required to break the crust that forms on the surface of the bath before alumina is charged: in other words, it can substitute the crustbreaker.[Figure 2.11c]

- *Anode Covering Vehicles*: are used to cover the part of the cell, that would be uncovered after the breakage of the crust or the change of the anode, of milled bath (recycling material from the grinding of anodic remains). The covering with milling bath has mainly mechanical function: it thermally insulates the fused part from the atmosphere and protects it from oxidation.[Figure 2.11d]

(a) Multipose anode changer.

(b) Crustbreaking vehicle.

(c) Alumina feeder vehicle.

(d) Anode covering vehicle.

Figure 2.11: Vehicle for end-to-end layout potroom.

Side-by-Side Potroom Vehicles

All these vehicles perform functions typically of PTM competence. For the reasons cited previously in the introduction of side-by-side potroom, many smelters choose to remove some PTM functions, leaving only the essential ones, i.e. the tasks that can not be performed by such vehicles.

- *Fluoride Feeders*: are very similar for appearance and function to alumina feeders. The mainly difference is that, in this case, aluminium fluoride is managed instead of alumina.[Figure 2.12a]
- *Taphole Breaking and Covering Vehicles*: are used to open, through frontal hydraulic hammer, a hole in the crust on short side of cell, to allow the withdrawal of aluminium or melted bath through tapping. The hole is then re-closed pouring some milled bath.[Figure 2.12b]
- *Bath Tapping Vehicles*: withdraw the cryolite bath from a cell and reverse it in another, similarly to molten metal that is taken by a vacuum ladle to be transferred to another cell. This operation needs to balance the bath amount between cells in order to create uniform conditions, encouraging the control of electrical conduction.[Figure 2.12c]

(a) Fluoride feeder vehicle.

(b) Taphole breaking and covering vehicle.

(c) Bath tapping vehicle.

Figure 2.12: Vehicles for side-by-side layout potroom.

2.6.2 Potroom-to-Casthouse Vehicles

This type of vehicles concerns the passage of the aluminium from potroom to casthouse. In this subsection, reference will be made to the tapping operations (removal of metal from the cell) and pouring operations (pouring of metal into the furnace): depending on the solutions adopted, the ladle hanging on a PTM can get metal from the cell and later can be moved through vehicle, or can be entirely managed by vehicle.

Withdrawal through Ladle from PTM or Crane

After being hung on the PTM [Figure 2.13a], the ladle tube is inserted in the potcell with the beak at the height of the aluminium molten and it is created the vacuum inside the ladle thought a line of empty belonging to the factory or through a compressed air line that feeds a Venturi Ejector placed on the ladle lid. The created vacuum taps the aluminium from the inside of ladle and its quantity is controlled by load cells. When the ladle is full, tapping operation is stopped.
Subsequently, the ladle is posed on a carriage driven by a tractor or posed on a stand in the entrance hall of potroom and withdrawn by a Crucibile Transport Vehicle (CTV) (or Crucibile Transport and Tilting Vehicle (CTTV), in the case of equipment with ladle tilter) that transfers it in the casthouse and empties it inside the holding furnace. This latter operation can be done in different way:

- Ladle transported by CTV or carriage $\longrightarrow$ emptying through Siphoning [Figure 2.13b] or pouring through a fixed position of tilting.
- Ladle transported by CTTV $\longrightarrow$ emptying through pouring of ladle directly from vehicle.[Figure 2.13c]

(a) Tapping operation by PTM with hanging ladle.

(b) Operation of shiphoning.

(c) Operation of tilting pouring.

Figure 2.13: Operation of tilting pouring.

Withdrawal through Tapping Truck

In this case, the PTM is absent and the ladle is transported by appropriated vehicle, the *Tapping Truck* [Figure 2.14], which deals with tapping, transferring and pouring: the machine has often on board a compressor to provide the compressed air needed to tapping and, sometimes, also to pouring through specific pipes or channel called *Launder*.

Figure 2.14: Tapping truck.

2.6.3 Casthouse Vehicles

As the smelting process is continuous, additional materials need to be added to the potcell periodically.

- *Skimming Machines*: are provided of a long telescope with a rake at the end and are used to remove the superficial waste of the metal. They also stick the metal and clean the bottom and the walls of furnace.[Figure 2.15a]
- *Scarps Loaders*: are used to load the furnaces with the waste aluminium. They can use boxes, forks or other loading devices. Moreover, they can be used also as skimming machines.[Figure 2.15b]

(a) Skimming vehicle.

(b) Scarps loading vehicle.

Figure 2.15: Casthouse vehicles.

2.6.4 Bonus Vehicle: Anode Transport Vehicle

This type of vehicle does not belong properly to casthouse or potroom, therefore it is decided to discuss in apart subsection.
The Anode Transport Vehicle (ATV) [Figure 2.16] is responsible for transporting anode pallets. Anode pallets may be empty or contain a certain amount of either new or used anodes. Pallets with new anodes are transported from the rodding shop to nearby the cell, while empty pallets or pallets with burned anodes are transported from nearby the cell to the rodding shop.
The pickup of pallets should be done by executing a special backward driving maneuver because physical vehicle properties prohibit the ATV to pick-up pallets by forwards driving.

(a) Front view.

(b) Rear view.

Figure 2.16: Anode transport vehicle.

2.6.5 Moveable Objects

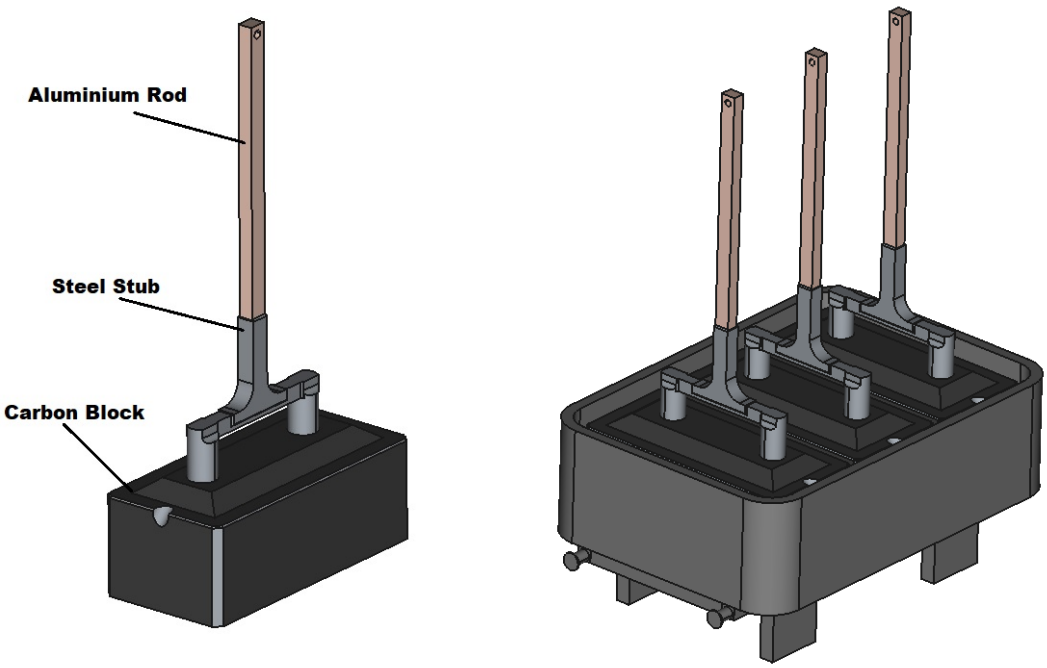
Moveable objects that are transported throughout the production facility include, for example, anode pallets and crucibles. Below we discuss them more detailed.

Anodes and Anode Pallets

The anode [Figure 2.17a] is formed by a block of carbon and an aluminium rod, both joined by a steel stub. The block is immersed in the bath of the cell while the rod puts the block in contact with the electrical system so that, with the passage of electric current, electrolysis occurs.
The transfer of the new/old anodes occurs through the pallet [Figure 2.17b]: a container suitable for posing and transporting of anode. The use of the pallet is necessary to speed up the replacement of the anodes: its rapid consumption in the electrolytic cells requires a continuous replacement cycle. To improve the replacement, it is dedicated a particular vehicle previously discussed: the ATV.
During anode pallet transport, the pallet can be driven in according the required orientation by letting the transporter drive forward/backward in a section. For safety reasons (to avoid electrical short cut) and to enable drive through by other traffic, pallets can not be placed anywhere, but in dedicated plant as anode carbon one.

Crucibles

A crucible, also known as ladle, [Figure 2.18] is a large metal bucket for transport of liquid aluminium. Crucibles are transported by means of CTV or CTTV and could be empty, filled or partly filled. A crucible is completely filled with metal after a certain amount of tapped potcells. A lid is placed on the crucible before it is being transported.



(a) Fresh anode. (b) Anode pallet with fresh anodes.

Figure 2.17: Example of anodes and anode pallets objects.

Figure 2.18: Example of ladle object.

2.6.6 The Enemies of Vehicles

The potroom and casthouse environments are particularly hostile for the vehicles which must operate in difficult conditions. The enemies are:

- Temperature: depends on climate zone. In hot locations can be reached 50-60°C with peaks of 80-90°C in proximity of the furnaces and cells, due to the thermal radiation issued. In the casthouse, especially, the radiation is extremely intense that can fuse the plastic parts of vehicle not sufficiently protected.
- Magnetic field [only in potroom]: is very intensive with peaks of 0.08T. Normally, it is higher near the floor and near the upper end of potlines where converge all the conductors. This phenomenon influences all the parts with ferromagnetic components and can cause undesired actuations or locks with consequences also disastrous. Therefore, some precautions are adopted:
  - use as little as possible components with ferromagnetic materials.
  - protect the ferromagnetic parts with steel plates of important thickness: although they do not shield the field, they can deviate it.
  - pose the sensible part in upper end of the vehicle so that the chassis works as protection.
  - avoid the entrance in potroom to the people with pacemaker.
  - keep out all sensible materials to magnetic field (smartphones, credit cards and clocks).
- Electric energy [only in potroom]: there exist some elements continuously under voltage. Although the potential difference between anode and cathode is about 4.5-5 V, the cells are linked connected in series. It can happen that a vehicle accidentally touches two cells and generates a very high potential difference: the machines acts like a resistor and becomes subjected to a very high Joule effect, leading to the fire the machine with possible damages to the operator.
  To avoid this situation, every vehicle has an electrical insulation consisting on plates of insulating material that are interposed between chassis and the equipment parts entering in the potcell. Insulations for chassis and cockpit can be added, too. However, no vehicle is perfectly insulated due to many factors as humidity, breakages and poor maintenance.
- Possible explosions [only in casthouse]: it may happen that the water, contained in the scarps, dissociate in hydrogen and oxygen when the scarps are released in the liquid mass at high temperature, causing explosion.

- Powders: the alumina and fluoride powders are very thin and hard and tend to penetrate in every aperture. So, all the air intakes of the machine are very critical parts. In particular, the engine aspiration must be sealed and the seal leaked is the main cause of engine destruction. Moreover, these powders settle on mechanical parts as cylinder stems in phase of extraction, so the adoption of efficient seals or bellows is extremely important.

- Gas: they are not a very important problem for the vehicles, except for fluorine gas [only in potroom] that stick on glass machine. So the glass must be protected with films or made with sheets of polycarbonate.

- Drivers: are one of the principal problems of vehicles. Since they directly manage the vehicle, a bad utilization leads to serious damages. From here, the need to design AGV.

3
Literature Review

In this chapter, the literature review of the various algorithms used for AGV navigation is presented. The navigation algorithms are divided into two types: graph-based and sampling-based algorithms. In particular, we analyze the algorithms of Dijkstra, A* and D* for the first type, instead RRT, RRT* and BIT* for the second type. In the end, it is presented a bonus algorithm that is totally different from the two previous approaches. For each algorithm, the pseudocode, time complexity and considerations on the advantages/disadvantages introduced are presented.

3.1 Graph-Based Algorithms

Given a graph, an abstract data type composed by pairs of vertices/nodes V and edges E, the operations performed can be categorised as elementary or high level operations.
Elementary operations of graphs add and delete a vertex or an edge or find adjacent and neighbours of a vertex. Instead, high level operations performed on graphs find degree of graph, the connectivity between nodes using clustering coefficient algorithms or the shortest path between nodes using the particular algorithms. The key passage, which makes these algorithms popular in the path planning literature, is to consider a scenario as a graph, whose equations indicate an obstacle or free path.
This section deals about this last type of algorithms, focusing on the shortest path between start and target nodes.

3.1.1 Dijkstra's Algorithm

The Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.
The algorithm exists in many variants: Dijkstra's original algorithm finds the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree. Specifically, Dijkstra's selects the path that minimizes the following quantity:

$$f(n) = g(n) \tag{3.1}$$

where n is the next node on the path and $g(n)$ is the cost of the path from the start node to n.
It can also be used for finding the shortest path from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road (for simplicity, ignoring red lights, stop signs, toll roads and other obstructions), Dijkstra's algorithm can be used to find the shortest route between one city and all other cities.

Algorithm

Let the node from which we start be called the *initial node*. Let *the distance of node y* be the distance from the initial node to y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step:

1. Mark all nodes unvisited and create a set of all the unvisited nodes called the *unvisited set*.

2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as *current*.

3. For the current node, consider all of its unvisited neighbours and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.

4. When all the unvisited neighbours of the current node have been considered, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.

5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal: it occurs when there is no connection

between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.

6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Algorithm 3.1 Dijkstra's Algorithm

```
1:  procedure MAIN(Graph, source)
2:      for ∀ node v in Graph                                  ▷ Initialization
3:          dist[v] = ∞
4:          previous[v] = undefined
5:      dist[source] = 0                                       ▷ Distance from source to source
6:      Q = the set of all nodes in Graph
7:      while Q is not empty
8:          u = node in Q with smallest dist[ ]
9:          remove u from Q
10:         for ∀ neighbour v of u
11:             alt = dist[u] + dist.between(u,v)
12:             if alt < dist[v]                               ▷ relax (u,v)
13:                 dist[v] = alt
14:                 previous[v] = u
15:     return dist[ ], previous[ ]                            ▷ distances/paths from source to all nodes
```

When planning a route, it is actually not necessary to wait until the destination node is "visited" as above: the algorithm can stop once the destination node has the smallest tentative distance among all "unvisited" nodes (and thus could be selected as the next "current"). In this case, the algorithm stops at the line 9 if *u = destination* and the shortest path between source-destination is given by:

```
1: S = empty sequence
2: u = target
3: if prev[u] is defined or u = source
4:     while u is defined
5:         insert u at the beginning of S
6:         u = prev[u]
```

Time Complexity

Denoting the number of edges as |E| and the number of vertices as |V| and using Big-O Notation, the time complexity of Dijkstra's algorithm can be expressed as a function of edges E and vertices V of the graph. For dense graph, i.e. |E|=O(|V|^2), the simplest implementation of Dijkstra's algorithm stores the vertex set Q as an ordinary linked list or array, and extracting minimum is simply a linear search through all vertices in Q: in this case, the time complexity is O(|E|+|V|^2) = O(|V|^2).
If the implementation stores the graph as an adjacency list, the time complexity is $\Theta(|E| \log |V|)$ = $\Theta(|V|^2 \log |V|)$. For sparse graph, i.e. |E|=O(|V|), Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a Self-Balancing Binary Search Tree, Binary Heap, or Fibonacci Heap as a priority queue to implement extracting minimum efficiently. To perform the decrease of key steps in a binary heap efficiently, it is necessary to use an auxiliary data structure that maps each vertex to its position in the heap, and to keep this structure up to date as the priority queue Q changes. With a self-balancing binary search tree or binary heap, the algorithm requires $\Theta((|E| + |V|) \log |V|)$ time in the worst case; instead, for connected graphs, this time bound can be simplified to $\Theta(|E| \log |V|)$.

3.1.2 A*

The A* algorithm is an algorithm published in 1968 by the computer scientists of Stanford Research Institute: Peter Hart, Nils Nilsson and Bertram Raphael. It can be seen as an extension of Dijkstra's algorithm because A* improves the performance by using heuristics to guide its search. It is widely used in path finding and graph traversal, which is the process of visiting each vertex of the graph. It enjoys widespread use due to its performance and accuracy.

Algorithm

A* algorithm is an Informed Search Algorithm and a Best First Search, i.e. it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.
At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes:

$$f(n) = g(n) + h(n) \tag{3.2}$$

where n is the next node on the path, $g(n)$ is the cost of the path from the start node to *n*, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from *n* to the goal.
A* terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended.
The heuristic function is specific of the problem: if the heuristic function is admissible, i.e. it never overestimates the actual cost to get to the goal, A* is guaranteed to return a least-cost path from start to goal.
Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the *open set* or *fringe*. At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and g values of its neighbours are updated accordingly and these neighbours are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). The f value of the goal is then the cost of the shortest path, since h at the goal is zero in an admissible heuristic.
The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.
As an example, when searching for the shortest route on a map, $h(x)$ might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points.
If the heuristic $h(x)$ satisfies the additional condition $h(x) \leq d(x, y) + h(y)$ for every edge (x,y) of the graph (where d denotes the length of that edge), then $h(x)$ is called *monotone* or *consistent*. With a consistent heuristic, A* is guaranteed to find an optimal path without processing any node more than once.
It is easy to note that A* is equivalent to Dijkstra's algorithm when the $h(x) = 0$, i.e. the heuristic function does not exist: in other words, Dijkstra's algorithm is the default version of A* algorithm.[28]
Definitely, the A* algorithm can be summarized in ten steps as follows:

1. Creation of *Open list*: consists on nodes that have been visited but not expanded (meaning that successors have not been explored yet). This is the list of pending tasks.

2. Creation of *Closed list*: consists on nodes that have been visited and expanded (successors have been explored already and included in the open list).

3. Insertion in the open list of the starting node with $f(n) = h(n)$.

4. If the open list is empty, the algorithm returns that the solution can not be found.

5. Extraction of the best node to visit (lowest $f(n)$).

6. If the extracted node is equal to destination node, the algorithm terminates: solution found.
7. Examination of child nodes.
8. Transfer of child nodes already visited and suboptimal from open list to closed one.
9. Inserting the remaining nodes in the open list.
10. Return to step 4.

Algorithm 3.2 A* Algorithm

```
1:  procedure MAIN(Graph, source, destination)
2:      Open = new list                          ▷ visited nodes but with unexplored successors
3:      Close = new list                         ▷ visited nodes and with explored successors
4:      Open ← source with f(start) = h(start)                              ▷ initialization
5:      while Open is not empty
6:          node ← Open with the lowest:                                    ▷ extract node from Open
7:          f(current.node) = g(current.node) + h(current.node)
8:          if current.node = destination
9:              break
10:         for ∀ successor of current.node
11:             successor.cost = g(current.node) + w(current.node, successor)   ▷ w(x,y) =
    weight of path between x and y
12:             if successor ∈ Open
13:                 if g(current)≤ successor.cost
14:                     continue to line 24
15:             else if  successor ∈ Closed
16:                 if g(current)≤ successor.cost
17:                     continue to line 24
18:                 move successor from Closed to Open
19:             else
20:                 successor → Open                                        ▷ move successor to Open
21:                 h(successor) = heuristic distance to destination
22:             g(successor) = successor.cost
23:             parent(successor) = current
24:         current → Closed                                    ▷ move current node to Closed
25:     if current ≠ destination
26:         launch error
```

Time Complexity

The time complexity of A* depends on the heuristic.
The heuristic function has a major effect on the practical performance of A* search, since a good heuristic allows A* to prune away many of the b^d nodes that an uninformed search would expand. Its quality can be expressed in terms of the effective *branching factor* b^* (the average number of successors per state), which can be determined empirically for a problem instance by measuring the number N of nodes expanded and the depth of the solution, then solving:

$$N + 1 = 1 + b^* + (b^*)^2 + ... + (b^*)^d \tag{3.3}$$

Good heuristics are those with low effective branching factor (the optimal being b* = 1).
In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution d with a time complexity of $O(b^d)$. This assumes that a goal state exists and is reachable from the start state; if it were not so and the state space is infinite, the algorithm would not terminate.
The time complexity is polynomial when the search space is a tree, there is a single goal state and the heuristic function h meets the following condition:

$$|h(x) + h^*(x)| = O(\log h^*(x)) \tag{3.4}$$

where h^* is the optimal heuristic, the exact cost to get from x to the goal. In other words, the error of h will not grow faster than the logarithm of the perfect heuristic h^* that returns the true distance from x to the goal.[28]

3.1.3 D*

The D* algorithm was introduced by the researcher Anthony Stentz in 1994. The name D* comes from the term *Dynamic A**, because the algorithm behaves like A* algorithm, except that the arc costs can change as the algorithm runs. It is known as graph search algorithm capable of fast replanning in changing environments: the D* algorithm finds the shortest path in graphs in which weights change during the time, i.e. occupancy values become higher or lower due to obstacles movement.

Overview

D* algorithm is a Incremental Search Algorithm and there exist three different versions in literature:

- *Original D**.
- *Focused D**: combines ideas of A* and the original D* algorithms.

- *D* Lite*: builds on LPA*, an incremental heuristic search algorithm that combines ideas of A* and Dynamic SWSF-FP (an algorithm used in solving the grammar problem).

D* and its variants have been widely used for mobile robot and autonomous vehicle navigation. When a AGV has to navigate to given goal coordinates in unknown terrain, the algorithm makes assumptions about the unknown part of the terrain (for example that it contains no obstacles) and finds a shortest path from its current coordinates to the goal coordinates under these assumptions. The AGV then follows the path. When algorithm receives new map informations (such as previously unknown obstacles), it adds the information to its map and, if necessary, replans a new shortest path from its current coordinates to the given goal coordinates. It repeats the process until AGV reaches the goal coordinates or determines that the goal coordinates can not be reached. So, the execution of the D* algorithm can be divided into *initial planning* and *replanning phases*. Initial planning is performed if the AGV is standstill at the start position (R = start) and replanning is performed if the AGV detects nodes with changed occupancy values during its motion.
Like A* algorithm, D* maintains a list of nodes to be evaluated, known as the *Open list*. Nodes are marked as having one of several states:

- *NEW*: it has never been placed on the list.
- *OPEN*: it is currently on the list.
- *CLOSED*: it is no longer on the list.
- *RAISE/LOWER*: its cost is higher/lower than the last time it was on the list.

The algorithm works by iteratively selecting a node from the OPEN list and evaluating it. It then propagates the node's changes to all of the neighbouring nodes and places them on the OPEN list. This propagation process is termed *expansion*. In contrast to A*, which follows the path from start to finish, D* begins by searching backwards from the goal node. Each expanded node has a *backpointer*, which refers to the next node leading to the target, and each node knows the exact cost to the target. When the start node is the next node to be expanded, the algorithm is done and the path to the goal can be found by simply following the backpointers. When an obstruction is detected along the intended path, all the points that are affected are again placed on the OPEN list, this time marked RAISE. Before a RAISED node increases in cost, however, the algorithm checks its neighbours and examines whether it can reduce the node's cost. If not, the RAISE state is propagated to all of the nodes' descendants, that is, nodes which have backpointers to it. These nodes are then evaluated, and the RAISE state passed on, forming a wave. When a RAISED node can be reduced, its backpointer is updated, and passes the LOWER state to its neighbours. These waves of RAISE and LOWER states are the heart of D*.

By this point, a whole series of other points are prevented from being *touched* by the waves. The algorithm has therefore only worked on the points which are affected by change of cost. This time, the Deadlock can not be bypassed so elegantly. None of the points can find a new route via a neighbour to the destination. Therefore, they continue to propagate their cost increase. Only outside of the channel points can be found, which can lead to destination via a viable route. This is how two LOWER waves develop, which expand as unattainably marked points with new route information.

Original D*

The Original D* algorithm consists primarily of two functions:

- *PROCESS-STATE*: used to compute optimal path-costs to the goal.
- *MODIFY-COST*: used to change the arc cost function $c()$ and enter affected states on the OPEN list.

that use the following variables:

- t(x): tag variable equal to NEW if x has never been on the OPEN list, OPEN if it is currently on the OPEN list and CLOSED if it is no longer on the OPEN list.
- b(x): backpointer from state x to the next state y.
- c(x,y): the cost of traversing an arc from state y to state x. If its value is a positive number, x and y are neighbours.
- h(x): path cost variable that represents an estimate of the sum of the arc cost from x to the (G)oal.
- k(x): key variable classifies a state x on the OPEN list into RAISE state (information about path cost increases) or LOWER state (information about path cost reductions). k_{min} is equal to min(k(x)) instead k_{old} is equal to k_{min} prior to the most recent removal of a state from the OPEN list.

In function PROCESS-STATE, the state x with the lowest $k()$ value is removed from the OPEN list. If x is a LOWER state ($k(x) = h(x)$), its path cost is optimal since $h(x)$ is equal to the old k_{min}. In the lines 13-20, each neighbour y of x is examined to see if its path cost can be lowered. Additionally, neighbour states that are NEW receive an initial path cost value, and cost changes are propagated to each neighbour y that has a backpointer to x, regardless of whether the new cost is greater than o less than the old. Since these states are descendants of x, any change to the path cost of x affects their path costs as well. The backpointer of y is redirected, if needed, so that the monotonic sequence y is constructed, All neighbours tha

receive a new path cost are placed on the OPEN list, so that they propagate the cost changes to their neighbours.
If x is a RAISE state, its path cost may not be optimal. Before x propagates cost changes to its neighbours, its optimal neighbours are examined to see if $h(x)$ can be reduced. At lines 22-26, cost changes are propagated to NEW states and immediate descendants in the same ways as for LOWER states. If x is able to lower the path cost of a state that is not an immediate descendant, x is placed back on the OPEN list for future expansion. This action is required to avoid creating a closed loop in backpointers. If the path cost of x is able to be reduced by a suboptimal neighbour, the neighbour is placed back on the OPEN list. Thus, the update is "postponed" until the neighbour has an optimal path cost.
In function MODIFY-COST, the arc cost function is updated with changed value. Since the path cost for state y will change, x is placed on the OPEN list. When x is expanded via PROCESS-STATE, it computes a new $h(y) = h(x) + c(x, y)$ and places y to OPEN list. Additional state expansions propagate the cost to the descendants of y.
The MAIN algorithm illustrates how to use PROCESS-STATE and MODIFY-COST to move the AGV from the state S through the environment to G along an optimal traverse. It starts setting $t()$ equal to NEW for all states, $h(G)$ is set to 0, and G is placed on the OPEN list. PROCESS-STATE is called repeatedly (line 7) until either an initial path is computed to the AGV's states (t(S) = CLOSED) or it is determined that no path exists (val = NO-VAL and t(S) = NEW). The AGV then proceeds to follow the backpointers in the sequence {R} until it either reaches the goal or discovers a discrepancy between the sensor measurement of an arc cost $s()$ and the stored arc cost $c()$ (e.g. due to a detected obstacle). This discrepancies may occur anywhere, not just in the sequence {R}.
MODIFY-COST is called to correct $c()$ and placed affected states on he OPEN list. PROCESS-STATE is then called repeatedly at line 15 until $val \geq h(R)$ to propagate costs and compute a possibly new sequence {R} to the goal. The AGV continues to follow the backpointers in the sequence towards the goal. The function returns GOAL-REACHED if the goal is found and NO-PATH if it is unreachable.
To underline the presence of two news sub-functions: LESS(a,b) that returns True if a < b and False otherwise and COST(x) that returns *h(x)* for state x.
It should be noted that line 9 only detects the condition of not existence of sequence of arcs from the AGV's state to the goal (for example, if the graph is disconnected): it does not detect the condition that all paths to the goal are obstructed by obstacles.[18]

Focused D*

The algorithm is an extension to D* that focuses the repairs to significantly reduce the total time required for the initial path calculation and subsequent replanning operations. This extension completes the development of the D* algorithm as a full generalization of A* for dynamic environments, where arc costs can change during the traverse of the solution path. The Focused D* algorithm can be configured to outperform Original D* in either total time

```
1:  procedure PROCESS-STATE(())
2:      x = MIN-STATE()                    ▷ return the OPEN state with minimum k()
3:      if x = NULL
4:          return NO-VAL
5:      k_old = k(x)
6:      DELETE(x)                          ▷ delete x from OPEN list and sets r(x) = CLOSED
7:      if k_old < h(x)
8:          for ∀ neighbour y of x
9:              if t(y) ≠ NEW and h(y) ≤ k_old and
10:                 h(x) > h(y) + c(y,x)
11:                 b(x) = y
12:                 h(x) = h(y) + c(y,x)
13:     if k_old = h(x)
14:         for ∀ neighbour y of x
15:             if t(y) = NEW or
16:                 (b(y) = x and h(y) ≠ h(x) + c(x,y)) or
17:                 (b(y) ≠ x and h(y) > h(x) + c(x,y))
18:                 b(y) = x
19:                 INSERT(y, h(x)+c(x,y))          ▷ change h(y) with the second given
20:                 input and inserts/repositions y on the OPEN list
21:     else
22:         for ∀ neighbour y of x
23:             if t(y) = NEW or
24:                 (b(y) = x and h(y) ≠ h(x) + c(x,y))
25:                 b(y) = x
26:                 INSERT(y, h(x)+c(x,y))
27:             else
28:                 if b(y) ≠ NEW and h(y) > h(x) + c(x,y) and
29:                     t(x) = CLOSED
30:                     INSERT(x, h(x))
31:                 else
32:                     if b(y) ≠ NEW and h(x) > h(y) + c(y,x) and
33:                         t(y) = CLOSED and h(y) > k_old
34:                         INSERT(y, h(y))
35:     return MIN-VAL()                   ▷ returns k_min for the OPEN list
```

```
1:  procedure MODIFY-COST(x, y, cval)
2:      c(x,y) = cval
3:      if t(x) = CLOSED
4:          INSERT(x, h(x))
5:      return MIN-VAL()                   ▷ returns k_min for the OPEN list
```

Algorithm 3.3 Original D* Algorithm

```
1:  procedure MAIN(S, G)
2:      for ∀ state x in the graph
3:          t(x) = NEW
4:      INSERT(G,0)
5:      val = 0
6:      while t(S) ≠ CLOSED and val ≠ NO-VAL
7:          val = PROCESS-STATE()
8:      if t(S) = NEW
9:          return NO-PATH
10:     R = S
11:     while R ≠ G
12:         for ∀ (x,y) such that s(x,y) ≠ c(x,y)
13:             val = MODIFY-COST(x, y, s(x,y))
14:         while LESS(val ,COST(R)) and val ≠ NO-VAL
15:             val = PROCESS-STATE()
            R = b(R)
16:     return GOAL-REACHED
```

or the on-line portion of the operation, depending on the requirements of the task.
The algorithm uses a heuristic function similar to A* to both propagate cost increases and focus cost reductions. A biasing function is used to compensate for AGV motion between replanning operations.
It computes an initial path from the goal state to the start state and then efficiently modifies this path during the traverse as arc costs change. The algorithm produces an optimal traverse, meaning that an optimal path to the goal is followed at every state in the traverse, assuming all known information at each step is correct. The focused version of D* outperforms the basic version: the addition of a heuristic focusing function to D* completes its development as a generalization of A* to dynamic environments. In fact, A* can be seen as the special case of D* where arc costs do not change during the traverse of the solution path.[19]
Let the focusing heuristic $g(x, R)$ be the estimated path cost from R the robot's location to x. Define a new function, the *estimated robot path cost*:

$$f(x, R) = g(x, R) + h(x) \tag{3.5}$$

and sort all LOWER states on the OPEN list by increasing $f()$ value. The function $f(x, R)$ is the estimated path cost from the state R through x to G (goal node). Provided that $G(x, R)$ satisfies the monotone restriction, then since $h(x)$ is optimal when LOWER state is removed from the OPEN list, an optimal path will be computed to R. Most of the extensions of the

Original D* algorithm to Focused D* one are confined to the functions for cost comparisons and management of the OPEN list; therefore, the functions COST, LESS, INSERT, MIN-STATE, and MIN-VAL are affected.

D* Lite

Building on LPA* algorithm, the D* Lite implements the same navigation strategy as D* but is algorithmically different. D* Lite is substantially shorter than D*, uses only one tie-breaking criterion when comparing priorities, which simplifies the maintenance of the priorities, and does not need nested if-statements with complex conditions that occupy up to three lines each, which simplifies the analysis of the program flow.
Lifelong Planning A* (LPA*) is an incremental version of A*. It applies to finite graph search problems on known graphs whose edge costs increase or decrease over time (which can also be used to model edges or vertices that are added or deleted). LPA* always determines a shortest path from a given start vertex $s_{start} \in S$ to a given goal vertex $s_{goal} \in S$, knowing both the topology of the graph and the current edge costs. *g*(s)* denotes the start distance of vertex s $\in$ S, that is, the length of a shortest path from s_{start} to s. Like A*, LPA* uses heuristic h(s,s_{goal}) that approximate the goal distances of vertices s.
So, LPA* repeatedly determines shortest paths between the start vertex and the goal vertex as the edge costs of a graph change.[15]
From LPA*, it is developed D* Lite, that repeatedly determines shortest paths between the current vertex of the AGV and the goal vertex as the edge costs of a graph change while the AGV moves towards the goal vertex.
In particular, the D* Lite algorithm is based on the backward LPA*, an extensions of A* algorithm, and searches from the goal to start locations while estimating distances from the goal, whereas the LPA* searches from the start to goal locations while estimating distances from the start. The D* Lite performs a search by first executing the *Compute Shortest Path* function which operates on a priority queue that initially contains only the goal node. Then, overtime, the predecessors of each node in the priority queue are added until the algorithm reaches the goal node.
Unlike LPA*, D* Lite uses the estimate $g()$ value which denotes the distance of a node to the goal (for example, $g(s)$ indicates the distance from the node s to the goal s_{goal}) and the *rhs* values are one-step look-ahead values based on the $g()$ values.

$$rhs = \begin{cases} 0 & if s = s_{start} \\ \min_{s' \in Pred(s)}(g(s') + c(s', s)) & otherwise \end{cases} \tag{3.6}$$

The *rhs* value of a node s with respect to its predecessor s' is defined as the cost of moving from s to s' plus the distance from s to the goal. Once the *rhs* values have been defined and are stable, the shortest path from the start to goal node can be found by moving from the start node towards its successor with the lowest *rhs* value until the goal node is reached. A

vertex is called *consistent* if its g-values equal its rhs-value, otherwise it is either *over-consistent* ($g(s) > rhs(s)$) or *under-consistent* ($g(s) < rhs(s)$). As AGV moves, the current vertex changes to its adjacent cells, the heuristic estimate needs to satisfy:

$$h(s_{start}, s) = \begin{cases} 0 & if s = s_{start} \\ \leq c(s', s)) + h(s_{start}, s') & otherwise \end{cases} \tag{3.7}$$

for all the s ∈ S and s' ∈ Pred(s) vertices and s_{start} ∈ S. When the cost of the edge changes, the algorithm must update the affected vertices and re-order the priority queue. The priority queue holds exactly the inconsistent states and these states need to be updated and made consistent. The priority or key value of a vertex s in the queue is:

$$\begin{aligned} k(s) &= [k_1(s), k_2(s)] \\ k_1(s) &= min(g(s), rhs(s)) + h(s_{start}, s) \\ k_2(s) &= min(g(s), rhs(s)) \end{aligned} \tag{3.8}$$

D* Lite expands vertices from the queue in increasing priority, updating their g-values and their predecessors rhs-values, until there is no vertex in the queue with a priority less than that of the start vertex. When AGV moves every vertex, the heuristic value associated with each inconsistent vertex needs to be updated. D* Lite is efficient because it uses an heuristic to restrict attention to only the vertices that could possibly be relevant to repairing the current solution path from a start s_{start} to the goal vertex s_{goal}.[13]

Time Complexity

Determining precisely the time complexity of the algorithm D * requires considerations, assumptions and proofs that are beyond the scope of this thesis. For more detailed information, we can consult the related papers discussing the time complexity of these versions of the algorithm. For our purpose, it needs to remember that he time complexity of D* depends on the heuristic function and on the type of implementation used (Lite, Focused and Original). About the heuristic, hold different considerations from A*, based on dynamic part of the algorithm. To these, the computational complexity of the various different implementations of the D* is added.

3.2 Sampling-Based Algorithms

The motion-planning problem can be solved also by first discretizing the continuous state space with either a grid for graph-based searches or through random sampling for stochastic incremental searches. Graph-based searches, such as the previous A* , are often resolution complete and resolution optimal. They guarantee to find the optimal solution, if a solution exists, and return failure otherwise (up to the resolution of the discretization). These graph-

based algorithms do not scale well with problem size (e.g. state dimension or problem range). Stochastic searches, such as Rapidly-exploring Random Trees (RRTs), Probabilistic Roadmaps (PRMs), and Expansive Space Trees (ESTs) use sampling-based methods to avoid requiring a discretization of the state space. This allows them to scale more effectively with problem size and to consider directly kinodynamic constraints; however, the result is a less-strict completeness guarantee. RRTs are probabilistically complete, guaranteeing that the probability of finding a solution, if one exists, approaches unity as the number of iterations approaches infinity.
Since the time complexity of this type of algorithms extremely depends on the number of samples, it is very difficult to determine exactly it and so it is neglected in this section.

3.2.1 RRT

The RRT, acronym di *Rapidly-exploring Random Tree*, is algorithm designed for efficiently searching unknown environment and non convex high-dimensional spaces, based on stochastic search strategies. RRT is constructed incrementally in a way that quickly reduces the expected distance of a randomly-chosen point to the tree.
RRT is particularly suited for path planning problems that involve obstacles and differential constraints (nonholonomic or kinodynamic). RRT can be considered as a technique for generating open-loop trajectories for nonlinear systems with state constraints. Usually, RRT alone is insufficient to solve a planning problem. Thus, it can be considered as a component that can be incorporated into the development of a variety of different planning algorithms.
The RRT approach to path planning introduces a technique of determining a planning path by selecting random points within a known environment and moving towards that point an incremental distance from the nearest node of an expanding tree. The movement from existing traversed points to random points in the environment will make a path that looks like a tree, and will cover most of the free space in the environment. A planned path will be found when a branch in the tree comes close the goal positions. The RRT algorithm has advantages at exploring free space in large and unknown environments as well as parallelizable on unlike many other path-finding algorithms.
However, the termination condition of tree formation may be limited by the success in finding goal. This condition may result in the tree containing many nodes in the order of hundred. Due to these drawbacks, there are some strategies that can help the original RRT algorithm to reduce the number of nodes. The number of node is an important factor in computation to find an optimal path using the tree.
The basic RRT construction algorithm relies on a tree structure being built that contains the start point as its root node, and eventually, the goal point as one of its leaves. The construction of the tree revolves around picking random points in the environment, finding the nearest point in the tree to this random point, then moving towards that point an incremental distance. If the incremental movement does not encounter an obstacle, insert the point as a new node in the tree. Eventually, the newly inserted node will be close to the goal node.

Once this occurs, a path from goal to start point can be made by traversing up the tree until the root node is reached.[8]

Algorithm 3.4 RRT Algorithm

```
1:  procedure BUILD_RRT(x_init)
2:      T.init(x_init
3:      for ∀ k = 1:K                          ▷ K = number of vertices
4:          x_rand ← RANDOM_CONFIG()
5:          EXTEND(T, x_rand)
6:      return T
1:  procedure EXTEND(T, x)
2:      x_near ← NEAREST_NEIGHBOUR(T, x)
3:      if NEW_CONFIG(x, x_near, x_new)
4:          T.add_vertex(x_new)
5:          T.add_edge(x_near, x_new)
6:          if x_new = x
7:              return Reached
8:          else
9:              return Advanced
10:     return Trapped
```

The basic RRT construction algorithm is give by the Alg.3.4.
A simple iteration in performed in which each step attempts to extend the RRT by adding a new vertex that is biased by a randomly-selected configuration. The EXTEND function selects the nearest vertex already in the RRT to the given sample configuration x. The function NEW_CONFIG makes a motion towards x with some fixed incremental distance ρ and tests for collision. This can be performed quickly using incremental distance computation algorithms. There situations can occur:

- *Reached*: x is directly added to the RRT because it already contains a vertex within ρ of x.
- *Advanced*: a new vertex $x_{new} \neq x$ is added to the RRT.
- *Trapped*: the proposed new vertex is rejected because it does not lie in C_{free} (set of configurations for which there is not collision with any static obstacles).

3.2.2 RRT*

RRT* is an optimized version of RRT.
When the number of nodes approaches infinity, the RRT* algorithm will deliver the shortest possible path to the goal. While realistically unfeasible, this statement suggests that the algorithm does work to develop a shortest path. The basic principle of RRT* is the same as RRT, but two key additions to the algorithm result in significantly different results.
First, RRT* records the distance each vertex has travelled relative to its parent vertex. This is referred to as the cost() of the vertex. After the closest node is found in the graph, a neighbourhood of vertices in a fixed radius from the new node is examined. If a node with a cheaper *cost()* than the proximal node is found, the cheaper node replaces the proximal node. The effect of this feature can be seen with the addition of fan shaped twigs in the tree structure. The second difference RRT* adds is the rewiring of the tree. After a vertex has been connected to the cheapest neighbour, the neighbours are again examined. Neighbours are checked if being rewired to the newly added vertex will make their cost decrease. If the cost does decrease, the neighbour is rewired to the newly added vertex. This feature makes the path more smooth.
RRT* creates incredibly straight paths. Additionally, its graphs are characteristically different from those of RRT. For finding an optimal path, especially in a dense field of obstacles, the structure of RRT* is incredibly useful. The graph vines around objects, finding shorter paths in comparison to RRT. If the destination were to change, the original graph can still be used as it represents the quickest path to most locations in the region. RRT* suffers from a reduction in performance. Due to examining neighbouring nodes and rewiring the graph, RRT* takes more time to complete a single path on average than the default version. The majority of computing effort comes from obstacle avoidance: this condition must be checked when a node is placed, when a node is connected to its neighbour and for each node that is to be rewired. This is a considerable number of checks to make.[9]

3.2.3 BIT*

The BIT* algorithm, acronym of *Batch Informed Tree*, is a planning algorithm that balances the benefits of graph-search and sampling-based techniques. It uses batches [Figure 3.1] of samples to perform an ordered search on a continuous planning domain, maintaining anytime performance. By processing samples in batches, its search can be ordered around the minimum solution proposed by a heuristic, as in A*. By processing multiple batches of samples, it converges asymptotically towards the global optimum with anytime resolution, as in RRT*. This is done efficiently by using incremental search techniques to incorporate the new samples into the existing search, as in LPA*.
Informally, BIT* works as follows. An initial Random Geometric Graph (RGG) with implicit edges is defined by uniformly distributed random samples in the free space. The RGG parameter (r or k) is chosen to reduce graph complexity, maintaining asymptotic optimality requirements as a function of the number of samples. An explicit tree is then built outwards

from the start towards the goal by a heuristic search. This tree includes only collision-free edges and its construction stops when a solution is found or it can no longer be expanded. This concludes a batch. To start a new batch, a denser implicit RGG is constructed by adding more samples and updating r (or k). If a solution has been found, these samples are limited to the sub-problem that could contain a better solution (e.g. an ellipse for path length). The tree is then updated using LPA*-style incremental search techniques that reuse existing information. As before, the construction of the tree stops when the solution can not be improved or when there are no more collision-free edges to traverse. The process continues with new batches as time allows.[12]

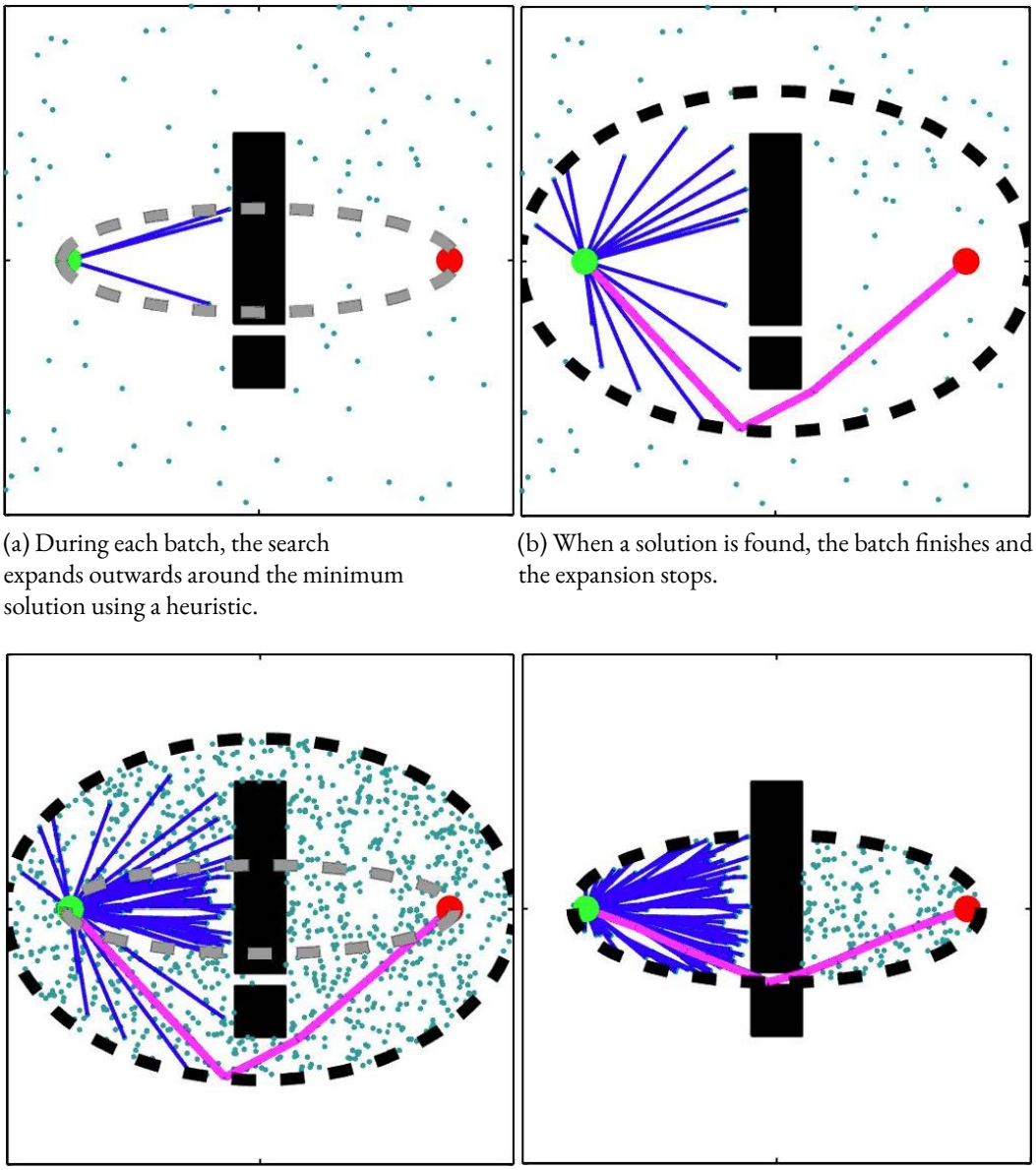
3.3 Bonus Algorithm: FMM

[22]The FMM, acronym of *Fast Marching Method*, is an efficient computational numerical algorithm for tracking and modeling the motion of a physical wave interface (front) Γ, developed in the 1990's by J.A. Sethian, a professor of mathematics at the University of California. This method has been applied to different research fields including computer graphics, medical imaging, computational fluid dynamics, image processing, computation of trajectories, etc.
The wave interface can be a flat curve in 2D, or a surface in 3D. The fast marching method calculates the time T that a wave needs to reach every point of the space. The wave can be originated from more than one point, each source point originates one wave. Source points have an associated time $T = 0$.
The Fast Marching Method is known in literature as a particular case of *Level Set Methods*: while the first is designed for problems in which the speed function never changes sign, so that the front is always moving forward or backward, the level set methods are designed for problems in which the speed function can be positive in some places are negative in others, so that the front can move forwards in some places and backwards in others.
The goal of FMM is to solve a discretised version of the *Eikonal equation* on a uniformly sized spatial grid: it is a non-linear partial differential equation encountered in problems of wave propagation, when the wave equation is approximated using the WKB method. It is derivable from Maxwell's equations of electromagnetics, and provides a link between physical (wave) optics and geometric (ray) optics.
In detail, the Eikonal equation for the motion of the front at a given point is:

$$F(x)|\nabla T(x)| = 1 \tag{3.9}$$

where x is the position, $F(x)$ the expansion speed of the wave at that position and $T(x)$ the time that the wave interface requires to reach x. The magnitude of the gradient of the arrival function $T(x)$ is inversely proportional to the velocity:

$$|\nabla T(x)| = \frac{1}{F(x)} \tag{3.10}$$



(a) During each batch, the search expands outwards around the minimum solution using a heuristic.

(b) When a solution is found, the batch finishes and the expansion stops.

(c) A new batch of samples is then added and the search restarts.

(d) The process repeats indefinitely, restarting each time an improved solution is found.

Figure 3.1: An illustration of the informed search procedure used by BIT*. The start and goal states are shown as green and red, respectively. The current solution is highlighted in magenta. The sub-problem that contains any better solutions is shown as a black dashed line, while the progress of the current batch is shown as a grey dashed line.

In this context, they are common assumptions that the front Γ evolves by motion in the normal direction and speed F does not have to be the same everywhere, but it is always non-negative ($F > 0$): they imply that the arrival time T is single valued.
Sethian proposed a discrete solution for the Eikonal Equation.
In 2D, the area is discretized using a gridmap: with i and j are denoted the row i and column j of the gridmap, corresponding to a point $p(x_i; y_j)$ in the real world.
The discretization of the gradient ∇T drives to the following equation:

$$\begin{Bmatrix} max(D_{ij}^{-x}T, 0)^2 + min(D_{ij}^{+x}T, 0)^2 \\ max(D_{ij}^{-y}T, 0)^2 + min(D_{ij}^{+y}T, 0)^2 \end{Bmatrix} = \frac{1}{F_{ij}^2} \tag{3.11}$$

In Sethian proposes a simpler but less accurate solution for the above equation, expressed as follows:

$$\begin{Bmatrix} max(D_{ij}^{-x}T, -D_{ij}^{+x}T, 0)^2 \\ max(D_{ij}^{-y}T, -D_{ij}^{+y}T, 0)^2 \end{Bmatrix} = \frac{1}{F_{ij}^2} \tag{3.12}$$

where:

$$\begin{aligned} D_{ij}^{-x} &= \frac{T_{i,j} - T_{i-1,j}}{\Delta x} \\ D_{ij}^{+x} &= \frac{T_{i+1,j} - T_{i,j}}{\Delta x} \\ D_{ij}^{-y} &= \frac{T_{i,j} - T_{i,j-1}}{\Delta x} \\ D_{ij}^{+y} &= \frac{T_{i,j+1} - T_{i,j}}{\Delta x} \end{aligned} \tag{3.13}$$

and Δx and Δy are the grid spacing in the x and y directions.
Substituting (3.13) in (3.12) and making:

$$\begin{aligned} T &= T_{ij} \\ T_1 &= min(T_{i-1,j}, T_{i+1,j}) \\ T_2 &= min(T_{i,j-1}, T_{i,j+1}) \end{aligned} \tag{3.14}$$

the Eikonal equation can be rewritten for a discrete 2D space as:

$$max\left(\frac{T - T_1}{\Delta x}, 0\right)^2 + max\left(\frac{T - T_2}{\Delta y}, 0\right)^2 = \frac{1}{F_{ij}^2} \tag{3.15}$$

Assuming that the speed of the front is positive ($F > 0$) T must be greater than T_1 and T_2 whenever the front wave has not already passed over the coordinates i, j. Consequently

(3.15) can be solved as the following quadratic:

$$\left(\frac{T-T_1}{\Delta x}\right)^2 + \left(\frac{T-T_2}{\Delta y}\right)^2 = \frac{1}{F_{ij}^2} \tag{3.16}$$

whenever $T > T1$ and $T > T2$. If $T < T1$ and $T < T2$, (3.15) is reduced to:

$$max\left(\frac{T-T_1}{\Delta x}, 0\right) = \frac{1}{F_{ij}} \tag{3.17}$$

if T resulted to be smaller than T_1 when solving (3.16), or:

$$max\left(\frac{T-T_1}{\Delta x}, 0\right) = \frac{1}{F_{ij}} \tag{3.18}$$

if T resulted to be smaller than T_2 when solving (3.16).
In the end, it is important to highlight a property of the waves expansion. The $T(x)$ function, originated by a wave that grows from one single point, presents only a global minima at the source and no local minima. As $F > 0$ the wave only grows (expansion), and hence, points farther from the source have greater T. A local minima would involve that a point has a T value lesser than a neighbour point which is nearer to the source, which is impossible, as this neighbour must have been reached by the wave before.

Algorithm

Now the FMM for solving the Eikonal Equation over a gridmap is presented in detail.
To solve iteratively 3.15 over a gridmap G of size $m \times n$, the cells of the map must be labeled of one of the following types:

- *Unknown*: cells whose T value is not known yet (the wave front has not reached the cell).
- *Narrow Band*: candidate cells to be part of the front wave in the next iteration. They are assigned a T value that can still change in future iterations of the algorithm.
- *Frozen*: cells that have already been passed over by the wave and hence their T value is fixed.

Algorithm 3.5 FMM Algorithm

```
1:  procedure MAIN(G, Ori)
2:      for ∀g_ij ∈ Ori
3:          g_ij.T ← ∞
4:          g_ij.state ← FROZEN
5:          for ∀g_kl ∈ g_ij.neighbours
6:              if g_kl = FROZEN
7:                  skip
8:              else
9:                  g_kl.T ← solveEikonal(g_kl)
10:                 if g_kl.state = NARROW BAND
11:                     narrow_band.update_position(g_kl)
12:                 if g_kl.state = UNKNOWN
13:                     g_kl ← NARROW BAND
14:                     narrow_band.insert_in_position(g_kl)
15:         while narrow_band NOT EMPTY
16:             g_ij ← narrow_band.popfirst()
17:             for ∀ g_kl ∈ g_ij.neighbours
18:                 if g_kl = FROZEN
19:                     skip
20:                 else
21:                     g_kl.T ← solveEikonal(g_kl)
22:                     if g_kl.state = NARROW BAND
23:                         narrow_band.update_position(g_kl)
24:                     if g_kl.state = UNKNOWN
25:                         g_kl ← NARROW BAND
26:                         narrow_band.insert_in_position(g_kl)
27:     return G.T
```

The algorithm has three stages: initialization, main loop, and finalization. These stages are described bellow.

1. *Initialization*: the algorithm starts by setting T = 0 in the cell or cells that originate the wave (*Ori*). These cells are labeled as *frozen*. Afterward it labels all their neighbours as *narrow band*, computing T (3.15) for each of them.

2. *Main Loop*: in each iteration the algorithm will solve the Eikonal Equation (3.15) for the neighbours (that are not yet frozen) of the narrow band cell with the lesser T value. This cell is then labeled as *frozen*. The narrow band maintains an ordered list of its cells. Cells are ordered by increasing T value (first cells have lesser T values).

3. *Finalization*: when all the cells are *frozen* (the *narrow band* is empty) the algorithm finishes

If we want to compute the path between two points p_0 and p_1 we could expand a wave from p_1 until it reaches p_0. Due to the wave expansion properties, the path that has followed the wave interface from the target to the source point will be always the shortest trajectory in time. As the wave expansion speed is constant, this trajectory is also the shortest solution in distance. The wave is originated from the target point, hence, the computed $T(x)$ field will have only one minima at the target point. Hence, following the maximum gradient direction from the initial point we will reach the target point, obtaining the trajectory. This solution is unique and complete.
The resulting gridmap stores at any pixel the time T required by the front wave to reach that pixel. The isocurves join together all the points that have been passed through at the same instant of time. These curves are the trace of the front wave. If we compute the maximum gradient direction at any point of the gridmap we obtain the normal direction to the isocurve, that is, the direction the curve has followed when expanding. The maximum gradient direction is computed applying the *Sobel Operator* over the gridmap, a differential operator which calculates an approximate value of the gradient of a function that represents the brightness of the image:

$$\begin{aligned} grad_x &= \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \star T \\ grad_y &= \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \star T \end{aligned} \tag{3.19}$$

where $\star$ indicates the two-dimensional convolution operation.
For tracing the path between p_0 and p_1 we just need to follow the maximum gradient direction starting at p_0. The path is computed iteratively. $grad_{ix}$ and $grad_{iy}$ are computed

at every point p_i. From pi is computed p_{i+1} and successively until arriving to p_1. As p_1 is located at the global minima, it is always reached (whenever there is path).

$$\begin{aligned} mod_i &= \sqrt{grad_{ix}^2 + grad_{iy}^2} \\ alpha_i &= \arctan\left(\frac{grad_{iy}}{grad_{ix}}\right) \\ p_{(i+1)x} &= p_{ix} + mod_i \cdot \cos\left(alpha_i\right) \\ p_{(i+1)y} &= p_{iy} + mod_i \cdot \sin\left(alpha_i\right) \end{aligned} \tag{3.20}$$

Time Complexity

The algorithm is closely related to Dijkstra's single-source shortest path algorithm and computes the grid solution, starting with the grid points adjacent to the boundary, by traversing the computational domain along increasing values of T.
The total complexity of this method is $O(n\log(n))$, where N denotes the number of grid points. Here the factor $log(n)$ comes from the administration of a priority queue.
John N. Tsitsiklis, a professor of Electrical Engineering at the Massachusetts Institute of Technology, independently suggested a similar algorithm in the 1995. The discretization of the Eikonal equation in Tsitsiklis was obtained in the framework of optimal control theory. Moreover, Tsitsiklis showed that a bucket sort technique, together with a slightly different discretization, may be used to compute an approximate solution to the Eikonal equation in $O(n)$ time.
Another approach has recently been suggested by Liron Yatziv, a researcher of Department of Electrical and Computer Engineering of the University of Minnesota: using an untidy priority queue in the FMM also reduces the complexity to optimal $O(n)$. The idea is to use a bucket sort technique together with a quantization that does not distinguish between values of T within a small range. This way, while sorting the values of T becomes less expensive, an error is introduced, however, which should be of the same order as the local truncation error of the discretization.[16]

4
Experimental Design

In this chapter, the experimental design of the AGV is presented. Initially, the softwares used to get a robust and efficient simulation are described. Subsequently, the steps leading to the creation of the AGV model and the simulation scenario are highlighted. It should be emphasized that precision and accuracy have been two key parameters in the design of the simulation.

4.1 Operative Environment

To carry out simulations fairly faithful to reality, it has been decided to use ROS as a development framework and Gazebo as a graphic viewer of the computer code. With the help of other environments as: MATLAB & Simulink to improve the dynamics of the simulation, Autodesk Inventor to build 3D objects and Blender to manage the visualization of the 3D objects in the Gazebo, the interaction between these systems makes it possible to design a fluoride feeder model and to visualize its behaviour in a semi-real scenario, which in our case is the potroom.

4.1.1 ROS

ROS, acronym of *Robot Operative System*, is an open source framework used to build advanced robot applications. It was originally developed in 2007 by the Sanford Artificial Intelligence Laboratory (SAIL) and through the years it has become a de facto standard in the research field. Its appeal is growing even in the industry, thanks to the ROS-Industrial consortium. ROS is designed to be flexible, general-purpose and robust. It includes a constantly

increasing number of tools, libraries and interfaces that can be reused and improved by anyone.
One of the key features of this framework is the possibility to use virtually any programming language. At this moment the main supported languages are C++, Java and Python.[6]
The system is based on a modular concept, that consists in dividing a complex task in simpler and reduced subtasks: they are encapsulated in executable files called *nodes*, which communicate amongst each other to generate the final behaviour of the robot. Each node has a unique name that distinguishes it from the rest of the existent nodes. The nodes can communicate with one another through three different communication procedures: publishing or subscribing it to a topic, providing or using one service, or using actions. One robot can have many independent nodes working in a cooperative manner with the global goal to achieve a complex behavior of the robot. With this modularity we obtain an important advantage that is an easy error correction, due to the fact that we can localize and solve errors easily inside small functions, reducing the complexity in comparison with the monolithic codes. Also, ROS is used since it allows the sharing of information between all kind of nodes, and it is possible to use previously developed processes and driver can easily be encapsulated in the nodes. Furthermore, it offers powerful tools for debugging, which saves time and allows the programmer to correct any error found.
There are three communication protocols that the nodes use to communicate in ROS:[11]

- *Topics*: these are data buses that are used by nodes to exchange messages among them. The nodes can be of two different types:
  - *publishers*: generate data of one topic, for example the nodes that corresponds to the encapsulated drivers of the sensors are publishers that publish messages that contain the sensed values.
  - *subscribers*: nodes that are subscribed to the topics that publish another node.

  All the nodes can be at the same time publishers and subscribers of different topics: we can also have multiple publishers and subscribers of the same topic. The nodes can be subscribed or can published topics in anonymous form, therefore, the production of information is independent of its use. In general, nodes do not know with whom they are communicating. The unit that performs communication and knows whether nodes are published or subscribed to a topic is the *roscore*.

- *Services*: These allows communication between nodes that have requests and answers. One service is defined by two types of messages, one for the request and other for the answers. In these situations one node takes up the role of the client: it sends the request to obtain a service and it waits until the server node sends the answer.

- *Actions*: These are based on the same principle as the services: a request is sent and a response is received. The difference is that the action adds the ability to cancel the

service, and therefore the nodes do not need to wait until it gets the answer. An action is defined by three messages: goal, feedback and result. The first contains the reason of the request, the second periodically provides the information of the state of the system and the last is the result of the request.

To complete the ROS interface, they have been created the *Rosbags*: data files where the publisher messages are stored. In these files we only save the information of the publisher nodes that we need to replay the real robot experiment offline in the computer. This data can be stored in topics of one node that performs a complex behaviour of the robot or can be only nodes that encapsulate one driver of one sensor and return the sensed data. These files allows us to recreate real situations offline and provide the option to verify or improve our created algorithms. Therefore, the rosbags may be designated for two functions: use our node in the environment Gazebo of our computer to finish the debug of some errors or record the experiments on the robot to analyze them further.

4.1.2 Gazebo

Gazebo is an open-source 3D robotics simulator. Its development began in the fall of 2002 at the University of Southern California by Dr. Andrew Howard and his student Nate Koenig. The name has stuck despite the fact that most users of Gazebo simulate indoor environments. In 2009, it happens the integration of ROS into Gazebo, which has since become one the primary tools used in the ROS community.
It offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments and it can use multiple high-performance physics engines, such as ODE, Bullet, etc. It provides realistic rendering of environments including high-quality lighting, shadows and textures and it can model sensors that "see" the simulated environment, such as laser range finders, cameras (including wide-angle) and Kinect style sensors.[25]

4.1.3 Maltab & Simulink

Matlab, acronym of *MATrix LABoratory*, is an environment for numerical calculation that allows to manipulate matrices, view functions and data, implement algorithms, create user interfaces, and interface with other programs. It was developed in the late 1970s by Cleve Moler, the chairman of the computer science department at the University of New Mexico. The MATLAB application uses a own programming language, very similar to Python programming language. Common usage of the MATLAB application involves using the *Command Window*, as an interactive mathematical shell, or *Script*, executing text files containing MATLAB code: the involved variables are saved in the *Workspace* window.
It can be interface with many libraries: for example, the ones written in Perl, Java, ActiveX or .NET can be directly called from MATLAB. To interface MATLAB with Java is more complicated, but can be done with a MATLAB toolbox which is sold separately by MathWorks,

or using an undocumented mechanism called JMI (Java-to-MATLAB Interface)
Simulink is an additional package of MATLAB: it is a graphical programming environment for modelling, simulating and analysing multi-domain dynamical systems. Its primary interface is a graphical block diagramming tool and a customizable set of block libraries. It offers tight integration with the rest of the MATLAB environment and can either drive MATLAB or be scripted from it. Simulink is widely used in automatic control and digital signal processing for multi-domain simulation and model-based design.[26]

4.1.4 Autodesk Inventor

Autodesk Inventor is a 3D modeling software for mechanical design, produced by Autodesk, the software company that produces AutoCAD.
The 3D mechanical design software, or modeller, incorporates a module for creating parts, a module for creating the assembly of the same and the module for putting on the table, or the realization of a drawing with views and all the necessary quotas for the realization of the piece.
Inventor gives the possibility to work in different specific environments for each prototyping sector; however, it is also possible to obtain other types of information, such as rendering, interference analysis, exploded views, motion and resistance simulations of materials and objects, wiring and 3D piping. Inventor is also equipped with standard libraries that can be customized with the parts created by the user himself, equipped with guided tools (also known as *wizards*) for managing and inserting the pieces they contain and the static calculation of the seal of most of them.[23]

4.1.5 Blender

Blender is a free and open-source 3D computer graphics software tool-set used for creating animated films, visual effects, art, 3D printed models, motion graphics, interactive 3D applications, and computer games. The program was developed as an internal application by the Dutch animation studio NeoGeo in 1998. The main author is considered Ton Roosendaal.
The software is rich in features typical of advanced modelling systems. Among its potential, it can remember:[24]

- support for a wide variety of geometric primitives, including Polygon Meshs (3D computer object in space, consisting of vertices, edges and faces).

- Python scripting to automate and/or control numerous aspects of the program and the scene.

(a) ROS and Gazebo.

(b) Matlab & Simulink.

(c) Autodesk Inventor.

(d) Blender.

Figure 4.1: Brands.

4.2 Fluoride Feeder Vehicle

Among the vehicles that work in the potroom, it has been decided to choose the fluoride feeder as a possible simulation vehicle.[Figure 4.2]
As already mentioned in Section 2.6.1, the task of this vehicle is to fill its own tank of aluminium fluoride, reach the side of the potcell and transfer it into it. The purpose of aluminium fluoride is to lower the melting point of the compound found in the potcells to make electrolysis occur at suitable temperatures.[Appendix B]
The fluoride transfer occurs through a mechanical arm that goes to position itself on the top of the potcell in a special hole [Figure 4.2e-4.2f]. Generally the mechanical arm can reach about 6m in length and is equipped with a central joint to allow topping up at different points on the cell surface.
On the other hand, the vehicle's fluoride tank has a top hole to allow it to be filled cyclically by a charging silo: the vehicle is positioned below it and, after having connected the hole to the pipe, it receives the aluminium fluoride.
In the following subsections, the kinematics and dynamics of the vehicle model are described. This is useful to the reader in order to understand in detail aspects that Gazebo deals with in a transparent manner.

(a) Right side rear view.

(b) Righ side view.

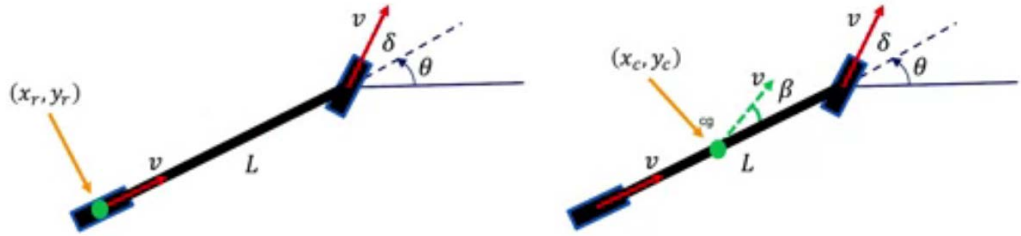
(c) Right side front view.

(d) Left side front view.

(e) Mechanical arm.

(f) Mechanical arm in action.

Figure 4.2: Fluoride feeder vehicle.



(a) Body frame centered in the rear wheel. (b) Body frame centered in the CoM.

Figure 4.3: Bicycle kinematic model.

4.2.1 Kinematic Model

The kinematic model derives from geometrical relations between different elements of the vehicle. To obtain a simple model, it is useful to approximate the vehicle structure with the bicycle kinematic model [Figure 4.3]:

$$\begin{cases} \dot{x}_r(t) = v(t)\, cos(\theta(t)) \\ \dot{y}_r(t) = v(t)\, sin(\theta(t)) \\ \omega(t) = \dot{\theta}(t) = \frac{v(t)}{R} = \frac{v(t)}{L}\, tg(\delta(t)) \end{cases} \quad \text{using} \quad tg(\delta(t)) = \frac{L}{R} \tag{4.1}$$

in the case of origin of the body frame positioned at the rear wheel [Figure 4.3a], or:

$$\begin{cases} \dot{x}_c(t) = v(t)\, cos(\theta(t) + \beta(t)) \\ \dot{y}_c(t) = v(t)\, sin(\theta(t) + \beta(t)) \\ \omega(t) = \dot{\theta}(t) = \frac{v(t)}{L}\, cos(\beta(t))\, tg(\delta(t)) \end{cases} \tag{4.2}$$

in the case of origin of the body frame positioned at the *Center of Mass* (CoM).[Figure 4.3b] In these models compare different variables, including:

- $\mathbf{x}_r(t) = [x_r(t)\, y_r(t)]^T$: coordinates of the rear wheel center, w.r.t. the world reference frame.
- $\mathbf{x}_c(t) = [x_c(t)\, y_c(t)]^T$: coordinates of CoM, w.r.t. the world reference frame.
- R: the distance of the center of the rear wheel from the *Instantaneous Center of Rotation* (ICR).
- $\theta(t)$: the angle between the world frame and body frame abscissa coordinates.
- $\delta(t)$: the angle defined by the direction of the front wheel w.r.t. the back one or, equivalently, w.r.t. the body frame abscissa, the heading of the bicycle.

- $v(t)$: the linear velocity amplitude, derived from the point of contact between the rear wheel and the ground.
- $\beta(t)$: the *(side) slip angle*, i.e. the angle between the velocity vector of the CoM (of amplitude $v(t)$) and the body frame abscissa coordinate.

The last value, using the *non slipping condition*, can be computed as:

$$\beta(t) = arctg\left(\frac{l_r}{L}\, tg(\delta(t))\right) \tag{4.3}$$

where l_r is the distance of the CoM from the rear wheel center.
In both cases, but for simplicity only the first one can be considered because in the second one the position of the CoM could change due to variable fluoride load, the model can be rewritten as a state space one:

$$\begin{cases} \dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \\ \mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t)) \end{cases} \tag{4.4}$$

where:

- $\mathbf{x}(t) = [x_b(t)\, y_b(t)\, \theta(t)]^T$: the state that represents the pose of the vehicle, w.r.t. the world reference frame.
- $\mathbf{u}(t) = [v(t)\, \delta(t)]^T$: the control input.
- $\mathbf{y}(t) = [x_b(t)\, y_b(t)]^T$: the output defined as the position of the body frame, w.r.t. the world one.

Alternatively, it is possible to consider $\delta(t)$ as part of the state and its derivative $\psi(t) = \dot{\delta}(t)$ together with $v(t)$ as inputs.
To notice that in this model some uncertainties have been discarded, such as the *slipping phenomenon* and other possibles external disturbances.
At this point, in order to obtain a good description for a car-like model as the one considered, it is necessary to define the relation between (4.4) and the variables that describe the so called *Ackermann steering* [Appendix C] model. Such model best describes the behaviour all the car-like vehicles, from the kinematic point of view. In particular, the steering angles of the two front wheels can be retrieved from the one of the bicycle model using geometrical relations:

$$\begin{cases} tg(\delta_i(t)) = \frac{L}{R_i} = \frac{L}{R - \frac{W}{2}} \\ tg(\delta_o(t)) = \frac{L}{R_o} = \frac{L}{R + \frac{W}{2}} \end{cases} \tag{4.5}$$

where W is the vehicle width and the subscripts i and o indicates respectively the quantities related to the inner and outer steering wheels.
Similarly, from the fact that both the rear wheels circle around the ICR with the same angular velocity $\omega(t)$ but with different radii R_i and R_o respectively, it follows that:

$$\begin{cases} v_i(t) = \omega(t)R_i = v(t)\frac{R_i}{R} = v(t)\frac{R-\frac{W}{2}}{R} = v(t)\left(1 - \frac{W}{2R}\right) = v(t)\left(1 - \frac{Wtg(\delta(t))}{2L}\right) \\ v_o(t) = \omega(t)R_o = v(t)\frac{R_o}{R} = v(t)\frac{R+\frac{W}{2}}{R} = v(t)\left(1 + \frac{W}{2R}\right) = v(t)\left(1 + \frac{Wtg(\delta(t))}{2L}\right) \end{cases} \tag{4.6}$$

and, from this result, it holds that:

$$\begin{cases} \omega_i(t) = \frac{v_i(t)}{r} \\ \omega_o(t) = \frac{v_o(t)}{r} \end{cases} \tag{4.7}$$

where $v_i(t), v_o(t), \omega_i(t), \omega_o(t)$ are respectively the linear and angular velocities of the inner and outer rear wheels. To underline that $\omega(t)$ is not the angular velocity of the rear wheel of the bicycle model but is the angular velocity of body frame w.r.t. the ICR.

4.2.2 Dynamic Model

The dynamic model considers all the forces applied to the vehicle and can be divided into two sub-models: *longitudinal* and *lateral* dynamics. For simplicity the subscript b has been removed from the notation of the vehicle position w.r.t. the world frame, in order to match with the images.

Longitudinal Dynamics

Longitudinal dynamics considers the balancing between all the forces acting in the vehicle along the longitudinal direction.

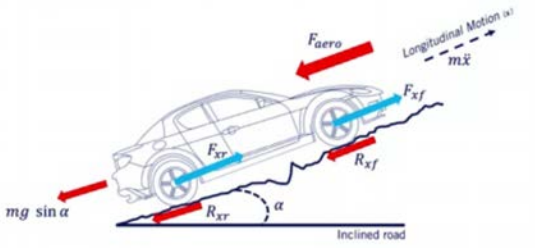


Figure 4.4: Longitudinal dynamics.

In the case of the considered vehicle, it holds that:

$$m\ddot{x}(t) = F_x - R_x - F_{aero} - mgsin(\alpha) \tag{4.8}$$

where:

- m: the mass of the vehicle.
- $F_x = F_{x,f} + F_{x,r}$: the total longitudinal force generated by the car engine.
- $R_x = R_{x,f} + R_{x,r}$: the rolling resistance of the wheels.
- $mgsin(\alpha)$: the gravitational contribution, assuming the road inclined of angle α.
- F_{aero}: the aerodynamic drag force.

The displacement between these forces defines the acceleration $\ddot{x}(t)$ along the longitudinal direction.
The F_{aero} quantity can be modelled as:

$$F_{aero} = \frac{1}{2} C_a \rho A \dot{x}(t)^2 = c_a \dot{x}(t)^2 \tag{4.9}$$

where A is the front area of the vehicle, ρ is the air density and C_a is the *vehicle's friction coefficient*. Instead the R_x quantity can be approximated as:

$$R_x \approx c_{r,1} |\dot{x}(t)| \tag{4.10}$$

with $c_{r,1}$ the *linear rolling resistance coefficient*.
It is possible to verify that, during the acceleration and breaking phases, the longitudinal velocity of the vehicle does not match the one expected using the pure rolling constraint because of the sliding between the wheels and the road produced by the tires deformation. This phenomenon can be represented using the *slip ratio*:

$$s = \frac{\omega_w(t) r_e - v(t)}{v(t)} \tag{4.11}$$

in which $\omega_w(t)$ is the wheel angular velocity, r_e is the effective radius of the wheel and $v(t)$ is the actual vehicle longitudinal velocity. In this context there are three cases:

- wheels are skidding ($\omega_w(t) r_e < v(t)$): this happens during the deceleration of the vehicle, in normal breaking condition.
- wheels are spinning ($\omega_w(t) r_e > v(t)$): this happens during the acceleration of the vehicle, especially in low friction condition.

- wheels are locked ($\omega_w(t)r_e = 0$ and $v(t) \neq 0$): this happens during heavy breaking, when the vehicle loses its desired traction.

Lateral Dynamics

Lateral dynamics considers the effect of all the moments and lateral forces acting in the vehicle. In particular it extends the bicycle model, seen in the kinematic case, relaxing the *non slipping condition*. In order to simplify the analysis, the following assumptions have been made:

- Constant longitudinal velocity, in order to decouple longitudinal and lateral dynamics.
- Non linear effects such as the suspension movement, road inclination and aerodynamic effects have been neglected.
- The body frame is attached to the center of mass in order to simplify the application of the Newton's second law.

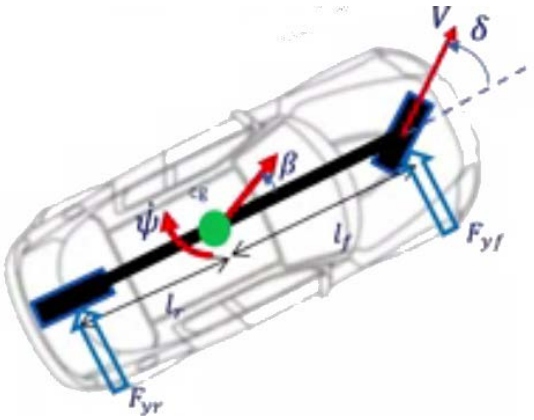


Figure 4.5: Lateral dynamics.

The lateral dynamics can be written as:

$$m\ddot{y}(t) = F_{yf} + F_{yr} \tag{4.12}$$

where the acceleration along the world y axis can be defined as the sum of the acceleration along the body y axis and the *centripetal acceleration*, i.e.:

$$\ddot{y}(t) = \ddot{y}_b(t) + \omega(t)^2 R = v\dot{\beta}(t) + v\dot{\theta}(t) \tag{4.13}$$

from the derivative of $\beta(t)$ and $v = R\omega(t)$ with $\omega(t) = \dot{\theta}(t)$. The F_{yf} and F_{yr} are respectively the *front* and *rear tire forces* while v is the (constant) longitudinal velocity in this case. Combining (4.12) with (4.13), it follows that:

$$mv(\dot{\beta}(t) + \dot{\theta}(t)) = F_{yf} + F_{yr} \tag{4.14}$$

In addition to this relation, also the rotational behaviour can be taken into account using the Euler's law:

$$I_z\dot{\omega}(t) = I_z\ddot{\theta}(t) = l_f F_{yf} - l_r F_{yf} \tag{4.15}$$

where I_z is the vehicle inertia about the z axis (directed upwards) while l_f and l_r represent the distance of the CoM respectively from the front and rear wheels.
Even if in general the lateral tire forces are difficult to be estimated, they can be approximated through a linear function, for small tire slip angles. So, let $\alpha_f(t)$, $\alpha_r(t)$ be the *front* and *rear tire slip angles* defined similarly to the vehicle slip angle $\beta(t)$ (i.e. angles between the respective wheel direction and the velocity vector at the considered wheel center), it holds that:

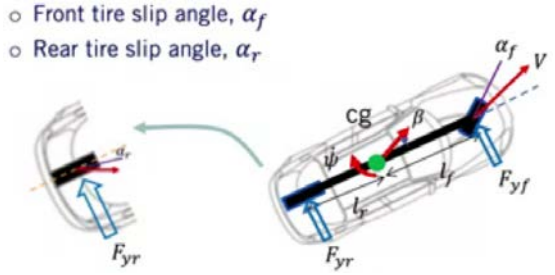


Figure 4.6: Front and rear tire slip angles.

$$\begin{cases} F_{yf} = C_f\alpha_f(t) = C_f\left(\delta(t) - \beta(t) - \frac{l_f\dot{\theta}(t)}{v}\right) \\ F_{yr} = C_r\alpha_r(t) = C_r\left(-\beta(t) + \frac{l_r\dot{\theta}(t)}{v}\right) \end{cases} \tag{4.16}$$

with C_f, and C_r *front* and *rear wheel linearized cornering stiffness*.
Substituting (4.16) in (4.14) and (4.15), rearranging the results, it is possible to obtain the linear state space model:

$$\begin{aligned} \dot{\mathbf{x}}_{lat}(t) &= A_{lat}\mathbf{x}_{lat}(t) + B_{lat}\delta(t) = \\ &= \begin{bmatrix} 0 & v & v & 0 \\ 0 & -\frac{C_r+C_f}{mv} & 0 & \frac{C_r l_r - C_f l_f}{mv^2} - 1 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{C_r l_r - C_f l_f}{I_z} & 0 & -\frac{C_r l_r^2 - C_f l_f^2}{I_z v} \end{bmatrix} \mathbf{x}_{lat}(t) + \begin{bmatrix} 0 \\ \frac{C_f}{mv} \\ 0 \\ \frac{C_f l_f}{I_z} \end{bmatrix} \delta(t) \end{aligned} \tag{4.17}$$

where $\mathbf{x}_{lat}(t) = [y(t)\ \beta(t)\ \theta(t)\ \dot{\theta}(t)]^T$ is the lateral state vector with:

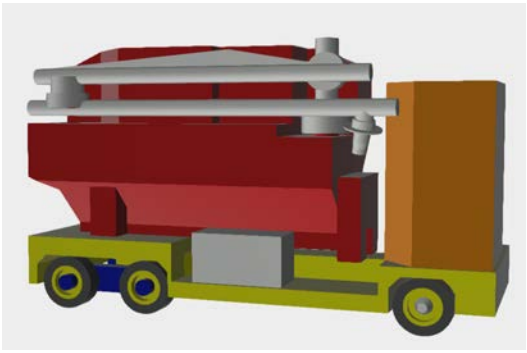
- $y(t)$: lateral position in the world frame.
- $\beta(t)$: side slip angle.
- $\theta(t)$: yaw angle.
- $\dot{\theta}(t)$: yaw rate.

4.2.3 Simulation Model

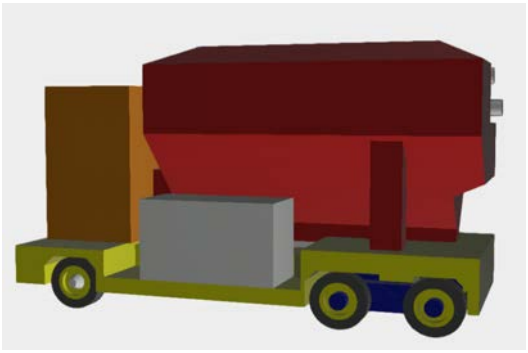
To obtain a high precision simulation model, it is good practice, first of all, to start from the model closest to reality in terms of shape, size and descriptive parameters: it has been therefore decided to use the AutoCAD file in .dwg format of the initial project and extrapolate a model that is easy to simulate and very similar to the real vehicle. The result can be seen in the Figure 4.7: despite some details have been neglected or replaced with simple three-dimensional geometric figures (comparing the Figure 4.7b and Figure 4.7d, we can see how the engine is replaced by a gray box), the final result is robust and very adherent to reality.
To bring the AutoCAD model to ROS & Gazebo, generally, a .urdf file is defined that represents the vehicle model in the form of a code (XML) and, subsequently, in a transparent manner to the user, the code is converted into a three-dimensional model identical to the AutoCAD one, but movable in a Gazebo.
In the .urdf file particular constructs appear to define the various parts of the model and the type of union between them: they take the name, respectively, of *link* and *joint*.

(a) Real right side.

(b) Real left side.



(c) Virtual right side.



(d) Virtual left side.

Figure 4.7: Real fluoride feeder vehicle vs simulative fluoride feeder vehicle.

Link

The link element describes a rigid body with an inertia, visual features, and collision properties. Here is an example of a link element:

```
<link name="my_link">

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="package://name_package/name.dae"/>
    </geometry>
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="package://name_package/name.dae"/>
    </geometry>
  </collision>

  <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia ixx="100" ixy="0" ixz="0"
    iyy="100" iyz="0" izz="100"/>
  </inertial>

</link>
```

Listing 4.1: XML code for link element.

The *link* tag defines the name of the link and encases sub-tags as:[27]

- *visual*: specifies the shape of the object for visualization purposes with:
  - *origin*: specifies the reference frame of the visual tag, relative to the reference frame of the link, through X, Y and Z offsets and R(oll), P(itch) and Y(aw) angles in radians.
  - *geometry*: describes the shape of the visual object, which can be three-dimensional geometric figure or a *mesh* (tag that allows the import of Polygon Mesh files .dae or .stl).

- *collision*: gives consistency to the visual tag. It has the same sub-tags of *visual* tag.
- *inertial*: quantifies the consistency through:
  - *origin*: same role of visual tag.
  - *mass*.
  - *inertia matrix*: represents the six above-diagonal elements of the symmetric rotational inertia matrix.

Joint

The joint element describes the kinematics and dynamics between two links: it represent the simplest way to merge two different links. Here is an example of a joint element:

```
<joint name="my_joint" type="...">

  <origin xyz="0 0 1" rpy="0 0 3.1416"/>

  <parent link="link1"/>
  <child link="link2"/>

  <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7"/>

</joint>
```

Listing 4.2: XML code for joint element.

The *joint* tag defines the name of the link and it followed by the *type* tag. This latter specifies the type of joint, where type can be one of the following:[27]

- *fixed*: the simplest joint because merge two links, blocking them together.
- *revolute*: a joint that rotates along a specified axis and has a limited range specified by *limit* tag.
- *continuous*: similar to *revolute*, but without *limit* tag.
- *prismatic*: a sliding joint that slides along the axis with a limited range specified by the upper and lower limits.
- *floating*: allows motion for all 6 degrees of freedom.
- *planar*: allows motion in a plane perpendicular to the axis.

These two tags encase sub-tags as:

- *origin*: same role of link element.
- *parent*: the parent link in the joint.
- *child*: the child link in the joint.
- *limit*: contains the followings attributes:
  - *lower*: specifies the lower joint limit (radians for *revolute* joints, meters for *prismatic* joints).
  - *upper*. specifies the upper joint limit (radians for *revolute* joints, meters for *prismatic* joints).
  - *effort*: enforces the maximum joint effort (|applied effort| < |effort|).
  - *velocity*: enforces the maximum joint velocity.

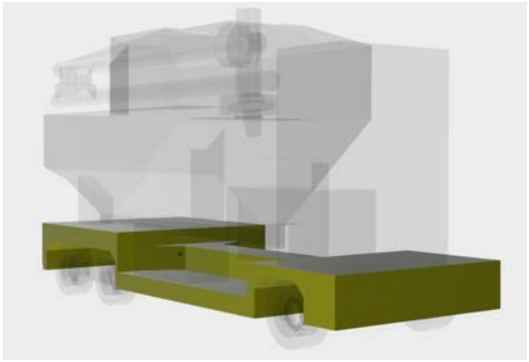
The parent-child relation establishes a hierarchy between links and specifies, for example, what should be taken as a reference (parent) during a particular type of movement (continuous, prismatic, revolute, etc.) of a link (child).

4.2.4 The Viewing in Gazebo

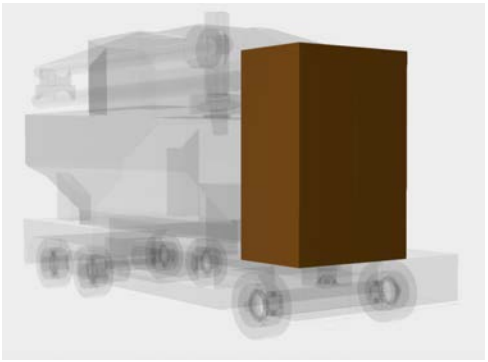
After defining the concept of link and joint, to visualize the vehicle model in Gazebo, it must associate a link to the different parts of the vehicle and establish connection joint.
However, it is necessary to make a clarification before proceeding: the available file is of type .dwg while the links can import, through the mesh sub-tag, only files of type .dae or .stl; therefore, it is necessary to carry out a conversion between file formats.
The procedure that is adopted can be summarized in the following points:

1. separate the main components of the vehicle, each assigned to a suitable .dwg file.
2. convert through Blender graphics software the respective files to the .dae format.
3. associate a link to each component through a *mesh* sub-tag.
4. establish connections between links through appropriate joints.

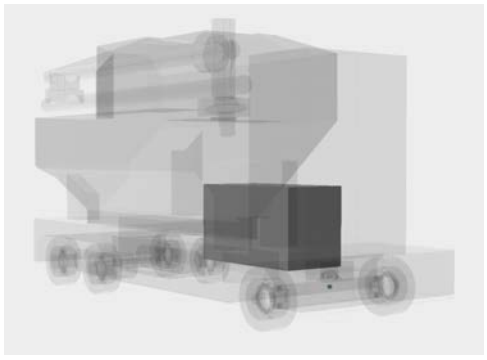
Taking as a reference Figure 4.8, it is possible to see the nine parts into which the vehicle has been divided: to underline the approximation of the engine and the two radiators, respectively, in big [Figure 4.8c] and small tanks [Figure 4.8d].
Each of these parts is assigned to a link with the appropriate name: the values of datasheet shown in the Table 4.1 are assigned to the values in the *inertial* tag.
Another possible solution could be to separate the moving parts from the original vehicle, i.e. wheels and steering, and to consider the rest as single fixed component: in this way, however, the detailed characterization provided by the *inertial* tag for each component would have been lost, since it would have been assigned a single link to the fixed component with a single *inertial* tag.
The merge between various links are made by the joints: most are fixed, except for wheel/hub and wheel/traction axle joints of continuous type, as the wheels must be free to rotate, and the hub/steering axle joint of the revolute type, as it is necessary to limit the steering of the wheel within a precise range of angles (*lower* = -45° and *upper* = 45°).



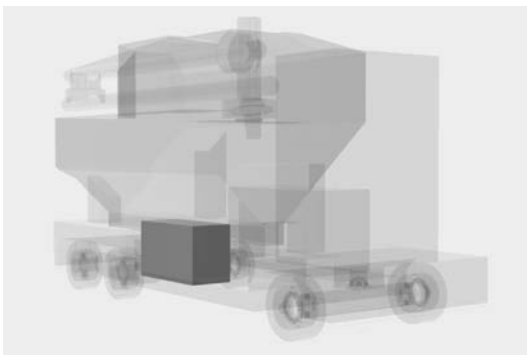
(a) Frame.



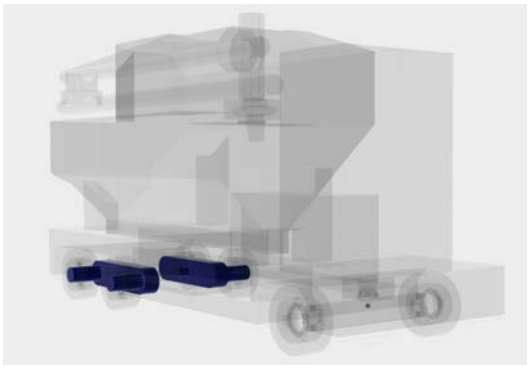
(b) Cockpit.



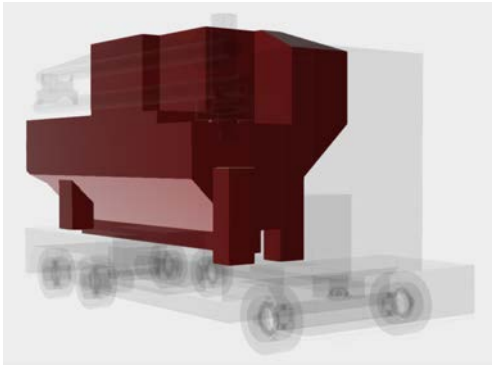
(c) Big tank.



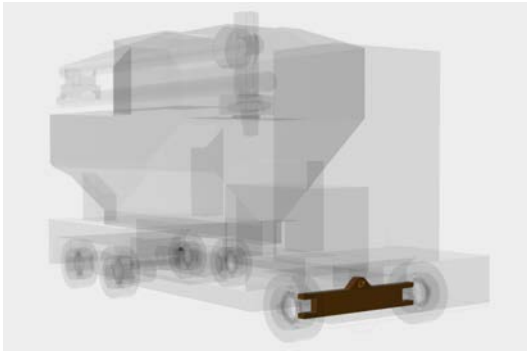
(d) Small tank.



(e) Rear traction axle.

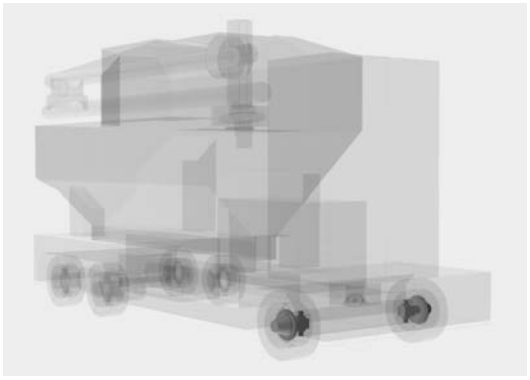


(f) Fluoride container.



(g) Front steering axle.

(h) Wheels.



(i) Hubs.

Figure 4.8: Components of fluoride feeder model.

Component	X	Y	Z	Weight	Ixx	Iyy	Izz
Frame	2.2	6.1	0.72	3143.09	11307.587	1522.908	12542.738
Cabin	1.4	1.3	2.236	662.98	509.869	465.222	329.16
Big Tank	0.86	1.8	1.075	1070	445.262	173.159	374.917
Small Tank	0.53	1.190	0.73	425.8	86	38.391	84.158
Traction Axle	0.747	0.36	1.5	429.78	17.736	87.943	79.785
(Empty) Container	2.2	5.028	2.83	3685.473	10410.335	409.827	9449.361
(Full) Container	2.2	5.028	2.83	14959.174	25648.297	9829.787	23811.725
Steering Axle	1.55	0.385	0.15	285.69	2.496	1.975	42.73
Wheel	0.72	0.228	0.72	91.05	3.376	5.984	3.376
Hub	0.338	0.32	0.32	63	0.342	0.867	0.863

Table 4.1: Parameters of vehicle components: cartesian measures [m], weights [kg] and moments of inertia [kgm^2].

Now, the vehicle is ready to be displayed in the Gazebo: the *roslaunch* command, executed via the terminal, indicates the .launch file containing the parameters to set and nodes to launch, including the .urdf file of the fluoride feeder model. The transformation from code to simulative vehicle is transparent to the user and it is beyond the scope of this thesis.

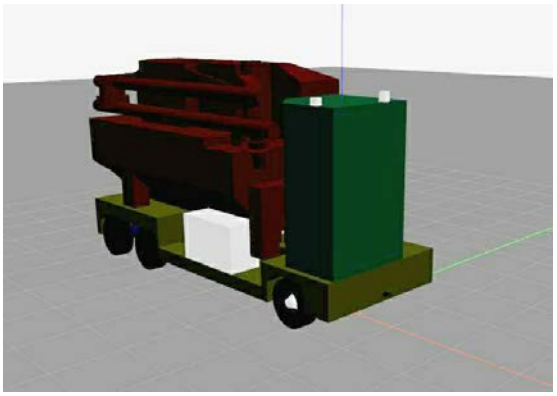
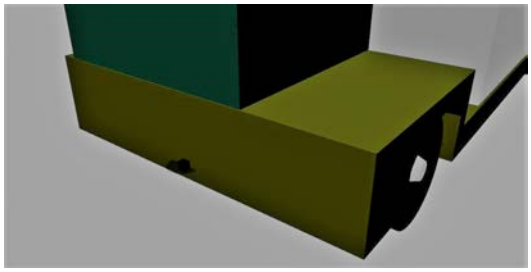


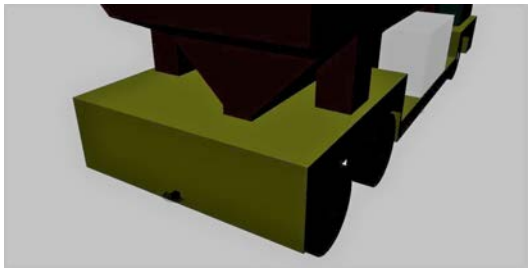
Figure 4.9: Fluoride feeder in Gazebo environment.

4.2.5 Lasers and Cameras

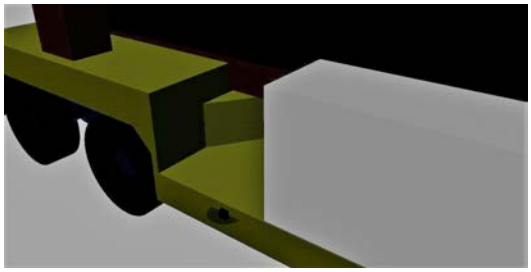
To the model of the vehicle previously described, in a second time, three cameras have been added on the cockpit roof and four lasers corresponding to the four sides of the vehicle.[Figure 4.10] The purpose is to make the vehicle autonomous in driving: through the adoption of these devices, it is able to perceive the presence of an obstacle, fixed or mobile. With appropriate algorithms which use the information coming from these sensors, the vehicle is able to brake or avoid the obstacle. In other words, the presence of these tools allows the vehicle to "see" the surrounding world and subsequently, through appropriate electronics and algorithms, to "reason" about the decision to be made, all in complete autonomy without the direct control of a worker.
In Gazebo, these sensors appear aesthetically like simple small cubes: the part of vision is managed independently through appropriate plugins that deal with converting images in the form of numerical data.



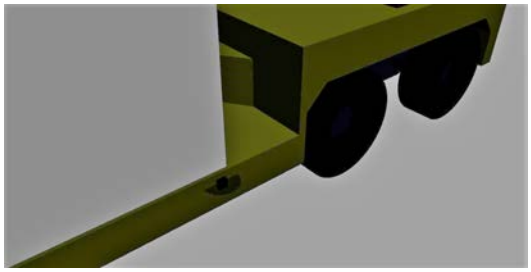
(a) Front laser.



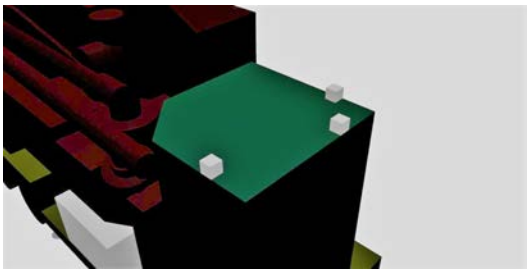
(b) Rear laser.



(c) Right laser.



(d) Left laser.



(e) Cameras on the cockpit roof.

Figure 4.10: Lasers and cameras in the model.

4.2.6 ROS Nodes and Topics

To make the vehicle movable, a group of ROS nodes must be implemented, each one specialized in a specific task.
Taking as a reference Figure 4.11, the nodes are the elements with a circular shape while the topics are the elements with a rectangular shape: the communication between nodes happens through the topics. Nodes and topics are connected by arrows that allow us to understand who is communicating with what to whom. The network of connections is called *rosgraph*.
In particular, the nodes that make up the graph are:

- */controller_spawner*: isolated node, since it has no connection with the other nodes, whose task is simply to generate the node */ackermann_controller*.

- */gazebo*: through the topics */joint_states*, */link_states* and */model_states* the model provides information on its position in the plant, its general speed and wheels speed. This node is the equivalent of an inertial platform equipped with a gyroscope, accelerometer and magnetometer, tools that provide information on roll, pitch and yaw. Moreover it can also say that the inertial platform also includes a hypothetical GPS platform, since the position of the vehicle is available.

- */simple_path_planning*: node that manages the route to follow for the vehicle. It receives the information from the node */gazebo* and, after processing it, communicates to the node */ackermann_controller* an appropriate command to move.

- */vehicle_state_publisher*: This node allows us to publish the state of the vehicle on the topic */tf* : it takes the joint angles of the vehicle as input and publishes the 3D poses of the vehicle links, using a kinematic tree model of the vehicle.

- */ackermann_controller*: node that manages the acceleration, speed and steering angle of the vehicle by adopting Ackermann steering geometry [Appendix C]: it plays an essential role in modelling because it allows the vehicle to be very similar to the real one, as it makes front wheel steering possible and the turning of the individual wheels. In the other 4-wheel modelling present in the ROS & Gazebo environment, only a linear or angular speed is provided to the vehicle to move it, assuming a three-dimensional reference system in the center of mass.

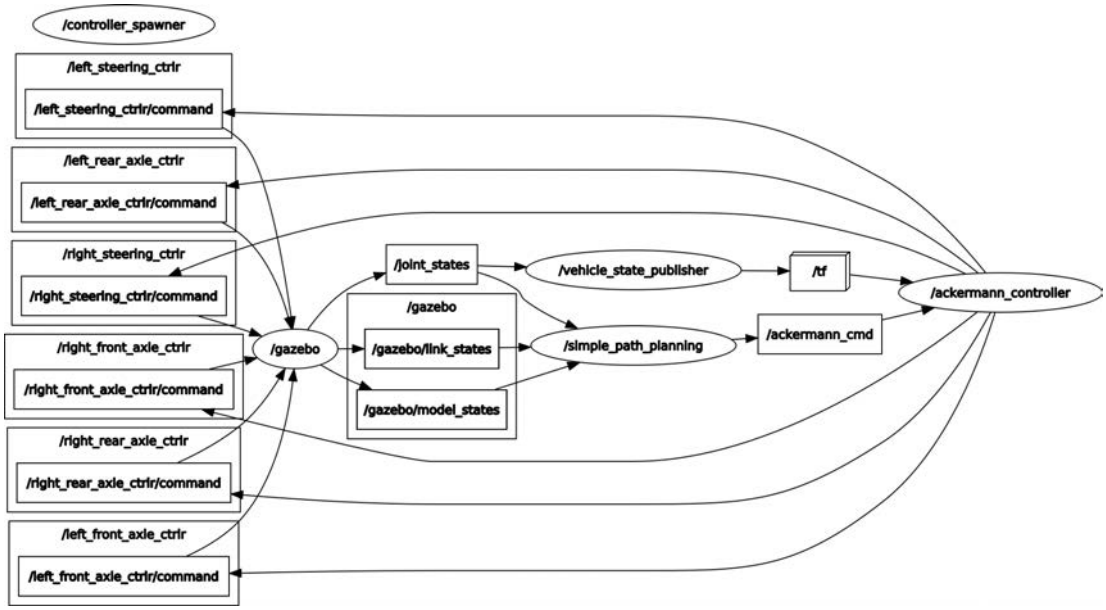


Figure 4.11: Nodes and topic.

4.3 Scenario Evaluation Model

In order to perform Gazebo simulations, so that they are as realistic as possible, it is necessary to define a model for the 3D world. The latter represents the environment where the vehicle can move and complete the tasks assigned to it. In particular, for the purpose of this project, a simplified version of an aluminium production plant has been adopted. This choice does not affect the reliability of the simulation and allows to reduce the computational burden required by the latter.
Drawing inspiration from real aluminium smelters as in Figure 4.12, the 3D world can be built using three main elements:

- *Potroom*: already introduced in Section 2.4.1, it is the building where the aluminium production process is performed. Each of the potcells, contained within it, could represent the final point for the path that the vehicle has to follow.

- *Aluminium Fluoride Storage and Handling (AFSH)*: it is the building where the vehicle's tank is filled with the aluminium fluoride, which will then have to be distributed in the potcells. According to the given tasks' schedule, the AFSH may represents the starting or an intermediate area through which the vehicle's path has to pass.

- *Streets*: they define the areas that the vehicle can cover to move from one building to an other. In general, the streets constitute the main parts of the vehicle's path.

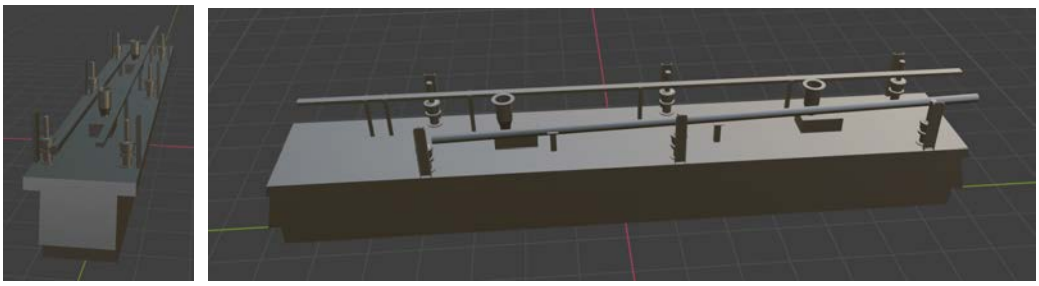
Figure 4.12: Sohar's smelter at Sultanate of Oman (Arab Peninsula).

In the following subsections, a brief description of how these three elements have been implemented is given.

4.3.1 Potrooms

As it has been written previously, in general a potroom can be described as a rectangular building that contains a sequence of potcells, placed in series or parallel, and one or more hallways that allows vehicles and human operators to easily move from one point to an other inside the room. Additionally a crane could be present to move heavy loads. Since for the accomplishment of the objectives of this analysis such device is not influential, it has been discarded in the potroom modelling phase. For the same reason also the design of the building's roof has been neglected.
At this point, in order to obtain a 3D representation of the potroom, the 3D computer graphics software Blender [Section 4.1.5] has been adopted. The design procedure can be divided in three phases:

- *Potcell design*: first, a model for a potcell need to be obtained. Therefore, a 3D rendering of the object, kindly provided by Techmo company, has been imported into the aforementioned software. The potcell's local reference frame has then been moved in order to make the positioning of the object inside the building easier. Furthermore a metal texture has been added in order to get a more realistic visualization of the model. The result of this step can be seen in Figure 4.13.



(a) Frontal view. (b) Lateral view.

Figure 4.13: Potcell model.

- *Room design*: in this phase it is desired to get a model for the building. To this aim it is necessary to define the walls and room: about the first ones, it has been obtained as the union of parallelepipeds of different dimensions. A part of these is in contact with the ground while another part is raised, in order to simulate the shape of the doors entering and leaving the room. In order to get a more realistic visual of the result, a texture similar to the one of a concrete wall has been added to each defined element.
  About the floor, it has been simply defined using a rectangular plane with the same dimensions of the building plan. Then, an asphalt texture has been added to the floor model in order to obtain a better visual result. The room obtained from this procedure can be seen in Figure 4.14.

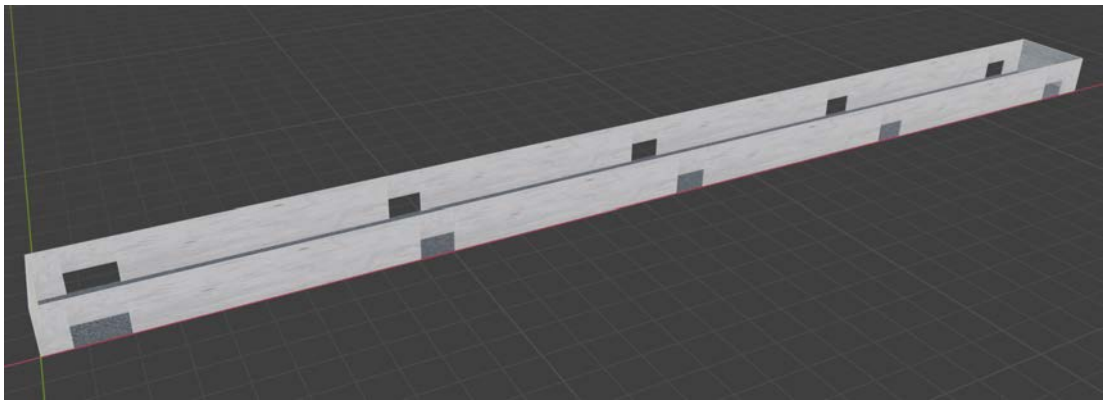


Figure 4.14: Empty potroom model.

- *Potrooms design*: in this last phase the results of the two previous steps have been added in order to obtain the final 3D model of the potroom. In particular 40 potcells have been inserted in parallel in the room model, as depicted in Figure 4.15. This arrangement allows to obtain an area that can serve as a large hallway.

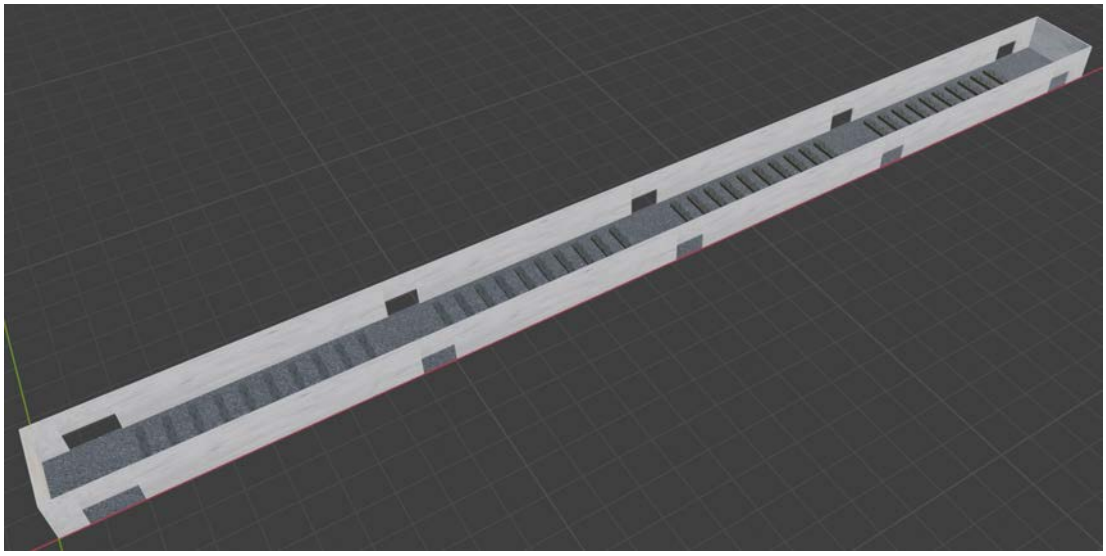
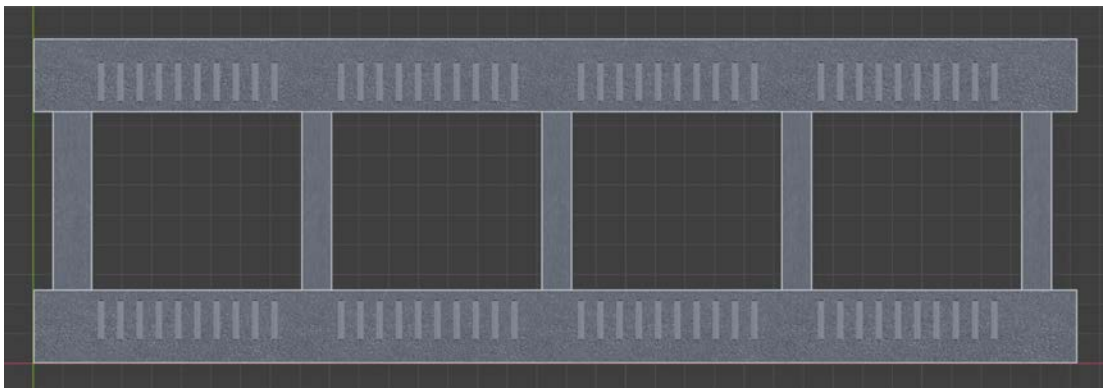
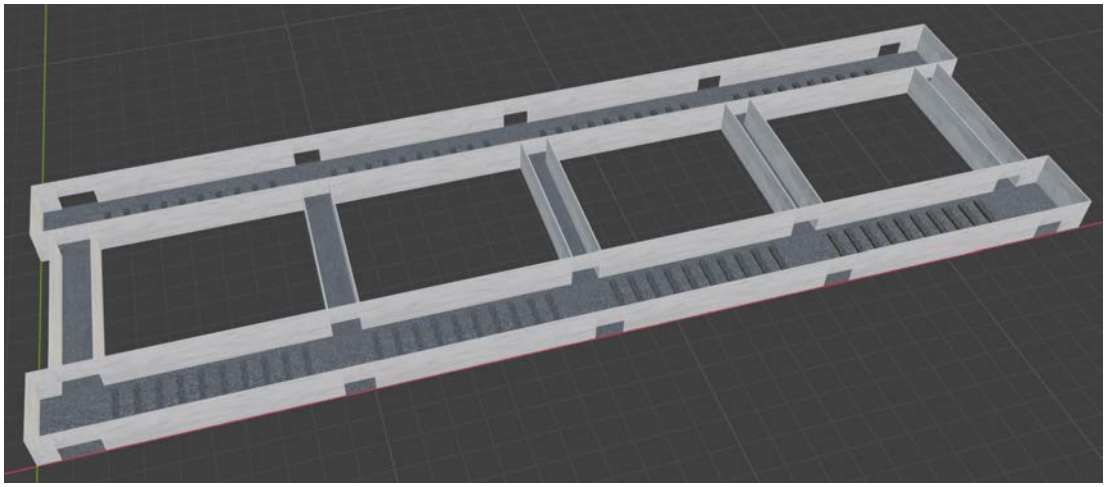


Figure 4.15: Potroom model with potline.

In order to get a more rich simulation, it has been decided to include in the world representation a second potroom, identical and connected to the first one through two long hallways. Both have been implemented using the same procedure used to define the room during the second step. The final 3D model of the potrooms is visible in Figure 4.16.



(a) Top view.



(b) Panoramic view.

Figure 4.16: Double potroom model.

4.3.2 AFSH

The *Aluminium Fluoride Storage and Handling* is a rectangular building, smaller than a potroom, in which the stocks of aluminium fluoride, useful for the production of aluminium, are stored. Such room presents a unique door that allows the entry and exit of vehicles. The design of such structure has been performed in a way that is similar to the one used in the second step of the potroom model definition, neglecting the representation of not fundamental elements such as the roof and the aluminium fluoride's stocks. The result of this phase is visible in Figure 4.17.

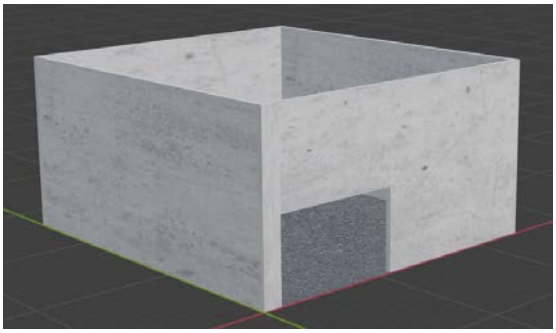
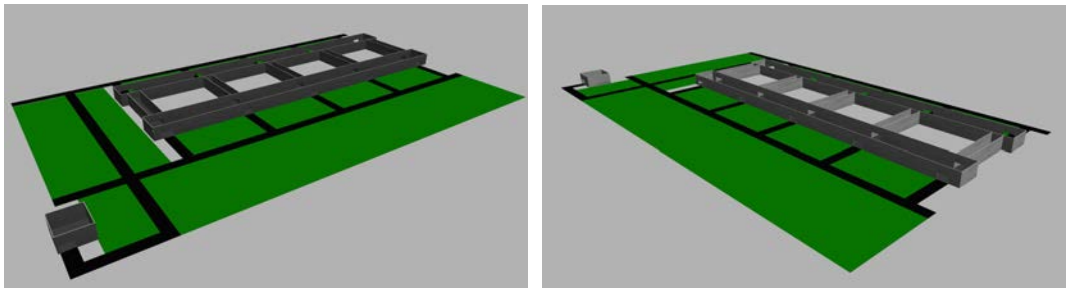


Figure 4.17: AFSH model.

4.3.3 Streets and External Environment

To increase the reality of the simulation and, in particular, to obtain a more suitable outdoor environment, a 3D external map is created.
Firstly, in order to not overload the simulations, are defined only the roads which link the AFSH to the potrooms and the *no-street areas* where the fluoride feeder vehicle can not pass through; all the not influential elements such as buildings, trees and rivers are not taken into account. An easy way to design the elements of the map is to define them as a concatenation of rectangular planes: black for the roads and green for the no-street areas.
Finally, the previous buildings as potrooms and AFSH are added to obtain the final simulation map. The addition of Wi-Fi transmitters complete the map and they allow to simulate different processes as, for example, path planning, obstacle avoidance and localization [Figure 4.18].



(a) Frontal panoramic view.

(b) Rear panoramic view.

Figure 4.18: 3D world map for Gazebo simulation.

5 Experimental Results

This chapter presents the experimental results on path planning obtained with the MATLAB software. The attention is directed to the generation of the path: the AGV is approximated as a 2D material point, thus leaving out all the kinematic and dynamic aspects of the vehicle, while importance is given to the algorithms as the graph-based ones (A*, Dijkstra's, D* Lite and Focused D*) and the sampling-based ones (RRT, RRT* and BIT*), plus the bonus algorithm FMM. They are evaluated in terms of the time taken to provide the result and type of route generated. The Original D*, presented in chapter 3, has not been used for simulations because the other two versions are considered to be much more efficient and easy to implement.
The scenario has a key role in path planning, therefore two totally opposite scenarios have been chosen to evaluate the simulations: a purely ideal one (a simple maze) and an almost real one (the potroom).

5.1 Simulations with an Ideal Scenario

The world of autonomous driving is complex and requires overcoming many difficulties and inconveniences, only to arrive at simple acceptable results. A valid approach to tackle this problem, but more generally many other problems of various order and gender, is to divide the main problem into sub-problems, which are easier to solve. To this, we often add a series of assumptions that allow us to obtain quite satisfactory results.
Following the previous statements, one of the main sub-problems of the autonomous driving of a vehicle is the path, more or less rigid, to follow. In this case, the approximation of the vehicle model to 2D material point, with the consequent elimination of its kinematics,

allows to easily evaluate the various algorithms present in the literature and previously mentioned.
A first and simple test to evaluate the effectiveness of these algorithms in path planning is the following: given a 2D scenario, find a path that connects an initial position with a collision-free final position with possible obstacles.
The algorithms, to pass the test, receive in input any map in .pgm format and elaborate two types of trajectories: an ideal and a real one, actually travelled by the vehicle. These two paths differ in distance from obstacles: in the case of a fluoride feeder vehicle with a length of 6.1m and a width of 2.2m (frame dimension in Table 4.1), the route must follow a certain distance from the obstacle to avoid collisions.
The format name .pgm is an acronym derived from *Portable Gray Map* and represents a grayscale graphic image: algorithms have been imposed to receive this type of file, in order to easily identify the free zones and the occupied zones. Imagining a decimal gray scale that goes from 0 to 1, it is easy to associate, for example, to 1 the black colour (an obstacle) and to 0 the white colour (free space). Therefore the conversion from any map format to the format in question is necessary for the correct interpretation of the scenario.
Going into the detail of the simulation, we initially considered an ideal scenario: a simple labyrinth, presented in Figure 5.1 of 593×545px dimension. The choice is justified by the fact that in this first approach we simply want to verify the performance of the algorithms, without pretensions to the reality of the simulation.

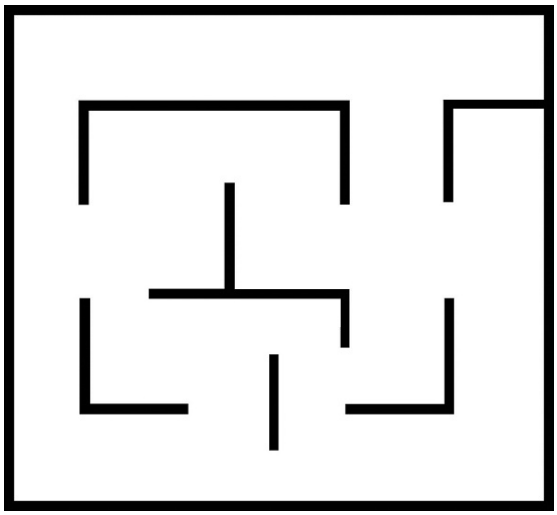
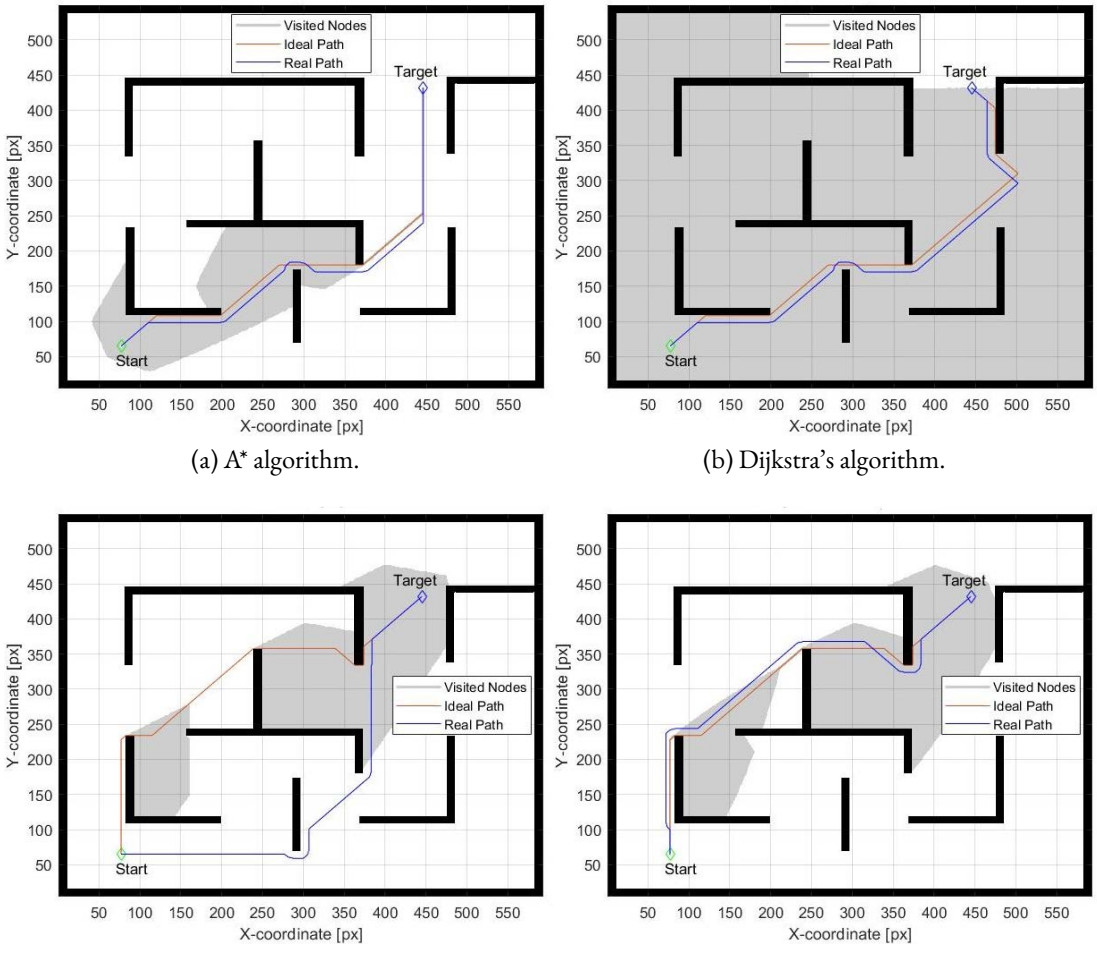


Figure 5.1: Ideal scenario: 2D labyrinth.

Providing it as input to the graph-based algorithms, the results are obtained in Figure 5.2. We initially noticed the two paths, ideal and real, mentioned above that have the X-Y coordinates as start position [77, 60]px and as target position [445, 427]px.

In this case, fixed positions are used to precisely compare the results of the algorithms: in reality, each algorithm accepts any position on the map, as long as it does not coincide with an obstacle.
Pixels have been chosen as units of measurement, coherently with the fact that images are processed: in the case of using the path processed by the algorithms on a purely real scenario, it can be assumed that 1px coincides with 1m. Therefore, by adopting this assumption, the construction of the map assumes great importance: precisely measuring the scenario in meters and subsequently building a map according to which 1px = 1m, the path can be used by a real vehicle, adapting the kinematics to the sections in which it must go straight, steer or brake.



(a) A* algorithm.

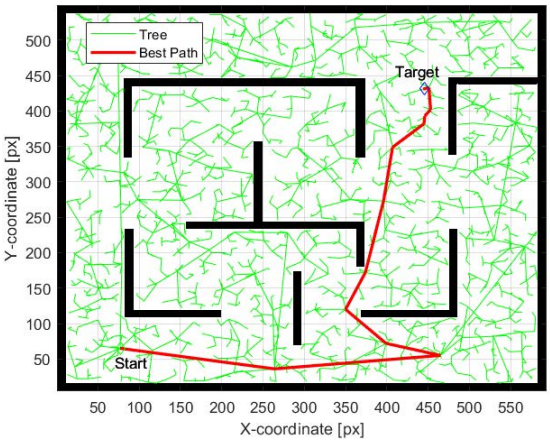
(b) Dijkstra's algorithm.

(c) D* Lite algorithm.

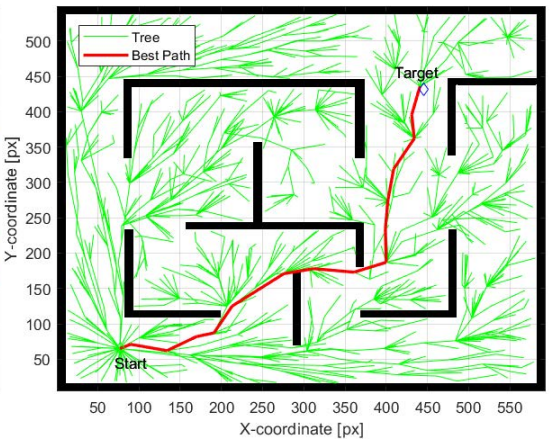
(d) Focused D* algorithm.

Figure 5.2: Path planning in the ideal scenario, employing graph-based algorithms.

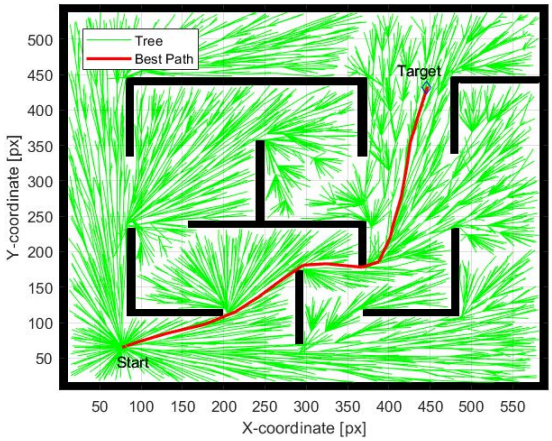
Subsequently, gray areas can be seen in the figures that are referred to in the legend as *visited nodes*: they represent the nodes of the hypothetical graph that are inspected by the algorithm to determine the best path. These areas are quantitatively different from algorithm to algorithm: starting from this difference, we can begin to qualitatively discuss the performance of this type of algorithm.
At first glance, we can see the big difference between the number of nodes visited by Dijkstra's algorithm and the other algorithms. The reason is simple: Dijkstra's one does not have the heuristic function *h(n)* of the cheapest path from any node to a target node. The lack of this information leads to consider the almost total scenario, as long as it need visited nodes does not coincide with the target node.
A second difference, less evident, is the arrangement of the visited nodes between A* and D* (both versions): we can note how in the A* algorithm the gray zone is concentrated around the start position while in D* it is concentrated around the target position. This difference derives from the algorithm principles since the A* starts to process starting from the start node instead the D* starts from the target node. The advantages and disadvantages of this policy will be discussed later, considering also some given numbers.
A third difference concerns the real path of the two versions D*: while in the Focused D* the path follows the ideal one, in D* Lite the path is totally different. The hypothesis is that, taking into account the distance of the obstacles, D* Lite chooses a different real path because the one similar to the ideal path is no longer the minimum path, according to the algorithm policy. The Focused D* instead limits itself to elaborating a path similar to the ideal one because it uses a heuristic to focus the propagation of RAISE and LOWER states towards the vehicle and not towards the path, in a similar way to the A*.
Instead supplying it in input to the sampling-based algorithms, the results in Figure 5.3 are obtained. In this case it has only one path from the start position to the target position: the choice is due to the fact that this type of algorithms is based on the sampling of the scenario and can generate a different path from simulation to simulation. Consequently the hypothetical real path travelled by the vehicle would be totally different from a hypothetical ideal path, since the position of the samples changes from simulation to simulation.
As start position, the X-Y coordinates [77, 60]px are adopted while for the target it has been decided to use a precision window (X-Y coordinates [445$\pm$5, 427$\pm$5]px), since, based on samples, the algorithm does not took too many resources to arrive at the result.
In this case, instead of the visited nodes, we find a green tree. It is the result of the union of the various samples taken from the scenario: meaning as a sample an X-Y coordinate point in the map, the algorithm creates a tree that does not collide with obstacles, that is, the edges do not cross obstacles. At first glance, it can be seen that the trees are quantitatively different from algorithm to algorithm. Starting from this difference, we can begin to qualitatively discuss the performance of this type of algorithm, as done for graph-based algorithms.
The first major difference can be observed between the RRT and the others: the first expands into *spider web* while the other trees expand into *comb*. This difference derives from the fact



(a) RRT algorithm.



(b) RRT* algorithm.



(c) BIT* algorithm.

Figure 5.3: Path planning in the ideal scenario, employing sampling-based algorithms.

that RRT does not have a *expansion radius* in which to consider the samples with the shortest distance: the union of the samples through the edges is random and takes into account only the non-collision with the obstacles. The consequences are evident on the best path, which is very irregular.
The other two trees, on the other hand, using a specific *expansion radius* of 50px in this specific simulation: if we look closely at Figure 5.3b-5.3c, we can see that the edges starting from a node describe an arc.
A second difference can be seen from the comparison between the RRT* and BIT* tree: the former is less dense, but produces a slightly less regular path than the latter. The cause is due to the fact that the BIT* considers batches (ellipses in our case) that widen and shrink depending on the distance between start and target position. This means a better focus on the best route.

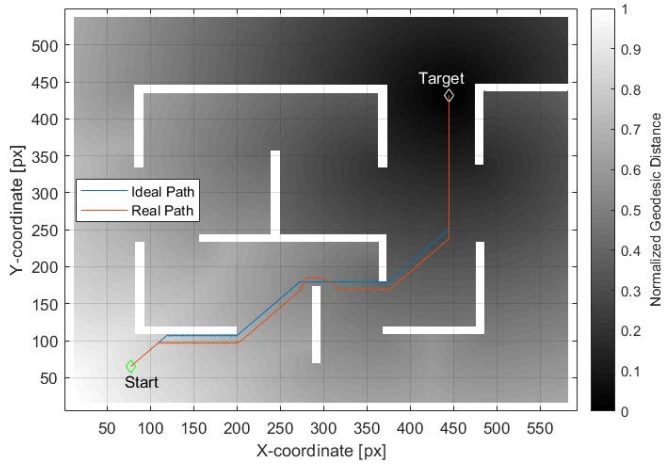


Figure 5.4: Path planning in the ideal scenario, employing Fast Marching Method.

Taking Figure 5.4 as a reference, we consider the FMM algorithm. The first thing we notice is the grayscale that starts from the black area of the target position and arrives at an almost white area of the start position: more precisely, we notice the concetrical circles that converge on the target position. They are the wave fronts introduced in the chapter 3, during the presentation of the algorithm.
In detail, the FMM calculates the *normalize geodesic distances*: distances obtained by readjusting the time T of arrival of the wave from the point of generation to any other point on the map. From the legend of the figure, we can see how 1 represents the maximum distance, while 0 indicates distance that coincides with the generation point: in our case, the latter is the target position. A three-dimensional interpretation of the grayscale can be the following: the wave fronts create a sort of "funnel" which has as its global point of minimum the target position while as start position any other position. At this point, it is easy to understand the

procedure of the gradient descent technique: by calculating the gradient at each point on the map, are obtained the normal directions to the concentric curves, which point towards the origin point. Specifically, a series of points are obtained which give the path from the start position to the target position, being the lowest point existing. It can be said that the approach is very similar to the graph-based algorithms: while on one side the visited nodes expand, on the other the wave fronts expand.

	Steps		Distance		Time	
Path:	Ideal	Real	Ideal	Real	Ideal	Real
A*	548	567	625.7	648.4	1.390	1.395
Dijkstra's algorithm	548	567	672.1	694.8	9.637	7.691
D* Lite	558	597	650.6	658.5	2.901	2.277
Focused D*	558	586	650.6	687.7	3.497	3.911
RRT	13	-	876.4	-	33.87	-
RRT*	17	-	629.0	-	416.7	-
BIT*	19	-	600.1	-	5413	-
FMM	500	572	726.1	762.5	0.128	0.103

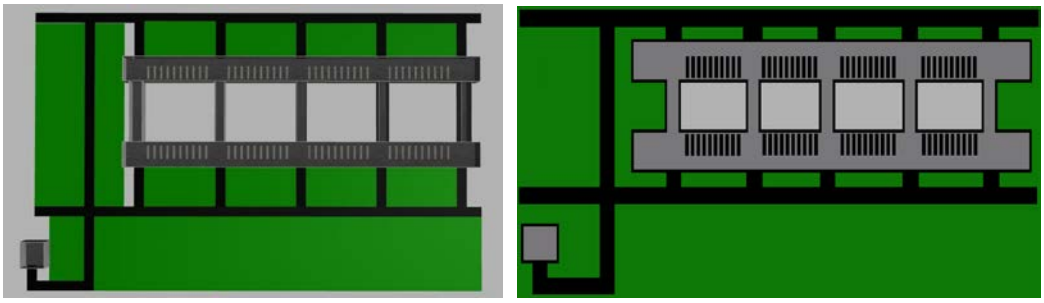
Table 5.1: Performances of graph-based algorithms, sampling-based algorithms and the FMM algorithm in the ideal scenario: steps to arrive to the target position, distance [px] of path between start and target position and the computational time [s] employed by the algorithm to produce the path.

Continuing the comparison between algorithms, we consider the data reported in the Table 5.1, moving therefore on numerical type performances.
Analysing the *steps* field, one immediately notices the great disparity between the types of algorithms: while the graph-based ones and FMM require a high number of nodes to reach their destination, the sampling-based ones require few samples to build a more or less acceptable path. The cause of this trend derives from the different policies, underlying the algorithms. The ideal path with fewer steps turns out to be that produced by the RRT, however from the qualitative point of view it is scarce because it is not regular; instead the real path with fewer steps is produced on a par by A* and Dijkstra's algorithms.
Analysing the *distances* field, we immediately notice the "path irregularity" characterizing the RRT, as expressed above: it is the longest path to travel, despite having the least number of steps, index of an algorithm that is easily found in path, but in rough ways. Instead, the smaller distances are represented by the BIT* for the ideal route and by the A* for the real route. We can note that, although A* and Dijkstra's have the same number of steps, the first produces a path a lower distance, due to the presence of the heuristic function *h(n)*.
Finally, analysing the *times* field, we can see how graph-based are very quick to generate a path, unlike sampling-based: even, BIT* can take an hour and a half. In reality, the fastest algorithm to generate the path is FMM algorithm, even if in terms of distance and steps it

does not excel, but the results are still good.
Although it can be said that A* represents the best trade-off for the performances, these considerations remain unfortunately still not concrete, because they are the result of simulations performed on an ideal scenario. To sum up definitively, we need to consider a scenario very similar to the real one, as it is done in the following section.

5.2 Simulations with a Real Scenario

After obtaining the first information on the performance of the various algorithms in an ideal scenario, it is now decided to use a real scenario, to obtain performances very similar to reality.
The scenario is the plant of potrooms and its external environment [Figure 5.5a]. The potrooms feature two parallel potlines with 40 potcells each: the latter are divided into groups of 10 each, with a side-by-syde arrangement. Through a road link it connects to the areas of the aluminium industry to allow its use: among them there is the AFSH, the fluoride feeder vehicle reloading area, from which the vehicle starts to supply the potcells with fluoride.
Taking into account that the size of the map is 1873×808px, it is logical to choose ASFH as start position and as target position any space adjacent to a potcell. In terms of coordinates, they are respectively at [109, 154]px and [1564, 677]px: as far as the target position is concerned, it has been allocated very far from the start position so that the resulting path represents a *worst case* outcome of crossing the potroom.

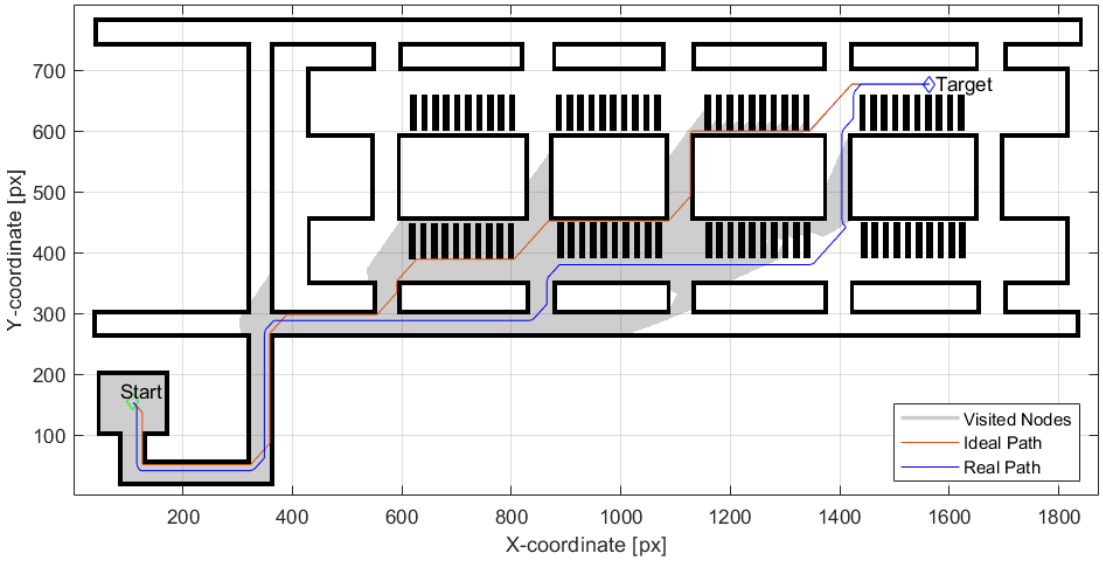


(a) Gazebo 3D world. (b) 2D map.

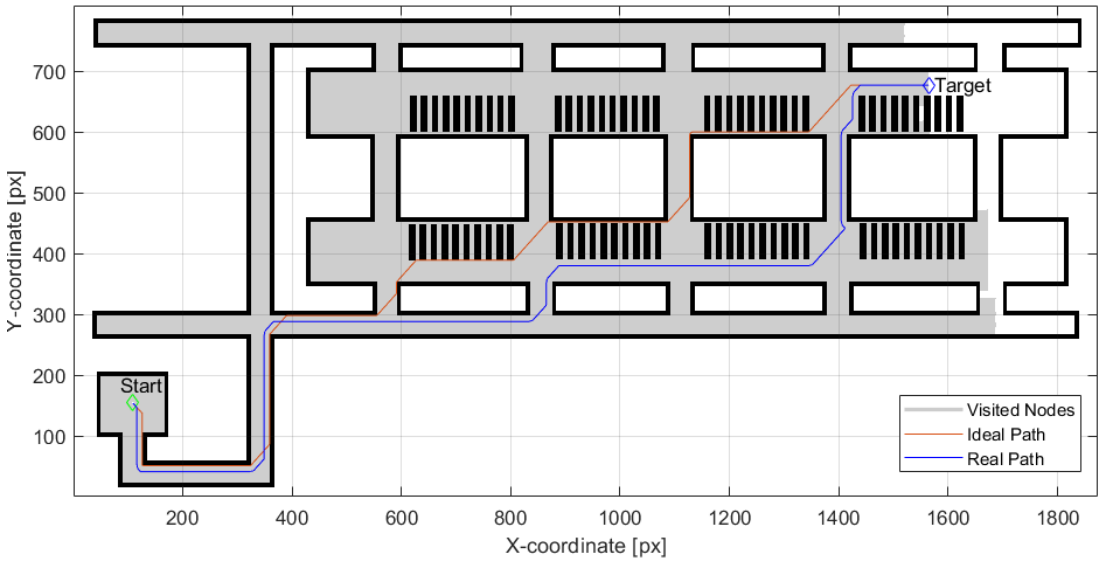
Figure 5.5: Potroom plant scenario: from Gazebo world to 2D map.

Following the analysis procedure of the previous section, we begin to qualitatively evaluate the results of the simulations in Figure 5.6, obtained from the graph-based algorithms.
A marked difference that is noted is the difference between in all four cases of a real path different from the ideal path: the motivation, already explained in the previous section, resides in the fact that the vehicle can not pass through deterrents or move too close to obstacles, as it would be with them. Therefore the true path, in this case, is quite different from the ideal one: for example, it can not pass through the narrow passage between the wall of the

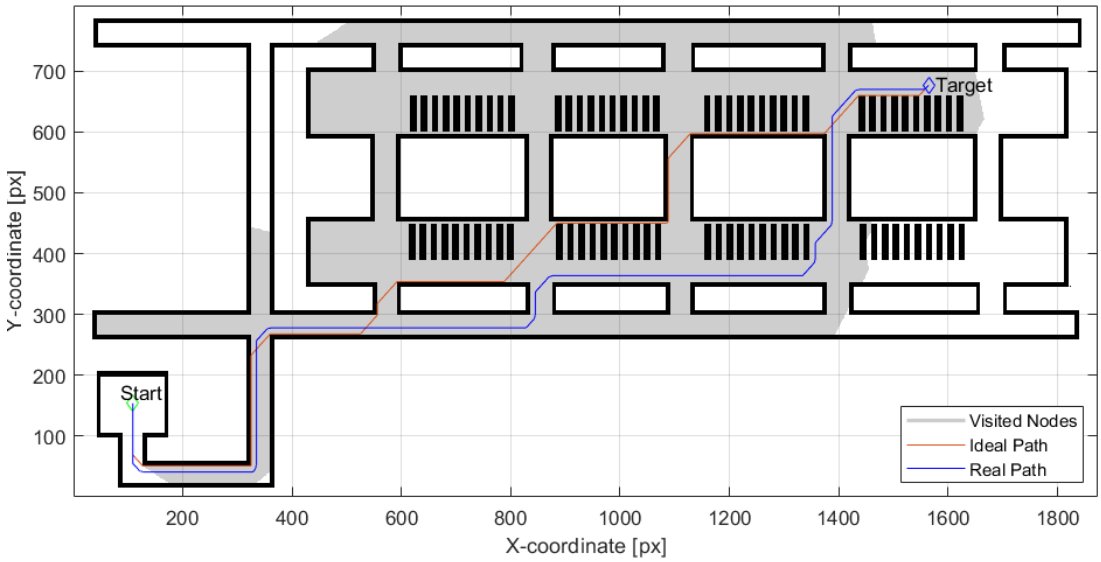
potroom and the potcells.
Regarding the nodes visited, the behaviour seen in the ideal scenario is confirmed: Dijkstra's algorithm remains the one that visits the most, while A* the one that visits the least. In this case it is also known how the nodes visited by the A* are oriented by the path, while those visited by D* (Lite and Focused), in this case, tend to be placed on a wider area. This derives from the fact that D*, when generating the path, also considers alternative routes to the one proposed, to be output only if new obstacles to the map are added.
As for the similarities, it should be emphasized that all four ideal paths have the same trend and do not take different paths. Also the real path does not seem very different between the four algorithms, except in the vertical section that crosses the two potlines, to the left of the target position: while for the first two algorithms the path widens from the right side of the road, in the other two it widens from the left side. The cause derives from the fact that the latter two generate the path starting from the target while the first two starting from the start position, as previously expressed.



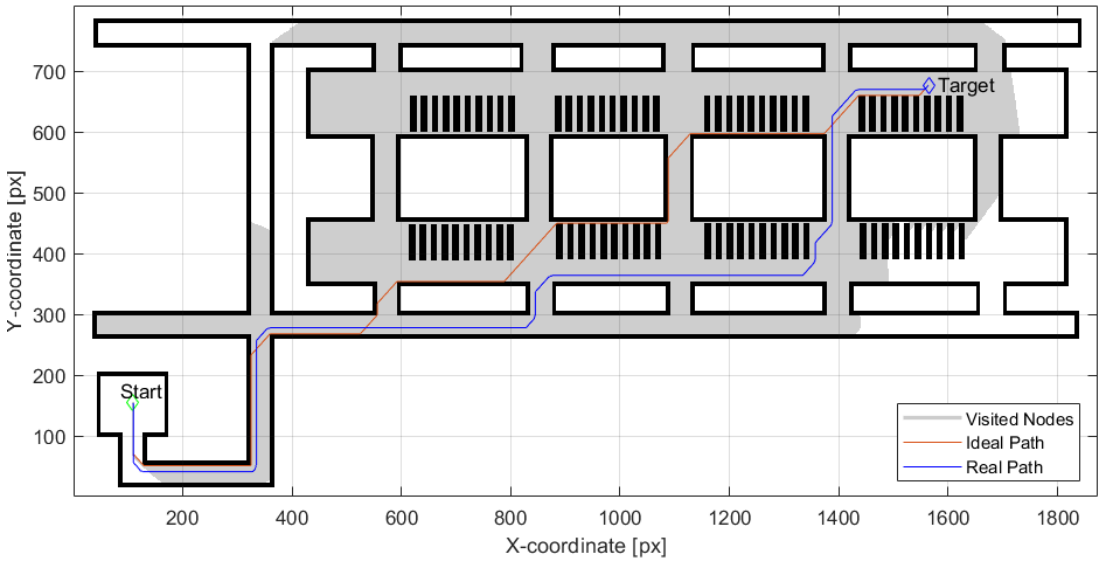
(a) A* algorithm.



(b) Dijkstra's algorithm.



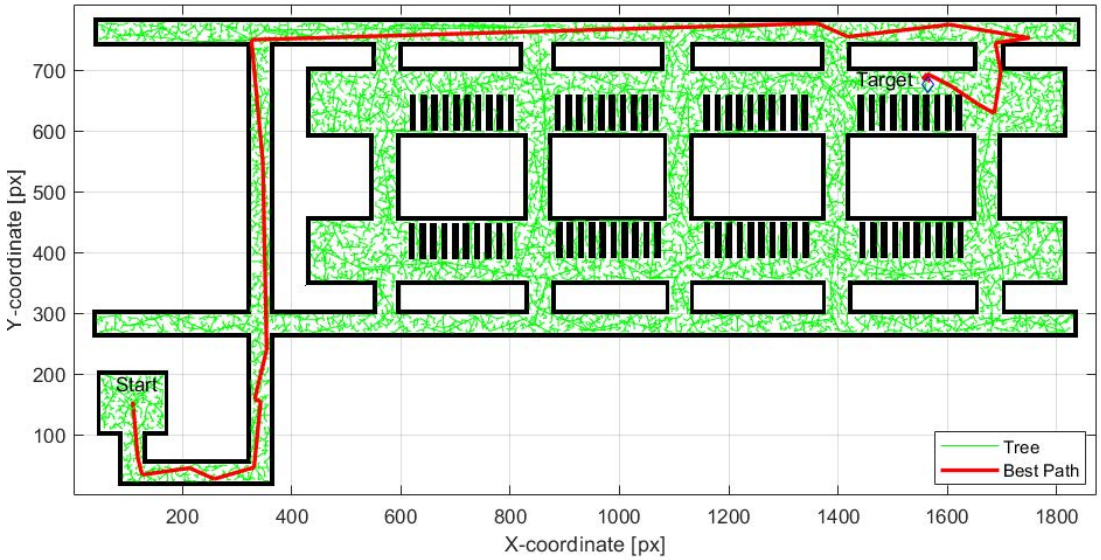
(c) D* Lite algorithm.



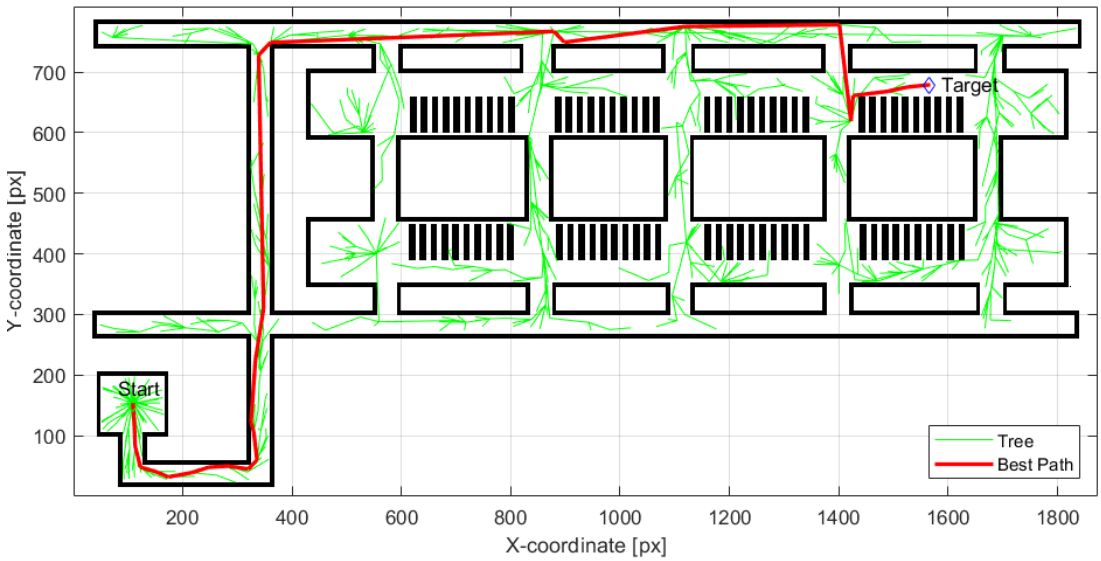
(d) Focused D* algorithm.

Figure 5.6: Path planning in the real scenario, employing graph-based algorithms.

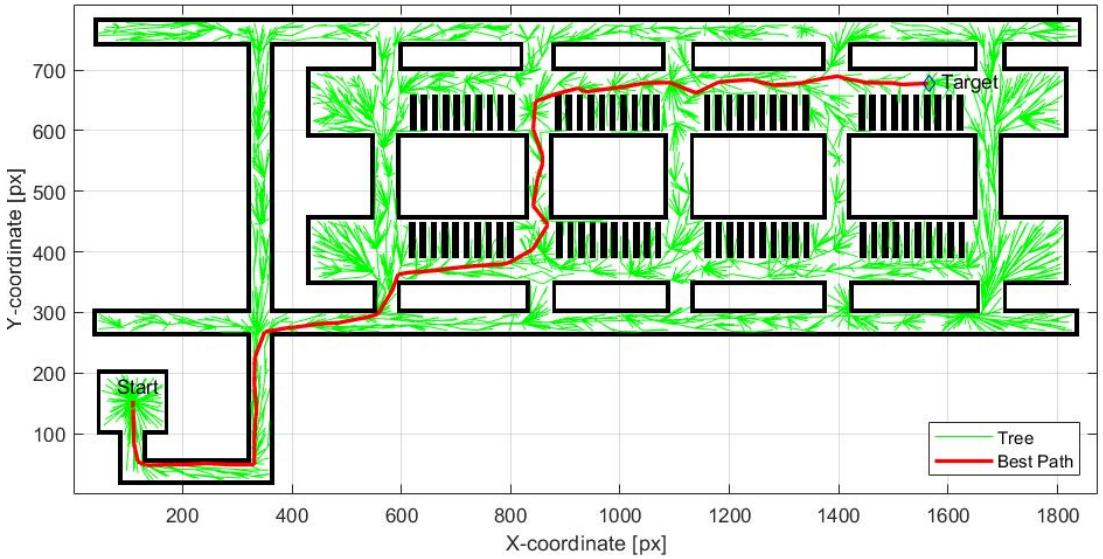
Considering now the sampling-based algorithms, we obtain the results in Figure 5.7. This time, for the target it has been decided to use a precision window with these X-Y coordinates $[1564 \pm 5, 677 \pm 5]$px.
Starting to discuss qualitatively the performance of this type of algorithms, we can confirm the first major difference, observed between the RRT and the other sampling-based algorithms, in the ideal scenario: first tree expands into *spider web*, while the others expand into *comb*.
Another difference that is confirmed is that between the RRT* and BIT*. The first is less dense than the second, underlining the diversity of path planning at the base of the two algorithms: in fact, the first is based on an expansion circumference while the second is based on an expansion ellipse; the elongated shape of the ellipse allows us to explore narrower areas or areas that can not be reached by a circumference.
In this case, however, with a scenario that presents more obstacles and complicated paths, the path offered by these algorithms is much less linear than the ideal scenario. Even the BIT* is very fragmented compared to the ideal case, a sign of the fact that the sampling-based algorithms work well in large and low-obstacle scenarios.
Regarding the FMM algorithm in a real scenario case [Figure 5.8], it is evident that the paths generated are similar to those of graph-based algorithms. It will be interesting to see what their performance is in numerical terms, compared to this type of algorithm.
Continuing the comparison between algorithms, we move on to numerical type performances, considering the data reported in the Table 5.2.



(a) RRT algorithm.



(b) RRT* algorithm.



(c) BIT* algorithm.

Figure 5.7: Path planning in the real scenario, employing sampling-based algorithms.

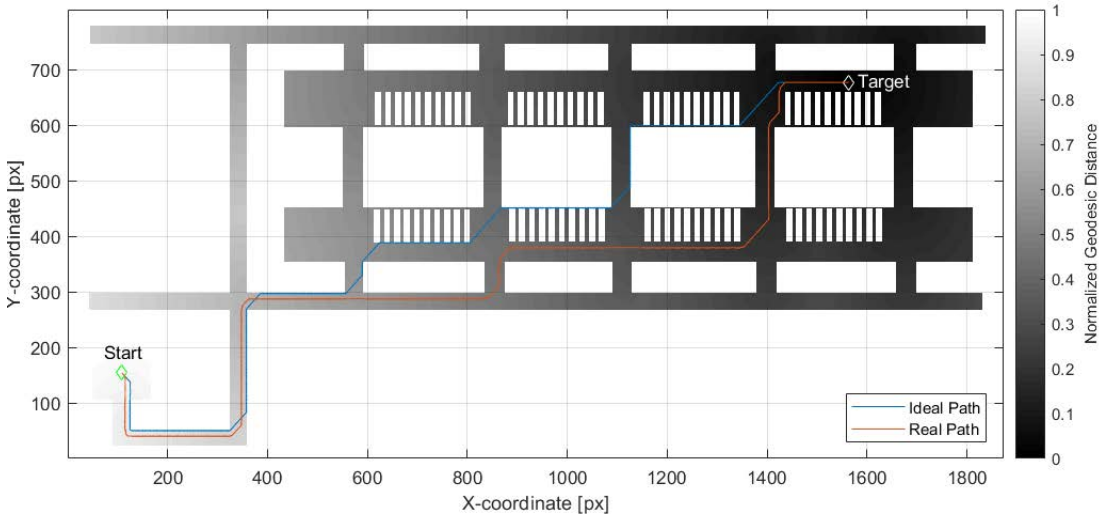


Figure 5.8: Path planning in the real scenario, employing Fast Marching Method.

Starting from the field *steps*, we can confirm the great disparity between the types of algorithms: about 2000 steps for graph-based, while about 30 steps for sampling-based. The best algorithms with the fewest number of ideal steps remains the RRT, however at the expense of the shape of the path, which is very irregular; while for the real path they are all equal (if we approximate the steps of the Focused D* to 2014), except the FMM algorithm that has a greater number.
In fact, in the field *distances*, we immediately notice the "irregularity of the path" that characterizes the RRT, as expressed above: it is the longest path to travel, despite having the least number of passages. Instead, the smallest distances are represented by the Focused D* both for the ideal path and for the real, an indication of the fact that this algorithm works well in generating paths in tortuous areas and with many obstacles.
A separate discussion should be made for the field *distances* of the FMM algorithm: the values are much higher than normal, both for the ideal field (2957.7) and for the real field (3205.3), although the path is very similar to that of graphs-based algorithms. The cause can be seen from the Figure 5.9: in the rectilinear sections there is no straight line but a zigzag pattern. This implies a distance much higher than the norm. A possible explanation derives from the fact that while in the ideal case we had relatively large spaces on which to expand the wave front, in this case with narrow spaces, the wave front is not very regular and consequently the normal direction produced by the gradient descent tends to vary many times along the way.

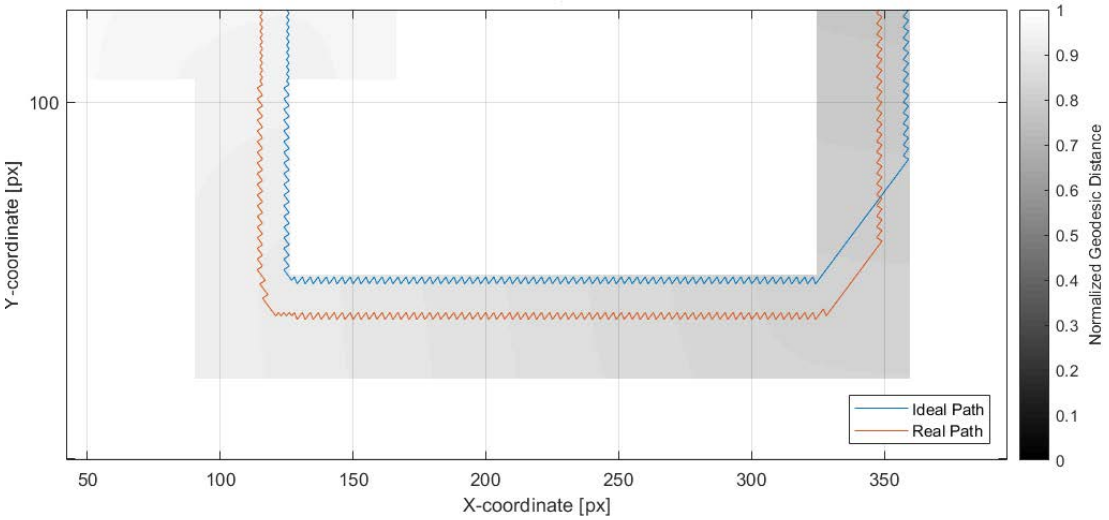


Figure 5.9: Zoom on FMM path for the real scenario.

Finally, by analysing the field *times*, it is confirmed that FMM algorithm is very fast in generating a path, unlike sampling-based and graph-based.
Therefore, with respect to what has been said for the ideal one, in the real scenario it seems

to partially overturn what has been said previously: A* continues to be a good algorithm while the Focused D* has highlighted important improvements if used on maps similar to real plants. Instead the FMM algorithm undergoes a decrease in performance, especially in the field *distances*, despite the fact that the path generation time is the best.

	Steps		Distance		Time	
Path:	Ideal	Real	Ideal	Real	Ideal	Real
A*	1845	2014	1984.8	2097.9	19.16	14.73
Dijkstra's algorithm	1845	2014	1984.8	2097.9	48.52	29.29
D* Lite	1845	2014	1984.8	2092.1	48.28	40.50
Focused D*	1843	2013	1983.6	2091.5	82.42	49.09
RRT	26	-	2831.1	-	877.4	-
RRT*	27	-	2426.3	-	378.4	-
BIT*	28	-	2078.3	-	3012	-
FMM	1860	2023	2957.7	3205.3	0.270	0.196

Table 5.2: Performances of graph-based algorithms, sampling-based algorithms and the FMM algorithm in the real scenario: steps to arrive to the target position, distance [px] of path between start and target position and the computational time [s] employed by the algorithm to produce the path.

6
Conclusions and Future Work

In this final chapter, we present the obtained conclusions of the work: we evaluate the fulfilled objectives of the project and present some possible future lines of work based on the open issues identified during the project.

6.1 Conclusions

Real world vehicle navigations have significant challenges such as path planning, obstacle avoidance, fault isolation, and system resiliance. AGV's navigation is a collection of design systems which is capable of handling all aspects of navigation requirement.
In this work, algorithms have been presented for the path planning of a vehicle in a known scenario. Starting from the known informations on the aluminium industry and the respective operating vehicles, a model of fluoride feeder vehicle has been created, through the use of various computer platforms as ROS or Autodesk Inventor. After having explained in detail the different algorithms suitable for path planning in the literature, they have been tested through the MATLAB environment to evaluate their efficiency and quality.
The results are very encouraging because in all cases the algorithms always manage to complete their task, although with some differences. Among the proposed algorithms, the A* and Focused D* performances emerge in the tests: both in ideal scenario and in real one, they manage to produce a minimum path between the initial and final position in a relatively short time. However it is preferable to use the second algorithm because it is valid also in partially known environments, i.e. known scenarios but with mobile obstacles, whose position is unknown: when the vehicle discovers an obstacle in movement through a camera or a laser, the Focused D* reschedules the path between the current position and the final

position. A separate discussion must be made for the FMM algorithm: although it has an interesting approach to the topic of path planning, its performance is not convincing in such a way as to consider it for a real application. The zigzag trend, for example, is an aspect in which the algorithm must be improved.
The main limitation of these simulations is the great distance from reality: in order to obtain more expendable results in a real 3D scenario, it is appropriate to consider a future continuation of what is stated in the thesis. Some development ideas are proposed in the following section.

6.2 Future Work

To improve the results obtained from the simulations of the algorithms, different tricks can be followed in the future of this investigation project, including:

- use a map with real measurements of the aluminium industries, in order to obtain a resulting path that can be easily implemented in the AGV.
- interface lasers and cameras with path planning algorithms, in order to increase the efficiency and reliability of the AGV in unknown scenarios.
- develop an algorithm that manages the kinematics of the vehicle in relation to a path to follow in input and test it in a real autonomous driving scenario.

A

The Bayer Process

The Bayer process was invented by the Austrian chemist Carl Josef Bayer in 1887 while working in Saint Petersburg (Russia) for developing a method to supply Alumina to the textile industry. The process started gaining importance after the invention of Hall-Héroult aluminium process and till today it is unchanged and used to produce nearly all the world's alumina supply.
Since the main component (30%-60%) of bauxite is alumina (other components are mixture of silica, iron oxide and titanium dioxide), the Bayer process is used to produce mainly alumina from the refined bauxite.
In practise, alumina is obtained into following steps:[4]

1. *Milling*: bauxite is washed and crushed, reducing the particle size and increasing the available surface area for the digestion stage. Lime and *Spent Liquor* (caustic soda returned from the precipitation stage) are added at the mills to make a pumpable slurry.

2. *Desilication*: bauxite with high levels of silica goes through a process to remove this impurity. In fact, silica can cause problems with scale formation and quality of the final product.

3. *Digestion*: a hot caustic soda solution is used to dissolve the aluminium-bearing minerals (Gibbsite, Böhmite and Diaspore) in the bauxite to form a *Pregnant Liquor* (sodium aluminate supersaturated solution).
   The dissolution reaction of gibbsite is:

$$Al(OH)_3 + Na^+ + OH^- \rightarrow Al(OH)_4^- + Na^+ \tag{A.1}$$

and the one of böhmite and diaspore is:

$$AlO(OH) + Na^+ + OH^- + H_2O \rightarrow Al(OH)_4^- + Na^+ \quad (A.2)$$

Conditions (caustic concentration, temperature and pressure) within the digester (huge vessel where chemical or biological reactions are carried out) are set according to the properties of the bauxite ore.
Ores with a high gibbsite content can be processed at 140°C, while böhmitic bauxites require temperatures between 200°C and 280°C. The pressure is not important for the process as such, but is defined by the steam saturation pressure of the process. At 240°C the pressure is approximately 3.5MPa.
The slurry is then cooled in a series of flash tanks to around 106°C at atmospheric pressure and by flashing off steam. This steam is used to preheat *Spent Liquor*. In some high temperature digestion refineries, higher quality bauxite is injected into the flash train to boost production. This *sweetening* process also reduces the energy usage per tonne of production.
Although higher temperatures are often theoretically advantageous, there are several potential disadvantages, including the possibility of oxides other than alumina dissolving into the caustic liquor.

4. *Clarification/Settling*: the first stage of clarification is to separate the solids (bauxite residue) from the *Pregnant Liquor* (sodium aluminate remains in solution) via sedimentation. Chemical additives, as flocculants, are added to assist the sedimentation process. The bauxite residue sinks to the bottom of the settling tanks, then is transferred to the washing tanks, where it undergoes a series of washing stages to recover the caustic soda, which is reused in the digestion process.
Further separation of the pregnant liquor from the bauxite residue is performed utilizing a series of security filters: the purpose of the filters is to ensure that the final product is not contaminated with impurities present in the residue.
Depending on the requirements of the residue storage facility, further thickening, filtration and/or neutralization stages are employed prior to be pumped to the bauxite residue disposal area.

5. *Precipitation*: in this stage, the alumina is recovered by crystallization from the pregnant liquor, which is *supersaturated* in sodium aluminate. The crystallization process is driven by progressive cooling of the pregnant liquor, resulting in the formation of small crystals of aluminium trihydroxite, commonly known as *Hydrates*, which then grow and agglomerate to form larger crystals.
The precipitation reaction is the reverse of the gibbsite dissolution reaction in the digestion stage. In fact:

$$Al(OH)_4^- + Na^+ \rightarrow Al(OH)_3 + Na^+ + OH^- \quad (A.3)$$

6. *Evaporation*: the spent liquor is heated through a series of heat exchangers and subsequently cooled in a series of flash tanks. The condensate formed in the heaters is re-used in the process, for instance as boiler feed water or for washing bauxite residue. The remaining caustic soda is washed and recycled back into the digestion process.

7. *Classification*: The gibbsite crystals formed in precipitation are classified into size ranges. This is normally done using cyclones or gravity classification tanks (a series of thickeners utilizing the same principles as settlers/washers on the clarification stage). The coarse size crystals are destined for calcination after being separated from spent liquor utilizing vacuum filtration, where the solids are washed with hot water.
   The fine crystals, after being washed to remove organic impurities, are returned to the precipitation stage as fine seed to be agglomerated.

8. *Calcination*: The filter cake is fed into calciners where they are roasted at temperatures of up to 1100°C to drive off free moisture and chemically-connected water, producing alumina solids. There are different calcination technologies in use, including gas suspension calciners, fluidised bed calciners and rotary kilns.
   The following equation describes the calcination reaction:

$$2\,Al(OH)_3 \longrightarrow Al_2O_3 + 3\,H_2O \tag{A.4}$$

The final alumina, a white powder, is the product of this step and the final product of the Bayer Process, ready for shipment to aluminium smelters or the chemical industry.

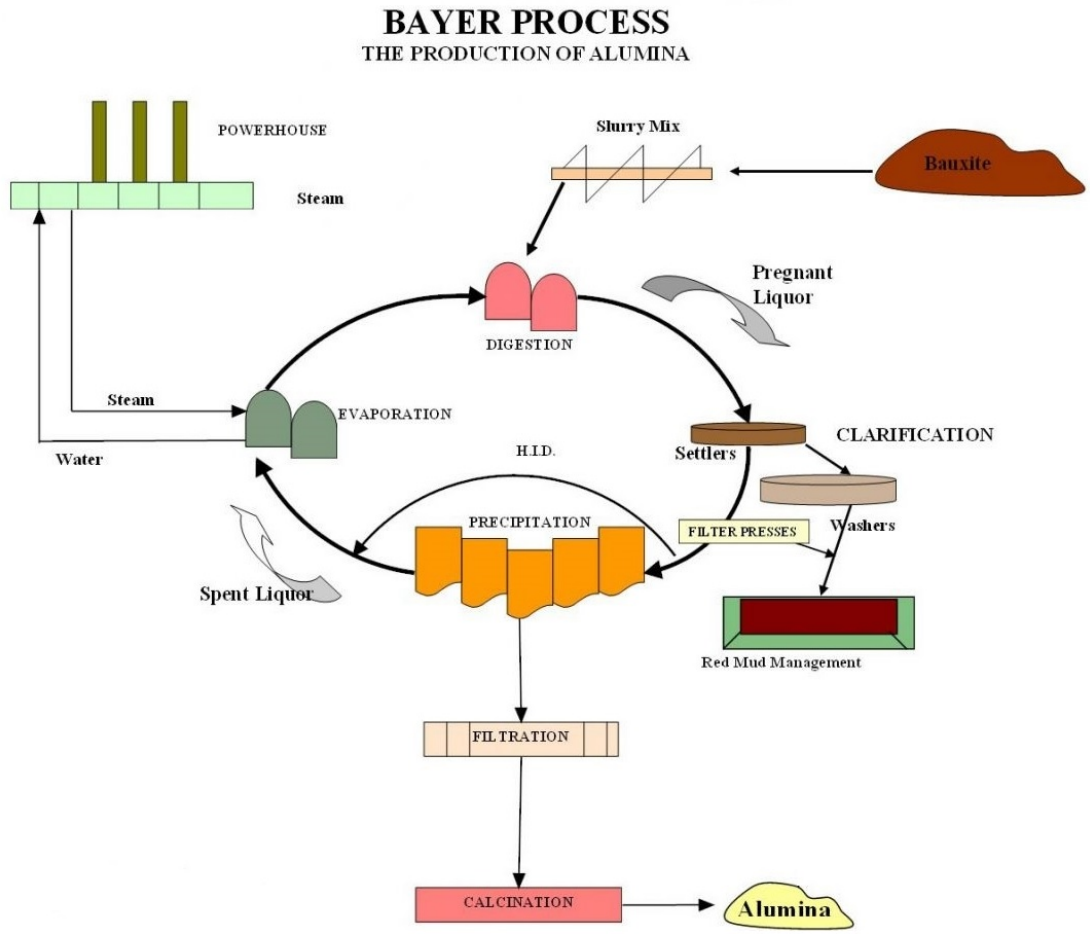


Figure A.1: Main steps of Bayer process.

B

The Hall-Héroult Process

The Hall–Héroult process was invented independently and almost simultaneously in 1886 by the American chemist Charles Martin Hall and by the Frenchman Paul Héroult, both only 22 years old. In 1888, Hall opened the first large-scale aluminium production plant in Pittsburgh. It later became the Alcoa corporation. In 1997 the Hall–Héroult process was designated as *National Historic Chemical Landmark* by the American Chemical Society in recognition of the importance of the process in the commercialization of aluminium.
Nowadays, the process is the major industrial process for smelting aluminium. It involves dissolving the Alumina in the molten Cryolite and electrolysing the molten salt bath, typically in a purpose-built cell. The process happens at 940–980°C and produces 99.5%–99.8% pure aluminium.[7]

Problem and Possible Solutions

Elemental aluminium can not be produced by the electrolysis of an aqueous aluminium salt because hydronium ions readily oxidize elemental aluminium. A possible solution is to use a molten aluminium salt, however the aluminium oxide has a melting point of 2072 °C, so electrolysis is impractical. A reasonable solution is the Hall–Héroult process where alumina is dissolved in molten synthetic cryolite to lower its melting point for easier electrolysis.

Background Theory

In the Hall–Héroult process the following simplified reactions take place at the carbon electrodes [Figure B.1]:[20]

- cathode: $Al^{3+} + 3\,e^- \longrightarrow Al$
- anode: $O^{2-} + C \longrightarrow CO + 2\,e^-$
- overall: $Al_2O_3 + 3\,C \longrightarrow 2\,Al + 3\,CO$

In reality much more CO_2 is formed at the anode than CO:

$$2\,Al_2O_3 + 3\,C \longrightarrow 4\,Al + 3\,CO_2 \tag{B.1}$$

Pure cryolite has a melting point of 1009°C ± 272.15°C but, with a small percentage of alumina dissolved in it, its melting point drops to about 1000°C. Besides having a relatively low melting point, cryolite is used as an electrolyte because it also dissolves alumina well, conducts electricity, dissociates electrolytically at higher voltage than alumina and has a lower density than aluminium at the temperatures required by the electrolysis.
Aluminium Fluoride is usually added to the electrolyte. The ratio NaF/AlF_3 is called *Cryolite Ratio* and it is approximately equal to 3 in pure cryolite. In industrial production, aluminium fluoride is added so that the cryolite ratio is approximately 2–3 to further reduce the melting point so that the electrolysis can happen at temperatures between 940°C and 980°C. The density of liquid aluminium is 2.3g/ml at temperatures between 950°C and 1000°C instead the density of the electrolyte should be less than 2.1g/ml, so that the molten aluminium separates from the electrolyte and settles properly to the bottom of the electrolysis cell. In addition to aluminium fluoride, other additives like lithium fluoride may be added to alter different properties (melting point, density, conductivity, etc.) of the electrolyte.

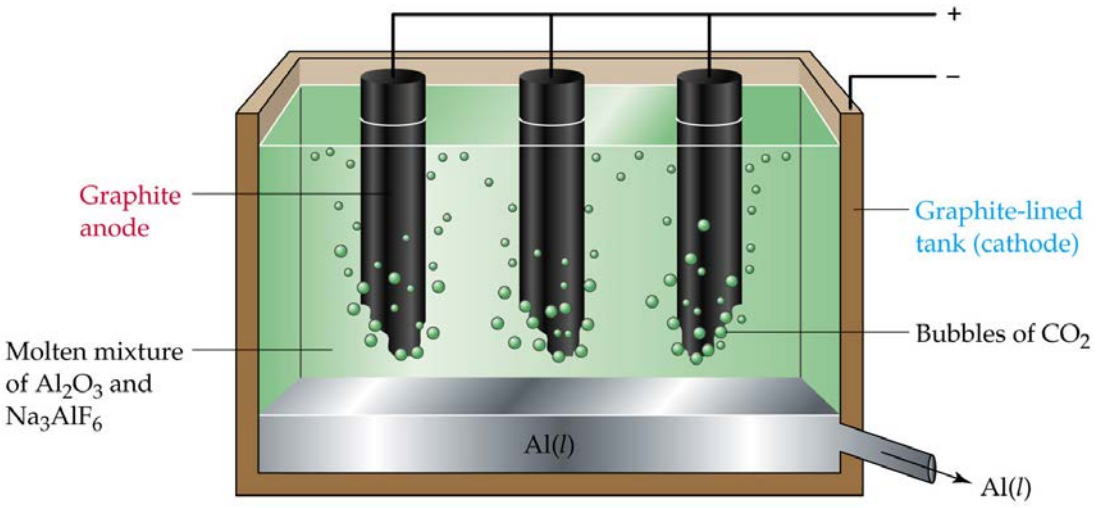


Figure B.1: Hall-Héroult process.[5]

The mixture is electrolysed by passing a low voltage (under 5 V) direct current at 100–300kA through it: this causes liquid aluminium metal to be deposited at the cathode while the oxygen from the alumina combines with carbon from the anode to produce mostly carbon dioxide.

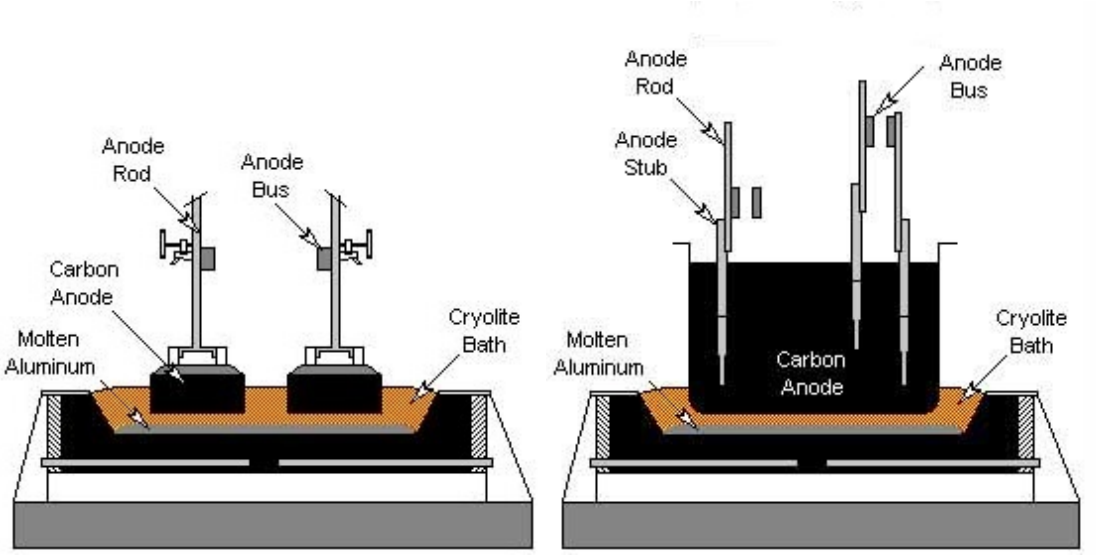
Cell Operation

Industrial cells are operated 24 hours a day so that the molten material does not solidify. Temperature within the cell is maintained via electrical resistance. Oxidation of the carbon anode increases the electrical efficiency at a cost of consuming the carbon electrodes and producing carbon dioxide.
While solid cryolite is denser than solid aluminium at room temperature, liquid aluminium is denser than molten cryolite at temperatures around 1000°C. The aluminium sinks to the bottom of the electrolytic cell, where it is periodically collected. The liquid aluminium is removed from the cell via a siphon every 1 to 3 days in order to avoid having to use extremely high temperature valves and pumps. Alumina is added to the cells as the aluminium is removed. Collected aluminium from different cells is finally melted together to ensure uniform product and made into e.g. metal sheets. The electrolytic mixture is sprinkled with coke to prevent the anode's oxidation by the oxygen evolved.
The cell produces gases at the anode. The exhaust is primarily CO_2, produced from the anode consumption, hydrogen fluoride from the cryolite, and flux. In modern facilities, fluorides are almost completely recycled to the cells and therefore used again in the electrolysis. Escaped hydrogen fluoride can be neutralized to its sodium salt, sodium fluoride. Particulates are captured using electrostatic or bag filters. The CO_2 is usually vented into the atmosphere.
Agitation of the molten material in the cell increases its production rate at the expense of an increase in cryolite impurities in the product. Properly designed cells can leverage magnetohydrodynamic forces induced by the electrolysing current to agitate the electrolyte. In non-agitating static pool cells the impurities either rise to the top of the metallic aluminium or else sink to the bottom, leaving high-purity aluminium in the middle area.

Electrodes

Electrodes in cells are mostly coke which has been purified at high temperatures. The materials most often used in anodes, coke and pitch resin, are mainly residues from petroleum industry and should be of high enough purity, so no impurities end up into the molten aluminium or the electrolyte.
What differentiate the two technologies is the way this carbon block is produced. There are two primary anode technologies, which differentiate the production way of carbon block:[21]

- *Prebake technology* [Figure B.2a]: the petroleum coke is mixed with pitch, which acts as a binder. Then, at this mixture, usually called *green paste*, it is given a parallelepiped



(a) Prebake technology (b) Soderberg technology

Figure B.2: The primary anode technologies.

shape with either a press or a vibro-compactor. The formed carbon block is then baked into furnaces in order to be transformed into a solid carbon block. The electric current arrives to the carbon block through a rod linked to it through nippels. A pre-bake potcell contains several single anodes (usually $14 \div 40$, mainly depending on the line current), which stay on the potcells for a fixed amount of days (generally 26 to 30 days). Then, before being completely consumed, they are removed together with the rod, and the remaining carbon reused to produce new anodes.

- *Söderberg technology* [Figure B.2b]: the basic idea is to eliminate the sub-plants which form, bake and join the carbon block with the rod. A Soderberg cell has only one big anode, housed in a steel container, which gives to the anode its shape. From the upper part of this container it is introduced the green paste. During its movement from the top to the bottom of the container the green paste is baked. Unfortunately, the quality of the baked Soderberg anode is lower than the quality of the prebaked one, hence the Soderberg cells are always characterized by a lower current efficiency and a higher potcell voltage, needed also to produce the extra heat necessary for the anode baking.

Presently all the new built smelters adopt the pre-bake technology, because of the higher current efficiency, lower specific energy consumption and lower emission (especially PAHs). However, a good number of Soderberg plants are still in operations, sometimes retrofitted with additional technology aimed at increasing current efficiency and reduce emissions.

Another advantage of prebake technology cells is the absence of *Anode Effect* that is, instead, mainly a problem in Söderberg technology cells: it is a situation where too many gas bubbles form at the bottom of the anode and join together forming a layer and decreases the energy-efficiency and the aluminium production of the cell. It also induces the formation of TetraFluoromethane in significant quantities, increases formation of CO and to a lesser extent also causes the formation of HexaFluoroethane. CF_4 and C_2F_6 are not CFCs, and although not detrimental to the ozone layer, are still potent greenhouse gases.

C
Ackermann Steering Geometry

Ackermann steering geometry is a geometric arrangement of linkages in the steering of a car or other vehicle, designed to solve the problem of wheels on the inside and outside of a turn needing to trace out circles of different radii.
It was invented by the German carriage builder Georg Lankensperger in Munich in 1817, then patented by his agent in England, Rudolph Ackermann in 1818 for horse-drawn carriages. Erasmus Darwin may have a prior claim as the inventor dating from 1758.

The History

Figure C.1: Typical nineteenth-century service carriage.

[10]Before cars, there exist only carriages with large open wheels, typically pulled by horses [Figure C.1]. In particular, short wheelbase carriages had fixed wheels, relying on slippage of the skinny wheels to compensate by skidding when pulled by the horses, but maneuverability at slow speeds, and tighter corners, or longer wheelbases, was enabled by steering the front axle.

This first kind of simple steering is often called *Turntable Steering* [Figure C.2]. There's a single pivot in the center of the front axle. Both the front and rear axles are solid. By rotating the front axle, the geometry is adjusted so that both axles are pointing towards a common point. Each wheel travels around in a circle with a common center (albeit it with different radii), removing the need for side-slip or skidding. In a simple carriage, each wheel is independently free-wheeling, and the traction force is not provided by the wheels, so the differences in the circumferences is compensated by each wheel having a slightly different rotation rate. If the vehicle were travelling over a muddy field, we might see four circular arcs as it turned. This is an elegant solution for maneuverability, but it has quite a few problems: the axle has to swing through large arcs to make turns, and this reduces the usable space inside the wheelbase (or requires the occupied space to be high above both axles). Moreover, it has a single stress point, making the addition of suspension challenging, and the long axle acts like a lever amplifying small variations in road surface.

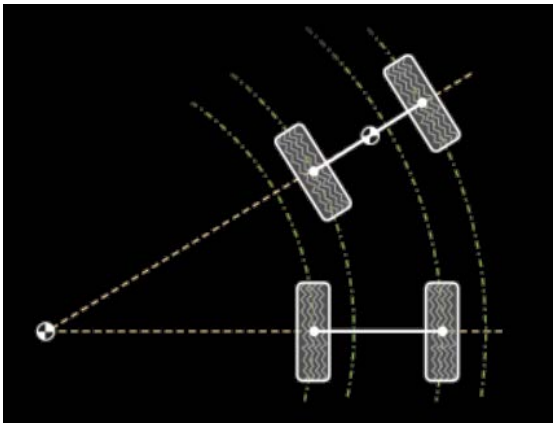


Figure C.2: The turntable steering.

The next solution is the *Simple Steering* [Figure C.3]: each wheel is given its own pivot. This solves some of the problems above, but we get the return of the side-slip issue.

It's certainly better than a totally fixed front axle, which requires a lot of side-slip but, by turning both wheels the same angle, we can see that each of the front wheels has a different center of rotation.

With simple steering, at least one of the front wheels will be experience the problem of side-slip: in an hypothetical vehicle with simple steering manoeuvring over a gravel driveway, on every turn, irrespective of speed, the front wheels would plough through the gravel leaving

unsightly ruts, and require excess traction power to overcome this additional friction compared to simply rolling along.

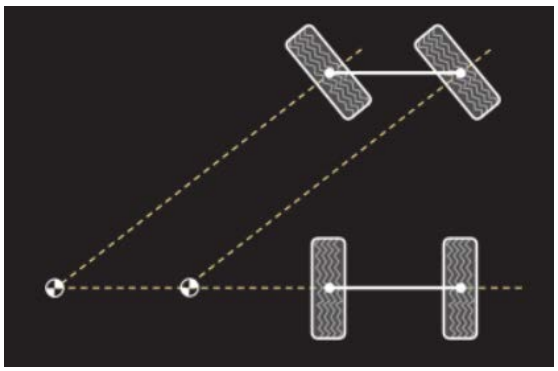


Figure C.3: The simple steering.

The ideal solution would be for each of the front wheels to be independently steerable: in this way, each one could be set to the perfect angle to make a tangent to the circular arc with a common center. With each wheel running on a circular arc of the correct radius there should be no side-slip. To deal with the issue that the different powered wheels running at different rotational speeds, the drivetrain typically has a *differential* placed in the axle.

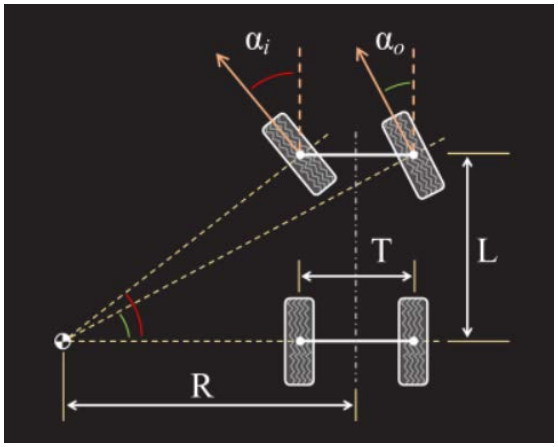


Figure C.4: Steering equipped with differential mechanism.

Taking as a reference Figure C.4, it can see that, when turning, the inside wheel turns with a larger angle than the outside. Defining the following quantities:

- L: the wheelbase of the vehicle (distance between the two axles).

- T: the track (distance between center line of each tyre).
- R: the radius of the turn as experienced by the center line of the vehicle.
- α_i: the angle of inside wheel from straight ahead.
- α_0: the angle of outside wheel from straight ahead.

Assuming constant speed (no weight transfer or external forces), no body roll or suspension effects, and only the front wheels steering, then the ideal angles for the wheels is given by the simple trigonometry:

$$\alpha_i = \tan^{-1}\left(\frac{L}{R - \frac{T}{2}}\right) \qquad \alpha_0 = \tan^{-1}\left(\frac{L}{R + \frac{T}{2}}\right) \tag{C.1}$$

In Figure C.5, using some constants for L and T, the internal wheel angle is plotted on the x-axis while the external wheel angle, that corresponds to a range of possible curve radii, is plotted on the y-axis.

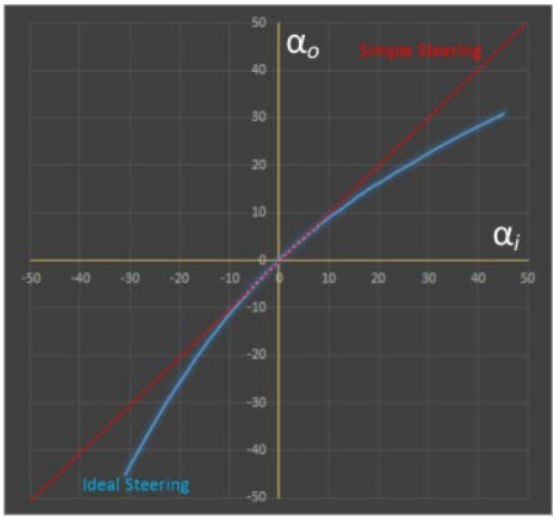


Figure C.5: Parametric angles [deg] plot.

The red line shows the relationship between the two wheel angles for simple steering: here both wheels turn in parallel so the result is a straight line with unity gradient. The blue line shows the relationship between the two wheel angles for ideal steering. For all radii of turns, the absolute value of the internal angle is always higher than that of the outside angle. The tighter the turn, the higher the ratio of the internal to outside angle.
The relationship showed by the blue line is not mathematically simple: the challenge for how

to create a mechanism to turn the wheels in this way is the problem solved by the Ackermann solution.

Ackermann Steerring

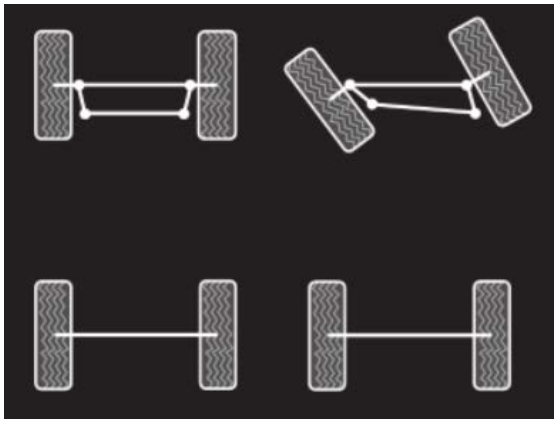


Figure C.6: Ackermann steering mechanism.

In Ackermann steering [Figure C.6], each wheel is given its own pivot (which is typically close to the hub of the wheel). Tie rods create a trapezium shape with two additional pivots. As the shape is a trapezium, as the inside wheel turns, the outside wheel turns at a different rate. By adjusting the geometry (length of the tie rods and angle which they form with the straight ahead), the relative rotations of the wheels can be configured. If the track rod joining the two wheels is the same length as this distance between the two pivots, then the rectangle they make deforms to a parallelogram as the wheels turn, and the configuration is the same as simple steering, and the both wheels turn at the same rate. As the track rod is made shorter (*toe-out*) than the axle, then the inside wheel turns at a higher rate as the trapezium deforms. The equations for the value of the wheel angles based on the rod lengths and initial angles are more complex calculations than first might appear: for this reason, they are not reported here but in Figure C.7 it can see an example of how close the Ackermann steering geometry conforms to ideal steering.
Ackermann steering solves most of the problems of turntable steering: the space required (*fore-and-aft travel*) by each wheel and the moment arm transmitting back imperfections in the road are significantly reduced. For small steering wheel inputs, the difference between the wheel angles is small. As the radius of the required turn decreases, the difference in angles increases.

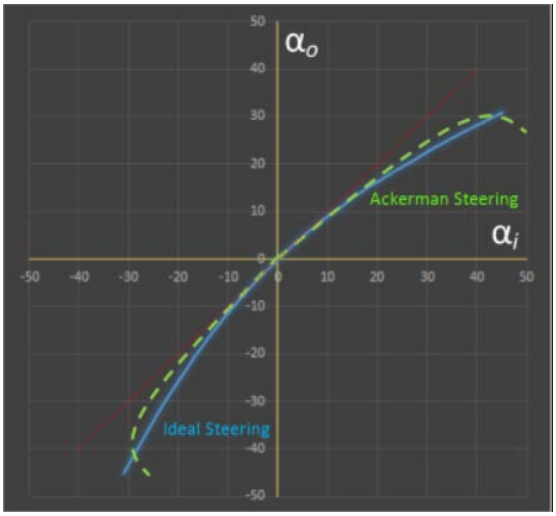


Figure C.7: Parametric angles plot with Ackermann steering trend.

Final Considerations

All of the above calculations are applicable to slow speeds. However, when we need to consider vehicle dynamics, things get more complex. When a vehicle is in motion or in a turn, the tyre deforms and there a difference between the direction of travel and the direction the tyre is pointing. This is called the *slip angle*.
Professional racers adjust the Ackermann configuration of their vehicles to balance needs. There is a limit to the grip that any tyre can provide and, depending on the difference between the slip at the front or rear of the vehicle, understeer or oversteer can be created. Some drivers even configure their vehicles with other similar architectures as *neutral-Ackermann* or even *anti-Ackermann* geometry, to reduce heating and load in the their tyres.

Bibliography

[1] D. Adams. *The Hitchhiker's Guide to the Galaxy*. San Val, 1995. ISBN: 9781417642595. URL: http://books.google.com/books?id=W-xMPgAACAAJ.

[2] Consorzio Imballaggi Alluminio. *Il Processo di Produzione*. 2019. URL: http://www.cial.it/il-processo-di-produzione/.

[3] Consorzio Imballaggi Alluminio. *Il Riciclo dell'Alluminio*. 2019. URL: http://www.cial.it/category/alluminio-e-riciclo/il-riciclo/.

[4] Data Research Analyst and Worldofchemicals.com. *Manufacturing of Alumina through Bayer Process*. 2019. URL: https://www.worldofchemicals.com/591/chemistry-articles/manufacturing-of-alumina-through-bayer-process.html.

[5] R.H. Bemthuis. "Development of a Planning and Control Strategy for AGVs in the Primary Aluminium Industry". Master's Thesis. University of Twente, 2017.

[6] Marco Bergamin. "Indoor Localization Using Visual Information and Passive Landmarks". Master's Thesis. University degli Studi di Padova, 2015.

[7] George J. Binczewski. *The Point of a Monument: A History of the Aluminum Cap of the Washington Monument*. 1995.

[8] Liu Chang-an et al. *Mobile Robot Path Planning Based on an Improved Rapidly-exploring Random Tree in Unknown Environment**. Paper. North China Electric Power University, 2008.

[9] Tim Chin. *Robotic Path Planning: RRT and RRT**. 2019. URL: https://medium.com/@theclassytim/robotic-path-planning-rrt-and-rrt-212319121378.

[10] DataGenetics. *Ackerman Steering*. 2016. URL: http://datagenetics.com/blog/december12016/index.html.

[11] Marta Galvan. "Approaching Path Planning In Dynamic Environment". Master's Thesis. University degli Studi di Padova, 2019.

[12] Jonathan D. Gammell, Siddhartha S. Srinivasa, and Timothy D. Barfoot. *Batch Informed Trees (BIT*): Sampling-based Optimal Planning via the Heuristically Guided Search of Implicit Random Geometric Graphs*. Paper. University of Toronto, 2015.

[13] Matt Garratt, S. G. Anavatti, and Jeifei Wang. "Real-Time Path Planning Algorithm for Autonomous Vehicles in Unknown Environments". In: *International Journal of Mechatronics and Automation* (2017).

[14] Techmo Industry. *La Produzione di Alluminio: Processo Hall-Heroult - Potlines - Cast-House*. 2015.

[15] Sven Koenig and Maxim Likhachev. *D* Lite*. Paper. Georgia Institute of Technology, 2002.

[16] Christian Rasch and Thomas Satzger. *Remarks on the $O(n)$ implementation of the fast marching method*. Paper. Technical University of Munich, 2008.

[17] Techmo Car SPA. *Techmo History*. 2019. URL: https://www.techmo.com.

[18] Anthony Stentz. *The D* Algorithm for Real-Time Planning of Optimal Traverses*. Paper. Carnegie Mellon University, 1994.

[19] Anthony Stentz. *The Focused D* Algorithm for Real-Time Replanning*. Paper. Carnegie Mellon University, 1995.

[20] Totten et al. *Handbook of Aluminum: Alloy production and materials manufacturing*. 2003.

[21] Unknown. *Prebake and Soderberg*. 2009. URL: http://bauxite.world-aluminium.org/refining/process/#top.

[22] Alberto Valero-Gomez et al. *Fast Marching Methods in Path Planning*. Paper. Carlos III University, 2013.

[23] Autodesk Inventor Website. *About Autodesk*. 2019. URL: https://www.autodesk.it/products/inventor/overview.

[24] Blender Website. *About Blender*. 2019. URL: https://www.blender.org/.

[25] Gazebo Website. *Gazebo History*. 2019. URL: http://gazebosim.org/.

[26] Matlab & Simulink Website. *About MathWorks*. 2019. URL: https://it.mathworks.com/products/matlab.html.

[27] Ros Website. *Documentation*. 2019. URL: http://wiki.ros.org/.

[28] Wikipedia. *A* Search Algorithm*. 2019. URL: https://en.wikipedia.org/wiki/A*_search_algorithm#cite_note-10.