POLITECNICO DI TORINO

Master Degree in Mechatronic Engineering

Master Degree Thesis

Adaptation of a path planning algorithm for UGV in precision agriculture

Supervisor:
Prof. Andrea Maria Lingua

Co-Supervisor:
Dr. Vincenzo Di Pietra

Candidate:
Francesco Messina

December 2020

Contents

Acknowledgements

This thesis work represents the conclusion of a difficult path, made of tears, sleepless nights, sacrifices but also of many satisfactions and happy moments. All these moments together create an experience that has forged me as a person and as a professionist. To all the people who have supported and accompanied me along this path I would like to express my most sincere gratitude.

My first thoughts go to my parents who have allowed me to start this path despite the difficulties faced due to the distance that separates us and that has supported me in the darkest moments of this journey.

A special thanks goes to Dr. Vincenzo di Pietra who wisely guided and patiently helped me during this work and thanks to whom it was possible to complete even the last phases of the experimentation despite the difficulties imposed by the sanitary circumstances of this year.

I thank Dr. Gianluca Dara and Dr. Diego Aghi and more generally all members of PIC4SeR for the collaboration and valuable advice from which the work has benefited.

I would also like to thank all the members of DIATI who supported me with their valuable work experience and passion.

Last but not least, I would like to thank all the friends who have been by my side and shared both the most difficult and the happiest moments of this path with me.

Finally, I dedicate this thesis work to my grandparents who would surely be proud of achieved goal.

Abstract

A great interest in self-driven vehicles has developed in the field of robotics and more generally in the industrial sector in the last few decades, and the number of applications where these vehicles are used is growing strongly. A clear example of this phenomenon are the efforts made by several automotive in terms of research and development, to manufacture increasingly competitive, safe and precise Autopilots. This is also due to cutting-edge on-board computers, the development of highly precise and performing sensors, and the advent of the 5G network, which is also aimed to ensure an unprecedented V2X (Vehicle-to-everything) communication, especially thanks to a new architecture called "network slicing".

The robotics sector is now focused on providing solutions that may promote and improve the man-machine collaboration in agriculture as well as in other areas closely related to the civil sphere, therefore not only for military uses and space exploration, as it used to be. This process has seen an acceleration this year for the emergence of Covid-19, which has become a global pandemic. This situation has highlighted, now more than ever, the need to rely on autonomous instruments that, for example, transport medical equipment or essential commodities, without running the risk to get in contact with other people or the need for anyone to leave home.

In this context, the aim of this thesis work is the creation of a UGV (Unmanned Ground Vehicle) that can run independently, safely and efficiently through rough and uneven terrain, as for agricultural fields or unpaved roads, by using the RRT* (Rapidly-exploring Random Tree) path planner. The latter manages to accomplish its task also thanks to the collaboration with a UAV (Unmanned Aerial Vehicle) that, by means of the aerial photogrammetric survey method, can generate high-precision georeferenced orthophotos and DTM (Digital Terrain Model), that are later processed by a GIS (Geographic Information System) software to generate the static binary mask. In detail, the project is designed to cover different project phases. Starting from the design phase, when the criticalities of the problem are evaluated and possible solutions are identified, to the programming phase, in which a code has been developed that allows the planning of the route adapting to the problems that may arise in this type of terrain, up to the simulation and real test

phase in the field.

Introduction

That of self-driven vehicles is one of the fastest growing sector nowadays, whether they are land-based, airborne or otherwise, and this is due to multiple factors. Among the many, the main one can be traced back to an exponential growth of the technological sector, both in terms of investments and resources spent for research activities in the sector and increasingly high level of efficency, safety and precision with which the new instrumentation can fulfil their purpose.

Another factor that has contributed to the growing interest in this sector, especially in recent years, has been the advent of the 4G and later the 5G network, aimed at providing unprecedented V2X communication, thanks to its new architecture, called "network slicing". A last important contribution surely comes from the increasing use of artificial intelligence, i.e. of automatic (or autonomous) learning techniques, such as neural networks, machine learning and deep learning, whose use in many applications represent a big step forward, also for autonomous driving. In short, there is a clear will and need for a closer and more dynamic collaboration between man and machine that developed over time, in a worldwide panorama where most people already live interconnected with machines thanks to the increasingly pervasive use of IoT (Internet of Things) technologies.

Hence, autonomous driving is evidently becoming a concrete and reliable feature for many different sectors such as: military, aerospace, emergency (earthquakes, floods, search for missing persons, etc..), agriculture, mining, construction, manufacturing, surveillance, etc..

In this scenario, the objective of this thesis work is the realization of a UGV that can travel independently on uneven terrain, for instance in an agricultural field or on unpaved roads, through the use of the route planning algorithm known as RRT* and its adaptation to the problems that may arise in terrain whose characteristics may give the vehicle a hard time during navigation.

The UGV is able to achieve its task by collaborating with a UAV that, provides the algorithm with high resolution static maps made in several steps, through the aerial photogrammetric survey technique. Starting from the 3D model of the place where the rover will operate, it is possible to obtain 2 georeferenced maps, a DTM with the terrain altitude information and a binary mask where the obstacles are

identified, through different elaborations with a GIS software.

The algorithm calculates different possible routes that will allow the vehicle to reach the objective, analyzing the surface characteristics of the terrain that the vehicle will face in any possible route. The evaluations use a trial and error approach, in terms of distance, slope, slope variation and roll. This type of approach allows the exclusion of all those paths not meeting the imposed constraints.

The algorithm provide a text file as output. All the waypoints of the route that the vehicle will follow to arrive at its destination are transcribed in this file in Geographic coordinates. The file is then loaded on a GCS (Ground Control Station) software connected to the flight controller mounted on the vehicle. The vehicle was also equipped with several sensors including a high performance GNSS (Global Navigation Satellite System) module. With the interaction and collaboration of these sensors, the vehicle should faithfully follow the route planned by the algorithm.

The work was divided into five chapters, here below is a little overview of each of them.

- **PathPlanning** - A brief presentation of the world of route planning and architecture needed to create an interface between the surrounding environment and the rover, plus a general introduction to the sensors that can be used to create the above interface and the different types of representations possible, is provided. Then, through a short digression, different types of route planning algorithms that can be found in the literature are presented.

- **Materials and methods** - This part deals with the design choices regarding the algorithm, the creation of the maps, the vehicle that was used for the mission, the various software and hardware components selected to achieve the objective.

- **Code and Simulations** - Several flow charts, tables and pseudo-code parts are here used to examine in detail how the code works. Afterwards, different aspects of the code are evaluated through several simulations, even by examining the variation of the code behavior, according to some parameters.

- **Validation** - The results of several tests that were run on the field were compared with the ones from simulations, and an analysis of the issues arising from the vehicle and the instruments was also performed.

- **Conclusions** - The problems that were faced during the whole project are evaluated and discussed in this final part, drawing the appropriate conclusions and giving possible suggestions for future changes and possible steps forward.

Chapter 1

Path Planning

Path planning is defined as the computational problem to find a sequence of valid configurations that moves the object from the source to the destination.

In the world of robotics, a movement task can be completed autonomously within a certain environment in many different ways. A first distinction to make is between guided and unguided vehicles, the former are classically named AGV (Automated Guided Vehicle), an industrial term indicating vehicles handling goods in a factory.

These vehicles can typically take predetermined routes in most cases, given the nature of the environment where they operate. The routes may be indicated by a range of technologies, such as guide tape, wired, laser target navigation, inertial navigation and vision guidance.

On the other hand, unguided vehicles are not limited to a predefined driving path. Therefore, it is essential to collect detailed information about the environment where the vehicle is operating before this is able to make a decision. As a consequence, two situations can occur:

- the environment is completely known;
- the environment is completely unknown.

Path planning algorithms dealing with known environments are called *piano-movers problem* problem algorithms, while the ones for unknown environments are named *path planning with uncertainty* [10]. In general, the system diagram shown in Figure 1.1 applies to unguided vehicles.

A priori information can be gathered, for example, from pre-existing maps of the place where the vehicle is set to operate, or, from the vehicle itself, in case is equipped with the necessary tools and sensors to investigate the surrounding environment. As it is stressed in [11], the sensors provide data that may fall into into three main categories:

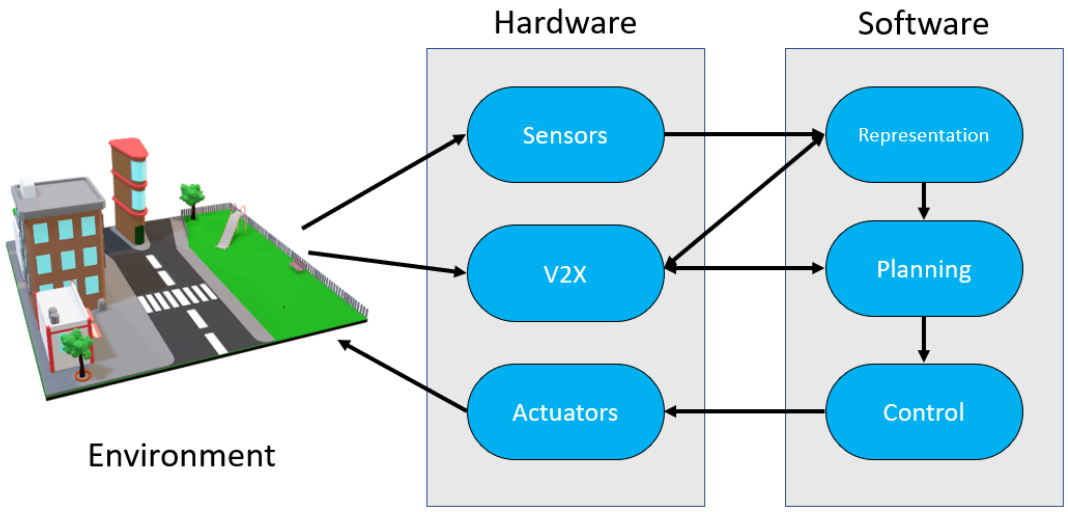


Figure 1.1: General process scheme for vehicle autonomous guide.

1. the world (terrain shape, temperature, color, composition);

2. the system and its relationship to the world (battery charge, location, acceleration);

3. other concepts of interest (collaborator, adversary, goal, reward).

Depending on the type of information obtained from the sensors, different representations of the environment surrounding the vehicle can be created, ranging from the very basic ones, based on single values, to complete 3D geometric models.

The planner strictly depends on the type of information available and, therefore, on what type of representation is going to be built. The very structure and content of the representation will define what decisions the planner can actually make and, ultimately, the set of action plans that the robot can pick [11].

For example, some algorithms can function just in some specific representations, such as bitmaps, while others may adapt to multiple types of representations with some common characteristics.

Lastly, one of the controller's main task is to send the correct inputs to actuators, such as power signal to motors, servos and other devices, and, possibly, fix them in real time to keep the vehicle moving as the algorithm planned.

1.1 Sensors and Representation

In the context of autonomous driving, sensors are an essential part of the process. As already mentioned, sensors can collect utterly necessary and precious information to build a model of the environment to move through.

Many types of sensors have been designed and tested for this purpose, and most of the times they are combined together to conceive a useful representation of the surrounding environment. Sensors can be grouped into two main categories:

1. active sensors, whose instruments rely on their own source of radiation to measure the propagating signal, which comes back to the source by reflection, to detect the position, shape, distance of the objects surrounding the vehicle and other information on the environment;

2. passive sensors, whose instruments make use of information already present in the environment, i.e. they capture radiations, such as light, heat, vibrations or other phenomena emitted by the environment itself.

The first family includes sensors such as radars, lidars, sonars and ultrasonic sensors. In autonomous driving applications, these sensors are generally integrated with passive ones, infrared or hyperspectral cameras, RGB cameras, GPS (Global Positioning System) and inertial sensors to obtain more complete representations of the environment.

Plus, it is worth mentioning all those sensors needing to establish a physical contact with environment area that they are sensing. These devices are thermometers, tactile sensors, strain gauges, and bumper sensors.

When speaking of a detailed representation of the environment, this must not mean complex, since a complex representation does not necessarily lead to a better result. This also depends on the type of environment where the vehicle is operating. Obviously, the more flexible and general is an application, the more the representation can adapt to new events.

Nowadays the most common types of representations are the following:

- metric topological maps, shown in Figure 1.2, that combine the 2D metric framework and the topological framework. In the former, objects are identified by precise coordinates but is excessively noise sensitive, while in the latter only locations and the relationships between them are considered, like a graph;

- full metric maps, shaping the entire operating environment into a fixed coordinate system.

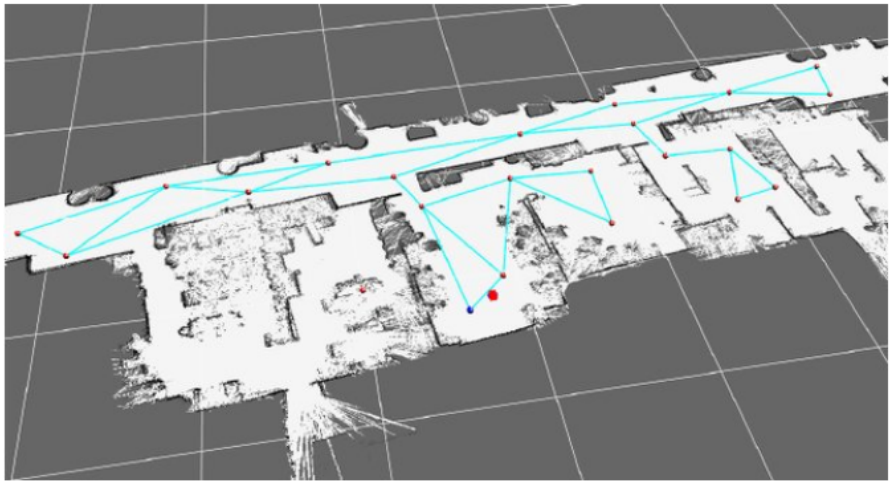


Figure 1.2: Example of Metric Topological Map [4].

As underlined by [11], the trend to use this type of representations in recent years is due to various factors, the main ones being:

- more accurate localization techniques, thanks also to the development of systems such as GNSS improving GPS integrity, accuracy and availability;
- increasingly powerful computers capable of storing and processing a higher and higher amount of information in less time. This means that these computers can interact with increasingly large and complete maps.The example in Figure 1.3 shows the trend for various computer parameters over the last 48 years;
- more accurate range sensors such as lidar and stereo vision;
- recent developments in planning algorithms making use of these types of maps.

Maps can be either created in real time thanks to the sensors on the vehicle or be entirely provided by the algorithm, if this is obtained in advance. A map can be static, unable to change, or dynamic, constantly updated as the robot moves along.

As with most engineering solutions, hybrid versions integrating the above-mentioned examples can be used for optimal results. For example, two representations, a global one and local one, can be used in parallel. A local representation, containing highly detailed information, depicts what surrounds the robot, whereas, the global representation contains all the most relevant information that the robot perceived in the local frame.

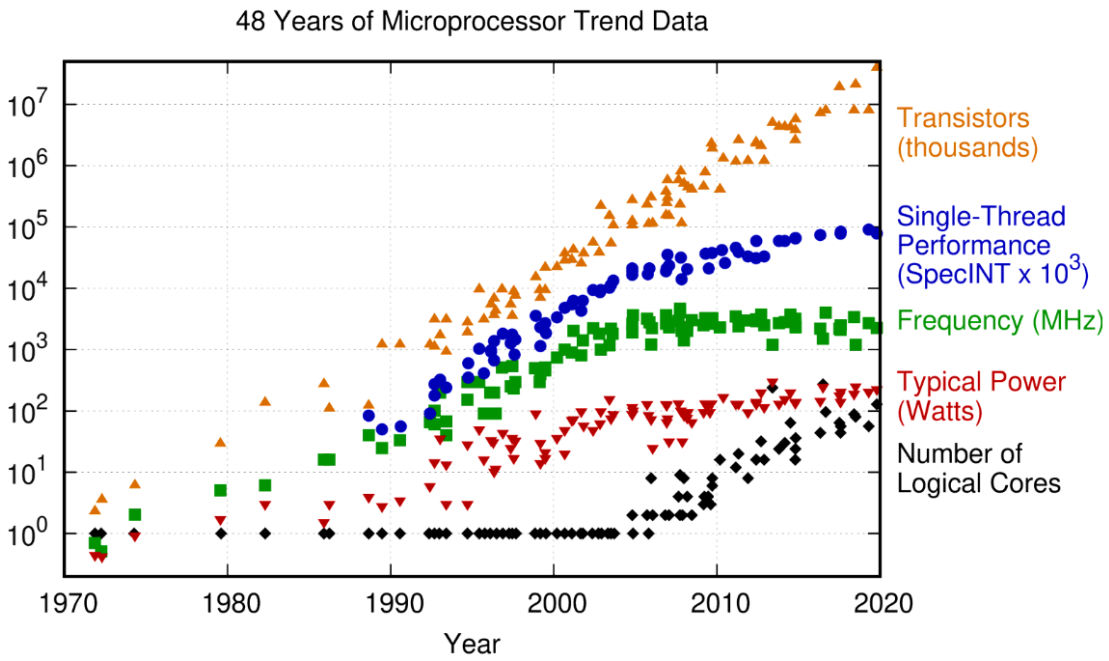


Figure 1.3: 48 Years of Microprocessor Trend Data (`https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/`).

This type of planning is widely used to cope with the problems that each single representation brings with it. Locally, the robot risks to show a myopic behaviour, while it is likely to overload its memory and extend the algorithm computation times globally, specially with very large and information-rich maps.

1.2 Planning Algorithms

After having made an overview of the sensors and the types of representations used in this field, we want to focus on the types of algorithms that exist in the literature. Before that, however, for a better understanding it is convenient to introduce a canonical definition of the problem that these algorithms are called to solve and that highlights its fundamental issues.

Consider a robot, which can be either a mobile one or a standard manipulator, now let $C = (0,1)^d$ be the configuration space, where $d \in N$ represent the dimension of the space in which we are working and $d \geq 2$. There can be static obstacles in this space, the region they occupy within the configuration space will be indicated with C_{obst}. The remaining portion of the configuration space that is free from obstacles, will be denoted with $C_{free} = C/C_{obst}$.

It should be noted that the above term *configuration space*, is somehow different from that of a geometric map. In general, the C-space represents the valid configuration states within the geometric map. Some planning algorithms, in fact, turn the latter into the former, by applying a specific method.

The C-space may be sufficient to find a valid path, as described by LaValle in [9], since it provides a powerful abstraction that converts the complicated models and coordinates transformations into the general problem of computing a path that traverses a manifold.

The term *manifold* indicates a specific type of topological space that behaves at every point like our intuitive notion of a surface, and the size of this topological space is generally related to the degrees of freedom of the robot. It is assumed both that the geometry of the robot, the obstacles and the pose of obstacles in C are known, and that the robot is free from any kinematic constraints.

The planning problem can be summarized as follows: given an initial pose of the robot x_{init} and a final goal region X_{goal} in the obstacle free space C_{free}, find, if exist, a path that drives the robot between the two poses, avoiding collisions at the same time, as it is shown in Figure 1.4. An algorithm addressing this problem is said to be complete if it terminates in a finite time, returning a valid solution if one exists, and a failure otherwise.

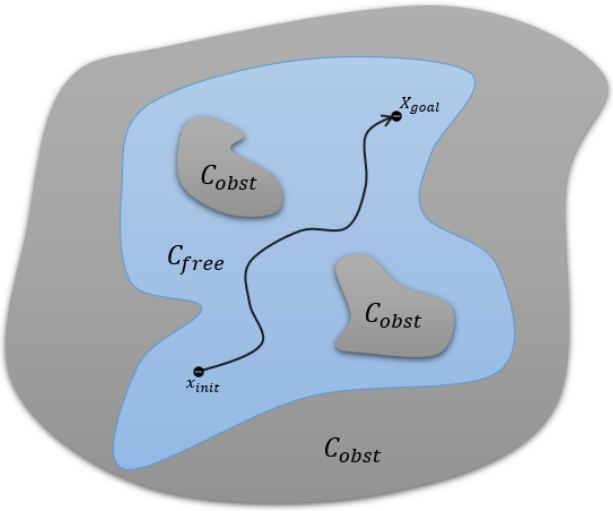


Figure 1.4: Representation of a simple C-space where the task is to find a path from x_{init} to X_{goal} through the C_{free} space.

Motion planning under differential constraints can be considered as a variant of classical two-point boundary value problems (BVPs). In mathematical analysis, a function of bounded variation is a real-valued function whose total variation is bounded (finite).

Let $\sigma : [0,1] \rightarrow R^d$; the total variation of σ is defined as:

$$V(\sigma) = \sup \sum_{i=0}^{n-1} \mid \sigma(x_{i+1}) - \sigma(x_i) \mid \tag{1.1}$$

The total variation of a path is essentially its length, i.e., the Euclidean distance traversed by the path in R^d. A function σ with $V(\sigma) < \infty$ is said to have bounded variation. In this way a collision free path can be defined as a continuous function σ of a bounded variation where $\sigma(x) \in C_{free}$ for all $x \in [0,1]$ [7].

Considering this definition, the literature shows several algorithms approaching the problem in different ways, both in terms of complexity and success. Some of these will be cited and briefly described below to better understand the choices made during the design phase of this work.

- Grid-based or Graph-based algorithms
- Reward-based algorithms
- Artificial Potential Field algorithms
- Sample-based algorithms

1.2.1 Grid-based or Graph-based

A grid is superimposed with a certain resolution (in terms of the size of each cell) on the configuration space in these types of algorithms. The grid can be of several shapes, each shape lead to a different number of degrees of movement, for example: square (4), hexagonal (6) or octagonal (8) and there are many other possible solutions as shown in Figure 1.5.

Each point of the grid, that is classically located at the centre of the cell, represents a specific configuration that the robot may assume. There are two basic constraints governing the grid:

1. two points of the grid can be connected if contained in two adjacent cells;
2. the line connecting the two points must reside entirely within the free space C_{free}.

The last constraint is verified thanks to the use of geometric algorithms detecting the collision between the robot and a possible obstacle. As highlighted in the first constraint, the shape of the grid should be carefully designed depending on the specific type of search algorithm that is being implemented.

In general, a certain cost value is associated with each single point of the grid, on the base of what is to be minimized or maximized, such as the distance between

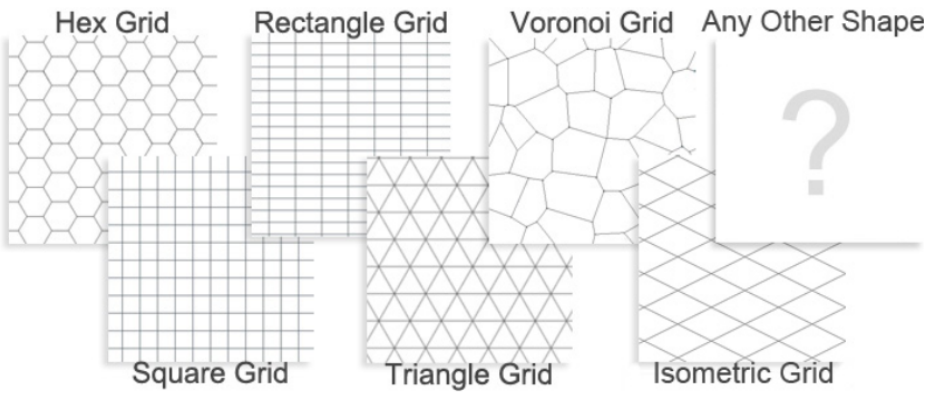


Figure 1.5: Possible types of grid shapes (`http://www.davetech.co.uk/gamemakerpathfindingnode`).

starting and ending points or the distance between a point and an obstacle. Hence, an attempt is made to find a sequence of points or poses, fulfilling the imposed constraints, for the robot to reach its destination.

According to the above, the grid can therefore be thought of as a graph $G = (V, E)$ composed of a set of vertices V, i.e. the points of the grid, and arcs E, i.e. the lines connecting the nodes that form the path. Algorithms that make this kind of interpretation are called *graph-based*. The arcs can be undirected i.e. with a direction but without a verse, in this case they are called edges, or oriented i.e. with defined direction and verse as showed in 1.6.

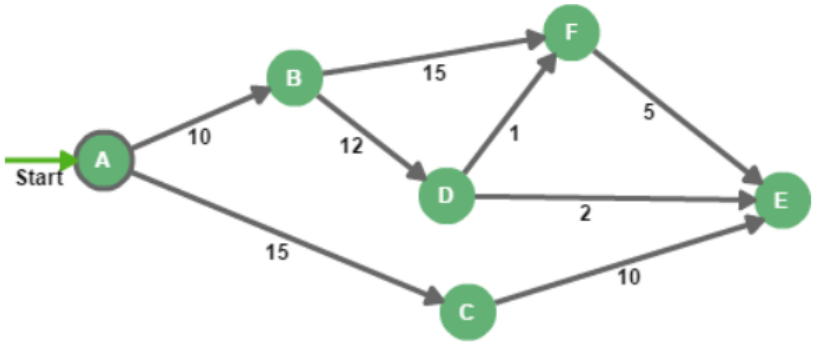


Figure 1.6: Example of directed graph (`https://www.baeldung.com/java-dijkstra`).

The generic problem is addressed in the following terms, let x_i represent an arbitrary node in the graph and denote x_{goal} and x_{start} as a set of goal nodes and start nodes respectively. Let $e_{i,j}$ represent an edge that goes from node x_i to node

x_j that, as already said before, can be oriented so $e_{i,j} \neq e_{j,i}$ or undirected where $e_{i,j} = e_{j,i}$. Node x_j is considered a neighbor of node x_i if $e_{i,j} \in e_i$ where e_i is the set of all edges that leave node x_i [11].

As the search begins, the algorithm starts at x_{start} and attempts to find a path to x_{goal} with a node-to-node exploration via edges, this method is known as forward search, or vice-versa using the backward search method. Each node can be seen in different states during the search, a node is said to be unexpanded if the search algorithm has not reached it yet, whereas it is said to be alive or open if it was reached by at least one of its neighbor, it is said to be expanded, dead or closed, if it was reached as well as all of its neighbors.

In general, as the search progress, two lists are maintained: an open list where all open nodes are recorder and a closed list containing all dead nodes. At each step the neighbors of the current node x_i that has not been reached x_j are added to the open list and x_i is removed from the open list and added to the closed one. The search is either successful when the goal node is added to the open list at the end of it, or unsuccessful if the goal node has not been reached yet and the open list is empty [11].

The various graph-search algorithms like Breadth-first (BFS), Depth-first (DFS), Best-first, A*, D* and many others differ in the way they choose a node x_i from the open list.

1.2.2 Reward-based

It is assumed, for these algorithms, that the robot may choose between different actions to perform, any time and in any given pose. However, the result of a given choice is not known a priori and a sort of trial and error approach is used, where if the goal is reached a positive evaluation is assigned, while if the performed action leads to a dead end or collides against an obstacle, a negative rating is assigned.

The goal of this type of algorithm is to maximize positive evaluations. One of the most used mathematical decision models in this very approach is the Markov Decision Process, known as MDP, which can be defined as a discrete time stochastic control process, conceptualized as a nondeterministic graph representation of the world.

As can be seen in Figure 1.7 at each time step the agent is in state s, it receive from the Reward function $R(a, s, t)$ information about the reward obtained through an action performed at time t in state s and information about the next state s at time $t + 1$ through the transition function $S(a, s, t)$. Then following the chosen Policy, it performs a decision or an action and the loop goes by until the goal state is reached. The "Policy" can be interpreted as a function P that specifies the action $P(s)$ that the decision maker will choose when is in state s. The goal is to select a policy which could maximize some cumulative function of the random

rewards, typically the expected discounted sum over a potentially infinite horizon [3].

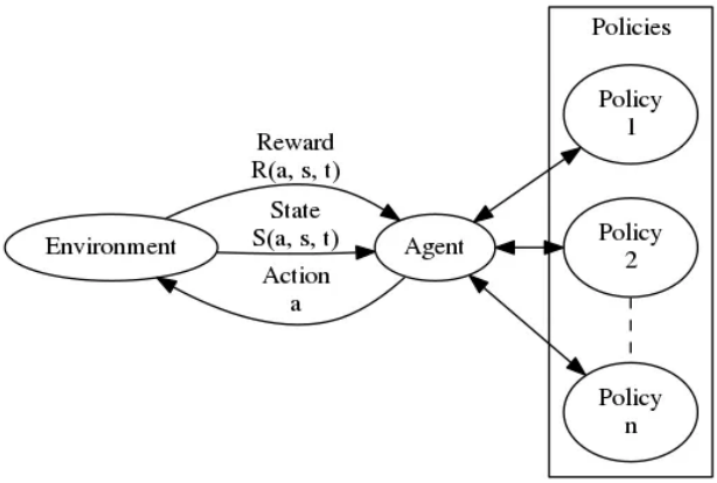


Figure 1.7: Reward-based loop.

The main advantage of using MDP is that they generate optimal path, but they limit the robot to choose from a finite set of actions, therefore the resulting path will be not smooth. In order to solve this last problem usually a fuzzy interference system is usually applied to the MDP.

1.2.3 Artificial Potential Field based

Artificial Potential Field (APF) algorithms make use of the potential field concept to find a path along the obstacles. It simply matches every point in a given environment with a potential artificial field, using the potential field function defined over free space as the sum of an attractive potential $U_{att}(q)$ pulling the robot toward the goal configuration and a repulsive potential $U_{rep}(q)$ pushing the robot away from obstacles [12].

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{1.2}$$

The attractive potential is a function of the distance between the robot current position and the target and can be modelled as follow:

$$U_{att}(q) = \frac{1}{2} k_a (q - q_{goal})^2 \tag{1.3}$$

where k_a is a positive coefficient of gravity for APF. Therefore, knowing that the force due to this potential field $F_{att}(q)$ is

$$F_{att}(q) = -\nabla U_{att}(q) = k_a(q_{goal} - q) \tag{1.4}$$

That corresponds to the negative value of the gradient of the potential field with respect to the configuration q.

The repulsive force requires a distance function between the C space obstacle B and the configuration q identified as $d(q, B)$ instead. In this case the model of the repulsive potential is:

$$U_{rep}(q) = \frac{k_r}{2d^2(q, B)} \tag{1.5}$$

where k_r represent the repulsion gain coefficient. The force is the negative gradient again, due to this potential field, and it points in the direction where the distance between q and the C space obstacle grows the fastest.

$$F_{rep}(q) = -\nabla U_{rep}(q) = \frac{k_r}{d^3(q, B)} \frac{\partial d}{\partial q} \tag{1.6}$$

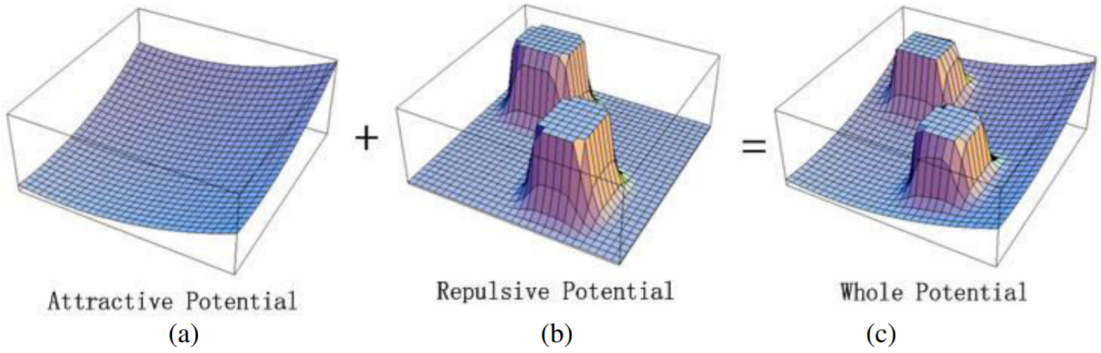


Figure 1.8: (a) The attractive potential without obstacle, (b) The repulsive potential set the highest value to the obstacle, (c) Whole potential shows the combination of the two forces to get the final potential field result.

The resulting force between the attractive and the repulsive field is

$$F_T(q) = -\nabla U_T(q) = F_{att}(q) + \sum_{i=1}^{N} F_{rep,i}(q) \tag{1.7}$$

The goal node is matched with the lowest potential, while the highest one is matched with the starting node. As described in [14] there are three possible approaches to plan a collision-free path with this method:

1. let $\tau = F_T(q)$ where τ is the vector of generalized forces hence let $F_T(q)$ represent the vector of generalized forces that guide the motion of the robot according to its dynamic model;

2. treat the robot as if it were a lumped mass point moving under the influence of $F_T(q)$, as in $\ddot{q} = F_T(q)$;

3. interpret the force field $F_T(q)$ as the desired speed for the robot, imposing $\dot{q} = F_T(q)$.

The main drawback of this approach is the fact that the robot can be easily trapped into local minima, as shown in Figure 1.9, for example when an obstacle is located between the goal and the rover and it is very close to the goal so that the sum of the repulsive force of the obstacle and the attractive force of the goal node will cancel each other.

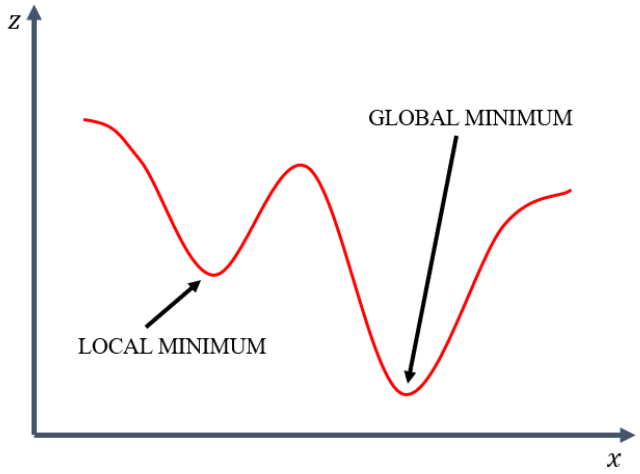


Figure 1.9: Example of local minimum.

1.2.4 Sample-based

Sampling-based methods randomly sample a set of states from the free configuration space C_{free} and connect them in a roadmap fashion. Each sampled state represents a graph node and edges indicate what are the possible transitions between states.

The sampling process continue until the start and goal node are contained in the graph and connected. The algorithm can fail in finding a path for two reason:

1. there is not a traversable path connecting the start node and the goal one in C_{free};

2. the maximum number of pre-set samples is not enough to reach the target node.

The connectivity between nodes is checked through methods not requiring explicit construction of obstacles in the configuration space C, which is proved to lead to a considerable saving of time computation [6].

These types of algorithms are perfectly suitable for high-dimension problems because the running time does not exponentially depend on the dimension of the configuration space. For sample-based algorithms, unlike others seen previously, a weaker definition of *complete* can be accepted in the sense that they may not signal when the path does not exist.

Therefore as LaValle points out in [9] the notion of denseness gains importance in these cases, which means that the samples come arbitrarily close to any configuration as the number of iterations tends to infinity. A deterministic approach that samples densely will be called *resolution complete*. This means that if there is a solution, the algorithm will find it in finite time; however, if a solution is missing, the algorithm may run forever.

Sample-based algorithms that are based on random sampling are defined as *probabilistic complete* meaning that the probability of finding a solution tends to one as the number of samples increases. It is also proven that the rate of decay of the probability of failure is exponential under the assumption that the environment has good "visibility" properties [1].

The most famous algorithms of this family are the Probabilistic RoadMap (PRM) and the Rapidly-exploring Random Tree (RRT) although the base idea of sampling random points in the configuration space and connect them to create a path is present in both algorithms, they differ in the way they construct the graph that connects nodes [7].

The Probabilistic RoadMap algorithm is primarily aimed at multi-query applications. The Algorithm 1 shows the simplified version of the algorithm, where an initial point x_{init} that is the given starting point and n sampled points from the C_{free} space are taken and added to the vertex set V. Then a virtual ball of radius r, starting from the point v, is created for all the vertices in the set V and all the points in this radius are added to the vertex set U . For each point u of the set that is connected to v that results in a free collision connection, a new edge is added to the set of edges E.

A practical implementation of the PRM algorithm often considers different methods to choose the set U of vertices where connections are attempted, an example of it can be the well-known k-Nearest method [7].

On the other hand, the Rapidly-exploring Random Tree algorithm can be seen as a single-query counterpart of the PRM. The RRT algorithm incrementally builds tree branches from randomly sampled points within the state space and it is biased

Algorithm 1: sPRM algorithm.

V $\leftarrow \{x_{init}\} \cup \{SampleFree_i\}_{i=1,\ldots,n}$; E $\leftarrow \emptyset$;
foreach $v \in V$ **do**
  U $\leftarrow$ Near($G = (V, E)$,v,r)$\setminus \{v\}$;
  foreach $u \in U$ **do**
    if *CollisionFree(v,u)* **then**
      E $\leftarrow$ E $\cup \{(v, u), (u, v)\}$;
    end
  end
end
return $G = (V, E)$;

to grow towards large unsearched areas of the graph 1.10.

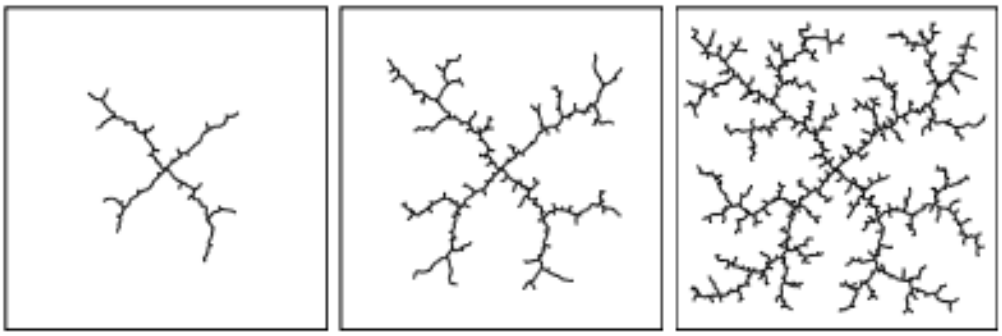


Figure 1.10: Example of RRT branches expansions [8]

This is evident from the Voronoi diagram of the RRT vertices. Since it is proven that the larger regions of Voronoi can be found in the "frontier" of the expanding tree the fact that vertex choice is based on nearest neighbors principle leads to increase the probability of the selecting nodes with a large Voronoi region [8].

The use of the term state space, in this algorithm, indicates a greater generality than what is normally considered in path planning. For a standard problem, $X = C$ which is the configuration space of a rigid body in a 2D or 3D environment, but $X = T(C)$, for a kinodynamic planning problem, which is the tangent bundle of the configuration space containing information on both the configuration and the velocity of the rigid body [8].

An outline of the algorithm is given in Algorithm 2 where the initial state x_{init} is added to the vertex set V and the edges set is empty. For n iterations a state is randomly picked from the free state space called x_{rand} and from that point using the nearest neighbor search $x_{nearest}$ is selected.

The final state of the trajectory that extends the nearest vertex ($x_{nearest}$) towards the sample (x_{rand}) through the Steer procedure is called x_{new} and in the case of a collision-free trajectory, this is added to the edges set E and the state x_{new} is added to the vertex set V.

The search can end before the n-th iteration is reached, if a node making part of the goal region is added to the vertex set V.

RRT includes some properties of the PRM but it has the unique advantage that it can be directly applied to nonholonomic and kinodynamic planning problems. This advantage stems from the fact that RRT does not require any connections to be made between pairs of states, while probabilistic roadmaps typically require tens of thousands of connections [8].

Algorithm 2: RRT algorithm.

V $\leftarrow \{x_{init}\}$; E $\leftarrow \emptyset$;
for *i= 1,...,n* **do**
  $x_{rand} \leftarrow$ SampleFree;
  $x_{nearest} \leftarrow$ Nearest (G=(V,E), x_{rand});
  $x_{new} \leftarrow$ Steer ($x_{nearest}, x_{new}$);
  if *ObstacleFree*($x_{nearest}, x_{new}$) **then**
    V $\leftarrow$ V $\cup \{x_{new}\}$, E $\leftarrow \cup \{(x_{nearest}, x_{new})\}$;
  end
end

Chapter 2

Materials and Methods

In the first chapter we saw what path planning is in general, what aspects are crucial (like the type of sensors and representation used) what types of algorithms are fit for the task and how they work. Whereas, in this chapter, we are going to deal with a series of considerations and choices based on what has been said before.

Therefore, in the following sections, , the choices made during the design phase also with regards to maps, vehicle, navigation software, algorithm development and hardware components will be illustrated, as well as the explanation of the selected algorithm, to provide a complete overview of the project.

The algorithm was chosen as a consequence of some considerations on the problems that may arise also due to the environment where this algorithm is planned to operate and the circumstances that may occur.

Starting with the previously mentioned graph-based algorithms, it can be stated that their main feature is to comply with the assumption that the whole representation is explicitly known, as if the structure of the graph was known a priori, according to these algorithms.

This method can be very effective for simple and relatively small environments, however, this type of approach can become a real challenge on a computational level or even infeasible, when we deal with very large maps and complex environments.

In these cases an implicit representation, using a black box function to determine whether a point in the configuration space is valid or not, may be the right choice to address the computational complexity of large and complex environments [11].

As far as the Artificial Potential Field is concerned, its main criticality is to easly get stuck in local minima, as it was previously stated. Plus this type of algorithm is difficult to implement for a real application.

APF also shows poor performance when the vehicle must go through narrow

passages and have to cope with symmetric obstacles, that is exactly our situation here, since the case study is a vineyard.

Shifting the focus on reward-based algorithms in general these are hard to implement, as already said, and opting for a simple MDP model may entail a narrow choice of actions for the rover, not to forget that this type of algorithms is most successful if supported by a machine learning algorithm. Therefore, reward-based and APF types were discarded.

The sample-based algorithms were finally selected, because of some interesting properties of theirs that may prove to be fruitful if used in this application.

2.1 Algorithm identification

Before shedding some light on the advantages of a sample-based algorithm and its detailed functioning, a reference to the beginning of this chapter will help prove why these algorithms are being used.

The A* algorithm, one of the most commonly-used graph-based algorithms, works with 2 lists, an open one and a closed one and inserting an element in them or selecting a specific element among hundreds is nothing difficult, but in an environment with hundreds of thousands of items, even keeping a list is not such a trivial task. This situation may occur in case of a large map or if the grid resolution is high. Actually in this very case study both difficulties are present.

Moreover, something else is worth stating regarding the pre-set goal of controlling the slope and its variation along the path.

All the algorithms generally seek a node-by-node path, referring to the position of the rover (classically the centre line of the rear axle of the vehicle) as if the vehicle was entirely included in that node. This may work when the cells are more or less the size of the rover and the cost value of each cell is associated to the average height of the ground in that portion of space, but if the grid resolution is higher than the size of the vehicle, the path that the algorithm finds, trying to minimize the total cost, could generate an unexpected result in the real application.

Basically, the path that the simulation found may be optimal for the center of the rear axle, but the real points of contact with the ground, the centers of the wheels, are running along paths with differently-inclined slopes that can lead to strong instability during navigation as showed in Figure 2.1.

The use of a grid with cells of dimensions that may match those of the vehicle was excluded, given the nature of the project which aims to develop a precision positioning method.

Suppose that the surface area occupied by the chosen rover is about 0.25 square meters, with the resolution used for the square cell grid of 0.1 meter side, results that the surface area of each cell is 0.01 square meters and therefore within the

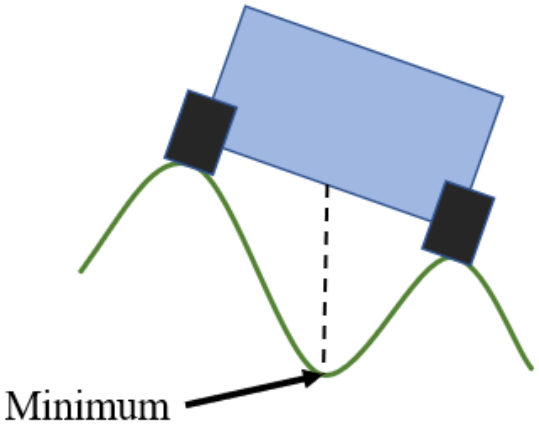


Figure 2.1: Front view of a possible unstable situation

surface area of the vehicle can ideally house 25 cells.

As it was anticipated at the beginning of the chapter the sample-based algorithm was picked for path planning purpose, due to some interesting properties that can be useful to achieve the set objective.

One of the first advantages that these algorithms display is that, unlike graph-based, they use an implicit representation and, therefore, the structure of the graph is not known a priori as they randomly sample a set of states.

This feature determines their speed in building a graph as well as the fact that trying to simulate the same task twice in a row, i.e. the same starting and ending point, two different paths are most likely to be found.

Since the computational time that the algorithm needs to find a feasible path is very short, as underlined in [7], additional heuristics are used to improve the solution, in many field implementations of sampling-based algorithms. This feature opens the way to subsequent checks, for example, on the slope and its variation along the entire path as in our case.

The other fundamental feature is that the nodes are selected starting from random numbers generated by a certain seed. When changing the seed, as a consequence, also the generated path will be different for each single test.

This leads to two obvious but fundamental considerations. The first is that using the same seed with the same start and end coordinates the same path will always be generated. It must also be considered that, given the speed at which the algorithm can build a graph, it is possible to have a different path by changing the seed for each cycle, in a sort of trial-and-error version of the algorithm.

In detail, the algorithm used for this thesis work is an optimal version of the RRT called *RRT** proposed by Sertac Karaman and Emilio Frazzoli in [5] and [6],

that ensures asymptotic optimality, with respect to a given cost function, while maintaining the same properties and the same complexity of the basic RRT.

In particular in [5] the authors demonstrate that the RRT algorithm converges to a non-optimal solution with probability 1 and an RRT* version is proposed that results optimal for systems without differential constraints.

In the second article [6] Karaman and Frazzoli extend the properties of RRT* to achieve optimal results even with systems with differential constraints by identifying a set of conditions sufficient to ensure asymptotic optimality. The following example consists of the problem presented by them.

Given the domain X, obstacle region X_{obs}, goal region X_{goal} and a smooth function f that describes the system dynamics, find a control $u \in U$ with domain $[0, T]$ for some $T \in \mathbb{R} > 0$ such that the unique corresponding trajectory $x \in \chi$, with $\dot{x}(t) = f(x(t), u(t))$ for all $t \in [0, T]$:

- avoids the obstacles, i.e., $x(t) \in X_{free}$ for all $t \in [0, T]$;
- reaches the goal region, i.e., $x(T) \in X_{goal}$;
- and minimizes the cost functional $J(x) = \int_0^T g(x(t))\, dt$.

In terms of algorithm, the extended version partly takes up what was reported in Chapter 1 in subsection 1.2.4 for the RRT algorithm, adding a part of the procedure that we report in Algorithm 4 in detail.

As before, only the initial vertex x_{init} is inserted into the set of vertices V at first, while the set of edges E is left empty. In iterative mode, there is a random sample from the X_{free} space, called z_{rand} and then the tree branch is extended by adding this sample to the vertex set.

In the extended version through the *Nearest* function the vertex closest to the sample is searched, which will be called $z_{nearest}$. Then a trajectory is searched through the function *Steer*, which will be called x_{new}, between the near vertex and the sample, to define the final state of the newly found trajectory as z_{new}.

If the trajectory falls entirely in an area without any obstacles, then z_{new} is added to the set of vertices V. Then the cost of this vertex, calculated through the function *Cost*(z_{new}), is defined as the minimum cost c_{min} and the vertex $z_{nearest}$ is set as a minimum vertex (z_{min}).

The set of vertices that are close to z_{new} called Z_{nearby} are found by means of the function *NearVertices* and for each of these vertices defined as z_{near} a trajectory with z_{new} is to be found where the sum of the cost of the vertex *Cost*(z_{near}) plus the cost of the trajectory *J*(x_{near}) is lower than the one previously found.

Anything provided that the trajectory belongs to an unobstructed area. Once the new vertex z_{new} is inserted into the tree together with the edge connecting it to its parent, the extend operation also attempts to connect z_{new} to vertices that are already in the tree.

Algorithm 3: The RRT* algorithm.

V $\leftarrow \{z_{init}\}$; E $\leftarrow \emptyset$; i $\leftarrow$ 0;
while $i < N$ **do**
  $G \leftarrow (V, E)$;
  $z_{rand} \leftarrow$ Sample(i); i $\leftarrow$ i+1;
  $(V, E) \leftarrow$ Extend(G, z_{rand});
end

Algorithm 4: Extended procedure of the RRT* algorithm.

V' $\leftarrow$ V; E' $\leftarrow$ E;
$z_{nearest} \leftarrow$ Nearest(G, z);
$(x_{new}, u_{new}, T_{new}) \leftarrow$ Steer$(z_{nearest}, z)$;
$z_{new} \leftarrow x_{new}(T_{new})$;
if *ObstacleFree*(x_{new}) **then**
  V' := V' $\cup \{z_{new}\}$;
  $z_{min} \leftarrow z_{nearest}$; $c_{min} \leftarrow$ Cost (z_{new});
  $Z_{nearby} \leftarrow$ NearVertices $(G, z_{new}, |V|)$;
  for *all* $z_{near} \in Z_{nearby}$ **do**
    $(x_{near}, u_{near}, T_{near}) \leftarrow$ Steer(z_{near}, z_{new});
    if *ObstacleFree*(x_{near}) *and* $x_{near}(T_{near}) = z_{new}$ **then**
      if *Cost*(z_{near}) + *J*$(x_{near}) < c_{min}$ **then**
        $c_{min} \leftarrow$ Cost$(z_{near}) + J(x_{near})$;
        $z_{min} \leftarrow z_{near}$;
      end
    end
  end
  E' $\leftarrow$ E' $\cup \{(z_{min}, z_{new})\}$;
  for *all* $z_{near} \in Z_{nearby} \setminus \{z_{min}\}$ **do**
    $(x_{near}, u_{near}, T_{near}) \leftarrow$ Steer (z_{new}, z_{near});
    if $x_{near}(T_{near}) = z_{near}$ *and ObstacleFree*(x_{near}) *and Cost*$(z_{near}) >$
     Cost(z_{new}) + *J*(x_{near}) **then**
      $z_{parent} \leftarrow$ Parent(z_{near});
      E' $\leftarrow$ E' $\setminus \{(z_{parent}, z_{near})\}$;
      E' $\leftarrow$ E' $\cup \{(z_{new}, z_{near})\}$;
    end
  end
end
return $G' = (V', E')$;

2.2 Maps

As previously mentioned, representation is an important, if not vital, element for the correct functioning of the algorithm.

Given the initial desire to create an autonomous UGV that collaborates with a UAV (Unmanned aerial vehicle) two maps were then used, a DTM (Digital Terrain Model) and a binary map that were extracted starting from a previously realized point cloud, shown in Figure 2.2, resulting from an aerial survey performed by a drone.

The maps here used refer to a vineyard in Baldicchieri d'Asti in the province of Asti Piemonte.

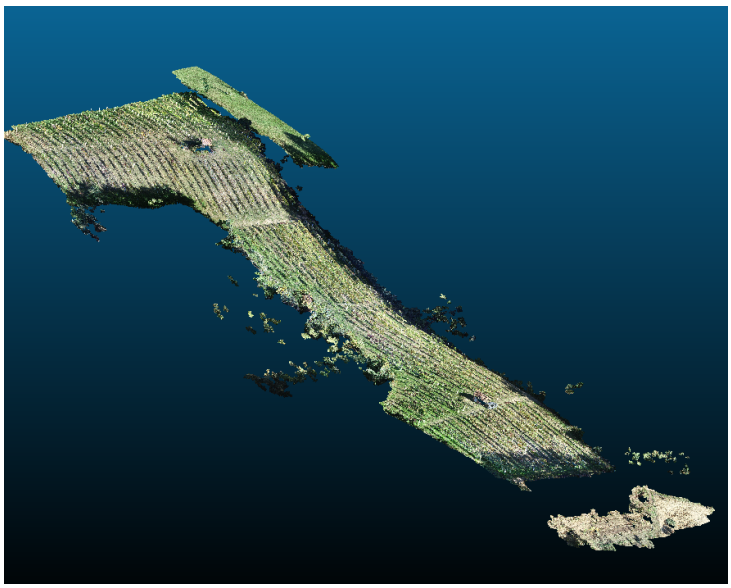


Figure 2.2: Point Cloud of Baldicchieri Vineyard.

The rather recent technique consists in taking a series of high-resolution images in sequence using a drone that flies over the area to be analyzed. Clearly visible markers are positioned uniformly on the ground to accurately scale the survey and to carry out checks on the accuracy returned in post-processing. The exact position of each marker was previously measured with a GNSS receiver.

All the resulting images obtained are subsequently post-processed according to the photogrammetric procedure consisting of various algorithms including SFM (Structure From Motion) and image matching. In this way a 3D model is obtained, from which to derive the orthophoto and the DTM.

Orthophotos, DSM and DTM can be generated through different functions, as described in [15], from the three-dimensional model. The binary map has been realized in two steps.

Figure 2.3: Final orthophoto generated through the series of high-resolution images of the vineyard.

In the first one the difference between the altitude values of the digital model of the surface and the terrain is computed using a function of the ArcMap software, called *Raster mathminus*.

The second step involves the use of a tool called *Raster Reclass* classifying certain elements in two categories according to the values of the selected thresholds.

The two categories either refer to positions with or without obstacles. The threshold values that were selected for the two categories refer to all the elements whose height difference is below 0.5 cm and all those whose height difference falls above the 0.5 cm threshold respectively.

Both maps, in Figure 2.4, are in GeoTiff format and georeferenced with a 0.01-0.02 meters accuracy along the plane and 0.02-0.03 meters in altitude.

In the final phase of the project, given the period of restrictions due to the covid-19 pandemic, running the validation tests where the case study actually is was very hard, therefore, we opted to carry them out in an airfield near Rivoli in the province of Turin.

This site neither has vineyards nor a particularly rough terrain, unlike the case study. Since the conformation and the specificities of the vineyard had to be replicated, the DTM and the starting orthophoto of the airfield were modified as follows:

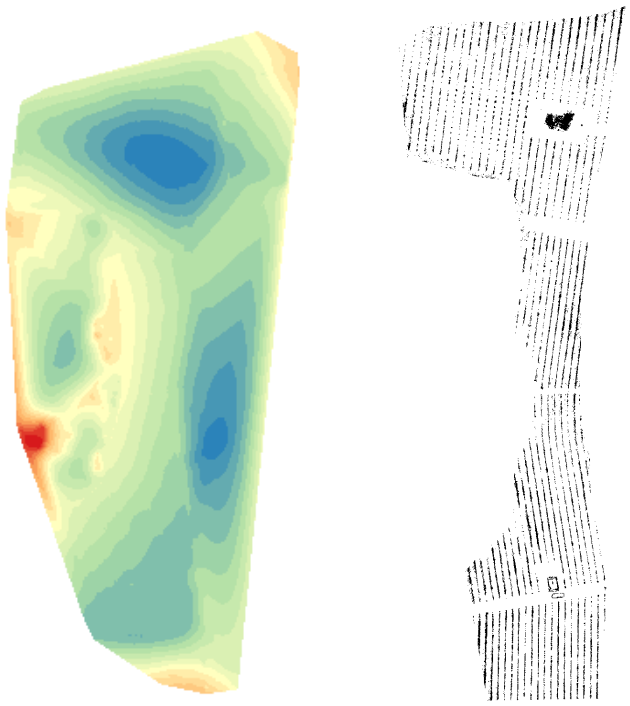


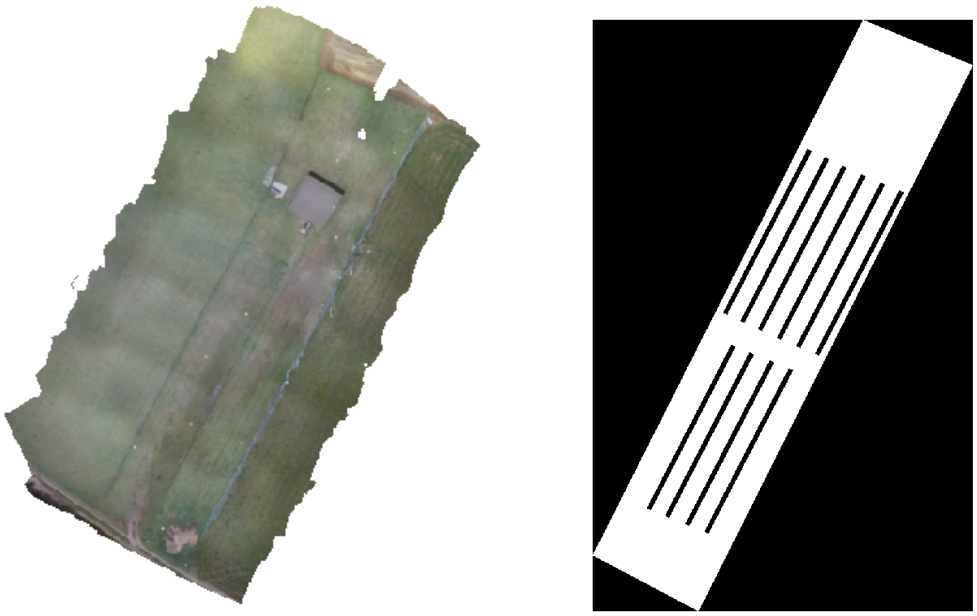
Figure 2.4: (a)Spectral view of the DTM, (b)Base mask of the vineyard.

1. the orthophoto shown in Figure 2.5a was modified through QGIS, shifting from three bands to one;
2. two polygonal shape files were created using the function offered by QGIS, one is a rectangular section of the modified orthophoto that occupies almost all the useful terrain of the airfield and some polygonal vectors having the same size and shapes like vineyard rows were also created, to be placed as obstacles;
3. using the *Symmetrical difference* function between the two shape files, a third then is generated in practice, in other words the base ground minus the vineyard rows we have subtracted;
4. the binary mask shown in Figure 2.5b is generated using the *rasterize* function, setting the parameters as shown in Table 2.1;
5. the DTM is cut out using the same dimensions of the raster of the binary map.

Obviously both the DTM and the mask files were set with the same resolution used for the Baldicchieri's map (0.1 m).

Table 2.1: Setting the correct parameters of the *Rasterize* function for creating the binary map.

Parameters	**Value**
Input layer	Final shape file
A fixed value to burn	1
Output raster unit of measurement	Georeferenced units
Horizontal resolution	0.1
Vertical resolution	0.1
Extension	Same as the input layer
Output data type	Byte



(a) *Orthophoto of the flying field* (b) *Binary mask*

Figure 2.5: (a) Orthophoto and (b) binary mask of the flight field near Rivoli (TO).

2.3 Unmanned ground vehicle

One of the significant and necessary element to choose is the UGV to be used for both the simulations and the field tests. First of all, a definition and a glimpse of what is the UGV landscape will be provided. The term UGV defines a land vehicle without the physical presence of people in control. This type of vehicle is

used in many fields and applications and three main types of vehicle control can be pinpointed as described in [13]:

1. manual, in which direct signals are usually sent to the vehicle actuators by means of a remote radio control;

2. supervisory control, where a human supervisor gives very general commands to the vehicle remotely and the vehicle can perfectly interpret those commands and break them down into simple, more detailed actions. At the same time the supervisor can check if the vehicle is performing the task correctly. This level of control represents an intermediate stage between manual and fully automatic control;

3. automatic, in which the vehicle can interact with the surrounding enviroment and make decisions autonomously on the base of the assigned task.

The most classic areas of use are those where a unmanned vehicle must be used for hazardous tasks such as the defusing of a bomb or the treatment of dangerous material, or if the place to reach is difficult to access for a human being or even to better handle heavy objects. As a consequence, the areas of interest can range from military, to extraterrestrial exploration or even civilian ones such as agriculture, manufacturing, mining and emergency rescue missions.

The vehicle that was selected for this project is the Traxxas X-Maxx, a very compact radio-controlled off-road vehicle, that can reach an over 80 km/h speed on flat terrain.

This vehicle is chosen due to two main factors making it suitable for the final application. The first of these two factor is the size of both the wheelbase and the axle track reported in Figure 2.6 which ensures a particularly good rollover stability even when the vehicle is climbing steep climbs and also provides a large central surface, as can be seen in Figure 2.7, that can be used to install various navigation components.

The second factor is represented by the combination of the four-wheel drive with 4 high-performance GTX Alluminum Shocks suspensions that give the vehicle excellent traction and stability even on rough terrain. Another interesting aspect of this vehicle is its turning radius, so that allows for maneuvering even in small spaces. Considering a maximum turning angle of $\theta = 35°$, the minimum turning radius is calculated as follows:

$$radius = \frac{wheelbase}{\tan(\theta)} = 0.69m \tag{2.1}$$

All the other specifications of the vehicle are reported in the Table 2.2 below.

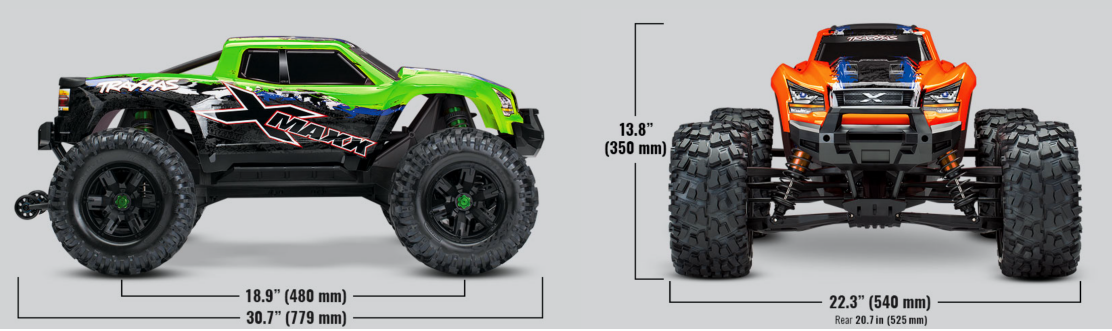


Figure 2.6: Traxxas X-Max external dimensions

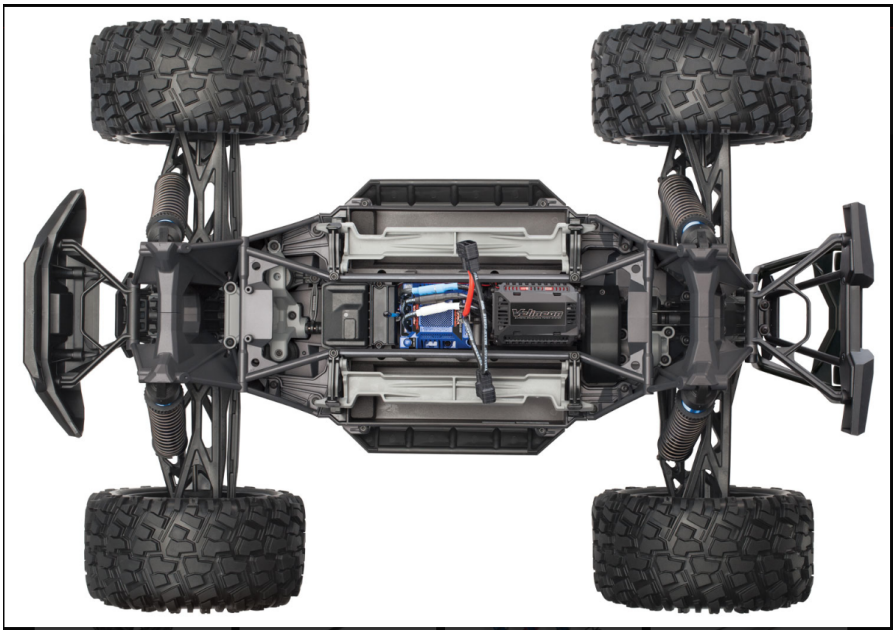


Figure 2.7: View from above of the internal anatomy of the Traxxas X-Max.

The vehicle here presented, was not originally ready for sensors to be installed on, therefore, some adjustments were made. Firstly, the upper body and its bearing structure were disassembled to have bare structure as shown in Figure 2.7. Then a custom made fiberglass support plate was made, to be used as a base to install the sensors and the flight controller with the addition of two other smaller plates mounted above the base plate, on the front and rear axes respectively, as shown in Figure 2.8.

This configuration, although not definitive, allowed us to have a comfortable support surface so that the sensors were configured more freely, and some room was made avaiable to install additional sensors in the future, for the vehicle to be

Table 2.2: Detailed list of all the vehicle specifications.

Components or Features	**Specifications**
Length	30.67 inches (779mm)
Front Track	22.26 inches (540mm)
Rear Track	20.66 inches (525mm)
Ground Clearance	4 inches (102mm)
Weight	19.1lbs (8.66kg)
Height (ride)	13.79 inches (350mm)
Wheelbase	18.92 inches (480mm)
Shock Length	7.4 inches (187mm)
Tires (pre-glued)	8.0x4.0 inches (203mm)
Wheels	4.3x5.7 inches (110x145mm)
Wheels Hex Size	24mm splined hex
Speed Control	Velineon VXL-8s
Motor (electric)	Velineon 1200XL
Transmission	Single-Speed
Overall Drive Ratio	8.11 (stock, out-of-box)
Differential Type	Sealed, Hardened Steel Bevel, Limited Slip
Gear Pitch	1.0-Pitch
Chassis Structure/Material	Composite Nylon Tub
Brake Type	Electronic
Drive System	Shaft-Driven 4WD
Steering	Bellcrank
Radio System	TQi™ 2.4GHz Transmitter with TSM® receiver™
Servo	Torque: 365oz. Speed: 0.17 sec/ 60 degrees(6.0V)
Top Speed	50+ MPH with two 4s LiPo batteries and optional pinion gear
Skill Level	6
Battery Tray	197mm x 51.5mm x 44mm
Required Batteries	4 “AA” (transmitter)

enhanced.

The upper plated that was mounted on the rear axle of the vehicle is a highly remarkable detail since it leaves some room to mount the antenna of the GNSS

Figure 2.8: Vehicle with its custom made fiberglass backing plate.

module, which will be discussed in detail in the hardware section of this chapter. The antenna is placed in a raised position thus favouring the reception of satellite signals and coincides exactly with the position in which the waypoints are calculated in the algorithm.

Then some holes were drilled both on the base plate, to let the flight controller power, steering and throttle control cables pass through, and on the rear top plate to secure the GNSS module antenna. The final result can be seen in Figure 2.9.

Figure 2.9: UGV final configuration.

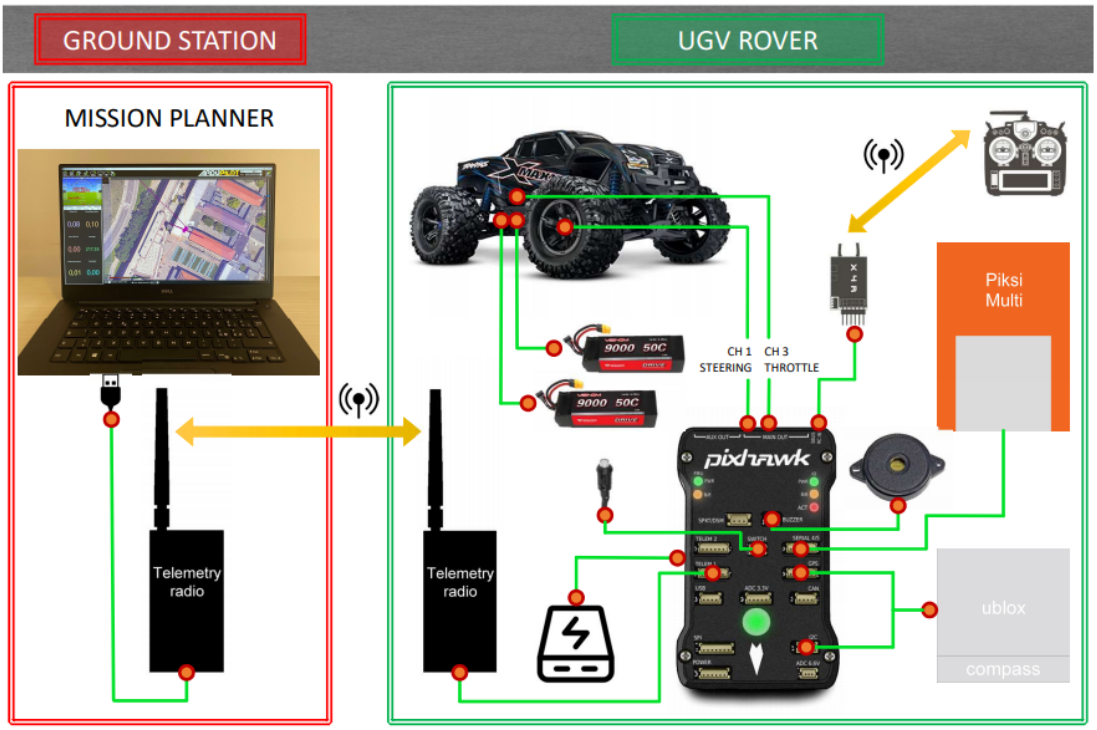


Figure 2.10: UGV connections diagram [2].

2.4 Software

With reference to chapter 1, in particular with Figure 1.1, there are three main types of software needed for the application:

- a software responsible for acquisition, recording, analysis, display, sharing and presentation of the information resulting from the geographic data to correctly generate the representation layer, this type of system is called *GIS* (Geographic Information System);
- an *IDE* (Integrated development environment) working efficiently with large vector or matrix data (deriving from raster maps), that can implement and simulate algorithms, plot graphics and generate the code in the most suitable language for the chosen hardware in case an embedded implementation is required;
- a *GCS* (Ground Control Station) software transmitting the useful information, which resulted through the algorithm, to the flight controller and receive the information from the sensors on board, in order to supervise the mission and various parameters in real time.

GIS As regards the GIS software, it was decided to use one of the most popular software in this field known as *QGIS*. *QGIS* is an open source GIS desktop application, the project was born in 2002 from a Gary Sherman's idea and subsequently incubated by the Open Source Geospatial Fundation in 2007.

Its first version was officially released in January 2009, the last stable published version is 3.12 *Bucuresti* of February 2020. The software is published as cross-platform and is available for MacOS, Linux, UNIX, Microsoft Windows and Android.

This software was mainly used to analyze the raster maps and the point cloud of the vineyard, adapt the resolution and size of the DTM from the previously created binary mask and store these files in GeoTiff format, which allows for embedding geographic references within a Tiff image.

Potentially, it can include projections, ellipsoids, datums, coordinates, and everything else needed to establish the exact spatial reference for the file. Storing files in GeoTiff format was useful to georeference raster maps within the developed algorithm.

IDE The algorithm was developed thanks to a well know software known as *MATLAB*, also used for numerical computation and statistical analysis. This software was created in the late 1970s by Cleve Moler, who together with two

other engineers rewrote MATLAB in C language and founded *The MathWorks* in 1984. The main features of this software are:

- its ability to manipulate matrices in an easy and intuitive way;
- its ability to visualize functions and data in detail with highly customizable plots;
- the possibility of implementing algorithms;
- its ability to create user interfaces;
- the possibility to interface with other programs;
- the possibility of generation and verification procedures in C/C++ for prototyping, implementation and integration of the software;
- its extensive library of functions and several toolboxes that greatly facilitate the development of algorithms and control systems for applications in various fields;
- the possibility to use NVIDIA® GPUs to accelerate the activities of artificial intelligence, deep learning and other computationally expensive analysis.

MATLAB is used by millions of people in the industrial and the academic world industry due to its wide range of tools to support the most diverse applied fields of study and runs on a variety of operating systems, including Windows, Mac OS, GNU/Linux and Unix. The choice of the programming environment was dictated by three essential factors:

1. matrices and vectors, widely present considering how the nature of the representation was conceived, had to be easy to manipulate;
2. its ability to read GeoTiff files, work separately on the matrix of values and other geographic information in the GeoTiff format, and merge all this information to create a cost map where to navigate;
3. the inner presence of the RRT* algorithm, identified in the previous paragraph 2.1 as the basic starting algorithm to achieve the final objective, with the possibility to modify the main parameters simply and intuitively.

GCS The third necessary element is the identification of a Ground Control Station that can plan, monitor and correct the autonomous driving operations of the vehicle starting from what the algorithm provides, the georeferenced waypoints, in our case.

Starting from this assumption, the choice fell on Ardupilot's Mission Planner Figure 2.11 that is a free, open-source, community-supported application designed and built by Michael Oborne for Plane, Copter and Rover controll in 2010.

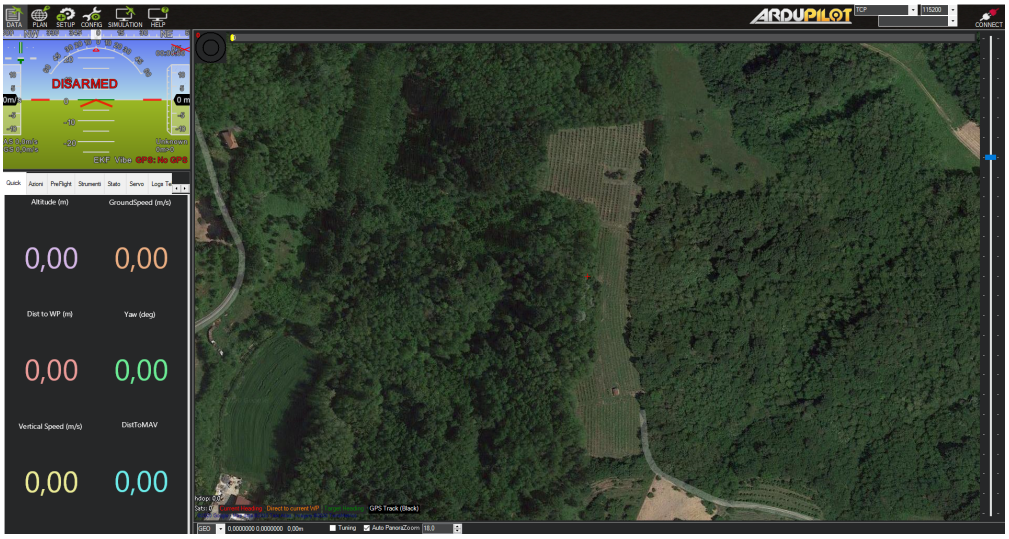


Figure 2.11: Home of Mission Planner software overlooking the Baldicchieri d'Asti vineyard.

It is arguably one of the most comprehensive open-source GCS and offers all the configuration and analysis tools needed for vehicle control. This GCS can perform many different tasks, here are some of the key ones:

- load the firmware into the autopilot board (i.e. Pixhawk series) that controls your vehicle;
- setup, configure, and tune your vehicle for optimum performance;
- simulate planned missions through the SITL (Software In The Loop) simulator;
- plan, save and load autonomous missions into you autopilot with simple point-and-click way-point entry on Google or other maps;
- download and analyze mission logs created by your autopilot;

- interface with a PC flight simulator to create a full hardware-in-the-loop UAV simulator;
- monitor your vehicle's status while in operation;
- record telemetry logs which contain much more information the the on-board autopilot logs;
- view and analyze the telemetry logs;
- operate your vehicle in FPV (First Person View);

One of the fundamental features of this GCS software is its ability to import *waypoints* files to plan the mission. The GCS will be provided with the detailed information of the flight plan that the algorithm offers us, through this method.

MAVLink is a communication protocol used by Mission Planner, a very lightweight messaging protocol to communicate with drones (and between onboard drone components). Reading the protocol guide we see that the waypoints format that the software can read is a structured text format as shown in Figure 2.12 where the first line contains the file format and version information, while the subsequent line(s) are mission items.

Thanks to this, it can be ensured that the output of the algorithm is a text file containing the waypoints of the mission and it can be uploaded to the GCS easly and quickly.

```
QGC WPL <VERSION>
<INDEX> <CURRENT WP> <COORD FRAME> <COMMAND> <PARAM1> <PARAM2> <PARAM3> <PARAM4>...
 <PARAM5/X/LATITUDE> <PARAM6/Y/LONGITUDE> <PARAM7/Z/ALTITUDE> <AUTOCONTINUE>
```

Figure 2.12: Waypoints file format from MAVLink guide.

2.5 Hardware

For the vehicle to move around autonomously, some hardware components adapting to the needs of the project (information/instructions provided by the algorithm) and act as an intermediary between the software part and the actuators of the vehicle are required.

The thesis work of a colleague [2], carried out with the same intent, precision positioning, was considered for the set of hardware components, their installation

and the setting of their fundamental parameters. Therefore only the main aspects will be reported, for all the details refer to the aforementioned thesis.

Referring to the choices made in terms of software where Ardupilot's Mission Planner was identified as Ground Control Station, the same software guide recommends using *Pixhawk 1* as flight controller, reported in Figure 2.13.

The other necessary nevigation peripherals were some of those recommended by the guide given the plug&play compatibility.

Figure 2.13: Pixhawk 1

Pixhawk The Pixhawk 1 autopilot is a flight controller based on the FMUv2-Pixhawk hardware project (combines the functionality of PX4FMU + PX4IO) and runs the PX4 code on the NuttX operating system, all the specifications on this product can be found in table 2.3.

The feature of this autopilot that is most interesting to the goal of this work, it is its support to the redundancy of GNSS modules and inertial measurement units (IMUs). The necessary peripherals identified in [2] to complete the operational configuration are:

- a high-performance Piksi Multi GNSS module;
- a GNSS module with U-Blox chip and integrated compass (used as secondary module);
- two radios for telemetry, operating on the 433 MHz frequency;
- a power supply module (Battery Eliminator Circuit - BEC);
- an 8Gb micro SD memory card;

- a Safety Switch + LED;
- an audio buzzer;
- a FrSky X4R model receiver, with S.BUS OUT[1] channel;
- a radio transmitter model FrSky Taranis X9D Plus;
- wiring with JST-GH connector at one end and DF-13 at the other;
- support with 4 anti-vibration dampers.

Table 2.3: Pixhawk 1 Specifications

Component	**Specifications**
Processor	32-bit ARM Cortex M4 core with FPU
	168 Mhz
	256 KB RAM
	2 MB Flash
	32-bit failsafe co-processor
Sensors	MPU6000 as main accel and gyro
	ST Micro 16-bit gyroscope
	ST Micro 14-bit accelerometer/compass (magnetometer)
	MEAS barometer
Interfaces	5x UART serial ports, 1 high-power capable, 2 with HW flow control
	Spektrum DSM/DSM2/DSM-X Satellite input
	Futaba S.BUS input (output not yet implemented)
	PPM sum signal
	RSSI (PWM or voltage) input
	I2C, SPI, 2x CAN, USB
	3.3V and 6.6V ADC inputs
Characteristic	External safety switch
	Multicolor LED main visual indicator
	High-power, multi-tone piezo audio indicator
	microSD card for high-rate logging

[1]The presence of the S.BUS OUT (or PPM OUT) channel allows you to forward the inputs of the transmitter to the flight controller, remembering that as you can see from the Pixhawk specifications table 2.3 there is only the S.BUS input.

GNSS module The other hardware component that is fundamental for this thesis work is the aforementioned high performance GNSS module Piksi Multi Figure 2.14 made by the American *Swift Navigation* company. This type of module, as already mentioned in section 1.1 dedicated to sensors, improve the integrity, accuracy and availability of GPS.

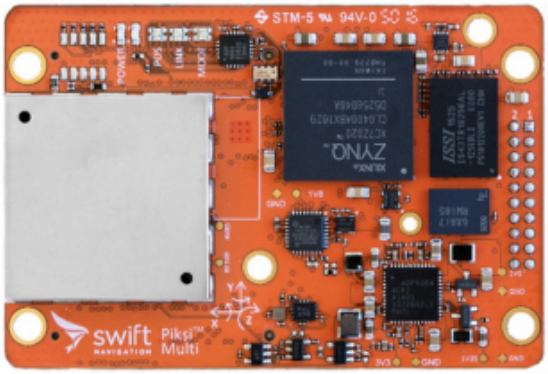


Figure 2.14: Piksi Multi board.

The module was chosen for its peculiar characteristics that make it suitable for the purpose such as dimensions, weight, its ability to determine the position quickly (necessary element in controlled navigation via satellite positioning systems) and compatibility with the Pixhawk flight controller. In detail, the main features are:

- centimeter accuracy;
- receiving RTK corrections;
- dual frequency multicostellation receiver;
- robustness of positioning information;
- rapid convergence to the RTK solution;
- low latency of solutions, <30 ms;
- adaptable interfaces: UART, Ethernet, CAN, USB;
- communication protocols: Swift Binary Protocol (SPB) and NMEA 0183
- integrated Micro Electro Mechanical Systems (MEMS) IMU and magnetometer;

The above mentioned RTK corrections are necessary for precision guidance and result from a differential technique called *Real Time Kinematic*. Since the understanding of how the technique works is beyond the scope of this thesis, a brief summary is provided below.

Two or more receivers are used in this technique, in which one of the two called *master station* is positioned in a fixed way in a known point acquiring its coordinates in a precise way and calculating the *Pseudorange correction (PRC)* and the *Carrier phase correction (CPC)* and transmitting the aforementioned corrections almost instantaneously to the receiver mounted on the vehicle so as to greatly increase its accuracy. As underlined in [2] the rapid convergence to the RTK solution is of great importance in case the signal is lost when the navigation start and the fact of being a multi-constellation module increases the robustness of the solution and it also ensures a significantly better performance in environments with poor or hindered sky visibility.

The other interesting aspect in the use of this product is that two L1/L2/L5 GPS/GLN/BDS survey antennas reported in Figure 2.15 are also included in the Piksi Multi kit. These antennas are well-performing given the possibility of using the GPS L5 signal providing a safe and robust radio navigation means for critical applications. The signal is broadcast in a frequency band protected by the ITU for aeronautical radio-navigation services. The L5 band providing additional robustness in the form of interference mitigation, the band being internationally protected, the redundancy with existing bands, the geostationary satellite augmentation, and the ground-based augmentation are all valid aspects for both land and non-land vehicles, but given their dimensions, antennas like these are rarely used on medium and small drones.

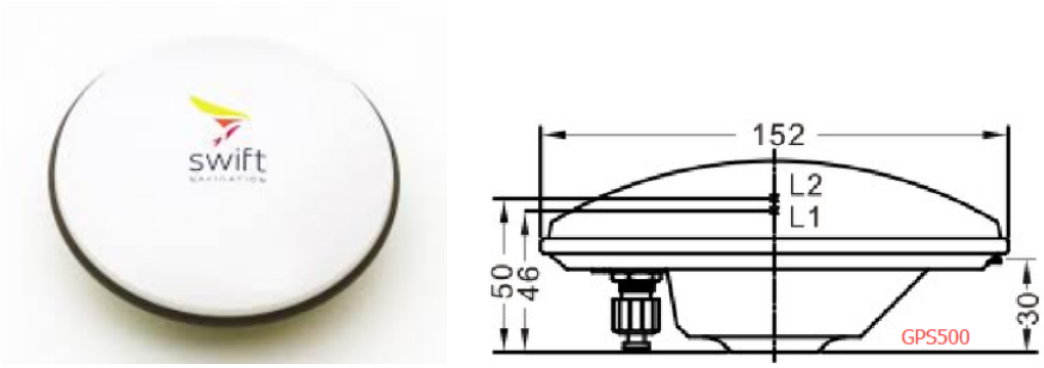


Figure 2.15: Piksi Multi survey antenna by Swift Navigation and its dimensions.

Chapter 3

Code

The main objective of the developed Code is to provide a text file, where the waypoints are saved, which can be immediately load on the Ground Control Station. This, however, is not the only thing that is aimed to achieve, as already mentioned in the introduction. In fact, since the environment where the rover will move is a predominantly uneven terrain, the most relevant purpose of this project is the creation of a tool providing a safe path in terms of vehicle stability and trying to minimize the distance between the starting and the arrival points.

Identifying the basic algorithm and the type of representation to use was the most demanding task of this work. At first, a more complex representation of the environment had to be provided, probably through the use of cameras or tools like lidar, to create a representation directly in 3D in real time or offline.

However, two problems arose during the early stages of the thesis work, the first was the Italian national lockdown in early March, which made impossible any vehicle sensor testing, and spread a great deal of confusion, in general. The second problem was the size and therefore the excessive weight of the map as a point cloud. For these reasons, the method of representation through raster maps was adopted, as described in section 2.2.

The basic idea was to build a code allowing the user to interactively select the start point, the end point and even possible intermediate points on the map to have the algorithm iteratively searching a complete route, checking thoroughly every single pose, to guarantee a certain stability, if high precision standards for the map are met.

This is because, as it happens most path planners based on a representation of this type, in non-perfectly planar environments, the controller does not grasp that the vehicle configuration prevents the same to run that route, and so the wheels receive too much or too little torque or, in general, this leads to an unnatural reaction and may drift the vehicle off the planned route.

An overview of the highly articulated logic and the full development of the code

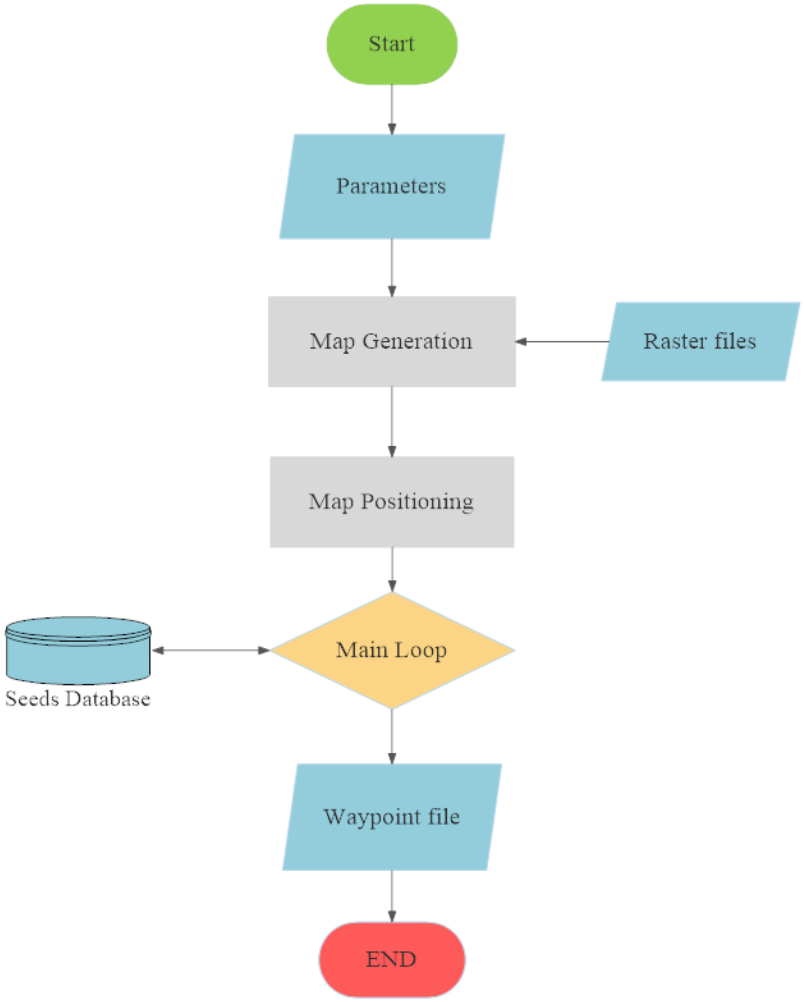


Figure 3.1: High-level code flow chart.

is better schematized in the flow chart shown in Figure 3.1, where the process is split into four fundamental sections.

- The first one regards which specific parameters will better adapt the part of the code meeting the user's needs and those of the environment to navigate.
- The second section is dedicated to the loading of Geotiff files regarding the raster maps and the consequent creation of the cost map.
- The third section concerns a part of the code that allows the user to interactively select the points on the cost map to easily designate the initial and the final positions and any intermediate points, with the consequent creation

of a table to store the coordinates so that the algorithm can work on one section of the route at a time.

- The last section is the heart of the whole code, the main loop performing different tasks, including planning and controlling in a trial and error version, as it will be explained in the next section.

3.1 Code detailed explanation

3.1.1 Parameters Settings

Some parameters, affecting the whole code, are initialized in this first section. All the parameters and variables that must be configured and tuned to make the code work properly are listed in Table 3.1.

The first and perhaps most important parameter of the code is called *PlanarDiff*. This variable indicates the tolerance, expressed in meters of difference between one pose and the next, for both the control on the slope and its variation.

In this case, the 0.1 meter value proved to fit perfectly with this map in terms of trade-off between computation time and the average slope level of the terrain, but this value may be insufficient or too strict in other environments other than the specific case.

Obviously, applying increasingly stringent values, the computation time increases dramatically and no path may fall within the selected parameters in some cases.

This parameter is closely related to another one called *PoseDistance*. This represents the maximum distance imposed to the RRT* algorithm to connect two successive poses. This makes clear how, if the distance between the two points and their maximum height difference is know in advance the slope and its variation can be controlled.

This assumption is certainly valid as long as the distance between the two points remains below a cartain value. This is the reason why the map and the GPS measurements must be fairly accurate and precise. The usage of the mentioned values for these two parameter leads to the ideal construction of a maximum slope that is graphically visible in Figure 3.2.

A series of parameters is then used to build the vehicle and the map to simulate within. The first of these series, called *Cellsize*, represents the side length of each square cell forming the grid to build the cost map. This parameter is dictated by the size of the pixels constituting the raster maps, even if the resolution is editable through QGIS, a certain compromise with the accuracy value of the tools with which the maps were created has to be found.

Table 3.1: Parameters Settings

Parameter	**Value**
PlanarDiff	0.1
Cellsize	0.1
PoseDistance	0.3
VehicleDims	refer to Figure 2.2
ObstacleIR	0.4
maxtrial	3
DynamicIterations	8
surfsize	3
WeightMatrix	[1,1,1; 1,3,1; 1,1,1]
utmzone	32T
WPdistance	2
DistEffThreshold	1.3
SeedDestination	'File name.txt'
MissionWPDestination	'File name.txt'

The other two parameters for the construction of the map are *VehicleDims* and *ObstacleIR*, the first refers to an object created by matlab called *vehicleDimensions* allowing to insert and store the dimensions of the vehicle, including a very complete description of it, as can be seen in Figure 3.3.

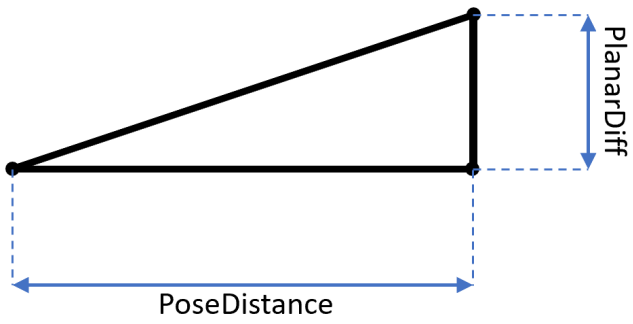


Figure 3.2: Visual example of what type of slope can result using the same value used in the code.

The second parameter, where the name stands for Obstacle inflation radius, was introduced. This parameter is used to modify the radius created by the *vehicleCostmap* function around obstacles, as can be seen in Figure 3.4, to prevent

the vehicle from approaching them during the simulated creation of the route and therefore also in real navigation.

This function is usually based on the parameters regarding the size of the vehicle, for the radius to be calculated, but since this was the only parameter to operate on, in case this had to be increased for the sake of safety or decreased for other factors, it was decided to maintain some sort of differentiation between the vehicle inflaction radius and that of the obstacles.

Therefore the parameter might be modified according to the users's personal needs knowing that the choice is also a function of how well the main binary mask is made, the recommended values for the vehicle used range from 0.35 to 0.5.

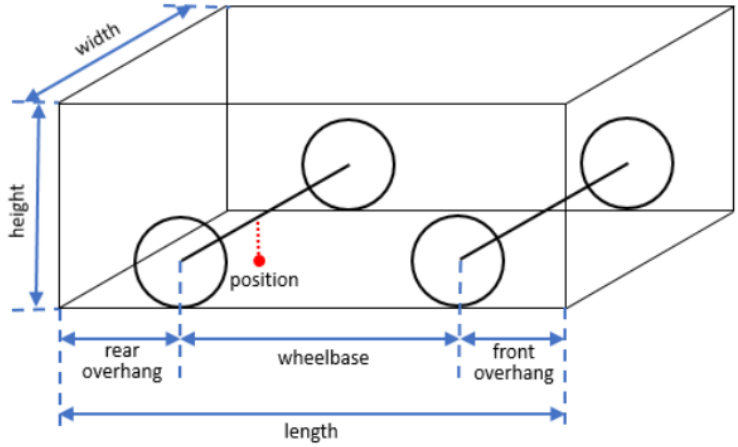


Figure 3.3: Vehicle parameters can be edited using the *vehicleDimensions* object (`https://it.mathworks.com/help/driving/ref/vehicledimensions.html`).

Then a series of useful variables is configured for the main loop part. Two of these are counters whose only purpose is to dynamize the algorithm, in the sense that if there is no valid solution for the path within a certain number of cycles, a series of parameters and thresholds, that could increase the probability of finding a valid path, automatically update. These are *maxtrial* and *DynamicIterations*, and their exact function will be later explained with the part about the main loop.

The other two parameters are used to define: the size of the contact surface of the wheels to be taken into account and the weighed value of each individual cell within the chosen contact surface. They are *surfsize* and *WeightMatrix* respectively.

The last five parameters are used to import and save seeds and waypoints correctly. The first of these is called *utmzone* and is used to specify the geographic area of the map to be used with UTM coordinates and then convert them into

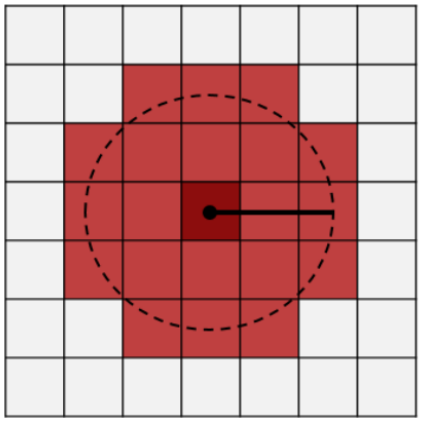


Figure 3.4: Visual representation of the inflated area around an obstacle, the obstacle is the one in deeper red at the center of the grid (`https://it.mathworks.com/help/driving/ref/vehiclecostmap.html`).

Geographic coordinates, used by the Mission Planner. The variable called *WPdistance* is used to write only a portion of the waypoints effectively found by the algorithm. The third one is called *DistEffThreshold* and is a coefficient working as a discriminator to possibly save path found in the seeds file. The remaining two only pinpoint the destination text file name for seeds and waypoints respectively.

3.1.2 Map Generation

Two functions provided by MATLAB for this type of work are used in the section about the construction of the cost map. The first is *geotiffread* that, as the name suggests, reads a georeferenced grayscale, RGB, or multispectral image or data grid from a specific GeoTIFF file and creates a spatial referencing object, so that only the grayscale image is saved in a matrix and all the other relative data of the map are stored in another structure.

The map data are stored in an object structure called *MapCellsReference* by MATLAB containing information shown in Table 3.2.

This function is used for both the mask raster and the DTM raster. It should be noted that MATLAB does not recommend the use of this function and suggests the usage of a new one present in the 2020b version called *readgeoraster*, but the first one had to be used since the version used in this project is the 2019b.

The second function offered by MATLAB is called *vehicleCostmap* and creates a costmap representing the planning search space around a vehicle. It therefore contains information about the surrounding environment and possible obstacles as well as areas that cannot be traversed by vehicle.

The cost map is stored as a 2-D grid of cells. Each grid cell in the cost map

Table 3.2: *MapCellsReference* object stucture.

Propertiy	**Value**
RasterInterpretation	'cells'
XIntrinsicLimits	[0.5,854.5]
YIntrinsicLimits	[0.5,2622.5]
CellExtentInWorldX	0.1
CellExtentInWorldY	0.1
XWorldLimits	[427691.48,427776.88]
YWorldLimits	[4973555.05,4973817.25]
RasterSize	[2622,854]
ColumnsStartFrom	'north'
RowsStartFrom	'west'
RasterExtentInWorldX	85.4
RasterExtentInWorldY	262.2
TransformationType	'rectilinear'
CoordinateSystemType	'planar'

has a value ranging from 0 to 1 representing the cost of navigating through that grid cell as can be seen in Figure 3.5. The status of each grid cell is free, busy or unknown.

Figure 3.5: Example of how the *vehicleCostmap* function works on the costruction of the occupancy grid (`https://it.mathworks.com/help/driving/ref/vehiclecostmap.html`).

The possible input arguments and the editable properties of this function are shown in Table 3.3. As you can see from this table, these were used as input only:

the *C-Cost values*, specified as a matrix of real values in the range [0, 1] called *MapMatrix* which is nothing but the matrix obtained through the *geotiffread* function, and *mapLocation* where the first element of the *XIntrinsicLimits* and *YIntrinsicLimits* vectors from the previously obtained structure *MapCellsReference* was inserted, corresponding to the initial coordinates of the map.

These data are used to georeference the map, since the size of cells of the chosen cost map are the same size as the pixels of the map, this perfect match provides us with a correct georeferencing. In the part about properties, in fact, the variable described in the previous section 3.1.1 called *Cellsize* was inserted as avalue for the *CellSize* parameter.

Table 3.3: List of inputs and properties that the *vehicleCostmap* function provides to obtain a cost map according to specific needs.

Arguments	**Name**	**Assigned Value**
Inputs	C (Cost values)	single(MapMatrix)
	mapWidth	(not used)
	mapLenght	(not used)
	costVal	(not used)
	occMap	(not used)
	mapLocation	[XWorldLimits(1) YWorldLimits(1)]
Properties	FreeThreshold	0.2 (default)
	OccupiedThreshold	0.65 (default)
	CollisionChecker	inflationCollisionChecker (default)
	MapExtent	[xmin xmax ymin ymax] (default)
	CellSize	Cellsize
	MapSize	[nrows ncols] (default)

The properties of the *CollisionChecker* are then updated with the previously selected values regarding the vehicle size and the inflation radius of the obstacle called *VehicleDims* and *ObstacleIR* respectively. The cost map shown in Figure 3.6 is plotted on screen as the last step of this section. This map plot will be useful in the next section to enter the coordinates of the route.

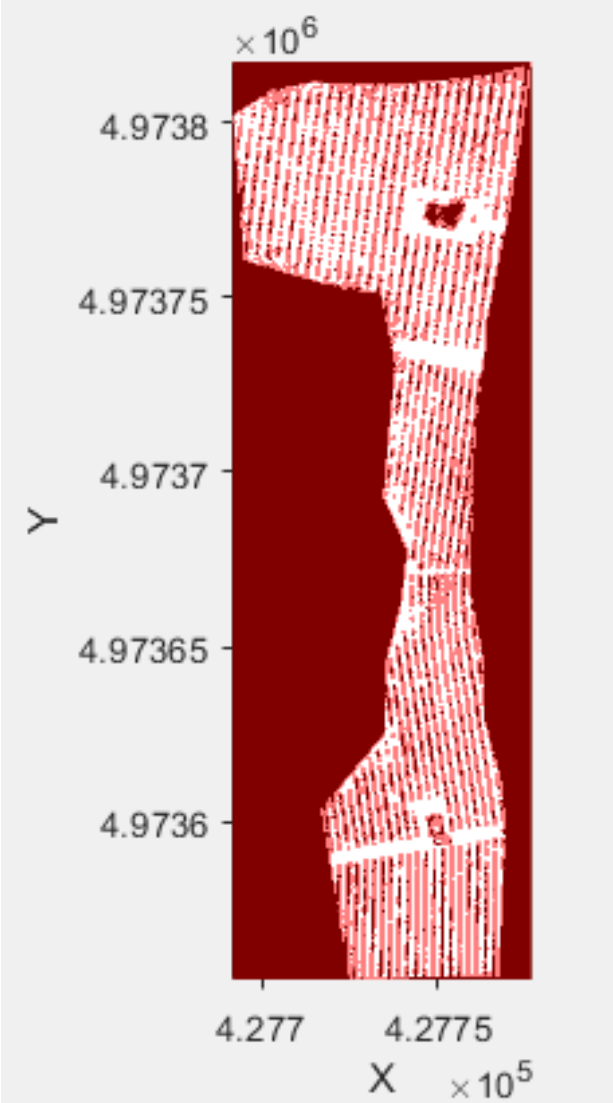


Figure 3.6: Plot of the georeferenced cost map.

3.1.3 Map Positioning

The third section of the code regards the positioning part on the map and consists of the coordinates of the start, end and intermediate points, by defining these the total route plan can be subsequently created. The coordinates on the axes of the map can be identified through the *ginput* function provided by MATLAB, moving the cursor to a certain location and press either the mouse button or a key on the keyboard. The right button of the mouse must be pressed to stop selecting the points.

This function has a structure thanks to which the x and y coordinates can be stored in a vector. The structure of the route plan table is reported in Table 3.4.

Table 3.4: Global route plan table example where with *Midn* we indicate the n-th intermidiate point.

StartPose	**EndPose**	**Attributes**
StartX, StartY, Startθ	Mid1X, Mid1Y, Mid1θ	Attributes
Mid1X, Mid1Y, Mid1θ	Mid2X, Mid2Y, Mid2θ	Attributes
Mid2X, Mid2Y, Mid2θ	Mid3X, Mid3Y, Mid3θ	Attributes
Mid3X, Mid3Y, Mid3θ	EndX, EndY, Endθ	Attributes

In the table it is shown that in the nth row there are three elements representing the n-th section of the route to search for: the starting position, the finishing position and a part about attributes.

The attributes part is nothing more than a structure where some parameters such as the maximum allowed speed and the speed to reach the end point of the route section are indicated.

The so built table structure of the route plan is necessary given the subsequent use of a class of help provided by MATLAB called *HelperBehavioralPlanner* which acts as a simplified behavioral layer in a hierarchical planning workflow, whose operation will be explained in the next subsection about the main loop.

In addition to the above, a cellular structure called *TotalRefPath* was also created, to save every single segment, of each section of the path, between one pose and another, to make it easier for us to compile the text file with the waypoints.

3.1.4 Main Loop

The fourth section is about the main cycle but before describing it some variables, initialized before the cycle starts, need to be explained in detail. The first of these is the behavioral planner, whose important function is to call each segment of the path once the search for the previous one is over.

This processs is managed thanks to the help block called *HelperBehavioralPlanner* that needs the table structure of the route plan and the value of the maximum steering angle, matching the one described in section 2.3 about the vehicle used, as input.

The initial speed of the vehicle, assuming this starts its navigation from standstill, is set equal to zero. The counter that will save each macro segment within the previously explained structure called *TotalRefPath*, and the variable called *SeedVariation*, which was initially set to zero, are then updated. The latter will help with the random variation of the seed at each internal cycle.

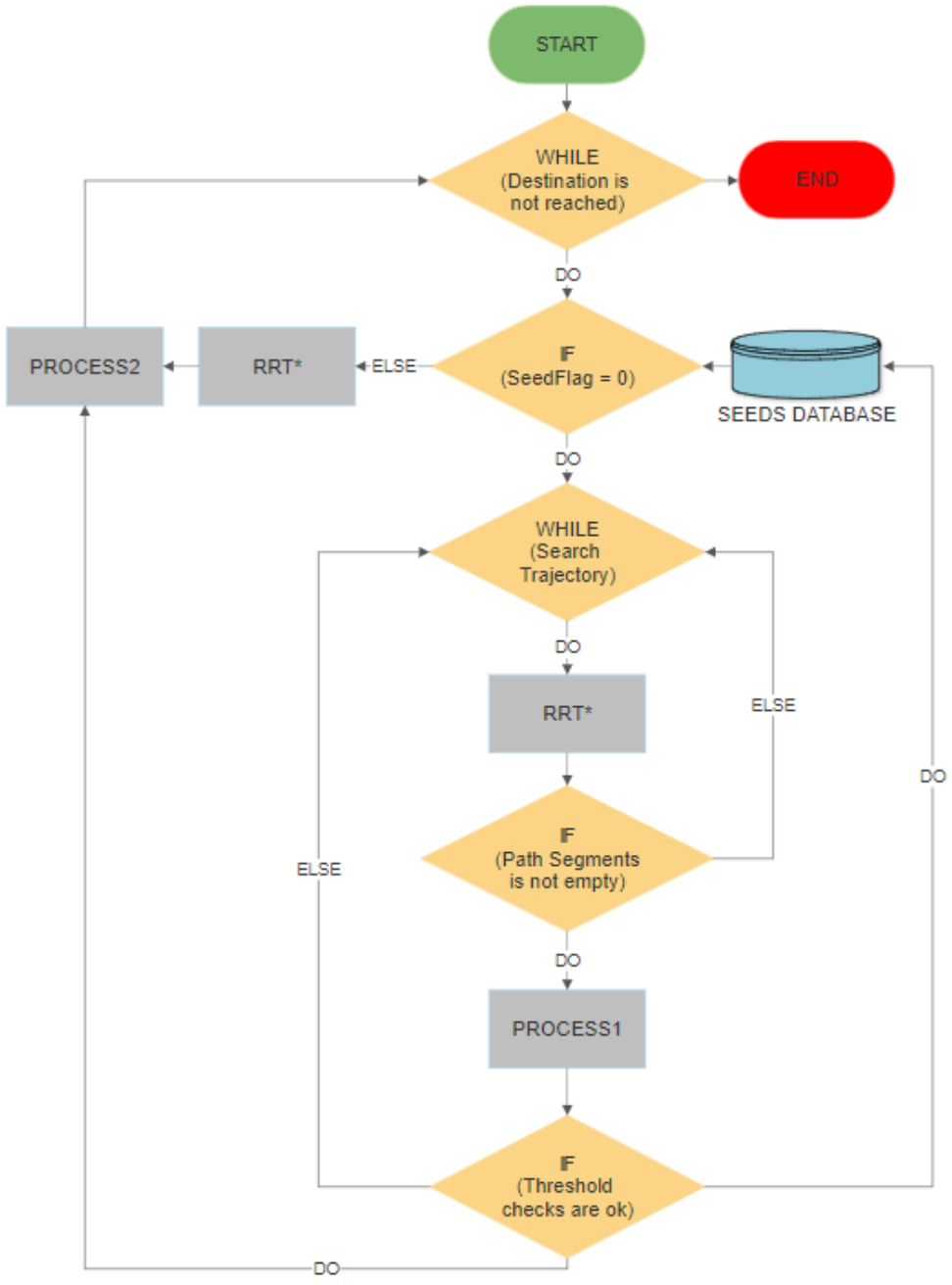


Figure 3.7: Simplified flow chart scheme of the main loop of the code.

Since the logic of the loop is articulated, a simplified flow chart of the main loop is reported in Figure 3.7, i.e. what is inside the decision block, previously shown in Figure 3.1, to have an initial overview of what will be explained in detail below.

The process starts with a *while* loop that will repeat as many times as the number of macro segments of the route plan. All this thanks to an internal function of the behavioral planner called *reachedDestination* telling whether the segments to be calculated are over or not. Therefore, the loop is set as follows: until a valid route to reach the final goal is found, do the following.

The coordinates and the various configurations (which you get from the *Attributes* section of the route plan) of the next manoeuvre are then saved through another function of the behavioral planner called *requestManeuver*.

Some counters shown in Table 3.5 are also initialized. These will have different purposes in the inner loop that will be tackled in due time.

Table 3.5: List of some useful counters for the inner loop.

Variable	**Initial value**
slopetrial	1
slopedottrial	1
attempt	0
trial	0
attemptcount	1

Immediately afterwards, the specifically created function, named *filescan*, is used to see whether the macro path segment resulting from the *requestManeuver* function is present in the seed database or not. This function reads the text file containing the seeds that were found and stored from previous simulations. The input and output data are shown in Table 3.6.

Table 3.6: Input and output variables of the *filescan* function.

Input variables	**Output variables**
SeedDestination	flagSeed
currentPose	DistEfficacy
nextGoal	SlopeThreshold
	SlopeDotThreshold
	RNG
	maxIterations

The names of the file to read, already mentioned in Table 3.1, the current location vector of the vehicle and the next goal of the path segment are present as inputs, in the reported order.

The first output is called *flagSeed* and is useful to understand if the seed referring to the input coordinates of the macro segment is present or not. This flag will be used to skip a part of the internal loop to immediately get all the poses of that segment so we can move to the next one.

The other parameters are, in order: the value of the ratio between the real length of the path and the Euclidean distance, the value of the two thresholds found for that segment, the actual seed and the number of maximum iterations with which the segment was found.

Therefore, if the value of the flag is 1, the segment is present in the database and the information on the seed and the number of maximum iterations is directly entered in the RRT algorithm* that immediately finds the segment to get directly to what we called *Process2* within the flow chart 3.7. If the value is 0, the segment is not there and the inner loop must be entered to find a possible path.

Before starting internal cycle, if *flagSeed* is 0, further parameters shown in Table 3.7 are initialized: the maximum number of iterations of the initial algorithm and the three thresholds for slope, slope variation and roll, respectively.

Table 3.7: Parameters initialization needed in the internal loop.

Variables	**Value**
TrajectorySearch	True
maxIterations	10000
SlopeThreshold	PlanarDiff/PoseDistance
SlopeDotThreshold	SlopeThreshold
RollThreshold	30

The threshold for the variation of the slope was set with the same value of the slope that can be seen in Table 3.7, this is for being more stringent with the slope variation. For example, using what is shown in the Figure 3.2 as a reference, the slope may shift from positive to negative, between one pose and another, and according to the values set for the slope threshold, it would be possible to have a variation that is two times bigger as shown in Figure 3.8 for pose 2.1.

On the contrary, by setting the same value for both the thresholds, the maximum variation is bounded to be like in Figure 3.8 for pose 2.2, the exact half. The same holds for a change in slope that goes from negative to positive.

A *while* loop is used for the innermost loop, with the above mentioned *TrajectorySearch* variable, which continues as long as the loop is true.

This becomes false when in the found path there are no verse changes in the direction vectors connecting the poses and when all the flags concerning the different controls are equal to 0, that is they are not activated. In the Algorithm 5 we report

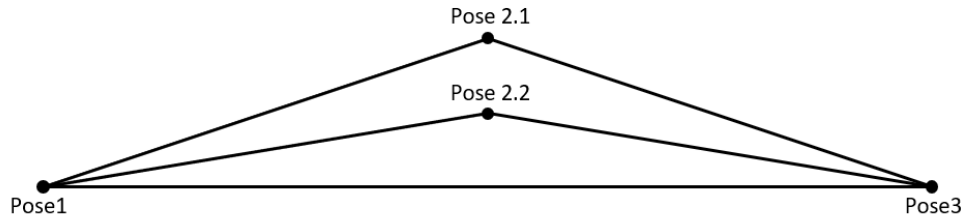


Figure 3.8: Visual example of acceptable and unacceptable slope variation.

the first part of the internal loop until the second condition, which containing the heart of the control, that will be explained later.

Algorithm 5: Inner Loop part 1.

```
while TrajectorySearch do
    rng('shuffle');
    Seed ← Seed + SeedVariation;
    SlopeFlag ← 0;
    SlopeDotFlag ← 0;
    RollFlag ← 0;
    if (attempt/attemptcount) ≥ DynamicIterations then
        maxIterations ← maxIterations + 2000;
        SeedVariation ← number · attemptcount;
        attemptcount ← attemptcount + 1;
    end
    motionPlanner ← pathPlannerRRT;
    [refPath, tree] ← plan(motionPlanner,currentPose,nextGoal);
    attempt ← attempt +1;
    if isempty (PathSegments) ≠ 1 then
        Process1 and Threshold checks;
    end
end
```

What happens inside the *while* is the following:

- a random seed based on the pc clock is generated through the function *rng('shuffle')* offered by MATLAB;
- the seed is updated by adding a variable called *SeedVariation*, that as said in the first part of this subsection is equal to 0, but as we see in the first "*if*" condition it is updated during the various cycles;

- then the flags for the control of the thresholds are initialized;
- in the first condition it is required that every time the ratio between the attempts and the counter (*attemptcount*) exceeds the value of the parameter set at the beginning of the code called *DynamicIterations*, the number of maximum iterations must be increased and the value of *SeedVariation* is updated according to the counter, that will also be updated;
- an object called *pathPlannerRRT* is generated, it configures the vehicle path planner based on the optimal rapidly exploring random tree (RRT*) algorithm;
- the *plan* function, that plan vehicle path using RRT* path planner and save in two vectors the segments of each pose and the ramifications, is then used;
- the counter *attempt* is updated;
- a final check on the status of path segments vector is made, to verify if it is not empty.

In Table 3.8, are shown the properties of the *pathPlannerRRT* function that refers to the algorithm of Karaman and Frazzoli [6] as described in chapter 2 in section 2.1.

Table 3.8: Table of *pathPlannerRRT* properties.

Properties	**Value**
Costmap	map
GoalTolerance	[0.1 0.1 3]
GoalBias	(default)
ConnectionMethod	Dubins (default)
ConnectionDistance	PoseDistance
MinTurningRadius	1
MinIterations	(default)
MaxIterations	maxIterations
ApproximateSearch	false

The *Costmap* parameter needs the previously created cost map called *map* as input. *GoalTolerance* is used to indicate the tolerance within which the algorithm is considered to have reached the target in terms of [x y θ].

ConnectionDistance represents the distance between one pose and the next and the already discussed *PoseDistance* variable is then provided as an input,

same thing for the *MaxIterations* parameter. While the last parameter called *ApproximateSearch* is set to false, to increase the precision and accuracy of the algorithm with which the vertices can be sampled.

The internal code is the focus of the following part. This code is visible in Algorithm 6, regarding to what was defined as "*Process1*" so far, in the flow chart in Figure 3.7, and threshold controls.

If the path planner finds a path connecting the *currentPose* with the *nextGoal* and therefore the *PathSegments* vector residing inside the *refPath* object is not empty, the path control phase can be started. Firstly, the *trial* counter, showing the successful attempts of the planner, is updated and then the Euclidean distance between the initial and the final pose is calculated through the *distance* function.

The poses along the planned route of the vehicle are then interpolated, thanks to the *interpolate* function which, besides providing us with transition poses, also establishes a direction vector, whose 1 value indicates that the vehicle is making a forward movement in that segment, while with -1 a reverse movement is made.

The transition poses are then associated to the center of the rear axis of the vehicle and through the *rear2front* function, the corresponding coordinates in relation to the frontal axis are calculated, assigning them to the *FrontPoses* vector. Since the *transitionPoses* vector has the usual structure [x, y, θ], the above mentioned function takes respectively the x and y coordinates of the rear axle and adds the product of the distance, between the rear center and the front called *VFrontFromRear*, with the cosine and sine of the yaw angle of the vehicle respectively.

Subsequently, the exact coordinates of the center of each wheel, both front and rear, are evaluated for each pose using the function called *axle2wheel*. This function is conceptually identical to the previous one, the value of the angle changes, it can be either $-90°$ or $+90°$ depending on whether the wheel is the right or left one and the distance to consider is the one between the center of the wheel and the center of the axis. An example of what results from the above steps is shown in Figure 3.9.

Once the coordinates of the contact center of the wheels are found, these coordinates must be connected with those of the matrix generated by the second map, the DTM, containing the altitude information of the vineyard. The *Coordinate2matrix* function, transforming the geographical coordinates found on the main map into rows and columns to make them match with the second matrix of the DTM as summarized in Figure 3.10, is used for this purpose. The *meansurface* function is then used to place the points found in the center of a matrix of weights, that extends the contact surface of each wheel by a certain number of cells, according to the value entered in the already mentioned *surfsize* parameter and evaluates a weighted average. Therefore, this function provides us a weighted average of the altitude values of the points of contact that each wheel has with the

Algorithm 6: Inner Loop part 2 (related to what we have defined as *Process1*).

```
if isempty (PathSegments) ≠ 1 then
    trial ← trial + 1;
    MinimumDistance ← distance(stateSpaceSE2,currentPose,nextGoal);
    [transitionPoses, directions] ← interpolate(refPath,lengths);
    RearPoses ← transitionPoses;
    FrontPoses ← rear2front(RearPoses,VFrontFromRear);
    FWheel ← axle2wheel(FrontPoses,HalfFrontAxle,90 or -90);
    RWheel ← axle2wheel(RearPoses,HalfRearAxle,90 or -90);
    FPoints ← Coordinate2matrix(map,FWheel);
    RPoints ← Coordinate2matrix(map,RWheel);
    FWheelSurf
     ←meansurface(FPoints,SlopeMatrix,surfsize,WeightMatrix);
    RWheelSurf
     ←meansurface(RPoints,SlopeMatrix,surfsize,WeightMatrix);
    FSlope ← slopefunction (PoseDistance,FWheelSurf);
    RSlope ← slopefunction (PoseDistance,RWheelSurf);
    FSlopeDiff ← diff(FSlope);
    RSlopeDiff ← diff(RSlope);
    FRoll ← rollfunction(FWheelSurf,Wheelbase);
    RRoll ← rollfunction(RWheelSurf,Wheelbase);
    if max(abs(FSlope or RSlope)) > SlopeThreshold then
        SlopeFlag ← 1;
    end
    if max(abs(FSlopeDiff or RSlopeDiff)) > SlopeDotThreshold then
        SlopeDotFlag ← 1;
    end
    if max(FRoll or RRoll) > RollThreshold then
        RollFlag ← 1;
    end
    TrajectorySearch ← sum(abs(diff(directions))) ≥ 2 || SlopeFlag > 0 ||
     SlopeDotFlag > 0 || RollFlag > 0;
end
```

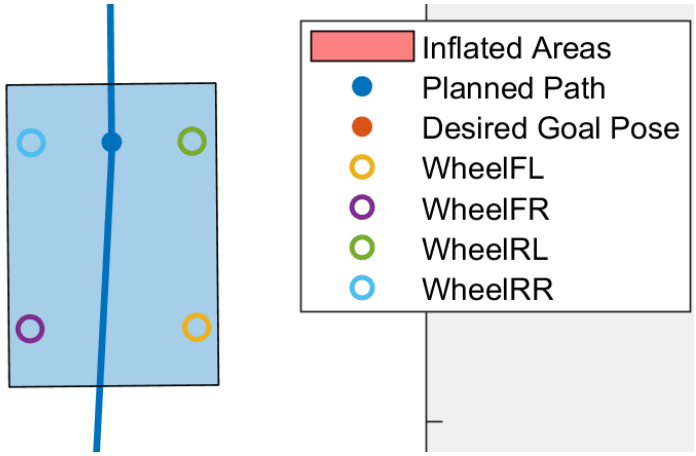


Figure 3.9: Visual example of wheel contact points resulting from a simulation with MATLAB (the front of the vehicle points downwards).

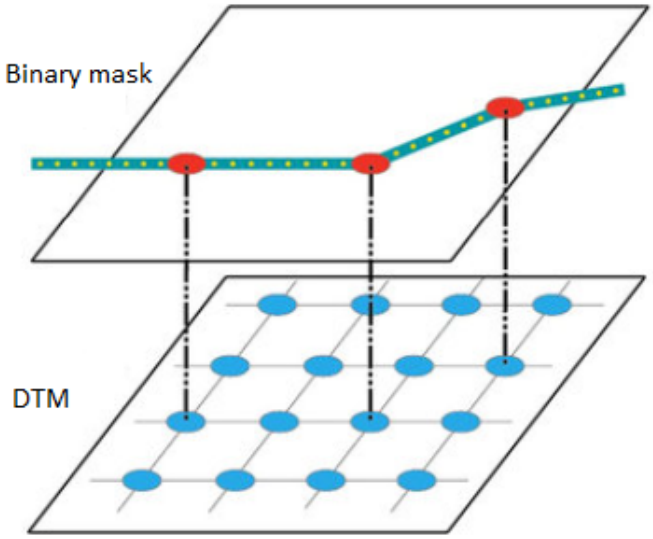


Figure 3.10: Abstraction of the link created between the binary mask and the DTM.

ground.

The choice, to extend the contact surface of each wheel by more than a single cell, is determined by two considerations. The first is that the dimensions of the wheel do not necessarely match those of a single cell (this could be both smaller and larger as in the case of the vehicle used in this work). The second motivation has to do with the fact that the center of contact of the wheel could not match

that of the simulaton for different reasons, during the real mission on the field. This little deviation, only of a few decimeters maybe, could lead one of the wheels to unexpectedly end in a ditch or on a road bump.

This situation may trigger an unexpected behavior of the vehicle or lead to a greater drift from the established route. The visual result compared to the dimensions of the vehicle using the previously *surfsize* value is shown in Figure 3.11.

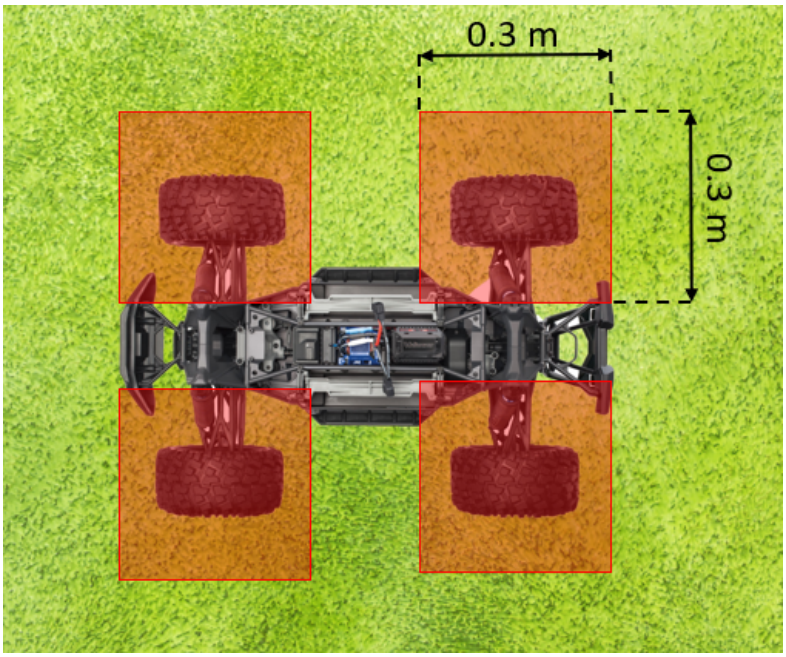


Figure 3.11: Visual result of the idea to use a 3x3 cells as ideal contact surface compared to the vehicle used.

Once the averaged altitude value of each pose for each wheel are obtained using the *slopefunction* function, the slope value of each segment between one pose and the next is calculated and stored in a vector.

Since the distance between two successive poses is small and the value imagined as punctual is averaged over a larger area, the slope value found with this technique can be considered very close to the real slope that the vehicle will face while driving through that segment.

Consequently, the slope variation that each wheel will undergo is evaluated from this series of vectors, containing the slope information for each wheel, using the *diff* function.

The analysis of this piece of information is very important, even more than that on the slope, since a vineyard is rarely a perfectly flat terrain substantial slopes are then common. However, the aim is to find a path allowing a stable

crossing with no sudden holes or bumps that, as mentioned earlier, can lead to a substantial deviation from the planned mission or make the vehicle unstable.

Especially for this reason, the threshold of the slope variation is much more severe than the slope threshold.

In addition to these parameters, the *rollfunction* function, which is called immediately afterwards, calculates the slopes percentage that the front and rear axes will face. The evaluation of this parameter is necessary to check that the vehicle does not experience a difference in slope between the right and left side, such as to risk overturning.

Then, the fact that the maximum absolute value of the slope, its variation and roll must fall within the chosen thresholds is verified through three conditional instructions. In addition within the first two conditions, even if not reported in Algorithm 6, it is also checked if the flags have already been activated a number of times equal to *maxtrial*, through the above mentioned counters, *slopetrial* and *slopedottrial*.

This additional control is added to make the code dynamic, which is to say that if one of the two threshold values is too strict to find a route in that part of the map, the threshold automatically increases by a certain value after a number of failed attempts equal to *maxtrial*.

In case all the checks are successful, i.e. the flags have remained set to 0 and the planned route does not include any reverse maneuver (with reference to $sum(abs(diff(\text{directions}))) \geq 2$), the *TrajectorySearch* variable becomes false from here on, and it is possible to exit the innermost loop to move to what was defined as *Process2* in the flow chart shown in 3.7.

Before providing the description of the last functional block, once it is sure that the path falls within the established values, a series of useful information is saved in the text file about seeds, shown in the Table 3.9, thanks to the so called *filewrite* function that completely characterize the found segment. The latter can be used again in a new path search if the initial and final coordinates of the segment match the previously searched ones, as already explained when the *filescan* function was tacled.

The storage of the seed inside the file is determined by the parameter that was initialized at the outset of the code, called *DistEffThreshold*. In fact, if the ratio between the real distance of the path and the Euclidean one, called *DistEfficacy*, exceeds the selected threshold this step is skipped.

The last procedure of the outermost loop is visible in Algorithm 7, where the current position is updated with the final postion of the recently found segment which is then plotted on the map. In the final step of the process, the segment is inserted in the *TotalRefPath* object, thus updating also the counter needed to split the segments in rows.

Table 3.9: Example of the parameters that are stored in a line of the text file dedicated to seeds.

Parameters	**Values**
currentPose	427742.62 4973778.61 260
nextGoal	427752.08 4973768.23 350
Seed	1301811950
DistEfficacy	1.20
SlopeThreshold	0.33
SlopeDotThreshold	0.33
maxIterations	10000

Once all the segments have been found, up to the final goal, the external loop is exited and thanks to the function called *filewriteWP*, based on the type of format used by by Mission Planner as already stated in section 2.4, using the segments stored in *TotalRefPath* the waypoints file is thus generated. Within *filewriteWP* the *utm2deg* function is called, through which the coordinates from UTM are converted into Geographic (latitude and longitude) using the parameter initialized at the beginning of the code called *utmzone*.

Algorithm 7: Part of the code defined as *Process2*.

```
currentPose ← nextGoal;
plot(refPath);
TotalRedPath(m,1) ← refPath;
m ← m + 1;
```

3.2 Code Simulations

In the above code, the values of some parameters, as well as the presence of some structures inside the code, derive from the choices made during several simulations aimed at evaluating different aspects and behaviors of the algorithm, in a map that is so vast and full of obstacles like the one of the case of study.

Two types of simulations were conducted:

1. The first one was used to evaluate the probability for the RRT* to find a path in three different scenarios, depending on the parameters set. The choice to perform this type of simulations originated when it was realized that the basic algorithm could not always find a possible path to traverse, and this

greatly affects the timing with which the code can provide a valid path, i.e. a path that falls within the established constraints.

2. The second one, carried out using the simulator on the chosen GCS, served to simulate the hypothetical behavior of the vehicle on the field while trying to follow the waypoints. This is done in order to set the basic parameters of the flight controller in the best possible way so that the vehicle can follow the traced path as accurately as possible.

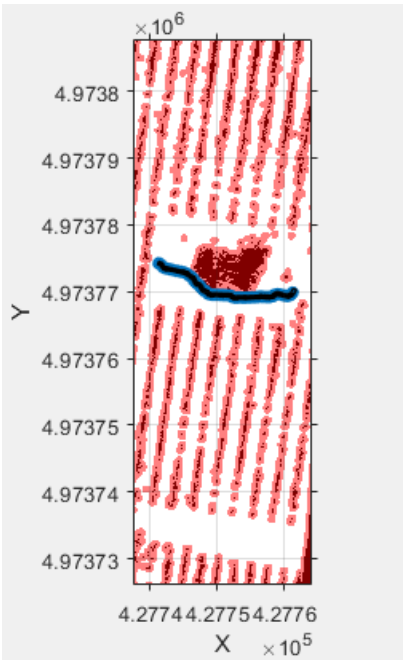
The first simulations were run to investigate if the parameters and the probability to find a path are somehow correlated. This analysis aims to adapt the basic algorithm to the type of map provded as good as well possible. For this very purpose, different functionalities of the code were blocked and very low values were assigned to the slope thresholds, to be sure that none of the paths that were found was actually valid.

In detail, the following were blocked: all functions regarding the various input and output text files, the possibility to update the maximum number of iterations, the possibility to update the initially entered thresholds and the section of the code regarding the interactive positioning on the map.

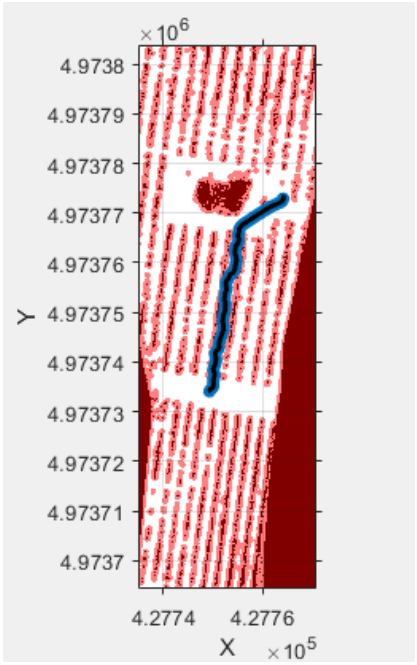
By doing so, using the same initial and final coordinates for each scenario a series of loop simulations were conducted, where the different parameters were modified and the results in terms of probability of success and computational time were recorded. When the above changes are applied, what happen is that, the RRT* algorithm searches for a possible path and as soon as it finds one, this shifts to the control phase, providing a negative response because of the deliberately too low thresholds, and then the algorithm searches for a new possible path through the RRT* again.

This procedure is repeated in loop for 24 minutes, a time that was arbitrarily chosen to obtain an accurate statistical evaluation of the process. The simulations stops after the 24 minute time and the results of the number of *attempt* (total number of attempts of the algorithm) and the number of *trial* (number of found paths that have been subjected to the threshold check) are recorded. The part of the code dealing with control was maintained to estimate the temporal impact that this part has on the whole code.

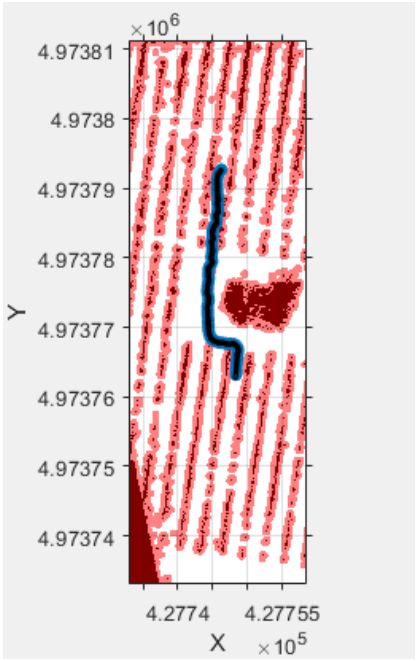
The three analysed scenarios are shown in Figure 3.12. The simplest case is presented in the first one, as it is clearly visible, a relatively short path (20 meters) with plenty of room for maneuvering and without any significant obstacle in between was chosed. In the second scenario, of intermediate difficulty, the distance of the path is doubled and the vehicle is forced to pass through one of many rows to reach the other side of the vineyard. In the third one, highly difficult, the vehicle starts its journey between two given rows crossing a large central section and then returning to a specific row and finish its journey in the middle of it.



(a) *Simple scenario*



(b) *Intermediate scenario*



(c) *Difficult scenario*

Figure 3.12: The three different scenarios that were used for the simulations of the first type.

The simulations that were conducted in these scenarios were made using incremental values of maximum iterations and two different settings. In the first one, the distance between one pose and the next was set to 0.2 meters with an inflation radius of the obstacles that is equal to what the MATLAB *vehicleCostmap* function automatically assigns, according to the size of the vehicle, which is 0.475 meters in this case; in the second one, the distance between one pose and the next was set to 0.3 meters and the inflation radius was reduced to 0.35 meters.

As it can be seen in Figure 3.13, 3.14 and 3.15 regarding the graphs on the probability of success, the most general phenomenon that all scenarios have in common is that the higher is the number of maximum iterations the higher the probability of finding a path. However, the point here is to understand how and how much the probability of success changes case by case, trying to contextualize and give a valid meaning or motivation to the results.

It is fair to remember that the goal is to have the highest probability of success of the RRT* with no detriment to the level of accuracy required for this application, so that even the found path is unlikely to meet the constraints, this probability is not further affected by the probability of finding a path in general. In addition to this, the use of the *DynamicIterations* and *maxtrial* parameters within the code as well as the presence of the variable *ObstacleIR* need to be addressed as well.

More in details, looking at the two different settings used, it is evident that if the same number of iterations is used, the increase in the distance between one pose and the next brings some benefits since, by increasing the distance between the nodes, the ramifications generated by the algorithm can grow further on the map, thus increasing the probability of reaching the target vertex.

When the overall picture and the nature of the basic algorithm are considered, it is clear that the excellent results of the second setting are not only due to the modification of the above mentioned parameter but, also and above all, are an effect of the reduction of the inflation radius of the obstacles.

The reduction of the inflation radius involves an enlargement of the Voronoi regions of the map, with the consequent increase of the probability of the algorithm to extend the ramifications also within the rows, particularly visible in the second and third scenarios. This phenomenon is due, as explained in subsection 1.2.4 of chapter 1, to the bias method used in the RRT algorithm that is based on the *k-nearest neighbors* method.

Vehicle safety is another factor to take into account, because the risk of a possible collision between the vehicle and obstacles also increases as the radius of inflaction is reduced. Therefore, the value of the parameter *ObstacleIR* must be set so to maintain the right trade-off between the increase of the probability of success of the algorithm and vehicle safety. An inflaction radius value of 0.4 meters was chosen for the final code, as explained in the previous section.

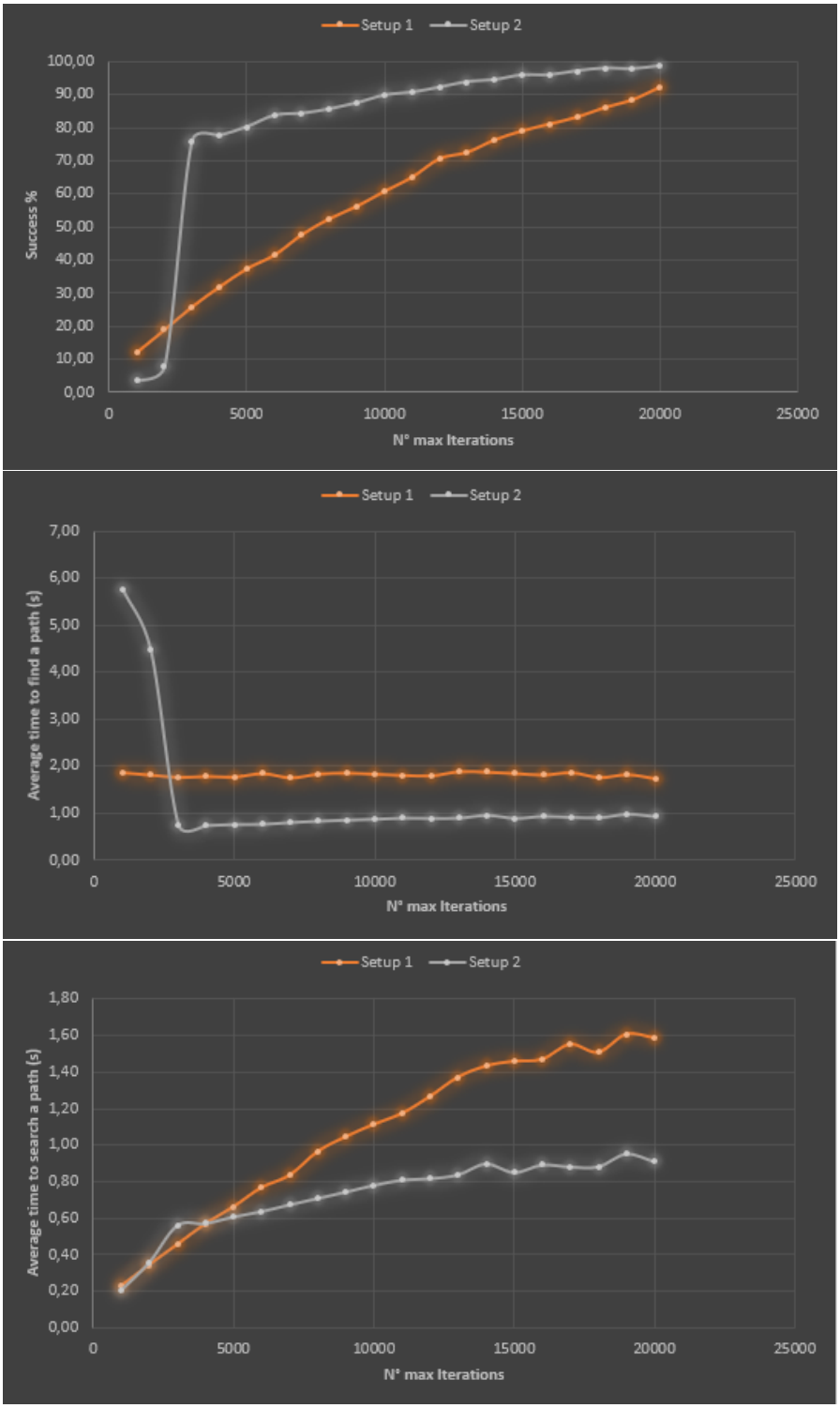


Figure 3.13: Main results obtained from simulations carried out for the simple scenario.

Figure 3.14: Main results obtained from simulations carried out for the intermediate scenario.

Figure 3.15: Main results obtained from simulations carried out for the difficult scenario.

The second graph, that is obtained, represents the average time it takes for the algorithm to find a path to validate, for each of the scenarios. The graph, gives a general idea of how long it takes for the algorith to find that kind of path. As it can be seen, the harded is the path to find, the longer the algorithm takes to generate a possible solution to be verified. The second setting leads to much better results than the first in each scenario here as well, especially in the third one where the algorithm can find a path in 5 seconds less on average.

The third graph is used to evaluate the impact of the number of iterations on the average calculation time, since most of the time spent by the algorithm is used to sample the vertices and connect them. The impact generally increases as well as the number of maximum iterations, because the algorithm can sample more vertices to try to find a valid connection with the objective vertex, as seen in the third scenario where the increasing trend is very similar for both settings.

This situation changes drastically when analysing the graphs regarding the first and the second scenarios. The slope of the curve for the second setting is different since the algorithm is much more likely to extend its ramifications up to the objective, before the number of samples reaches the maximum threshold, in the first two scenarios, i.e. beyond a cartain value of iterations the algorithm very often succeeds in reaching the objective, way before all the avaiable iterations are used.

The results clearly show that some of the dynamic structures used in the code, such as the one updating the number of maximum iterations after a certain number of cycles or the one used to selectively increase the maximum thresholds after a certain number of failed verifications, help reduce the time for the algorithm to provide a solution and to avoid that this gets stuck, unable to find a possible path, in some situations.

These simulations were also useful to identify an ideal number of ten thousand initial maximum iterations, since excellent success probability and computation time results were obtained with that number, both in the easy and in the intermediate scenarios.

In the second type of simulations the Mission Planner simulator is used, as previously stated, to accurately tune some parameters of the controller for the vehicle to follow the trajectory as precisely as possible.

This typology of simulations has demanded the use of 3 software in a row to elaborate the data correctly, seeing as, once the simulation on Mission Planner is carried out, the relative datalogs of the simulation, in *.BIN* format originally, need to be converted in a *.mat* file to be easily manipulated and analysed.

Once the key data of the simulated mission are obtained, the difference between the GPS data of the route taken by the vehicle and the ideal one represented by the waypoints that the algorithm provides as outputs need to be estimated. This

operation is processed through the *C2C* function of CloudCompare, a well-known open source software to manipulate and analyse point clouds, to evaluate the absolute distance between two point clouds in 3D or 2D. The adopted approach is similar to the one in [2] even if the results lead to a different parameters setting, given the different level of accuracy required by the type of application.

The procedure is explained below:

1. Once the file containing the waypoints from the algorithm are obtained, these are upload on Mission Planner and the type of vehicle is selected, in this case a rover, in the simulation section.

2. Some parameters are tuned in the configuration section called *Basic Tuning*, as shown in Figure 3.16 for the simulation to start.

3. The Datalogs of the mission are then extracted and converted in the *.mat* format.

4. The GPS data inside the Datalogs are converted, from geographical latitude and longitude coordinates into UTM, through the *deg2utm* function, by means of a specifically created Matlab code and stored on a text file.

5. The two text files, the one related to the waypoints provided by the algorithm and the one related to the GPS data, are uploaded on CloudCompare and the results are evaluated through the *C2C* function.

6. If the results obtained are not satisfactory, the parameters are better re-tuned going back to point 2.

In the first simulations, carried out with a quarter of the waypoints provided by the code, the focus was on tuning the avaiable parameters of the *Basic Tuning* section as well as possible. At the end of this first phase, when acceptable results are obtained, as reported in Table 3.10, an attempt to evaluate the behavior of the vehicle during simulations was done, by providing a higher and lower number of waypoints.

The results of this second phase are summarised in Table 3.11 and shown in Figures 3.17 3.18 3.19 3.20 3.21. When the number of waypoints through which the vehicle can better stick to the trajectory of the route was identified, control points to change the speed of the vehicle in and out of curves, named *DO_CHANGE_SPEED* by Mission Planner, were manually entered to further improve the overall vehicle performance, especially in tight curves.

Since the final goal of the project is to have the vehicle running along the desired trajectory as accurately as possible, the cruising speed was set at 1 m/s. The vehice, enter curves at a 0.5 m/s speed and then goes back to cruising speed once

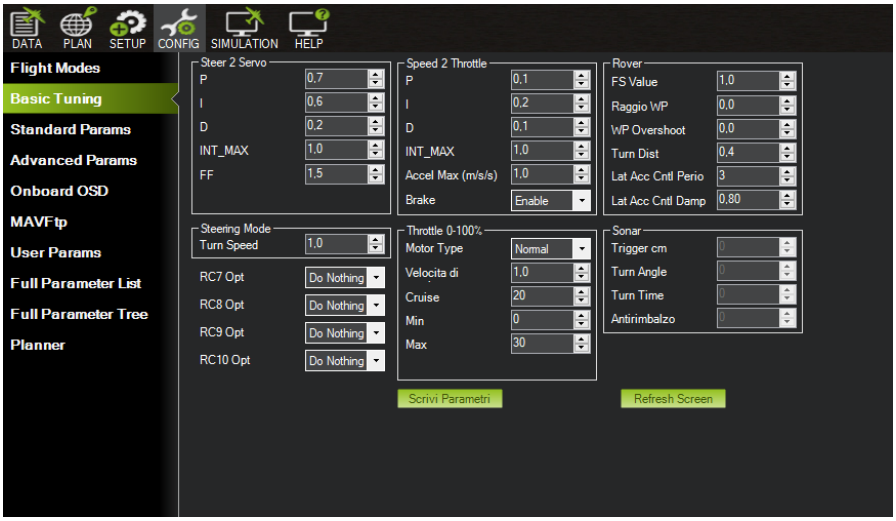


Figure 3.16: Basic Tuning section in Mission Planner software configuration settings.

out. The result of this further refinement leads to a real improvement especially in curves, as can be seen in Figure 3.22, getting a 0.109 meter average distance and a 0.068 meter standard deviation. The values thus obtained are considered excellent, also because further refinements would be irrelevant and not quantifiable in the real test. The results of these simulations were obtained using a simulated GPS accuracy of 0.3 meter in order to be conservative, compared to the GNSS system mounted on the vehicle that should have an accuracy level ranging from 0.1 to 0.2 meters.

Table 3.10: Final values assigned to the parameters for the simulation on the Mission Planner.

Section	**Parameters**	**Values**
Speed 2 Throttle	P (proportional gain)	0.1
	I (integral gain)	0.2
	D (derivative gain)	0.1
	INT_MAX	1
	Accel Max	1
	Brake	Enable
Throttle 0-100%	Motor Type	Normal
	Cruise Speed	1
	Cruise Throttle (%)	20
	Min (%)	0
	Max (%)	30
Steer 2 Servo	P (proportional gain)	0.7
	I (integral gain)	0.6
	D (derivative gain)	0.2
	INT_MAX	1
	FF (feed forward)	1.5
Rover	FS Value (Waypoint speed m/s)	1
	Raggio WP	0
	WP Overshoot	0
	Turn Dist	0.4
	Lat Acc Control Period	3
	Lat Acc Control Damping	0.8

Table 3.11: Values resulting from the calculation of the absolute distance between the ideal trajectory and the one resulting from simulations, through the *C2C* function of CloudCompare.

WP Ratio	**Mean Distance**	**Standard Deviation**
5	0.225 m	0.117 m
4	0.378 m	0.221 m
3	0.238 m	0.142 m
2	0.176 m	0.088 m
1	0.176 m	0.087 m
2-Refined	0.109 m	0.068 m

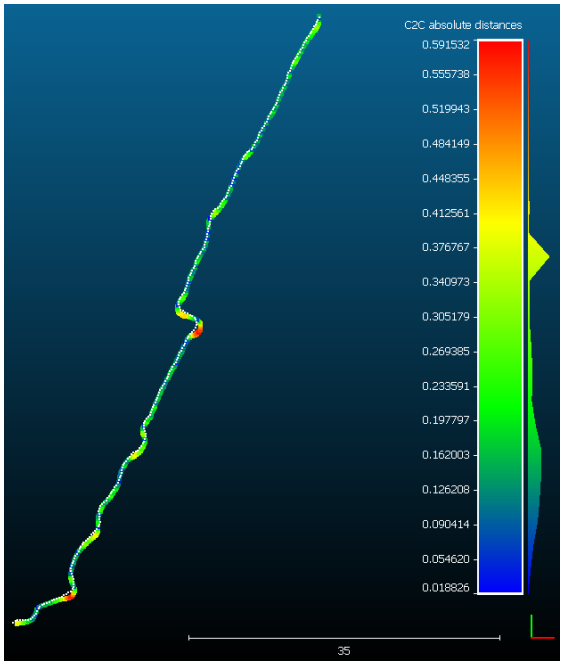


Figure 3.17: Result obtained using a ratio of one WP every five.

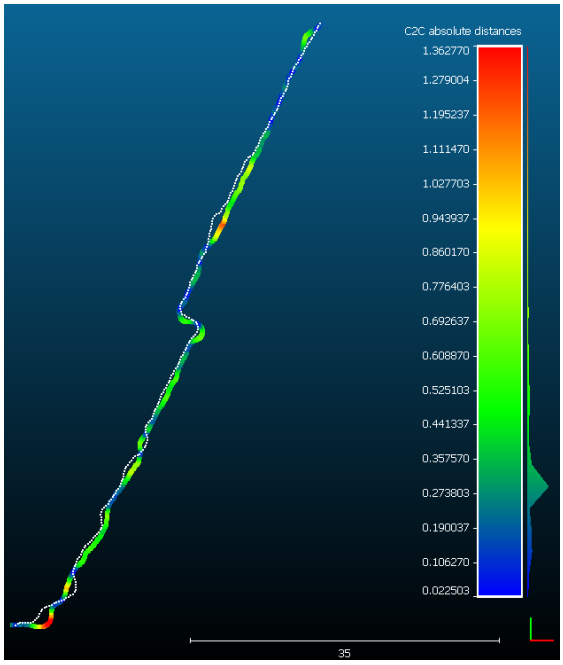


Figure 3.18: Result obtained using a ratio of one WP every four.

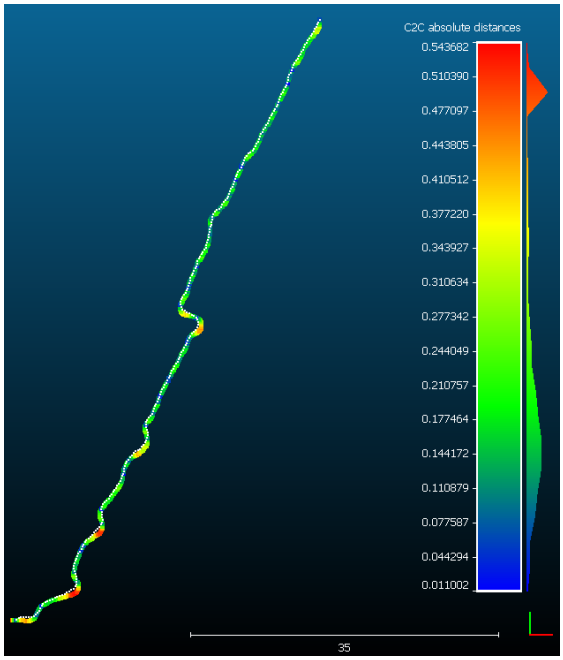


Figure 3.19: Result obtained using a ratio of one WP every three.

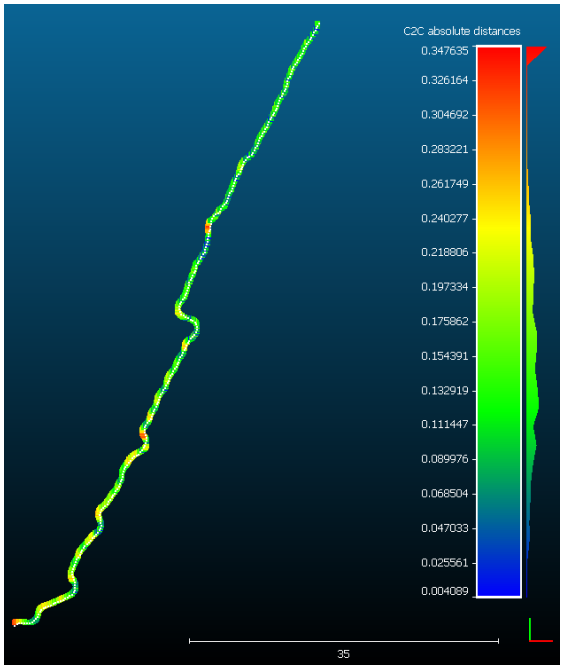


Figure 3.20: Result obtained using a ratio of one WP every two.

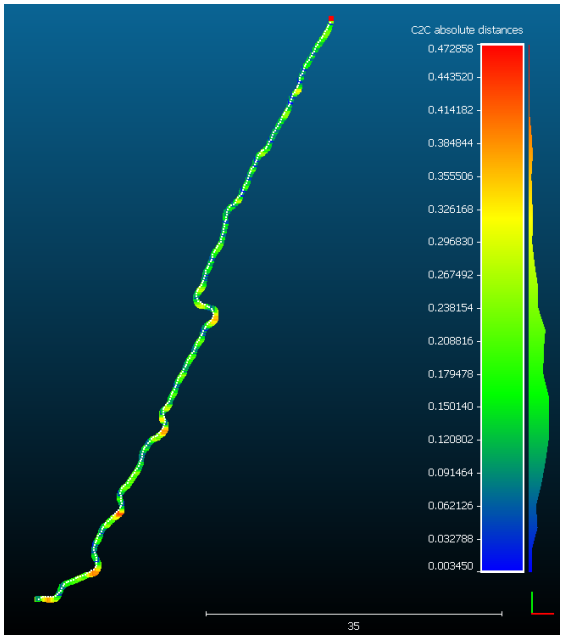


Figure 3.21: Result obtained using a WP ratio of one to one.

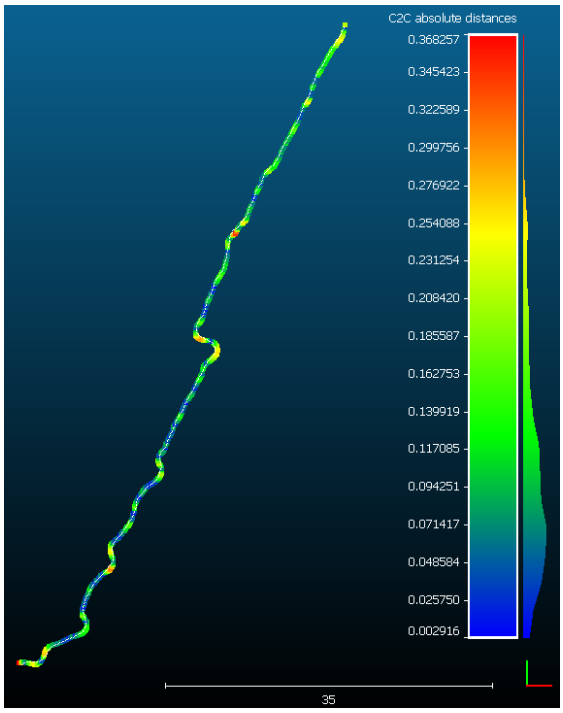


Figure 3.22: Result obtained using a ratio of one WP every two with the refining on the speed in and out of the curves.

Chapter 4

Validation

As mentioned in chapter 2 in the section 2.2, given the persistence of the covid-19 pandemic, the validation was carried out in a different site from the one initially proposed as a case study, at Tetti Neirotti airfield near Rivoli (TO).

The tests were performed with the instruments, configured on the hardware section as shown in chapter 2. In addition to this, a Tersus BX316 GNSS module and a total station were used to have different positioning references to compare with the obtained data.

The Tersus module was connected to the same antenna to which the Piksi module was connected through a splitter, while a prism was mounted on the same axis of the Piksi antenna, in this way the station hooks to the prism to trace the path, so that the reference provided by the station is as accurate as possible compared to the position that the antenna receives as shown in Figure 4.1.

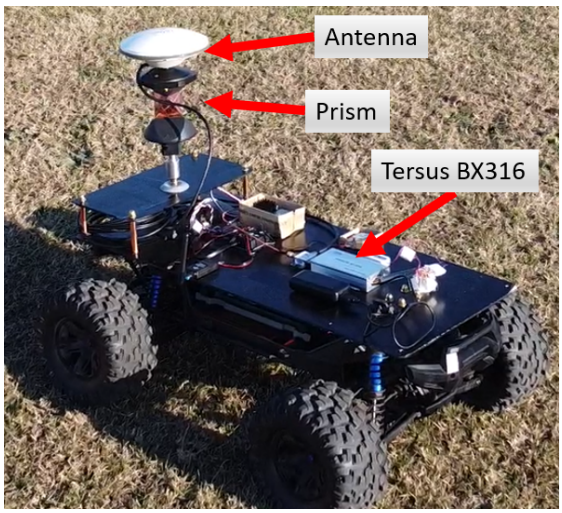


Figure 4.1: Vehicle configuration adopted for the validation.

Unfortunately, only few tests have been carried out, given the impossibility to continue in the absence of daylight in the only day when the tests were allowed. On the other hand, several errors and malfunctions of the tools, used as reference, only came to light once the data were post-processed and analyzed, forcing to consider valid only two out of three among the tests performed for the overall analysis.

The planned mission used in the valid tests is different from the one shown in the simulations and covers a greater distance, ideally passing through several rows.

Some considerations should be made about the basic problems of the vehicle, before analysing the results. One of the problems that became evident during the tests was represented by the fact that the vehicle does not navigate fluidly at low speeds (1 m/s), which forced to use a much higher speed value of 1.5 m/s to ensure a smooth driving.

Another problem with wheel camber was also noticed due to the fully discharged shock absorbers, which lead the vehicle to skid to the right during what should be a straight line. This phenomenon is visible from the comparison between the total station reference and the Pixhawk's Kalman filter solution during straights.

Figures 4.2 and 4.3 shows the speeds in m/s, with respect to North and East during Test 2 and 3 respectively, recorded by Pixhawk. As can be seen despite the speed value was increased, the vehicle was not able to maintain a constant speed even on straights. The peaks, on the other hand, correspond to the stretches of the route where the vehicle takes a curve.

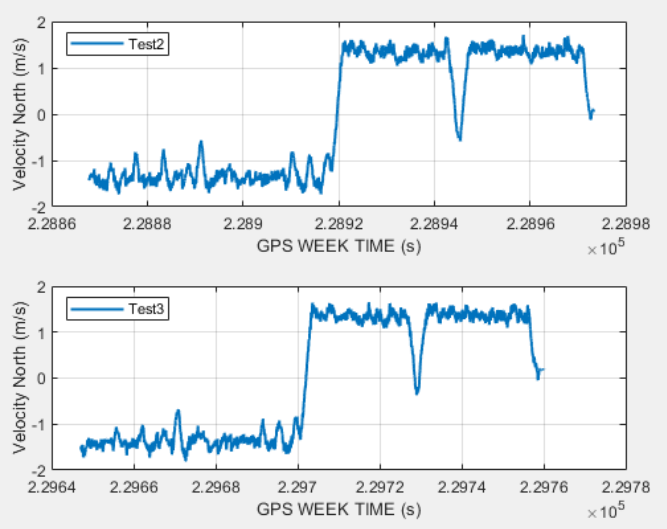


Figure 4.2: Velocity w.r.t. North in Test 2 and 3.

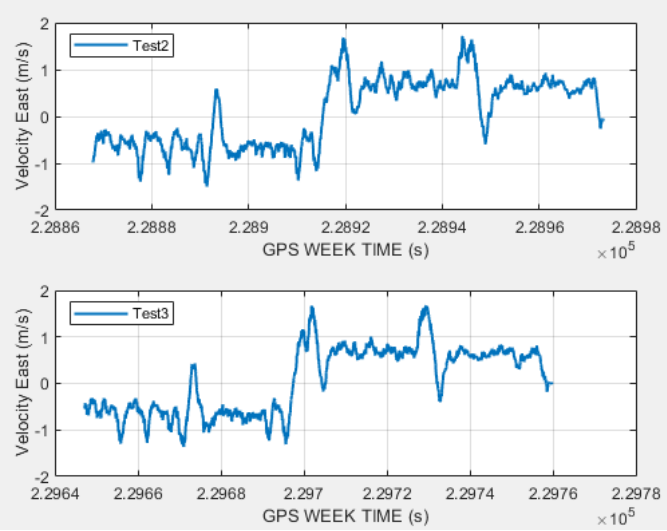


Figure 4.3: Velocity w.r.t. East in Test 2 and 3.

Figures 4.4, 4.5 and 4.6 show the roll, the pitch and the yaw recorded by Pixhawk during Test 2 and 3 respectively. Given the predominantly flat terrain, in which the tests were carried out, both the roll and the pitch values are very low, although not almost zero, as expected. This can be traced back to an unbalance of the vehicle due to the presence of several instruments mounted in the rear area of the vehicle, a factor that is further accentuated by the discharged shock absorbers.

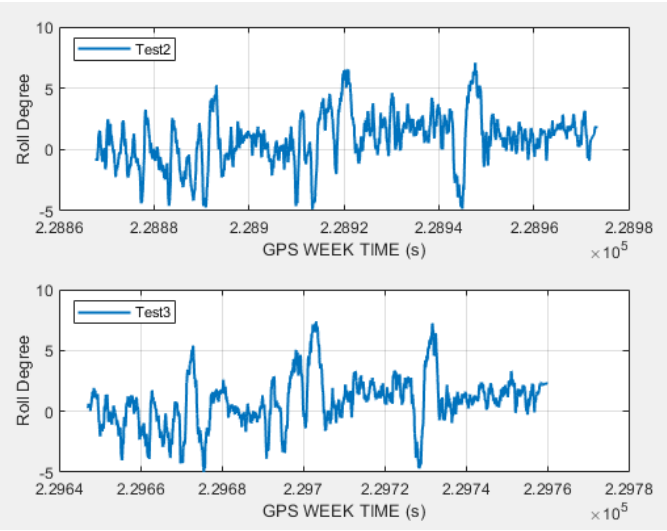


Figure 4.4: Roll Test 2 and 3.

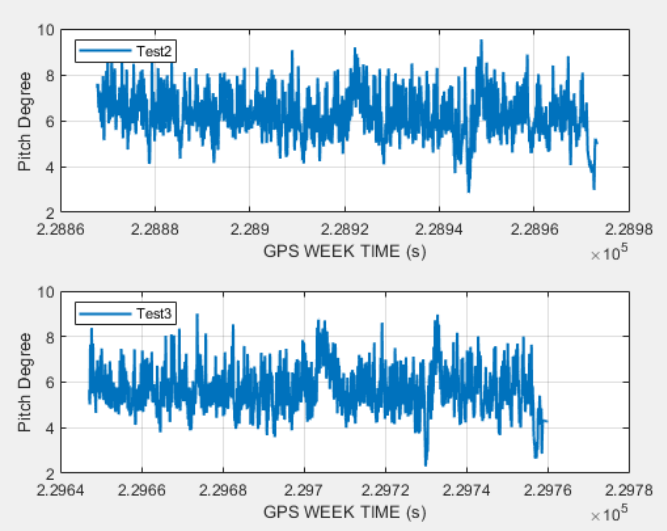


Figure 4.5: Pitch Test 2 and 3.

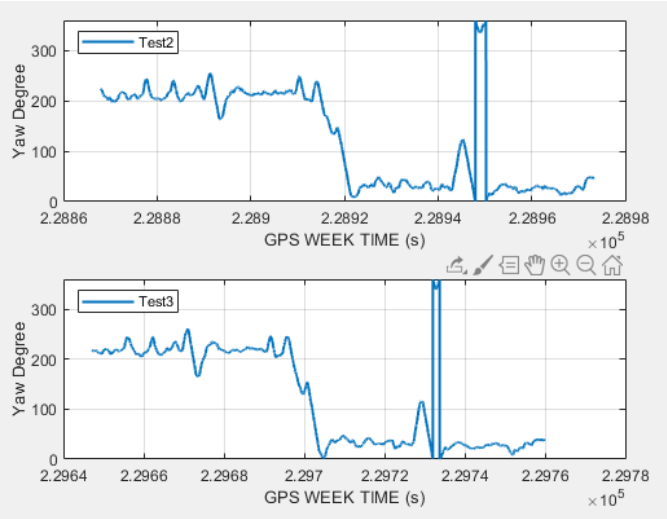


Figure 4.6: Yaw Test 2 and 3.

Some parameters of the controller, other than speed, have been changed compared to those discussed in the simulations section. The parameters used in the two tests are reported in Table 4.1.

The most important changes were made on the derivative and integrative part of the steer PID regulator and on the lateral acceleration control period. These factors heavily influenced the steering control of the vehicle during the first trials, producing a continuous zig-zag pattern even in the straights as the controller tried to continuously correct the trajectory.

Table 4.1: Basic Tuning parameters of the flight controller used for Test 2 and 3.

Section	**Parameters**	**Values**
Speed 2 Throttle	P (proportional gain)	0.1
	I (integral gain)	0.2
	D (derivative gain)	0
	INT_MAX	1
	Accel Max	1
	Brake	Enable
Throttle 0-100%	Motor Type	Normal
	Cruise Speed	1
	Cruise Throttle (%)	20
	Min (%)	0
	Max (%)	30
Steer 2 Servo	P (proportional gain)	0.2
	I (integral gain)	0.4
	D (derivative gain)	0
	INT_MAX	1
	FF (feed forward)	1.5
Rover	FS Value (Waypoint speed m/s)	1.5
	Raggio WP	0.1
	WP Overshoot	0.1
	Turn Dist	0.6
	Lat Acc Control Period	8
	Lat Acc Control Damping	0.8

The most important considerations on the results can be made with reference to Figures 4.7 and 4.10 showing three different trajectories plotted onto the binary mask: a real one regarding the vehicle position measured by the total station with millimetric accuracy, one calculated by the EKF of Pixhawk starting from the signals of the two GNSS (Piksi and UBlox) systems and an ideal one consisting of the Waypoints generated by the algorithm.

Since the results obtained by Tersus match almost perfectly with those provided by the total station but are recorded at a lower frequency, only the data of the total station will be used as reference in subsequent analysis.

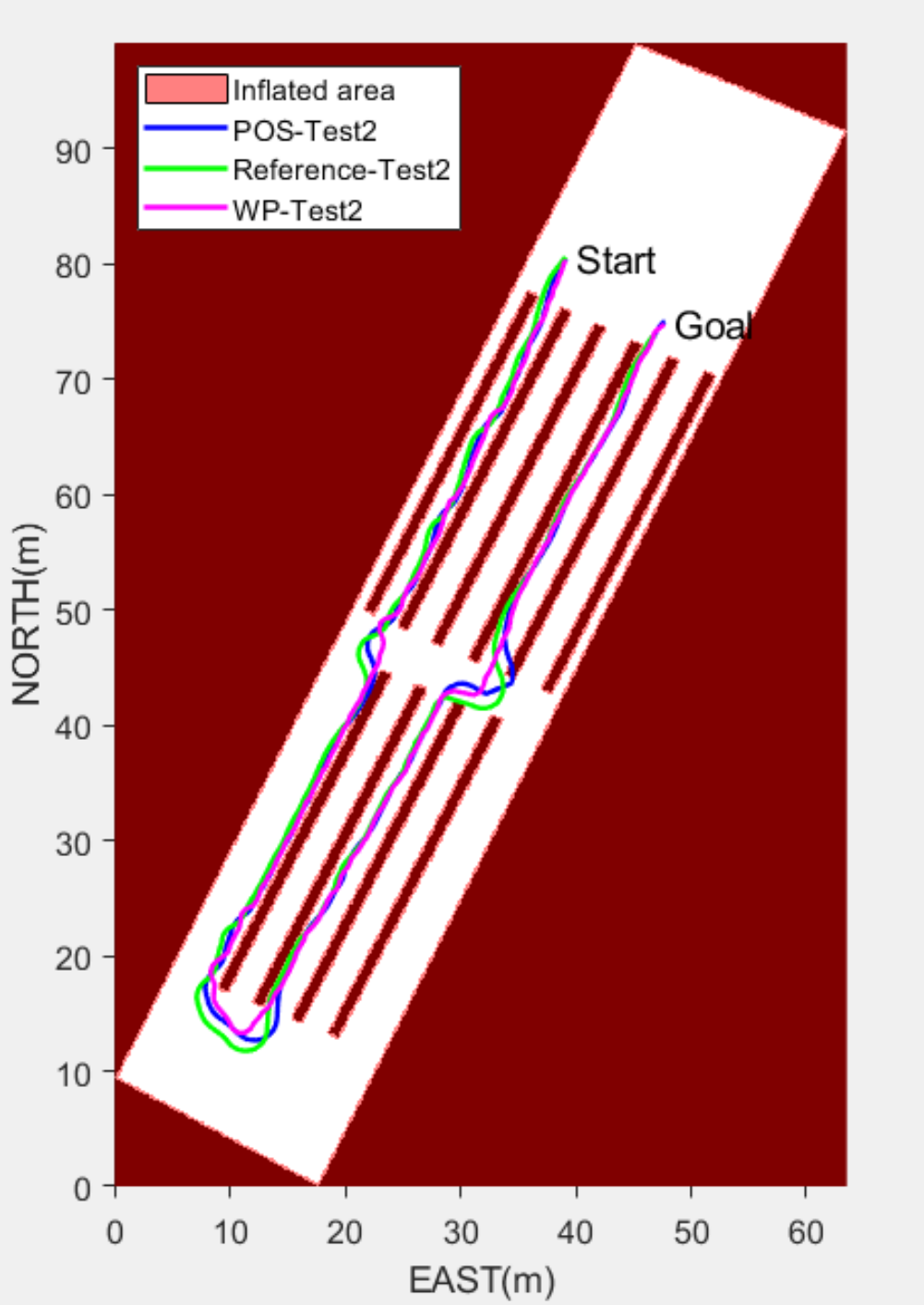


Figure 4.7: Test 2 Total Station, EKF and Waypoints trajectories plotted onto the binary mask with the presence of virtual rows.

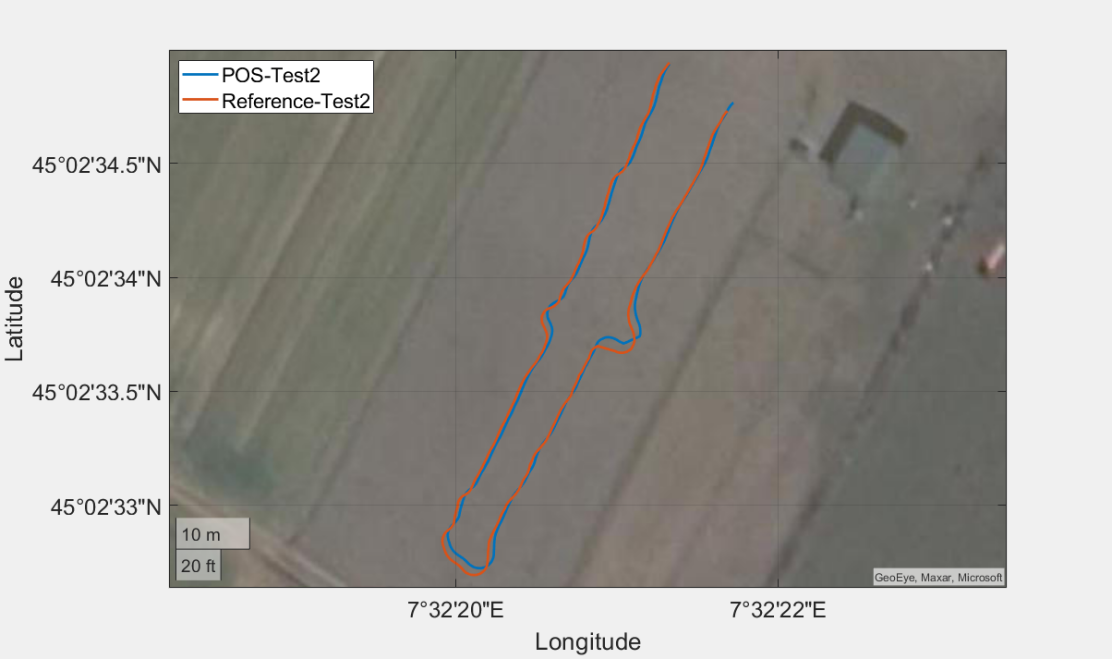


Figure 4.8: Geographic plot of the trajectories obtained in Test 2.

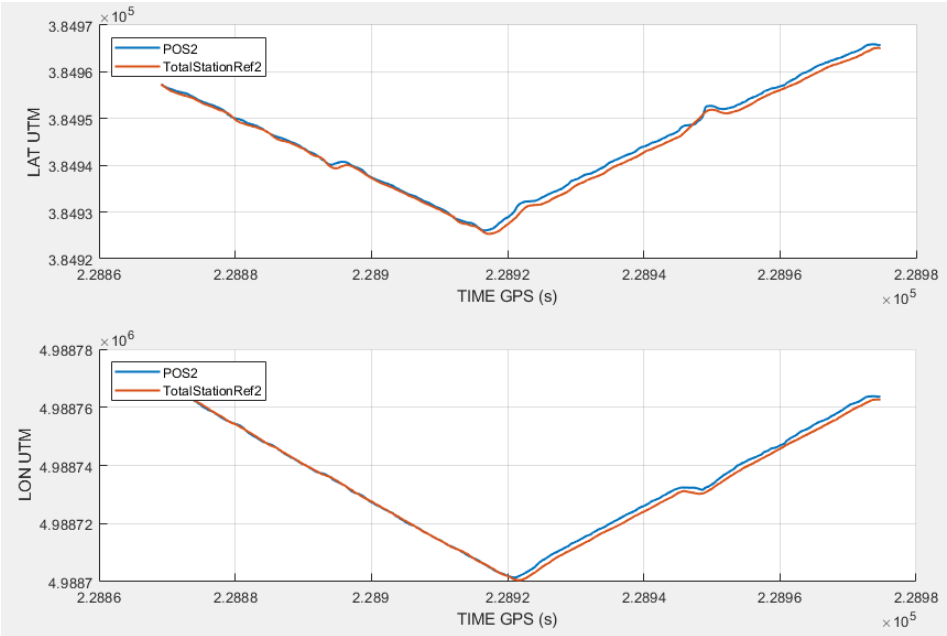


Figure 4.9: Test 2 displacements with respect to Latitude and Longitude in UTM coordinates.

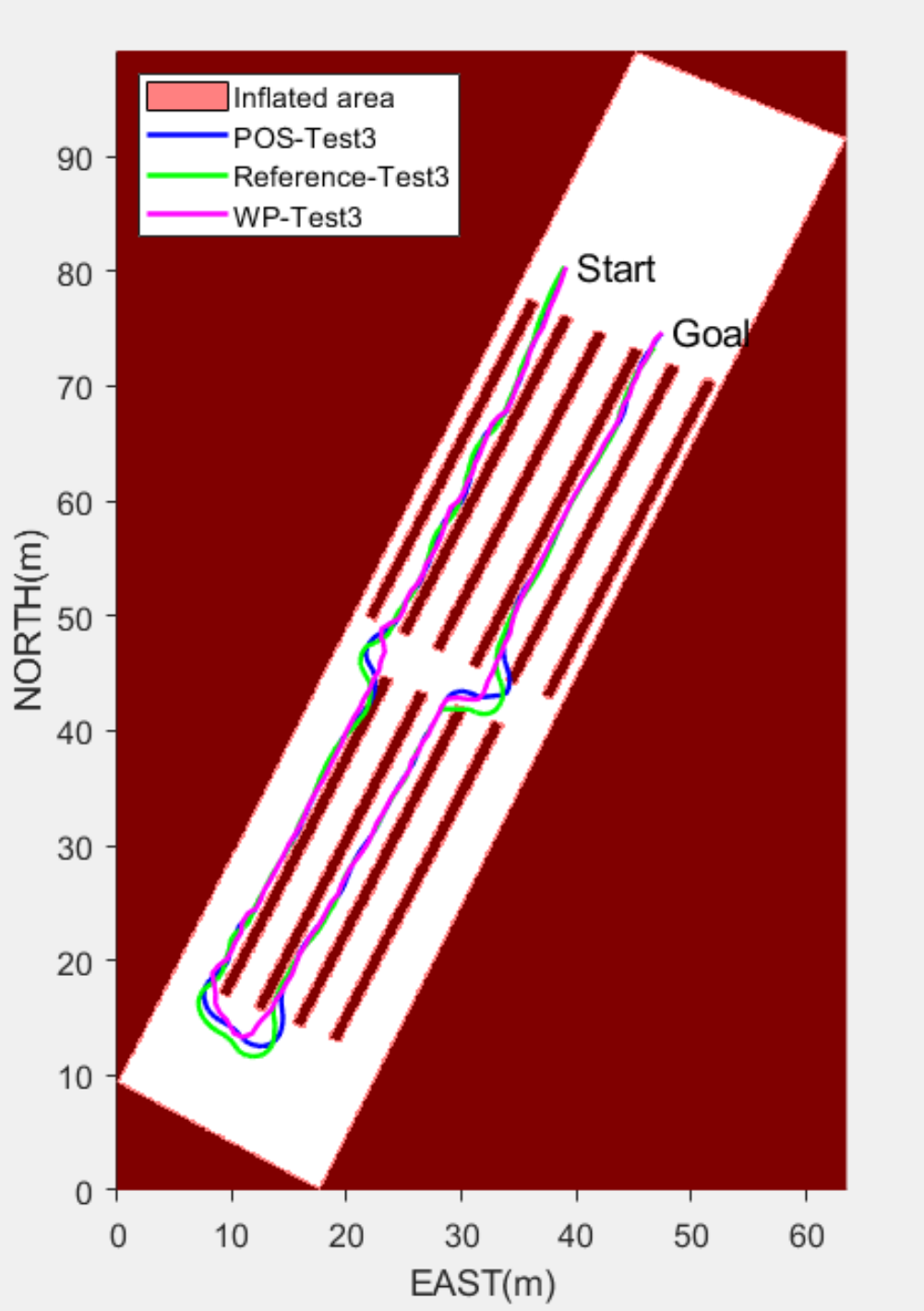


Figure 4.10: Test 3 Total Station, EKF and Waypoints trajectories plotted onto the binary mask with the presence of virtual rows.

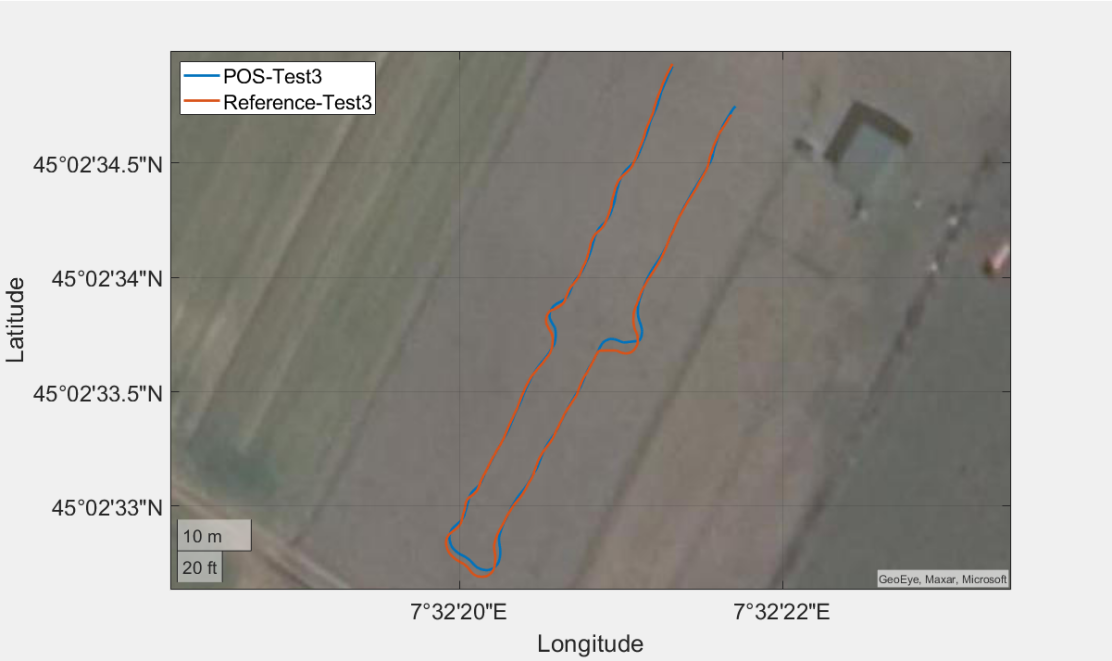


Figure 4.11: Geographic plot of the trajectories obtained in Test 3.

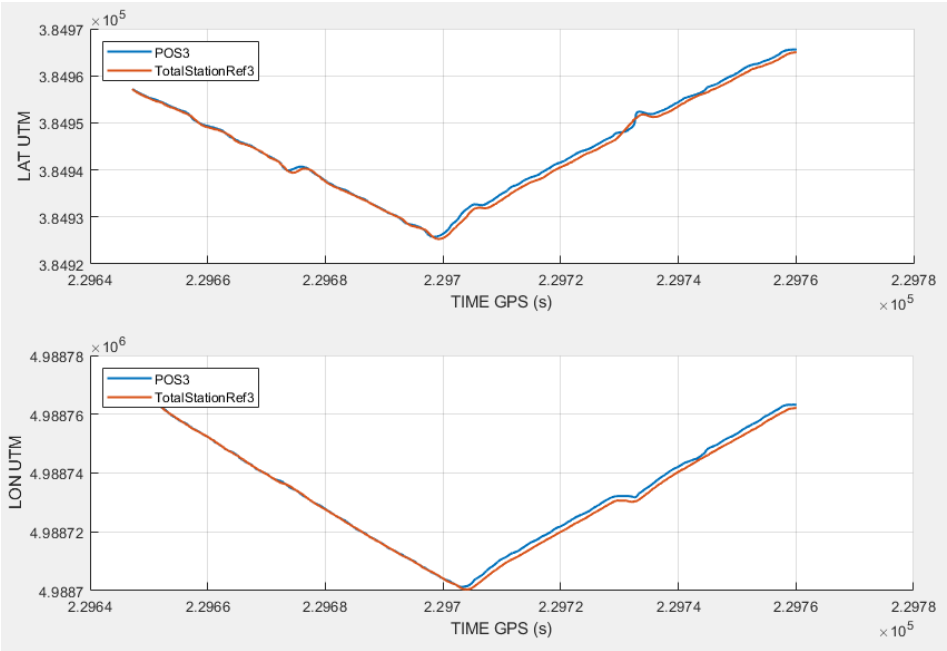


Figure 4.12: Test 3 displacements with respect to Latitude and Longitude in UTM coordinates.

The difference between the trajectories is minimal along straights, as it can be seen, and falls within the necessary values to be obtained for this type of application. It can be stated that this difference is mostly caused by the vehicle problems that were described above.

Something totally different happens if the behavior of the vehicle is observed in curves, where the difference between trajectories is relevant. The cause of this behavior during curves had already been highlighted during simulations and is due to the vehicle speed that results unsuitable for precision maneuvers. Plus, these errors are further accentuated if we consider the inherent problems of the vehicle. The vehicle does not immediately realign with the waypoints track since the gains of the proportional and integral part of the steering PID regulator were reduced.

The comparisons between trajectories were made through Cloud Compare using the C2C function, as in the simulations. The results in terms of mean distance and standard deviation are shown in Tables 4.2 and 4.3 for test 2 and test 3 respectively and refer to Figures 4.13, 4.14, 4.15, 4.16, 4.17 and 4.18. The bar laterally present in the plots represents the normal distribution of the absolute distance between points of the compared trajectories. The two tests were performed using the same mission and the same controller parameters, the only difference is the number of waypoints. In test 2, half of the waypoints found by the algorithm were used, while in test 3 a third of them was used.

Unlike what the simulations showed, it seems that a better solution can be achieved with a smaller number of waypoints, even if the improper speed of the vehicle may have affected this comparison.

Even if the results are not the expected ones, they are considered quite good, since it was impossible to try further setups and therefore accurately tune the parameters of the flight controller.

Table 4.2: Test 2 trajectories error evaluation.

Compared cloud	**Mean Distance**	**Standard Deviation**
Total Station VS EKF POS	0.409 m	0.317 m
Ideal WP VS Total Station	0.464 m	0.421 m
Ideal WP VS EKF POS	0.268 m	0.313 m

Table 4.3: Test 3 trajectories error evaluation.

Compared cloud	**Mean Distance**	**Standard Deviation**
Total Station VS EKF POS	0.294 m	0.301 m
Ideal WP VS Total Station	0.384 m	0.451 m
Ideal WP VS EKF POS	0.283 m	0.347 m

Table 4.4: Results obtained for straight trajectories (best case) in the third test using the same comparison process already seen using CloudCompare.

Compared cloud	**Mean Distance**	**Standard Deviation**
Total Station VS EKF POS	0.144 m	0.055 m
Ideal WP VS Total Station	0.140 m	0.100 m
Ideal WP VS EKF POS	0.141 m	0.060 m

Table 4.5: Results obtained for curves (worst case) in the third test using the same comparison process already seen using CloudCompare.

Compared cloud	**Mean Distance**	**Standard Deviation**
Total Station VS EKF POS	0.606 m	0.495 m
Ideal WP VS Total Station	0.913 m	0.631 m
Ideal WP VS EKF POS	0.624 m	0.558 m

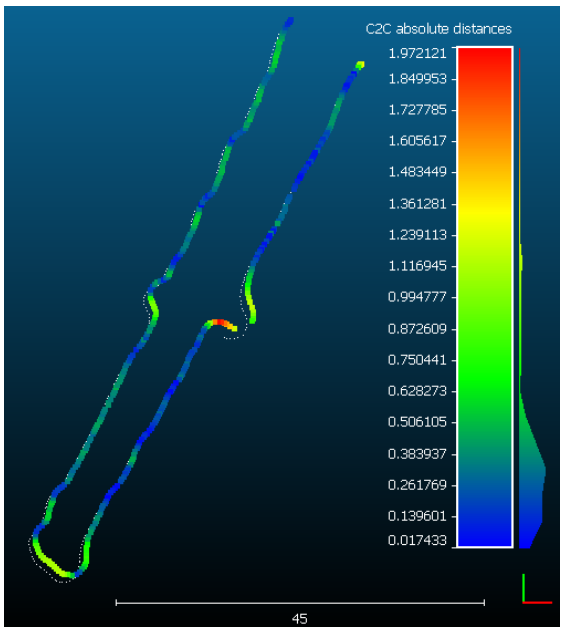


Figure 4.13: Total Station VS EKF POS Test2.

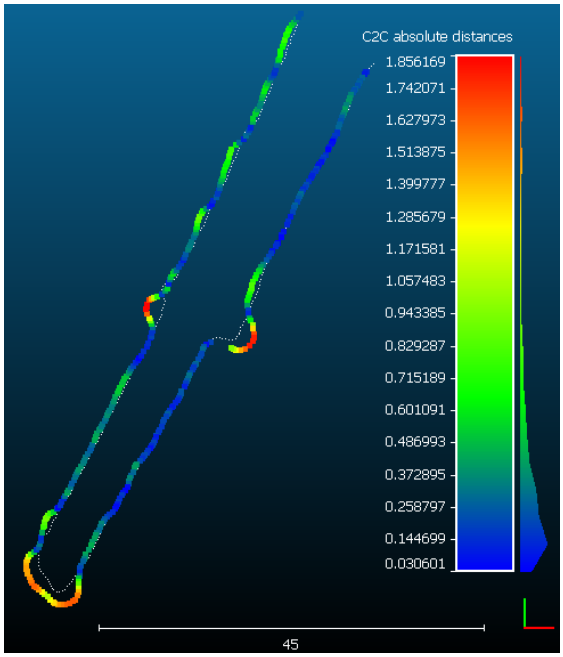


Figure 4.14: Ideal WP VS Total Station Test2.

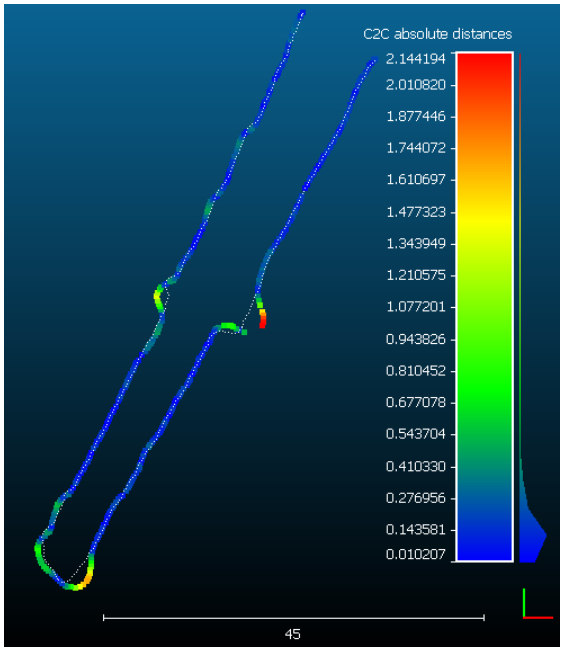


Figure 4.15: Ideal WP VS EKF POS Test2.

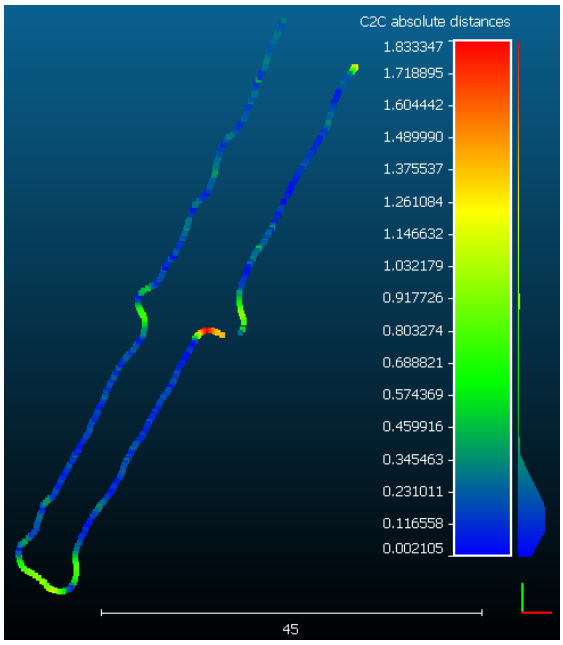


Figure 4.16: Total Station VS EKF POS Test3.

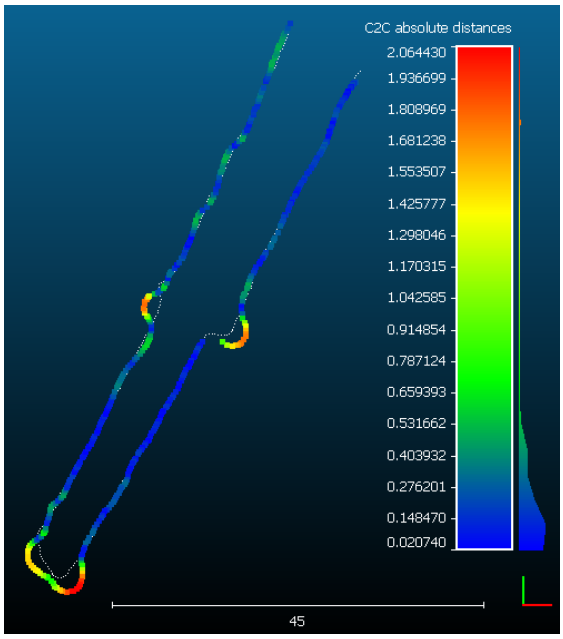


Figure 4.17: Ideal WP VS Total Station Test3.

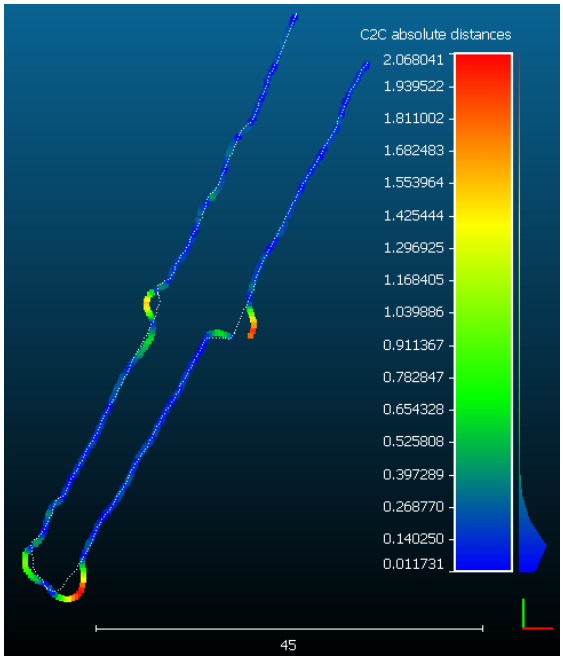


Figure 4.18: Ideal WP VS EKF POS Test3.

Chapter 5

Conclusions

This thesis work aims to realize a vehicle that can navigate in uneven terrain independently, through the use of different hardware and software technologies and tools. The main contribution to achieve this goal was the realization of a code in the MATLAB development environment, which would be able to generate a safe and efficient route with respect to different parameters such as distance, slope and its variation, to have a precision navigation route through very hard environments, like a vineyard.

The RRT* planning algorithm that was used and the superstructure that was created all around it, to adapt the algorithm to the purpose and the problems of this type of terrain, works quite well, thanks to the collaboration of an UAV that, through an air-photogrammetric survey, provides the algorithm with high resolution static maps with centimeter accuracy.

The RRT* has a hard time to find a path in the first type of simulations carried out on MATLAB, especially when the vehicle has to pass between one row and another, and this is because of the very narrow space that can be explored in those stretches, as already discussed, which is further decreased by the inflation radius applied by the *vehicleCostmap* function.

Therefore, to have a higher chance to find a possible path and therefore drastically decrease the time to search for a path, the value of inflation radius of the obstacles should be lower than that calculated as a function of the dimensions of the vehicle. Despite a significant error between the expected trajectory and the real one, using a 0.35 meters obstacle inflation radius, the vehicle touched the ideal rows only once over 170 meters of navigation. As already recommended in chapter 3, an optimal value to balance safety and the probability of finding a path could be set on 0.4 meters, in our case.

Some precision navigation limits were highlighted during the simulations carried out on Mission Planner, due to the speed that the vehicle is forced to maintain in order to accurately follow the waypoints provided by the code. These limits have

become significant during the field validation phase, highlighting also some differences in the behavior of the vehicle compared to the controller parameters set during the simulation phase. As shown in the previous chapter, the difference between simulations and the tests on the field is especially remarkable in curves, even if most errors are determined by the vehicle. It should be also pointed out that the presented results were obtained in stand-alone mode, therefore without the use of the RTK differential technique which, if used, should lead to a significant improvement in positioning with respect to the assigned path.

Despite the many advantages of the Traxxas X-Maxx, including the large space that was made on the vehicle to mount the different sensors and the convenient size of the track and wheelbase, this UGV used has proved to be unsuitable for this application, mainly because of the difficulties encountered when navigation at low speeds.

The high precision Piksi GNSS module and the antenna, part of the kit that was used, turned out to be quite good when compared to the solution given by the reference obtained via the total station, while the same cannot be said for the Ublox GNSS module used as a secondary GPS.

It is then clear that further field tests, a finer tuning of the controller parameters and a vehicle maintaining a stable driving at a 1 m/s speed, as planned during the simulations, would have led to better results.

5.1 Possible future improvements

There are several possible improvements to make, given the complexity of the problem.

The implementation of a stereo camera or other sensors preventing the vehicle from hitting the vineyards or other dynamic obstacles not provided in the static maps in case it goes off course, was deemed as necessary for safety reasons. This system could further improve the quality and accuracy of the navigation through further visual corrections in addition to those provided by the sensors already on board.

Another important point is to identify a vehicle that better suits the need for driving at low speeds, perhaps equipped with tracks instead of wheels. The advantage of the track over the wheel is given by the greater footprint on the ground, which allows to operate on terrain with poor bearing capacity, on which a wheeled vehicle would not be able to move (sand, mud, snow), thus helping the vehicle overcome particularly challenging bumps.

In general, tracked vehicles have a higher mobility than vehicles equipped with tires on rough terrain, in fact these vehicles reduce jolts by slipping on small obstacles. Finally, tracked vehicles can rotate with smaller radii of curvature, since

it is possible to rotate with a turning radius as wide as the vehicle,by blocking a track, and rotating while remaining in place is made possible by using a track in reverse.

This choice may lead to small changes in the algorithm, with regards to the evaluation of the support surface and the consequent calculation of the slopes, but it could significantly affect the real navigation.

Finally, since the algorithm works by randomly exploring the free configuration space in a sort of trial and error version, a new external superstructure based on neural networks or deep learning may be created. Therefore, starting from the solutions provided by the algorithm in a given map, after an initial phase of training, it can predict what could be the best path once the desired starting and ending point are known.

Acronyms

AGV Automated Guided Vehicle

APF Artificial Potential Field

BEC Battery Eliminator Circuit

BFS Breadth-First Search

BVP Boundary Value Problem

CPC Carrier phase Compensation

DFS Depth-First Search

DSM Digital Surface Model

DTM Digital Terrain Model

EKF Extended Kalman Filter

FPV First Person View

GCS Ground Control Station

GIS Geographic Information System

GNSS Global Navigation Satellite System

GPS Global Positioning System

GPU Graphics Processing Unit

IDE Integrated Development Environment

IMU Inertial Measurement Unit

IoT Internet of Things

ITU International Telecommunication Union

MDP Markov Decision Process

MEMS Micro Electro-Mechanical Systems

PID Proportional Integrative Derivative

PRC Pseudo Range Carrier

PRM Probabilistic roadmap

RRT Rapidly-exploring Random Tree

RTK Real Time Kinematic

SITL Software In The Loop

SPB Swift Binary Protocol

UAV Unmanned Aerial Vehicle

UGV Unmanned Ground Vehicle

UTM Universal Transverse of Mercator

V2X Vehicle-to-everything

WP WayPoint

List of Figures

List of Tables

Bibliography

[1] Jérôme Barraquand, Lydia Kavraki, Jean-Claude Latombe, Rajeev Motwani, Tsai-Yen Li, and Prabhakar Raghavan. A random sampling scheme for path planning. *The International Journal of Robotics Research*, 16(6):759–774, 1997.

[2] Federico Faedda. *Veicoli autonomi terrestri per il rilievo automatico speditivo in aree critiche= Uncrewed Ground Vehicles for expeditious survey in critical areas*. PhD thesis, Politecnico di Torino, 2020.

[3] Mahdi Fakoor, Amirreza Kosari, and Mohsen Jafarzadeh. Humanoid robot path planning with fuzzy markov decision processes. *Journal of applied research and technology*, 14(5):300–310, 2016.

[4] Javier Gonzalez-Jimenez, J Ruiz-Sarmiento, and Cipriano Galindo. Improving 2d reactive navigators with kinect. In *10th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*, 2013.

[5] Sertac Karaman and Emilio Frazzoli. Incremental sampling-based algorithms for optimal motion planning. *Robotics Science and Systems VI*, 104(2), 2010.

[6] Sertac Karaman and Emilio Frazzoli. Optimal kinodynamic motion planning using incremental sampling-based methods. In *49th IEEE conference on decision and control (CDC)*, pages 7681–7687. IEEE, 2010.

[7] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[8] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[9] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[10] Gh Lazea and As E Lupu. Aspects on path planning for mobile robots. *TEMPUS M-JEP 11467: Intensive Course on Computer Aided Engineering in Flexible Manufacturing*, pages 19–23, 1996.

[11] Michael W Otte. A survey of machine learning approaches to robotic path-planning. *University of Colorado at Boulder, Boulder*, 2015.

[12] EN Sabudin, Rosli Omar, and CKANH Che Ku Melor. Potential field methods and their inherent approaches for path planning. *ARPN Journal of Engineering and Applied Sciences*, 11(18):10801–10805, 2016.

[13] Thomas B Sheridan. *Telerobotics, automation, and human supervisory control*. MIT press, 1992.

[14] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, and Giuseppe Oriolo. *Robotics: modelling, planning and control*. Springer Science & Business Media, 2010.

[15] Jurgen Zoto, Maria Angela Musci, Aleem Khaliq, Marcello Chiaberge, and Irene Aicardi. Automatic path planning for unmanned ground vehicle using uav imagery. In *International Conference on Robotics in Alpe-Adria Danube Region*, pages 223–230. Springer, 2019.