UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

MASTER THESIS

Structured Flight Plan Interpreter for Drones in AirSim

Francesco Rose

SUPERVISED BY

Cristina Barrado Muxi

Universitat Politècnica de Catalunya
Master in Aerospace Science & Technology
January 2020

This Page Intentionally Left Blank

Structured Flight Plan Interpreter for Drones in AirSim

BY

Francesco Rose

DIPLOMA THESIS FOR DEGREE

Master in Aerospace Science and Technology

AT

Universitat Politècnica de Catalunya

SUPERVISED BY:

Cristina Barrado Muxi

Computer Architecture Department

"E come i gru van cantando lor lai,
faccendo in aere di sé lunga riga,
così vid'io venir, traendo guai,
ombre portate da la detta briga"

Dante, Inferno, V Canto

This Page Intentionally Left Blank

ABSTRACT

Nowadays, several Flight Plans for drones are planned and managed taking advantages of Extensible Markup Language (XML). In the mean time, to test drones performances as well as their behavior, simulators usefulness has been increasingly growing. Hence, what it takes to make a simulator capable of receiving commands from an XML file is a dynamic interface.

The main objectives of this master thesis are basically three. First of all, the handwriting of an XML flight plan (FP) compatible with the simulator environment chosen. Then, the creation of a dynamic interface that can read whatever XML FP and that will transmit commands to the drone. Finally, using the simulator, it will be possible to test both interface and flight plan.

Moreover, a dynamic interface aimed at managing two or more drones in parallel has been built and implemented as extra objective of this master thesis. In addition, assuming that two drones will be used to test this interface, it is required the handwriting of two more FPs.

In order to achieve all the goals of this project, it has been chosen AirSim as drone-simulator and Python as programming-language for the development of the dynamic interfaces. Python and AirSim can “talk” to each other thanks to the really good list of APIs (Application Programming Interface) provided by the AirSim library for Python.

On the other hand, to write the XML FPs, I took advantages of the RAISE+ documentation (simulator for fixed and rotary wing aircrafts) for building a flight plan (see [10]). I implemented a total of six FPs: two FPs to test the interface for the single drone and four FPs to test the multiple-drones interface (two FPs for each drone). Each pair of FPs has the same path; one uses Geographical coordinates (latitude, longitude, altitude), the other one uses AirSim’s NED coordinates (north, east, down). Since take off and landing are obtained through two Python APIs for AirSim, the flight plan will concern only the mission waypoints.

In the end, I obtained two dynamic interfaces with a high degree of independence from any XML flight plan and AirSim environment chosen. The only requirement is that the FP waypoints have to be compatible with the simulator environment. Moreover, the FP has been created involving four out of all the possible legs that describe drone maneuvers and it has been planned for the Neighborhood AirSim environment. All the limitations will be further discussed in the “Recommendations” section (6.3).

All the topics will be deeply analyzed and successively explained along the master thesis, highlighting the most important features and the problem-solving methodology carried on during the whole project.

This Page Intentionally Left Blank

ACKNOWLEDGEMENTS

I am deeply grateful to my master thesis supervisor Professor Cristina Barrado Muxi for providing me with professional guidance and clear directive during the development of the project.

I am greatly indebted to my flatmate Riccardo Salis and to my collegue Thomas Fili. Their moral and work support is incomparable; I will always bring this great experience with me.

Special appreciation to PhD Ender Çetin for providing me with assistance and support during the development of the project.

I would like to thank my parents and my whole family for supporting me during the master course as well as during all my life. I know I would never reach this goal without them. I owe everything I am to my family.

I am deeply grateful to my lovely girlfriend Cristiana. Her support and patience as well as her comprehension played a key role during the entire master course. I love you.

I would like to thank my Master coordinator Professor Ricard Gonzalez Cinca for providing advice and support during the whole master course. Last but not least, many thanks to the UPC University for all the resources made available.

This Page Intentionally Left Blank

Table of Contents

List of Figures

List of tables

INTRODUCTION

Drones are becoming more and more an active part of our life. This technology is useful for a wide range of application field like panoramic scan, monitoring and surveillance, low-altitude photograph as well as cargo delivery, agriculture, etc.

On the other hand, several issues are directly linked to the drone world. In the “Area for further studies” section (6.2) I will present all these issues that have to be taken into account to fly a real drone in a populated area. The first problem that has to be analyzed is the respect for privacy. Then, it has to be considered that this technology is “easy” to be used: thieves, terrorists, arsonists and thousands of other evil-minded people will badly take advantages of drones to reach their purpouse. Hence, the need for “police-drones” that will protect us catching the “evil-drones”.

Currently, humans pilot most of the drones and they introduce all the instructions using telecommands or drone ground stations. Moreover, automated drones are already on market but their autopilot allows only easy movements. The next step is to implement an autopilot that allows more difficult maneuvers.

This master thesis will be focused on the building of two dynamic interfaces that will allow the final customer to upload whatever “Extensible-Markup-Language Flight-Plan” (XML FP) on drones in the simulator enviornment. Then, waypoints informations will be translated into commands for the drones in order to follow the required path. After that, drones will be able to follow the flight plan without any external help to succeed take-off, mission and landing.

In order to test both my interfaces and FPs, I created a use case: Neighborhood-Surveillance-Mission (NSM) is aimed to successfully monitor the chosen AirSim Enviornment and detect possible thieves. With respect to the single-drone interface, my drone *Carlo* has been commissioned to monitor the Neighborhood through three different scanning paths. *Paolo* e *Francesca* will carry out the same mission, but they will be operated by the multiple-drones interface. Later on in the master thesis, my FPs and the strategies I adopted to reach the objectives of this work will be deeply treated.

Below, the structure of the doc is presented. Chapter 1 points out the metodology followed during the development of the project. Chapters 2 and 3 deal with two of the tools used in this master thesis (XML and AirSim), giving a quick look at both their general features and history. Moreover, it is explained how and when I used them, underlining the important aspects that have to be taken into account. In Chapter 4, Python, its history and its usage are deeply analyzed. I will clarify how I built my scripts, explaining all the functions and the several parts forming the codes. Chapter 5 contains all the significant plots and data obtained running the single-drone interface. In the last section of this chapter the results are presented. In Chapter 6 I draw the main conclusions of the whole project, treating the issues related with the drone world and the recommendations to be followed to correctly use my dynamic interfaces. Chapter 7 is the bibliography. In Chapter 8 all the images of the six XML FPs and of the two Python codes are presented.

Chapter 1

GENERAL DEVELOPMENT OF THE MASTER THESIS

The goal of this chapter is to put in evidence the procedure followed in order to achieve the objectives of the master thesis. To reach all the requirements of this project, I took an inverse-engineering approach as well as a problem-solving metodology.

1.1. Step 1: XML FPs implementation for Single-Drone Interface

The first goal of this work is the creation of an XML file containing a Flight Plan for *Carlo* inside the AirSim's Neighborhood environment. Hence, the selection of compatible waypoints plays a key role for the correct development of the mission. Phyton-AirSim's APIs allow us to move the drone around the map while getting information on its position (both Geographical and NED coordinates). After that, I built the FPs according to the RAISE+ documentation (see [10]) taking advantages of the chosen points. In the end, I obtained two flight plans with the same path but with different coordinates, one with Geographical and one with NED (North, East, Down). Since AirSim simulator takes only NED coordinates as input, a conversion of the Geographical coordinates is also required. In sub-section 2.5.1, all the information related to these XML Flight Plans and the strategy I adopted will be shown; the conversion parameters are in section 3.3. Images of both flight plans are in the "Annexes" (Chapter 8).

1.2. Step 2: Python Interface implementation for Signle Drone

In order to accomplish the second objective, I took advantage of Python as programming language. First of all, I searched how parse an XML file on Python and how reach and store all the data encoded inside. Then, I created functions both to represent the different legs of the flight plan and to execute it in the correct order. I also implemented lines of code to get different significant plots and to store position as well as time data. The dynamic interface has some limits that will be treated in the "Recomendations" section (6.3), but overall it could be said that the interface does not depend on anything, it automatically works with any kind of flight plan and inside all Airsim enviornments. In section 4.3, the whole code will be deeply examined to put in evidence the characteristics of all the functions and to easly understand how the dynamic interface has been built. Chapter 5 shows all the plots and the results of the project. Images of all the code in the "Annexes" (Chapter 8).

1.3. Step 3: AirSim Test

The last step is the test of the dynamic interface with the implemented Flight Plan. If the drone successfully completes the flight plan from take off to landing, the dynamic interface and the flight plan are well-written. If the drone stops, falls, breaks, reaches undesired postions, collides with obstacles or cuts the scans, a rework of the xml file or of the dynamic interface or both reworks are required.

1.4. Step 4: Problem Solving Approach

To write the “perfect” code and the “best” flight plan, I encoutered a series of problems that I was able to solve taking both a problem-solving metodology and an inverse-engineering approach. I usually write the code on paper to better understand how it works and how I can obtain what I want; then, I improve and fix it directly on the machine.

1.5. Step 5: XML FPs implementation for Multiple-Drones Interface

As well as in Step 1, I built four FPs. *Paolo* will use two of them containing the same path (one with Geographical coordinates, the other one with NED coordinates); *Francesca* will use the other two FPs that contain another path. In sub-section 2.5.2, all the information related to these XML Flight Plans and the strategy I adopted will be shown. Images of the four flight plans are in the “Annexes” (Chapter 8).

1.6. Step 6: Python Interface implementation for Multiple Drones

In order to implement the multiple-drones interface, one theorical concept more is required. Threading must be used to allow Python’s simultaneous handling of two different drones. In section 4.4, the whole code will be deeply examined to put in evidence the characteristics of the Python’s *class* for threading and its functions. Limits for this interface are treated in the “Recomendations” section (6.3). Images of all the code in the “Annexes” (Chapter 8).

1.7. Step 7: AirSim Test

The last step is the test of the dynamic interface with the implemented Flight Plans. If both drones successfully complete their flight plans from take off to landing, the dynamic interface and the flight plans are well-written. If just one of the drones stops, falls, breaks, reaches undesired postions, collides with obstacles or cuts the scans, a rework of the xml files or of the dynamic interface or both reworks are required.

1.8. Step 8: Problem Solving Approach

To write the “perfect” code and the “best” flight plans, I encoutered a series of problems that I was able to solve taking both a problem-solving metodology and an inverse-engineering approach. I usually write the code on paper to better understand how it works and how I can obtain what I want; then, I improve and fix it directly on the machine.

Chapter 2

EXTENSIBLE MARKUP LANGUAGE: XML FILE

2.1. XML General Features

XML stands for "*Extensible Markup Language*" and it allows the encoding of documents through both a set of rules and the definition of elements. The great newness is the encoding format that is "both human-readable and Machine-readable" [21]. XML is a "restricted form of SGML" (Standard Generalized Markup Language) and it has to be completely interoperable with SGML and HTML [5].

This language has been implemented to simplify data sharing and data transport as well as data storage and data availability; XML is well known to be self-descriptive and it easily allows platform changes [22]. Moreover, XML has to support numbers of applications as well as be compatible with SGML (see [5]).

Meanwhile extsensible-markup-language started to catching on among the informatic community, programmers developed many APIs (Application Programming Interface) to both read and process XML data [21].

2.2. A little bit of history

In 1996 the World Wide Web Consortium (W3C), "the main international standards organization for the World Wide Web" (see [20]), constituted an "*XML Working Group*" which successively developed the XML language [5].

This working group was headed by Jon Bosak of "Sun Microsystems" who collaborated with Tim Bray and James Clark. Bosak decided that HTML could not be able to satisfy the great information trade required. Exchanging data without their meaning is not enough, the machine will not understand such information. Hence, he focused his attention on SGML language and its power. On the other hand, Clark introduced the name XML and the idea of "self-closing elements" [3].

2.3. Characters of an XML file

XML files are composed by units called entities. These units have storage capability and they can contain parsed or unparsed data. Parsed-data characters are divided in "*data character*" and "*markup character*" [5]; these two objects have different applications depending on different syntactic rules. Markup strings generally begin with "<" and end with ">" ("&" and ";" is another form); then, every string that is not a markup character is a data character or "*content*" [21]. The other three main characters for the implementation of an XML file are: "*Tag*", "*Element*" and "*Attribute*".

2.3.1. Tag

"A tag is a markup construct that begins with < and ends with >". There are three different kind of tags but I only used two out of these for the master thesis: "*start-tag*" (e.g. <stage>) and "*end-tag*" (e.g. </stage>).

2.3.2. Element

An element always "begins with a start-tag and ends with a matching end-tag". In between, it can be found the "*element's content*" that can contain markup characters such as other elements called "*child elements*".

2.3.3. Attribute

A start-tag may be complemented with one or more attributes, a markup construct that associates a value to a name, for the sake of example " <leg id="zero_point" xsi_type="TF_Leg"> ". Here, "id" and "xsi_type" are the names of the attributes; "zero_point" and "TF_Leg" are respectively the values.

2.4. How to build an XML Flight Plan

All this section has been written taking advantages of the reference [10]. RAISE+ documentation takes into account RPAS but it is possible to follow the general guideline to implement a flight plan for drones. Hence, the unnecessary parts will be reasonely skipped.

2.4.1. General

In order to design an optimal flight plan, it is mandatory to know how the XML code has to be organized and implemented. The first two childs of the principal root are "*Locale Settings*" and "*MainFP*".

Locale settings indicates distances, speed and altitude measure units. Moreover, it is indicated the decimal and group separators. Figure 2.1 shows all the possible values for these elements; it also shows an exemple of the "Locale Settings" XML code.

speedUnits		altitudeUnits distanceUnits		decimalSeparator	groupSeparator
ms	m/s	m	meters	in principle it could be any string, but most probably '.' and ','	as in decimalSeparator plus empty
kt	knots	nm	nautical miles		
		ft	feet		

```
<!-- Locale settings -->
<Locale>
  <speedUnits>kt</speedUnits>
  <altitudeUnits>ft</altitudeUnits>
  <distanceUnits>nm</distanceUnits>
  <decimalSeparator>.</decimalSeparator>
  <groupSeparator/>
</Locale>
```

Figure 2.1 Locale Settings

Figure 2.2 is an exemple of the "MainFP" XML code.

```
<MainFP id="FPID">
   <name>Name of the flight plan</name>
   <description>Text describing the flight plan</description>
   <!-- List of stages that form the flight plan follows -->
   <stages> ... </stages>
   <!-- List of whitespace separated emergency plan IDs -->
   <emergency> ... </emergency>
</MainFP>
```

Figure 2.2 MainFP

A drone follows the path contained in the main flight plan; moreover, it has also a name and a description of the mission. Then, a list of all the stages is compiled and they must be executed in the correct order.

I chose to not include an emergency plan since it is not the scope of this project. In the “Area for further studies” section (6.3), the necessity of emergency Flight Plans to fly a real drone will be discussed. Moreover, take-off and landing part are directly performed through python’s APIs for AirSim. This means that the corresponding stages are not implemented in the XML codes.

2.4.2. Leg: definition and classification

Each stage has an identifier and contains all the legs belonging to it. A leg identifies the course to the next point along the flight plan; furthermore, each leg is recognized through its “*xsi_type*” attribute.

A leg can be classified depending on both behavior and functionality. I took into account four types of leg which are the most significant for the purposes of my work.

2.4.2.1. Track to a Fix (TF leg)

This type of leg performs a straight path from waypoint to waypoint and it is identified by the “ *xsi_type=”fp_TFLeg”* ” attribute. The “*dest*” tag contains as child all the informations related to the point that has to be reached. In addition, “*next*” tag highlights the name of the next waypoints corresponding to another leg. The last leg of the flight plan does not require the next waypoint child.

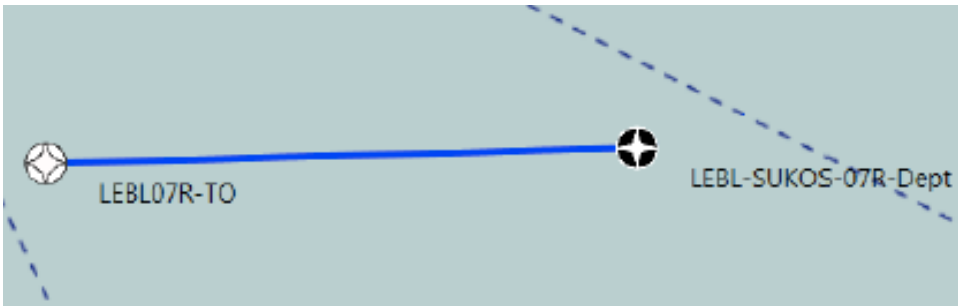


Figure 2.3 To Fix Leg

2.4.2.2. Intersection leg

This kind of leg identifies waypoints where more than one path can be selected depending on a condition that will be chosen by the user. This condition is not inserted along the flight plan. The possible legs to be selected are encoded in the tag "*nextList*" with an unique identifier. The drone will wait hovering until the user chose the needed path. This leg must not be used to emulate an iterative behavior.

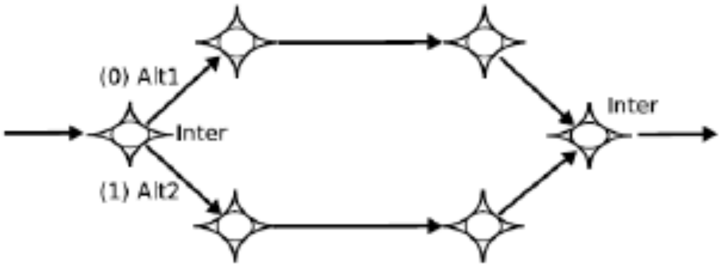


Figure 2.4 Intersection Leg

2.4.2.3. Parametric leg

Parametric legs are very useful, especially for scan paths. Identifying the key parameters, we can chose the corners of an area that will be covered by the scan. "*point1*" tag identifies the entry point of the area and the "*trackseparation*" set the distance between tracks. Then, an operative speed and an operative altitude are required.

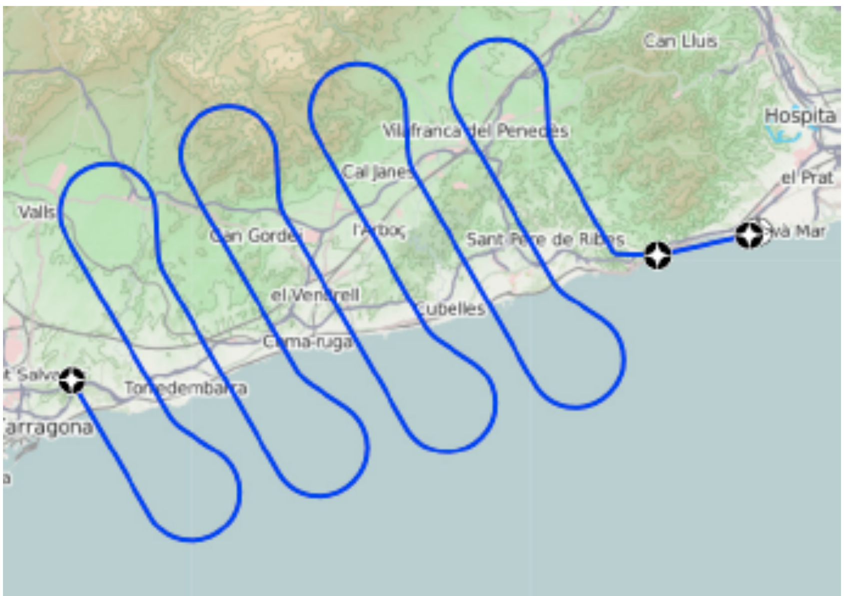


Figure 2.5 Scan Leg

2.4.2.4. Iterative leg

Iterative legs allow users to iterate a sequence of maneuvers a certain number of time. The body contains the legs which have to be iterated and initial as well as final legs have to be highlighted. Every time the drone performs the last leg, an iteration counter will be incremented. Once the number written in "*UpperBound*" tag is reached, "*next*" point will be executed ending the *Iterative Leg*.

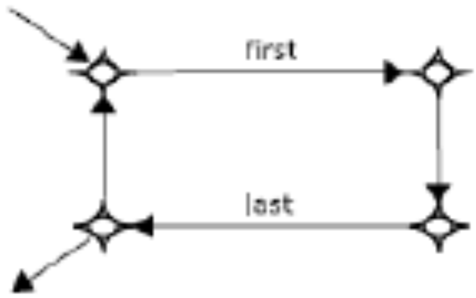


Figure 2.6 Iterative Leg

2.4.2.5. Legs not taken into account

I decided to not take into account Radius to fix leg (RF leg), Holding pattern (HF leg) and Eight leg since they are not useful maneuvers for my work and for a drone in general. All curved paths are avoided along this master thesis since a drone can use 90 degrees maneuvers to turn, to hold and to scan.

2.5. FPs Strategy

In order to build my own flight plans, I used the Neighborhood AirSim environment as reference to collect a list of coordinates and to plan a series of maneuvers. In chapter 3 the Neighborhood environment and its reference frame system as well as the conversion parameters from Geographical coordinats to AirSim NED coordinates will be deeply analyzed.

It has to be noticed that all tables and graphs, as well as in the text section, "m" stands for meters, "°" stands for degrees, "s" stands for seconds.

2.5.1. Single-Drone Flight Plan

After take off (NED coordinates: 0, 0, -2), *Carlo* enters the first stage consisting of two consequently *To Fix Leg* to arrive at the *Intersection Leg* point.The parameters of the two points for both FPs are summarized in the table below (Table 2-1).

STAGE 1				
Point Name	**Speed [m/s]**	**North Coordinate [m] / Latitude [°]**	**East Coordinate [m] / Longitude [°]**	**Altitude [m] / Altitude [m]**
zero-point (N0)	5	125 / 47.64260464285712	0 / -122.140365	-20 / 143.199297198
first-point (N1)	5	125 / 47.64260464285712	125 / -122.1386985308925	-40 / 163. 199297198

Table 2-1 N0 and N1 waypoints coordinates

Image 2.7 shows an example of *To Fix Leg* (N0 Point, NED FP) written with XML language.

```xml
<leg id="zero-point" xsi_type="fp_TFLeg">
    <dest>
        <name>N0</name>
        <north_coordinate>125</north_coordinate>
        <east_coordinate>0</east_coordinate>
        <altitude>-20</altitude>
        <speed>5</speed>
        <next>first-point</next>
    </dest>
</leg>
```

Figure 2.7 Example of To Fix Leg

Entering the second stage which is composed only by the *Intersection Leg*, the user can choose among a list of three possibilities through an input given from keyboard. The following image (2.8) shows the heading of this leg and the “nextList” composed by the three possibilities, without his body.

```xml
<leg id="second-point" xsi_type="fp_IntersectionLeg">
    <nextList>third-point-a third-point-b third-point-c</nextList>
</leg>
```

Figure 2.8 Heading of Intersection Leg

The first possibility (third-point-a, NED FP) is a *Parametric Leg* with a trackseparation of *50m* starting at point1 coordinates. The image 2.9 shows the XML code for this choice (NED FP).

```xml
<leg id="third-point-a" xsi_type="fp_Scan">
    <dest>
        <coordinates>0 0</coordinates>
    </dest>
    <trackseparation>50</trackseparation>
    <area>
        <point1>125 -125</point1>
        <point2>-125 -125</point2>
        <point3>-125 125</point3>
        <point4>125 125</point4>
    </area>
    <speed>5</speed>
    <altitude>-40</altitude>
</leg>
```

Figure 2.9 Example of Parametric Leg

The second one (third-point-b, Image 2.10, NED FP) is an *Iterative Leg* composed by a To Fix Leg followed by a scan with a trackseparation of *125m.* The UpperBound value has been fixed to *2.* After the repetition of this path, Carlo will directly fly to fourth-point with a *To Fix Leg.*

```xml
<leg id="third-point-b" xsi_type="fp_IterativeLeg">
    <next>fourth-point</next>
    <body>third-point-b-one third-point-b-two</body>
    <upperBound>2</upperBound>
    <first>third-point-b-one</first>
    <last>third-point-b-two</last>
</leg>
    <leg id="third-point-b-one" xsi_type="fp_TFLeg">
        <dest>
            <name>N3B1</name>
            <north_coordinate>-125</north_coordinate>
            <east_coordinate>125</east_coordinate>
            <altitude>-40</altitude>
            <speed>5</speed>
            <next>Home1</next>
        </dest>
    </leg>
    <leg id="third-point-b-two" xsi_type="fp_Scan">
        <dest>
            <coordinates>0 0</coordinates>
        </dest>
        <trackseparation>125</trackseparation>
        <area>
            <point1>-125 -125</point1>
            <point2>-125 -125</point2>
            <point3>125 -125</point3>
            <point4>125 125</point4>
        </area>
        <speed>5</speed>
        <altitude>-40</altitude>
    </leg>
    <leg id="fourth-point" xsi_type="fp_TFLeg">
        <dest>
            <name>N4</name>
            <north_coordinate>125</north_coordinate>
            <east_coordinate>125</east_coordinate>
            <altitude>-40</altitude>
            <speed>5</speed>
            <next>Home1</next>
        </dest>
    </leg>
```

Figure 2.10 Example of Iterative Leg

The last one (third-point-c) is another Scan starting in a different point and with a trackseparation of *62.5m*.

Once the path related to the possibility chosen is finished, to come back to the starting point *Carlo* will enter the third stage composed by two consequently *To Fix Leg* (Table 2-2). Then, the drone will land exactly where it took off.

GO HOME STAGE				
Point Name	**Speed [m/s]**	**North Coordinate [m] / Latitude [°]**	**East Coordinate [m] / Longitude [°]**	**Altitude [m] / Altitude [m]**
Home1	4	60/ 47.642021060971416	0 / -122.140365	-20 / 143.199297198
Home2	2	0 / 47.64260464285712	0 / -122.140365	-4 / 127. 199297198

Table 2-2 Home1 and Home2 waypoints coordinates

Images of the complete XML file are in “Annexes” (Chapter 8).

2.5.2. Multiple-Drones Flight Plans

Paolo will be the first drone to start its mission; its FP consists of a *To Fix leg* and a successively *Iterative leg* composed by four *To Fix legs. Paolo* will fly from corner to corner, covering all the Neighborhood external peremeter for twenty times. After this *Iterative leg*, the drone will come back to the starting point. *Francesca* takes off with *Paolo*, but it has to wait to execute the mission until the user gives an input from keyboard. Its FP only consists of an *Intersection leg*: the user has to choose among four different scanning paths, one for each section of the Neighborhood enviornment corresponding to one of the four quadrants of the Cartesian Plane (more details in section 3.3). The idea is that *Paolo* starts to monitor the Neighborhood covering the external peremeter; if it founds something wrong in one of the section of the map, *Francesca* will start its scanning path over that area. Images of the complete XML files are in “Annexes” (Chapter 8).

2.6. XML Processing

In chapter 4, it will be explained how parse an XML file on Python and how reach the different childs through their tags and attributes. Moreover, the implementation of the Python functions representing the four different legs taken into account will be deeply described. Finally, both data-storage and transmission-commands codes will be outlined to show how the dynamic interfaces store waypoints informations and turns these data into commands for the drones in the simulator environment.

Chapter 3

AIRSIM SIMULATOR

3.1. Overview

In order to save money and time, simulators usefulness has been incrisingly growing in the last years. Simulation provides necessary data and it allows users to test mathematical models as well as the behavior of facilities and dynamic systems. The great advantages are the cost and time optimization. Since a drone can crash numbers of time in the simulator, we can deeply learn how it behaves and how fix problems arising during tests.

This master thesis will take advantages of this technology making a step torward the implementation of an autonomous drone, which can move without any external aid. The dynamic interface will help ICARUS project (UPC) to test several flight plans on several AirSim environments without any kind of dependencies.

3.2. General Features: Unreal Engine and AirSim

"Unreal Engine is a complete suite of creation tools designed to meet ambitious artistic visions while being flexible enough to ensure success for teams of all sizes" [19]. In 1998 Epic Games presented "*Unreal Engine (UE)*" that is an engine developed for a wide variety of game genres, especially first-person shooters games. Later on, this exceptional softaware has found several other applications and actually, the most recent version is UE4 (released in 2014) [15].

AirSim (2017) is an open-source simulator still under development and it takes advantage of Unreal Engine and its enviornments. UE enviornments are shaped with physics and aerodynamics models, taking into account all the forces and torques acting on the vehicles. All these models are taken as inputs by the physics engine to allow the computation of the vehicle kinematics-state in the simulated world. AirSim is a very complex and well-made simulator and it is not easy to be used. Figure 3.1 shows the AirSim interface respectively with a car in the "City Virtual Enviornment" and a drone in the "Neighborhood Virtual Enviornment".

AirSim has been provided with a set of APIs to control vehicles in the simulator through several programming languages. This cross-platform capability allows control transmission from both C++ and Python programming codes as well as a support for Windows and LinuxOS platforms.

Moreover, one of the most important AirSim capability concerns deep-learning as well as reinforcement-learning algorithms for vehicles moving independently [18].

Figure 3.1 Car in City Enviornment - Drone in Neighborhood Environment

3.3. Neighborhood Environment: Reference Frame System

The environment chosen for the implementation of the XMLs is the "*Neighborhood*" AirSim enviornment. All the coordinates that appear in the six Flight Plans have been rationaly taken to allow Carlo, Paolo and Francesca safe flight.

AirSim reference-frame-system is a North, East, Down frame (NED). Hence, x-axis represents North-coordinates, y-axis represents Est-coordinates and z-axis (representing altitude) is pointing down, so all the altitude coordinates will be negative.

Neighborhood envioronment (figure 3.2) can be approximated to a square with a side of *250m*. The default starting-point $(NED\ coordinates\colon 0m, 0m, \approx -2m)$ is at the diagonals intersection-point and the front camera is initially pointing towards the positive North direction. Since the z-axis is pointing down, taking advantages of the "right-hand rule" it will be clear that the positive East direction is on the drone right-side at the default starting point.

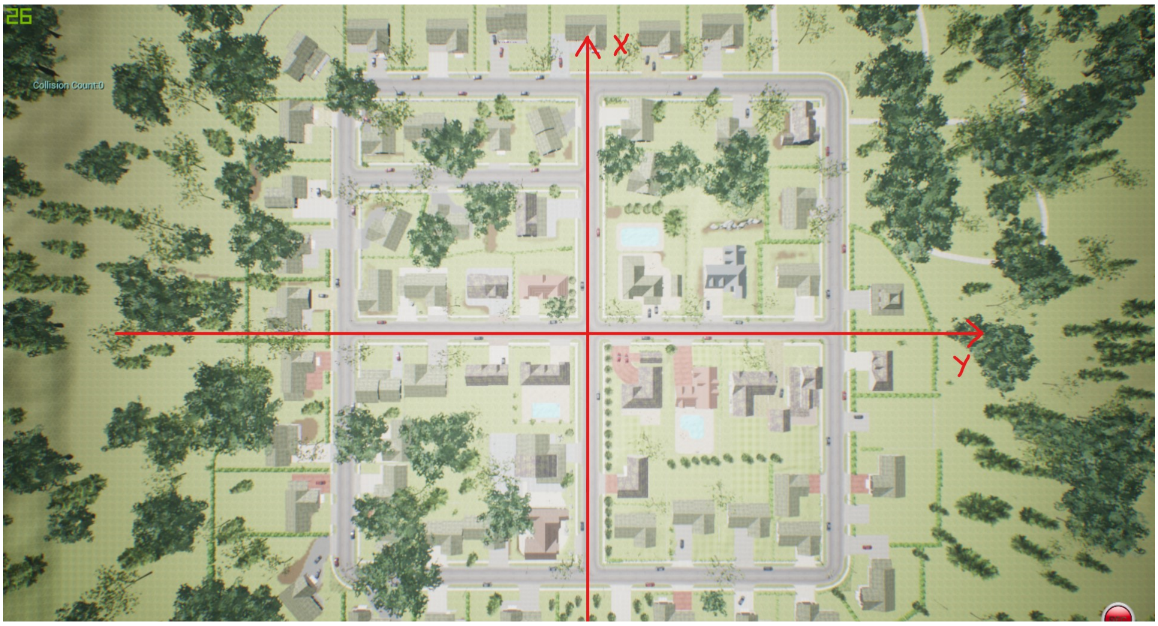


Figure 3.2 Neighborhood Reference Frame System

AirSim Simulator only takes NED coordinates as input hence, for the Geographical FP it is required that the interfaces convert the coordinates before executing commands. To find the conversion parameters (ΔLat and $\Delta Long$), the Python's library *geographiclib* has been used; the altitude does not require a conversion parameter. In the table below symbols and values for the three initial Geographical coordinates (default starting point; evaluated with "getMultirotorState" AirSim's function), the two conversion parameters for the chosen environment (calculated with *geographiclib*) and the initial NED altitude are summarized.

SYMBOLS AND VALUES		
Data	**Symbol**	**Value**
Initial Latitude	Lat_0	$47.64148237°$
Initial Longitude	$Long_0$	$-122.140364°$
Initial Geographical Altitude	h_{geo_0}	$125.1653061\mathrm{m}$
Initial NED Altitude	h_{NED_0}	$-1.966008902\ \mathrm{m}$
Latitude conversion Parameter	ΔLat	$8.99415308151351 \times 10^{-6}\ °/m$
Longitude Conversion Parameter	$\Delta Long$	$1.33084196690962 \times 10^{-5}\ °/m$

Table 3-1 Default Starting Point Coordinates - Conversion Parameters

Knowing the current latitude (Lat), longitude ($Long$) and Geographical altitude (h_{geo}), we can evaluate the actual NED coordinates (N_C, E_C, h_{NED}) with the following equations (1, 2 and 3):

$$N_C = (Lat - Lat_0)/\Delta Lat \qquad (1)$$

$$E_C = (Long - Long_0)/\Delta Long \qquad (2)$$

$$h_{NED} = h_{NED_0} - (h_{geo} - h_{geo_0}) \qquad (3)$$

Moreover, it is possible to change the default strating point coordinates. Figure 3.3 shows the "setting.json" file and how to set these coordinates.

```
{
  "SeeDocsAt": "https://github.com/Microsoft/AirSim/blob/master/docs/settings.md",
  "SettingsVersion": 1.2,
    "SimMode": "Multirotor",

  "OriginGeopoint": {
    "Latitude": 47.641468,
    "Longitude": -122.140165,
    "Altitude": 122
  }
}
```

Figure 3.3 Set default starting point coordinates in "Setting.json"

3.4. Surveillance Mission

For the development of the project, according to the surveillance-mission objective and its scan-paths, I chose a service altitude of $-40m$ in order to avoid any conflict with trees and houses. The real drone will use the bottom camera to scan and control the area and it will transmit real-time photos from above, so the AirSim frontal camera has not been employed (neither to detect and avoid obstacles). Moreover, to avoid conflicts between drones in the multiple-drones scenario, the first drone will fly at $-30m$ and the second one at $-40m$. Since the first drone follows only the external peremeter, it will not have any collision with trees or houses. In the “Recommendations” section (6.3) all the responsablities of the FPs designer will be explained.

In chapter 4, the connection-to-AirSim code of the interfaces and how transimt commands to the drones in the AirSim enviornment taking advantages of Python and its APIs will be explained.

Chapter 4

DEVELOPMENT OF THE FP INTERPRETER

4.1. Python features

Python (see [8]) is an interpreted, interactive and object-oriented programming language. “Interpreted” means that Python directly launches the source file. “Interactive” means that it is allowed the handwriting of istructions directly in the command prompt. Neverthless, it is possible to download several *Integrated Development Enviornments (IDE)* that simplify Python usage.

This programming language takes advantages of moduls (import command), exceptions and their management (try, except, finally, else instructions), dynamic typing and high-level and high-class data like lists, set and dictionaries and their *comprehension*. These and others are the features that make Python one of the best programming language in the world: e.g. strings, tuple, mandatory indentation, standard libraries, slicing, application libraries, functions, dictionaries etc. The sintax is extremely clear and easy; the usage of Boolean values (True, False, None) is allowed.

4.2. A little bit of history

Python was created by Gudio Van Rossum and released in 1991 for free directly on the web. Guido is a very famous expert in programming languages and, immediately after its released, Python gained popularity among the informatic community. Guido Van Rossum gave this name to its “creature” in honor of the 70’ rock group *Monty Python*, who choose this name because “*it sounded funny*”.

4.3. Single-Drone Interface

Each sub-section presents one of the several parts that make up the code. Moreover, this section has been divided in as many sub-sections as the number of topics covered.

4.3.1. AirSim

AirSim’s APIs for Python allow us to connect and disconnect our script from the AirSim simulator; moreover, take off and landing are directly executed by two different functions.

Figure 4.1 (from code line 612 to 633) shows how to connect the script to the simulator, enable APIs commands, get the drone state and store the initial coordinates into variables. Initial coordinates are necessary for plots, for landing and for the coordinates conversion if the Geographical FP is used. Figure 4.2 (code lines 830 and 831) presents the code lines to execute the take off. The method Async calls future, so we have to wait until the take off is completed. To do this, we use the function “sleep” of the “time” library; the input of this function represents the number of seconds to wait.

```
client = airsim.MultirotorClient()
client.confirmConnection()
client.enableApiControl(True)
client.armDisarm(True)

state = client.getMultirotorState()
s = pprint.pformat(state)
print("state: %s" % s)

x_home = state.kinematics_estimated.position.x_val
y_home = state.kinematics_estimated.position.y_val
z_home = state.kinematics_estimated.position.z_val

lat0 = state.gps_location.latitude
long0 = state.gps_location.longitude
h_geo0 = state.gps_location.altitude

arcWE = geodesic.Geodesic.WGS84.Inverse(lat0, long0-0.5, lat0, long0+0.5)
arcNS = geodesic.Geodesic.WGS84.Inverse(lat0-0.5, long0, lat0+0.5, long0)

lat_conv = 1 / arcNS['s12']
long_conv = 1 / arcWE['s12']
```

Figure 4.1 Connection to the simulator code lines

```
client.takeoffAsync()
time.sleep(4)
```

Figure 4.2 Take off code lines

Figure 4.3 (from code line 856 to 859) shows how to implement drone landing and disconnect the script from the simulator. Since the last point of the FP is perfectly above the landing point (default starting point), instead of using the "land" function, "moveByVelocityZAsync()" function has been used to have a vertical movement to reach the ground (".join()" replaces "time.sleep" function).

```
client.moveByVelocityZAsync(0, 0, z_home, 2, airsim.DrivetrainType.MaxDegreeOfFreedom, airsim.YawMode(False, 0)).join()

client.armDisarm(False)
client.enableApiControl(False)
```

Figure 4.3 Langind code lines

To allow *Carlo*'s movement, the remaining function used in the code is the "moveToPositionAsync()". This function takes at least four inputs: north coordinate, east coordinate and altitude of the point to be reached and the cruise speed. In all the Python's functions only this AirSim's function is used to move the drone and to execute all the possible maneuvers. Again, "Async" method calls future, so "time.sleep" function is required. In sub-section 4.3.9.2 it will be shown how wait the right amount of time to perfectly complete each section of the FP with my "check_position" function.

4.3.2. XML Parsing

In order to make Python capable of reading and decoding an XML file, this file has to be parsed. To do this, it can be used one of the many Python's libraries that allow the processing of XML files.

The “ElementTree” (ET) library (the import of this library will be shown in sub-section 4.3.8) has been chosen. Using the function “ET.parse()” that takes a string containing the name of the file as input, the XML FP can be parsed and saved into a variable “tree”. It has to be noticed that the file has to be in the same folder of the main script. After that, to decode the file, other two lines of code have to be added.

Figure 4.4 (from code line 588 to 596) shows the implementation of the parsing code for the XML FPs. As the code starts to run, the user has to choose which flight plan wants to parse and use it to move *Carlo* in the simulator environment. The variable “FP” will be set giving an input from keyboard. If “FP” is set to “0”, the parsed geographical flight plan will be used; if “FP” is set to “1”, parsed NED flight plan will be used. The last two lines of code allow us to decode into strings the whole XML FP.

This part of code could be improved by browsing the desired file to be parsed in the machine. It was not a key point of the project but, with an eye towards the marketing of the interface, this improvment can be easily done.

```
FP = input("Which Flight Plan do you want to use? Please, select 0 for Lat/Long FP or 1 for NED FP:")

if FP == '0':
    tree = ET.parse('DroneFlightPlan_Neighborhood_Final_Version_LongLat.xml')
elif FP == '1':
    tree = ET.parse('DroneFlightPlan_Neighborhood_Final_Version_meters.xml')

root = tree.getroot()
ET.tostring(root, encoding='utf8').decode('utf8')
```

Figure 4.4 XML Parsing code lines

4.3.3. Flight Plan Processing

From code line 637 to 826, it has been implemented a *dictionary* (*waypoints_data*) aimed at storing all the informations contained in the Flight Plan.

First of all, an empty dictionary to be filled has to be created. After that, the variable “FP” (previous sub-section) will be the discriminating factor in the selection of the right “if-loop-branch”. Indeed, the code lines to search inside the geographical flight plan are different with respect to the code lines to search inside the NED one. It has to be noticed that the working process is the same but the content to be searched inside the decoded XML is different (i.e. latitude or n_coord, longitude or e_coord, ...). Moreover, the other difference between the two “if-loop-brunches” is the presence of the code lines aimed at converting the geographical coordinates of the geographical flight plan (“FP=0”) into NED coordinates for AirSim. Hence, the dictionary will only contain NED coordinates (whatever XML has been used).

The working process to search the required informations inside the XML file is very cumbersome. With a “for loop”, we can iterate over the whole XML (variable “tree”) to search all the objects with a “tag = leg”. The name of each leg is stored as element of the dictionary and it will in turn be a dictionary (nested dictionary). Then, dependening on the leg’s attribute that specifies the leg type, all the significant parameters will be saved inside this nested dictionary. In the end, a dictionary containing as many elements as the flight-plan-waypoints are (an unique key-word identifies each element) is obtained; these elements are themselves dictionaries containing as many elements as the significant parameters of each leg are. Each significant parameter will be saved with its key-word. For a *To Fix Leg* the significant parameters are 4 (x-coord, y-coord, altitude and speed); for a *Scan Leg* are 6 (trackseparation, x-coord and y-coord of the "point1", altitude, speed and the number of tracks); for an *Iterative Leg* are 3 (name of the points in the "body", name of the "next point" and the "UpperBound" value); for an *Intersection Leg* are 2 (name and amount of the points contained in the "nextList").

This dictionary allow us to store these parameters that will be needed to perform the required maneuvers. Each leg function contains code lines to search inside the dictionary (using the right key-word). Then, storing the elements into variables, the function will be able to execute the desired path.

4.3.4. Processing of Flight Plan Legs

To make the drone capable of following the required path, I implemented four functions, one for each type of leg taken into account along this master thesis. It has to be noticed that the "check position" function is employed everytime an AirSim's movement function is used inside these "legs functions". Its utility will be clarified in sub-section 4.3.9.2. Moreover, assuming that we want to plot graphs at the end of the simulation (LOG = "ON", 4.3.6), a certain number of code lines are implemented to store time and position data.

4.3.4.1. Tf_leg Function

The first function to be presented is the *To Fix Leg* function (code lines 75 and 76). Image 4.5 shows that this function only consists of a "moveToPositionAsync" AirSim's function. This calls will return a straight movement from the current position to the point described by the four inputs passed to this function (North coordinate = n_coord, East coordinate = e_coord, altitude = alt, velocity = speed).

```
def tf_leg(n_coord, e_coord, alt, speed):
    return client.moveToPositionAsync(n_coord, e_coord, alt, speed)
```

Figure 4.5 To Fix Leg Processing code lines

4.3.4.2. Scan Function

The scan function (from code line 81 to 391) allows drones to perform a scan path over the desired area. It takes as input the coordinates of the scan starting-point (n_coord, e_coord, h, v) and the trackseparation (ts). These inputs represent the size of the scan area. The last input (i) is obtained by dividing the side length of the scan-area by the trackseparation; this number represents the number of tracks of the scan.

First of all, a *To Fix Leg* will bring the drone at the scan starting-point. After that, the waypoints forming the scan path will be evaluated, depending on the side length of the scan-area and the trackseparation as well as the starting point coordinates. Once the points are calculated, a "for loop" containing a "moveToPositionAsync" AirSim's calls will pass point by point all the waypoints previously stored in a list.

This function is composed by an external "if loop" (one "if" and three "elif", one for each corner of the scan-area, including the possibility that the north coordinate or the east coordinate of the corner can be equal to zero). This is due to the different behaviour of the scan path which depends on the starting point coordinates. Then, inside each branch of this “if loop”, there is another “if loop” (one “if” and one “elif”). This is due to the number of tracks (i) that can be odd or even.

4.3.4.3. Iterative_leg Function

At the beginning of this function (from code line 396 to 446), the whole path (composed by one or more legs) to be iterated is stored. Then, using a “while loop” it can be ensured that the drone will perform the desired path a number of time equal to the “UpperBound” limit. Hence, the code will run inside this loop as long as a counter (starting from zero and updated at the end of each iteration) will be minor than the UpperBound value. After that, the “next point” coordinates are evaluated and passed to the *To Fix Leg* function.

The single-drone interface allows us to put another Iterative Leg or an Intersection Leg as leg of the path to be iterated (as well as to-fix legs and scan legs).

4.3.4.4. Intersection_leg Function

This function takes as input the option corresponding to the user selection and the name of the point to be reached. Then, the respective leg function will be used depending on the leg type.

The single-drone interface allows us to put an Iterative Leg or another Intersection Leg as possible choice to be selected (as well as to-fix legs and scan legs).

4.3.5. Flight Recording

In order to plot the path of the drone in the AirSim enviornment (LOG = “ON”, 4.3.6), the position of the drone while it is moving around the map has to be stored. I chose to create an empty spread sheet that is filled by “get_data()” function (sub-section 4.3.9.1).

Figure 4.6 (from code line 602 to 606) shows how create an spread sheet, how add a worksheet and how write some text into a cell. “worksheet.write()” take the row’s number as first input, the column’s number as second input and a string containing the text to be written as third input.

Figure 4.7 (code line 861) represents the closing function for our spread sheet. After the spread sheet has been closed, it can be found in the same folder of the main script.

```
data_collection = xlsxwriter.Workbook('Data.xlsx')
worksheet = data_collection.add_worksheet()
worksheet.write(0, 0, "North Coordinates")
worksheet.write(1, 0, "East Coordinates")
worksheet.write(2, 0, "Altitudes")
```

Figure 4.6 Spread Sheet Creation code lines

```
data_collection.close()
```

Figure 4.7 Spread Sheet Closure code line

4.3.6. Visualization of Flight Recording

At the beginning of the main program, once the XML file is parsed, the user has to choose if he wants to plot graphs at the end of the execution of the FP or not. As previously said, this option will affect the check_position function. The LOG variable will be stored though an input from keyboard (code line 598). To plot the graphs, “ON” has to be written; on the contrary, to avoid graphs, “OFF” has to be written.

```
LOG = input("To run the code with plots enter ON; to run the code without plots enter OFF: ")
```

Figure 4.8 LOG variable code line

After the disconnection from AirSim and the spread sheet closure, the code presents all the lines to plot (from code line 867 to 1044) the desired significant graphs. These graphs will only appear the LOG variable is set to “ON”. I chose to plot 6 graphs for each choice of the *Intersection leg* that will be shown in Chapter 5.

4.3.7. Execution of the FP

From code line 837 to 852 the creation of the empty vectors that will contain all the data to be plotted can be found; moreover, few code lines to execute the XML FP can be also found (we need only to call the “stage” function inside a for-loop to pass every stage name).

4.3.8. Libraries Used

At the beginning of the script, all the libraries imports required to execute the code (Figure 4.9, from code line 5 to 20) can be found. The following list wants to give a quick look at all libraries purposes:

- “airsim” library allow us to interact with the simulator;
- “pprint” library is used to print Carlo’s state parameters after the connection to the simulator;
- “time” library allow us to evaluate the time required by the drone to execute each stage; moreover, this library is necessary to allow Python to sleep while the drone is reaching the desired postion (“check_postion” function);
- “math” library contains all the mathematical operations as the square root (“sqrt” function) and others;
- “xlsxwriter” library allow us to write into the excel file;
- “openpyxl” provides the functions needed to open the excel file once it is closed as well as to use all the data inside to plot graphs;
- “xml.etree.ElementTree” is the library that permits the parsing and the decoding of XML files;
- “geographiclib” library is used to find the conversion parameters to obtain NED Coordinates from Geographic Coordinates. This library allows the interface to perfectly work all around the simulated world of Unreal Engine;
- the remaining three libraries (mplot3d, numpy and matplotlib.pyplot) allow us to plot different 3D plots.

```
import airsim

import pprint
import time
from math import *

import xlsxwriter

import xml.etree.ElementTree as ET

from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
import openpyxl

from geographiclib import geodesic
```

Figure 4.9 Libraries Used code lines

4.3.9. Other Functions

4.3.9.1. Get_data Function

"get_data" function (from code line 27 to 35) allows us to store the drone current position (NED coordinates) directly in an excel file.

This function is employed inside the "check_position" function (sub-section 4.3.9.2) and will be only used if the variable "LOG" is set to "ON" (sub-section 4.3.6).

Figure 4.10 shows the body of the function. The variable "column" is created in the main program (code line 845) and it is initially set to "0". The first time that "get_data" is called, "column" will be set to "1" and it will be the counter that slides column by column inside the excel file. In the end an excel file composed by several columns containing all the positions covered by the drone is obtained, from take off to landing.

```
def get_data(state):
    global column
    column += 1
    n_c = state.kinematics_estimated.position.x_val
    e_c = state.kinematics_estimated.position.y_val
    h = state.kinematics_estimated.position.z_val
    worksheet.write(0, column, n_c)
    worksheet.write(1, column, e_c)
    worksheet.write(2, column, h)
```

Figure 4.10 Get_data Function code lines

4.3.9.2. Check_position Function

“check_position” function (from code line 40 to 70) can be considered the most important one of the code. To allow Carlo’s movement, the “moveToPositionAsync()” function has been used. “Async” method calls future, so “time.sleep” function is required.

“check_position” firstly evaluates the theoretical time required to perform the desired stretch of path (more details in section 5.7). After that, if the variable “LOG” is set to “ON”, it will be used a “busy-waiting-method“: python will sleep *0.1* seconds at a time *while* the desired position is not reached. Moreover, each *0.1* seconds “get_data” function will be called to store all the current positions covered by the drone. On the contrary, if “LOG” is set to “OFF”, we will not need plots and the function “get_data”: python will sleep the whole theoretical time at once. Then, a safety “while loop” will check if the desired position is reached or not; python will sleep *0.1* seconds at a time while checking. The second method presents some advantages, first and foremost the reduction of the machine workload.

4.3.9.3. XML Tree Traversing

“stage” function (from code line 573 to 583) is employed in the execution of the main program inside a “for loop”. This function takes as input the name of the stage to be performed, employes “get_first_child” function to create a list of all the stage waypoints and uses “leg” function inside a “for loop” to pass this list point by point. Moreover, it is evaluated the real-simulation-time to perform the whole stage that is stored in a vector. This vector will be used by the plot code-lines.

“get_first_child” function (from code line 522 to 568) takes as input the name of the stage to be performed and returns the stage’s first-childs. This is a safety function aimed to perform the FP in the correct order and avoid the repetition of already covered waypoints.

“leg” function (from code line 487 to 517) takes as input the name of the point to be reached and returns one of the leg functions previously described (depending on the leg type corresponding to the point).

4.4. Multiple-Drones Interface

This section wants to show the tool used to allow Python’s simultaneous handling of two or more drones (threading). Moreover, the whole code will be presented and analyzed to underline the main parts that make it up.

4.4.1. Threading

To allow Python’s simultaneous handling of two or more drones, threading is required. “A thread is a separate flow of execution” (see [7]); the script will have two or more tasks to be simultaneously accomplished. Moreover, to obtain a perfect thread’s management, threads’ synchronization is required. A “ThreadPoolExecutor” has been used to create and start threads (it submits one function to each thread); a Python’s “Class” has been implemented to obtain threads’ synchronization through the “Lock” function of the “threading library”.

4.4.2. Libraries Used

At the beginning of the script, all the libraries imports required to execute the code (Figure 4.11, from code line 5 to 17) can be found. With respect to the sub-section 4.3.8, all the libraries to plot graphs are not required and two more libraries have been added.

“Concurrent.futures” library allows the creation and the launch of the required number of threads using a “ThreadPoolExecutor” and a “for-loop”; each thread (executor) will execute a specific function.

“Threading” library, in particular the function “Lock” of this library, allows a basic synchronization of the threads created by the “ThreadPoolExecutor”.

```
import airsim

import threading
import time
import concurrent.futures

from math import *

import xml.etree.ElementTree as ET

import numpy as np

from geographiclib import geodesic
```

Figure 4.11 Libraries used code lines

4.4.3. AirSim

After the import of the libraries, the code lines required for the connection to AirSim can be found. Then, the user has to chose how many drones will fly in the simulator enviornment though an input given by the keyboard (“population” variable); a “for-loop” will create the desired number of drones (each one with an unique name: Drone1, Drone2,...). Moreover, “population” variable is also used in the “for-loop” of the “ThreadPoolExecutor”.

To allow the take off of all the drones and to put them at different heights, a “for-loop” has been implemented using the “moveToZAsync” function. The first drone (Drone1) will hover at $-2m$ until all drones finish the take off; the second one (Drone2) will hover 3 meters above ($-5m$), the third one 6 meters above ($-8m$) and so on.

To implement the landing part, a specific function has been created (sub-section 4.4.4)

4.4.4. Execution of the FP

First of all, a Python’s “Class” (MainProgram) has been created (from code line 43 to 2058). Moreover, it has been initialized with the definition of the “self._lock” variable that takes advatages of the “Lock” function of the “threading” library. To synchronize threds, “self._lock.acquire” and “self._lock.release” functions will be used.

The first function encountered in the class is the “execution” function (from code line 47 to 2042). It takes as input the “self” variable and the “drone number”; this second input identifies one of the multiple drones created at the beginning of the main script with an unique name. This function combines all the functions previously described for the single-drone interface. As a result, each thread will only use one function to execute all the FP. This approach imposes new limits on the interface that will be discussed in the “Recmmendations for Multiple-Drones Interface” sub-section (6.3.3). Moreover, it has to be noticed that “self._lock.acquire” and “self._lock.release” functions are only used to parse the XML and to create the dictionary (from code line 49 to 275). After that, threads’ synchronization is obtained and the usage of these two functions is no longer required.

The second and final function belonging to the “MainProgram” class is the “landing” function. It takes as input the “self” variable and the “drone number”; this second input identifies one of the multiple drones created at the beginning of the main script with an unique name. Here, threads’ synchronization is obtained using “with self._lock:” code line (2046): each thread has to wait for the completion of the landing stage of the previous one.

Using a “for-loop”, the ThreadPoolExecutor will submit to each thread one of the functions contained in the MainProgram Class at time. Then, to submit another function to each thread with another “for-loop”, all threads have to finish their tasks. From code line 2062 to 2072, the two “for-loops” required to submit the “execution” function and the “landing” function to the two threads can be found.

Chapter 5

PLOTS AND RESULTS

This chapter shows all the plots and results obtained running the single-drone dynamic interface. I tested both Geographical and NED flight plans; I run each FP three times, once for each *Intersection leg* possibility. I Decided to plot six significant graphs. The first one compares the theoretical path (waypoints of the FP) and the simulated path (the path followed by the drone in the simulator). The second plot compares the theoretical North coordinate with the simulated one (first graph), the theoretical East coordinate with the simulated one (second graph) and the theoretical Altitude with the simulated one (third graph). Finally, the last four graphs compare the theoretical and the simulated time spent to execute each stage of the FP and the total time spent for the mission. Hence, each section presents six plots and a table containing the detailed time values.

5.1. Plots and Data First Choice Intersection NED FP

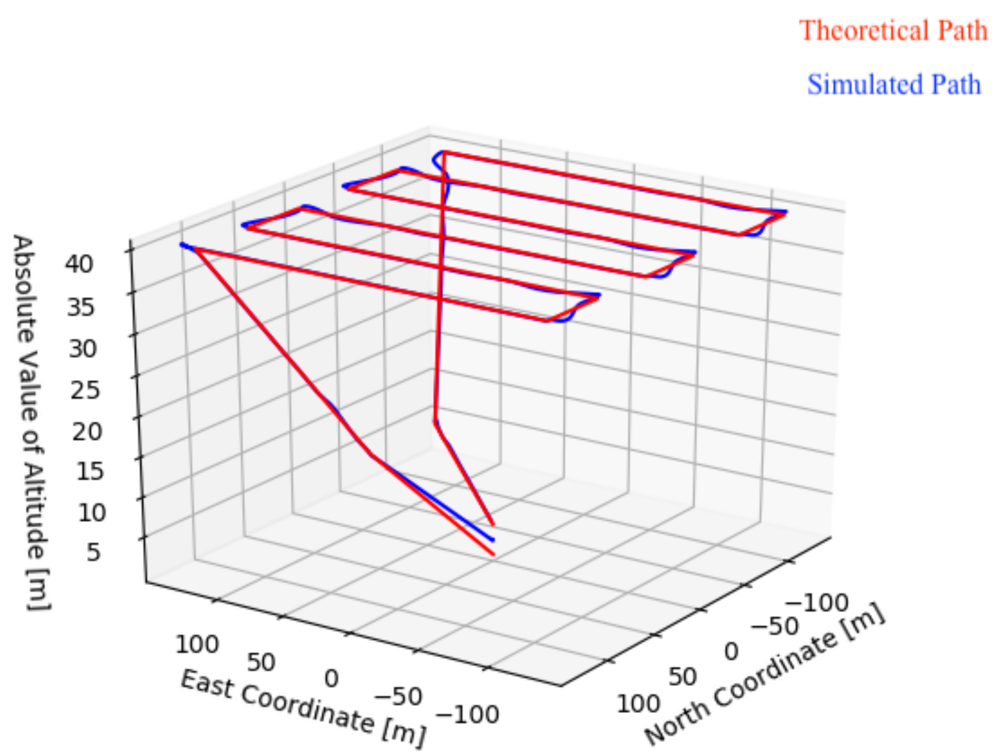


Figure 5.1 Theoretical Path - Simulated Path comparison

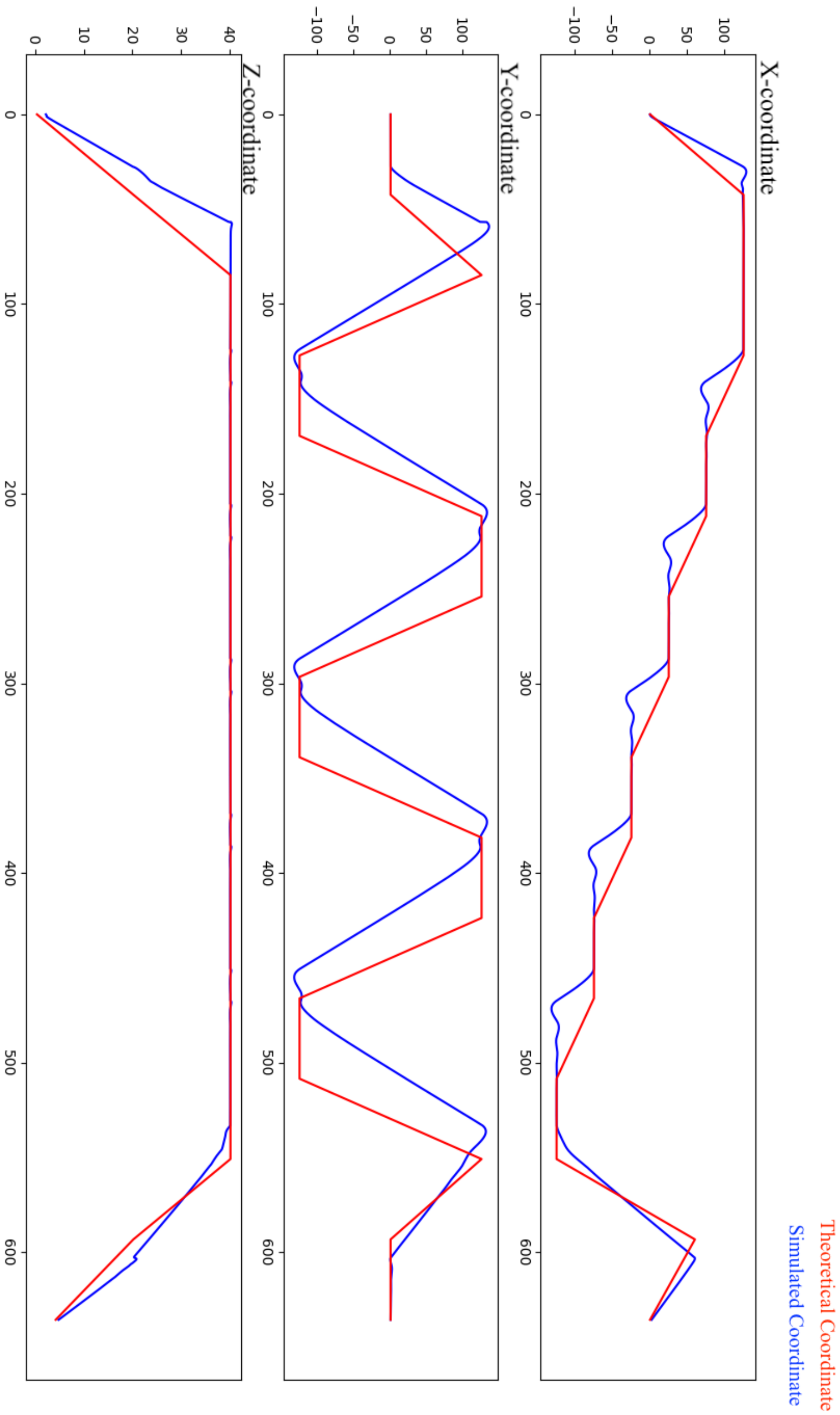


Figure 5.2 Theoretical Coordinates - Simulated Coordinates comparison

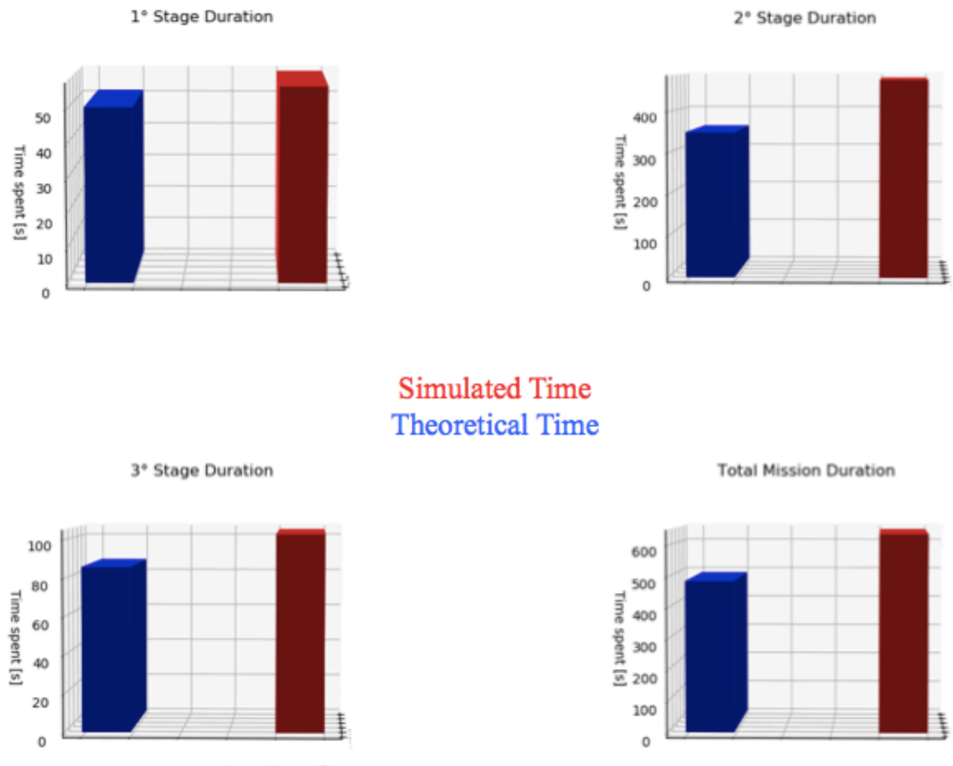


Figure 5.3 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	**Real Time [s]**
1° Stage	50.59	56.45
2° Stage	349.01	474.05
3° Stage	85.95	102.88
Mission Duration	485.55	633.38

Table 5-1 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is $147.83s$, the 30,4% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.2. Plots and Data Second Choice Intersection NED FP

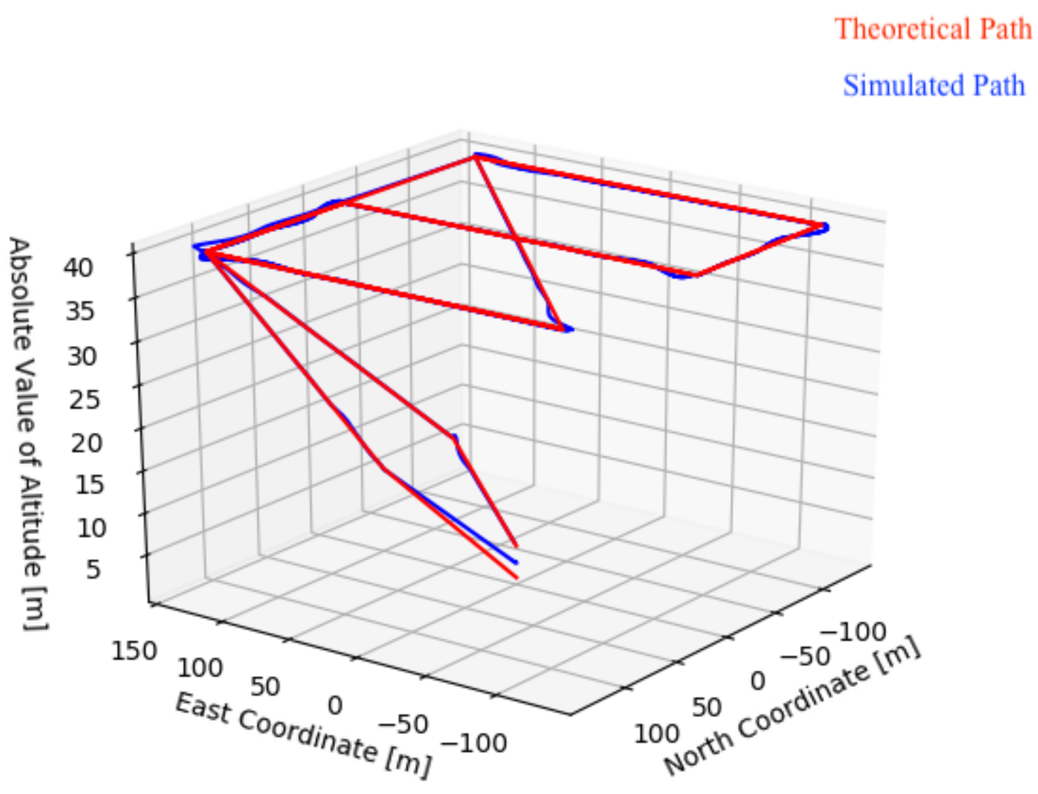


Figure 5.4 Theoretical Path - Simulated Path comparison

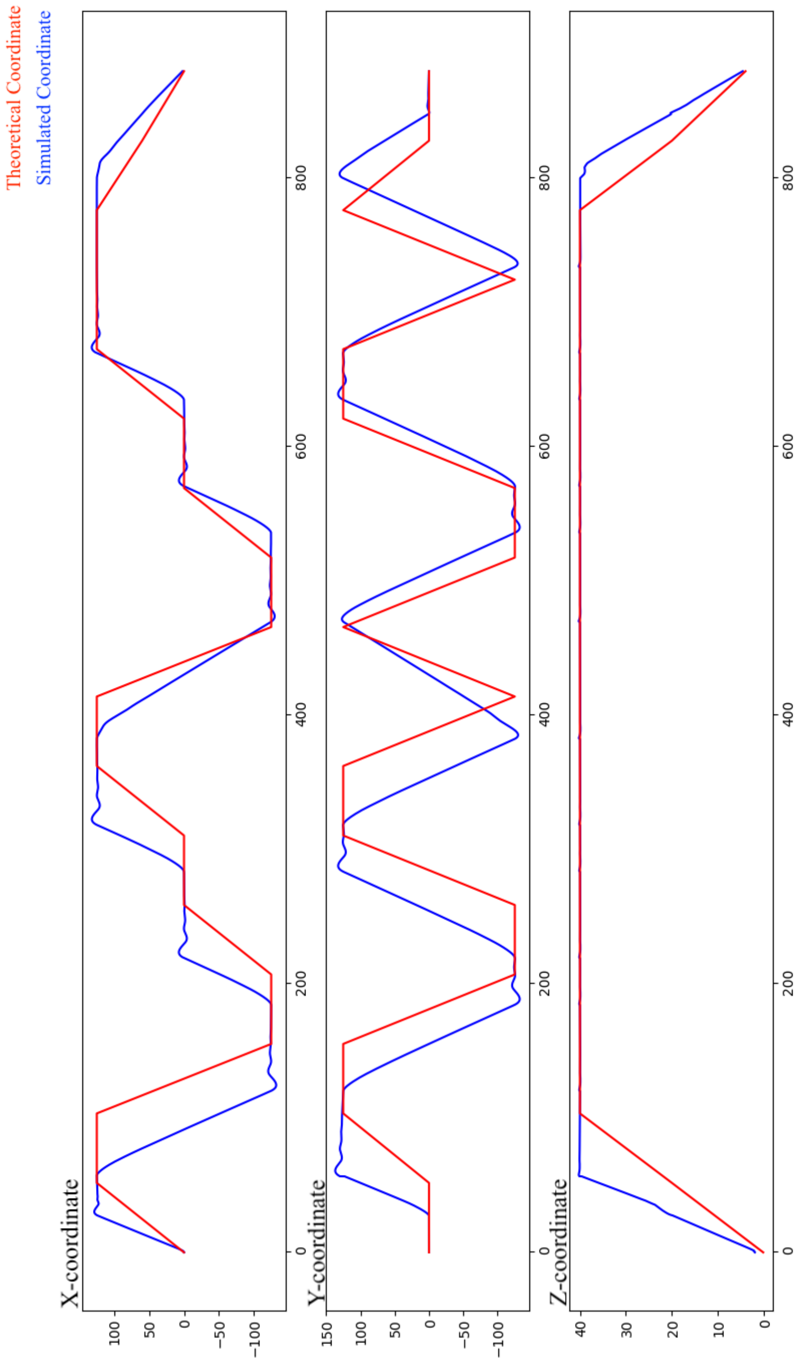


Figure 5.5 Theoretical Coordinates - Simulated Coordinates comparison

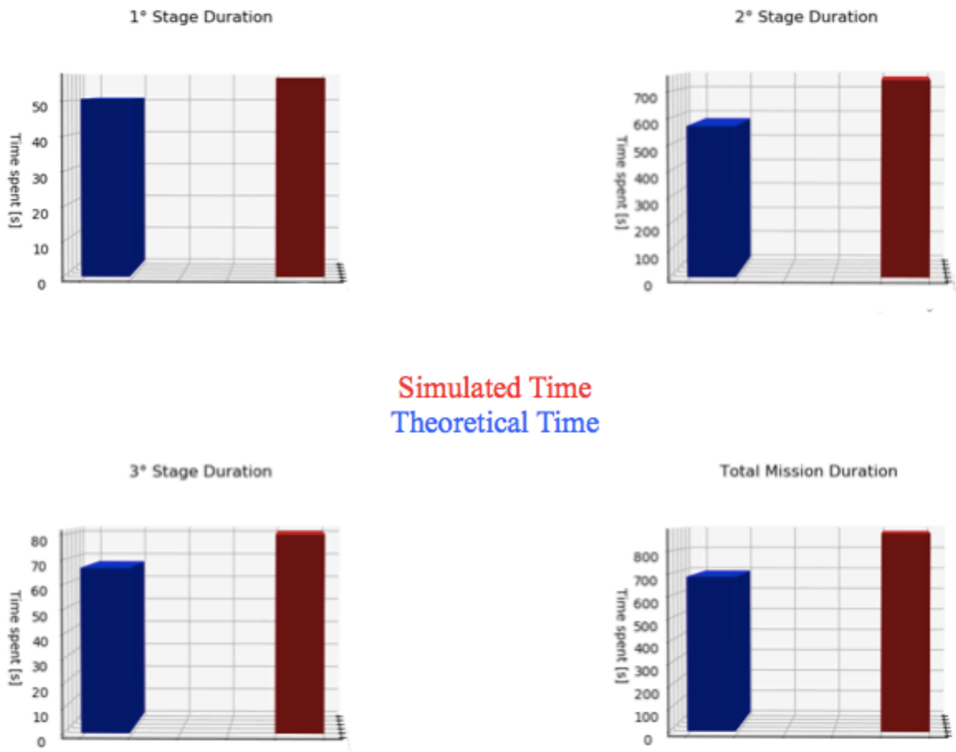


Figure 5.6 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	**Real Time [s]**
1° Stage	50.59	56.58
2° Stage	570.05	742.62
3° Stage	66.78	80.16
Mission Duration	687.42	879.36

Table 5-2 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is $191.94s$, the 27.9% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.3. Plots and Data Third Choice Intersection NED FP

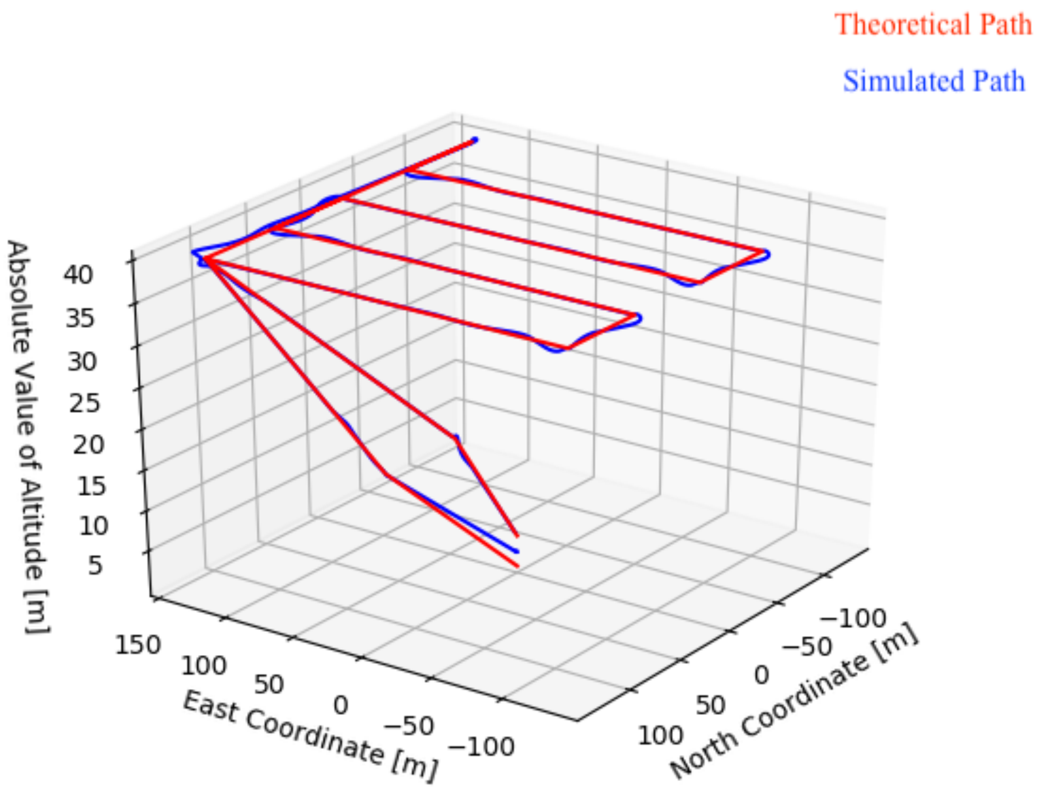


Figure 5.7 Theoretical Path - Simulated Path comparison

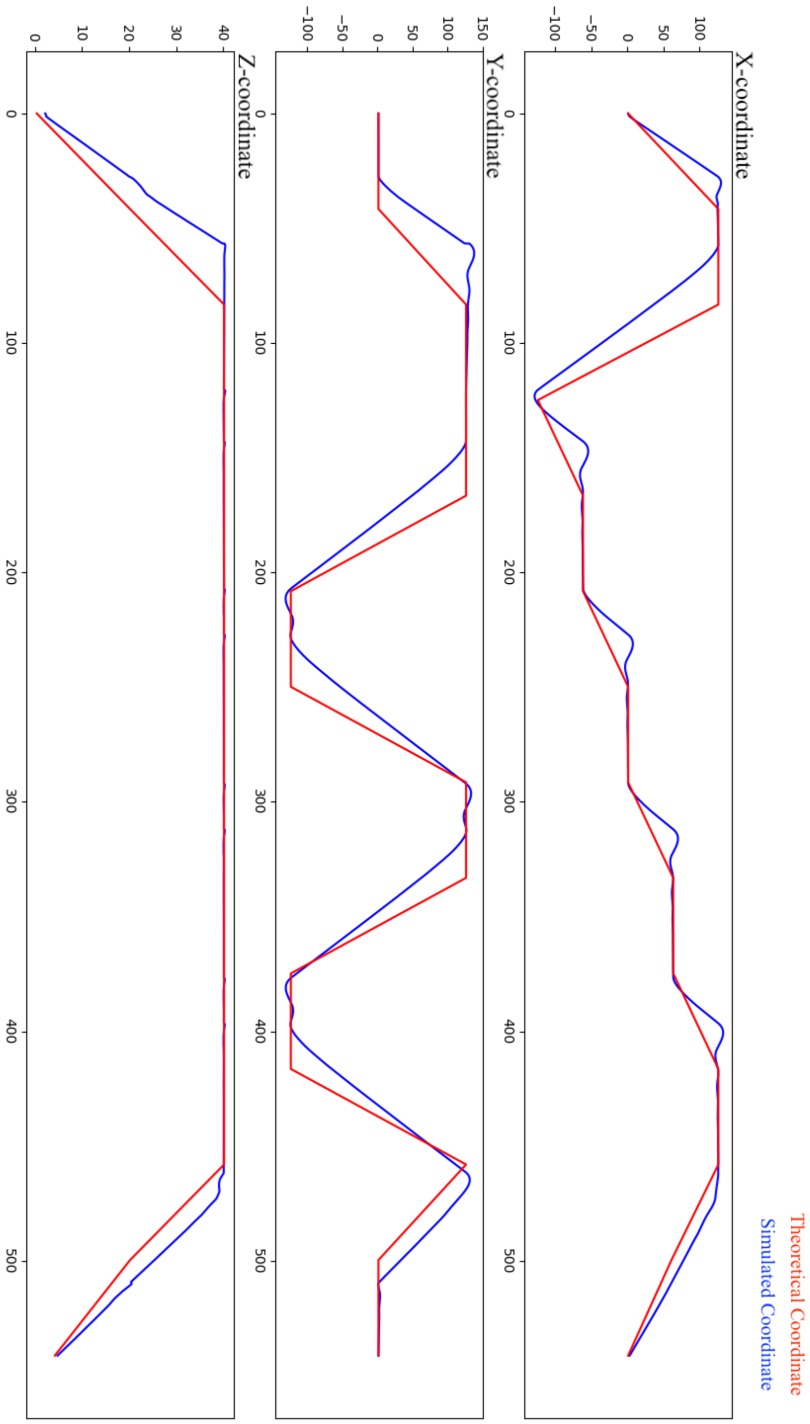


Figure 5.8 Theoretical Coordinates - Simulated Coordinates comparison

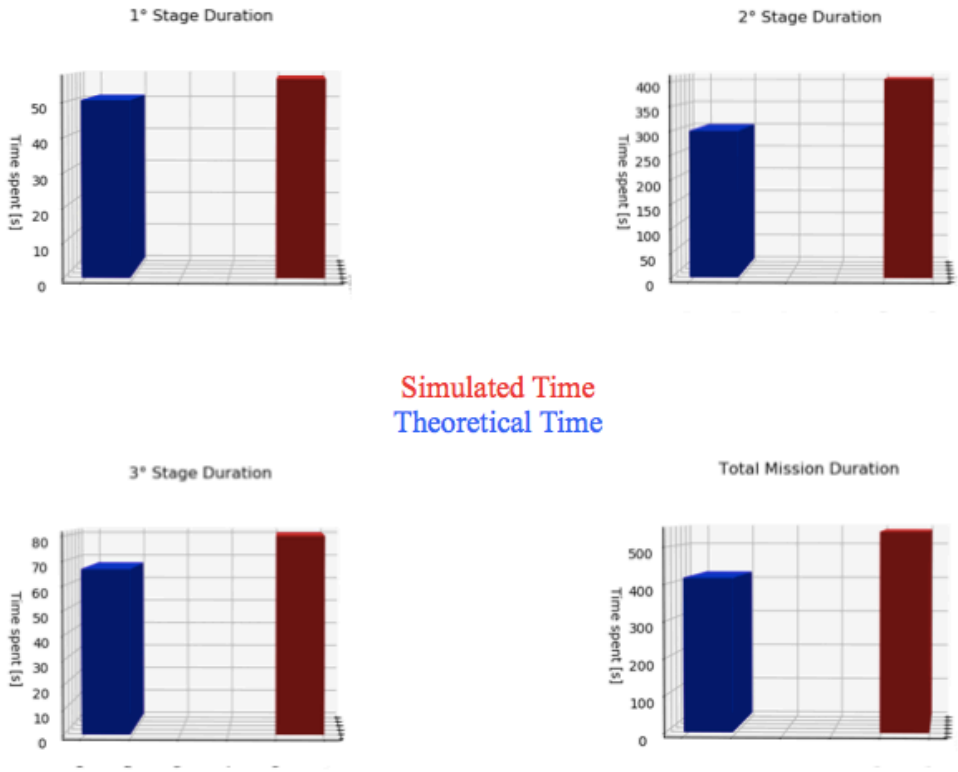


Figure 5.9 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	**Real Time [s]**
1° Stage	50.59	56.44
2° Stage	299.85	404.19
3° Stage	66.71	80.05
Mission Duration	417.15	540.68

Table 5-3 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is $123.53s$, the 29,6% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.4. Plots and Data First Choice Intersection Geographical FP

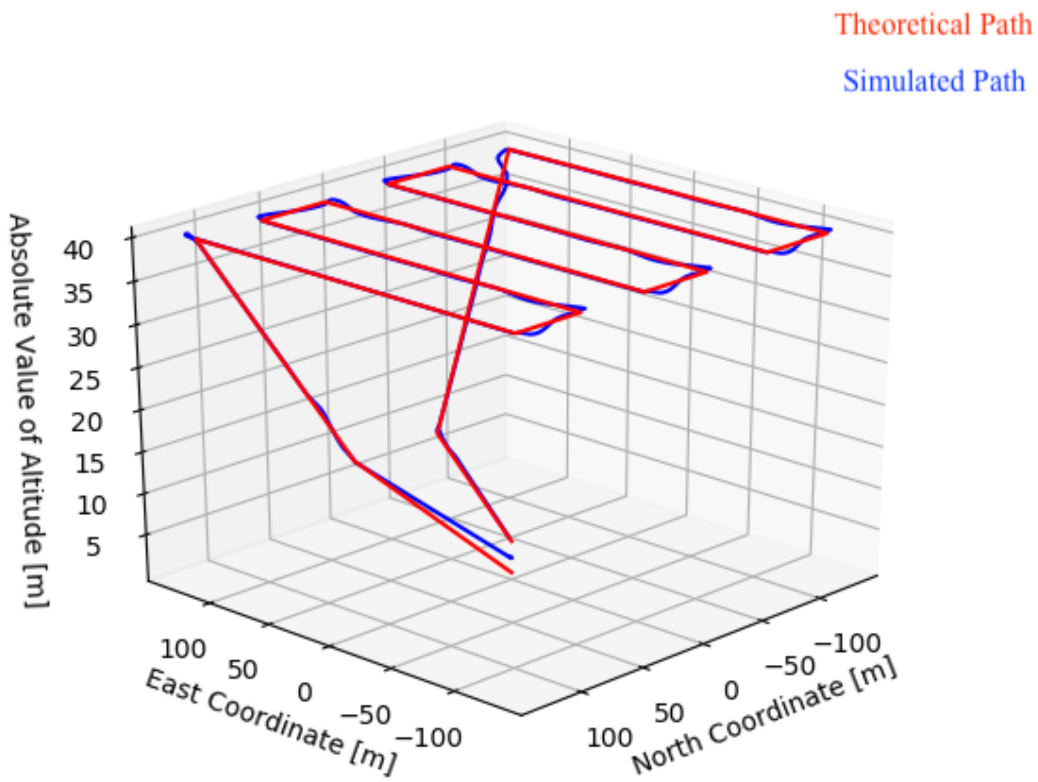


Figure 5.10 Theoretical Path - Simulated Path comparison

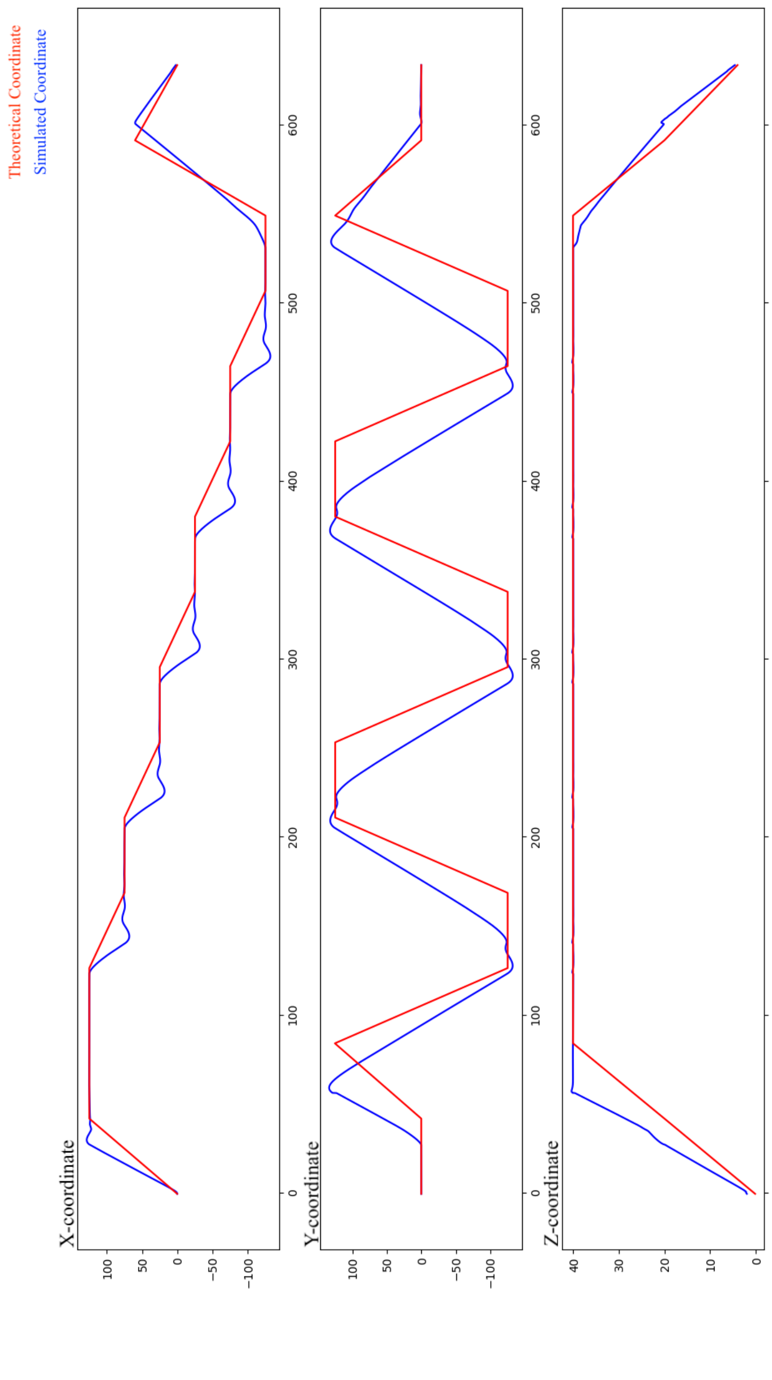


Figure 5.11 Theoretical Coordinates - Simulated Coordinates comparison

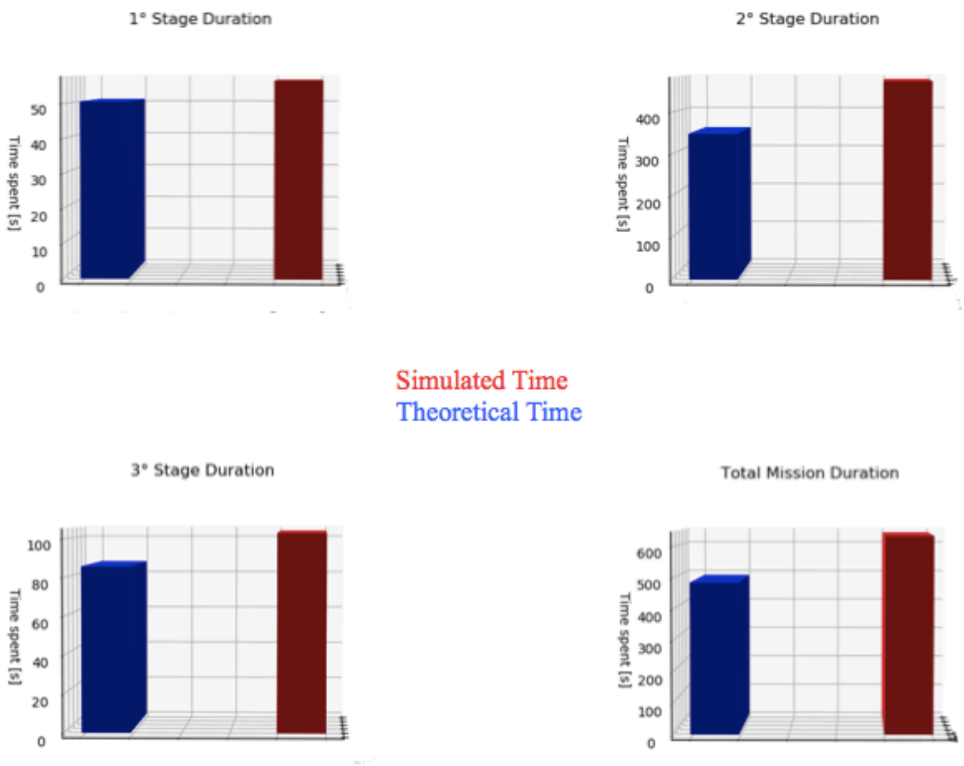


Figure 5.12 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	**Real Time [s]**
1° Stage	50.59	57.16
2° Stage	349.15	485.91
3° Stage	85.94	104.83
Mission Duration	485.68	647.90

Table 5-4 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is 162.22, the 33.4% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.5. Plots and Data Second Choice Intersection Geographical FP

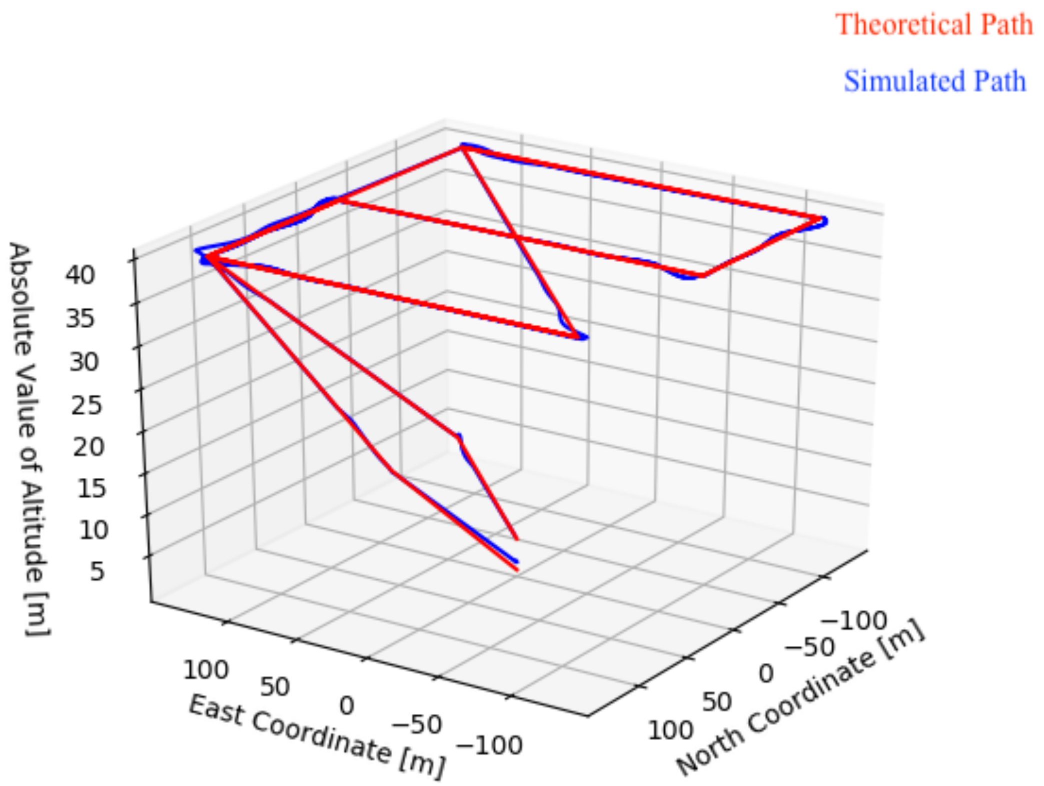


Figure 5.13 Theoretical Path - Simulated Path comparison

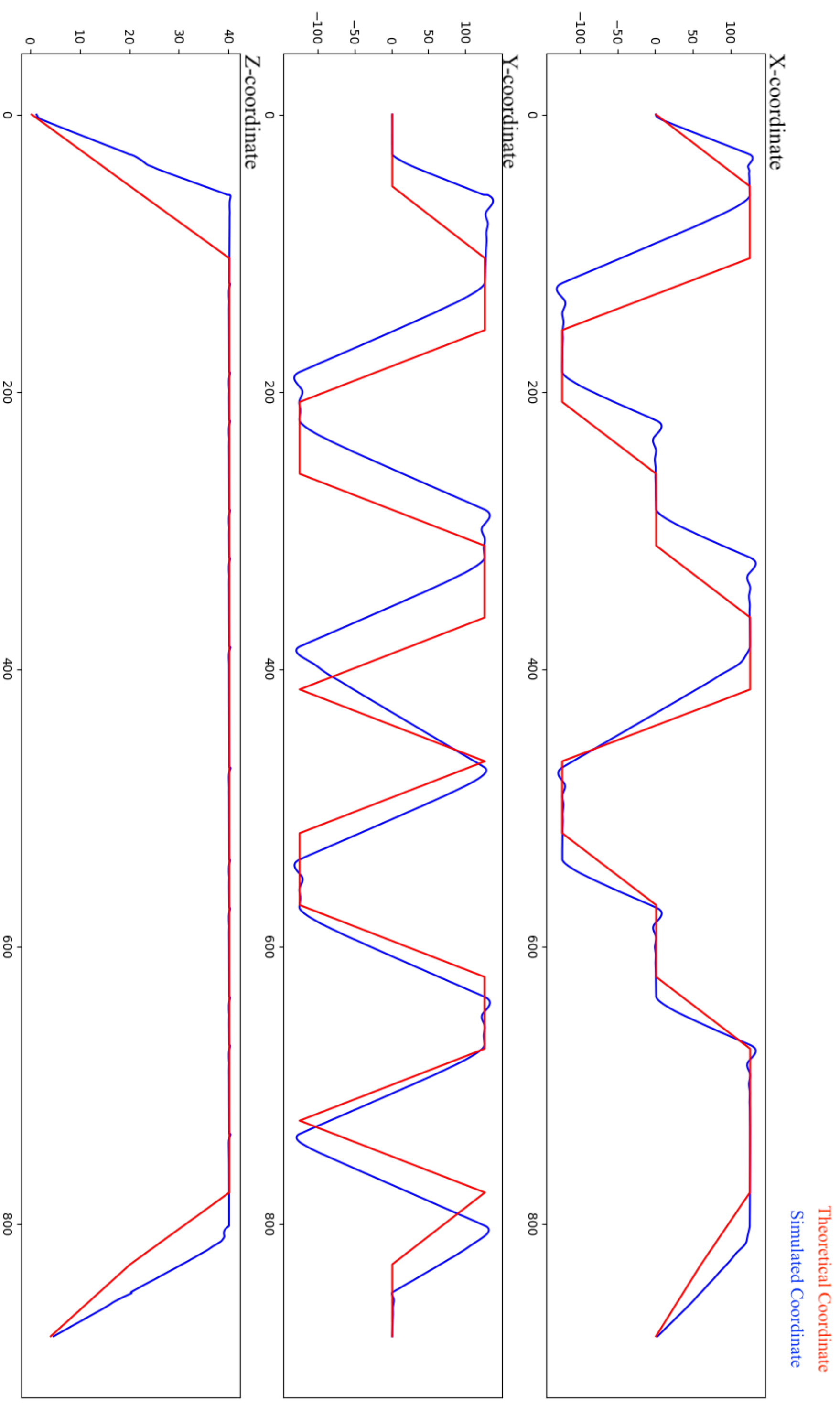


Figure 5.14 Theoretical Coordinates - Simulated Coordinates comparison

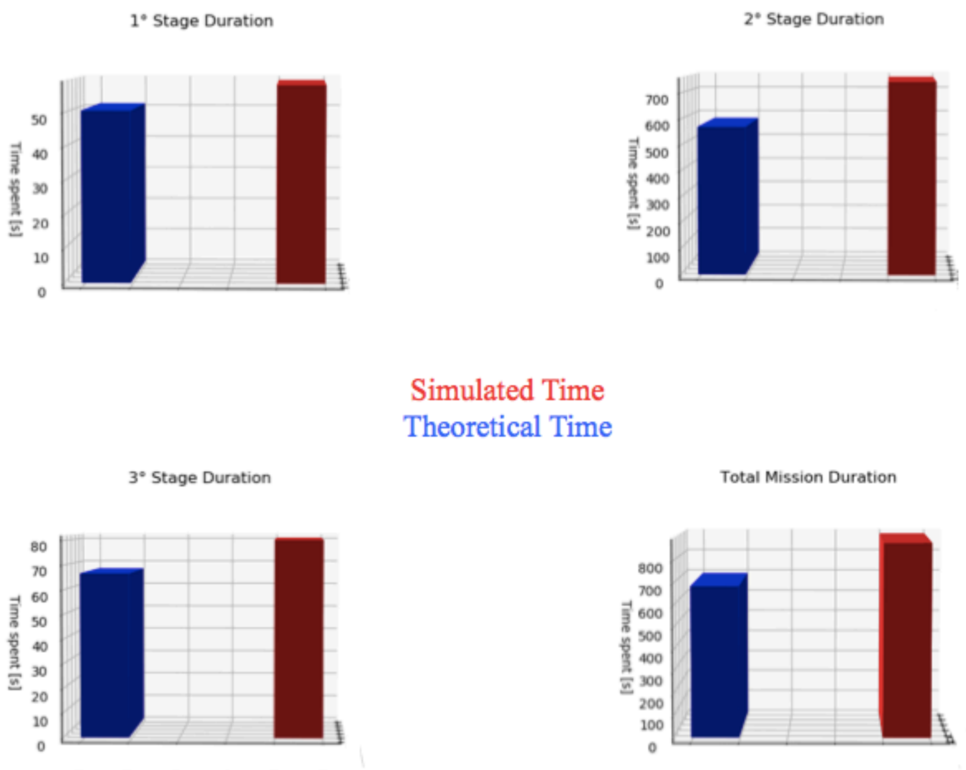


Figure 5.15 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	**Real Time [s]**
1° Stage	50.59	56.44
2° Stage	570.30	742.85
3° Stage	66.77	80.16
Mission Duration	687.66	879.45

Table 5-5 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is $191.79s$, the 27.9% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.6. Plots and Data Third Choice Intersection Geographical FP

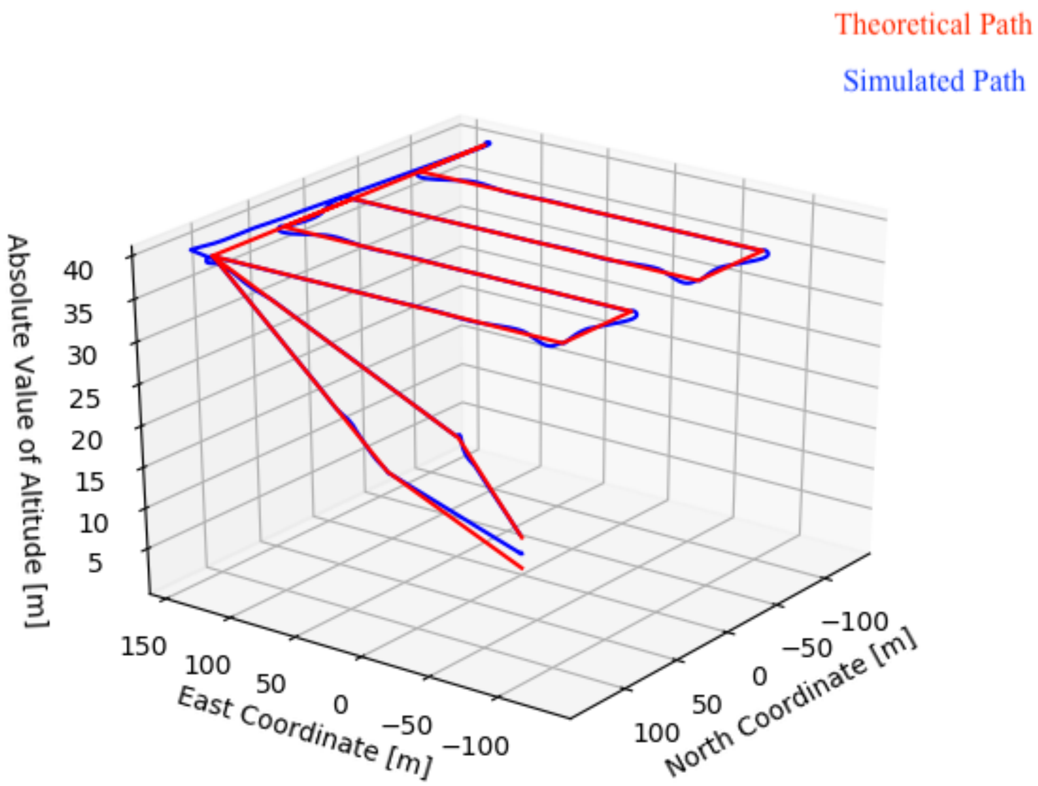


Figure 5.16 Theoretical Path - Simulated Path comparison

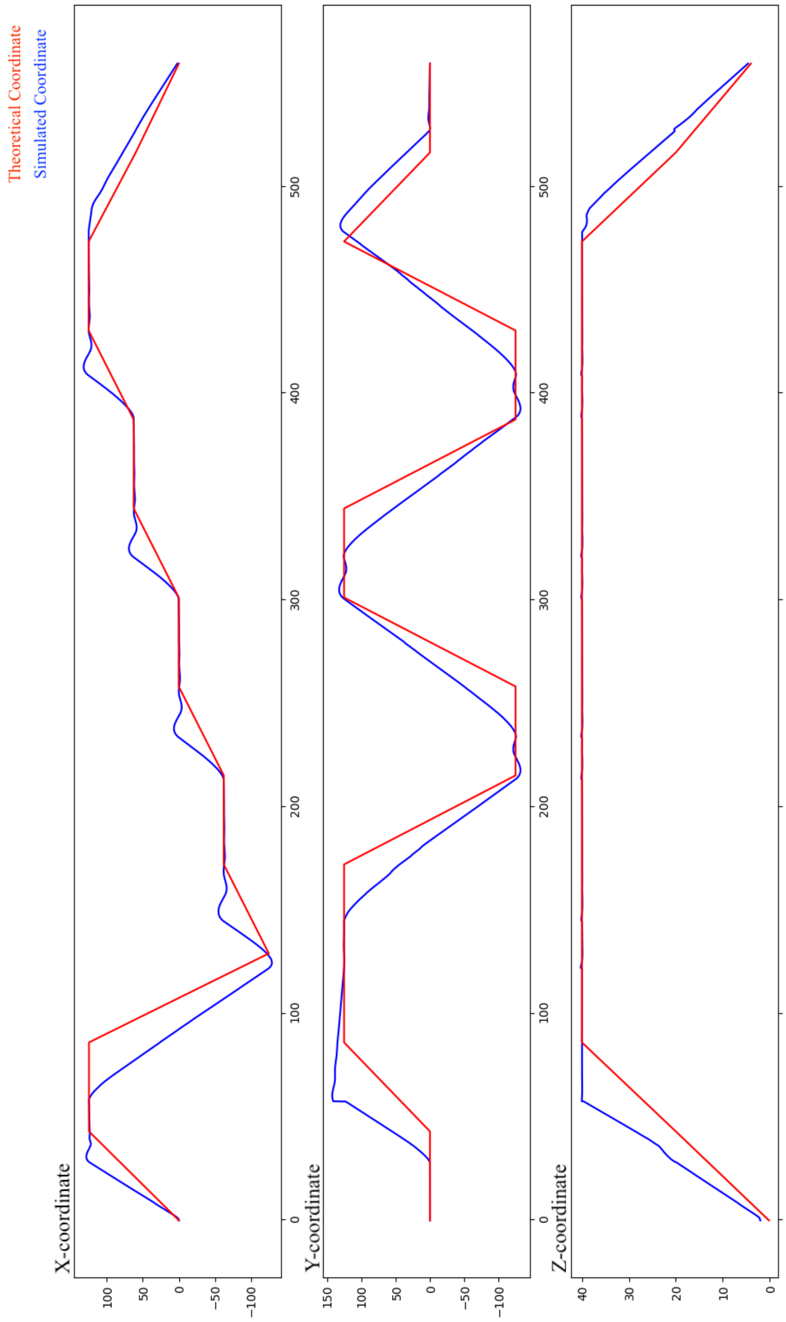


Figure 5.17 Theoretical Coordinates - Simulated Coordinates comparison

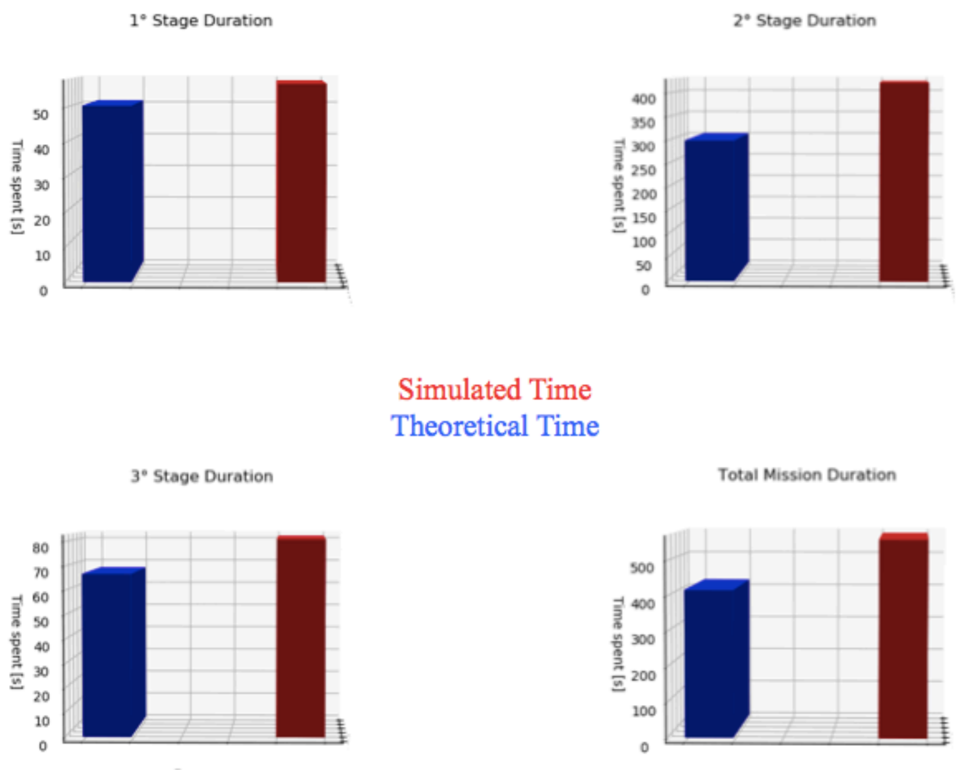


Figure 5.18 Theoretical Time - Simulated Time comparison

TIME TABLE		
	Theoretical Time [s]	**Real Time [s]**
1° Stage	50.59	56.54
2° Stage	299.74	404.21
3° Stage	66.76	80.15
Mission Duration	417.09	540.91

Table 5-6 Theoretical Time Values - Simulated Time Values comparison

The difference between theoretical and simulated time of the global mission is $123.82s$, the 29.7% more than planned duration. In sub-section 5.7 it will be given a reasonable justification for this result.

5.7. General Comments and Results

The first plot in each section puts in evidence that *Carlo* follows more or less the theoretical path uploaded. In both first and second plots, the discrepancies (represented as oscillations in the graphs) are due to the inertia forces acting on the drone. While the drone is moving, its momentum increases. Hence, when the quadricopter reaches the desired waypoint, it has still momentum to be dissipated: this means that it will go a little bit forward before turning. Moreover, drone motion is an accelerated one, it means that the drone will accelerate and decelerate when it is close to the turning points.

To evaluate the theoretical time the uniform motion equation has been used:

$$x = x_0 + v_0 t$$

By isolating t and considering $x_0 = 0$, we obtain:

$$t = \frac{x}{v_0}$$

To evaluate the simulated time, the "time" library has been used. Before taking off, the initial time is stored with the code line (line 833) "$t0 = time.time()$". Then, the time required to execute each stage is evaluated with the code line (line 577) inside the "stage" function "$t\_stage = time.time() - t0$". We can see that the maximum discrepancy between the total theoretical time and the total simulated time is quite large (33.4%). This is due to the approximation of the motion: to have a thoretical time equal to the real one we should use the same motion equations implemented in the simulator. On the contrary, since it is called "theoretical time", it has to be evaluated with the data we have before running the simulation. The FP is all we have and inside the XML file we can only find position and velocity informations.

It has to be noticed that running the single-drone interface with the same FP, the results are every time slightly different. This is due to both simulator behaviour and machine workload. The table below presents the time values obtained for 5 different simulations of the same FP (NED FP, 1° option intersection leg).

TIME TABLE					
	1° Test	**2° Test**	**3° Test**	**4° Test**	**5° Test**
1° Stage	56.45	56.55	56.64	56.75	56.36
2° Stage	474.04	476.04	476.22	474.37	474.86
3° Stage	102.88	102.91	103.27	103.13	102.96
Mission Duration	633.37	635.50	636.13	634.25	634.18

Table 5-7 Simulated Time Values comparison of the same path

Chapter 6

CONCLUSIONS AND RECOMMENDATIONS

6.1. Preliminary Conclusions

Finally, it can be stated that the three main objectives and the extra one have been achieved. A total of six XML Flight Plans as well as the two interfaces have been implemented.

The single-drone interface allows *Carlo* to execute its FP containing four different kind of legs; moreover, six significant graphs can be plotted at the end of each simulation to analyse drone behaviour.

The multiple-drones interface allows *Paolo* and *Francesca* to simultaneously complete their respective FP. A video of the simulation can be found on Youtube via this link: https://youtu.be/ml-OH3kf6Io

All FPs and both Interfaces could be found in GitHub website via this link: https://github.com/Francesco-Rose/Python-Interfaces-for-AirSim

6.2. Area for further studies

Neighborhood-Surveillance-Mission is aimed to monitor the chosen AirSim Enviornment. In the real world, the organization of a mission like this points out several issues that have to be deeply analyzed.

First of all, we have to think to the respect for privacy. It is clear that people do not want a drone equipped with a camera flying out of their houses. It should be created a set of rules that do not allow drones to take photos or videos inside houses; the drone should be able to only monitor the external zone.

Secondly, aerial traffic management should be taken into account. In densely populated areas, a collision between two or more drones could have disastrous effects. Moreover, we have also to consider possible collisions with buildings and other obstacles as well as the running out of batteries. Emergency and Contingency Flight Plans have to be implemented to allow drones safe flight.

Another important aspect that has to be pondered is the need for “good” drones. The more this technology improves, the more evil-minded people take advantages of drones for bad purposes. Hence, the presence of police-drones is becoming more and more important to ensure population safety from any kind of danger.

Finally, drones operating in a real enviornment should be able to adapt themselves to any kind of change of the physical parameters (air pressure, air speed, temperature, air flow, etc). Moreover, wheater conditions must not interfere with both drone performances and the development of the mission.

6.3. Recommendations

This section wants to point out all the “rules and limitations” to use both dynamic interfaces. Some of the limitations refer to the legs functions and to the legs implementation in the XML file; rest of limitations refer to other part of the code.

6.3.1. Recommendations for both interfaces

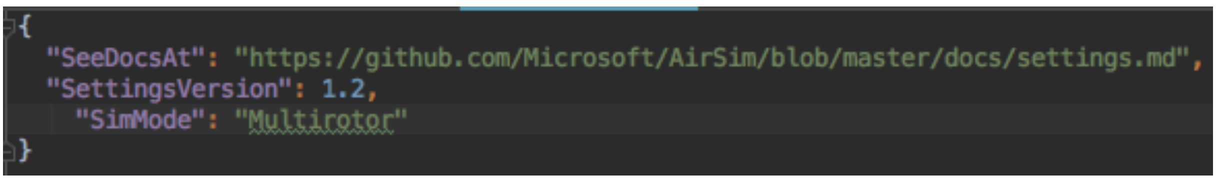
The following list contains the recommendations that have to be followed to correctly use both interfaces.

1. Along the handwriting of the XML files, it should not be left any empty space at the beginning and at the end of the text of both “<Nextlist>” (intersection leg) and “<body>” (iterative leg). White spaces are only allowed to separate the name of the points inside these lists.
2. In the XML file, for a *scan leg*, the points tags that delimit the scan area have to be named as “<point1>, <point2>, <point3>, <point4>” and not as “<point>” (RAISE+ build the *scan leg* with <point>). Moreover, the attribute’s name and the attribute’s value that identify the leg type have to be written in the form “xsi_type = fp_TFLeg” and not as “xsi:type = fp:TFLeg”. This is due to the presence of colons that is troublesome to Python while searching for the leg’s type in the parsed XML.
3. The organization of the scan path has to be carefully implemented. The scan area has to be a square centered in the reference-frame-system origin or a square corresponding to one of the four quadrants formed by the Cartesian plane; its sides have to be parrallel to the NED axis. Moreover, the starting point coordinates of the scan have to be written inside the “<point1>” element. It has also to be noticed that the short section of the scan will follow the x-axis.
4. The dynamic interface as well as all its functions is independent from the number of stages, number of legs, waypoints selected, etc. Take off and landing are automatically executed and the landing point will be the same of the take off one. To implement the FP, it has to be remembered that we can use only the four legs described in section 2.4.2.

6.3.2. Recommendations for Single-Drone interface

To perfectly execute landing, the last point of the FP has to be above the take off point. This is due to the utilization of the “moveByVelocityZAsync” function that allows a vertical movement, increasing or decreasing the altitude.

Figure 6.1 shows the AirSim’s file “settings.json” to be used for the single-drone interface.



```
{
  "SeeDocsAt": "https://github.com/Microsoft/AirSim/blob/master/docs/settings.md",
  "SettingsVersion": 1.2,
    "SimMode": "Multirotor"
}
```

Figure 6.1 Setting.json for Single-Drone Interface

6.3.3. Recommendations for Multiple-Drones interface

The following list contains the recommendations that have to be followed to correctly use the multiple-drones interface.

1. The FPs designer has to carefully select the waypoints covered by the drones to avoid collisions between drones or with obstacles.
2. Figure 6.2 shows the AirSim's file "settings.json" to be used for the multiple-drones interface. The starting point of each drone can be manually set inserting the desired NED coordinates.

```
{
  "SeeDocsAt": "https://github.com/Microsoft/AirSim/blob/master/docs/settings.md",
  "SettingsVersion": 1.2,
    "SimMode": "Multirotor",

    "Vehicles": {
        "Drone1": {
          "VehicleType": "SimpleFlight",
          "X": 0, "Y": 0, "Z": -2,
      "Yaw": -180
        },
        "Drone2": {
          "VehicleType": "SimpleFlight",
          "X": 2, "Y": 0, "Z": -2
        }

    }
}
```

Figure 6.2 Setting.json for Multiple-Drones Interface

3. The multiple-drones interface does not allow us to put an Iterative Leg or an Intersection Leg in the path to be iterated for an iterative leg; the multiple-drones interface does not allow us to put an Iterative Leg or an Intersection Leg as possible choice to be selected for an intersection leg.
4. It has to be implemented at least one FP for each drone that is going to be used.

Chapter 7
BIBLIOGRAPHY

The bibliography has been built with “Mendeley Desktop” Software.

[1] 19.7. xml.etree.ElementTree — The ElementTree XML API — Python 2.7.16 documentation. (n.d.). Retrieved October 2, 2019, from https://docs.python.org/2/library/xml.etree.elementtree.html

[2] 20.5. xml.etree.ElementTree — The ElementTree XML API — Python 3.5.7 documentation. (n.d.). Retrieved October 2, 2019, from https://docs.python.org/3.5/library/xml.etree.elementtree.html#xml.etree.ElementTree.Element

[3] A Brief History of Markup & XML. (n.d.). Retrieved October 2, 2019, from https://codepunk.io/a-brief-history-of-markup-and-xml/

[4] AirSim APIs — airsim documentation. (n.d.). Retrieved October 2, 2019, from https://airsim-fork.readthedocs.io/en/docs/apis.html

[5] Allen, J. D., & Unicode Consortium. (2007). *The Unicode standard 5.0*. Addison-Wesley. Retrieved from https://www.w3.org/TR/xml/

[6] An easy introduction to 3D plotting with Matplotlib. (n.d.). Retrieved October 5, 2019, from https://towardsdatascience.com/an-easy-introduction-to-3d-plotting-with-matplotlib-801561999725

[7] An Intro to Threading in Python – Real Python. (n.d.). Retrieved November 22, 2019, from https://realpython.com/intro-to-python-threading/

[8] Beri, M. (2018). *Python 3*. (F. Brivio, Ed.). Monza (MI): Apogeo.

[9] Come lavorare con i Documenti di Excel usando Python. (n.d.). Retrieved October 8, 2019, from https://code.tutsplus.com/it/tutorials/how-to-work-with-excel-documents-using-python--cms-25698

[10] Illera, T. M. (2014). Flight Plan Documentation, (March), 0–120.

[11] Image APIs - AirSim. (n.d.). Retrieved October 2, 2019, from https://microsoft.github.io/AirSim/docs/image_apis/

[12] Processing XML in Python with ElementTree - Eli Bendersky’s website. (n.d.). Retrieved October 2, 2019, from https://eli.thegreenplace.net/2012/03/15/processing-xml-in-python-with-elementtree/

[13] Python – Read XML file (DOM Example) – Mkyong.com. (n.d.). Retrieved October 2, 2019, from https://www.mkyong.com/python/python-read-xml-file-dom-example/

[14] Python time clock() Method - Tutorialspoint. (n.d.). Retrieved October 5, 2019, from https://www.tutorialspoint.com/python/time_clock.htm

[15] Python XML with ElementTree (article) - DataCamp. (n.d.). Retrieved October 2, 2019, from https://www.datacamp.com/community/tutorials/python-xml-elementtree

[16] Tutorial 1: Create a simple XLSX file — XlsxWriter Documentation. (n.d.). Retrieved October 8, 2019, from https://xlsxwriter.readthedocs.io/tutorial01.html

[17] Unreal Engine. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Unreal_Engine

[18] Welcome to AirSim — airsim documentation. (n.d.). Retrieved October 2, 2019, from https://airsim-fork.readthedocs.io/en/docs/#

[19] What is Unreal Engine 4. (n.d.). Retrieved October 2, 2019, from

https://www.unrealengine.com/en-US/what-is-unreal-engine-4
[20] World Wide Web Consortium. (n.d.). Retrieved from https://en.wikipedia.org/wiki/World_Wide_Web_Consortium
[21] XML. (n.d.). Retrieved from https://en.wikipedia.org/wiki/XML
[22] XML Syntax. (n.d.). Retrieved October 2, 2019, from https://www.w3schools.com/xml/xml_syntax.asp
[23] XPath Support in ElementTree. (n.d.). Retrieved October 2, 2019, from http://effbot.org/zone/element-xpath.htm

ANNEX

All FPs and both Interfaces could be found in GitHub website via this link:
https://github.com/Francesco-Rose/Python-Interfaces-for-AirSim

- **SINGLE DRONE'S FLIGHT PLAN (NED Coordinates).**

```
<FlightPlan>

    <Locale>
        <speedUnits>ms</speedUnits>
        <altitudeUnits>m</altitudeUnits>
        <distanceUnits>m</distanceUnits>
        <decimalSeparator>.</decimalSeparator>
        <groupSeparator/>
    </Locale>

    <MainFP>
        <name>NEIGHBORHOOD FLIGHT PLAN.</name>
        <description>Neighborhood Surveillance Mission.</description>

        <stages>

            <stage id="First Part">
                <leg id="zero-point" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N0</name>
                        <north_coordinate>125</north_coordinate>
                        <east_coordinate>0</east_coordinate>
                        <altitude>-20</altitude>
                        <speed>5</speed>
                        <next>first-point</next>
                    </dest>
                </leg>
                <leg id="first-point" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N1</name>
                        <north_coordinate>125</north_coordinate>
                        <east_coordinate>125</east_coordinate>
                        <altitude>-40</altitude>
                        <speed>5</speed>
                        <next>second-point</next>
                    </dest>
                </leg>
            </stage>
```

```xml
<stage id="Intersection Leg Part">
    <leg id="second-point" xsi_type="fp_IntersectionLeg">
        <nextList>third-point-a third-point-b third-point-c</nextList>
    </leg>
        <leg id="third-point-a" xsi_type="fp_Scan">
            <dest>
                <coordinates>0 0</coordinates>
            </dest>
            <trackseparation>50</trackseparation>
            <area>
                <point1>125 -125</point1>
                <point2>-125 -125</point2>
                <point3>-125 125</point3>
                <point4>125 125</point4>
            </area>
            <speed>5</speed>
            <altitude>-40</altitude>
        </leg>
        <leg id="third-point-b" xsi_type="fp_IterativeLeg">
            <next>fourth-point</next>
            <body>third-point-b-one third-point-b-two</body>
            <upperBound>2</upperBound>
            <first>third-point-b-one</first>
            <last>third-point-b-two</last>
        </leg>
            <leg id="third-point-b-one" xsi_type="fp_TFLeg">
                <dest>
                    <name>N3B1</name>
                    <north_coordinate>-125</north_coordinate>
                    <east_coordinate>125</east_coordinate>
                    <altitude>-40</altitude>
                    <speed>5</speed>
                    <next>Home1</next>
                </dest>
            </leg>
```

```
            <leg id="third-point-b-two" xsi_type="fp_Scan">
                <dest>
                    <coordinates>0 0</coordinates>
                </dest>
                <trackseparation>125</trackseparation>
                <area>
                    <point1>-125 -125</point1>
                    <point2>-125 -125</point2>
                    <point3>125 -125</point3>
                    <point4>125 125</point4>
                </area>
                <speed>5</speed>
                <altitude>-40</altitude>
            </leg>
            <leg id="fourth-point" xsi_type="fp_TFLeg">
                <dest>
                    <name>N4</name>
                    <north_coordinate>125</north_coordinate>
                    <east_coordinate>125</east_coordinate>
                    <altitude>-40</altitude>
                    <speed>5</speed>
                    <next>Home1</next>
                </dest>
            </leg>
        <leg id="third-point-c" xsi_type="fp_Scan">
            <dest>
                <coordinates>0 0</coordinates>
            </dest>
            <trackseparation>62.5</trackseparation>
            <area>
                <point1>-125 125</point1>
                <point2>-125 -125</point2>
                <point3>125 -125</point3>
                <point4>125 125</point4>
            </area>
            <speed>5</speed>
            <altitude>-40</altitude>
        </leg>
    <initialLegs> second-point </initialLegs>
    <finalLegs> third-point-a third-point-b third-point-c </finalLegs>
</stage>
```

```xml
            <stage id="Go Home Part">
                <leg id="Home1" xsi_type="fp_TFLeg">
                    <dest>
                        <name>Home1</name>
                        <north_coordinate>60</north_coordinate>
                        <east_coordinate>0</east_coordinate>
                        <altitude>-20</altitude>
                        <speed>4</speed>
                        <next>Home2</next>
                    </dest>
                </leg>
                <leg id="Home2" xsi_type="fp_TFLeg">
                    <dest>
                        <name>Home2</name>
                        <north_coordinate>0</north_coordinate>
                        <east_coordinate>0</east_coordinate>
                        <altitude>-4</altitude>
                        <speed>2</speed>
                    </dest>
                </leg>
            </stage>

        </stages>

    </MainFP>

</FlightPlan>
```

- **SINGLE DRONE'S FLIGHT PLAN (Geographical Coordinates)**


```xml
<FlightPlan>

    <Locale>
        <speedUnits>ms</speedUnits>
        <altitudeUnits>m</altitudeUnits>
        <trackseparationUnits>m</trackseparationUnits>
        <distanceUnits>°</distanceUnits>
        <decimalSeparator>.</decimalSeparator>
        <groupSeparator/>
    </Locale>

    <MainFP>
        <name>NEIGHBORHOOD FLIGHT PLAN.</name>
        <description>Neighborhood Surveillance Mission.</description>

        <stages>

            <stage id="First Part">
                <leg id="zero-point" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N0</name>
                        <latitude>47.64260464285712</latitude>
                        <longitude>-122.140365</longitude>
                        <altitude>143.199297198</altitude>
                        <speed>5</speed>
                        <next>first-point</next>
                    </dest>
                </leg>
                <leg id="first-point" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N1</name>
                        <latitude>47.64260464285712</latitude>
                        <longitude>-122.1386985308925</longitude>
                        <altitude>163.199297198</altitude>
                        <speed>5</speed>
                        <next>second-point</next>
                    </dest>
                </leg>
            </stage>
```

```xml
<stage id="Intersection Leg Part">
    <leg id="second-point" xsi_type="fp_IntersectionLeg">
        <nextList>third-point-a third-point-b third-point-c</nextList>
    </leg>
        <leg id="third-point-a" xsi_type="fp_Scan">
            <dest>
                <coordinates>0 0</coordinates>
            </dest>
            <trackseparation>50</trackseparation>
            <area>
                <point1>47.64260464285712 -122.14203146910751</point1>
                <point2>47.640360097142874 -122.14203146910751</point2>
                <point3>47.640360097142874 -122.1386985308925</point3>
                <point4>47.64260464285712 -122.1386985308925</point4>
            </area>
            <speed>5</speed>
            <altitude>163.199297198</altitude>
        </leg>
        <leg id="third-point-b" xsi_type="fp_IterativeLeg">
            <next>fourth-point</next>
            <body>third-point-b-one third-point-b-two</body>
            <upperBound>2</upperBound>
            <first>third-point-b-one</first>
            <last>third-point-b-two</last>
        </leg>
            <leg id="third-point-b-one" xsi_type="fp_TFLeg">
                <dest>
                    <name>N3B1</name>
                    <latitude>47.640360097142874</latitude>
                    <longitude>-122.1386985308925</longitude>
                    <altitude>163.199297198</altitude>
                    <speed>5</speed>
                    <next>Home1</next>
                </dest>
            </leg>
```

```
            <leg id="third-point-b-two" xsi_type="fp_Scan">
                <dest>
                    <coordinates>0 0</coordinates>
                </dest>
                <trackseparation>125</trackseparation>
                <area>
                    <point1>47.640360097142874 -122.14203146910751</point1>
                    <point2>47.640360097142874 -122.1386985308925</point2>
                    <point3>47.64260464285712 -122.14203146910751</point3>
                    <point4>47.64260464285712 -122.1386985308925</point4>
                </area>
                <speed>5</speed>
                <altitude>163.199297198</altitude>
            </leg>
            <leg id="fourth-point" xsi_type="fp_TFLeg">
                <dest>
                    <name>N4</name>
                    <latitude>47.64260464285712</latitude>
                    <longitude>-122.1386985308925</longitude>
                    <altitude>163.199297198</altitude>
                    <speed>5</speed>
                    <next>Home1</next>
                </dest>
            </leg>
        <leg id="third-point-c" xsi_type="fp_Scan">
            <dest>
                <coordinates>0 0</coordinates>
            </dest>
            <trackseparation>62.5</trackseparation>
            <area>
                <point1>47.640360097142874 -122.1386985308925</point1>
                <point2>47.640360097142874 -122.14203146910751</point2>
                <point3>47.64260464285712 -122.14203146910751</point3>
                <point4>47.64260464285712 -122.1386985308925</point4>
            </area>
            <speed>5</speed>
            <altitude>163.199297198</altitude>
        </leg>
    <initialLegs> second-point </initialLegs>
    <finalLegs> third-point-a third-point-b third-point-c </finalLegs>
</stage>
```

```xml
            <stage id="Go Home Part">
                <leg id="Home1" xsi_type="fp_TFLeg">
                    <dest>
                        <name>Home1</name>
                        <latitude>47.642021060971416</latitude>
                        <longitude>-122.140365</longitude>
                        <altitude>143.199297198</altitude>
                        <speed>4</speed>
                        <next>Home2</next>
                    </dest>
                </leg>
                <leg id="Home2" xsi_type="fp_TFLeg">
                    <dest>
                        <name>Home2</name>
                        <latitude>47.64148237</latitude>
                        <longitude>-122.140365</longitude>
                        <altitude>127.199297198</altitude>
                        <speed>2</speed>
                    </dest>
                </leg>
            </stage>

        </stages>

    </MainFP>

</FlightPlan>
```

- **SINGLE DRONE'S PYTHON CODE**

```python
# Author of the Code: Francesco Rose
# Master Thesis Project, Universitat Politecnica de la Catalunya
# Dynamic Interface AirSim/XML Fight Plan

import airsim

import pprint
import time
from math import *

import xlsxwriter

import xml.etree.ElementTree as ET

from mpl_toolkits import mplot3d
import numpy as np
import matplotlib.pyplot as plt
import openpyxl

from geographiclib import geodesic


# --------------------------- Creation of the Functions ----------

# Get_Data_to_Plot_drone's_track Function

def get_data(state):
    global column
    column += 1
    n_c = state.kinematics_estimated.position.x_val
    e_c = state.kinematics_estimated.position.y_val
    h = state.kinematics_estimated.position.z_val
    worksheet.write(0, column, n_c)
    worksheet.write(1, column, e_c)
    worksheet.write(2, column, h)


```

```python
# Check_Position Function

def check_position(list1):
    state = client.getMultirotorState()
    p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]
    distance = sqrt((list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])
    Time_For_Each_Leg.append(time_required)
    if LOG == 'ON':
        a = True
        while a:
            if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (list1[1] + 2) and \
                    (list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                a = False
            else:
                get_data(state)
                time.sleep(0.1)
                state = client.getMultirotorState()
                p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
    else:
        time.sleep(time_required*0.9)
        a = True
        while a:
            if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (list1[1] + 2) and \
                    (list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                a = False
            else:
                time.sleep(0.1)
                state = client.getMultirotorState()
                p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]


# To_Fix_Leg Function

def tf_leg(n_coord, e_coord, alt, speed):
    return client.moveToPositionAsync(n_coord, e_coord, alt, speed)
```

```
# Scan_Leg Function

def scan_leg(ts, f_p_n, f_p_e, v, h, i):
    client.moveToPositionAsync(f_p_n, f_p_e, h, v)
    list = [f_p_n, f_p_e, h, v]
    North_Coordinates2.append(list[0])
    East_Coordinates2.append(list[1])
    Altitudes2.append(list[2])
    Speeds.append(list[3])
    check_position(list)
    if (f_p_e < 0 < f_p_n) or (f_p_n > 0 and f_p_e == 0):
        if i % 2 == 0:
            p = []
            s = []
            t = []
            q = []
            for num in range(1, i, 2):
                sott1 = num * ts
                a = f_p_n - sott1
                P = (a, f_p_e, h, v)
                p.append(P)
                S = (a, f_p_e + ll, h, v)
                s.append(S)
            for mol in range(2, i+1, 2):
                sott2 = mol * ts
                b = f_p_n - sott2
                T = (b, f_p_e + ll, h, v)
                t.append(T)
                Q = (b, f_p_e, h, v)
                q.append(Q)
            Total = []
            for n in range(len(q)):
                total = [p[n], s[n], t[n], q[n]]
                Total.append(total)
            Totale = []
            for n in range(len(Total)):
                for a in range(4):
                    totale = Total[n][a]
                    Totale.append(totale)
            print(Totale)
            for n in range(len(Totale)):
                client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
                list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
                North_Coordinates2.append(list1[0])
```

```
                East_Coordinates2.append(list1[1])
                Altitudes2.append(list1[2])
                Speeds.append(list1[3])
                check_position(list1)
        else:
            p = []
            s = []
            t = []
            q = []
            for num in range(1, i+1, 2):
                sott1 = num * ts
                a = f_p_n - sott1
                P = (a, f_p_e, h, v)
                p.append(P)
                S = (a, f_p_e + ll, h, v)
                s.append(S)
            for mol in range(2, i+2, 2):
                sott2 = mol * ts
                b = f_p_n - sott2
                T = (b, f_p_e + ll, h, v)
                t.append(T)
                Q = (b, f_p_e, h, v)
                q.append(Q)
            Total = []
            for n in range(len(s)):
                total = [p[n], s[n], t[n], q[n]]
                Total.append(total)
            Totale = []
            for n in range(len(Total)):
                for a in range(4):
                    totale = Total[n][a]
                    Totale.append(totale)
            print(Totale)
            limit = len(Totale)-2
            for n in range(limit):
                client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
                list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
                North_Coordinates2.append(list1[0])
                East_Coordinates2.append(list1[1])
                Altitudes2.append(list1[2])
                Speeds.append(list1[3])
                check_position(list1)
    elif (f_p_n > 0 and f_p_e > 0) or (f_p_n == 0 and f_p_e > 0):
        if i % 2 == 0:
```

```python
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i, 2):
        sott1 = num * ts
        a = f_p_n - sott1
        P = (a, f_p_e, h, v)
        p.append(P)
        S = (a, f_p_e - ll, h, v)
        s.append(S)
    for mol in range(2, i+1, 2):
        sott2 = mol * ts
        b = f_p_n - sott2
        T = (b, f_p_e - ll, h, v)
        t.append(T)
        Q = (b, f_p_e, h, v)
        q.append(Q)
    Total = []
    for n in range(len(q)):
        total = [p[n], s[n], t[n], q[n]]
        Total.append(total)
    Totale = []
    for n in range(len(Total)):
        for a in range(4):
            totale = Total[n][a]
            Totale.append(totale)
    print(Totale)
    for n in range(len(Totale)):
        client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
        list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
        North_Coordinates2.append(list1[0])
        East_Coordinates2.append(list1[1])
        Altitudes2.append(list1[2])
        Speeds.append(list1[3])
        check_position(list1)
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i+1, 2):
        sott1 = num * ts
        a = f_p_n - sott1
```

```
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e - ll, h, v)
            s.append(S)
        for mol in range(2, i+2, 2):
            sott2 = mol * ts
            b = f_p_n - sott2
            T = (b, f_p_e - ll, h, v)
            t.append(T)
            Q = (b, f_p_e, h, v)
            q.append(Q)
        Total = []
        for n in range(len(s)):
            total = [p[n], s[n], t[n], q[n]]
            Total.append(total)
        Totale = []
        for n in range(len(Total)):
            for a in range(4):
                totale = Total[n][a]
                Totale.append(totale)
        print(Totale)
        limit = len(Totale) - 2
        for n in range(limit):
            client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
            list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
            North_Coordinates2.append(list1[0])
            East_Coordinates2.append(list1[1])
            Altitudes2.append(list1[2])
            Speeds.append(list1[3])
            check_position(list1)
elif (f_p_n < 0 < f_p_e) or (f_p_n < 0 and f_p_e == 0):
    if i % 2 == 0:
        p = []
        s = []
        t = []
        q = []
        for num in range(1, i, 2):
            sott1 = num * ts
            a = f_p_n + sott1
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e - ll, h, v)
            s.append(S)
        for mol in range(2, i+1, 2):
```

```
                sott2 = mol * ts
                b = f_p_n + sott2
                T = (b, f_p_e - ll, h, v)
                t.append(T)
                Q = (b, f_p_e, h, v)
                q.append(Q)
            Total = []
            for n in range(len(q)):
                total = [p[n], s[n], t[n], q[n]]
                Total.append(total)
            Totale = []
            for n in range(len(Total)):
                for a in range(4):
                    totale = Total[n][a]
                    Totale.append(totale)
            print(Totale)
            for n in range(len(Totale)):
                client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
                list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
                North_Coordinates2.append(list1[0])
                East_Coordinates2.append(list1[1])
                Altitudes2.append(list1[2])
                Speeds.append(list1[3])
                check_position(list1)
        else:
            p = []
            s = []
            t = []
            q = []
            for num in range(1, i+1, 2):
                sott1 = num * ts
                a = f_p_n + sott1
                P = (a, f_p_e, h, v)
                p.append(P)
                S = (a, f_p_e - ll, h, v)
                s.append(S)
            for mol in range(2, i+2, 2):
                sott2 = mol * ts
                b = f_p_n + sott2
                T = (b, f_p_e - ll, h, v)
                t.append(T)
                Q = (b, f_p_e, h, v)
                q.append(Q)
            Total = []
```

```
        for n in range(len(s)):
            total = [p[n], s[n], t[n], q[n]]
            Total.append(total)
        Totale = []
        for n in range(len(Total)):
            for a in range(4):
                totale = Total[n][a]
                Totale.append(totale)
        limit = len(Totale) - 2
        for n in range(limit):
            client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
            list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
            North_Coordinates2.append(list1[0])
            East_Coordinates2.append(list1[1])
            Altitudes2.append(list1[2])
            Speeds.append(list1[3])
            check_position(list1)
elif (f_p_n < 0 and f_p_e < 0) or (f_p_n == 0 and f_p_e < 0):
    if i % 2 == 0:
        p = []
        s = []
        t = []
        q = []
        for num in range(1, i, 2):
            sott1 = num * ts
            a = f_p_n + sott1
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e + ll, h, v)
            s.append(S)
        for mol in range(2, i+1, 2):
            sott2 = mol * ts
            b = f_p_n + sott2
            T = (b, f_p_e + ll, h, v)
            t.append(T)
            Q = (b, f_p_e, h, v)
            q.append(Q)
        Total = []
        for n in range(len(q)):
            total = [p[n], s[n], t[n], q[n]]
            Total.append(total)
        Totale = []
        for n in range(len(Total)):
            for a in range(4):
```

```
            totale = Total[n][a]
            Totale.append(totale)
    print(Totale)
    for n in range(len(Totale)):
        client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
        list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
        North_Coordinates2.append(list1[0])
        East_Coordinates2.append(list1[1])
        Altitudes2.append(list1[2])
        Speeds.append(list1[3])
        check_position(list1)
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i+1, 2):
        sott1 = num * ts
        a = f_p_n + sott1
        P = (a, f_p_e, h, v)
        p.append(P)
        S = (a, f_p_e + ll, h, v)
        s.append(S)
    for mol in range(2, i+2, 2):
        sott2 = mol * ts
        b = f_p_n + sott2
        T = (b, f_p_e + ll, h, v)
        t.append(T)
        Q = (b, f_p_e, h, v)
        q.append(Q)
    Total = []
    for n in range(len(s)):
        total = [p[n], s[n], t[n], q[n]]
        Total.append(total)
    Totale = []
    for n in range(len(Total)):
        for a in range(4):
            totale = Total[n][a]
            Totale.append(totale)
    print(Totale)
    limit = len(Totale) - 2
    for n in range(limit):
        client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3])
        list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
```

```
                North_Coordinates2.append(list1[0])
                East_Coordinates2.append(list1[1])
                Altitudes2.append(list1[2])
                Speeds.append(list1[3])
                check_position(list1)


# Iterative_Leg Function

def iterative_leg(name_point, num_of_iter, name_next_point):
    l = waypoints_data[name_point]["len_iter_path"]
    iter_path = []
    for n in range(l):
        iter_path.append(waypoints_data[name_point]["name_point_iter_path" + str(n)])
    rip = 0
    while rip < num_of_iter:
        for n in range(len(iter_path)):
            name_point = iter_path[n]
            if waypoints_data[name_point]["leg_type"] == "fp_TFLeg":
                n_coord = waypoints_data[name_point]["n_coord"]
                e_coord = waypoints_data[name_point]["e_coord"]
                alt = waypoints_data[name_point]["altitude"]
                speed = waypoints_data[name_point]["speed"]
                tf_leg(n_coord, e_coord, alt, speed)
                North_Coordinates2.append(n_coord)
                East_Coordinates2.append(e_coord)
                Altitudes2.append(alt)
                Speeds.append(speed)
                list = [n_coord, e_coord, alt, speed]
                check_position(list)
            if waypoints_data[name_point]["leg_type"] == "fp_Scan":
                ts = waypoints_data[name_point]["trackseparation"]
                f_p_n = waypoints_data[name_point]["point1_n"]
                f_p_e = waypoints_data[name_point]["point1_e"]
                v = waypoints_data[name_point]["speed"]
                h = waypoints_data[name_point]["altitude"]
                i = waypoints_data[name_point]["iteration"]
                scan_leg(ts, f_p_n, f_p_e, v, h, i)
            if waypoints_data[name_point]["leg_type"] == "fp_IterativeLeg":
                name_next_point = waypoints_data[name_point]["name_next_point"]
                num_of_iter = waypoints_data[name_point]["num_of_iter"]
                iterative_leg(name_point, num_of_iter, name_next_point)
            if waypoints_data[name_point]["leg_type"] == "fp_IntersectionLeg":
                limit = waypoints_data[name_point]["limit"]
```

```
                option = input("Intersection Leg: to select one of the possibilities, insert a "
                               "number included between 1 and " + str(limit) + ": ")
                option = int(option)
                intersection_leg(name_point, option)
        rip += 1
    n_coord = waypoints_data[name_next_point]["n_coord"]
    e_coord = waypoints_data[name_next_point]["e_coord"]
    h = waypoints_data[name_next_point]["altitude"]
    v = waypoints_data[name_next_point]["speed"]
    tf_leg(n_coord, e_coord, h, v)
    list1 = [n_coord, e_coord, h, v]
    North_Coordinates2.append(list1[0])
    East_Coordinates2.append(list1[1])
    Altitudes2.append(list1[2])
    Speeds.append(list1[3])
    check_position(list1)


# Intersection_Leg Function

def intersection_leg(name_points, option):
    name_point = waypoints_data[name_points]["name_point_possibility" + str(option-1)]
    if waypoints_data[name_point]["leg_type"] == "fp_TFLeg":
        n_coord = waypoints_data[name_point]["n_coord"]
        e_coord = waypoints_data[name_point]["e_coord"]
        alt = waypoints_data[name_point]["altitude"]
        speed = waypoints_data[name_point]["speed"]
        tf_leg(n_coord, e_coord, alt, speed)
        North_Coordinates2.append(n_coord)
        East_Coordinates2.append(e_coord)
        Altitudes2.append(alt)
        Speeds.append(speed)
        list = [n_coord, e_coord, alt, speed]
        check_position(list)
    if waypoints_data[name_point]["leg_type"] == "fp_Scan":
        ts = waypoints_data[name_point]["trackseparation"]
        f_p_n = waypoints_data[name_point]["point1_n"]
        f_p_e = waypoints_data[name_point]["point1_e"]
        v = waypoints_data[name_point]["speed"]
        h = waypoints_data[name_point]["altitude"]
        i = waypoints_data[name_point]["iteration"]
        scan_leg(ts, f_p_n, f_p_e, v, h, i)
    if waypoints_data[name_point]["leg_type"] == "fp_IterativeLeg":
        name_next_point = waypoints_data[name_point]["name_next_point"]
```

```python
        num_of_iter = waypoints_data[name_point]["num_of_iter"]
        iterative_leg(name_point, num_of_iter, name_next_point)
    if waypoints_data[name_point]["leg_type"] == "fp_IntersectionLeg":
        limit = waypoints_data[name_point]["limit"]
        option = input("Intersection Leg: to select one of the possibilities, insert a "
                       "number included between 1 and " + str(limit) + ": ")
        option = int(option)
        intersection_leg(name_point, option)


# Leg Function

def leg(name_point):
    if waypoints_data[name_point]["leg_type"] == "fp_TFLeg":
        n_coord = waypoints_data[name_point]["n_coord"]
        e_coord = waypoints_data[name_point]["e_coord"]
        alt = waypoints_data[name_point]["altitude"]
        speed = waypoints_data[name_point]["speed"]
        tf_leg(n_coord, e_coord, alt, speed)
        North_Coordinates2.append(n_coord)
        East_Coordinates2.append(e_coord)
        Altitudes2.append(alt)
        Speeds.append(speed)
        list = [n_coord, e_coord, alt, speed]
        check_position(list)
    if waypoints_data[name_point]["leg_type"] == "fp_Scan":
        ts = waypoints_data[name_point]["trackseparation"]
        f_p_n = waypoints_data[name_point]["point1_n"]
        f_p_e = waypoints_data[name_point]["point1_e"]
        v = waypoints_data[name_point]["speed"]
        h = waypoints_data[name_point]["altitude"]
        i = waypoints_data[name_point]["iteration"]
        scan_leg(ts, f_p_n, f_p_e, v, h, i)
    if waypoints_data[name_point]["leg_type"] == "fp_IterativeLeg":
        name_next_point = waypoints_data[name_point]["name_next_point"]
        num_of_iter = waypoints_data[name_point]["num_of_iter"]
        iterative_leg(name_point, num_of_iter, name_next_point)
    if waypoints_data[name_point]["leg_type"] == "fp_IntersectionLeg":
        limit = waypoints_data[name_point]["limit"]
        option = input("Intersection Leg: to select one of the possibilities, insert a "
                       "number included between 1 and " + str(limit) + ": ")
        option = int(option)
        intersection_leg(name_point, option)
```

```python

# GetFirstChild_of_a_Stage Function

def get_first_child_stage(name_stage):
    legs = []
    for stages in tree.iter(tag='stage'):
        if stages.attrib['id'] == name_stage:
            for components in stages.iter(tag='leg'):
                name_point = components.attrib['id']
                legs.append(name_point)
    possibilities = []
    for elem in tree.iter(tag='leg'):
        if elem.attrib['xsi_type'] == "fp_IntersectionLeg":
            for choices in elem.iter():
                possibilitie = choices.text
                possibilities = possibilitie.split()
    next = []
    iter_path = []
    for elem in tree.iter(tag='leg'):
        if elem.attrib['xsi_type'] == "fp_IterativeLeg":
            for components in elem.iter():
                if components.tag == "body":
                    iterative_path = components.text
                    iter_path = iterative_path.split()
                elif components.tag == "next":
                    name_next_point = components.text
                    next.append(name_next_point)
    matches1 = []
    for leg1 in legs:
        for possibility in possibilities:
            if leg1 == possibility:
                matches1.append(leg1)
    for n in range(len(matches1)):
        legs.remove(matches1[n])
    matches2 = []
    for leg2 in legs:
        for iter in iter_path:
            if leg2 == iter:
                matches2.append(leg2)
    for n in range(len(matches2)):
        legs.remove(matches2[n])
    matches3 = []
    for leg3 in legs:
        for nex in next:
```

```
                if leg3 == nex:
                    matches3.append(leg3)
        for n in range(len(matches3)):
            legs.remove(matches3[n])
        first_child = legs
        return first_child


# Stage Function

def stage(name_stage):
    name_points = get_first_child_stage(name_stage)
    for n in range(len(name_points)):
        leg(name_points[n])
    t_stage = time.time() - t0
    Time.append(t_stage)
    sum = 0
    for n in range(len(Time_For_Each_Leg)):
        sum = sum + Time_For_Each_Leg[n]
    Theoretical_Time_to_Plot.append(sum)
    del Time_For_Each_Leg[:]


# ----------------------------- Reading and parsing XML File ----------------------------- #

FP = input("Which Flight Plan do you want to use? Please, select 0 for Lat/Long FP or 1 for NED FP: ")

if FP == '0':
    tree = ET.parse('DroneFlightPlan_Neighborhood_Final_Version_LongLat.xml')
elif FP == '1':
    tree = ET.parse('DroneFlightPlan_Neighborhood_Final_Version_meters.xml')

root = tree.getroot()
ET.tostring(root, encoding='utf8').decode('utf8')

LOG = input("To run the code with plots enter ON; to run the code without plots enter OFF: ")

# ------------- Creation of an empty excel file to store points of real path -------------#

data_collection = xlsxwriter.Workbook('Data.xlsx')
worksheet = data_collection.add_worksheet()
worksheet.write(0, 0, "North Coordinates")
worksheet.write(1, 0, "East Coordinates")
worksheet.write(2, 0, "Altitudes")
```

```python
# ------------------------------------ Connection to AirSim ----------------------------------- #

# Connection to AirSim simulator

client = airsim.MultirotorClient()
client.confirmConnection()
client.enableApiControl(True)
client.armDisarm(True)

state = client.getMultirotorState()
s = pprint.pformat(state)
print("state: %s" % s)

x_home = state.kinematics_estimated.position.x_val
y_home = state.kinematics_estimated.position.y_val
z_home = state.kinematics_estimated.position.z_val

lat0 = state.gps_location.latitude
long0 = state.gps_location.longitude
h_geo0 = state.gps_location.altitude

arcWE = geodesic.Geodesic.WGS84.Inverse(lat0, long0-0.5, lat0, long0+0.5)
arcNS = geodesic.Geodesic.WGS84.Inverse(lat0-0.5, long0, lat0+0.5, long0)

lat_conv = 1 / arcNS['s12']
long_conv = 1 / arcWE['s12']

# ----------------- Creation of a dictionary with all waypoints informations --------------#

waypoints_data = {}

if FP == '0':
    for legs in tree.iter(tag='leg'):
        if legs.attrib['xsi_type'] == "fp_TFLeg":
            name_point = legs.attrib['id']
            waypoints_data[name_point] = {}
            leg_type = legs.attrib['xsi_type']
            waypoints_data[name_point]["leg_type"] = leg_type
            for components in legs.iter():
                if components.tag == "latitude":
                    lat = components.text
                    lat = float(lat)
                    if lat == lat0:
```

```
                n_coord = 0
            else:
                n_coord = (lat-lat0)/lat_conv
            waypoints_data[name_point]["n_coord"] = n_coord
        elif components.tag == "longitude":
            long = components.text
            long = float(long)
            if long == long0:
                e_coord = 0
            else:
                e_coord = (long-long0)/long_conv
            waypoints_data[name_point]["e_coord"] = e_coord
        elif components.tag == "altitude":
            altitude = components.text
            altitude = float(altitude)
            alt = z_home - (altitude - h_geo0)
            waypoints_data[name_point]["altitude"] = alt
        elif components.tag == "speed":
            speed = components.text
            speed = float(speed)
            waypoints_data[name_point]["speed"] = speed
if legs.attrib['xsi_type'] == "fp_Scan":
    name_point = legs.attrib['id']
    waypoints_data[name_point] = {}
    leg_type = legs.attrib['xsi_type']
    waypoints_data[name_point]["leg_type"] = leg_type
    for components in legs.iter():
        if components.tag == "trackseparation":
            ts = components.text
            ts = float(ts)
            waypoints_data[name_point]["trackseparation"] = ts
        elif components.tag == "point1":
            f_p = components.text
            f_p = f_p.split()
            lat = float(f_p[0])
            if lat == lat0:
                f_p_n = 0
            else:
                f_p_n = (lat - lat0) / lat_conv
            long = float(f_p[1])
            if long == long0:
                f_p_e = 0
            else:
                f_p_e = (long - long0) / long_conv
```

```
                waypoints_data[name_point]["point1_n"] = f_p_n
                waypoints_data[name_point]["point1_e"] = f_p_e
            elif components.tag == "altitude":
                h = components.text
                h = float(h)
                alt = z_home - (h - h_geo0)
                waypoints_data[name_point]["altitude"] = alt
            elif components.tag == "speed":
                v = components.text
                v = float(v)
                waypoints_data[name_point]["speed"] = v
        l_l = abs(f_p_n) + abs(f_p_e)
        ll = round(l_l)
        waypoints_data[name_point]["iteration"] = int(ll / ts)
    if legs.attrib['xsi_type'] == "fp_IterativeLeg":
        name_point = legs.attrib['id']
        waypoints_data[name_point] = {}
        leg_type = legs.attrib['xsi_type']
        waypoints_data[name_point]["leg_type"] = leg_type
        for components in legs.iter():
            if components.tag == "body":
                iterative_path = components.text
                iter_path = iterative_path.split()
                len_iter_path = len(iter_path)
                waypoints_data[name_point]["len_iter_path"] = len_iter_path
                for n in range(len(iter_path)):
                    waypoints_data[name_point]["name_point_iter_path" + str(n)] = iter_path[n]
            elif components.tag == "next":
                name_next_point = components.text
                waypoints_data[name_point]["name_next_point"] = name_next_point
            elif components.tag == "upperBound":
                num_of_iteration = components.text
                num_of_iter = int(num_of_iteration)
                waypoints_data[name_point]["num_of_iter"] = num_of_iter
    if legs.attrib['xsi_type'] == "fp_IntersectionLeg":
        name_point = legs.attrib['id']
        waypoints_data[name_point] = {}
        leg_type = legs.attrib['xsi_type']
        waypoints_data[name_point]["leg_type"] = leg_type
        for choices in legs.iter():
            possibilities = choices.text
            possibilities = possibilities.split()
            for n in range(len(possibilities)):
                waypoints_data[name_point]["name_point_possibility" + str(n)] = possibilities[n]
```

```
                limit = len(possibilities)
                waypoints_data[name_point]["limit"] = limit
elif FP == '1':
    for legs in tree.iter(tag='leg'):
        if legs.attrib['xsi_type'] == "fp_TFLeg":
            name_point = legs.attrib['id']
            waypoints_data[name_point] = {}
            leg_type = legs.attrib['xsi_type']
            waypoints_data[name_point]["leg_type"] = leg_type
            for components in legs.iter():
                if components.tag == "north_coordinate":
                    n_c = components.text
                    n_coord = float(n_c)
                    waypoints_data[name_point]["n_coord"] = n_coord
                elif components.tag == "east_coordinate":
                    e_c = components.text
                    e_coord = float(e_c)
                    waypoints_data[name_point]["e_coord"] = e_coord
                elif components.tag == "altitude":
                    alt = components.text
                    alt = float(alt)
                    waypoints_data[name_point]["altitude"] = alt
                elif components.tag == "speed":
                    speed = components.text
                    speed = float(speed)
                    waypoints_data[name_point]["speed"] = speed
        if legs.attrib['xsi_type'] == "fp_Scan":
            name_point = legs.attrib['id']
            waypoints_data[name_point] = {}
            leg_type = legs.attrib['xsi_type']
            waypoints_data[name_point]["leg_type"] = leg_type
            for components in legs.iter():
                if components.tag == "trackseparation":
                    ts = components.text
                    ts = float(ts)
                    waypoints_data[name_point]["trackseparation"] = ts
                elif components.tag == "point1":
                    f_p = components.text
                    f_p = f_p.split()
                    f_p_n = float(f_p[0])
                    f_p_e = float(f_p[1])
                    waypoints_data[name_point]["point1_n"] = f_p_n
                    waypoints_data[name_point]["point1_e"] = f_p_e
                elif components.tag == "altitude":
```

```
                    h = components.text
                    h = float(h)
                    waypoints_data[name_point]["altitude"] = h
                elif components.tag == "speed":
                    v = components.text
                    v = float(v)
                    waypoints_data[name_point]["speed"] = v
            l_l = abs(f_p_n) + abs(f_p_e)
            ll = round(l_l)
            waypoints_data[name_point]["iteration"] = int(ll / ts)
        if legs.attrib['xsi_type'] == "fp_IterativeLeg":
            name_point = legs.attrib['id']
            waypoints_data[name_point] = {}
            leg_type = legs.attrib['xsi_type']
            waypoints_data[name_point]["leg_type"] = leg_type
            for components in legs.iter():
                if components.tag == "body":
                    iterative_path = components.text
                    iter_path = iterative_path.split()
                    len_iter_path = len(iter_path)
                    waypoints_data[name_point]["len_iter_path"] = len_iter_path
                    for n in range(len(iter_path)):
                        waypoints_data[name_point]["name_point_iter_path" + str(n)] = iter_path[n]
                elif components.tag == "next":
                    name_next_point = components.text
                    waypoints_data[name_point]["name_next_point"] = name_next_point
                elif components.tag == "upperBound":
                    num_of_iteration = components.text
                    num_of_iter = int(num_of_iteration)
                    waypoints_data[name_point]["num_of_iter"] = num_of_iter
        if legs.attrib['xsi_type'] == "fp_IntersectionLeg":
            name_point = legs.attrib['id']
            waypoints_data[name_point] = {}
            leg_type = legs.attrib['xsi_type']
            waypoints_data[name_point]["leg_type"] = leg_type
            for choices in legs.iter():
                possibilities = choices.text
                possibilities = possibilities.split()
                for n in range(len(possibilities)):
                    waypoints_data[name_point]["name_point_possibility" + str(n)] = possibilities[n]
                limit = len(possibilities)
                waypoints_data[name_point]["limit"] = limit

print(waypoints_data)
```

```
# ---------------------------------- Take Off Part ---------------------------------- #

client.takeoffAsync()
time.sleep(4)

t0 = time.time()

# ----------------------------- Execution of the Flight Plan ---------------------------- #

Time = []
North_Coordinates2 = [x_home]
East_Coordinates2 = [y_home]
Altitudes2 = [z_home]
Speeds = []
Time_For_Each_Leg = []
Theoretical_Time_to_Plot = []

column = 0

Stages = []
for element in root.iter(tag='stage'):
    Stages.append(element.attrib['id'])

for n in range(len(Stages)):
    stage(Stages[n])

# ------------------------------ Landing + Carlo Disarm ------------------------------ #

client.moveByVelocityZAsync(0, 0, z_home, 2, airsim.DrivetrainType.MaxDegreeOfFreedom, airsim.YawMode(False, 0)).join()

client.armDisarm(False)
client.enableApiControl(False)

data_collection.close()

# ------------------------------- Position Plots Part ------------------------------- #

# Real Path vs Theoretical Path Plot

if LOG == 'ON':
    data = openpyxl.load_workbook('Data.xlsx')
    sheet = data['Sheet1']
```

```
    estremo_sx = []
    estremo_dx = []
    estremo_cn = []

    all_columns = sheet.columns

    for tupla in all_columns:
        lista = list(tupla)
        listino = [str(lista[0]), str(lista[1]), str(lista[2])]
        sx = listino[0]
        cn = listino[1]
        dx = listino[2]
        if len(dx) == 18 and len(sx) == 18 and len(cn) == 18:
            e_dx = dx[15:17]
            estremo_dx.append(e_dx)
            e_cn = cn[15:17]
            estremo_cn.append(e_cn)
            e_sx = sx[15:17]
            estremo_sx.append(e_sx)
        elif len(dx) == 19 and len(sx) == 19 and len(cn) == 19:
            e_dx = dx[15:18]
            estremo_dx.append(e_dx)
            e_cn = cn[15:18]
            estremo_cn.append(e_cn)
            e_sx = sx[15:18]
            estremo_sx.append(e_sx)
        elif len(dx) == 20 and len(sx) == 20 and len(cn) == 20:
            e_dx = dx[15:19]
            estremo_dx.append(e_dx)
            e_cn = cn[15:19]
            estremo_cn.append(e_cn)
            e_sx = sx[15:19]
            estremo_sx.append(e_sx)

    estremo_sx.remove('A1')
    estremo_cn.remove('A2')
    estremo_dx.remove('A3')

    Tutti_Punti = []

    for n in range(len(estremo_sx)):
        a = estremo_sx[n]
        c = estremo_cn[n]
        b = estremo_dx[n]
```

```
    multiple_cells = sheet[a:b]
    for row in multiple_cells:
        for cell in row:
            list = cell.value
            Tutti_Punti.append(list)

North_Coordinates1 = []
East_Coordinates1 = []
Altitudes1 = []
l = len(Tutti_Punti)

for n in range(0, l, 3):
    North_Coordinates1.append(Tutti_Punti[n])
    East_Coordinates1.append(Tutti_Punti[n+1])
    Altitudes1.append(-Tutti_Punti[n+2])

fig = plt.figure()
ax = plt.axes(projection='3d')
plt.gca().invert_yaxis()

Altitudes2_abs = []
for n in range(len(Altitudes2)):
    alt = - (Altitudes2[n])
    Altitudes2_abs.append(alt)

ax.plot3D(North_Coordinates1, East_Coordinates1, Altitudes1, color='blue')
ax.plot3D(North_Coordinates2, East_Coordinates2, Altitudes2_abs, color='red')
ax.set_xlabel('North Coordinate [m]')
ax.set_ylabel('East Coordinate [m]')
ax.set_zlabel('Absolute Value of Altitude [m]')


# Real and Theoretical Coordinates vs Time Plot

fig = plt.figure()
ax = plt.axes(projection='3d')

Tot = Time[len(Time)-1]

iteraz1 = len(North_Coordinates1)
iteraz2 = len(North_Coordinates2)
iteraz3 = len(East_Coordinates1)
iteraz4 = len(East_Coordinates2)
iteraz5 = len(Altitudes1)
```

```python
iteraz6 = len(Altitudes2_abs)

x1 = np.linspace(0.0, Tot, iteraz1)
x2 = np.linspace(0.0, Tot, iteraz2)
x3 = np.linspace(0.0, Tot, iteraz3)
x4 = np.linspace(0.0, Tot, iteraz4)
x5 = np.linspace(0.0, Tot, iteraz5)
x6 = np.linspace(0.0, Tot, iteraz6)

y1 = North_Coordinates1
y2 = North_Coordinates2
y3 = East_Coordinates1
y4 = East_Coordinates2
y5 = Altitudes1
y6 = Altitudes2_abs

plt.subplot(3, 1, 1)
plt.plot(x1, y1, 'blue', x2, y2, 'red')

plt.subplot(3, 1, 2)
plt.plot(x3, y3, 'blue', x4, y4, 'red')

plt.subplot(3, 1, 3)
plt.plot(x5, y5, 'blue', x6, y6, 'red')
ax.grid()

# ------------------------------------------- Time Plots Part -------------------------------------------------- #

Real_Time_to_Plot = []
Real_Time_to_Plot.append(Time[0])
for n in range(1, len(Time)):
    Real_Time_to_Plot.append(Time[n]-Time[n-1])
Real_Time_to_Plot.append(Time[len(Time)-1])
print(Real_Time_to_Plot)

Total_Theorical_Time = 0
for n in range(len(Theoretical_Time_to_Plot)):
    Total_Theorical_Time = Total_Theorical_Time + Theoretical_Time_to_Plot[n]
Theoretical_Time_to_Plot.append(Total_Theorical_Time)
print(Theoretical_Time_to_Plot)

# First, Second, Third Stage Time Plot

for n in range(len(Real_Time_to_Plot)-1):
```

```
        fig = plt.figure()
        ax = plt.axes(projection="3d")

        num_bars = 2
        x_pos = [1, 1]
        y_pos = [1, 5]
        z_pos = [0] * num_bars
        x_size = np.ones(num_bars)
        y_size = np.ones(num_bars)
        z_size = [Theoretical_Time_to_Plot[n], Real_Time_to_Plot[n]]

        ax.bar3d(x_pos[0], y_pos[0], z_pos[0], x_size[0], y_size[0], z_size[0], color='blue')
        ax.bar3d(x_pos[1], y_pos[1], z_pos[1], x_size[1], y_size[1], z_size[1], color='red')

        a = str(n+1)

        ax.set_zlabel('Time spent [s]')
        ax.set_title(a + '° Stage Duration')

    # Total Mission Time Plot

    fig = plt.figure()
    ax = plt.axes(projection="3d")

    l1 = len(Theoretical_Time_to_Plot)-1
    l2 = len(Real_Time_to_Plot)-1

    num_bars = 2
    x_pos = [1, 1]
    y_pos = [1, 5]
    z_pos = [0] * num_bars
    x_size = np.ones(num_bars)
    y_size = np.ones(num_bars)
    z_size = [Theoretical_Time_to_Plot[l1], Real_Time_to_Plot[l2]]

    ax.bar3d(x_pos[0], y_pos[0], z_pos[0], x_size[0], y_size[0], z_size[0], color='blue')
    ax.bar3d(x_pos[1], y_pos[1], z_pos[1], x_size[1], y_size[1], z_size[1], color='red')

    ax.set_zlabel('Time spent [s]')
    ax.set_title(' Total Mission Duration')

    plt.show()
```

- **MULTIPLE DRONES' FLIGHT PLAN: 1° DRONE (NED Coordinates)**

```xml
<FlightPlan>

    <Locale>
        <speedUnits>ms</speedUnits>
        <altitudeUnits>m</altitudeUnits>
        <trackseparationUnits>m</trackseparationUnits>
        <distanceUnits>m</distanceUnits>
        <decimalSeparator>.</decimalSeparator>
        <groupSeparator/>
    </Locale>

    <MainFP>
        <name>NEIGHBORHOOD FLIGHT PLAN.</name>
        <description>Neighborhood Surveillance Mission.</description>

        <stages>

            <stage id="First Part">
                <leg id="zero-point" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N0</name>
                        <north_coordinate>125</north_coordinate>
                        <east_coordinate>0</east_coordinate>
                        <altitude>-20</altitude>
                        <speed>5</speed>
                        <next>first-point</next>
                    </dest>
                </leg>
            </stage>

            <stage id="Iterative Leg Part">
                <leg id="first-point" xsi_type="fp_IterativeLeg">
                    <next>third-point</next>
                    <body>second-point-one second-point-two second-point-three second-point-four</body>
                    <upperBound>20</upperBound>
                    <first>second-point-one</first>
                    <last>second-point-four</last>
                </leg>
                    <leg id="second-point-one" xsi_type="fp_TFLeg">
                        <dest>
                            <name>N21</name>
                            <north_coordinate>125</north_coordinate>
                            <east_coordinate>125</east_coordinate>
                            <altitude>-30</altitude>
                            <speed>10</speed>
                            <next>second-point-two</next>
                        </dest>
                    </leg>
                    <leg id="second-point-two" xsi_type="fp_TFLeg">
```

```xml
                <leg id="second-point-two" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N22</name>
                        <north_coordinate>-125</north_coordinate>
                        <east_coordinate>125</east_coordinate>
                        <altitude>-30</altitude>
                        <speed>10</speed>
                        <next>second-point-three</next>
                    </dest>
                </leg>
                <leg id="second-point-three" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N23</name>
                        <north_coordinate>-125</north_coordinate>
                        <east_coordinate>-125</east_coordinate>
                        <altitude>-30</altitude>
                        <speed>10</speed>
                        <next>second-point-four</next>
                    </dest>
                </leg>
                <leg id="second-point-four" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N24</name>
                        <north_coordinate>125</north_coordinate>
                        <east_coordinate>-125</east_coordinate>
                        <altitude>-30</altitude>
                        <speed>10</speed>
                        <next>third-point</next>
                    </dest>
                </leg>
                <leg id="third-point" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N4</name>
                        <north_coordinate>0</north_coordinate>
                        <east_coordinate>0</east_coordinate>
                        <altitude>-30</altitude>
                        <speed>3</speed>
                    </dest>
                </leg>
            </stage>

        </stages>

    </MainFP>

</FlightPlan>
```

- **MULTIPLE DRONES’ FLIGHT PLAN: 1° DRONE (Geographical Coordinates)**

```xml
<FlightPlan>

    <Locale>
        <speedUnits>ms</speedUnits>
        <altitudeUnits>m</altitudeUnits>
        <trackseparationUnits>m</trackseparationUnits>
        <distanceUnits>°</distanceUnits>
        <decimalSeparator>.</decimalSeparator>
        <groupSeparator/>
    </Locale>

    <MainFP>
        <name>NEIGHBORHOOD FLIGHT PLAN.</name>
        <description>Neighborhood Surveillance Mission.</description>

        <stages>

            <stage id="First Part">
                <leg id="zero-point" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N0</name>
                        <latitude>47.64260464285712</latitude>
                        <longitude>-122.140365</longitude>
                        <altitude>143.199297198</altitude>
                        <speed>5</speed>
                        <next>first-point</next>
                    </dest>
                </leg>
            </stage>

            <stage id="Iterative Leg Part">
                <leg id="first-point" xsi_type="fp_IterativeLeg">
                    <next>third-point</next>
                    <body>second-point-one second-point-two second-point-three second-point-four</body>
                    <upperBound>20</upperBound>
                    <first>second-point-one</first>
                    <last>second-point-four</last>
                </leg>
                    <leg id="second-point-one" xsi_type="fp_TFLeg">
                        <dest>
                            <name>N21</name>
                            <latitude>47.64260464285712</latitude>
                            <longitude>-122.1386985308925</longitude>
                            <altitude>153.199297198</altitude>
                            <speed>10</speed>
                            <next>second-point-two</next>
                        </dest>
                    </leg>
                    <leg id="second-point-two" xsi_type="fp_TFLeg">
```

```xml
                <leg id="second-point-two" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N22</name>
                        <latitude>47.640360097142874</latitude>
                        <longitude>-122.1386985308925</longitude>
                        <altitude>153.199297198</altitude>
                        <speed>10</speed>
                        <next>second-point-three</next>
                    </dest>
                </leg>
                <leg id="second-point-three" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N23</name>
                        <latitude>47.640360097142874</latitude>
                        <longitude>-122.14203146910751</longitude>
                        <altitude>153.199297198</altitude>
                        <speed>10</speed>
                        <next>second-point-four</next>
                    </dest>
                </leg>
                <leg id="second-point-four" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N24</name>
                        <latitude>47.64260464285712</latitude>
                        <longitude>-122.14203146910751</longitude>
                        <altitude>153.199297198</altitude>
                        <speed>10</speed>
                        <next>third-point</next>
                    </dest>
                </leg>
                <leg id="third-point" xsi_type="fp_TFLeg">
                    <dest>
                        <name>N4</name>
                        <latitude>47.64148237</latitude>
                        <longitude>-122.140365</longitude>
                        <altitude>153.199297198</altitude>
                        <speed>3</speed>
                    </dest>
                </leg>
            </stage>

        </stages>

    </MainFP>

</FlightPlan>
```

- **MULTIPLE DRONES' FLIGHT PLAN: 2° DRONE (NED Coordinates)**


```xml
<FlightPlan>

    <Locale>
        <speedUnits>ms</speedUnits>
        <altitudeUnits>m</altitudeUnits>
        <trackseparationUnits>m</trackseparationUnits>
        <distanceUnits>m</distanceUnits>
        <decimalSeparator>.</decimalSeparator>
        <groupSeparator/>
    </Locale>

    <MainFP>
        <name>NEIGHBORHOOD FLIGHT PLAN.</name>
        <description>Neighborhood Surveillance Mission.</description>

        <stages>

            <stage id="Intersection Leg Part">
                <leg id="zero-point" xsi_type="fp_IntersectionLeg">
                    <nextList>third-point-a third-point-b third-point-c third-point-d</nextList>
                </leg>
                    <leg id="third-point-a" xsi_type="fp_Scan">
                        <dest>
                            <coordinates>0 0</coordinates>
                        </dest>
                        <trackseparation>20</trackseparation>
                        <area>
                            <point1>125 0</point1>
                            <point2>125 125</point2>
                            <point3>0 125</point3>
                            <point4>0 0</point4>
                        </area>
                        <speed>5</speed>
                        <altitude>-40</altitude>
                    </leg>
                    <leg id="third-point-b" xsi_type="fp_Scan">
                        <dest>
                            <coordinates>0 0</coordinates>
                        </dest>
                        <trackseparation>20</trackseparation>
                        <area>
                            <point1>-125 0</point1>
                            <point2>-125 -125</point2>
                            <point3>0 -125</point3>
                            <point4>0 0</point4>
                        </area>
                        <speed>5</speed>
                        <altitude>-40</altitude>
                    </leg>
                    <leg id="third-point-c" xsi_type="fp_Scan">
```

```xml
                <leg id="third-point-c" xsi_type="fp_Scan">
                    <dest>
                        <coordinates>0 0</coordinates>
                    </dest>
                    <trackseparation>20</trackseparation>
                    <area>
                        <point1>0 -125</point1>
                        <point2>125 -125</point2>
                        <point3>125 0</point3>
                        <point4>0 0</point4>
                    </area>
                    <speed>5</speed>
                    <altitude>-40</altitude>
                </leg>
                <leg id="third-point-d" xsi_type="fp_Scan">
                    <dest>
                        <coordinates>0 0</coordinates>
                    </dest>
                    <trackseparation>20</trackseparation>
                    <area>
                        <point1>0 125</point1>
                        <point2>-125 125</point2>
                        <point3>-125 0</point3>
                        <point4>0 0</point4>
                    </area>
                    <speed>5</speed>
                    <altitude>-40</altitude>
                </leg>
            <initialLegs> zero-point </initialLegs>
            <finalLegs> third-point-a third-point-b third-point-c third-point-d </finalLegs>
        </stage>

        <stage id="Go Home Part">
            <leg id="Home1" xsi_type="fp_TFLeg">
                <dest>
                    <name>Home1</name>
                    <north_coordinate>0</north_coordinate>
                    <east_coordinate>0</east_coordinate>
                    <altitude>-4</altitude>
                    <speed>3</speed>
                    <next>Home2</next>
                </dest>
            </leg>
        </stage>

    </stages>

</MainFP>

</FlightPlan>
```

- **MULTIPLE DRONES’ FLIGHT PLAN: 2° DRONE (Geographical Coordinates)**


```xml
<FlightPlan>

    <Locale>
        <speedUnits>ms</speedUnits>
        <altitudeUnits>m</altitudeUnits>
        <trackseparationUnits>m</trackseparationUnits>
        <distanceUnits>°</distanceUnits>
        <decimalSeparator>.</decimalSeparator>
        <groupSeparator/>
    </Locale>

    <MainFP>
        <name>NEIGHBORHOOD FLIGHT PLAN.</name>
        <description>Neighborhood Surveillance Mission.</description>

        <stages>

            <stage id="Intersection Leg Part">
                <leg id="zero-point" xsi_type="fp_IntersectionLeg">
                    <nextList>third-point-a third-point-b third-point-c third-point-d</nextList>
                </leg>
                    <leg id="third-point-a" xsi_type="fp_Scan">
                        <dest>
                            <coordinates>0 0</coordinates>
                        </dest>
                        <trackseparation>20</trackseparation>
                        <area>
                            <point1>47.64260464285712 -122.140365</point1>
                            <point2>47.640360097142874 -122.1386985308925</point2>
                            <point3>47.64148237 -122.1386985308925</point3>
                            <point4>47.64148237 -122.140365</point4>
                        </area>
                        <speed>5</speed>
                        <altitude>163.199297198</altitude>
                    </leg>
                    <leg id="third-point-b" xsi_type="fp_Scan">
                        <dest>
                            <coordinates>0 0</coordinates>
                        </dest>
                        <trackseparation>20</trackseparation>
                        <area>
                            <point1>47.640360097142874 -122.140365</point1>
                            <point2>47.640360097142874 -122.1420314691075</point2>
                            <point3>47.64148237 -122.1420314691075</point3>
                            <point4>47.64148237 -122.140365</point4>
                        </area>
                        <speed>5</speed>
                        <altitude>163.199297198</altitude>
                    </leg>
                    <leg id="third-point-c" xsi_type="fp_Scan">
```

```xml
                <leg id="third-point-c" xsi_type="fp_Scan">
                    <dest>
                        <coordinates>0 0</coordinates>
                    </dest>
                    <trackseparation>20</trackseparation>
                    <area>
                        <point1>47.64148237 -122.14203146910751</point1>
                        <point2>47.64260464285712 -122.14203146910751</point2>
                        <point3>47.64260464285712 -122.140365</point3>
                        <point4>47.64148237 -122.140365</point4>
                    </area>
                    <speed>5</speed>
                    <altitude>163.199297198</altitude>
                </leg>
                <leg id="third-point-d" xsi_type="fp_Scan">
                    <dest>
                        <coordinates>0 0</coordinates>
                    </dest>
                    <trackseparation>20</trackseparation>
                    <area>
                        <point1>47.64148237 -122.1386985308925</point1>
                        <point2>47.640360097142874 -122.1386985308925</point2>
                        <point3>47.640360097142874 -122.140365</point3>
                        <point4>47.64148237 -122.140365</point4>
                    </area>
                    <speed>5</speed>
                    <altitude>163.199297198</altitude>
                </leg>
            <initialLegs> zero-point </initialLegs>
            <finalLegs> third-point-a third-point-b third-point-c third-point-d </finalLegs>
        </stage>

        <stage id="Go Home Part">
            <leg id="Home1" xsi_type="fp_TFLeg">
                <dest>
                    <name>Home1</name>
                    <latitude>47.64148237</latitude>
                    <longitude>-122.140365</longitude>
                    <altitude>127.199297198</altitude>
                    <speed>3</speed>
                    <next>Home2</next>
                </dest>
            </leg>
        </stage>

    </stages>

</MainFP>

</FlightPlan>
```

- **MULTIPLE DRONES' PYTHON CODE**

```python
# Author of the Code: Francesco Rose
# Master Thesis Project, Universitat Politecnica de la Catalunya
# Dynamic Interface AirSim/XML Fight Plan

import airsim

import threading
import time
import concurrent.futures

from math import *

import xml.etree.ElementTree as ET

import numpy as np

from geographiclib import geodesic


# ------------------------------------------------------------------------

client = airsim.MultirotorClient()
client.confirmConnection()

client.reset()

populations = input("How many drone do you want to use?: ")
population = int(populations)

for count in range(population):
    (client.enableApiControl(True, vehicle_name='Drone' + str(count + 1)))
    (client.armDisarm(True, vehicle_name='Drone' + str(count + 1)))

i = 1
h = -1

for count in range(population):
    client.moveToZAsync(h-i, 3, vehicle_name='Drone' + str(count + 1))
    i += 3
    time.sleep(4)
```

```python
class MainProgram:
    def __init__(self):
        self._lock = threading.Lock()

    def execution(self, drone_number):

        self._lock.acquire()

        FP = input("Which Flight Plan do you want to use? Please, select 0 for Geographical FP or 1 for NED FP: ")

        if FP == '0':
            tree = ET.parse(
                'Drone' + str(drone_number + 1) + '_FlightPlan_Neighborhood_Final_Version_Geographical.xml')
        elif FP == '1':
            tree = ET.parse('Drone' + str(drone_number + 1) + '_FlightPlan_Neighborhood_Final_Version_meters.xml')

        root = tree.getroot()
        ET.tostring(root, encoding='utf8').decode('utf8')

        vehicle_name = 'Drone' + str(drone_number + 1)

        state = client.getMultirotorState(vehicle_name)

        global x_home, y_home, z_home

        x_home = state.kinematics_estimated.position.x_val
        y_home = state.kinematics_estimated.position.y_val
        z_home = state.kinematics_estimated.position.z_val

        lat0 = state.gps_location.latitude
        long0 = state.gps_location.longitude
        h_geo0 = state.gps_location.altitude

        arcWE = geodesic.Geodesic.WGS84.Inverse(lat0, long0 - 0.5, lat0, long0 + 0.5)
        arcNS = geodesic.Geodesic.WGS84.Inverse(lat0 - 0.5, long0, lat0 + 0.5, long0)

        lat_conv = 1 / arcNS['s12']
        long_conv = 1 / arcWE['s12']

        waypoints_data = {}

        if FP == '0':
            for legs in tree.iter(tag='leg'):
                if legs.attrib['xsi_type'] == "fp_TFLeg":
```

```
                name_point = legs.attrib['id']
                waypoints_data[name_point] = {}
                leg_type = legs.attrib['xsi_type']
                waypoints_data[name_point]["leg_type"] = leg_type
                for components in legs.iter():
                    if components.tag == "latitude":
                        lat = components.text
                        lat = float(lat)
                        if lat == lat0:
                            n_coord = 0
                        else:
                            n_coord = (lat - lat0) / lat_conv
                        waypoints_data[name_point]["n_coord"] = n_coord
                    elif components.tag == "longitude":
                        long = components.text
                        long = float(long)
                        if long == long0:
                            e_coord = 0
                        else:
                            e_coord = (long - long0) / long_conv
                        waypoints_data[name_point]["e_coord"] = e_coord
                    elif components.tag == "altitude":
                        altitude = components.text
                        altitude = float(altitude)
                        alt = z_home - (altitude - h_geo0)
                        waypoints_data[name_point]["altitude"] = alt
                    elif components.tag == "speed":
                        speed = components.text
                        speed = float(speed)
                        waypoints_data[name_point]["speed"] = speed
            if legs.attrib['xsi_type'] == "fp_Scan":
                name_point = legs.attrib['id']
                waypoints_data[name_point] = {}
                leg_type = legs.attrib['xsi_type']
                waypoints_data[name_point]["leg_type"] = leg_type
                for components in legs.iter():
                    if components.tag == "trackseparation":
                        ts = components.text
                        ts = float(ts)
                        waypoints_data[name_point]["trackseparation"] = ts
                    elif components.tag == "point1":
                        f_p = components.text
                        f_p = f_p.split()
                        lat = float(f_p[0])
```

```
            if lat == lat0:
                f_p_n = 0
            else:
                f_p_n = (lat - lat0) / lat_conv
            long = float(f_p[1])
            if long == long0:
                f_p_e = 0
            else:
                f_p_e = (long - long0) / long_conv
            waypoints_data[name_point]["point1_n"] = f_p_n
            waypoints_data[name_point]["point1_e"] = f_p_e
        elif components.tag == "altitude":
            h = components.text
            h = float(h)
            alt = z_home - (h - h_geo0)
            waypoints_data[name_point]["altitude"] = alt
        elif components.tag == "speed":
            v = components.text
            v = float(v)
            waypoints_data[name_point]["speed"] = v
    l_l = abs(f_p_n) + abs(f_p_e)
    ll = round(l_l)
    waypoints_data[name_point]["lenght_side"] = ll
    waypoints_data[name_point]["iteration"] = int(ll / ts)
if legs.attrib['xsi_type'] == "fp_IterativeLeg":
    name_point = legs.attrib['id']
    waypoints_data[name_point] = {}
    leg_type = legs.attrib['xsi_type']
    waypoints_data[name_point]["leg_type"] = leg_type
    for components in legs.iter():
        if components.tag == "body":
            iterative_path = components.text
            iter_path = iterative_path.split()
            len_iter_path = len(iter_path)
            waypoints_data[name_point]["len_iter_path"] = len_iter_path
            for n in range(len(iter_path)):
                waypoints_data[name_point]["name_point_iter_path" + str(n)] = iter_path[n]
        elif components.tag == "next":
            name_next_point = components.text
            waypoints_data[name_point]["name_next_point"] = name_next_point
        elif components.tag == "upperBound":
            num_of_iteration = components.text
            num_of_iter = int(num_of_iteration)
            waypoints_data[name_point]["num_of_iter"] = num_of_iter
```

```
            if legs.attrib['xsi_type'] == "fp_IntersectionLeg":
                name_point = legs.attrib['id']
                waypoints_data[name_point] = {}
                leg_type = legs.attrib['xsi_type']
                waypoints_data[name_point]["leg_type"] = leg_type
                for choices in legs.iter():
                    possibilities = choices.text
                    possibilities = possibilities.split()
                    for n in range(len(possibilities)):
                        waypoints_data[name_point]["name_point_possibility" + str(n)] = possibilities[n]
                    limit = len(possibilities)
                    waypoints_data[name_point]["limit"] = limit
    elif FP == '1':
        for legs in tree.iter(tag='leg'):
            if legs.attrib['xsi_type'] == "fp_TFLeg":
                name_point = legs.attrib['id']
                waypoints_data[name_point] = {}
                leg_type = legs.attrib['xsi_type']
                waypoints_data[name_point]["leg_type"] = leg_type
                for components in legs.iter():
                    if components.tag == "north_coordinate":
                        n_c = components.text
                        n_coord = float(n_c)
                        waypoints_data[name_point]["n_coord"] = n_coord
                    elif components.tag == "east_coordinate":
                        e_c = components.text
                        e_coord = float(e_c)
                        waypoints_data[name_point]["e_coord"] = e_coord
                    elif components.tag == "altitude":
                        alt = components.text
                        alt = float(alt)
                        waypoints_data[name_point]["altitude"] = alt
                    elif components.tag == "speed":
                        speed = components.text
                        speed = float(speed)
                        waypoints_data[name_point]["speed"] = speed
            if legs.attrib['xsi_type'] == "fp_Scan":
                name_point = legs.attrib['id']
                waypoints_data[name_point] = {}
                leg_type = legs.attrib['xsi_type']
                waypoints_data[name_point]["leg_type"] = leg_type
                for components in legs.iter():
                    if components.tag == "trackseparation":
                        ts = components.text
```

```
            ts = float(ts)
            waypoints_data[name_point]["trackseparation"] = ts
        elif components.tag == "point1":
            f_p = components.text
            f_p = f_p.split()
            f_p_n = float(f_p[0])
            f_p_e = float(f_p[1])
            waypoints_data[name_point]["point1_n"] = f_p_n
            waypoints_data[name_point]["point1_e"] = f_p_e
        elif components.tag == "altitude":
            h = components.text
            h = float(h)
            waypoints_data[name_point]["altitude"] = h
        elif components.tag == "speed":
            v = components.text
            v = float(v)
            waypoints_data[name_point]["speed"] = v
    l_l = abs(f_p_n) + abs(f_p_e)
    ll = round(l_l)
    waypoints_data[name_point]["lenght_side"] = ll
    waypoints_data[name_point]["iteration"] = int(ll / ts)
if legs.attrib['xsi_type'] == "fp_IterativeLeg":
    name_point = legs.attrib['id']
    waypoints_data[name_point] = {}
    leg_type = legs.attrib['xsi_type']
    waypoints_data[name_point]["leg_type"] = leg_type
    for components in legs.iter():
        if components.tag == "body":
            iterative_path = components.text
            iter_path = iterative_path.split()
            len_iter_path = len(iter_path)
            waypoints_data[name_point]["len_iter_path"] = len_iter_path
            for n in range(len(iter_path)):
                waypoints_data[name_point]["name_point_iter_path" + str(n)] = iter_path[n]
        elif components.tag == "next":
            name_next_point = components.text
            waypoints_data[name_point]["name_next_point"] = name_next_point
        elif components.tag == "upperBound":
            num_of_iteration = components.text
            num_of_iter = int(num_of_iteration)
            waypoints_data[name_point]["num_of_iter"] = num_of_iter
if legs.attrib['xsi_type'] == "fp_IntersectionLeg":
    name_point = legs.attrib['id']
    waypoints_data[name_point] = {}
```

```
                    leg_type = legs.attrib['xsi_type']
                    waypoints_data[name_point]["leg_type"] = leg_type
                    for choices in legs.iter():
                        possibilities = choices.text
                        possibilities = possibilities.split()
                        for n in range(len(possibilities)):
                            waypoints_data[name_point]["name_point_possibility" + str(n)] = possibilities[n]
                        limit = len(possibilities)
                        waypoints_data[name_point]["limit"] = limit

        print(waypoints_data)

        self._lock.release()

        stages = []
        for element in root.iter(tag='stage'):
            stages.append(element.attrib['id'])

        legs = []
        for n in range(len(stages)):
            for stage in tree.iter(tag='stage'):
                if stage.attrib['id'] == stages[n]:
                    for components in stage.iter(tag='leg'):
                        name_point = components.attrib['id']
                        legs.append(name_point)
        possibilities = []
        for elem in tree.iter(tag='leg'):
            if elem.attrib['xsi_type'] == "fp_IntersectionLeg":
                for choices in elem.iter():
                    pox = choices.text
                    possibilities = pox.split()
        iter_path = []
        going = []
        for elem in tree.iter(tag='leg'):
            if elem.attrib['xsi_type'] == "fp_IterativeLeg":
                for components in elem.iter():
                    if components.tag == "body":
                        iterative_path = components.text
                        iter_path = iterative_path.split()
                    elif components.tag == "next":
                        name_next_point = components.text
                        going.append(name_next_point)
        matches1 = []
        for leg1 in legs:
```

```
        for possibility in possibilities:
            if leg1 == possibility:
                matches1.append(leg1)
    for n in range(len(matches1)):
        legs.remove(matches1[n])
    matches2 = []
    for leg2 in legs:
        for iter in iter_path:
            if leg2 == iter:
                matches2.append(leg2)
    for n in range(len(matches2)):
        legs.remove(matches2[n])
    matches3 = []
    for leg3 in legs:
        for nex in going:
            if leg3 == nex:
                matches3.append(leg3)
    for n in range(len(matches3)):
        legs.remove(matches3[n])
    first_child = legs

    for n in range(len(first_child)):
        if waypoints_data[first_child[n]]["leg_type"] == "fp_TFLeg":
            n_coord = waypoints_data[first_child[n]]["n_coord"]
            e_coord = waypoints_data[first_child[n]]["e_coord"]
            alt = waypoints_data[first_child[n]]["altitude"]
            speed = waypoints_data[first_child[n]]["speed"]

            client.moveToPositionAsync(n_coord, e_coord, alt, speed, vehicle_name=vehicle_name)

            list1 = [n_coord, e_coord, alt, speed]
            state = client.getMultirotorState(vehicle_name)
            p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
                         state.kinematics_estimated.position.z_val]

            distance = sqrt(
                (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                            list1[2] - p_attuale[2]) ** 2)
            time_required = (distance / list1[3])

            time.sleep(time_required * 0.9)
            a = True
            while a:
                if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (
```

```
                        list1[1] + 2) and (list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                    a = False
                else:
                    time.sleep(0.1)

                    state = client.getMultirotorState(vehicle_name)
                    p_attuale = [state.kinematics_estimated.position.x_val,
                                 state.kinematics_estimated.position.y_val,
                                 state.kinematics_estimated.position.z_val]
        if waypoints_data[first_child[n]]["leg_type"] == "fp_Scan":
            ts = waypoints_data[first_child[n]]["trackseparation"]
            f_p_n = waypoints_data[first_child[n]]["point1_n"]
            f_p_e = waypoints_data[first_child[n]]["point1_e"]
            v = waypoints_data[first_child[n]]["speed"]
            h = waypoints_data[first_child[n]]["altitude"]
            i = waypoints_data[first_child[n]]["iteration"]
            ll = waypoints_data[first_child[n]]["lenght_side"]

            client.moveToPositionAsync(f_p_n, f_p_e, h, v, vehicle_name=vehicle_name)

            list1 = [f_p_n, f_p_e, h, v]
            state = client.getMultirotorState(vehicle_name)
            p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
                         state.kinematics_estimated.position.z_val]

            distance = sqrt((list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                            list1[2] - p_attuale[2]) ** 2)
            time_required = (distance / list1[3])

            time.sleep(time_required * 0.9)
            a = True
            while a:
                if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (
                        list1[1] + 2) and (list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                    a = False
                else:
                    time.sleep(0.1)

                    state = client.getMultirotorState(vehicle_name)
                    p_attuale = [state.kinematics_estimated.position.x_val,
                                 state.kinematics_estimated.position.y_val,
                                 state.kinematics_estimated.position.z_val]
            if (f_p_e < 0 < f_p_n) or (f_p_n > 0 and f_p_e == 0):
                if i % 2 == 0:
```

```
p = []
s = []
t = []
q = []
for num in range(1, i, 2):
    sott1 = num * ts
    a = f_p_n - sott1
    P = (a, f_p_e, h, v)
    p.append(P)
    S = (a, f_p_e + ll, h, v)
    s.append(S)
for mol in range(2, i + 1, 2):
    sott2 = mol * ts
    b = f_p_n - sott2
    T = (b, f_p_e + ll, h, v)
    t.append(T)
    Q = (b, f_p_e, h, v)
    q.append(Q)
Total = []
for n in range(len(q)):
    total = [p[n], s[n], t[n], q[n]]
    Total.append(total)
Totale = []
for n in range(len(Total)):
    for a in range(4):
        totale = Total[n][a]
        Totale.append(totale)
print(Totale)
for n in range(len(Totale)):
    client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                               vehicle_name=vehicle_name)

    list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val,
                 state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])

    time.sleep(time_required * 0.9)
```

```
        a = True
        while a:
            if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
                1] < (
                    list1[1] + 2) and (
                    list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                a = False
            else:
                time.sleep(0.1)

                state = client.getMultirotorState(vehicle_name)
                p_attuale = [state.kinematics_estimated.position.x_val,
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i + 1, 2):
        sott1 = num * ts
        a = f_p_n - sott1
        P = (a, f_p_e, h, v)
        p.append(P)
        S = (a, f_p_e + ll, h, v)
        s.append(S)
    for mol in range(2, i + 2, 2):
        sott2 = mol * ts
        b = f_p_n - sott2
        T = (b, f_p_e + ll, h, v)
        t.append(T)
        Q = (b, f_p_e, h, v)
        q.append(Q)
    Total = []
    for n in range(len(s)):
        total = [p[n], s[n], t[n], q[n]]
        Total.append(total)
    Totale = []
    for n in range(len(Total)):
        for a in range(4):
            totale = Total[n][a]
            Totale.append(totale)
    print(Totale)
    limit = len(Totale) - 2
```

```
        for n in range(limit):
            client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                                       vehicle_name=vehicle_name)

            list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
            state = client.getMultirotorState(vehicle_name)
            p_attuale = [state.kinematics_estimated.position.x_val,
                         state.kinematics_estimated.position.y_val,
                         state.kinematics_estimated.position.z_val]

            distance = sqrt(
                (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                        list1[2] - p_attuale[2]) ** 2)
            time_required = (distance / list1[3])

            time.sleep(time_required * 0.9)
            a = True
            while a:
                if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
                    1] < (
                        list1[1] + 2) and (
                        list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                    a = False
                else:
                    time.sleep(0.1)

                    state = client.getMultirotorState(vehicle_name)
                    p_attuale = [state.kinematics_estimated.position.x_val,
                                 state.kinematics_estimated.position.y_val,
                                 state.kinematics_estimated.position.z_val]
elif (f_p_n > 0 and f_p_e > 0) or (f_p_n == 0 and f_p_e > 0):
    if i % 2 == 0:
        p = []
        s = []
        t = []
        q = []
        for num in range(1, i, 2):
            sott1 = num * ts
            a = f_p_n - sott1
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e - ll, h, v)
            s.append(S)
        for mol in range(2, i + 1, 2):
```

```
    sott2 = mol * ts
    b = f_p_n - sott2
    T = (b, f_p_e - ll, h, v)
    t.append(T)
    Q = (b, f_p_e, h, v)
    q.append(Q)
Total = []
for n in range(len(q)):
    total = [p[n], s[n], t[n], q[n]]
    Total.append(total)
Totale = []
for n in range(len(Total)):
    for a in range(4):
        totale = Total[n][a]
        Totale.append(totale)
print(Totale)
for n in range(len(Totale)):
    client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                               vehicle_name=vehicle_name)

    list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val,
                 state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])

    time.sleep(time_required * 0.9)
    a = True
    while a:
        if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
            1] < (
                list1[1] + 2) and (
                list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
            a = False
        else:
            time.sleep(0.1)

            state = client.getMultirotorState(vehicle_name)
            p_attuale = [state.kinematics_estimated.position.x_val,
```

```
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i + 1, 2):
        sott1 = num * ts
        a = f_p_n - sott1
        P = (a, f_p_e, h, v)
        p.append(P)
        S = (a, f_p_e - ll, h, v)
        s.append(S)
    for mol in range(2, i + 2, 2):
        sott2 = mol * ts
        b = f_p_n - sott2
        T = (b, f_p_e - ll, h, v)
        t.append(T)
        Q = (b, f_p_e, h, v)
        q.append(Q)
    Total = []
    for n in range(len(s)):
        total = [p[n], s[n], t[n], q[n]]
        Total.append(total)
    Totale = []
    for n in range(len(Total)):
        for a in range(4):
            totale = Total[n][a]
            Totale.append(totale)
    print(Totale)
    limit = len(Totale) - 2
    for n in range(limit):
        client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                                   vehicle_name=vehicle_name)

        list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
        state = client.getMultirotorState(vehicle_name)
        p_attuale = [state.kinematics_estimated.position.x_val,
                     state.kinematics_estimated.position.y_val,
                     state.kinematics_estimated.position.z_val]

        distance = sqrt(
            (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
```

```python
                        list1[2] - p_attuale[2]) ** 2)
            time_required = (distance / list1[3])

            time.sleep(time_required * 0.9)
            a = True
            while a:
                if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
                    1] < (
                        list1[1] + 2) and (
                        list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                    a = False
                else:
                    time.sleep(0.1)

                    state = client.getMultirotorState(vehicle_name)
                    p_attuale = [state.kinematics_estimated.position.x_val,
                                 state.kinematics_estimated.position.y_val,
                                 state.kinematics_estimated.position.z_val]
elif (f_p_n < 0 < f_p_e) or (f_p_n < 0 and f_p_e == 0):
    if i % 2 == 0:
        p = []
        s = []
        t = []
        q = []
        for num in range(1, i, 2):
            sott1 = num * ts
            a = f_p_n + sott1
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e - ll, h, v)
            s.append(S)
        for mol in range(2, i + 1, 2):
            sott2 = mol * ts
            b = f_p_n + sott2
            T = (b, f_p_e - ll, h, v)
            t.append(T)
            Q = (b, f_p_e, h, v)
            q.append(Q)
        Total = []
        for n in range(len(q)):
            total = [p[n], s[n], t[n], q[n]]
            Total.append(total)
        Totale = []
        for n in range(len(Total)):
```

```python
        for a in range(4):
            totale = Total[n][a]
            Totale.append(totale)
    print(Totale)
    for n in range(len(Totale)):
        client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                                   vehicle_name=vehicle_name)

        list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
        state = client.getMultirotorState(vehicle_name)
        p_attuale = [state.kinematics_estimated.position.x_val,
                     state.kinematics_estimated.position.y_val,
                     state.kinematics_estimated.position.z_val]

        distance = sqrt(
            (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                    list1[2] - p_attuale[2]) ** 2)
        time_required = (distance / list1[3])

        time.sleep(time_required * 0.9)
        a = True
        while a:
            if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
                1] < (
                    list1[1] + 2) and (
                    list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                a = False
            else:
                time.sleep(0.1)

                state = client.getMultirotorState(vehicle_name)
                p_attuale = [state.kinematics_estimated.position.x_val,
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i + 1, 2):
        sott1 = num * ts
        a = f_p_n + sott1
        P = (a, f_p_e, h, v)
        p.append(P)
```

```
    S = (a, f_p_e - ll, h, v)
    s.append(S)
for mol in range(2, i + 2, 2):
    sott2 = mol * ts
    b = f_p_n + sott2
    T = (b, f_p_e - ll, h, v)
    t.append(T)
    Q = (b, f_p_e, h, v)
    q.append(Q)
Total = []
for n in range(len(s)):
    total = [p[n], s[n], t[n], q[n]]
    Total.append(total)
Totale = []
for n in range(len(Total)):
    for a in range(4):
        totale = Total[n][a]
        Totale.append(totale)
limit = len(Totale) - 2
for n in range(limit):
    client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                               vehicle_name=vehicle_name)

    list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val,
                 state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])

    time.sleep(time_required * 0.9)
    a = True
    while a:
        if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
            1] < (
                list1[1] + 2) and (
                list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
            a = False
        else:
            time.sleep(0.1)
```

```
                                    state = client.getMultirotorState(vehicle_name)
                                    p_attuale = [state.kinematics_estimated.position.x_val,
                                                 state.kinematics_estimated.position.y_val,
                                                 state.kinematics_estimated.position.z_val]
                elif (f_p_n < 0 and f_p_e < 0) or (f_p_n == 0 and f_p_e < 0):
                    if i % 2 == 0:
                        p = []
                        s = []
                        t = []
                        q = []
                        for num in range(1, i, 2):
                            sott1 = num * ts
                            a = f_p_n + sott1
                            P = (a, f_p_e, h, v)
                            p.append(P)
                            S = (a, f_p_e + ll, h, v)
                            s.append(S)
                        for mol in range(2, i + 1, 2):
                            sott2 = mol * ts
                            b = f_p_n + sott2
                            T = (b, f_p_e + ll, h, v)
                            t.append(T)
                            Q = (b, f_p_e, h, v)
                            q.append(Q)
                        Total = []
                        for n in range(len(q)):
                            total = [p[n], s[n], t[n], q[n]]
                            Total.append(total)
                        Totale = []
                        for n in range(len(Total)):
                            for a in range(4):
                                totale = Total[n][a]
                                Totale.append(totale)
                        print(Totale)
                        for n in range(len(Totale)):
                            client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                                                       vehicle_name=vehicle_name)

                            list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
                            state = client.getMultirotorState(vehicle_name)
                            p_attuale = [state.kinematics_estimated.position.x_val,
                                         state.kinematics_estimated.position.y_val,
                                         state.kinematics_estimated.position.z_val]
```

```
                            distance = sqrt(
                                (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                                        list1[2] - p_attuale[2]) ** 2)
                            time_required = (distance / list1[3])

                            time.sleep(time_required * 0.9)
                            a = True
                            while a:
                                if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
                                    1] < (
                                        list1[1] + 2) and (
                                        list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                                    a = False
                                else:
                                    time.sleep(0.1)

                                    state = client.getMultirotorState(vehicle_name)
                                    p_attuale = [state.kinematics_estimated.position.x_val,
                                                 state.kinematics_estimated.position.y_val,
                                                 state.kinematics_estimated.position.z_val]
                    else:
                        p = []
                        s = []
                        t = []
                        q = []
                        for num in range(1, i + 1, 2):
                            sott1 = num * ts
                            a = f_p_n + sott1
                            P = (a, f_p_e, h, v)
                            p.append(P)
                            S = (a, f_p_e + ll, h, v)
                            s.append(S)
                        for mol in range(2, i + 2, 2):
                            sott2 = mol * ts
                            b = f_p_n + sott2
                            T = (b, f_p_e + ll, h, v)
                            t.append(T)
                            Q = (b, f_p_e, h, v)
                            q.append(Q)
                        Total = []
                        for n in range(len(s)):
                            total = [p[n], s[n], t[n], q[n]]
                            Total.append(total)
```

```
            Totale = []
            for n in range(len(Total)):
                for a in range(4):
                    totale = Total[n][a]
                    Totale.append(totale)
            print(Totale)
            limit = len(Totale) - 2
            for n in range(limit):
                client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                                           vehicle_name=vehicle_name)

                list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
                state = client.getMultirotorState(vehicle_name)
                p_attuale = [state.kinematics_estimated.position.x_val,
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]

                distance = sqrt(
                    (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                            list1[2] - p_attuale[2]) ** 2)
                time_required = (distance / list1[3])

                time.sleep(time_required * 0.9)
                a = True
                while a:
                    if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
                        1] < (
                            list1[1] + 2) and (
                            list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                        a = False
                    else:
                        time.sleep(0.1)

                        state = client.getMultirotorState(vehicle_name)
                        p_attuale = [state.kinematics_estimated.position.x_val,
                                     state.kinematics_estimated.position.y_val,
                                     state.kinematics_estimated.position.z_val]
if waypoints_data[first_child[n]]["leg_type"] == "fp_IterativeLeg":
    name_next_point = waypoints_data[first_child[n]]["name_next_point"]
    num_of_iter = waypoints_data[first_child[n]]["num_of_iter"]
    name_point = first_child[n]
    lip = waypoints_data[name_point]["len_iter_path"]
    iter_path = []
    for n in range(lip):
```

```
    iter_path.append(waypoints_data[name_point]["name_point_iter_path" + str(n)])
rip = 0
while rip < num_of_iter:
    for n in range(len(iter_path)):
        name_point = iter_path[n]
        if waypoints_data[name_point]["leg_type"] == "fp_TFLeg":
            n_coord = waypoints_data[name_point]["n_coord"]
            e_coord = waypoints_data[name_point]["e_coord"]
            alt = waypoints_data[name_point]["altitude"]
            speed = waypoints_data[name_point]["speed"]

            client.moveToPositionAsync(n_coord, e_coord, alt, speed, vehicle_name=vehicle_name)

            list1 = [n_coord, e_coord, alt, speed]
            state = client.getMultirotorState(vehicle_name)
            p_attuale = [state.kinematics_estimated.position.x_val,
                         state.kinematics_estimated.position.y_val,
                         state.kinematics_estimated.position.z_val]

            distance = sqrt(
                (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                        list1[2] - p_attuale[2]) ** 2)
            time_required = (distance / list1[3])

            time.sleep(time_required * 0.9)
            a = True
            while a:
                if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
                    1] < (
                        list1[1] + 2) and (
                        list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                    a = False
                else:
                    time.sleep(0.1)

                    state = client.getMultirotorState(vehicle_name)
                    p_attuale = [state.kinematics_estimated.position.x_val,
                                 state.kinematics_estimated.position.y_val,
                                 state.kinematics_estimated.position.z_val]
        if waypoints_data[name_point]["leg_type"] == "fp_Scan":
            ts = waypoints_data[name_point]["trackseparation"]
            f_p_n = waypoints_data[name_point]["point1_n"]
            f_p_e = waypoints_data[name_point]["point1_e"]
            v = waypoints_data[name_point]["speed"]
```

```python
h = waypoints_data[name_point]["altitude"]
i = waypoints_data[name_point]["iteration"]
ll = waypoints_data[name_point]["lenght_side"]
client.moveToPositionAsync(f_p_n, f_p_e, h, v, vehicle_name=vehicle_name)

list1 = [f_p_n, f_p_e, h, v]
state = client.getMultirotorState(vehicle_name)
p_attuale = [state.kinematics_estimated.position.x_val,
             state.kinematics_estimated.position.y_val,
             state.kinematics_estimated.position.z_val]

distance = sqrt(
    (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
            list1[2] - p_attuale[2]) ** 2)
time_required = (distance / list1[3])

time.sleep(time_required * 0.9)
a = True
while a:
    if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[
        1] < (
            list1[1] + 2) and (
            list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
        a = False
    else:
        time.sleep(0.1)

        state = client.getMultirotorState(vehicle_name)
        p_attuale = [state.kinematics_estimated.position.x_val,
                     state.kinematics_estimated.position.y_val,
                     state.kinematics_estimated.position.z_val]
if (f_p_e < 0 < f_p_n) or (f_p_n > 0 and f_p_e == 0):
    if i % 2 == 0:
        p = []
        s = []
        t = []
        q = []
        for num in range(1, i, 2):
            sott1 = num * ts
            a = f_p_n - sott1
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e + ll, h, v)
            s.append(S)
```

```
for mol in range(2, i + 1, 2):
    sott2 = mol * ts
    b = f_p_n - sott2
    T = (b, f_p_e + ll, h, v)
    t.append(T)
    Q = (b, f_p_e, h, v)
    q.append(Q)
Total = []
for n in range(len(q)):
    total = [p[n], s[n], t[n], q[n]]
    Total.append(total)
Totale = []
for n in range(len(Total)):
    for a in range(4):
        totale = Total[n][a]
        Totale.append(totale)
print(Totale)
for n in range(len(Totale)):
    client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
                               Totale[n][3],
                               vehicle_name=vehicle_name)

    list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val,
                 state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])

    time.sleep(time_required * 0.9)
    a = True
    while a:
        if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                p_attuale[
                    1] < (
                list1[1] + 2) and (
                list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
            a = False
        else:
            time.sleep(0.1)
```

```
                state = client.getMultirotorState(vehicle_name)
                p_attuale = [state.kinematics_estimated.position.x_val,
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i + 1, 2):
        sott1 = num * ts
        a = f_p_n - sott1
        P = (a, f_p_e, h, v)
        p.append(P)
        S = (a, f_p_e + ll, h, v)
        s.append(S)
    for mol in range(2, i + 2, 2):
        sott2 = mol * ts
        b = f_p_n - sott2
        T = (b, f_p_e + ll, h, v)
        t.append(T)
        Q = (b, f_p_e, h, v)
        q.append(Q)
    Total = []
    for n in range(len(s)):
        total = [p[n], s[n], t[n], q[n]]
        Total.append(total)
    Totale = []
    for n in range(len(Total)):
        for a in range(4):
            totale = Total[n][a]
            Totale.append(totale)
    print(Totale)
    limit = len(Totale) - 2
    for n in range(limit):
        client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
                                   Totale[n][3],
                                   vehicle_name=vehicle_name)

        list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
        state = client.getMultirotorState(vehicle_name)
        p_attuale = [state.kinematics_estimated.position.x_val,
                     state.kinematics_estimated.position.y_val,
```

```
                         state.kinematics_estimated.position.z_val]

            distance = sqrt(
                (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                        list1[2] - p_attuale[2]) ** 2)
            time_required = (distance / list1[3])

            time.sleep(time_required * 0.9)
            a = True
            while a:
                if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                        p_attuale[
                            1] < (
                        list1[1] + 2) and (
                        list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                    a = False
                else:
                    time.sleep(0.1)

                    state = client.getMultirotorState(vehicle_name)
                    p_attuale = [state.kinematics_estimated.position.x_val,
                                 state.kinematics_estimated.position.y_val,
                                 state.kinematics_estimated.position.z_val]
elif (f_p_n > 0 and f_p_e > 0) or (f_p_n == 0 and f_p_e > 0):
    if i % 2 == 0:
        p = []
        s = []
        t = []
        q = []
        for num in range(1, i, 2):
            sott1 = num * ts
            a = f_p_n - sott1
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e - ll, h, v)
            s.append(S)
        for mol in range(2, i + 1, 2):
            sott2 = mol * ts
            b = f_p_n - sott2
            T = (b, f_p_e - ll, h, v)
            t.append(T)
            Q = (b, f_p_e, h, v)
            q.append(Q)
        Total = []
```

```
    for n in range(len(q)):
        total = [p[n], s[n], t[n], q[n]]
        Total.append(total)
    Totale = []
    for n in range(len(Total)):
        for a in range(4):
            totale = Total[n][a]
            Totale.append(totale)
    print(Totale)
    for n in range(len(Totale)):
        client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
                                   Totale[n][3],
                                   vehicle_name=vehicle_name)

        list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
        state = client.getMultirotorState(vehicle_name)
        p_attuale = [state.kinematics_estimated.position.x_val,
                     state.kinematics_estimated.position.y_val,
                     state.kinematics_estimated.position.z_val]

        distance = sqrt(
            (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                    list1[2] - p_attuale[2]) ** 2)
        time_required = (distance / list1[3])

        time.sleep(time_required * 0.9)
        a = True
        while a:
            if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                    p_attuale[
                        1] < (
                    list1[1] + 2) and (
                    list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                a = False
            else:
                time.sleep(0.1)

                state = client.getMultirotorState(vehicle_name)
                p_attuale = [state.kinematics_estimated.position.x_val,
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
else:
    p = []
    s = []
```

```
t = []
q = []
for num in range(1, i + 1, 2):
    sott1 = num * ts
    a = f_p_n - sott1
    P = (a, f_p_e, h, v)
    p.append(P)
    S = (a, f_p_e - ll, h, v)
    s.append(S)
for mol in range(2, i + 2, 2):
    sott2 = mol * ts
    b = f_p_n - sott2
    T = (b, f_p_e - ll, h, v)
    t.append(T)
    Q = (b, f_p_e, h, v)
    q.append(Q)
Total = []
for n in range(len(s)):
    total = [p[n], s[n], t[n], q[n]]
    Total.append(total)
Totale = []
for n in range(len(Total)):
    for a in range(4):
        totale = Total[n][a]
        Totale.append(totale)
print(Totale)
limit = len(Totale) - 2
for n in range(limit):
    client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
                               Totale[n][3],
                               vehicle_name=vehicle_name)

    list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val,
                 state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])

    time.sleep(time_required * 0.9)
```

```
            a = True
            while a:
                if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                        p_attuale[
                            1] < (
                        list1[1] + 2) and (
                        list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                    a = False
                else:
                    time.sleep(0.1)

                    state = client.getMultirotorState(vehicle_name)
                    p_attuale = [state.kinematics_estimated.position.x_val,
                                 state.kinematics_estimated.position.y_val,
                                 state.kinematics_estimated.position.z_val]
elif (f_p_n < 0 < f_p_e) or (f_p_n < 0 and f_p_e == 0):
    if i % 2 == 0:
        p = []
        s = []
        t = []
        q = []
        for num in range(1, i, 2):
            sott1 = num * ts
            a = f_p_n + sott1
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e - ll, h, v)
            s.append(S)
        for mol in range(2, i + 1, 2):
            sott2 = mol * ts
            b = f_p_n + sott2
            T = (b, f_p_e - ll, h, v)
            t.append(T)
            Q = (b, f_p_e, h, v)
            q.append(Q)
        Total = []
        for n in range(len(q)):
            total = [p[n], s[n], t[n], q[n]]
            Total.append(total)
        Totale = []
        for n in range(len(Total)):
            for a in range(4):
                totale = Total[n][a]
                Totale.append(totale)
```

```
    print(Totale)
    for n in range(len(Totale)):
        client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
                                   Totale[n][3],
                                   vehicle_name=vehicle_name)

        list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
        state = client.getMultirotorState(vehicle_name)
        p_attuale = [state.kinematics_estimated.position.x_val,
                     state.kinematics_estimated.position.y_val,
                     state.kinematics_estimated.position.z_val]

        distance = sqrt(
            (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                    list1[2] - p_attuale[2]) ** 2)
        time_required = (distance / list1[3])

        time.sleep(time_required * 0.9)
        a = True
        while a:
            if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                    p_attuale[
                        1] < (
                    list1[1] + 2) and (
                    list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                a = False
            else:
                time.sleep(0.1)

                state = client.getMultirotorState(vehicle_name)
                p_attuale = [state.kinematics_estimated.position.x_val,
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i + 1, 2):
        sott1 = num * ts
        a = f_p_n + sott1
        P = (a, f_p_e, h, v)
        p.append(P)
        S = (a, f_p_e - ll, h, v)
```

```
    s.append(S)
for mol in range(2, i + 2, 2):
    sott2 = mol * ts
    b = f_p_n + sott2
    T = (b, f_p_e - ll, h, v)
    t.append(T)
    Q = (b, f_p_e, h, v)
    q.append(Q)
Total = []
for n in range(len(s)):
    total = [p[n], s[n], t[n], q[n]]
    Total.append(total)
Totale = []
for n in range(len(Total)):
    for a in range(4):
        totale = Total[n][a]
        Totale.append(totale)
limit = len(Totale) - 2
for n in range(limit):
    client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
                               Totale[n][3],
                               vehicle_name=vehicle_name)

    list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val,
                 state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])

    time.sleep(time_required * 0.9)
    a = True
    while a:
        if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                p_attuale[
                    1] < (
                list1[1] + 2) and (
                list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
            a = False
        else:
```

```
                    time.sleep(0.1)

                    state = client.getMultirotorState(vehicle_name)
                    p_attuale = [state.kinematics_estimated.position.x_val,
                                 state.kinematics_estimated.position.y_val,
                                 state.kinematics_estimated.position.z_val]
elif (f_p_n < 0 and f_p_e < 0) or (f_p_n == 0 and f_p_e < 0):
    if i % 2 == 0:
        p = []
        s = []
        t = []
        q = []
        for num in range(1, i, 2):
            sott1 = num * ts
            a = f_p_n + sott1
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e + ll, h, v)
            s.append(S)
        for mol in range(2, i + 1, 2):
            sott2 = mol * ts
            b = f_p_n + sott2
            T = (b, f_p_e + ll, h, v)
            t.append(T)
            Q = (b, f_p_e, h, v)
            q.append(Q)
        Total = []
        for n in range(len(q)):
            total = [p[n], s[n], t[n], q[n]]
            Total.append(total)
        Totale = []
        for n in range(len(Total)):
            for a in range(4):
                totale = Total[n][a]
                Totale.append(totale)
        print(Totale)
        for n in range(len(Totale)):
            client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
                                       Totale[n][3],
                                       vehicle_name=vehicle_name)

            list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
            state = client.getMultirotorState(vehicle_name)
            p_attuale = [state.kinematics_estimated.position.x_val,
```

```
                     state.kinematics_estimated.position.y_val,
                     state.kinematics_estimated.position.z_val]

        distance = sqrt(
            (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                    list1[2] - p_attuale[2]) ** 2)
        time_required = (distance / list1[3])

        time.sleep(time_required * 0.9)
        a = True
        while a:
            if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                    p_attuale[
                        1] < (
                    list1[1] + 2) and (
                    list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                a = False
            else:
                time.sleep(0.1)

                state = client.getMultirotorState(vehicle_name)
                p_attuale = [state.kinematics_estimated.position.x_val,
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i + 1, 2):
        sott1 = num * ts
        a = f_p_n + sott1
        P = (a, f_p_e, h, v)
        p.append(P)
        S = (a, f_p_e + ll, h, v)
        s.append(S)
    for mol in range(2, i + 2, 2):
        sott2 = mol * ts
        b = f_p_n + sott2
        T = (b, f_p_e + ll, h, v)
        t.append(T)
        Q = (b, f_p_e, h, v)
        q.append(Q)
    Total = []
```

```
                    for n in range(len(s)):
                        total = [p[n], s[n], t[n], q[n]]
                        Total.append(total)
                    Totale = []
                    for n in range(len(Total)):
                        for a in range(4):
                            totale = Total[n][a]
                            Totale.append(totale)
                    print(Totale)
                    limit = len(Totale) - 2
                    for n in range(limit):
                        client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2],
                                                   Totale[n][3],
                                                   vehicle_name=vehicle_name)

                        list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
                        state = client.getMultirotorState(vehicle_name)
                        p_attuale = [state.kinematics_estimated.position.x_val,
                                     state.kinematics_estimated.position.y_val,
                                     state.kinematics_estimated.position.z_val]

                        distance = sqrt(
                            (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                                    list1[2] - p_attuale[2]) ** 2)
                        time_required = (distance / list1[3])

                        time.sleep(time_required * 0.9)
                        a = True
                        while a:
                            if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                                    p_attuale[
                                        1] < (
                                    list1[1] + 2) and (
                                    list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                                a = False
                            else:
                                time.sleep(0.1)

                                state = client.getMultirotorState(vehicle_name)
                                p_attuale = [state.kinematics_estimated.position.x_val,
                                             state.kinematics_estimated.position.y_val,
                                             state.kinematics_estimated.position.z_val]
    rip += 1
n_coord = waypoints_data[name_next_point]["n_coord"]
```

```python
    e_coord = waypoints_data[name_next_point]["e_coord"]
    h = waypoints_data[name_next_point]["altitude"]
    v = waypoints_data[name_next_point]["speed"]

    client.moveToPositionAsync(n_coord, e_coord, h, v, vehicle_name=vehicle_name)

    list1 = [n_coord, e_coord, h, v]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val, state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])

    time.sleep(time_required * 0.9)
    a = True
    while a:
        if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (
                list1[1] + 2) and (
                list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
            a = False
        else:
            time.sleep(0.1)

            state = client.getMultirotorState(vehicle_name)
            p_attuale = [state.kinematics_estimated.position.x_val,
                         state.kinematics_estimated.position.y_val,
                         state.kinematics_estimated.position.z_val]
if waypoints_data[first_child[n]]["leg_type"] == "fp_IntersectionLeg":
    limit = waypoints_data[first_child[n]]["limit"]
    option = input("Intersection Leg: to select one of the possibilities, insert a "
                   "number included between 1 and " + str(limit) + ": ")
    option = int(option)
    name_points = first_child[n]
    name_point = waypoints_data[name_points]["name_point_possibility" + str(option - 1)]
    if waypoints_data[name_point]["leg_type"] == "fp_TFLeg":
        n_coord = waypoints_data[name_point]["n_coord"]
        e_coord = waypoints_data[name_point]["e_coord"]
        alt = waypoints_data[name_point]["altitude"]
        speed = waypoints_data[name_point]["speed"]

        client.moveToPositionAsync(n_coord, e_coord, alt, speed, vehicle_name=vehicle_name)
```

```
    list1 = [n_coord, e_coord, alt, speed]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val,
                 state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                    list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])

    time.sleep(time_required * 0.9)
    a = True
    while a:
        if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (
                list1[1] + 2) and (
                list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
            a = False
        else:
            time.sleep(0.1)

            state = client.getMultirotorState(vehicle_name)
            p_attuale = [state.kinematics_estimated.position.x_val,
                         state.kinematics_estimated.position.y_val,
                         state.kinematics_estimated.position.z_val]
if waypoints_data[name_point]["leg_type"] == "fp_Scan":
    ts = waypoints_data[name_point]["trackseparation"]
    f_p_n = waypoints_data[name_point]["point1_n"]
    f_p_e = waypoints_data[name_point]["point1_e"]
    v = waypoints_data[name_point]["speed"]
    h = waypoints_data[name_point]["altitude"]
    i = waypoints_data[name_point]["iteration"]
    ll = waypoints_data[name_point]["lenght_side"]
    client.moveToPositionAsync(f_p_n, f_p_e, h, v, vehicle_name=vehicle_name)

    list1 = [f_p_n, f_p_e, h, v]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val,
                 state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
```

```python
                list1[2] - p_attuale[2]) ** 2)
time_required = (distance / list1[3])

time.sleep(time_required * 0.9)
a = True
while a:
    if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < p_attuale[1] < (
            list1[1] + 2) and (
            list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
        a = False
    else:
        time.sleep(0.1)

        state = client.getMultirotorState(vehicle_name)
        p_attuale = [state.kinematics_estimated.position.x_val,
                     state.kinematics_estimated.position.y_val,
                     state.kinematics_estimated.position.z_val]
if (f_p_e < 0 < f_p_n) or (f_p_n > 0 and f_p_e == 0):
    if i % 2 == 0:
        p = []
        s = []
        t = []
        q = []
        for num in range(1, i, 2):
            sott1 = num * ts
            a = f_p_n - sott1
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e + ll, h, v)
            s.append(S)
        for mol in range(2, i + 1, 2):
            sott2 = mol * ts
            b = f_p_n - sott2
            T = (b, f_p_e + ll, h, v)
            t.append(T)
            Q = (b, f_p_e, h, v)
            q.append(Q)
        Total = []
        for n in range(len(q)):
            total = [p[n], s[n], t[n], q[n]]
            Total.append(total)
        Totale = []
        for n in range(len(Total)):
            for a in range(4):
```

```
            totale = Total[n][a]
            Totale.append(totale)
    print(Totale)
    for n in range(len(Totale)):
        client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                                   vehicle_name=vehicle_name)

        list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
        state = client.getMultirotorState(vehicle_name)
        p_attuale = [state.kinematics_estimated.position.x_val,
                     state.kinematics_estimated.position.y_val,
                     state.kinematics_estimated.position.z_val]

        distance = sqrt(
            (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                    list1[2] - p_attuale[2]) ** 2)
        time_required = (distance / list1[3])

        time.sleep(time_required * 0.9)
        a = True
        while a:
            if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                    p_attuale[
                        1] < (
                    list1[1] + 2) and (
                    list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                a = False
            else:
                time.sleep(0.1)

                state = client.getMultirotorState(vehicle_name)
                p_attuale = [state.kinematics_estimated.position.x_val,
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i + 1, 2):
        sott1 = num * ts
        a = f_p_n - sott1
        P = (a, f_p_e, h, v)
        p.append(P)
```

```
    S = (a, f_p_e + ll, h, v)
    s.append(S)
for mol in range(2, i + 2, 2):
    sott2 = mol * ts
    b = f_p_n - sott2
    T = (b, f_p_e + ll, h, v)
    t.append(T)
    Q = (b, f_p_e, h, v)
    q.append(Q)
Total = []
for n in range(len(s)):
    total = [p[n], s[n], t[n], q[n]]
    Total.append(total)
Totale = []
for n in range(len(Total)):
    for a in range(4):
        totale = Total[n][a]
        Totale.append(totale)
print(Totale)
limit = len(Totale) - 2
for n in range(limit):
    client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                              vehicle_name=vehicle_name)

    list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val,
                 state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])

    time.sleep(time_required * 0.9)
    a = True
    while a:
        if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                p_attuale[
                    1] < (
                list1[1] + 2) and (
                list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
            a = False
```

```
                else:
                    time.sleep(0.1)

                    state = client.getMultirotorState(vehicle_name)
                    p_attuale = [state.kinematics_estimated.position.x_val,
                                 state.kinematics_estimated.position.y_val,
                                 state.kinematics_estimated.position.z_val]
elif (f_p_n > 0 and f_p_e > 0) or (f_p_n == 0 and f_p_e > 0):
    if i % 2 == 0:
        p = []
        s = []
        t = []
        q = []
        for num in range(1, i, 2):
            sott1 = num * ts
            a = f_p_n - sott1
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e - ll, h, v)
            s.append(S)
        for mol in range(2, i + 1, 2):
            sott2 = mol * ts
            b = f_p_n - sott2
            T = (b, f_p_e - ll, h, v)
            t.append(T)
            Q = (b, f_p_e, h, v)
            q.append(Q)
        Total = []
        for n in range(len(q)):
            total = [p[n], s[n], t[n], q[n]]
            Total.append(total)
        Totale = []
        for n in range(len(Total)):
            for a in range(4):
                totale = Total[n][a]
                Totale.append(totale)
        print(Totale)
        for n in range(len(Totale)):
            client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                                       vehicle_name=vehicle_name)

            list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
            state = client.getMultirotorState(vehicle_name)
            p_attuale = [state.kinematics_estimated.position.x_val,
```

```
                     state.kinematics_estimated.position.y_val,
                     state.kinematics_estimated.position.z_val]

        distance = sqrt(
            (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                    list1[2] - p_attuale[2]) ** 2)
        time_required = (distance / list1[3])

        time.sleep(time_required * 0.9)
        a = True
        while a:
            if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                    p_attuale[
                        1] < (
                    list1[1] + 2) and (
                    list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                a = False
            else:
                time.sleep(0.1)

                state = client.getMultirotorState(vehicle_name)
                p_attuale = [state.kinematics_estimated.position.x_val,
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i + 1, 2):
        sott1 = num * ts
        a = f_p_n - sott1
        P = (a, f_p_e, h, v)
        p.append(P)
        S = (a, f_p_e - ll, h, v)
        s.append(S)
    for mol in range(2, i + 2, 2):
        sott2 = mol * ts
        b = f_p_n - sott2
        T = (b, f_p_e - ll, h, v)
        t.append(T)
        Q = (b, f_p_e, h, v)
        q.append(Q)
    Total = []
```

```
        for n in range(len(s)):
            total = [p[n], s[n], t[n], q[n]]
            Total.append(total)
        Totale = []
        for n in range(len(Total)):
            for a in range(4):
                totale = Total[n][a]
                Totale.append(totale)
        print(Totale)
        limit = len(Totale) - 2
        for n in range(limit):
            client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                                       vehicle_name=vehicle_name)

            list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
            state = client.getMultirotorState(vehicle_name)
            p_attuale = [state.kinematics_estimated.position.x_val,
                         state.kinematics_estimated.position.y_val,
                         state.kinematics_estimated.position.z_val]

            distance = sqrt(
                (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                        list1[2] - p_attuale[2]) ** 2)
            time_required = (distance / list1[3])

            time.sleep(time_required * 0.9)
            a = True
            while a:
                if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                        p_attuale[
                            1] < (
                        list1[1] + 2) and (
                        list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                    a = False
                else:
                    time.sleep(0.1)

                    state = client.getMultirotorState(vehicle_name)
                    p_attuale = [state.kinematics_estimated.position.x_val,
                                 state.kinematics_estimated.position.y_val,
                                 state.kinematics_estimated.position.z_val]
elif (f_p_n < 0 < f_p_e) or (f_p_n < 0 and f_p_e == 0):
    if i % 2 == 0:
        p = []
```

```
s = []
t = []
q = []
for num in range(1, i, 2):
    sott1 = num * ts
    a = f_p_n + sott1
    P = (a, f_p_e, h, v)
    p.append(P)
    S = (a, f_p_e - ll, h, v)
    s.append(S)
for mol in range(2, i + 1, 2):
    sott2 = mol * ts
    b = f_p_n + sott2
    T = (b, f_p_e - ll, h, v)
    t.append(T)
    Q = (b, f_p_e, h, v)
    q.append(Q)
Total = []
for n in range(len(q)):
    total = [p[n], s[n], t[n], q[n]]
    Total.append(total)
Totale = []
for n in range(len(Total)):
    for a in range(4):
        totale = Total[n][a]
        Totale.append(totale)
print(Totale)
for n in range(len(Totale)):
    client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                               vehicle_name=vehicle_name)

    list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val,
                 state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])

    time.sleep(time_required * 0.9)
    a = True
```

```
        while a:
            if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                    p_attuale[
                        1] < (
                    list1[1] + 2) and (
                    list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                a = False
            else:
                time.sleep(0.1)

                state = client.getMultirotorState(vehicle_name)
                p_attuale = [state.kinematics_estimated.position.x_val,
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i + 1, 2):
        sott1 = num * ts
        a = f_p_n + sott1
        P = (a, f_p_e, h, v)
        p.append(P)
        S = (a, f_p_e - ll, h, v)
        s.append(S)
    for mol in range(2, i + 2, 2):
        sott2 = mol * ts
        b = f_p_n + sott2
        T = (b, f_p_e - ll, h, v)
        t.append(T)
        Q = (b, f_p_e, h, v)
        q.append(Q)
    Total = []
    for n in range(len(s)):
        total = [p[n], s[n], t[n], q[n]]
        Total.append(total)
    Totale = []
    for n in range(len(Total)):
        for a in range(4):
            totale = Total[n][a]
            Totale.append(totale)
    limit = len(Totale) - 2
    for n in range(limit):
```

```
            client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                                       vehicle_name=vehicle_name)

            list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
            state = client.getMultirotorState(vehicle_name)
            p_attuale = [state.kinematics_estimated.position.x_val,
                         state.kinematics_estimated.position.y_val,
                         state.kinematics_estimated.position.z_val]

            distance = sqrt(
                (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                        list1[2] - p_attuale[2]) ** 2)
            time_required = (distance / list1[3])

            time.sleep(time_required * 0.9)
            a = True
            while a:
                if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                        p_attuale[
                            1] < (
                        list1[1] + 2) and (
                        list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                    a = False
                else:
                    time.sleep(0.1)

                    state = client.getMultirotorState(vehicle_name)
                    p_attuale = [state.kinematics_estimated.position.x_val,
                                 state.kinematics_estimated.position.y_val,
                                 state.kinematics_estimated.position.z_val]
elif (f_p_n < 0 and f_p_e < 0) or (f_p_n == 0 and f_p_e < 0):
    if i % 2 == 0:
        p = []
        s = []
        t = []
        q = []
        for num in range(1, i, 2):
            sott1 = num * ts
            a = f_p_n + sott1
            P = (a, f_p_e, h, v)
            p.append(P)
            S = (a, f_p_e + ll, h, v)
            s.append(S)
        for mol in range(2, i + 1, 2):
```

```
    sott2 = mol * ts
    b = f_p_n + sott2
    T = (b, f_p_e + ll, h, v)
    t.append(T)
    Q = (b, f_p_e, h, v)
    q.append(Q)
Total = []
for n in range(len(q)):
    total = [p[n], s[n], t[n], q[n]]
    Total.append(total)
Totale = []
for n in range(len(Total)):
    for a in range(4):
        totale = Total[n][a]
        Totale.append(totale)
print(Totale)
for n in range(len(Totale)):
    client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                               vehicle_name=vehicle_name)

    list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
    state = client.getMultirotorState(vehicle_name)
    p_attuale = [state.kinematics_estimated.position.x_val,
                 state.kinematics_estimated.position.y_val,
                 state.kinematics_estimated.position.z_val]

    distance = sqrt(
        (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                list1[2] - p_attuale[2]) ** 2)
    time_required = (distance / list1[3])

    time.sleep(time_required * 0.9)
    a = True
    while a:
        if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                p_attuale[
                    1] < (
                list1[1] + 2) and (
                list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
            a = False
        else:
            time.sleep(0.1)

            state = client.getMultirotorState(vehicle_name)
```

```python
                p_attuale = [state.kinematics_estimated.position.x_val,
                             state.kinematics_estimated.position.y_val,
                             state.kinematics_estimated.position.z_val]
else:
    p = []
    s = []
    t = []
    q = []
    for num in range(1, i + 1, 2):
        sott1 = num * ts
        a = f_p_n + sott1
        P = (a, f_p_e, h, v)
        p.append(P)
        S = (a, f_p_e + ll, h, v)
        s.append(S)
    for mol in range(2, i + 2, 2):
        sott2 = mol * ts
        b = f_p_n + sott2
        T = (b, f_p_e + ll, h, v)
        t.append(T)
        Q = (b, f_p_e, h, v)
        q.append(Q)
    Total = []
    for n in range(len(s)):
        total = [p[n], s[n], t[n], q[n]]
        Total.append(total)
    Totale = []
    for n in range(len(Total)):
        for a in range(4):
            totale = Total[n][a]
            Totale.append(totale)
    print(Totale)
    limit = len(Totale) - 2
    for n in range(limit):
        client.moveToPositionAsync(Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3],
                                   vehicle_name=vehicle_name)

        list1 = [Totale[n][0], Totale[n][1], Totale[n][2], Totale[n][3]]
        state = client.getMultirotorState(vehicle_name)
        p_attuale = [state.kinematics_estimated.position.x_val,
                     state.kinematics_estimated.position.y_val,
                     state.kinematics_estimated.position.z_val]

        distance = sqrt(
```

```
                                    (list1[0] - p_attuale[0]) ** 2 + (list1[1] - p_attuale[1]) ** 2 + (
                                            list1[2] - p_attuale[2]) ** 2)
                                time_required = (distance / list1[3])

                                time.sleep(time_required * 0.9)
                                a = True
                                while a:
                                    if (list1[0] - 2) < p_attuale[0] < (list1[0] + 2) and (list1[1] - 2) < \
                                            p_attuale[
                                                1] < (
                                            list1[1] + 2) and (
                                            list1[2] - 1) < p_attuale[2] < (list1[2] + 1):
                                        a = False
                                    else:
                                        time.sleep(0.1)

                                        state = client.getMultirotorState(vehicle_name)
                                        p_attuale = [state.kinematics_estimated.position.x_val,
                                                     state.kinematics_estimated.position.y_val,
                                                     state.kinematics_estimated.position.z_val]

    def landing(self, drone_number):

        with self._lock:

            vehicle_name = 'Drone' + str(drone_number + 1)

            client.moveToPositionAsync(x_home, y_home, -3, 3, vehicle_name=vehicle_name)

            client.moveByVelocityZAsync(x_home, y_home, z_home, 2, airsim.DrivetrainType.MaxDegreeOfFreedom,
                                        airsim.YawMode(False, 0),
                                        vehicle_name='Drone' + str(drone_number + 1)).join()

            client.armDisarm(False, vehicle_name='Drone' + str(drone_number + 1))

            client.enableApiControl(False, vehicle_name='Drone' + str(drone_number + 1))
```

```python
if __name__ == "__main__":

    program = MainProgram()

    with concurrent.futures.ThreadPoolExecutor(max_workers=population) as executor:
        for index in range(population):
            executor.submit(program.execution, index)

    with concurrent.futures.ThreadPoolExecutor(max_workers=population) as executor:
        for index in range(population):
            executor.submit(program.landing, index)
```