Design and implementation project of an aircraft mobile node module for the SeamSAT-LEO constellation simulator

Document:

Project Report

Author:

David Anton Dobarro

Director/Co-director:

Juan Jose Alins Delgado/Jorge Mata Diaz

Degree:

Bachelor in Aerospace Technologies Engineering

Examination Session:

Spring, 2023.

BACHELOR FINAL THESIS

Abstract

LEO satellite networks have become a growing item of interest in the last years. This new technology will revolutionize the internet, allowing communications all along the globe. Furthermore, thanks to LEO networks, it will now become possible to track flying aircraft with global coverage. Previous to LEO and GEO satellite networks, airlines did not have means to properly track aircraft flying through inter-oceanic routes. Hence, this technology will improve safety and reliability levels of aviation all over the world. As a result, air traffic is expected to be optimised, which shall in turn, allow to fly more sustainably.

Such interest in this new technology has motivated the development of this project. The aim of this thesis is to design and implement an aircraft mobile node module for the SILLEO-SeamSAT constellation simulator, which would allow to study communications between aircraft and LEO satellite networks. For this purpose, a software tool has been developed in order to save and process real flights which shall be uploaded to the simulator. The results of this project will allow to determine the viability and performance of communications between aircraft and LEO satellite networks with global coverage.

Resumen

Las constelaciones de satélites LEO se han convertido en un tema de creciente interés en los últimos años. Esta nueva tecnología revolucionará Internet, permitiendo las comunicaciones en todo el planeta. Además, gracias a las redes LEO, ahora será posible rastrear aviones en vuelo con cobertura mundial. Antes de las redes de satélites LEO y GEO, las compañías aéreas no disponían de medios para realizar un seguimiento adecuado de los aviones que volaban por rutas interoceánicas. Por tanto, esta tecnología mejorará los niveles de seguridad y fiabilidad de la aviación en todo el mundo. Como resultado, se espera optimizar el tráfico aéreo, hecho que a la vez, permitirá volar de forma más sostenible.

Tal interés por esta nueva tecnología ha motivado el desarrollo de este proyecto. El objetivo de esta tesis es diseñar e implementar un módulo para aeronaves móviles para el simulador de constelaciones SILLEO-SeamSAT, que permita estudiar las comunicaciones entre aeronaves y constelaciones de satélites LEO. Para ello, se ha desarrollado una herramienta software que permite guardar y procesar vuelos reales que se cargarán posteriormente en el simulador. Los resultados de este proyecto permitirán determinar la viabilidad y el rendimiento de las comunicaciones entre aeronaves y redes de satélites LEO con cobertura mundial.

Contents

List of figures

List of tables

Glossary

LEO	Low Earth Orbit
MEO	Medium Earth Orbit
GEO	High Earth Orbit
GS	Ground Station
WBS	Work Breakdown Structure
Mbps	Mega bytes per second
ISL	Inter-Satellite Links
GSL	Ground to Satellite Links
ASL	Aircraft to Satellite Links
CNS	Communication, Navigation and Surveillance
ATM	Air Traffic Management
VHF	Very High Frequency
HF	High Frequency
ADS-B	Automatic Dependent Surveillance Broadcast
ACARS	Aircraft Communications, Addressing and Reporting System
VDL2	VHF Digital Link Mode 2
ATC	Air Traffic Control
AOC	Aeronautical Operational Control
AAC	Airline Administrative Control
LDACS	L-band Digital Aeronautical Communication System
APNT	Alternative Positioning Navigation and Timing
EUROCONTROL	European Organization for the Safety of Air Navigation

TDMA	Time Division Multiple Access
TDOA	Time Difference of Arrival
AEROMacs	Aeronautical Mobile Airport Communications System
ATN	Aeronautical Telecommunications Network
IP	Internet Protocol
SATCOM	Satellite Communications
GADSS	Global Aeronautical Distress and Safety System
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
GBAS	Ground-Based Augmentation System
SBAS	Satellite-Based Augmentation System
ILS	Instrument Landing System
DME	Distance Measuring Equipment
VOR	VHF Omnidirectional Range
NDB	Non-Directional Beacon
ADF	Automatic Direction Finder
MLAT	Multilateration
OGN	Open Glider Network
FLARM	Flight Alarm
PSR	Primary Surveillance Radar
SSR	Secondary Surveillance Radar
TLE	Two-line Element set
API	Application Programming Interface
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
XML	Extensible Markup Language
GUI	Graphical User Interface

Chapter 1

Introduction

1.1 Aim

The aim of this project is to design and implement an aircraft mobile node module for the SILLEO-SeamSAT constellation simulator, as well as implement a tool which shall gather real aircraft flight routes from an external flight database and pre-process it for later use in the SILLEO-SeamSAT simulator[1].

1.2 Scope

The Silleo-SeamSAT simulator is an advanced improvement of the Silleo-SCNS simulator, further modified by the UPC research group TUAREG.

This project shall improve the features of the SILLEO-SeamSAT, which currently is a simulator that allows simulating the movement of satellites in LEO constellations and the communication links between them and ground stations (GS). The new aircraft mobile node module for SILLEO-SeamSAT should support the simulation of aircraft in flight to interact with the satellite network. In order to implement this module, two main tasks shall be developed.

On one hand, real flight route's data will be downloaded from a chosen flight databases (prior to that an analysis of different flight databases shall be conducted in order to know which one will be the one used in the project). Then, the data will be processed in order to format it in an appropriate manner so that it may be used in the next task. This project does not take into account the implementation of real-time aircraft data, hence all the information shall be restricted to that available on the historical data of the used website.

On the other hand, this data will be used in SILLEO-SeamSAT by including a new software module that will be both, designed and implemented. This software module shall enable the interaction of aircraft in flight with the satellite network at a specified timeline.

The study of the efficiency of the link's interaction between aircraft and the satellite network is an optional feature of the project and will only be implemented if the deadline allows it.

Finally, this project will include for the first part:

- Research of aircraft flight data
  - ADS-B
  - State vectors
- Search and selection of the most appropriate "on-line" aircraft flight databases
- Selection of the software libraries needed to access the chosen databases
- Flight route propagation from discrete input flight data
- Implementation of a software application for automatically downloading aircraft flight data

Regarding the second part, it will include:

- Introductory review of LEO satellite constellations
  - Artificial satellite constellations: Walker Star and Walker Delta
  - ISL and GSL
    - Geometrical and analytical definition
    - Constraints and limitations (atmospheric, link budget and attitude control)

- ASL
  - Determination of number of satellites in range
  - Limiting the number of antennas in aircraft
- Review of SILLEO-SeamSAT's software architecture
- Proposal for the software solution

1.3 Requirements

This project requires the following items to be fulfilled:

- The final code will be implemented using the programming language, Python.
- In order to keep all the documentation as similar as possible, all academic documents shall be written in English, which so far, has been the language used in previous projects related to SILLEO.
- The expected time invested in the project is about 300 hours, which would correspond to 12 ECTS (of 25 hours each).
- The data gathered from the chosen flight database must be in a format which may enable the pre-process of the data in order to use it in the Python scripts.
- The report must be redacted clearly and using the appropriate terminology of the matter in hand.
- A meeting with the thesis director will be conducted once a week.
- The Project Charter's deadline is March 17th, 2023. Three tracking reports shall be conducted within the timeline of the project. The final report's deadline is June 21st, 2023.

1.4 Justification

LEO satellite networks have become a growing item of interest in the last years because it is the technology that should revolutionize the internet, allowing communications all along the globe. The European Union will spend around 2.4 billion € in the next 5 years in a plan which aims to deliver faster communications with an improved broadband access to the region via a LEO satellite system. Big companies such as Space X or Amazon are investing in this field as well, with projects such as Starlink and Kuiper, respectively [2].

Starlink is the world's first and largest satellite constellation which uses a LEO orbit to deliver fast, low-latency broadband internet across the globe. Regarding aviation, it may deliver up to 350 Mbps and a latency as low as 20 ms to each plane which would allow passengers to use high demanding data rate activities such as gaming or streaming (functions that were not previously available). The access would cover land, the oceans and even the poles [3].

Project Kuiper aims to deliver low latency, fast broadband internet connectivity to remote and underserved communities around the world with their LEO satellite network [4].

These constellations of satellites may be comprised of thousands of interconnected satellites. The SILLEO constellation's simulator is a tool which pretends to generate and simulate such networks [5]. This function is already implemented and the aim of this thesis is to improve it by adding another node which would allow aircraft in flight to interact with the satellite network.

Using satellite constellation's networks across the globe, airlines may track aircraft in flight even covering the poles or the ocean. This would improve both security and redundancy of inter-continental routes [6]. Hence, cases such as the MH370 which disappeared nine years ago due to unknown reasons would not happen any more (or at least, they should be even more rare) [7]. Previous to LEO and GEO satellite networks, airlines did not have means to properly track aircraft routes above the ocean, hopefully this will change in the near future. Inclusion of aircraft nodes in the SILLEO-SeamSAT will allow to study and analyse interaction, coverage and communications for ATM, specially in inter-oceanic air routes where CNS services have not had until now a proper infrastructure to be provided (until recently CNS has been mostly ground-based)[8]. It is expected that this new technology will improve safety levels of aviation all over the world as well as optimise air traffic. At a point of the current interest, some new projects arise, as Startical and ECHOES.

Startical is a project which aims to provide surveillance as well as real-time voice and data communications using a constellation of more than 200 LEO satellites. Their solution would combine ADS-B surveillance, real-time VHF voice services and air traffic data in order to achieve a space based CNS. This would optimise air traffic over remote and inter-oceanic air routes, enabling aircraft to use different routes in such cases (hence helping to unsaturate current air routes), reduce the minimal separation between aircraft and overall, enable safe inter-oceanic air traffic to operate as continental traffic [9] [10]. This project is the result of a partnership between Indra and ENAIRE and seeks to become a leading provider of satellite ATM technology in the near future. Indra is a leading company in the sector of ATM systems. Nowadays, about 85% of aircraft use their ATM technology. ENAIRE, however, is not an ATM systems provider. ENAIRE is the air navigation manager in Spain, being the fourth in terms of size in all Europe.

On the other hand, ECHOES is a project being lead by Startical in a partnership with Indra, ENAIRE, DLR and NAV Portugal, among others. ECHOES seeks to demonstrate the feasibility of space-based VHF communications in ATM [11]. In order to do so they intend to investigate and develop technologies relating inter-satellite links, reception of VHF communications and on-site processing of data. They intend to test their technology by developing, manufacturing and finally launching 2 LEO satellites. Besides testing the technology, they will also provide some insight regarding the required minimum separation between aircraft over inter-oceanic routes.

Chapter 2

State of the art

In this chapter an overview on the main aspects and base knowledge required to understand this project is detailed.

In this chapter the following items shall be discussed:

- Types of orbits
- LEO networks implications in ATM
- CNS and Space Based CNS

2.1 Types of orbits

An orbit is the path that an object in space follows around another one due to gravity. There are different types of orbits which may offer satellites different view points, some may focus on a single area while others may pass over many different places along their path. Even so, orbits may be classified into three categories [12] [13][14]:

- LEO orbits
- MEO orbits
- GEO orbits

2.1.1 LEO

LEO orbits are used for scientific purposes. They have an altitude comprised between 180-2000 km. They don't need to follow a specific path, hence their plane may be tilted and, as a result, they have a lot of routes available. Since this kind of orbit is the nearest to Earth's surface, it enables taking high resolution satellite images.

The speed of a satellite orbiting Earth goes as follows:

$$v = \sqrt{\frac{GM}{R}}\ [m/s] \tag{2.1}$$

Where G=6.673×10^{-11} $\frac{Nm^2}{kg^2}$, M is the mass of the Earth and R is the radius orbit (distance taken from the satellite's position to the Earth's center).

From equation 2.1, it may be seen that as the satellite is closer to Earth, its gravitational pull is bigger, hence it needs a higher speed in order to keep orbiting and not crash unto the Earth.

Satellites in LEO orbit usually have a speed about 8 km/s, which means that they orbit the Earth periodically in about 90 minutes (or about 16 times a day). This high speed prevents single satellites in this kind of orbit from being useful in telecommunications. Instead, satellites in this type of orbit usually work in a net of constellations in order to enable coverage in certain areas around the globe.

Van Allen belts

The Van Allen belts are a collection of energetic charged particles, gathered in a specific zone by Earth's magnetic field. These belts were discovered in 1958 by James Van Allen. They usually work as a protective layer which shields the Earth from the fastest high-energy electrons. The inner belt is spread from 600 to 10000 km above Earth's surface, whilst the outer belt is spread from 14000 to 60000 km above Earth's surface. However, the space between the belts may temporarily disappear due to a displacement of the outer belt caused by strong solar winds or a remarkable solar eruption. The inner belt, in turn, may be displaced closer to the Earth's surface.

These radiation belts located in the inner section of Earth's magnetosphere must be taken into account when a space mission is designed, because they can damage satellites components. Solar cells, circuits and sensors may be damaged by radiation and even a single hit may damage a satellite's performance temporarily or even permanently. LEO satellites may orbit inside the inner belt of the Van Allen belts, hence they must be shielded to protect its components from radiation [15] [16].

2.1.2 MEO

MEO orbits are commonly used for navigation purposes and monitoring a designated area (Molniya orbits would be an example for this kind of purpose). They have an altitude comprised between 2000-36000 km (approximately a tenth of the distance to the Moon).

Molniya orbits, previously mentioned, are a type of orbit invented by the Russians. This type of orbit performs well at high latitudes, has a high eccentricity and a period about 12 hours. Molniya orbits are special because due to their high eccentricity, they spend most of their time close to the top of their orbit (farthest from the Earth), which makes them perfect for keeping coverage of a certain area. They are commonly used for telecommunications at high latitudes or spying.

2.1.3 GEO

GEO orbits are mainly used in weather forecast applications as well as some communications. They have an altitude higher than 36000 km. There is a special GEO orbit (geosynchronous orbit) at about 36000 km above the Earth's surface which has a period of a sidereal day (23 hours, 56 minutes and 4 seconds), which is the time it takes for the Earth to complete a full rotation about its axis with respect to distant celestial objects (such as stars outside the Solar System). This orbit in particular is special because it matches Earth's rotation speed and orbits above the Equator following Earth's rotation from west to east.

2.2 LEO networks implications in ATM

Placing satellites orbiting around the Earth may have many implications, such as telecommunications, navigation, weather forecast or even observe celestial objects for astronomical purposes. Regarding ATM, the coverage provided by these networks may enable building a proper CNS service infrastructure for inter-oceanic routes. In LEO constellations, communications latency is short enough to deal with voice communications and CNS (Communications, Navigation and Surveillance) services in ATM. This is a feature that GEO constellations are incapable to do.

Nowadays, some companies have already started some projects to build a LEO satellite network. These projects have different approaches depending on the company's aim: some are business focused, while others focus on research. There are some that focus on providing Internet access globally, while others focus on satellite communications, among others.

2.2.1 Iridium Next

Iridium completed their fully operational LEO satellite network at about 800 km above Earth's surface in 1998 and they upgraded it by replacing all previous satellites in 2019. Their network provides stronger signals and faster connections (through smaller antennas which have a lower power consumption) than most of the GEO networks used nowadays. Their network has coverage all over the globe, even at high latitudes and the poles. The signals used by Iridium's network have an L-band frequency, which have a lesser impact in performance due to weather events (like snowing, raining, ...) than the frequencies used by most GEO networks.

Their network is connected using a cross-link system, which means that each satellite is connected to other 4 satellites (when available): two on the sides, one above and one below, as shown in figure 2.1.

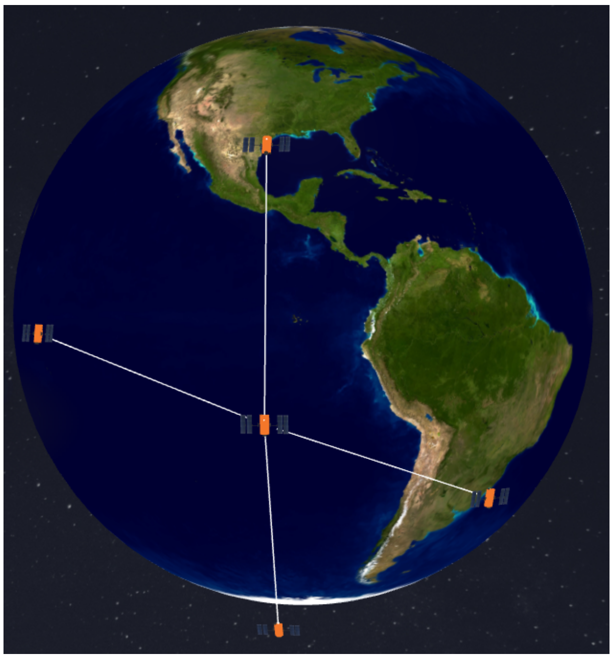


Figure 2.1: Iridium's network cross-link [17]

This kind of configuration, enables data to be rerouted and transmitted as fastest as possible. Moreover, it provides redundancy to the network (ensuring that the data will receive its destination even if a satellite is not available) and optimization (when available, data will follow the fastest route) [17].

2.2.2 Startical

Startical is a project which aims to provide surveillance as well as real-time voice and data communications using a constellation of more than 200 LEO satellites. Their solution would combine ADS-B surveillance, real-time VHF voice services and air traffic data in order to achieve a space based CNS. This would optimise air traffic over remote and inter-oceanic air routes, enabling aircraft to use different routes in such cases (hence helping to unsaturate current air routes), reduce the minimal separation between aircraft and overall, enable safe inter-oceanic air traffic to operate as continental traffic [9] [10]. This project is the result of a partnership between Indra and ENAIRE and seeks to become a leading provider of satellite ATM technology in the near future. Indra is a leading company in the sector of ATM systems. Nowadays, about 85% of aircraft use their ATM technology. ENAIRE, however, is not an ATM systems provider. ENAIRE is the air navigation manager in Spain, being the fourth in terms of size in all Europe.

2.3 CNS and Space Based CNS (SB-CNS)

CNS infrastructure and the radio frequencies it requires are the foundation of air transport as we know it nowadays. It is the source to deliver safe, redundant, efficient and sustainable air traffic performance.

2.3.1 Space segment

The SB-CNS refers to all satellites and their correspondent infrastructure that enable CNS for several applications. This infrastructure takes part in a lot of sectors such as aviation and telecommunications, among others. The main feature of this kind of system is the wide coverage they provide for their applications.

In terms of communications, this segment encompasses all communication satellites that serve as a long distance intermediary between ground-based communications systems.

Regarding navigation, there are several satellite navigation systems which enable accurate positioning globally. These types of systems are further explained below.

Finally, all the sensors and imaging devices that capture data may be used for monitoring and surveillance purposes. One of the most common uses of this tool is to provide useful data for meteorology or environmental monitoring.

2.3.2 Ground segment

Regarding the ground segment, it is composed by all of the GS and the infrastructure that allows them to communicate with either AC or satellites. These GS are the ones responsible for keeping communications with the other segments and allow connectivity between different points on the globe. This segment is responsible of monitoring AC and satellites and ensure an appropriate network operation.

2.3.3 Air segment

The air segment, on the other hand, refers to all AC and the infrastructure that allows communication with GS. The systems commonly used for CNS in this segment shall be explained in the following section.

2.3.4 Communications

ACARS is a digital link system used since 1978, which enables the transmission of messages between aircraft and GS. At first it used VHF data-link channels exclusively. However, with the innovation in telecommunication technologies, there have been implemented other means of data transmission in the last years [18] [19] [20].

Depending on the content of the message transmitted, ACARS messages may be categorized as:

- ATC: Messages that contain either request from aircraft of clearances, ATC issues of clearances or instructions to aircraft.
- AOC and AAC: Messages used to communicate an aircraft with its base. These messages shall follow the guidelines of ARINC Standard 618.

The current data-link methods used to transmit ACARS messages are the following ones:

VHF-HF

VHF and HF are the two traditional types of radio communication systems used in the aviation industry. They both have their own advantages and limitations, and they are used for different types of communication depending on the situation.

VHF is the most commonly used communication system in aviation, and it operates in the frequency band between 30-300 MHz. It is used for communication between aircraft and GS, as well as for communication between aircraft. VHF offers several advantages, including high clarity and low noise, making it ideal for voice communication. Additionally, VHF has a relatively short range, which helps to reduce the risk of interference from other radio signals. Nowadays, the VHF spectrum is pretty congested, and as a result, the need for other radio communication technologies which may ensure safe and reliable ATC has arisen.

HF, on the other hand, operates in the frequency band between 3-30 MHz. It is mostly used for long-range communication, such as over oceanic areas where VHF coverage is not available. HF offers several advantages, including the ability to communicate over long distances, even in remote areas. However, HF has several limitations, including poor clarity and a high level of noise, which can make communication difficult. Hence, this technology does not currently have means to support safe and reliable ATC with global coverage [21].

VDL2

It is an ICAO standardized radio technology which operates at VHF band. It is mainly used in the aviation industry for ATM purposes and provides digital means of communication for different applications, such as aircraft surveillance, flight planning or weather data transfer. The European data-link system is based on the ATN system and this radio technology as means to communicate aircraft with GS. Currently, as new radio technologies appear, the industry has begun the process to incorporate them in the near future.

VDL2 offers several advantages over traditional analog communication systems, including improved reliability, higher data transfer rates, and reduced interference from other radio signals. It also provides enables operating multiple channels simultaneously, allowing for more efficient use of the available communication bandwidth.

In terms of global coverage, VDL2 has a range of approximately 300-400 km (similar to VHF), depending on the altitude of the aircraft and the location of the ground station. It is currently used in various regions around the world, including Europe, Asia, and North America. The system is expected to be implemented more widely in the coming years as more countries adopt VDL2 as the standard communication system for air traffic management [22].

LDACS

LDACs are a type of communication system used in aviation. These systems operate in the L-band frequency range, which is between 1 and 2 GHz, and they are designed to provide reliable communication between aircraft and GS.

According to EUROCONTROL [23], LDACs offer several advantages over traditional communication systems. One of the main advantages is that LDACs can support a much higher data rate than other systems, which makes it possible to transmit large amounts of data quickly and efficiently. Moreover, LDACs are designed to be more robust and reliable than other systems, which helps to ensure that communication between aircraft and GS remains uninterrupted even when weather conditions are not ideal.

LDACs can be used for a variety of different types of communication, including voice communication and data communication, and the can even be used for navigation by ranging data. They are used by air traffic control organizations around the world, and they are an important tool for ensuring the safety and efficiency of air travel [24].

AEROMacs

AEROMacs is a type of data link system used in aviation for wireless broadband communication between aircraft and GS. AEROMacs is based on the WiMax technology and it operates in the C-band between 5091-5150 MHz, and it is designed to provide secure and reliable data communication for airport operations and air traffic control. They enable short-range communications (about 50 to a 100 km).

AEROMacs is designed to support a wide range of applications, including airport surface management, ATC, and aircraft surveillance. The system is also designed to be highly reliable, with multiple layers of redundancy built in to ensure that communication remains uninterrupted even in adverse weather conditions.

AEROMacs uses different communication protocols, including the ATN and IP protocols (which are used for ATC communication and data communication between aircraft and GS, respectively) among others.

One of the key advantages of AEROMacs is that it is designed to be highly secure. The system uses encryption and other security measures to protect communication from unauthorized access and to prevent interference from other radio signals. Additionally, it is designed to relieve the congested VHF spectrum. This makes AEROMacs suitable for use in sensitive applications, such as ATC and airport security [25] [26].

SATCOM

SATCOM is a space based technology that allows communication between aircraft and GS via satellites. It is widely used in the aviation industry for various applications, such as voice communication, data communication, and position reporting.

One of the main advantages it offers over traditional communication systems is that it provides a global coverage, allowing communication between aircraft and GS anywhere in the world. Additionally, it is designed to be highly reliable and efficient, which helps to ensure that communication remains uninterrupted even in adverse weather conditions.

SATCOM operates in various frequency bands, depending on the application and the type of satellite used. For example, L-band and Ku-band frequencies are commonly used for voice and data communication, while Ka-band is used for high-speed data communication.

According to EUROCONTROL, SATCOM is increasingly being used for air traffic control communication. The system, which is designed to provide reliable and secure data communication between aircraft and GS as previously mentioned, is being used to support different air traffic control applications, such as aircraft surveillance, weather data transfer or flight planning.

SATCOM technology is still being improved and evolving, and new advancements are being made to improve the performance and capabilities of the system. For example, the development of the GADSS has led to the implementation of new SATCOM features, such as automatic distress and safety communication [27] [28].

2.3.5 Navigation

GNSS

GNSS is a positioning and timing technology that uses a network of satellites to provide accurate location information to users on the ground, in the air, and in space. The most well-known GNSS is the United States' GPS, but there exist other GNSS systems such as Russia's GLONASS, Europe's Galileo or China's BeiDou.

In the aviation industry, GNSS technology is used for a variety of applications, including navigation, surveillance, and timing. GNSS systems provide aircraft with accurate position information, allowing for precise navigation and the ability to follow predetermined flight paths. GNSS also enables air traffic controllers to track aircraft movements more accurately and efficiently, improving safety and reducing the risk of collisions. In the event of an emergency, GNSS technology can also be used to provide accurate location information to search and rescue teams.

The implications of GNSS in the aviation industry are significant. With its ability to provide accurate location information, GNSS technology has enabled the development of more efficient flight paths and has reduced the need for ground-based navigation aids. This has resulted in cost savings for airlines and a reduction in carbon emissions, as aircraft can fly more direct routes. Hence, improving environmental sustainability [29] [30].

GBAS and SBAS

GBAS is a navigation system used in the aviation industry that provides precision approach and landing guidance to aircraft. GBAS is a satellite-based system that uses ground-based transmitters to augment GNSS signals, improving accuracy and reliability.

This system works by broadcasting correction signals to aircraft that are approaching an airport. These correction signals improve the accuracy of GNSS signals, allowing aircraft to navigate more precisely and safely. GBAS also provides information on the approach path, helping pilots to navigate more efficiently and safely.

GBAS has the potential to improve airport capacity and efficiency by allowing for more precise and reliable approach and landing guidance. This can reduce delays and improve safety, particularly in adverse weather conditions. It may also reduce costs for airlines by reducing the need for ground-based navigation aids and improving aircraft efficiency. With its ability to provide more precise and efficient approach and landing guidance, GBAS can also reduce fuel consumption and carbon emissions [31] [32].

On the other hand, SBAS is a navigation system used in the aviation industry that provides increased accuracy and integrity to GNSS signals. SBAS uses a network of ground-based reference stations and geostationary satellites to broadcast correction signals to GNSS receivers on board aircraft.

SBAS compare the signals received from GNSS satellites with those received from GS. It then calculates correction signals that are broadcast to aircraft using geostationary satellites. These correction signals improve the accuracy and integrity of GNSS signals, allowing for more precise navigation.

SBAS can improve the safety and efficiency of aviation operations by providing more accurate and reliable navigation signals. This can lead to reduced delays, improved fuel efficiency, and increased capacity at airports.

SBAS can also support operations in remote and challenging environments, such as mountainous regions or areas with limited ground-based navigation infrastructure. This can improve the accessibility and safety of air travel in these areas [33] [34].

ILS

ILS is a ground-based navigation aid used in the aviation industry to guide aircraft to a safe landing in low visibility conditions. ILS uses a combination of radio signals and visual aids to provide pilots with vertical and horizontal guidance during the final approach and landing phases of flight.

This system consists of two main items:

- Localizer: It provides lateral guidance to the aircraft, ensuring that it stays aligned with the centerline of the runway.
- Glide slope: It provides vertical guidance, indicating the aircraft's position above or below the ideal descent path.

ILS is a critical component of modern aviation, as it enables pilots to land aircraft safely in low visibility conditions, which may be due to rain, snow, fog... It is especially important for commercial aviation, where delays and cancellations due to adverse weather conditions can result in significant costs for airlines and passengers.

It is an essential system for airport operations, as it allows aircraft to land safely and efficiently in busy and congested airspace. Without ILS, airports would need to restrict operations during adverse weather conditions, which would result in delays and disruption for passengers and airlines [35] [36].

DME-VOR-NDB

DME, VOR, and NDB are all ground-based navigation aids used in the aviation industry to help pilots navigate and locate their position during flight.

DME uses radio signals to measure the distance between aircraft and DME's GS. This information is then displayed on the aircraft's instruments, providing the pilot with accurate information on their position and distance from a specific location (a runway for example).

VOR is another ground-based navigation aid that uses radio signals to provide pilots with information on their position and heading. VOR works by transmitting two signals, which are received by the aircraft's VOR receiver. By measuring the difference between the two signals, the receiver can determine the aircraft's position and heading relative to the VOR station.

NDB is a radio beacon that uses a low-frequency signal to provide pilots with directional information. NDB works by transmitting a signal in all directions, which is received by the aircraft's ADF instrument. This instrument, in turn, indicates the direction of the beacon relative to the aircraft.

DME, VOR, and NDB are essential navigation aids in the aviation industry, as they provide pilots with accurate and reliable information on their position and heading during flight. They are especially important for instrument approaches and in areas with poor visibility, where visual navigation is not possible [37] [38] [39] [40] [41].

2.3.6 Surveillance

ADS-B

ADS-B is a technology used in the aviation industry for aircraft surveillance and tracking. It works by broadcasting an aircraft's location, altitude, velocity, and identification to other aircraft and GS. This allows air traffic controllers to track aircraft with more accuracy and efficiency compared to traditional radar systems.

ADS-B technology uses GPS receivers and onboard transmitters to broadcast the aircraft's state vector, intent and other information. The GPS receiver calculates the aircraft's position and the onboard transmitter broadcasts this information along with other data to other aircraft and GS.

One of the main benefits of ADS-B is its ability to provide more accurate and reliable information to air traffic controllers compared to traditional radar systems, as well as being more cost-effective than them. Additionally, it allows pilots to see other aircraft in their vicinity on cockpit displays.

ADS-B is becoming increasingly important in the aviation industry, as many countries are mandating its use in certain airspace. In the United States, the FAA mandated that all aircraft flying in most controlled airspace must be equipped with ADS-B Out by January 1, 2020. In Europe, ADS-B became mandatory in 2020 in all airspace where a transponder is currently required [42] [43] [44].

Satellite based tracking

Satellite-based tracking refers to the use of satellite technology to monitor the position and movement of aircraft. This technology allows for real-time tracking and monitoring of flights, which has numerous benefits for the aviation industry.

Satellite-based tracking works by receiving data from an aircraft's GPS system, which provides precise location information. This information is then transmitted to GS or satellite networks, which can then track the aircraft's movement in real-time. The technology is especially useful for tracking aircraft in remote or oceanic areas where traditional radar coverage may not be available.

One implementation of satellite-based tracking in the aviation industry is through the use of ADS-B technology. As previously mentioned, ADS-B is a surveillance technology that uses satellite-based positioning to provide aircraft with accurate and reliable position information. Aircraft equipped with ADS-B broadcast this information to GS or other aircraft, allowing for real-time tracking and monitoring of aircraft movements.

ADS-B technology may be implemented in satellite-based tracking systems by equipping satellites with ADS-B receivers for receiving position and other data. This data may be transmitted via a satellite network and later it may be used by GS or other aircraft equipped with ADS-B receivers to track and monitor the aircraft's position in real-time. Satellites equipped with ADS-B receivers may gather data from aircraft outside the terrestrial ADS-B network's coverage and transmit it to GS or satellite networks.

The implementation of satellite-based tracking and ADS-B technology has several advantages for the aviation industry. This technology can improve ATM, enhance safety, and provide more accurate and reliable tracking of flights. It can also aid in the investigation of accidents or incidents by providing precise data on the location and movement of aircraft [45] [46]. Note that all saved routes in this project have been obtained either via ADS-B or space based ADS-B.

MLAT

MLAT is a surveillance technology used in the aviation industry to track the position of aircraft in flight. Unlike traditional radar systems, which rely on the aircraft's transponder to provide a signal, MLAT uses the TDOA of signals from multiple ground-based receivers to determine the aircraft's location.

In MLAT, multiple receivers are used to receive the signals transmitted by the aircraft. Each receiver records the time of arrival of the signal, and the difference in time between the signals received by different receivers is used to calculate the position of the aircraft. This is done by using the known locations of the receivers and solving for the aircraft's position using a mathematical algorithm.

MLAT has several advantages over traditional radar systems. It allows for more precise tracking of aircraft, as it can determine the position of the aircraft with greater accuracy. It is also more cost-effective than radar, as it uses existing infrastructure such as ADS-B receivers to collect data.

In the aviation industry, MLAT is used to track aircraft in areas where traditional radar coverage is limited or nonexistent, such as in remote or mountainous regions. It is also used in areas where radar coverage is limited by terrain or where there are other obstacles that block radar signals. MLAT technology can also be used to enhance safety and efficiency by providing more accurate and reliable data for air traffic control.

ADS-B technology can be implemented in MLAT systems to enhance the accuracy of aircraft tracking. ADS-B signals can be used as a reference signal to improve the accuracy of the MLAT calculations. This combination of technologies provides a more robust and reliable surveillance system for air traffic control [47] [48].

OGN-FLARM

OGN and FLARM are two technologies used in the aviation industry for glider tracking and collision avoidance.

OGN is a non-profit organization that provides a worldwide network of ADS-B receivers for gliders and other non-powered aircraft. The network allows glider pilots to track their flights in real-time using a variety of devices, including smartphones, tablets, and laptops. OGN works by receiving ADS-B signals from gliders and other aircraft equipped with ADS-B transponders and then forwarding this data to a central server. The server processes the data and makes it available to glider pilots and other interested parties through various applications and websites.

FLARM, on the other hand, is a collision avoidance system used in gliders and other non-powered aircraft. FLARM uses a combination of GPS and radio signals to detect nearby aircraft and provide collision warnings to pilots. It transmits its own position and receives the positions of nearby aircraft equipped with FLARM or similar systems. The system then calculates the potential for a collision and provides visual and audible warnings to the pilots.

On one hand, OGN provides glider pilots with an unprecedented level of situational awareness, allowing them to track their flights and monitor the location of other gliders in real-time. This technology is particularly important in competitive gliding, where pilots need to know the location of their competitors to gain an advantage. FLARM, on the other hand, provides an added layer of safety by alerting pilots to the presence of nearby aircraft and potential collision risks.

In terms of their coverage, OGN has a global network of ADS-B receivers, providing coverage in many remote areas that may not be covered by traditional ADS-B GS. FLARM, on the other hand, is primarily used in Europe, but its use is becoming more widespread in other parts of the world [49] [50].

PSR-SSR

PSR and SSR are two important radar systems used in the aviation industry for ATC and surveillance purposes.

PSR is a radar system that detects the position of an aircraft by measuring the time it takes for a radar signal to bounce off the aircraft and return to the radar antenna. The radar antenna is typically located on the ground, and the PSR is used to detect aircraft that are not equipped with a transponder. PSR provides only basic information such as the aircraft's position and altitude, but it does not provide any identification or other information.

SSR, on the other hand, is a radar system that uses a combination of radar and a transponder on the aircraft to provide more detailed information about the aircraft. SSR works by sending out a signal to the aircraft's transponder, which then responds with the aircraft's identification and other information such as its altitude, speed, and direction of flight. This information is then transmitted back to the radar system, which displays it to the air traffic controller.

SSR is used in conjunction with PSR to provide air traffic controllers with a complete picture of the airspace. By combining the information from both systems, air traffic controllers can track aircraft more accurately and efficiently. SSR is also used for other purposes such as ATC radar beacon system (ATCRBS), where it is used to identify the aircraft and to provide information to the air traffic controllers about the aircraft's altitude and position [51] [52].

2.3.7 Two Line Element (TLE)

A TLE is a standard format for conveying orbital elements of Earth-orbiting objects, including satellites and space debris. It is widely used in the aerospace industry for tracking and predicting the orbits of these objects.

A TLE consists of two lines of data that provide information about the object's position, velocity, and other orbital parameters. The first line contains information about the object's name, its orbital period, and the time at which the TLE was generated. The second line contains data on the object's inclination, eccentricity, argument of perigee, right ascension of the ascending node, mean anomaly, mean motion, and the revolution number at which the TLE was generated.

The TLE is generated by tracking the object from the ground using radar or optical instruments and then using computer algorithms to calculate its current and future positions. TLEs are constantly updated as the object's orbit changes due to various factors, such as atmospheric drag, solar radiation pressure, and gravitational perturbations from other objects.

TLEs have many implications in the aerospace industry, including satellite tracking, space situational awareness, and collision avoidance. They are used by ground-based tracking stations to locate and track satellites in real-time, and to predict their future positions for mission planning and spacecraft operations. TLEs are also used by space agencies and satellite operators to monitor and manage the growing population of space debris in orbit around the Earth, which poses a risk to spacecraft and astronauts [53] [54].

Chapter 3

Methodology and Software Architecture

In order to develop this project, the following items have been used:

- API: In order to download the data of different flight routes, a specific API which accesses the chosen flight database has been used. Then, this data has been pre-processed in order to set it in the proper format for later use.
- API's documentation: In order to understand how the API works, two elements have been taken as a source of information. On one hand, a document which states the main features of the API has been obtained. On the other hand, public API's scripts have been accessed in order to understand how to properly access the required data.
- Python: All scripts within this project have been developed using this open source programming language.
- Gitlab: All scripts have been uploaded to a Gitlab server. This has allowed to store the scripts in the cloud, as well as simplify the way to keep them updated.
- Visual Studio Code: In order to write the scripts using the chosen programming language, this tool has been used. It has simplified the way to work with the scripts. This has been achieved both aesthetically and by reducing (and sometimes skipping) some commands which have been integrated within Visual Studio Code. An example of the later is the way that Visual Studio Code has to push scripts to Gitlab. Instead of a command line (which may also be used), the script must only be staged and pushed with a button.

As stated previously in this project. The main tasks may be distributed into two different categories:

- Aircraft Data Gathering
- Use the output of the first task as an input for the second task as it is shown in figure 3.1

- Aircraft Module Implementation into SILLEO-SeamSAT's software

Regarding the first main task, the data gathering may be achieved using one of the previously mentioned flight databases. Once the data is downloaded, either via an API or via a python shell (Impala from Opensky Network), a code will be designed. This software shall ask the user in what route they are interested in, stating the origin and destination countries, as well as the window period to take into account (within the maximum limit of flights stated). Then, a list of possible routes shall pop up. Later, the user will have to choose a route to save, as well as the code-name for the chosen route. The aim of this developed software (which may be implemented within the SILLEO-SeamSAT tool or not), is not to access the chosen flight database every time a route is chosen, but to have the data previously stored and available to access it at any chosen time. The stored data for each route will consist of the route identification as well as the aircraft's position and speed during all the flight (with a time-step which may vary between 3 and 16 seconds when in-land to up 20 minutes when flying over the ocean).

On the other hand, regarding the second task, the new module software which will be designed for the SILLEO-SeamSAT constellation simulator may be implemented within a total new code or inside the `constellation.py` code. The reason is that this tool should, formally, be separated from the previous code but in case that a lot of redundancies were used, it may be easier to just expand the uses of the previous code. In any case, the `gui.py` and `simulation.py` codes, will be modified as well in order to implement the new data available.

The developed software shall calculate an aircraft's position and speed at all times, given a route and an absolute time (since the beginning of the flight). Then, it shall build the links with the satellites. An optional future feature is to build the links with GS as well, considering that when the satellite is close enough to a GS (usually within a radius of 300-400 km), the aircraft shall establish a link (if possible) with the GS. Whilst, if that is not possible, the link should be established with the satellite constellation network. This feature is not contemplated within the initial scope of this project.

Finally, an overview on the final software architecture is provided in figure 3.1. In the following chapters, an explanation of the software architecture shall be provided in detail.

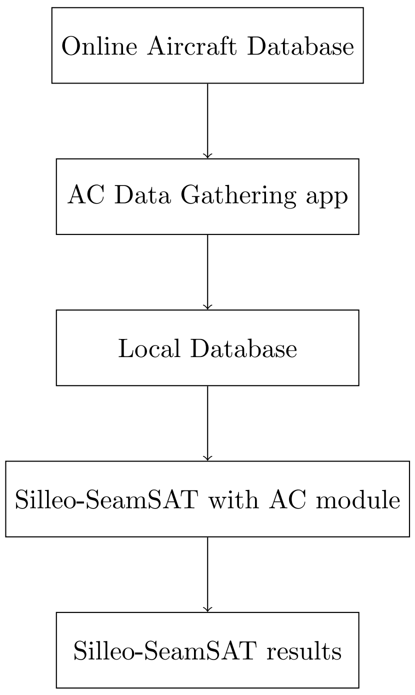


Figure 3.1: Project software overview

The approach and methodology used in each of these tasks are different and they shall be properly exposed in the next chapters.

Chapter 4

Aircraft Data Gathering

Nowadays there are flight databases that may be remotely accessed through the Internet. These databases offer a range of information about aircraft, airports and more. Some of these databases are open-source, while others require a paid license in order to be used. Furthermore, there are even others that allow a mix operation with certain limited open-source data and a greater range of data available with a paid license.

They all offer remote access to their database and provide the user with a specific software that enables the implementation of end-user applications.

In the following section, a discussion about different flight data bases and the services they provide shall be done.

4.1 Flight data bases

4.1.1 Flight Radar 24

Flight Radar 24 is a flight tracking service that provides real-time information about aircraft movement around the world. It is a web-based service that allows users to track commercial and private aircraft, and provides various information about each flight, including aircraft type, altitude, speed and heading, among others.

Flight Radar 24 receives data from a variety of sources, including ADS-B and MLAT receivers and radar data from government and airport authorities. Flight Radar 24 combines this data from various sources to provide a comprehensive view of air traffic around the world. Users can view the location of any aircraft in real-time, and search for specific flights or airports. In addition, Flight Radar 24 provides a range of other features, including alerts for specific flights, airport information, and historical data on flights.

One of the key benefits of Flight Radar 24 is that it allows users to track flights in real-time, providing up-to-date information about delays, cancellations, and other changes to flight schedules. This information can be particularly useful for travelers who need to plan their trips around flight schedules, or for people who are simply interested in tracking aircraft for fun or research purposes.

Certain services of Flight Radar 24 are offered free of charge. It is the case of the "Basic Service". Which includes live flight tracking, a limited amount of flight and aircraft details. These details are enough for the scope of this project, since they include position, altitude, track and speed of aircraft. The flight history available to be played back is of 7 days.

Their license does not allow to use commercially the information gathered by the use of their services, unless a Business plan is acquired [55] [56].

4.1.2 Opensky Network

OpenSky Network is a non-profit organization that provides a platform for collecting and sharing real-time global air traffic data. It was established in 2012 with the goal of improving the safety and efficiency of ATM by making air traffic data more accessible to researchers, aviation enthusiasts, and other stakeholders.

The OpenSky Network collects data from a variety of sources, including ground-based ADS-B receivers, portable receivers, and satellite-based ADS-B receivers. These data sources allow the network to track aircraft in real-time and provide accurate and up-to-date information on the location, speed, altitude, and heading of each aircraft.

The OpenSky Network uses an open data model, which means that the data collected is made available to the public for free. This allows researchers, developers, and other stakeholders to access and analyze the data in order to improve ATM systems, develop new technologies, and gain insights into the global aviation industry.

One of the key features of the OpenSky Network is its ability to collect and share data from areas that are traditionally under-served by traditional radar systems. For example, the network has been used to track some flights over the ocean, in remote regions, and in areas where radar coverage is limited.

In addition to providing real-time air traffic data, the OpenSky Network also provides historical data that can be used for analysis and research. This data can be used to study trends in air traffic, identify patterns and anomalies, and develop new algorithms and models for air traffic management.

However, most of their ADS-B network is ground-based and they are still developing their satellite-based ADS-B infrastructure. Hence, their API does not offer lots of data regarding inter-oceanic routes, which are the main focus of this project [57].

4.1.3 Flight Aware

FlightAware is a web-based aviation tracking service that provides real-time flight information to aviation enthusiasts, airlines, and airports. The company collects data from multiple sources to provide accurate and timely information to its users.

FlightAware is a global aviation tracking service that provides live flight tracking, airport information, weather maps, and other aviation-related data. It was founded in 2005 and is based in Houston, Texas. The company operates a web-based platform that allows users to access flight information from anywhere in the world.

FlightAware uses a combination of data sources to track flights in real-time. The primary data source is ADS-B. However, it also receives data from other sources, such as FAA radar sites, European radar sites, and satellite-based tracking systems.

Once the data is received, FlightAware processes it using proprietary algorithms to provide accurate and timely flight information. Users can access this information through the FlightAware website, mobile app, or by subscribing to various alerts and notifications. The historical data may also be accesed by the use of their API. This API, however, does not offer the data required in the scope of this project unless subscribed to a business plan [58].

4.2 Flight data base selection

Out of the previously mentioned flight databases, the chosen API for this project has been the Python FlightRadarAPI [59], which is further explained below. This decision has been done due to the following reasons:

	Flight Radar	**OpenSky Network**	**Flight Aware**
Open-source data	Yes	Yes	No
Inter-continental flights	Yes	No	Yes
Limited flights	No	No	No
Limited flight's age data	7 days	No	No

Table 4.1: Flight data bases features comparison

As it may be seen, only the Flight Radar satisfies the requirements of being open-source and having available inter-continental flights. In case that more flights were desired, the user could purchase one of the available services and gain access to more flights. For the purpose of this project, such thing is not required.

4.3 Data Gathering

4.3.1 API

An API is a set of protocols, routines, and tools for building software applications. It enables different software applications to communicate with each other and exchange data consistently. Overall, an API specifies how software components should interact, providing a simplified interface for developers to use complex functions.

APIs can be used in a variety of industries, including aviation, where they are used to retrieve flight data such as flight routes, arrival and departure times, aircraft type, and flight status. This information is critical to airlines, airports, and aviation enthusiasts, and is made available through various flight data providers like FlightRadar24, OpenSky Network, and FlightAware.

FlightRadar24 offers a web-based API that allows developers to access its flight tracking data, including real-time flight positions, flight routes, and aircraft data. It also offers a range of services, such as weather data and flight schedules. Similarly, OpenSky Network offers an API for accessing its data, including real-time flight positions and aircraft data. FlightAware also offers an API for accessing its data, which includes real-time flight positions, flight history, and airport information.

Developers can use these APIs to integrate flight data into their applications, such as airline reservation systems, airport operations software, and flight tracking applications. The APIs provide a standardized interface for accessing flight data, making it easier for developers to build applications that utilize this information [60] [61].

Flightradar24 API

The Flight Radar 24 API is an interface that allows developers to retrieve real-time flight tracking data from FlightRadar24. This data includes flight positions, flight plans, flight routes, and other relevant flight information.

To use the Flight Radar 24 API, users need to obtain an API key from FlightRadar24. Once they have the key, they can send HTTP requests to the API endpoint to retrieve the desired data. The data is returned in a structured format such as JSON or XML. Developers can then parse the data and use it to build applications or services that rely on real-time flight data. In case that no key is specified, the API considers that the user is subscribed to the Basic Services plan, and will limit the amount of data displayed in accordance with this plan. As previously mentioned, this plan offers the information required for the development of this project.

The Flight Radar 24 API provides several endpoints, each with a specific set of parameters to retrieve different types of data. For example, the ”flight data” endpoint returns data for a specific flight, while the ”radar” endpoint returns data for all flights within a specified geographic region.

There are several third-party libraries available that simplify the use of the Flight Radar 24 API. For example, the FlightRadarAPI Python library provides a Python interface to the API, allowing developers to easily integrate Flight Radar 24 data into their Python applications. However, this API doesn't have an extensive documentation to rely on which slows the users ability to work with it when they first use it [59] [62].

4.3.2 REST

REST is an architectural style for building web services. It is based on HTTP and uses its methods to expose data resources to clients. RESTful web services allow clients to access and manipulate resources using standard HTTP methods, such as GET, POST, PUT, and DELETE.

In REST architecture each request from a client to the server contains all the necessary information for the server to understand the request, and the server doesn't need to store any information about the previous requests.

The communication between the client and the server is done through messages that use the HTTP methods and the HTTP status codes to communicate the success or failure of the request. The messages are usually formatted in a standard way, such as JSON or XML, which make them easy to understand and process.

RESTful web services have become a popular way to build APIs that allow applications to communicate with each other over the internet. RESTful APIs are flexible, scalable, and easy to use, which makes them a good choice for many different types of applications [63].

4.3.3 JSON

JSON is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is often used for exchanging data between a server and a web application.

JSON is based on a key-value pair format. The data is represented as a collection of key-value pairs, where the key is a string and the value can be a string, number, boolean, null, array, or another JSON object.

For example, here is a JSON object of a flight identification: "id": "2f9f44f9", "row": 5341378796, "number": "default": "FX1624", "alternative": null In this example, the object has three key-value pairs. The id key has a string value "2f9f44f9", the row key has a number value 5341378796, the number key has in turn two other keys and values. The default key has a string value "FX1624" and finally the alternative key has a null value.

JSON is often used in web APIs to exchange data between a server and a client application. RESTful APIs, for example, typically use JSON as the data format.

JSON is language-independent, which means it can be used with any programming language. It is also widely supported by modern web browsers and server-side languages [64]. Several files have been saved or accessed by the software using this format as it shall be explained below.

4.3.4 Python

Python is a high-level programming language that is widely used for developing a variety of applications. It is known for its simplicity, readability, and ease of use. Python code is executed using an interpreter, which means that it can be run on any platform that has a Python interpreter installed.

NumPy is a Python library that provides support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. It is used extensively in scientific computing and data analysis. NumPy's functionality is built on top of the C and Fortran libraries, making it fast and efficient.

Matplotlib is a Python library that provides a wide range of 2D and 3D plotting options, allowing you to create high-quality visualizations of your data. It can be used to create line graphs, scatter plots, bar charts, and many other types of plots.

Pandas is a Python library that provides data analysis tools and data structures. It provides a flexible and powerful data manipulation and analysis environment, including tools for reading and writing data from and to a wide variety of data sources, including CSV, Excel, SQL databases, and more.

Astropy is a Python library that provides tools for astronomical data analysis. It includes a wide range of functions for dealing with astronomical data, including units and constants, coordinate systems and transformations, image processing, and more.

The previously mentioned libraries are some of the main libraries used in the development of this project [65] [66] [67] [68] [69].

4.4 Developed software architecture

A folder named tools has been added to the SILLEO-SeamSAT workspace. In it the user may find the `aircraft.py` code. The implemented code using the previously chosen API enables users to use two different features depending on the input used on the command line:

- routes: When users input something similar to: `aircraft.py routes US ES 2023 2 25` at the command line, the routes command is executed. The py aircraft.py routes input, calls the aircraft.py routes mode. Then the user may input the alpha-2 code [70] of the desired origin and destination countries of their desired route. In this example, flights originated from the US and with Spain as final destination have been considered. Finally, the user may establish the age of flights to look up. All flights between the established date (Y, M, D) until current time may be considered.

  When this mode is called, the user receives a list of flights ID for the specified route. Meaning that this list only contains flights with the previously set origin and destination countries and within the established time window. Note that with the Basic Service, users only have 7 days of historical flight data available.

- data: When users input something similar to: `aircraft.py data 2f9307af ./UsUs01.json 2023 2 25` at the command line, the data command is executed. Similar to the routes mode, when py aircraft.py data is used, the data mode is set. The next input is the flight ID of one of the flight ID's which have been listed when the routes mode has been called. Then, the name of the route must be chosen. The current notation is: origin country, origin destination and route's number ID. Finally, the same time window used for the routes mode must be set.

  Note that if the third input of the command line is neither routes or data, a message shall be displayed on the workspace indicating the way to properly use the software. The same applies if the format of the rest of the command line's input is not coherent with the previously mentioned examples.

See below an example on how the user interacts with the aircraft.py software:

```
py aircraft.py routes US ES 2023 5 27
Connecting to FlightRadar and getting flights ...
Processing flights ...
...............................................................
Found flights:
FlightId: 308e6246 --- From New York Newark Liberty International Airport (US) To Madrid Barajas
   Airport (ES)
```

As it may be appreciated, once the user chooses the routes mode and sets the origin and destination countries, as well as the age of the flights to search, the software displays a message indicating that it is connecting to the FlightRadar24 API and getting flights. Once the flights begin to be processed, they are divided into two categories: those which match with the conditions set by the user and those which don't. The later are displayed with a dot in order to indicate that the software is running. Once the software has run all the flights from the selected pool (which can be modified in the `aircraft.py` code), it displays a message indicating what flights have been found. The available information is the Flight ID and the origin and destination airports and countries. In case that no matching routes were found, the message displayed would be similar to the following one:

```
py aircraft.py routes US JP 2023 5 28
Connecting to FlightRadar and getting flights ...
Processing flights ...
.................................................................................
Found flights:


No matching routes were found, however you may see in the alternatives.json file some other found
   routes
```

As it may be seen, in case that no flights match the previously set conditions, the user may access the `alternatives.json` file in order to choose an alternative route. The available information of each alternative is the origin and destination countries and the flight ID. See below an example of the data saved in the `alternatives.json` file:

```
[
  {
    "origin": "US",
    "destination": "US",
    "flight␣ID": "308d3846"
  },
  {
    "origin": "JP",
    "destination": "US",
    "flight␣ID": "308d3a4d"
  },
  {
    "origin": "DE",
    "destination": "JP",
    "flight␣ID": "308d9d59"
  },
  {
    "origin": "GB",
    "destination": "SG",
    "flight␣ID": "308dace5"
  },
  {
    "origin": "GB",
    "destination": "HK",
    "flight␣ID": "308db29c"
  },
  {
    "origin": "US",
    "destination": "TR",
    "flight␣ID": "308db848"
  }
]
```

Finally, the user may run the data mode once the flight ID of the desired route or an alternative route has been chosen. See below the message that should be displayed when a route is saved, as well as the architecture of the saved route file:

```
py aircraft.py data 308e6246 ./UsEs01.json 2023 5 27
Connecting to FlightRadar and getting flights ...
Data saved for flight: 308e6246
```

```
{
  "identification": {
    "id": "308e6246",
    "row": 5365814586,
    "number": {
      "default": "UA51",
      "alternative": null
    },
    "callsign": "UAL51"
  },
  "trail": [
    {
      "lat": 40.477638,
      "lng": 356.428619,
      "alt": 0.0,
      "spd": 11.317768000000001,
      "ts": 1685787014,
      "hd": 143,
      "utc": "2023-06-03␣10:10:14",
      "fltime": 28088
    },
    {
      "lat": 40.478233,
      "lng": 356.428009,
      "alt": 0.0,
      "spd": 11.317768000000001,
      "ts": 1685787007,
      "hd": 143,
      "utc": "2023-06-03␣10:10:07",
      "fltime": 28081
    },
    .
    .
    .
    {
      "lat": 40.694023,
      "lng": 285.825539,
      "alt": 0.0,
      "spd": 0.0,
      "ts": 1685759558,
      "hd": 14,
      "utc": "2023-06-03␣02:32:38",
      "fltime": 632
    },
    {
      "lat": 40.694275,
      "lng": 285.825775,
      "alt": 0.0,
      "spd": 0.0,
      "ts": 1685759494,
      "hd": 33,
      "utc": "2023-06-03␣02:31:34",
      "fltime": 568
    },
    {
      "lat": 40.69429,
      "lng": 285.825806,
      "alt": 0.0,
      "spd": 0.0,
      "ts": 1685758926,
      "hd": 33,
      "utc": "2023-06-03␣02:22:06",
      "fltime": 0
    }
  ]
}
```

In the previous example it is displayed the information saved in the JSON file. There are two keys available: identification and trail. It may be appreciated that one of the values of the key identification is the flight ID (which corresponds to that of the chosen flight). The majority of the file consists of a list of the trail's data. The available data for each item is: latitude, longitude, altitude (in meters), speed (in m/s), ts (which corresponds to the Unix time), heading (in °), utc (which states the Unix time in a value readable by humans) and finally, the flight time (which represents the time since the start of the flight).

4.4.1 aircraft.py architecture

In the following paragraph, a pseudo-code analysis of the `aircraft.py` software architecture is developed.

1. First, the software imports all the required Python libraries which will be later used for pre-processing the desired data and sets the list of IBAN Alpha2 codes of 249 different countries.
2. Then, the initial configuration is set. The time window is obtained from the command line as previously mentioned. Since the information in the command line is a set of strings (Y, M and D), this data is converted into Unix time. The current time (time when the script is executed) is also converted to Unix time. Then, the window time is obtained from the difference between these two times. This will be a certain number which shall represent the time window in seconds. These means that any flight between current time and the specified time will be downloaded from the API.
3. Additionally, other fields such as not taking gliders and ground vehicles into account are set. Since complete flights are desired, flights that are on "air" are disabled. Hence, all flights downloaded will have either landed or will be waiting to take off. Currently, the script has limited the amount of downloaded flights to 100 in order to speed up the time it takes the script to run. However, this number may be changed manually within the script to have a greater pool of flights available. Note that there are about 150000 tracked flights by Flight Radar 24 each day all around the globe and that as the limit of flights downloaded increases, so does the time it takes the script to run.

4. Once the configuration has been set, the code will enter either the routes or data modes according to the command line's input.

5. If the routes mode is chosen, the script shall iterate through every flight that has been previously downloaded and get its details (the amount of information available without the details is not enough to filter the origin and destination countries). Then, it will filter all flights that don't match the origin and destination countries desired and also those whose status is not "landed". In case a flight fulfills every requirement, its flight ID shall be displayed on the terminal following the previously mentioned format. Otherwise, a dot shall appear on the script. Providing that no flights match the conditions, an `alternatives.json` file containing all flights that didn't match the conditions is created.

   Not all flights have all the necessary data available, hence in case that the software acknowledges that some key information is missing from a flight, such flight shall be skipped.

6. On the other hand, if the data mode is chosen, all details from the chosen flight are downloaded. Then, all the information available on the values that correspond to the trail key (altitude, speed, ...) are converted to SI. It is important to note that in case that a negative longitude is found, it shall be transformed into a positive value. Below may be seen an explanation for the reason to do this. Finally, all information that is not required is erased (such as information about the status of the flight, its flight-History which indicates the last flights that the plane has taken or the type of aircraft, among others). The only data that will be saved in the .json file are the flight identification and its trail.

   In case that there was some kind of problem saving the data, a `problems.json` file containing all the data available from the chosen flight would be created. Since some filters have been applied to the code in order to avoid coding errors, this file should not be created. However, in case that the `aircraft.py` code is modified in a future in a way that causes some error, this file should help in finding the cause of those errors.

7. Finally, the code defines the usage of the software. In case that none of the available modes are chosen, or that the format of the input on the command line is not correct, a message stating the proper use of the software shall be displayed on the terminal.

Positive longitudes

As mentioned previously, the code transforms any negative longitude found into a positive one (summing 360° to it). There is no formal problem by stating the longitude of an airplane in negative degrees. However, due to the architecture of the aircraft's pre-process in the Silleo-SeamSat's software, a bug would occur when passing from positive degrees to negative ones.

Formally it is the same to say that an airplane is at a longitude of 181° than at -179°. However, as it will be later explained in the Silleo-SeamSat's software architecture, the software uses interpolation in order to know the position of a flight at all times (this will be later explained when analyzing the Silleo-SeamSat's software architecture).

In the following example it shall be explained what would happen in case that the longitudes were not modified.

Let's suppose that an aircraft is at a longitude 180° at t=0 and by the time that t=10, the longitude is -179°. Then, at t=5 and due to the way that the software works, the longitude would be:

$$lng = \frac{t - t_1}{t_1 - t_0}(lng_1 - lng_0) + lng_1 = \frac{5 - 10}{10 - 0}(-179 - 180) + (-179) = 0.5$$

As it can be seen, the result obtained is 0.5º. Meanwhile, the correct result should be 180.5° or -179.5°. If, instead, the longitudes are previously set to positive, then the result would be:

$$lng = \frac{t - t_1}{t_1 - t_0}(lng_1 - lng_0) + lng_1 = \frac{5 - 10}{10 - 0}(181 - 180) + 181 = 180.5$$

In case that the bug wasn't solved, the aircraft would fly into a certain direction until this point where longitudes turn from positive to negative (180°) is reached. Then it would change its direction, do a whole turn around the Earth and then continue in the correct direction.

Chapter 5

SILLEO-SeamSat

SILLEO-SeamSat is a constellation simulator that allows the simulation of different constellation topologies for satellite communications. Silleo-SeamSAT is an advanced evolution of the base code SILLEO-SCNS developed be Benjamin Kempton [5] at Christopher Newport University, which has been later improved by the Tuareg research group at the Universitat Politècnica de Catalunya.

SILLEO-SeamSat is a satellite-based network simulator that allows the modeling and analysis of communication networks using different types of satellites. It is designed to simulate satellite networks for a range of different applications, including space science, earth observation, and telecommunications.

SILLEO-SeamSat works by calculating the establishment of communication links between satellites and between satellites and GS. It can be used to model a variety of different types of satellite constellations, including GEO, LEO, and polar orbiting satellites.

SILLEO-SeamSat also includes a range of different analysis tools, such as packet error rate analysis and link budget analysis, which can be used to evaluate the performance of different types of satellite networks. It also allows users to simulate different types of satellite missions and to evaluate the impact of different mission parameters on network performance [71][5].

There are a lot of features that distinct Silleo-SeamSAT from Silleo-SCNS: there has been added a SGP4 (Simplified General Perturbations-4 software) model to propagate satellite motion; the format of the GML generated with the export function has been extended and a new GML import function to retrieve the simulation state from a GML file has been developed; there has been added a latitude restriction to limit the operability of inter-satellite links between orbital planes when they are outside the operational zone, among other features.

5.1 SILLEO-SeamSat software overview

The software is based on a client-server architecture, where the server component is responsible for managing the network topology, the satellite configuration, and the propagation models, while the client component provides a user-friendly interface for configuring the simulation parameters and displaying the simulation results.

The simulation process begins with the user specifying the network topology, which consists of a set of GS and satellites that are interconnected by links. Each link has a specific bandwidth and delay, which are specified by the user.

Once the topology is defined, the user can configure the satellites by specifying their orbits and other parameters, such as the transmit power and antenna gain. The propagation models used in the simulation take into account the effects of atmospheric attenuation, rain attenuation, and other factors that affect the signal strength.

After the simulation parameters are configured, the simulation can be started, and the software will generate a time series of the network traffic, showing the data rates and delay experienced by each link.

One of the main advantages of SILLEO-SeamSAT is its flexibility, as it allows the simulation of a wide range of network typologies and configurations. It can be used to evaluate the performance of existing networks or to design new networks for specific applications.

The main modules of the simulator are:

- GUI: The GUI module in SILLEO-SeamSAT is responsible for providing an interface which the user may use to interact with the simulation. It allows the user to configure simulation parameters, start and stop the simulation, and view some data. The GUI is built using the Qt framework, which provides several widgets that can be used to create windows, buttons, menus, and other elements of the user interface.

- Constellation: The constellation module in SILLEO-SeamSAT is responsible for defining the satellite constellation that will be used in the simulation and computing the position of the constellation and Earth's rotation. It specifies the number of satellites in the constellation, their orbits, and their communication capabilities. The constellation can be defined using a TLE file, which is loaded into the simulation when it starts. The GS are loaded in this module too.

- Simulation: The simulation module in SILLEO-SeamSAT is responsible for running the simulation. It uses the constellation module and other input parameters to simulate the satellite network over time. The simulation is based on a discrete event model, where events (such as satellite transmissions or message arrivals) are processed sequentially for each timestamp.

During the simulation, the simulation module keeps track of the status of each satellite in the constellation, including its position and velocity, as well as the GS positions. It then determines the feasibility, state and propagation delay of the communication links. It also displays several statistics and performance data, such as the message delivery rate and the latency of message transmissions.

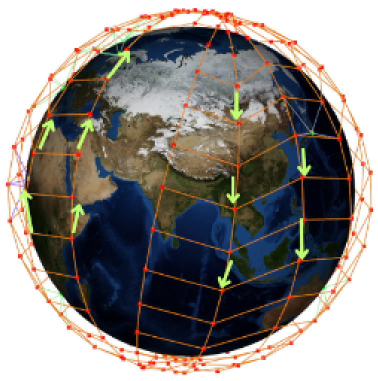
Overall, the GUI, constellation, and simulation modules work together to provide a flexible and powerful simulation tool for evaluating the performance of satellite networks [71].

5.1.1 Walker Star and Walker Delta configurations

The Walker Star and Walker Delta configurations are two commonly used designs for satellite constellations. Both configurations involve the use of multiple satellites in a specific pattern to provide global coverage.

The Walker Star configuration involves placing satellites in a circular orbit at a fixed altitude, equally spaced around the orbit. The number of satellites required for this configuration is determined by the desired coverage and the number of planes in the orbit. For example, a 24-satellite configuration would require six planes with four satellites equally spaced in each plane. The satellites travel in the same direction and at the same speed, providing continuous coverage of the Earth's surface. This configuration is known for its simplicity and uniform coverage, but it requires a large number of satellites. As shown in figure 5.1a, it also presents an orbital seam where neighbouring satellites are orbiting in opposite directions and are unable to establish a communication link.

The Walker Delta configuration involves placing satellites in three separate orbits with different inclinations. The satellites in each orbit are equally spaced, but the spacing between the orbits is not uniform. This configuration requires fewer satellites than the Walker Star configuration to achieve the same coverage. The disadvantage of this configuration is that it is more complex, as the satellites are traveling in different directions and at different speeds.


(a) Walker Star

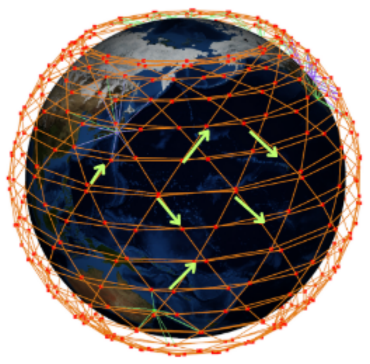

(b) Walker Delta

Figure 5.1: Constellation configurations

The choice between the two configurations depends on the specific requirements of the project. The Walker Star configuration is better suited for missions that require continuous, uniform coverage with a large number of satellites. The Walker Delta configuration is better suited for missions that require fewer satellites and can tolerate periodic gaps in coverage [72].

5.1.2 SILLEO-SeamSat software architecture

In the following section, a more precise analysis of the changes implemented to the SILLEO-SeamSat's three main codes shall be conducted. The developed class may be found at the annex (note that the rest of the software has been modified in order to fit in the developed class features), where it may also be found a flowchart which states all the main classes, methods and functions that take part into the software's architecture. Moreover, the way that they are related with one another, has been graphically represented.

No major explanation of all methods that form Silleo-SeamSat's software shall be conducted, since this information may be found at [71] [5]. Only the performance of the modified or developed methods shall be further explained.

In figure 5.2, an overview of the software simulator framework structure may be found.

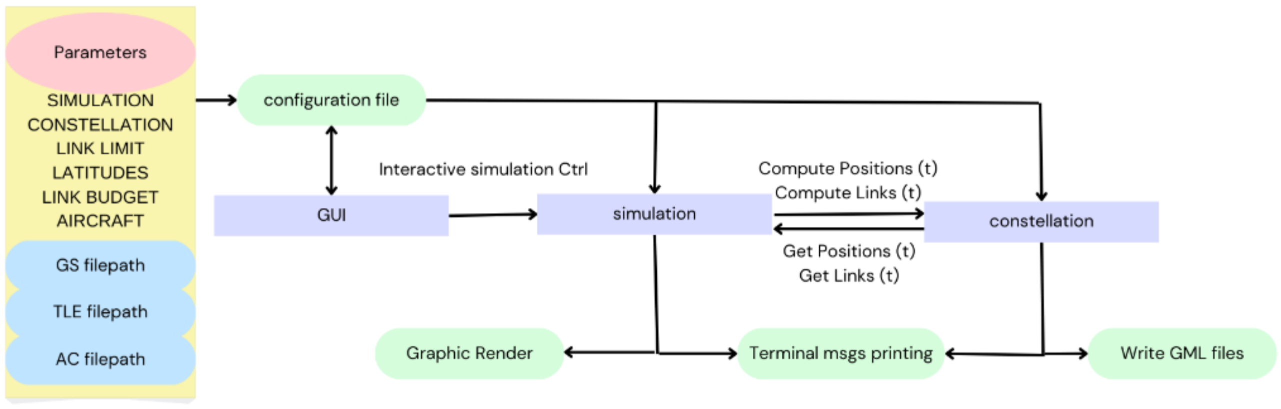


Figure 5.2: SILLEO-SeamSAT constellation simulator framework structure

Note that there is also a `nogui` module which allows batch simulation of the constellation and saving of the GMLs without a graphical render.

GUI and simulation overview

As mentioned previously, the GUI node is responsible for providing a user interface.

Each time that the GUI is called, the $sim\_confg\_INI.cfg$ file is loaded. From this file the required information to fill the GUI with the preferences stated in such file is extracted. In case that the GUI is manually changed and the simulation is executed pressing the "Generate" button, the current preferences of the GUI shall rewrite the configuration file. This file will also be used in the other SILLEO-SeamSat's modules: `simulation.py` and `constellation.py`. A toolbox to change the ac file, if desired, has been implemented.

Once the "Generate" button is pressed, a visualization window is opened (see figure 5.3) and the lower half of the GUI is enabled. Later on (when explaining the changes implemented to the `simulation.py` module) it shall be explained how the simulation that generates the visualization toolkit for the AC has been coded. The user may notice that the red squares (nodes) represent the satellites, the green ones represent the GS and the yellow ones the AC. The lines (edges) that unite the different nodes, represent the ISL, GSL and ASL and they are orange, green and purple, respectively.

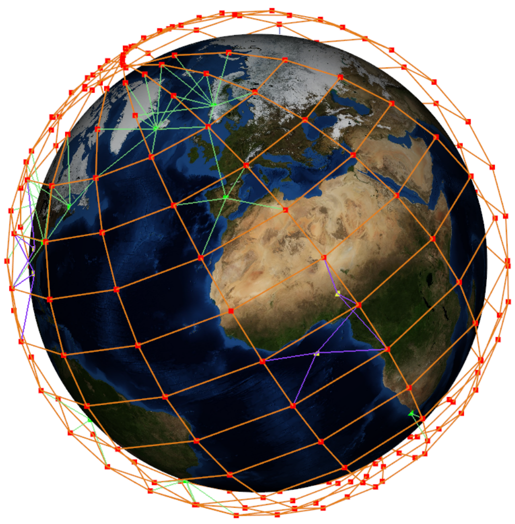


Figure 5.3: Visualization toolkit

Returning to the GUI, the user may appreciate at figure 5.4 that now the simulation may be started by pressing the run/stop button and that the time-step may be manually changed. The GMLs of each time-stamp of the simulation may be saved as well. In order to understand what a GML file is, one must know that Silleo-SeamSAT is able to write GML files which define the state of the simulation in a specific time. These GML files include all the information about the links among the communication nodes. Other information not specifically related with the communication links can be included too, in order to redo the current state with only the GML information. Additionally, note that Silleo-SeamSat uses the concept of modules to add new functionalities as previously explained. Each module defines the attributes that it uses for its own configuration setup. A GML file has different sections where attributes are present [73]:

- graph: General file attributes
- nodes: Attributes of each node
- edges: Attributes of each edge (link)

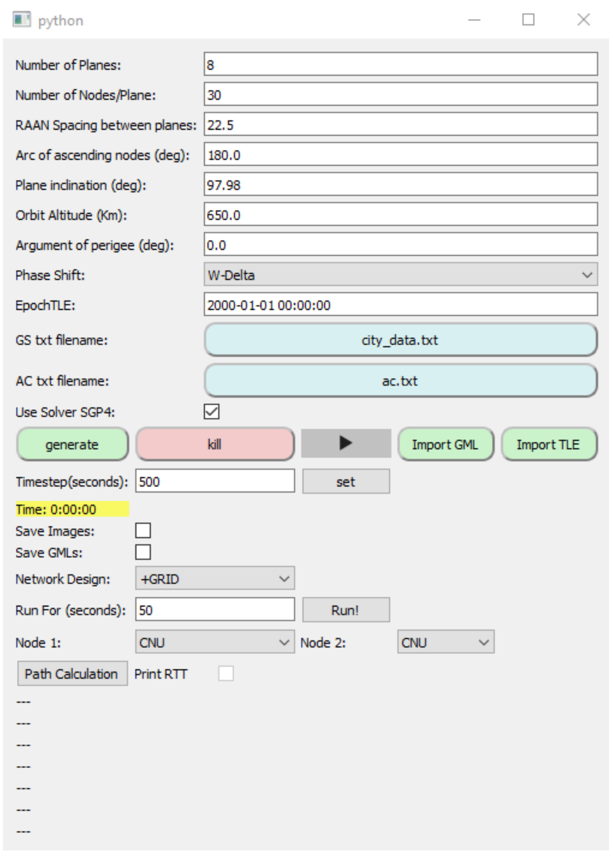
The following data has been added to the GMLs:

- AC nodes: The AC's ids, labels, names and node-types (AC) have been saved; as well as their latitudes and longitudes (in both degrees and radians), their altitudes and number of antennas. See in detail below
- ASL edges: The links' source and target nodes have been saved, as well as the link-type, the distance between the AC and the linked satellite and some data which indicates the state of the link. See in detail below.

```
node [
    id 253
    label "-14"
    nodeType "AC"
    acName "AC0"
    ac_lat_deg 27.97905888571428
    ac_lon_deg -82.53753728571428
    ac_lat_rad 0.4883266991650901
    ac_lon_rad -1.4405517821232978
    ac_alt_m 0.0
    maxAntennas 2
  ]
```

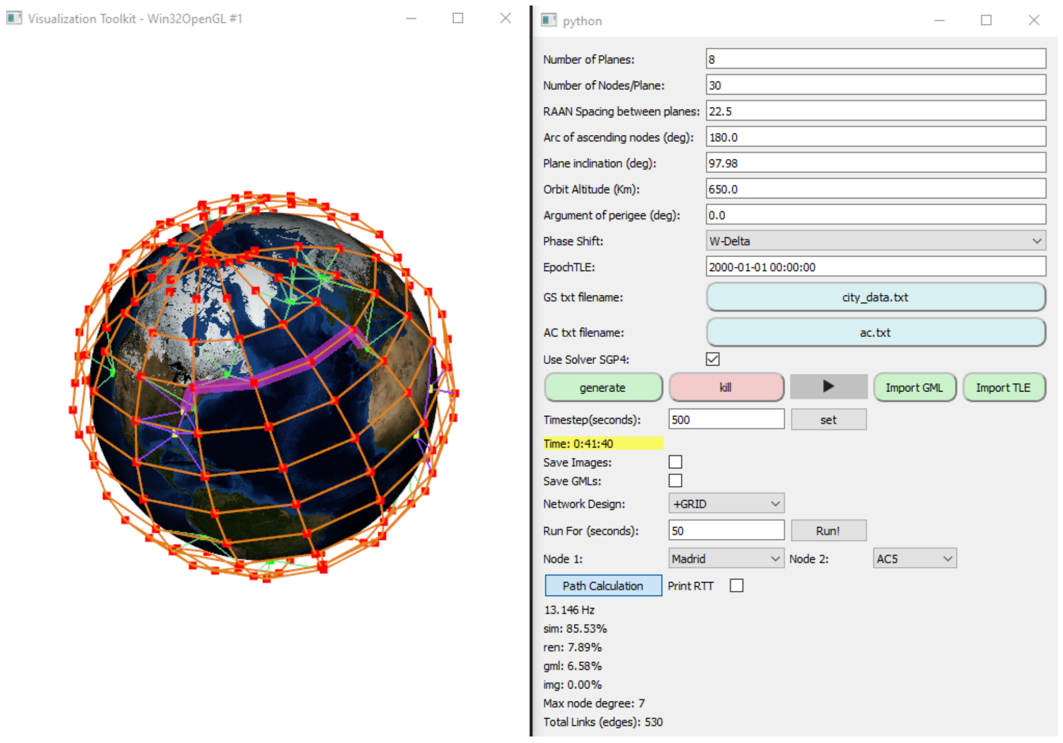
```
edge [
    source 31
    target 253
    linkType "ASL"
    distance 1732534
    delay 5.779111361100352
    weight 5.779111361100352
    priority -1
    bandwidth -1.0
    state 1
  ]
```

Finally, by pressing the *Path Calculation* button, the user may choose an origin and destination GS or AC and see the most optimal path to send a message from the origin to the destination. This is calculated by using path finding algorithms and analyzing the most optimal route for the communication's graph of each time-stamp. This function was already implemented in the SILLEO, however, the AC have been added on the display menu. See in detail in figure 5.5.



Number of Planes: 8
Number of Nodes/Plane: 30
RAAN Spacing between planes: 22.5
Arc of ascending nodes (deg): 180.0
Plane inclination (deg): 97.98
Orbit Altitude (Km): 650.0
Argument of perigee (deg): 0.0
Phase Shift: W-Delta
EpochTLE: 2000-01-01 00:00:00
GS txt filename: city_data.txt
AC txt filename: ac.txt
Use Solver SGP4: ☑
Timestep(seconds): 500
Time: 0:00:00
Save Images: ☐
Save GMLs: ☐
Network Design: +GRID
Run For (seconds): 50
Node 1: CNU Node 2: CNU
Print RTT ☐

Figure 5.4: GUI once the Generate button has been pressed



Number of Planes: 8
Number of Nodes/Plane: 30
RAAN Spacing between planes: 22.5
Arc of ascending nodes (deg): 180.0
Plane inclination (deg): 97.98
Orbit Altitude (Km): 650.0
Argument of perigee (deg): 0.0
Phase Shift: W-Delta
EpochTLE: 2000-01-01 00:00:00
GS txt filename: city_data.txt
AC txt filename: ac.txt
Use Solver SGP4: ☑
Timestep(seconds): 500
Time: 0:41:40
Save Images: ☐
Save GMLs: ☐
Network Design: +GRID
Run For (seconds): 50
Node 1: Madrid Node 2: AC5
Print RTT ☐
13.146 Hz
sim: 85.53%
ren: 7.89%
gml: 6.58%
img: 0.00%
Max node degree: 7
Total Links (edges): 530

Figure 5.5: Path Calculation Visualization

Configuration files

The SILLEO-SeamSAT uses configuration files (.cfg) in order to save the current state of the GUI interface as well as the simulation's preferences. The main simulation file is the *sim_confg_INI.cfg* file. This file has four sections with various data that will be used later in the different main nodes:

- SIMULATION
- CONSTELLATION
- LINKLIMITLAT
- LINKBUDGET
- AIRCRAFT: This new section is used to set the initial values for some items that involve the simulation. The location of the AC files, that contain all aircraft desired to be simulated, have been implemented in this section. See the example below for more details.

```
[AIRCRAFT]
ac_module = 1
ac_file = C:/Users/.../silleo4/source/ac_txt/ac.txt
```

Currently, there exists a folder that handles the AC files. Such folder is the *ac_txt* file, which stores the txt files that the GUI may call, as well as all the JSON files with the previously saved flight routes (using the `aircraft.py` software).

The txt file that the GUI may call (`ac.txt`) has the following structure:

```
route_filename aircraft_name timestamp max_antennas
UsUs01.json AC0 0 2
UsUs01.json AC1 4000 2
# testFlight aircraft_name initial_lat initial_lon alt_m init_hd spd_m/s max_antennas
testFlight AC2 10.00000 10.00000 10000 80.0 160.0 2
testFlight AC3 0.0 0.0 10000 0.0 160 2
JpUs01.json AC4 0 2
UsDe01.json AC5 0 2
```

As it may be seen in the previous example, there are two different types of aircraft.

First, there are the aircraft which have been created using a previously saved route. For this kind of aircraft, the first item is which route is desired (in the example both planes use the same route but any route could be called), then the name ID of the plane is stated and finally its timestamp and the number of antennas it has. The timestamp is used to indicate when the aircraft will begin the flight route, meaning that for a timestamp of 0, the plane will simply follow the route. However for greater timestamps, the aircraft will only start the route when the simulation time reaches such timestamp. The number of antennas has no special effect in this project but it may be used to further improve the SILLEO-SeamSat simulator in future projects.

Lastly, there are the test flights (the user may manually set the parameters for these flights by changing this file). These flights first item is `testFlight`. This item is used to differentiate these flights from the previous ones. After this parameter, all the parameters of the flight may be set. Specifically, the flight ID, the initial latitude and longitude, altitude, heading and speed may be set. The heading is set in degrees and indicates the angle between the magnetic north pole and the aircraft's direction.

Note that the test flights and the route ones do not require to be organized in any specific order. As a consequence, the user may input as many flights and in any order as they desire.

5.1.3 gui.py

As mentioned previously, the following changes have been implemented in the `gui.py` node:

- Added a toolbox which enables the user to change the ac file (or even remove it)
- Added all the AC in the drop-down menu for the path calculation, allowing to see the communication links (in pink) between GS-AC, AC-AC or GS-GS

In the following section it shall be performed a pseudo-code analysis of the developed software. Please note that no actual code shall be shown, in case that a more visual support is needed to understand how the methods are correlated, refer to figure 11.1 and to the Annex where the `gui.py` code may be found. This statement is true too for the later analysis of `simulation.py` and `constellation.py` codes.

First of all, the AC folder path has been saved in order to have an easier access to it later.

In the `makeInput` method, a button which loads the AC file when it is pressed has been added. This connects (as seen in figure 11.1 with the `LoadACfile` method. Which, in turn, accesses the AC folder path previously saved and displays all AC files available. Once the user chooses one of them, this file is saved onto the path.

In the $config\_read\_cfg\_ini$ method, the software reads the `AIRCRAFTS` section in the $sim\_confg\_INI.cfg$ file. In case that the *ac_module* is set to 0, AC are disabled. On the contrary, if it is set to 1, the user may choose an AC file.

The `gml2config` method, allows the user to choose one of the saved GMLs and use it as a configuration file, as previously explained. These GMLs don't have any AC by default. As a result, in case that the user desires to analyze these GMLs with AC, the user shall have to implement them by choosing an AC file as previously explained and generating a new simulation.

Finally, once the `makeConstellation` method is called, the software saves the current state of the GUI into the configuration file. Due to the way that the AC node module has been implemented, even if the AC are added into the simulation or not (depending on the value of *ac_module*, this information shall be saved into the configuration file.

No more changes have been implemented in this file, the functionality that allows the user to visually see the the path calculation between the chosen nodes shall be now explained in the analysis of `simulation.py` software, because this is the one that realizes the path finding algorithm resolution.

5.1.4 simulation.py

The `simulation.py` code is the one responsible for running the simulation. It calls the constellation node using the `self.model` variable, which enables the software to call methods from the `Constellation` class. The Silleo-SeamSAT simulation process may be explained using the following flowchart.

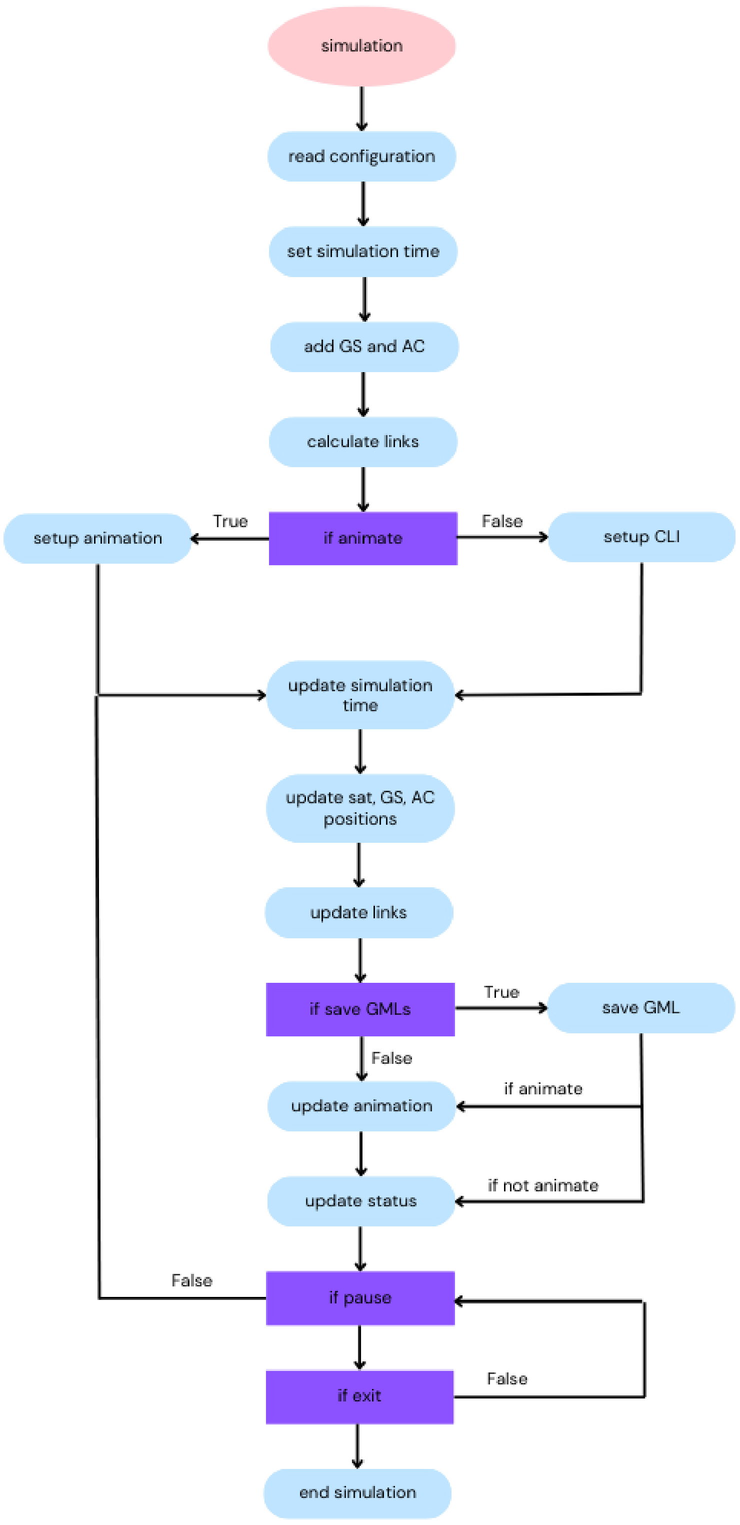


Figure 5.6: Silleo-SeamSat simulation overview

The main changes implemented in this code are the following ones:

- Visual representation of AC and the ASL links
- Use the selected item in the GUI's drop-down menu to implement path calculation, allowing to see communication links

In order to implement these functions the following changes have been developed:

First of all, once the `simulation.py` code is executed and the *__init__* method is executed, the configuration file is read and the chosen AC file is saved. Later on, the communication pipe is initiated and the constellation node is imported. Then, after adding the ground points to the constellation model, the AC may be added too by calling the `setACData` method. However, due to the way that the `ACflights` class has been developed and implemented into the `Constellation` class, it is required to previously call the methods `setGroundNodeCounter` and `setGhaAries` (which save the ground node counter and the GhaAries point, respectively, inside the `ACflights` class. These values are required to enable some methods of this class).

Then, the GS and AC (if any) are sent to the GUI so that the user may see them in the drop-down menu. Finally, the `initializeNetworkDesign` and `setupAnimation` methods are called.

On one hand, the `initializeNetworkDesign` method, sets the max ISL, GSL and ASL distances (see next section for more information). Additionally, it reads which linking method has been set in the GUI (+Grid or Sparse) and calls the `calculatePlusGridLinks` method with the appropriate input variables. Note that currently there also exists the IDEAL linking method. However, this method is not formally correct, it is currently outdated, out of use and it will be erased in a near future. It is for this reason that the AC node module has not been implemented in this method and, if chosen, the simulator won't work.

On the other hand, the `setupAnimation` method is the one which makes that creates the different actors that take place in the simulation and makes the render window using the `makeRenderWindow` method. The `makeACActor` method has been implemented in this method.

The `makeACActor` methods generates the point cloud to represent AC. It uses as parameters the total number of AC and an array of AC's positions. This method inserts this data in the simulation and creates an `acActor` with an opacity, color and point-size previously set (you may find these customisations at the beginning of the `simulation.py` code). The `makeLinkActors`, similarly to the `makeACActor`, is the method responsible for generating the lines which represent the links in the simulation. In this case, the ASL have been added. Just like for the `acActor`, the properties of the `aslActor` may be found at the beginning of the `simulation.py` code.

Let's first explain how does the software distinguish between ISL, GSL and ASL links. As previously explained, the Silleo-SeamSat software works with graphs where satellites, GS and AC are its nodes and the links are the edges. The satellites have been set with a positive node ID, while the GS and AC have been set with a negative node ID. The code has been defined in a way so that the first node ID of the first airplane is the next of the last GS. For example, if there are 10 GS, the last GS will have a node ID of -10 and the first

AC will have a node ID of -11. For a link to be created, two nodes must communicate. The links have been differentiated following the following hypothesis:

- If both nodes are positive, the link shall be ISL
- If one of the nodes is positive and the other is negative, being the negative one a GS (the node ID is bigger than the negative number of GS), the link shall be GSL
- Otherwise, the link shall be ASL

There is a method in the `constellation.py` software that does exactly this: the `getLinkType` method. Finally, once the software has determined whether a link is ISL, GSL or ASL; it customises it in a visual way that makes it easy for the user to differentiate them. Currently, ISL are red, GSL are green and ASL are purple.

To consider next is the `makeRenderWindow` method. This method creates a render window object using vtk, once that all the actors have been created. First of all, it adds all the actors (including AC) and then it calls the `updateAnimation` method. Finally, it starts the model.

The `updateAnimation` method, in turn, takes in new position data and updates the render window. First of all, it calls the `updateModel` method which sall be further explained below. Secondly, it grabs the new position data of satellites, GS and AC. In order to grab such data of the AC, the `getArrayOfACPositions` has been developed. This method shall be explained in the next section too. Then, the satellite, GS and AC points are updated. Finally, the data of links is grabbed too and all the updated data is uploaded into the Render Window.

Finally, let's explain what the `updateModel` method does. This method updates the model with a new time, recalculates the links and exports the GML files. In case that the user wishes to save the GMLs and has toggled on the save GMLs case on the GUI, then this method shall call the `generateNetworkGraph`, `getFileNumber` and `exportGMLFile` methods. The `getFileNumber` function creates the numeration for file naming, being the first GML file called *g_0000000*, the second *g_0000001* and so on. The `generateNetworkGraph` method shall be explained in the next section. Returning to the `updateModel` method, it then reads the simulation time and updates the links, the network design and the position of satellites, GS and AC by calling the `setConstellationTime` and `calculatePlusGridLinks` methods (see next section). In case that the *enable_path_calculation* option has been enabled in the GUI, then the $genereateNetworkGraph\_fast$ method is called and the shortest path between the chosen GS or AC is calculated using path finding algorithm methods. Finally, this route is displayed in pink on the Render Window.

5.1.5 constellation.py

Finally, an analysis on the `constellation.py` software shall be executed. This node module is the main node that calculates and characterizes every aspect of the orbital propagator.

Two main classes have been implemented in this code:

- `Constellation`: This class, which is the main one, controls the propagation of both satellites and GS and is the responsible for creating and updating all the links too.
- `ACflights`: This class, which has been called inside the previous one, is the responsible for creating and updating the positions of all AC.

5.1.6 ACflights class

First of all, let's explain how the `ACflights` sofware architecture works. At figure 5.7 all methods and their relationship are displayed. Once this class is called into the `Constellation` class, some global parameters required to enable several methods are set. The `setACData` method, mentioned previously, adds AC at initial position to the constellation model. This is done by calling the `aircraftData` method and saving the name of each AC, its latitude, longitude, altitude, speed, vertical speed, heading and number of antennas. For now, the software only uses the AC's name, latitude, longitude and altitude. However, in case the rest of the data is required to implement any new function in a future, it will be easy to access. Below there is a more detailed explanation on how this method works.

The `addACData` method is used to create an array of AC. The information available for each airplane is the following:

```
AIRCRAFT_DTYPE = np.dtype([
    ('ID', np.int16), # ID number, unique, = array index
    ('x', np.float64), # x position in meters
    ('y', np.float64), # y position in meters
    ('z', np.float64), # z position in meters
    ('alt_m', np.float32),  # GS altitude in meters
    ('maxAntenna', np.int16)]) # GS max Antennas (opque a.t.m.)
```

Note that the latitude, longitude and altitude have been converted to Cartesian coordinates. Finally, the `updateACPos` is used to update the array of AC with the positions of the new simulation time.

All codes from this class may be found at Annex B.

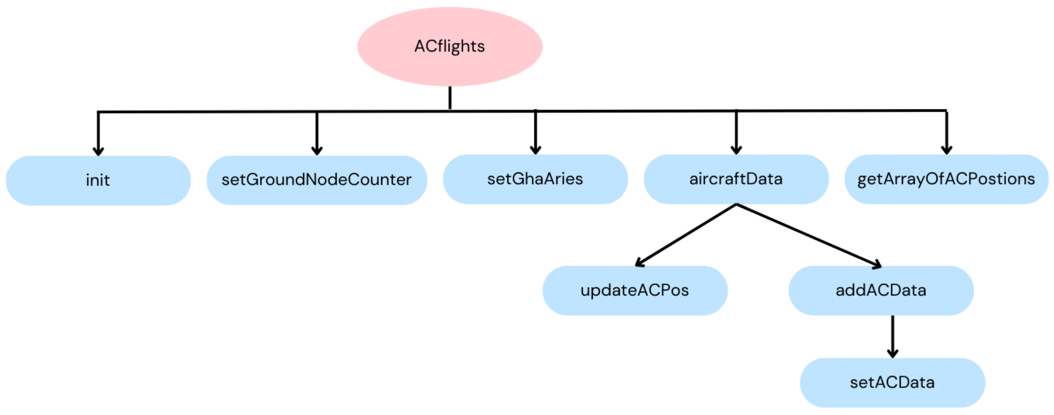


Figure 5.7: ACflights methods

5.1.7 Constellation class

Now let's explain the changes realized on the `Constellation` class to implement all AC. As mentioned previously, once the `Constellation` class is called by `simulation.py`, it is initiated. It is then, when the `ACflights` class is called. As a result, all methods from this class may be called from the `Constellation` class.

In the `getArrayOfNodePositions` method, the AC have been added, as mentioned previously, AC are considered a node in the Silleo-SeamSAT software. The `updateACPos` method, has been called into the `setConstellationTime` method, which in turn, updates the position of all nodes according to the current simulation time. In both `generateNetworkGraph` and $generateNetworkGraph\_fast$ methods, ASL links have been added to the graph. The saved data of the ASL links is the source and target nodes, the linkType, the distance of the link, its delay, weight, priority, band-with and state. For the latter method, only the source and target nodes and the distance have been saved. The method used to differentiate links is the `getLinkType` method previously explained. In the first method, the AC ID, the nodeType (AC), latitude and longitude in both degrees and radians, the altitude and the number of antennas have been saved in the graph too [73].

The `calculateMaxSpaceToACDistance` method is used to calculate the maximum communications distance between AC and satellites given a field of view for AC, which is defined by a minimum elevation angle above the horizon, and the altitude of AC. In order not to calculate this value for all AC each time that the simulation is updated, a list of distances for different flight levels is obtained the first time that the simulation is run. For now, the flight level step is of 500 m and communications distances from 0 to 16000 m have been calculated, hence ensuring that these distances may be calculated for all AC (note that most AC don't flight above 12000 m). These communications distances have been calculated as follows:

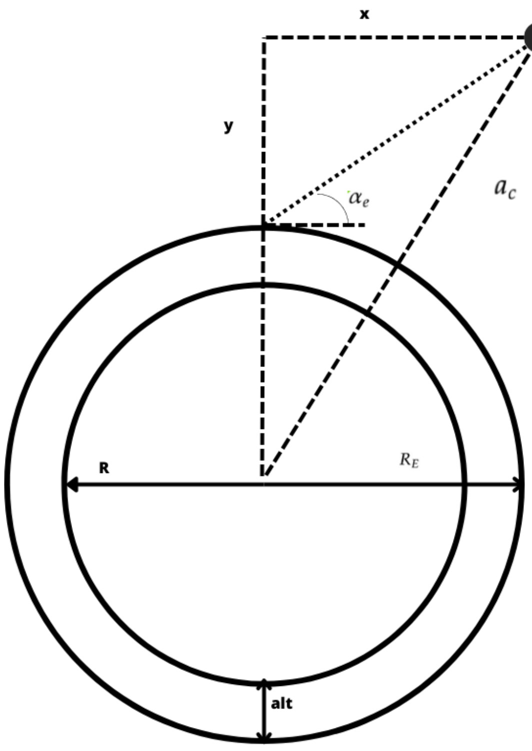


Figure 5.8: ASL communication distance overview

To find the ASL distance, it is required to find the intersection point between the AC field of view line path and the constellation orbit. This may be defined with the following system of equations. Note that R_E has been considered to be the sum of the Earth radius and the flight level of the AC:

$$y = R_E + tan\alpha_e \, x$$

$$\sqrt{x^2 + y^2} = R_E + a_c$$

This system may be rewritten as:

$$(1 + tan^2\alpha_e)x^2 + 2R_E tan\alpha_e \, x + R_E^2 - (R_E + a_c)^2 = 0$$

After obtaining x and y, the ASL distance may be obtained as:

$$ASL = \sqrt{x^2 + (y - R_E)^2}$$

Finally, the `calculatePlusGridLinks` method connects the satellites in a cross grid or sparse network, ASL and GSL links are added too. In order to implement the ASL links, both the AC and satellite arrays have been run. If the ASL distance is within the acceptable range for the flight level (as calculated using the method above), the link is created. The following information about the link array is saved:

```
LINK_DTYPE = np.dtype([
    ('node_1', np.int16), # an endpoint of the link
    ('node_2', np.int16), # the other endpoint of the link
    ('distance', np.int32), # distance of the link in meters
    ('SNR', np.float32), # SNR of the link
    ('status', np.int16)]) # status of the link 0-1(inactive/active)
```

ACflights.aircraftData

The `aircraftData` is the main method of this class. It reads the AC file, accesses all AC positions for the current simulation time and saves the data mentioned previously. In order to do that, the AC file is accessed and the types of flights are filtered differentiating between test flights and saved routes.

In case that a testFlight is detected, the software uses the Spherical Law of Cosines (see below) to calculate an AC new position taking into account it's initial position, speed, flight time and heading. The new heading of the next position, which will either be the same as the initial or +180º, depending upon the new position, is calculated using the Spherical Law of Sines.

On the contrary, if the software detects that a flight is not a testFlight, then it tries to read its route (note that all saved routes should be saved in the *ac_txt* folder). Then the software accesses the flight information which corresponds to the simulation time. In order to do that, the code runs through all the saved route's flight data information, until a matching flight time to the given simulation time is found. If such is the case, it returns the data belonging to that flight time. If the flight time at a given position isn't equal to that of the given simulation time (and it is smaller than it), the script checks the following position and repeats this process until either a matching time or a greater time are found. If the latter occurs, then the data is obtained by interpolating the data belonging to the current and previous flight times.

The following hypothesis have been made in order to obtain such data:

- The longitude and latitude have been obtained taking only into account the previous and posterior flight times (in accordance to the desired time), as well as their correspondent latitude and longitude. By simply interpolating the data, it is assumed that the speed is constant for each time-stamp. This assumption (which is not technically true since the speed varies during the flight) is assumed to be true for two main reasons: on one hand, the Δv for each time-step during cruise (which corresponds to the majority of the flight) (v_1 and v_2) is almost never more than 4 m/s, and it is at most of 15 m/s. Considering that the cruise speed is usually about 240 - 250 m/s, a difference of 4 m/s only represents an error of 1.6 %. On the other hand, the time-steps when flying above the ocean are usually about 10 - 20 minutes. This means that the error in position using this method would be at most:

$$\epsilon_r = \frac{\Delta v\, t}{R + altitude} = \frac{3\, m/s \times 20 \times 60\, s\, s}{6378135 + 10000\, m} = 0.056\%$$

Where Δv is the maximum speed gap found for a flight, t the maximum time-step, the altitude is the usual cruising altitude and R the radius of the Earth. Since the error is almost null, it is considered to be an appropriate approximation.

- When calculating the speed at a certain time-stamp (for display purposes, not to be used in the calculation of positions) the Uniformly Accelerated Motion hypothesis has been used. Since the initial speed, final speed and the time-step are known, the speed is calculated as: $v = v_0 + at$
- The heading between two time-stamps has also been obtained by interpolation.

Haversine Formula, Spherical Law of Cosines and Spherical Law of Sines

The versine of an angle θ is 1-cos(θ). The haversine, which is a trigonometric function, is is in turn half the versine: $hav(\theta) = \frac{1-cos(\theta)}{2}$

The previous equation may be rewritten using the double angle formula ($cos(2x) = 1 - 2sin^2(x)$), as follows:

$$hav(\theta) = \frac{1 - cos(\theta)}{2} = \frac{1 - cos(\frac{2\theta}{2})}{2} = \frac{1 - \left(1 - 2sin^2(\frac{\theta}{2})\right)}{2} = sin^2(\frac{\theta}{2})$$

In order to calculate the distance between two desired locations, navigators used the Haversine Formula. This formula is obtained by substituting the definition of a haversine in the Law of Cosines.

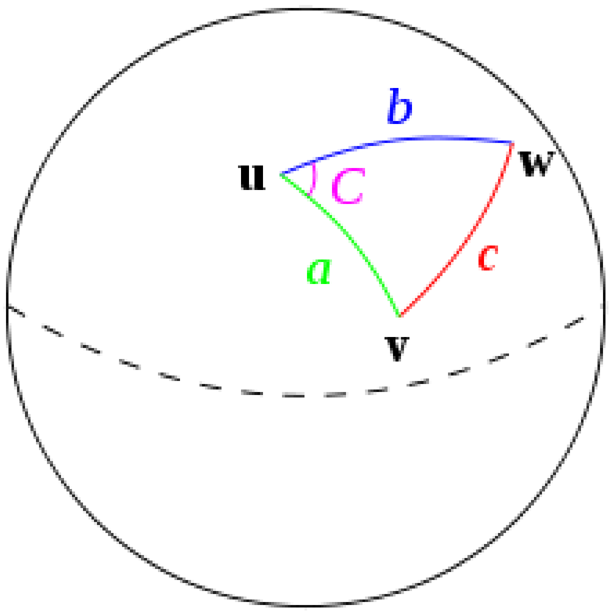


Figure 5.9: Spherical Law of Cosines

The Spherical Law of Cosines states that:

$$cos(c) = cos(a)cos(b) + sin(a)sin(b)cos(C) \tag{5.1}$$

Where a,b and c are spherical arcs and C is the spherical angle between a and b. Then by substituting cos(c) and cos(C) in terms of haversine:

$$1 - 2hav(c) = cos(a)cos(b) + sin(a)sin(b)[1 - 2hav(C)]$$

And then redistributing:

$$1 - 2hav(c) = cos(a)cos(b) + sin(a)sin(b) - 2sin(a)sin(b)hav(C)$$

And by replacing by the difference of cosines (cos(x-y)=cos(x)cos(y)+sin(x)sin(y)):

$$1 - 2hav(c) = 1 - 2hav(a - b) - 2sin(a)sin(b)hav(C)$$

Then substituting cos(a-b) in terms of haversine:

$$1 - 2hav(c) = 1 - 2hav(a - b) - 2sin(a)sin(b)hav(C)$$

And finally simplifying, the Law of Haversines is obtained:

$$hav(c) = hav(a - b) + sin(a)sin(b)hav(C) \tag{5.2}$$

From the Law of Haversines [74], let the central angle between two points on a sphere be $\theta = \frac{d}{R}$ where d is the distance between two points along a great circle of the sphere and R the radius of the sphere (in navigation the radius of Earth). Then, being φ_1 and φ_2 the latitudes of points 1 and 2, respectively; and λ_1 and λ_2 the longitudes of points 1 and 2, respectively; the Law of Haversines may be rewritten as follows:

$$hav(\theta) = hav(\frac{\theta}{R}) = hav(\varphi_2 - \varphi_1) + cos(\varphi_1)cos(\varphi_2)hav(\lambda_2 - \lambda_1)$$

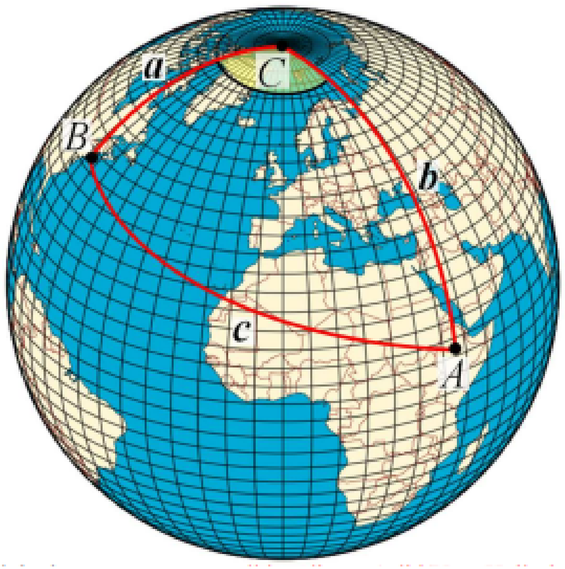


Figure 5.10: Haversine Formula [75]

From figure 5.10, it may be appreciated that the angle C is the difference between the points' longitudes, while edges a and b correspond to the difference between the North Pole's latitude (90°) and each point's latitude, respectively. In order to rewrite the equation, it has been taken into account the fact that $sin(\theta) = cos(\frac{\pi}{2} - \theta)$.

Then, by using the definition of the haversine:

$$sin^2(\frac{d}{2r}) = hav(\varphi_2 - \varphi_1) + cos(\varphi_1)cos(\varphi_2)hav(\lambda_2 - \lambda_1)$$

And finally:

$$d = 2r:\ arcsin\sqrt{sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + cos(\varphi_1)cos(\varphi_2)sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)} \tag{5.3}$$

Using the Spherical Law of Cosines, and knowing the following items of a test flight (initial latitude, initial longitude, altitude, speed, initial heading and a given time), the position of an aircraft may be calculated at any time.

From [75], considering point B to be the initial position and point A the final position, the spherical arc c may be calculated as follows:

$$c = \omega \times t = \frac{v}{R + altitude}t$$

Then, b may be calculated by using 5.1:

$$b = arccos\left(cos(a)cos(c) + sin(a)sin(c)cos(initial\ heading)\right)$$

From b, the final longitude may be obtained as: $lng = \frac{\pi}{2} - b$. Once b is known, using again 5.1, C may be obtained as:

$$C = arccos\left(\frac{cos(c) - cos(a)cos(b)}{sin(a)sin(b)}\right)$$

The heading of the new position is, in turn, obtained using the Spherical Law of Sines:

$$\frac{sin(a)}{sin(A)} = \frac{sin(b)}{sin(B)} = \frac{sin(c)}{sin(C)} \tag{5.4}$$

Considering the following hypothesis, the heading will either be the same as the initial heading (if the angle A is smaller than 90º), 180 + the initial heading (in case that the initial heading is smaller than 180° and the angle A is greater than 90°) or 180° - the initial heading (in case that the initial heading is greater than 180° and the angle A is greater than 90°). The angle A is used to establish whether the aircraft is on the "other side" of the world respect its initial position.

In order to view this hypothesis more clearly, let's set an example. Providing that an aircraft is heading south with an initial heading of 180º, it will eventually reach a point (past the South Pole), where it will start heading from South to North instead of North to South, hence the heading shall change from 180° to 0°. A combination of the Spherical Law of Cosines and the Spherical Law of Sines has been used to calculate the arc that an aircraft should do from its initial position to reach the critical point (where A equals 90°) and changes its heading.

By iterating through the following equation system, the software calculates such point:

$$b = arccos\left(cos(a)cos(c) + sin(a)sin(c)cos(initial\ heading)\right)$$

$$C = arccos\left(\frac{cos(c) - cos(a)cos(b)}{sin(a)sin(b)}\right)$$

$$\frac{sin\ a}{sin\ A} = \frac{sin\ c}{sin\ C} \longrightarrow sin\ a - \frac{sin\ c}{sin\ C} = 0$$

Note that A is 90° and that sin(90)=1. By iterating c from 0 to 360, at least a point should be found where the equation above is true.

Chapter 6

Results

In this chapter an analysis on the project's results shall be done.

Regarding the first task of this project, an application which successfully connects to the online flight database and saves flight data routes has been developed. On chapter 4, an overview on the architecture of such files has been displayed. These files are too big to fit in this project, which is why they won't be displayed in their totality. The output of the flight route data gathering application has been used as an input for the Silleo-SeamSAT software.

This next section shows the studies and results performed with the modified simulator and states the process that may be conducted to analyze communications in a LEO constellation between AC, GS and satellites.

As shown in the previous chapter, the user of the Silleo-SeamSAT simulator, may obtain the following items as a result of running the software:

- Visual representation of the LEO constellation network with satellites, GS, AC and their respective links
- GMLs

The GMLs are the most important output files of the simulator. They may be post-processed to analyze the performance of the links and improve the tool used to calculate the best communication paths. This tool is used to calculate path routes and its analysis is out of scope for this project.

The proper performance of the software has been tested as follows.

On one hand, some AC (either from routes or test flights) have been input to the simulator. It has been checked that both AC and their links were properly displayed by the graphical render. In figure 6.1 AC2, AC6 and AC3 and their respective links are shown. These AC are test flights and have been selected properly to easily check the coding of aircraft motion propagation algorithm.

AC name	lat	lon	alt m	hd	spd m/s	antennas
AC2	10.0	10.0	10000	80.0	250.0	2
AC3	0.0	0.0	10000	0.0	250.0	2
AC6	20.0	20.0	10000	90.0	250.0	2

Table 6.1: testFlight configuration

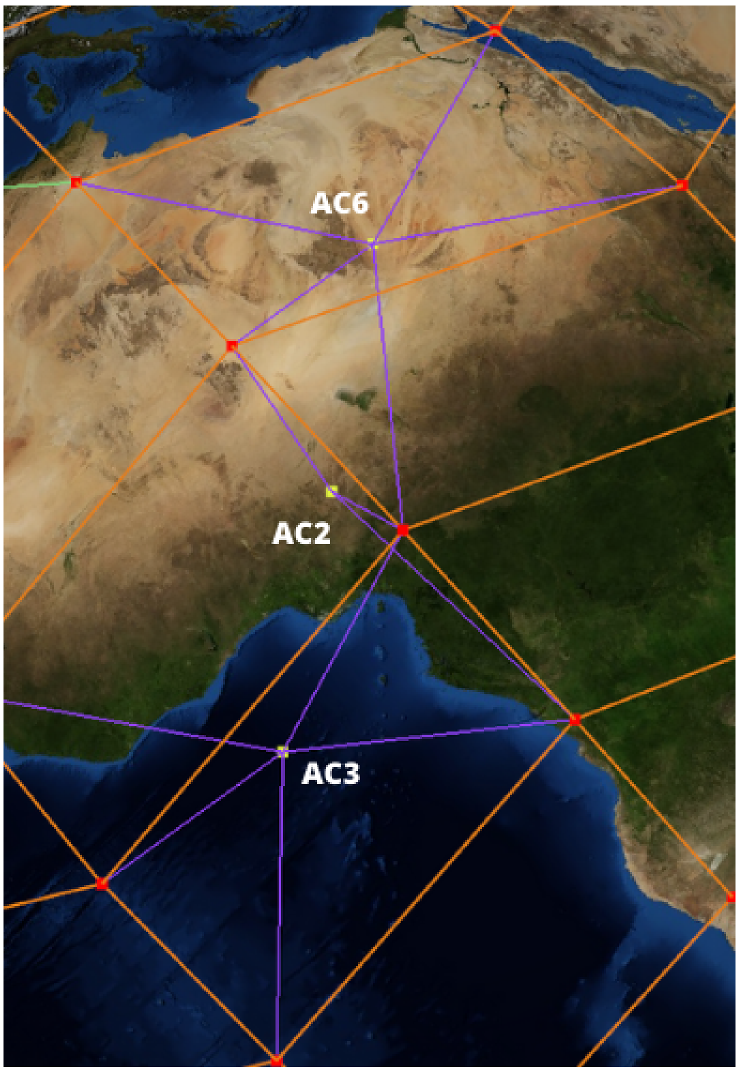


Figure 6.1: AC2, AC3 and AC6 graphic render display

On the other hand, the software has been executed with a time-step of 2 hours, the simulation status has been saved for each timestamp in a GML until these AC have completed a round-trip around the Earth. Below you may find the data stored in each GML for these AC, hence proving that the simulator works properly.

As it may be seen in the table 6.2, AC3 follows a vertical trajectory and its longitudes are either 0 or 180°, which makes the airplane easy to check the computed positions. Similarly, AC6 follows a trajectory parallel to the Equator, keeping always a latitude of 20º, which makes it easy to check too. Finally, AC2 follows a trajectory with a heading of 80º, this flight is used to check that aircraft that don't follow neither a vertical nor a horizontal trajectory, work properly.

	AC2		AC3		AC6	
t (s)	lat (º)	lon (º)	lat (º)	lon (º)	lat (º)	lon (º)
0	10.00	10.00	0.00	0.00	20.00	20.00
7200	12.38	26.28	16.14	0.00	20.00	36.14
14400	13.78	42.80	32.29	0.00	20.00	52.29
21600	14.07	59.43	48.43	0.00	20.00	68.43
28800	13.24	76.03	64.58	0.00	20.00	84.58
36000	11.35	92.44	80.72	0.00	20.00	100.72
43200	8.57	108.59	83.14	0.00	20.00	116.87
50400	5.14	124.48	66.99	180	20.00	133.01
57600	1.32	140.19	50.85	180	20.00	149.16
64800	-2.60	155.86	34.7	180	20.00	165.30
72000	-6.33	171.62	18.56	180	20.00	-178.56
79200	-9.57	-172.41	2.41	180	20.00	-162.41
86400	-12.08	176.17	-13.73	180	20.00	-146.27
93600	-13.64	159.68	-29.88	180	20.00	-130.12
100800	-14.10	143.06	-46.02	180	20.00	-113.98
108000	-13.44	126.44	-62.17	180	20.00	-97.84
115200	-11.69	110.00	-78.31	180	20.00	-81.69
122400	-9.03	93.81	-85.55	0.00	20.00	-65.55
129600	-5.68	77.88	-69.40	0.00	20.00	-49.01
136800	-1.90	62.15	-53.26	0.00	20.00	-33.26
144000	2.02	46.48	-37.11	0.00	20.00	-17.11
151200	5.79	30.75	-20.97	0.00	20.00	-0.97
158400	9.13	14.81	-4.82	0.00	20.00	15.18
165600	11.76	21.39	11.32	0.00	20.00	31.32
172800	13.47	37.84	27.46	0.00	20.00	47.46

Table 6.2: AC GMLs positions during 1 Earth's round-trip

Chapter 7

Budget

In this chapter, the costs associated to the development of this project are detailed.

7.1 Human resources

The human resources costs correspond to the human hours spent to the development of this project.

It may be assumed that the student and the tutor have spent the following amount of hours developing this project:

- Student: 400 hours
- Tutor: 40 hours

Assuming that the student has earned 15€/h and that the tutor has earned about 30€/h, the total human costs would increment up to 7200€.

7.2 Equipment

In order to develop this project, no license has need to be purchased. All codes have been developed using Python, which is an open source programming language (therefore free).

The only equipment required to the development of this project has been a computer with access to WiFi.

Since the weekly meetings with the tutor have been online, no fuel has been spent to go the university. Hence, fuel costs are not a factor.

Assuming that the computer has cost 1000€ and that the monthly fee for the WiFi access is of 15€, then the equipment costs have been of 1060€.

7.3 Total costs

The following table reflects the total costs required for the development of this project.

	Human Resources	**Equipment**	**Total**
Cost (€)	7200	1060	8260

Table 7.1: Total cost of the project

Chapter 8

Environmental and social implications

In this chapter the future environmental and social implications of this project are detailed. Furthermore, the carbon footprint generated by the development of this project is detailed too.

By post-processing the results of the software developed in this project, it may be analyzed the best way to improve security and ATM services for inter-continental flights. This wouldn't only make these flights more safe, but it would also optimize the used routes. Which would, in turn, the amount of fuel required to accomplish these flights. Globally, the aerospace industry would become more sustainable and environmentally friendly.

On the other hand, no major carbon footprint has been generated by developing this project. Since the weekly meetings have been performed via online, no pollution has been generated to travel to the university. Additionally, the electrical consumption of the computer has been fed by a solar installation. Which means that almost no energy has been consumed from the electrical grid.

Overall, this project has helped to have a more sustainable world.

Chapter 9

Conclusions

This project studies to analysis, development and implementation of an aircraft mobile node module for the Silleo-SeamSat LEO constellation simulator. As a result, a tool used to download aircraft routes has been developed.

In order to successfully achieve the objective of this project, a series of changes have been made to the Silleo-SeamSat simulation tool. This tool was able to simulate large LEO constellations provide information about the satellites and analyze the links between satellites and with ground stations. However, the software didn't have the capability to do so with aircraft. Especially, with aircraft flying inter-continental routes, which are the main aim of this project. The implementation of this aircraft node module for the simulation tool which has been explained in this project, shall allow future analysis of the performance of their links with the constellation network.

Finally, it has been shown that this tool would be useful to simulate the tracking of aircraft flying inter-continental routes. This next step in the aerospace industry will have significant implications improving ATM services, as well as improving reliability and safety of this type of flights. This system will also help to unsaturate current air routes and optimize them. As a result, this would reduce the amount of fuel required to perform inter-continental flights, making the aerospace industry more sustainable and environmentally-friendly.

Future work

To extend this work further, there are several avenues of research that may be expanded upon and some aspects that could be improved:

- Erase the `calculateIdealLink` method and all aspects related to it: As explained during this project, this way of calculating links for the satellite network is not realistic. It is a method outdated and out of use and, as a result, should be erased. However, this may be a tedious task because some parts of the software architecture should have to be redesigned in order not to have this feature.

- Aircraft antenna pointing: For this project, it has been assumed that all aircraft have antennas with the same field of view than ground station's antennas. However, in future works it may be implemented a function that allows the user to change such field of view in the configuration file. Moreover, an analysis on the appropriate inclination that antennas should have in order to improve communications may be effected. This study could determine whether there would be an optimal antenna pointing depending on the shape of the satellite LEO constellation.
- Analysis of the GMLs: The saved GMLs may be post-process to further improve the tool that calculates the best path for communications. The performance of such links may be analyzed too.
- Improve the user interface: Most of the parameters involved in the simulation must be entered through the program code. This results in a tedious process to study how the variation of a parameter affects the simulation output, because in order to implement such change, the simulation must be restarted every time. A more user friendly interface may be a future implementation for the Silleo-SeamSat simulator.

Chapter 10

Annex A

aircraft.py

```
# Import libraries
import numpy as np
import pandas as pd
import json
import os
import sys
import datetime
import time
import argparse

IBAN_ALPHA2_CODES= [ "AF", "AX", "AL", "DZ", "AS", "AD", "AO", "AI", "AQ", "AG",
                    "AR", "AM", "AW", "AU", "AT", "AZ", "BS", "BH", "BD", "BB",
                    "BY", "BE", "BZ", "BJ", "BM", "BT", "BO", "BQ", "BA", "BW",
                    "BV", "BR", "IO", "BN", "BG", "BF", "BI", "KH", "CM", "CA",
                    "CV", "KY", "CF", "TD", "CL", "CN", "CX", "CC", "CO", "KM",
                    "CG", "CD", "CK", "CR", "CI", "HR", "CU", "CW", "CY", "CZ",
                    "DK", "DJ", "DM", "DO", "EC", "EG", "SV", "GQ", "ER", "EE",
                    "ET", "FK", "FO", "FJ", "FI", "FR", "GF", "PF", "TF", "GA",
                    "GM", "GE", "DE", "GH", "GI", "GR", "GL", "GD", "GP", "GU",
                    "GT", "GG", "GN", "GW", "GY", "HT", "HM", "VA", "HN", "HK",
                    "HU", "IS", "IN", "ID", "IR", "IQ", "IE", "IM", "IL", "IT",
                    "JM", "JP", "JE", "JO", "KZ", "KE", "KI", "KP", "KR", "KW",
                    "KG", "LA", "LV", "LB", "LS", "LR", "LY", "LI", "LT", "LU",
                    "MO", "MK", "MG", "MW", "MY", "MV", "ML", "MT", "MH", "MQ",
                    "MR", "MU", "YT", "MX", "FM", "MD", "MC", "MN", "ME", "MS",
                    "MA", "MZ", "MM", "NA", "NR", "NP", "NL", "NC", "NZ", "NI",
                    "NE", "NG", "NU", "NF", "MP", "NO", "OM", "PK", "PW", "PS",
                  "PA", "PG", "PY", "PE", "PH", "PN", "PL", "PT", "PR", "QA",
                  "RE", "RO", "RU", "RW", "BL", "SH", "KN", "LC", "MF", "PM",
                  "VC", "WS", "SM", "ST", "SA", "SN", "RS", "SC", "SL", "SG",
                  "SX", "SK", "SI", "SB", "SO", "ZA", "GS", "SS", "ES", "LK",
                  "SD", "SR", "SJ", "SZ", "SE", "CH", "SY", "TW", "TJ", "TZ",
                  "TH", "TL", "TG", "TK", "TO", "TT", "TN", "TR", "TM", "TC",
                  "TV", "UG", "UA", "AE", "GB", "US", "UM", "UY", "UZ", "VU",
                  "VE", "VN", "VG", "VI", "WF", "EH", "YE", "ZM", "ZW"]
def main(args):
    # Creation of FlightRadar24 object
    from FlightRadar24.api import FlightRadar24API
    fr_api = FlightRadar24API()

    # Set initial configuration
    fr_api.get_real_time_flight_tracker_config()

    # Set seconds prior to current time, data gathered from that point onwards
    # Get current time in Unix time
    currentTime = datetime.datetime.now()
    currentTime = int(time.mktime(currentTime.timetuple()))
    # Change specfified time in Y M D into Unix time
    limitTime = datetime.datetime(
        int(args[len(args)-3]), int(args[len(args)-2]), int(args[len(args)-1]))
    limitTime = int(time.mktime(limitTime.timetuple()))
    fr_api.set_real_time_flight_tracker_config(maxage=str(currentTime-limitTime))

    # Do not take gliders into account
    fr_api.set_real_time_flight_tracker_config(gliders="0")

    # Do not take vehicles into account
    fr_api.set_real_time_flight_tracker_config(vehicles="0")

    # Set max limit
    fr_api.set_real_time_flight_tracker_config(limit="100")

    # Do not take aircraft that is still on air into account
    fr_api.set_real_time_flight_tracker_config(air="0")

    # Set new configuration
    fr_api.get_real_time_flight_tracker_config()

    # Get zones list
    zones = fr_api.get_zones()

    # Get airports list
    airports = fr_api.get_airports()

    # Set boundaries
    # bounds = fr_api.get_bounds(zones["europe"])

    # Get flights list
    print("Connecting␣to␣FlightRadar␣and␣getting␣flights␣...")
    flights = fr_api.get_flights()

    routes = []
    data = []
    alternatives = []
    print_routes = []

    # The command line must be similar to: py aircraft.py routes US US 2023 2 25
    # In order to know the country code, see the alpha-2 code in:
    # https://www.iban.com/country-codes
    if args[1] == "routes":
        print("Processing␣flights␣...")
        for i in range(np.size(flights)):
            # Set condition
            Flight = flights[i]
            details = fr_api.get_flight_details(Flight.id)
            try:
                if details["airport"]["origin"] is not None and details["airport"]["destination"]
                is not None:
                    #print("Flight id: %s and details %s" % (Flight.id, details))
                    # Filter flights that have not departed
                    if details["status"]["generic"]["status"]["text"] == "landed":
                        # Filter flights by origin and destination countries
                        origin_country = details["airport"]["origin"]["position"]
                            ["country"]["code"]
                        origin_airport = details["airport"]["origin"]["name"]
                        destination_country = details["airport"]["destination"]
                            ["position"]["country"]["code"]
                        destination_airport = details["airport"]["destination"]["name"]
                        if args[2] == origin_country and args[3] == destination_country:
                            #print("Fligth ID:", Flight.id)
                            routes.append(Flight.id)
                            print_routes.append((Flight.id,origin_country,origin_airport,
                                destination_country,destination_airport))
                        else:
                            #print(".", sep=' ', end='', flush=True)
                            information = {
                                "origin": origin_country, "destination": destination_country,
                                "flight␣ID": Flight.id}
                            alternatives.append(information)

                    print(".", end='', flush=True)
            except (TypeError):
                continue
        print("\nFound␣flights:␣")
        for flightId, orCountry, orAirport, dstCountry, dstAirport in print_routes:
           print("FlightId:␣%s␣---␣From␣%s␣(%s)␣To␣%s␣(%s)" % (flightId, orAirport,
                orCountry, dstAirport, dstCountry))

    # The command line must be similar to: py aircraft.py data 2f9307af ./UsUs01.json 2023 2 25
    elif args[1] == "data":
        details = fr_api.get_flight_details(args[2])
        try:
            if details["status"]["generic"]["status"]["text"] == "landed":
                # Change data to SI
                for key, value in details.items():
                    if key == "trail":
                        for j in range(len(value)):
                            # Change altitude from ft to m
                            value[j]["alt"] = value[j]["alt"]*0.3048
                            # Change speed from kts to m/s
                            value[j]["spd"] = value[j]["spd"]*0.514444
                            # Change Unix time to UTC time
                            value[j]["utc"] = str(pd.to_datetime(
                                value[j]["ts"], unit="s"))
                            # Set flight time in seconds
                            value[j]["fltime"] = value[j]["ts"] - \
                                details["firstTimestamp"]
                            # Set longitudes to positive
                            if value[j]["lng"]<0:
                                value[j]["lng"] = 360+value[j]["lng"]
                flightInfo = details.copy()

                # Erase all information not required
            rem_list = ["status", "level", "promote", "aircraft", "airline", "owner",
                  "airspace", "flightHistory", "ems", "availability", "airport",
                  "time", "firstTimestamp", "s"]
            for key in rem_list:
              flightInfo.pop(key, None)
            data.append(flightInfo)
            print("Data␣saved␣for␣flight:", args[2])

            # Create JSON file with flight data
            with open(args[3], "w", encoding="utf-8") as file:
              json.dump(flightInfo, file, ensure_ascii=False, indent="␣␣")

        except (TypeError):
          with open("./problems.json", "w", encoding="utf-8") as file:
            json.dump(details, file, ensure_ascii=False, indent="␣␣")
      else:
        print("No␣correct␣format␣used␣in␣the␣command␣line␣arguments")

      # Message printed if no routes have been found
      if args[1] == "routes" and np.size(routes) < 1:
        print("\n")
        print("No␣matching␣routes␣were␣found,␣however␣you␣may␣see␣in␣the␣alternatives.json␣file␣some␣other␣found␣routes")
        with open("./alternatives.json", "w", encoding="utf-8") as file:
          json.dump(alternatives, file, ensure_ascii=False, indent="␣␣")

      elif args[1] == "data" and np.size(data) < 1:
        print("No␣matching␣data␣found␣for␣flight:", args[2])


    def usage():
      msg = """
      This tool has two operation modes:
      1.- 'routes' mode is used to download from FlighRadar24 a list
        of flight ids which match with the provided parameters.
        Ex:

        aircrafts.py routes NL US 2023 5 10

        Download and displays flight ids from origin country NL to destination country US
        in the period from 2023/5/10 until now

    2.- 'data' mode is used to download the trail followed by an aircraft route
        identifier. This data is saved in a file to be used with silleo-SeamSAT
        Ex.

        aircrafts.py data 307cc5a6 Amsterdam-SaltLakeCity.json 2023 5 25

        Downloads trail data and saves it in the json file Amsterdam-SaltLakeCity.json
    """
    print(msg)
    sys.exit(1)

if __name__ == "__main__":

    if len(sys.argv[1:]) != 6:
        usage()

    if sys.argv[1] != 'routes' and sys.argv[1] != 'data':
        usage()

    year = int(sys.argv[4])
    month = int(sys.argv[5])
    day = int(sys.argv[6])

    if month not in range(1,12) or day not in range(1,31):
        usage()

    if sys.argv[1] == 'routes':
        if sys.argv[2] not in IBAN_ALPHA2_CODES or sys.argv[3] not in IBAN_ALPHA2_CODES:
            usage()



    main(sys.argv)
```

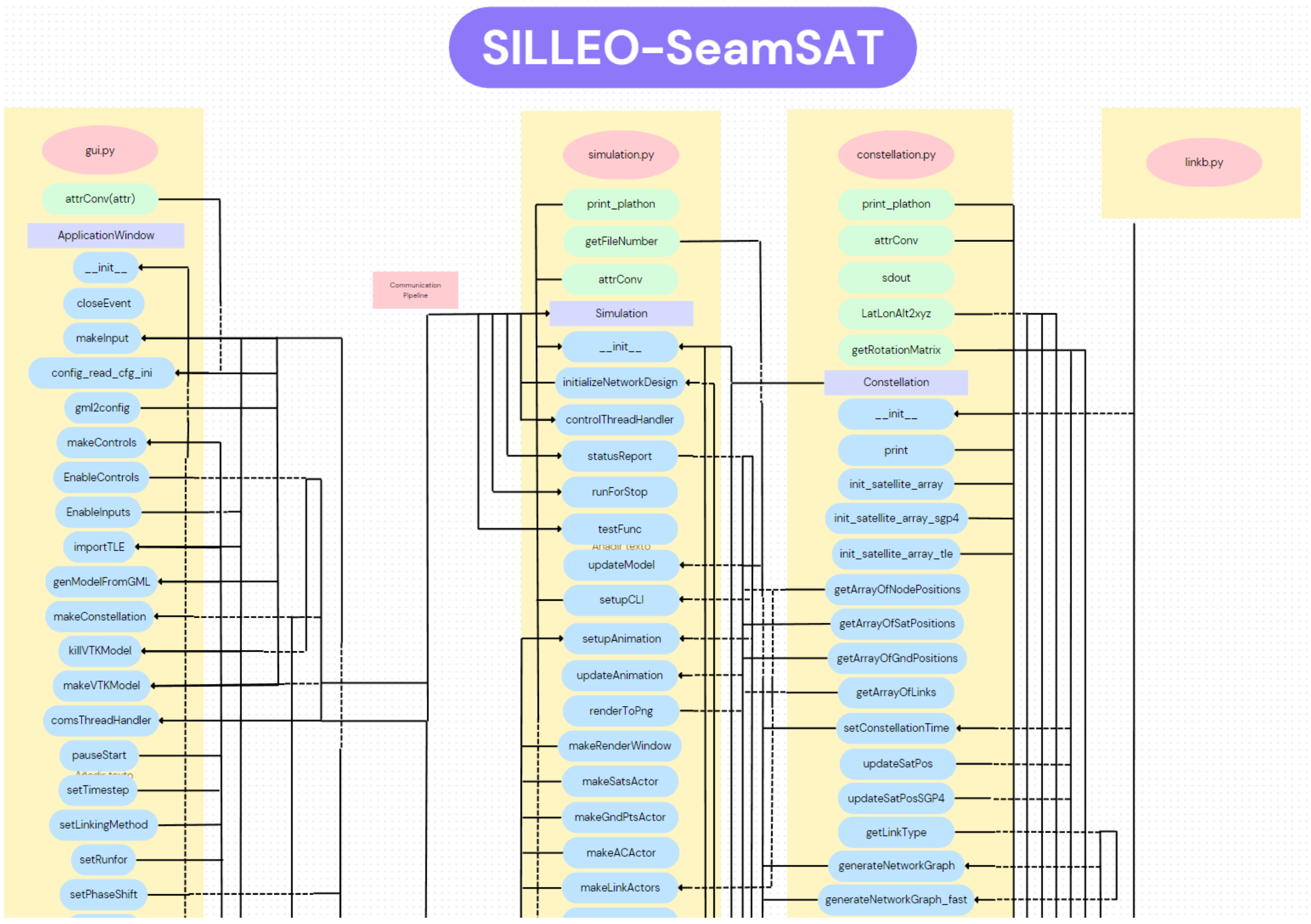
Chapter 11

Annex B

```
class ACflights():
  def __init__(self):
    self.aircraft_node_counter = 0
    self.ground_node_counter = 0

    # declare an empty aircraft
    self.aircraft_array = np.zeros(
      NUM_AIRCRAFT, dtype=AIRCRAFT_DTYPE)

  def setACData(self):
    # add aircraft at initial position to the constellation model
    self.aircraft_names = []
    self.ac_data = self.aircraftData(0)
    for i in range(0, len(self.ac_data)):
      self.aircraft_names.append(self.ac_data[i]["name"])
      self.addACData(float(self.ac_data[i]["lat"]), float(self.ac_data[i]["lng"]), float(
        self.ac_data[i]["alt"]), float(self.ac_data[i]["spd"]), float(self.ac_data[i]["vz"]),
          float(self.ac_data[i]["hd"]), int(self.ac_data[i]["maxAntennas"]))

  def setGroundNodeCounter(self, ground_node_counter):
    self.ground_node_counter = ground_node_counter

  def setGhaAries(self, GHA_Aries):
    self.GHA_Aries = GHA_Aries

  def aircraftData(self, time):
    time = int(time)

    data = []
    route_names = []
    aircraft_names = []
    timestamps = []
    antennas = []
    route_data = []
    flight_info = []
    initial_lat = []
    initial_lng = []
    alt_m = []
    initial_hd = []
    flight_spd = []
    test_antennas = []
    test_names = []

    # Read ac.txt file
    cwd = os.getcwd() # working folder path (depends on user)
    # aircraft folder path from working folder (source)
    relative_path = "ac_txt"
    # absolute path to folder for flight routes
    folderpath = os.path.join(cwd, relative_path)
    ac_file = "ac.txt"
    with open(os.path.join(folderpath, ac_file), 'r') as f:
        for line in f:
            # do not take comment lines into account
            if line[0] == "#":
                continue
            my_line = []
            for word in line.split():
                my_line.append(word)
            data.append(my_line)

    for i in range(1, len(data)):
        if data[i][0] == "testFlight":
            test_names.append(data[i][1])
            initial_lat.append(float(data[i][2]))
            initial_lng.append(float(data[i][3]))
            alt_m.append(float(data[i][4]))
            initial_hd.append(float(data[i][5]))
            flight_spd.append(float(data[i][6]))
            test_antennas.append(data[i][7])

        else:
            route_names.append(data[i][0])
            aircraft_names.append(data[i][1])
            timestamps.append(data[i][2])
            antennas.append(data[i][3])

    # Create TEST flight
    #if np.size(test_names) == 0:
    # print("No test flights found")

    else:
        for i in range(0, len(test_names)):
            if time == 0:
                vz = 0
                info = {"name": test_names[i], "lat": initial_lat[i], "lng": initial_lng[i], "alt": alt_m[i], "spd": flight_spd[i], "vz": vz, "hd": initial_hd[i], "fltime": time, "maxAntennas": test_antennas[i]}

            else:
                vz = 0
                if initial_hd[i] == 0 or initial_hd[i] == 180:
                    # Use Spherical Law of Cosines
                    a = math.radians(90 - initial_lat[i])
                    radius = EARTH_RADIUS + alt_m[i]
                    c = (flight_spd[i]/radius)*time
                    b = math.acos(math.cos(a)*math.cos(c) + math.sin(a)*math.sin(c)*math.cos(math.radians(initial_hd[i])))

                    if (math.cos(c)-math.cos(a) * math.cos(b))/(math.sin(a)*math.sin(b)) > 1:
                        C = math.acos(1)
                    elif (math.cos(c)-math.cos(a) * math.cos(b))/(math.sin(a)*math.sin(b)) < -1:
                        C = math.acos(-1)
                    else:
                        C = math.acos((math.cos(c)-math.cos(a) * math.cos(b))/(math.sin(a)*math.sin(b)))

                    lat = 90 - math.degrees(b)
                    lng = initial_lng[i] + math.degrees(C)

                    if lng - initial_lng[i] < 90:
                        lng = initial_lng[i]
                        hd = initial_hd[i]
                    else:
                        hd = (initial_hd[i] + 180) % 360

                    info = {"name": test_names[i], "lat": lat, "lng": lng, "alt": alt_m[i],
                            "spd": flight_spd[i], "vz": vz, "hd": hd, "fltime": time, "maxAntennas": test_antennas[i]}

                elif initial_hd[i] == 90 or initial_hd[i] == 270:
                    radius = EARTH_RADIUS + alt_m[i]
                    c = (flight_spd[i]/radius)*time
                    if initial_hd[i] == 90:
                        lng = math.degrees(
                            math.radians(initial_lng[i]) + c)
                    else:
                        lng = math.degrees(
                            math.radians(initial_lng[i]) - c)

                    info = {"name": test_names[i], "lat": initial_lat[i], "lng": lng, "alt": alt_m[i],
                            "spd": flight_spd[i], "vz": vz, "hd": initial_hd[i], "fltime": time,
                                    "maxAntennas": test_antennas[i]}

                else:
                    # Use Spherical Law of Cosines
                    a = math.radians(90 - initial_lat[i])
                    radius = EARTH_RADIUS + alt_m[i]
                    c = (flight_spd[i]/radius)*time
                    b = math.acos(math.cos(a)*math.cos(c) + math.sin(a)*math.sin(c)*math.cos(math.radians(initial_hd[i])))
                    C = math.acos((math.cos(c)-math.cos(a) * math.cos(b))/(math.sin(a)*math.sin(b)))
                    lat = 90 - math.degrees(b)
                    lng = initial_lng[i] + math.degrees(C)

                    # Iterate using Spherical Law of Sines
                    found = False
                    arc = 0
                    precision = 0.01
                    while found == False and arc <= 360:
                        beta = math.acos(math.cos(a)*math.cos(arc) + math.sin(a)*math.sin(arc)*math.cos(math.radians(initial_hd[i])))
                        cosAngle = (math.cos(arc)-math.cos(a) * math.cos(beta))/(math.sin(a)*math.sin(beta))
                        if cosAngle > 1:
                            cosAngle = 1
                        angle = math.acos(cosAngle)
```

```
                    if math.sin(angle) != 0:
                        if math.sin(a)-(math.sin(arc)/math.sin(angle)) < precision or (math.sin(arc)/math.sin(angle))-math.sin(a) < precision:
                            found = True
                            if ((c % math.radians(360)) <= arc) or ((c % math.radians(360)) >= ((arc + math.radians(180)) % math.radians(360))):
                                hd = initial_hd[i]
                            else:
                                hd = (initial_hd[i] + 180) % 360

                    else:
                        arc = arc+math.radians(0.5/360)

                info = {"name": test_names[i], "lat": lat, "lng": lng, "alt": alt_m[i], "spd": flight_spd[i], "vz": vz, "hd": hd, "fltime": time, "maxAntennas": test_antennas[i]}
        flight_info.append(info)

# Read routes's JSON file
if np.size(route_names) == 0:
    print("␣No␣flight␣routes␣found")

else:
    for i in range(0, len(route_names)):
        with open(os.path.join(folderpath, route_names[i]), 'r') as f:
            route_data.append(json.load(f))

    # Flight info when the plane has landed
    for j in range(0, len(route_data)):
        # Iterators
        i = 0
        data_found = False

        # Flight info when the plane has landed
        if time > route_data[j]["trail"][0]["fltime"]+int(timestamps[j]):
            lat = route_data[j]["trail"][0]["lat"]
            lng = route_data[j]["trail"][0]["lng"]
            alt = route_data[j]["trail"][0]["alt"]
            spd = route_data[j]["trail"][0]["spd"]
            hd = route_data[j]["trail"][0]["hd"]
            fltime = time
            vz = 0

            # Flight info before takeoff
            elif time <= route_data[j]["trail"][np.size(route_data[j]["trail"])-1]["fltime"]+
                   int(timestamps[j]):
               lat = route_data[j]["trail"][np.size(
                  route_data[j]["trail"])-1]["lat"]
               lng = route_data[j]["trail"][np.size(
                  route_data[j]["trail"])-1]["lng"]
               alt = route_data[j]["trail"][np.size(
                  route_data[j]["trail"])-1]["alt"]
               spd = route_data[j]["trail"][np.size(
                  route_data[j]["trail"])-1]["spd"]
               hd = route_data[j]["trail"][np.size(
                  route_data[j]["trail"])-1]["hd"]
               fltime = time
               vz = 0

            # Flight info after takeoff
            else:
               while i < len(route_data[j]["trail"]) and data_found == False:
                  if route_data[j]["trail"][np.size(route_data[j]["trail"])-1-i]["fltime"]+
                           int(timestamps[j]) == time:
                     lat = route_data[j]["trail"][np.size(
                        route_data[j]["trail"])-1-i]["lat"]
                     lng = route_data[j]["trail"][np.size(
                        route_data[j]["trail"])-1-i]["lng"]
                     alt = route_data[j]["trail"][np.size(
                        route_data[j]["trail"])-1-i]["alt"]
                     spd = route_data[j]["trail"][np.size(
                        route_data[j]["trail"])-1-i]["spd"]
                     hd = route_data[j]["trail"][np.size(
                        route_data[j]["trail"])-1-i]["hd"]
                     fltime = time
                     alt0 = route_data[j]["trail"][np.size(
                        route_data[j]["trail"])-i]["alt"]
                     t0 = route_data[j]["trail"][np.size(
                        route_data[j]["trail"])-i]["fltime"]
                     vz = (alt-alt0)/(time-t0)

                     data_found = True

                  elif route_data[j]["trail"][np.size(route_data[j]["trail"])-1-i]["fltime"]+
                             int(timestamps[j]) < time:
                     i = i+1

                 else:
                     # Iterate to find the correct fliht info for fltime == time
                     t1 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-1-i]["fltime"]+int(timestamps[j])
                     t0 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-i]["fltime"]+int(timestamps[j])
                     fltime = time

                     lat1 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-1-i]["lat"]
                     lat0 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-i]["lat"]
                     lat = ((time-t1)/(t1-t0))*(lat1-lat0)+lat1

                     lng1 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-1-i]["lng"]
                     lng0 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-i]["lng"]
                     lng = ((time-t1)/(t1-t0))*(lng1-lng0)+lng1

                     alt1 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-1-i]["alt"]
                     alt0 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-i]["alt"]
                     alt = ((time-t1)/(t1-t0))*(alt1-alt0)+alt1

                     spd1 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-1-i]["spd"]
                     spd0 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-i]["spd"]
                     # aircraft's speed in the plane perpendicular to the aircraft's yaw axis
                     spd = ((time-t1)/(t1-t0))*(spd1-spd0)+spd1

                     # aircraft's speed in the yaw axis
                     vz = (alt-alt0)/(time-t0)

                     hd1 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-1-i]["hd"]
                     hd0 = route_data[j]["trail"][np.size(
                         route_data[j]["trail"])-i]["hd"]
                     hd = ((time-t1)/(t1-t0))*(hd1-hd0)+hd1

                     data_found = True

             info = {"name": aircraft_names[j], "lat": lat, "lng": lng, "alt": alt, "spd": spd,
               "vz": vz, "hd": hd, "fltime": fltime, "maxAntennas": antennas[j]}
             flight_info.append(info)

         return flight_info

def getArrayOfACPositions(self) -> npt.NDArray[typing.Any]:
    """copies a sub array of only position data from aircraft array

    Returns
    -------
    aircraft_positions : np array
          a copied sub array of the aircraft point array, that only contains positions
    """

    aircraft_positions = np.copy(
        self.aircraft_array[['x', 'y', 'z']])

    return aircraft_positions

def addACData(self, latitude, longitude, altitude, speed, vz, heading, maxAntennas):
    """ adds a AC at given coordinates, assumes earth is perfect sphere

    Parameters
    ----------
    latitude : float
          latitude of aircraft (in degrees)
    longitude : float
          longitude of aircraft (in degrees)
    altitude : float
          altitude of aircraft in meters (0 = earth surface)
    speed : float
          speed of aircraft in meters per second
    heading : float
          heading of aircraft (in degrees)

    Returns
```

```
        -------
        unique_id : int
            the ID value assigned to the aircraft (will be < 0)
        """

        # must convert the lat/long/alt to cartesian coordinates
        pos = LatLonAlt2xyz(latitude, longitude, altitude)

        # XXX: PLATHON Rotate GMST for aircraft points
        rotation_matrix = getRotationMatrix(
            EARTH_ROTATION_AXIS, self.GHA_Aries)

        pos = np.dot(rotation_matrix, pos)

        # be sure to decrement this for the next aircraft
        self.aircraft_node_counter = self.aircraft_node_counter - 1
        unique_id = self.ground_node_counter + self.aircraft_node_counter

        # add the new aircraft to array
        # yes, append means a full array copy every time,
        # but this should be a very small array,
        # and aircraft are probably only added once
        # at the beginning of the simulation
        temp = np.zeros(1, dtype=AIRCRAFT_DTYPE)
        temp[0]['ID'] = np.int16(unique_id)
        temp[0]['x'] = np.float64(pos[0])
        temp[0]['y'] = np.float64(pos[1])
        temp[0]['z'] = np.float64(pos[2])
        temp[0]['alt_m'] = np.float32(altitude)
        temp[0]['maxAntenna'] = np.int16(maxAntennas)

        self.aircraft_array = np.append(self.aircraft_array, temp)
        return unique_id

    def updateACPos(self, current_time) -> None:
        self.current_time = int(current_time)
        ac_data = self.aircraftData(self.current_time)
        for ac_id in range(self.aircraft_array.size):
            latitude = float(ac_data[ac_id]["lat"])
            longitude = float(ac_data[ac_id]["lng"])
            altitude = float(ac_data[ac_id]["alt"])

            # must convert the lat/long/alt to cartesian coordinates
            pos = LatLonAlt2xyz(latitude, longitude, altitude)

            # PLATHON consideration of earth relative start position with Aries Point
            if self.current_time == 0 or self.current_time % SECONDS_PER_DAY == 0:
            degrees_to_rotate = 0
            else:
            degrees_to_rotate = 360.0 / \
                (SECONDS_PER_DAY / (self.current_time % SECONDS_PER_DAY))

            # XXX: PLATHON Rotate GMST for aircraft points
            rotation_matrix = getRotationMatrix(
            EARTH_ROTATION_AXIS, self.GHA_Aries+degrees_to_rotate)

            pos = np.dot(rotation_matrix, pos)

            self.aircraft_array[ac_id]['x'] = np.float64(pos[0])
            self.aircraft_array[ac_id]['y'] = np.float64(pos[1])
            self.aircraft_array[ac_id]['z'] = np.float64(pos[2])
            self.aircraft_array[ac_id]['alt_m'] = np.float32(altitude)
```


SILLEO-SeamSAT
gui.py
attrConv(attr)
ApplicationWindow
__init__
closeEvent
makeInput
config_read_cfg_ini
gml2config
makeControls
EnableControls
EnableInputs
importTLE
genModelFromGML
makeConstellation
killVTKModel
makeVTKModel
comsThreadHandler
pauseStart
setTimestep
setLinkingMethod
setRunfor
setPhaseShift
Communication Pipeline
simulation.py
print_plathon
getFileNumber
attrConv
Simulation
__init__
initializeNetworkDesign
controlThreadHandler
statusReport
runForStop
testFunc
updateModel
setupCLI
setupAnimation
updateAnimation
renderToPng
makeRenderWindow
makeSatsActor
makeGndPtsActor
makeACActor
makeLinkActors
constellation.py
print_plathon
attrConv
sdout
LatLonAlt2xyz
getRotationMatrix
Constellation
__init__
print
init_satellite_array
init_satellite_array_sgp4
init_satellite_array_tle
getArrayOfNodePositions
getArrayOfSatPositions
getArrayOfGndPositions
getArrayOfLinks
setConstellationTime
updateSatPos
updateSatPosSGP4
getLinkType
generateNetworkGraph
generateNetworkGraph_fast
linkb.py

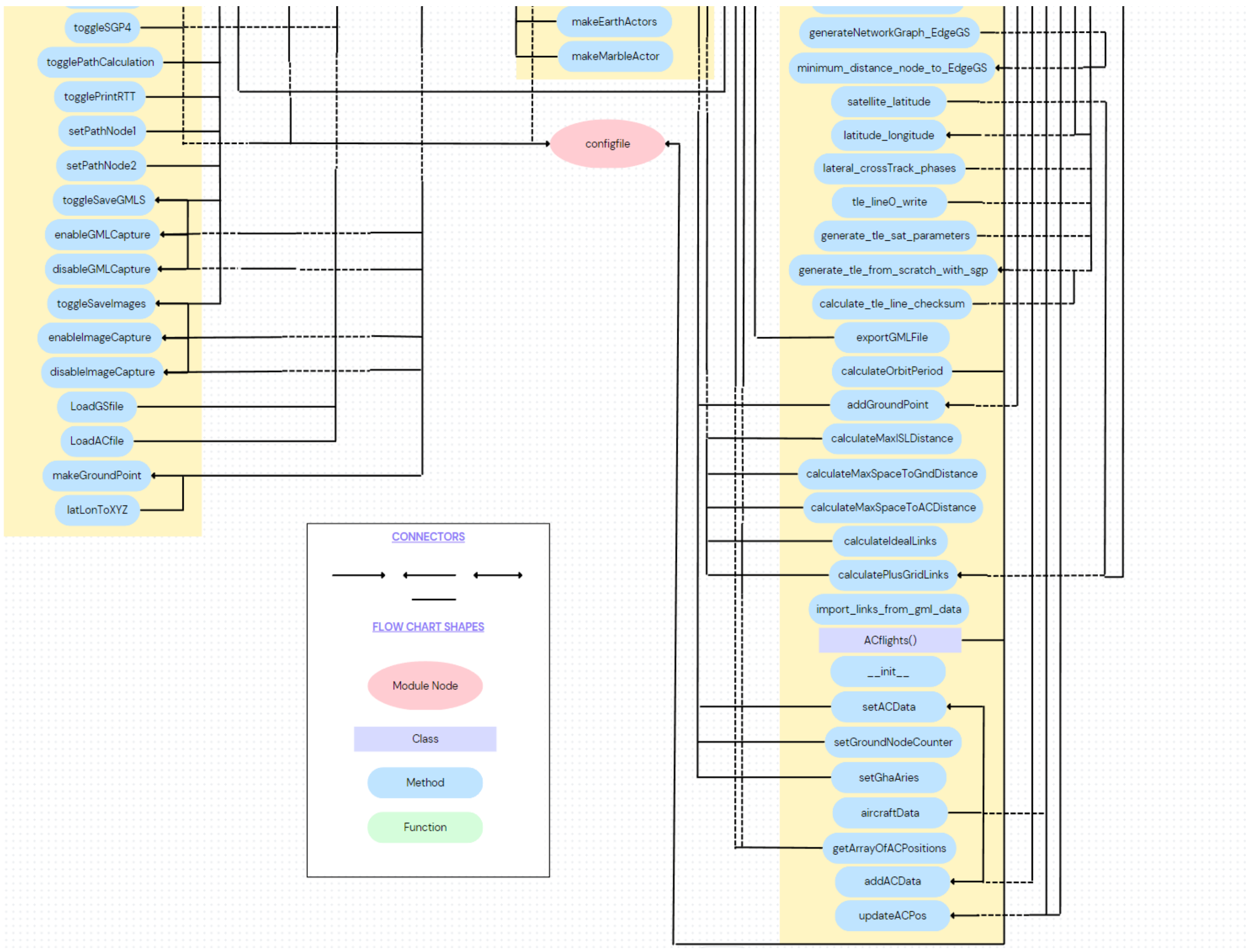


Figure 11.1: Silleo-SeamSat Flowchart

Bibliography

1. UPC. *Citar i elaborar la bibliografia*. 2021. Available also from: `https://bibliotecnica.upc.edu/investigadors/citar-elaborar-bibliografia`. Retrieved 16th February, 2023.
2. FORUM, World Economic. *How low-earth orbit satellite technology can connect the unconnected*. 2022. Available also from: `https://www.weforum.org/agenda/2022/02/explainer-how-low-earth-orbit-satellite-technology-can-connect-the-unconnected/`. Retrieved 26th February, 2023.
3. X, Space. *Starlink*. 2023. Available also from: `https://www.starlink.com`. Retrieved 25th February, 2023.
4. AMAZON. *Project Kuiper*. 2023. Available also from: `https://www.amazon.jobs/es/teams/projectkuiper`. Retrieved 27th January, 2023.
5. KEMPTON, Benjamin S. *A Simulation Tool to Study Routing in Large Broadband Satellite Networks*. 2020. ISBN 9798662568945. Available also from: `https://search.proquest.com/docview/2438605919?accountid=27868`. Retrieved 18th March, 2023.
6. SKYTRAC. *How are LEO and GEO Satellites Used in the Industry?* 2022. Available also from: `https://www.skytrac.ca/resources/magazine/how-are-leo-and-geo-satellites-used-in-the-industry/`. Retrieved 25th February, 2023.
7. BBC. *MH370: Could missing Malaysian Airlines plane finally be found?* 2021. Available also from: `https://www.bbc.com/news/business-59517821`. Retrieved 27th January, 2023.
8. UNDERTAKING, SESAR Joint. *Taking air traffic control into low orbit*. 2023. Available also from: `https://www.sesarju.eu/news/taking-air-traffic-control-low-orbit`. Retrieved 24th February, 2023.
9. INDRA. *Startical*. 2023. Available also from: `https://www.startical.com`. Retrieved 25th February, 2023.
10. INDRA. *Space Based CNS*. 2023. Available also from: `https://www.indracompany.com/sites/default/files/d7/Sectores/ATM/indra_space_based_cns.pdf`. Retrieved 18th February, 2023.
11. UNDERTAKING, SESAR Joint. *Extended Communications in vHf Over Enhanced Satellite segment (ECHOES)*. 2023. Available also from: `https://www.sesarju.eu/projects/ECHOES`. Retrieved 19th February, 2023.
12. NASA, Earth Observatory. *Catalog of Earth Satellite Orbits* [`https://earthobservatory.nasa.gov/features/OrbitsCatalog`]. 2009. Retrieved 25th February, 2023.

13. ESA. *Types of orbits* [https://www.esa.int/Enabling_Support/Space_Transportation/Types_of_orbits]. 2020. Retrieved 25th February, 2023.
14. NASA. *Basics of Space Flihgt*. 2023. Available also from: https://solarsystem.nasa.gov/basics/chapter5-1/. Retrieved 26th February, 2023.
15. HOUSTON, Space Center. *What are the Van Allen radiation belts?* 2020. Available also from: https://spacecenter.org/what-are-the-van-allen-radiation-belts/. Retrieved 26th February, 2023.
16. NASA. *NASA's Van Allen Probes Spot an Impenetrable Barrier in Space*. 2014. Available also from: https://www.nasa.gov/content/goddard/van-allen-probes-spot-impenetrable-barrier-in-space. Retrieved 25th February, 2023.
17. IRIDIUM. *Iridium* [https://www.iridium.com/network/]. 2023. Retrieved 27th January, 2023.
18. SKYBRARY. *aircraft-communications-addressing-and-reporting-system*. 2023. Available also from: https://www.skybrary.aero/articles/aircraft-communications-addressing-and-reporting-system. Retrieved 7th March, 2023.
19. EUROCONTROL. *Communications, Navigation and Surveillance*. 2023. Available also from: https://www.eurocontrol.int/communications-navigation-and-surveillance. Retrieved 7th March, 2023.
20. SESAR. *From Ground-based to space-based*. 2023. Available also from: https://www.sesarju.eu/sites/default/files/documents/webinars/webinar%20cns%20vision%20%20From%20ground%20to%20space%20%20Joan%20Manuel%20Cebrian%2C%20INDRA.pdf. Retrieved 8th March, 2023.
21. NEPAL, Aviation. *High frequency, very high frequency and transponder usage in aircraft*. 2020. Available also from: https://www.aviationnepal.com/high-frequency-hf-very-high-frequency-vhf-and-transponder-usage-in-aircraft/. Retrieved 10th March, 2023.
22. ETSI. *ETSI EN 301 841-1*. 2010. Available also from: https://www.etsi.org/deliver/etsi_en/301800_301899/30184101/01.03.01_40/en_30184101v010301o.pdf. Retrieved 11th March, 2023.
23. EUROCONTROL. *L-band digital aeronautical communication system*. 2023. Available also from: https://www.eurocontrol.int/system/l-band-digital-aeronautical-communication-system. Retrieved 6th March, 2023.
24. ICAO. *COMMUNICATIONS PANEL – DATA COMMUNICATIONS INFRASTRUCTURE WORKING GROUP*. 2019. Available also from: https://www.ldacs.com/wp-content/uploads/2013/12/ACP-DCIWG-IP01-LDACS-White-Paper.pdf. Retrieved 7th March, 2023.
25. EUROCONTROL. *Aeronautical mobile airport communications system datalink*. 2023. Available also from: https://www.eurocontrol.int/system/aeronautical-mobile-airport-communications-system-datalink. Retrieved 8th March, 2023.
26. WIMAX. *AeroMACS*. 2023. Available also from: https://wimaxforum.org/Page/AeroMACS. Retrieved 9th March, 2023.
27. SKYBRARY. *Satcom*. 2023. Available also from: https://www.skybrary.aero/articles/satcom.
28. EUROCONTROL. *Satellite communications datalink*. 2023. Available also from: https://www.eurocontrol.int/system/satellite-communications-datalink. Retrieved 9th March, 2023.

29. SPICEWORKS. *What is GNSS? Meaning, Working and Applications in 2022*. 2022. Available also from: https://www.spiceworks.com/tech/iot/articles/what-is-gnss/. Retrieved 12th March, 2023.
30. EUSPA. *What is GNSS?* 2021. Available also from: https://www.euspa.europa.eu/european-space/eu-space-programme/what-gnss. Retrieved 11th March, 2023.
31. SKYBRARY. *Ground based augmentation system GBAS*. 2023. Available also from: https://www.skybrary.aero/articles/ground-based-augmentation-system-gbas. Retrieved 12th March, 2023.
32. EUROCONTROL. *Ground based augmentation systems*. 2023. Available also from: https://www.eurocontrol.int/concept/ground-based-augmentation-systems. Retrieved 13th March, 2023.
33. EUSPA. *What is SBAS?* 2023. Available also from: https://www.euspa.europa.eu/european-space/eu-space-programme/what-sbas. Retrieved 13th March, 2023.
34. ESA. *SBAS fundamentals*. 2018. Available also from: https://gssc.esa.int/navipedia/index.php/SBAS_Fundamentals. Retrieved 14th March, 2023.
35. SKYBRARY. *Instrument landing system ILS*. 2023. Available also from: https://www.skybrary.aero/articles/instrument-landing-system-ils. Retrieved 14th March, 2023.
36. AIR, One. *What is ILS or instrument landing system*. 2023. Available also from: https://www.grupooneair.com/what-is-ils-instrument-landing-system/. Retrieved 15th March, 2023.
37. FLYING. *What is distance measuring equipment DME?* 2022. Available also from: https://www.flyingmag.com/guides/what-is-dme-everything-to-know/. Retrieved 15th March, 2023.
38. BOLDMETHOD. *How DME works*. 2021. Available also from: https://www.boldmethod.com/learn-to-fly/systems/understanding-dme-on-instrument-approaches-and-vfr-use/. Retrieved 16th March, 2023.
39. FLYING. *What is VOR? How a VOR Navigation system works*. 2022. Available also from: https://www.flyingmag.com/guides/what-is-vor-and-how-does-it-work/. Retrieved 16th March, 2023.
40. AIR, One. *What is VOR? What is DME?* 2023. Available also from: https://www.grupooneair.com/what-is-vor-dme/. Retrieved 17th March, 2023.
41. NOTEBOOK, CFI. *Non-directional radio beacon*. 2023. Available also from: https://www.cfinotebook.net/notebook/avionics-and-instruments/non-directional-radio-beacon. Retrieved 17th March, 2023.
42. FLIGHTRADAR24. *How flight tracking works*. 2023. Available also from: https://www.flightradar24.com/how-it-works. Retrieved 19th March, 2023.
43. SKYBRARY. *Automatic dependent surveillance broadcast ads-b*. 2023. Available also from: https://www.skybrary.aero/articles/automatic-dependent-surveillance-broadcast-ads-b. Retrieved 19th March, 2023.
44. AOPA. *WHAT YOU NEED TO KNOW ABOUT ADS-B*. 2023. Available also from: https://www.aopa.org/go-fly/aircraft-and-ownership/ads-b. Retrieved 20th March, 2023.
45. CANADA, Nav. *Satellite based ADS-B*. 2014. Available also from: https://www.icao.int/SAM/Documents/2014-SAMIG13/02%20NavCanada%20ADSB%20S.pdf. Retrieved 24th March, 2023.

46. GUY, The points. *How Satellite-Based Aircraft Tracking Will Revolutionize Flying.* 2019. Available also from: `https://thepointsguy.com/news/how-satellite-based-aircraft-tracking-will-revolutionize-flying/`. Retrieved 24th March, 2023.
47. Available also from: `https://www.icao.int/APAC/Documents/edocs/mlat_concept.pdf`. Retrieved 25th March, 2023.
48. ICAO. *Multilateration (MLAT) Concept of use.* 2007. Available also from: `https://www.flightradar24.com/blog/how-we-track-flights-with-mlat/`. Retrieved 25th March, 2023.
49. FLARM. *Flarm.* 2023. Available also from: `https://www.flarm.com`. Retrieved 26th March, 2023.
50. DEROSA, John. *Open Glider Network OGN.* 2021. Available also from: `http://aviation.derosaweb.net/presentations/documents/OGN_Open_Glider_Network.pdf`. Retrieved 26th March, 2023.
51. SKYBRARY. *Primary surveillance radar PSR.* 2023. Available also from: `https://skybrary.aero/articles/primary-surveillance-radar-psr`. Retrieved 27th March, 2023.
52. SKYBRARY. *Secondary surveillance radar SSR.* 2023. Available also from: `https://skybrary.aero/articles/secondary-surveillance-radar-ssr`. Retrieved 27th March, 2023.
53. GITHUB. *TLE.* 2018. Available also from: `https://juliaspace.github.io/SatelliteToolbox.jl/stable/man/orbit/tle/`. Retrieved 28th March, 2023.
54. STRATEGY, Space. *Detailed Description of "Two-Line Element (TLE)" Orbital Parameters.* 2017. Available also from: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119413585.app5`. Retrieved 28th March, 2023.
55. FLIGHTRADAR24. *Flightradar24.* 2023. Available also from: `https://www.flightradar24.com/`. Retrieved 29th March, 2023.
56. FLIGHTRADAR24. *Terms and conditions.* 2023. Available also from: `https://www.flightradar24.com/terms-and-conditions`. Retrieved 29th March, 2023.
57. NETWORK, Opensky. *Opensky Network.* 2023. Available also from: `https://opensky-network.org`. Retrieved 4th April, 2023.
58. AWARE, Flight. *Flight Aware.* 2023. Available also from: `https://flightaware.com`. Retrieved 4th April, 2023.
59. JESUS, Jean Loui Bernard Silva de. *FlightRadarAPI.* 2023. Available also from: `https://pypi.org/project/FlightRadarAPI/`. Retrieved 5th April, 2023.
60. HAT, Red. *What is an API?* 2022. Available also from: `https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces`. Retrieved 5th April, 2023.
61. IBM. *What is an API?* 2023. Available also from: `https://www.ibm.com/topics/api`. Retrieved 5th April, 2023.
62. JEANEXTREME002. *FlightRadar24.* 2023. Available also from: `https://github.com/JeanExtreme002/FlightRadarAPI/tree/main/FlightRadar24`. Retrieved 6th April, 2023.
63. RAYGUN. *SOAP vs REST vs JSON - a 2023 comparison.* 2023. Available also from: `https://raygun.com/blog/soap-vs-rest-vs-json/`. Retrieved 8th April, 2023.

64. RAPID. *What is JSON API? JSON vs GraphQL vs REST API Comparison.* 2023. Available also from: `https://rapidapi.com/blog/api-glossary/what-is-json-api-json-vs-graphql-vs-rest-api-comparison/`. Retrieved 9th April, 2023.
65. PYTHON. *El tutorial de Python* [`https://docs.python.org/es/3/tutorial`]. 2023. Retrieved 22nd December, 2022.
66. DOCS, Hektor. *Introducción a Python* [`https://docs.hektorprofe.net/academia/python/introduccion`]. 2021. Retrieved 10th December, 2022.
67. DOCS, Hektor. *Introducción al módulo analítico NumPy* [`https://docs.hektorprofe.net/academia/python/modulo-analitico-numpy`]. 2022. Retrieved 4th January, 2023.
68. DOCS, Hektor. *Introducción al módulo gráfico Matplotlib* [`https://docs.hektorprofe.net/academia/python/modulo-grafico-matplotlib`]. 2021. Retrieved 14th January, 2023.
69. IAC. *Astropy* [`http://research.iac.es/sieinvens/python-course/astropy.htm`]. 2022. Retrieved 18th January, 2023.
70. IBAN. *COUNTRY CODES ALPHA-2 and ALPHA-3* [`https://www.iban.com/country-codes`]. 2023. Retrieved 18th March, 2023.
71. UPC. *silleo4 new architecture issues and discussion* [`https://arpasat.upc.edu/network-satellite-simulator/silleo4/-/issues/2`]. 2023. Retrieved 28th March, 2023.
72. WOOD, Dr Lloyd. *Introduction to satellite constellations orbital types, uses and related facts* [`https://savi.sourceforge.io/about/lloyd-wood-isu-summer-06-constellations-talk.pdf`]. 2006. Retrieved 4th April, 2023.
73. ALINS, Juanjo. *GML attributes* [`https://arpasat.upc.edu/seamsat/seamsatdocs/-/blob/main/GML_attributes.md`]. 2023. Retrieved 28th May, 2023.
74. WILSON, J. *Haversine* [`$http://jwilson.coe.uga.edu/EMAT6680Fa2013/Lively/Forgotten\%20Trig/The\_Forgotten\_Trigonometric\_Functions.pdf$`]. 2013. Retrieved 20th April, 2023.
75. WHITTY, Robin. *The Spherical Law of Cosines* [`https://www.theoremoftheday.org/GeometryAndTrigonometry/SphericalCos/TotDSphericalCos.pdf`]. 2023. Retrieved 20th April, 2023.