INSTITUTO TECNOLÓGICO DE AERONÁUTICA

Lucas Balen Cardozo

MODELING AND SIMULATION OF THE OPERATION OF PESE SPACE SYSTEMS PROGRAM

Final Paper
2025

Course of Aerospace Engineering

CDU 629.78

Lucas Balen Cardozo

MODELING AND SIMULATION OF THE OPERATION OF PESE SPACE SYSTEMS PROGRAM

Advisor

Maj. Av. Lucas Oliveira BARBACOVI (ITA)

Co-advisor

Prof. Dr. Christopher Shneider Cerqueira (ITA)

AEROSPACE ENGINEERING

São José dos Campos
Instituto Tecnológico de Aeronáutica

2025

Cataloging-in Publication Data
Documentation and Information Division

Cardozo, Lucas Balen
Modeling and Simulation of the Operation of PESE Space Systems program / Lucas Balen Cardozo.
São José dos Campos, 2025.
96p.

Final paper (Undergraduation study) – Course of Aerospace Engineering– Instituto Tecnológico de Aeronáutica, 2025. Advisor: Maj. Av. Lucas Oliveira BARBACOVI. Co-advisor: Prof. Dr. Christopher Shneider Cerqueira.

1. PESE. 2. Modeling. 3. Simulation. I. Instituto Tecnológico de Aeronáutica. II. Modeling and Simulation of the Operation of PESE Space Systems program.

BIBLIOGRAPHIC REFERENCE

CARDOZO, Lucas Balen. **Modeling and Simulation of the Operation of PESE Space Systems program**. 2025. 96p. Final paper (Undergraduation study) – Instituto Tecnológico de Aeronáutica, São José dos Campos.

CESSION OF RIGHTS

Lucas Balen Cardozo
Rua H8B, Ap. 239
12.228-461 – São José dos Campos–SP

MODELING AND SIMULATION OF THE OPERATION OF PESE SPACE SYSTEMS PROGRAM

This publication was accepted like Final Work of Undergraduation Study

---

Lucas Balen Cardozo
Author

Maj. Av. Lucas Oliveira BARBACOVI (ITA)
Advisor

Prof. Dr. Christopher Shneider Cerqueira (ITA)
Co-advisor

São José dos Campos: november 07, 2025.

I dedicate this work to my parents, who built the entire foundation that allowed me to always do my best, and to my friends, who supported me in my most difficult moments. I know that without you I would not even have dreamed of being where I am today.

Acknowledgments

To my parents, for dedicating their lives to making mine the best it could be and giving me the freedom to choose my own path, always supporting my dreams and constantly wishing me the very best.

To my mother, Clarice, for teaching me focus, determination, and perseverance. The path to this point has not been easy, for either of us, but it has taught me the importance of facing every challenge with a smile.

To my father, Rogério, for teaching me calm, patience, and organization. We often do not choose our battles, but even for those we do choose, you have taught me the importance of critical thinking in decision-making.

To my grandmother, Mercedes, who, even though she is no longer with us, remains present in my life through all the love and affection of her memory.

To my grandfather, Egídio, who taught me so much about grit and resilience, the true spirit of a warrior on Earth.

To my dear friend Brian, for his companionship and partnership throughout this entire process. Your care and honesty make me a happier and lighter person. Your advice, though it may shake me, ultimately strengthens me each time. I have great respect and admiration for your journey, and I am very proud to be your friend.

To my long-time friends — Bernardo, Brian, João, Lucas, Guilherme, Bernardo, and Lorenzo. Our friendship inspires me to be a better person, regardless of time and distance.

To my new friends from my undergraduate studies, especially: Antônio, Gabriel, Pedro, Aluísio, Kim, Thiago, Ricardo, João, Campos, Henrique, and Lima. I know you will be part of my life until the end of time.

To Lucas, for persisting, fighting, and facing his fears, even in the face of all adversities.

*"If I have seen farther than others,
it is because I stood on the shoulders of giants."*
— Sir Isaac Newton

Abstract

The Strategic Program of Space Systems (PESE) aims to strengthen Brazilian sovereignty by developing satellite constellations for secure communications and monitoring. This work proposes and validates the development of a computational simulator in Python to model and simulate the integrated operation of these systems. The simulator's architecture separates the intensive physics simulation from the event logic. Orbital dynamics are pre-calculated using the SGP4 analytical propagator, from TLE (Two-Line Element) data, to establish trajectories and determine visibility windows between satellites, ground stations, and regions of interest. The core of the operational simulation is managed by Finite State Machines (FSMs) that govern communication logic and the autonomous execution of activities on the satellite. The simulator implements a registry module and a priority queue (heapq) for telecommands, capable of validating complex prerequisites, such as continuous target visibility, task dependencies, and time delays. Validation was performed using a 24-hour scenario of the SGDC-1 geostationary satellite. The results confirmed the success of the operational cycle: commands were executed in the correct priority order, and all prerequisites were respected. The simulation also revealed an emergent "polling" behavior from the ground station, which proactively re-initiated communication to downlink data generated by completed tasks. This work delivers a flexible, validated simulation platform, parameterized by JSON files, serving as a robust foundation for mission analysis and future developments, such as modeling subsystems (power, data) and integration with the CONCEPTIO laboratory.

Contents

1 Introduction

1.1 Motivation

The interest in developing national capabilities in the space domain has intensified in recent decades due to the growing dependence on satellites for strategic applications such as communications, remote sensing, environmental monitoring, and positioning systems. For Brazil, a country of continental dimensions and a vast territory to be monitored, technological and operational independence in space systems represents a critical factor for national sovereignty and security (Brasil. Ministério da Defesa. Estado-Maior Conjunto das Forças Armadas, 2018). In this context, the Strategic Space Systems Program (PESE) emerges as the official initiative of the Federal Government to consolidate a framework of projects capable of meeting both defense needs and civilian demands, ensuring the availability of dual-use space products under the full control of the Brazilian State (Agência Espacial Brasileira, 2022; Brasil. Ministério da Defesa. Estado-Maior Conjunto das Forças Armadas, 2018).

The primary motivation for the creation of PESE stems from the National Defense Strategy (END), which establishes as a guideline the construction of an integrated monitoring and communication complex that includes launch vehicles, geostationary and low Earth orbit satellites, as well as automated ground stations, in order to provide continuous and accurate surveillance of the country's airspace and continental areas. This guideline is materialized in the creation of the Commission for the Coordination and Implementation of Space Systems (CCISE), responsible for managing PESE and establishing technological and industrial development goals for the national space sector (Brasil. Ministério da Defesa. Estado-Maior Conjunto das Forças Armadas, 2018).

From a strategic standpoint, PESE provides Brazil with significant advantages:

- Operational and Technological Autonomy: Direct control over satellites and launch vehicles reduces dependence on foreign providers, increasing reliability and security in critical defense and monitoring operations (Brasil. Ministério da Defesa. Estado-Maior Conjunto das Forças Armadas, 2018);

- Territorial and Environmental Monitoring: The program foresees constellations of optical and radar remote sensing satellites, which enable continuous monitoring of the continental territory, the Exclusive Economic Zone (EEZ), and sensitive areas such as the “Blue Amazon” (Miranda, 2019);

- Development of the National Space Industry: By prioritizing the use of small platforms (LEO satellites) and fostering offset agreements with technology transfer clauses, PESE promotes the consolidation of a robust production chain, capable of sustaining annual launches and increasing the nationalization rate of systems (Brasil. Ministério da Defesa. Estado-Maior Conjunto das Forças Armadas, 2018);

- Enhancement of the Aerospace Defense System: The Brazilian Aerospace Defense System (SISDABRA), as an integral part of PESE, expands the coverage of detection and interception in the national airspace. Space monitoring will be an integral component and an indispensable condition for fulfilling the strategic tasks of the Brazilian Air Force (FAB), such as multiple surveillance and local air superiority (D’Amato, 2017);

- Economic and Social Applications: In addition to military uses, the satellites developed under PESE provide telecommunications services in remote areas, environmental monitoring, and support to activities related to natural disasters and agribusiness, directly benefiting civil society (Agência Espacial Brasileira, 2022);

- Integration among the Armed Forces: PESE promotes interoperability among the Army, Navy, and Air Force through data sharing and the use of integrated ground stations (COPE and ERDO), which reinforces SISDABRA (Brasil. Ministério da Defesa. Estado-Maior Conjunto das Forças Armadas, 2018).

Furthermore, the simulation of space systems offers substantial advantages for the successful implementation of the program:

- Preliminary Project Validation: Modeling orbits, payloads, and ground stations in a simulated environment makes it possible to identify design flaws before physical construction, reducing technical risks and costs associated with late-stage tests;

- Operational Training and Preparation: Simulations allow mission control teams to train telemetry, tracking, and telecommand procedures under varied scenarios, improving their ability to respond to critical events such as loss of attitude or unexpected orbit changes;

- Optimization of Launch Resources: By studying different constellation configurations and launch windows with orbital perturbation models, it is possible to better plan launch services—such as the use of the Alcântara Launch Center (CLA)—and to minimize fuel consumption and wear on ground stations;

- Acquisition of Operational Metrics for the Program: The determination of metrics such as the average revisit time for each point of interest and the dwell time guides launch planning, the sizing of the number of satellites required, and the definition of contact windows with ground stations, ensuring that surveillance, communications, and remote sensing goals are effectively achieved.

1.2 Hypothesis

This work is based on the premise that it is possible to develop a simulation algorithm that meets the following requirements:

- Study of Orbits and Constellation Patterns: Application of orbital elements for each satellite of the constellations envisioned in PESE, covering LEO, MEO, and GEO orbits, with the use of simplified gravitational perturbation models.
- Constellation Modeling: Representation of all constellations orbiting the Earth simultaneously, with global coverage analysis and cross-checking of visibility data.
- Simulation of Interaction between Satellites and Brazilian Ground Stations: Geolocation of ground stations according to the PESE topology, computation of contact windows, and simulation of active communication.
- Logging and Transmission of Telecommands: Implementation of a routine that records pre-programmed telecommands for each satellite, generates a priority queue for transmission, and automatically sends these commands when the satellite is within the coverage region of a ground station.
- Coupling to the CONCEPTIO Laboratory Environment: Interoperability between the algorithm developed and the existing or planned interfaces and simulation logics of the CONCEPTIO laboratory at ITA.

The central hypothesis of this work is that, by satisfying each of these technological requirements, it will be possible to simulate, albeit in a simplified manner, the operability of the mission proposed by PESE at low cost, allowing this work to be used for early project validation, operational training, and performance analysis of Brazilian space missions. It is also assumed that the Python programming language, combined with strategic libraries, is capable of providing adequate computational performance for near real-time simulations without the need for supercomputing infrastructure.

1.3 Objectives

Given the scope of PESE and the desired functionalities, the general and specific objectives of this work are described below.

1.3.1 General Objective

To develop and validate a computational simulator that enables the integrated modeling and simulation of the operation of the space systems envisioned in PESE, encompassing orbits, constellations, ground stations, telecommands, and orbital perturbations, and integrating the simulation into the environment of the CONCEPTIO laboratory at ITA.

1.3.2 Specific Objectives

- Requirements Elicitation for PESE: To analyze in detail the official PESE document and other references (PNDAE, END, PNAE) to identify orbital parameters, constellation topology, and operational specifications for each satellite and ground station.

- Implementation of the Orbital Dynamics Module: To create structures in a programming environment to represent Keplerian orbital elements, convert classical parameters into Cartesian states, and include simplified perturbation models to update orbital elements in discrete time steps.

- Constellation Modeling: To define the constellations required by PESE, assigning initial parameters of inclination, altitude, and temporal reference, so as to simulate all constellations simultaneously around the Earth and generate global coverage plots.

- Geolocation and Coverage Calculation for Ground Stations: To collect geographic coordinates of the COPE, ERDO, and COPE-S stations, implement a visibility-ellipse algorithm, and define communication windows for each satellite–station pair.

- Telecommand Logging and Queue Module: To develop a data structure for the registration of standardized telecommands, priority logic, and an automatic triggering routine based on visibility prediction.

- Integration with the CONCEPTIO Laboratory: To assess the architecture of the CONCEPTIO laboratory environment at ITA, identify integration points, and implement interfaces for reading and writing files containing orbital parameters and satellite states.

- Validation and Performance Testing: To design test cases involving representative scenarios of PESE missions and evaluate the accuracy of orbital positions and communication latency.

These specific objectives constitute the necessary steps to achieve the general objective, ensuring that the simulator covers all required functions and is validated within acceptable standards of accuracy and usability.

2 Literature Review

2.1 Orbits

2.1.1 Basic General Formulation

The study of orbits in celestial mechanics is based on Newton's Law of Universal Gravitation, which states that two point masses m_1 and m_2, separated by a distance r, attract each other with a force given by

$$F = G\,\frac{m_1\,m_2}{r^2}, \tag{2.1}$$

where G is the universal gravitational constant ($G \approx 6{,}67430 \times 10^{-11}\,\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-2}$) (Curtis, 2020; Kluever, 2018). From this relation, the standard gravitational parameter of a central body (in this case, the Earth) is defined as

$$\mu = G\,M, \tag{2.2}$$

in which M is the mass of the central body, which in this case is the Earth (Curtis, 2020). In the context of orbital dynamics, the satellite is approximated as a particle of mass $m \ll M$, resulting in the unperturbed two-body problem.

The equation of motion of the satellite is given by

$$\ddot{\vec{r}} = -\mu\,\frac{\vec{r}}{r^3}, \tag{2.3}$$

where $\vec{r}$ is the position vector of the satellite with respect to the Earth's center and $r = \|\vec{r}\|$ (Montenbruck; Gill, 2012).

From equation (2.3), it is observed that the specific angular momentum of the satellite, defined as

$$\vec{h} = \vec{r} \times \dot{\vec{r}}, \tag{2.4}$$

is a conserved quantity in the unperturbed two-body problem. Its magnitude $h = \|\vec{h}\|$ is directly related to the semi-major axis and eccentricity of the orbit.

The mechanical energy per unit mass of the satellite is given by

$$\varepsilon = \frac{1}{2}\dot{\vec{r}} \cdot \dot{\vec{r}} - \frac{\mu}{r}. \tag{2.5}$$

In the elliptical orbit regime, one has $e < 1$ and $\varepsilon < 0$. In the parabolic orbit regime, $e = 1$) and $\varepsilon = 0$. In a hyperbolic orbit, $e > 1$ and $\varepsilon > 0$ (Curtis, 2020; Kluever, 2018).

The relation frequently used to obtain the velocity $\|\dot{\vec{r}}\|$ as a function of the distance r and the semi-major axis a is the equation:

$$v^2 = \dot{\vec{r}} \cdot \dot{\vec{r}} = \mu \left(\frac{2}{r} - \frac{1}{a} \right). \tag{2.6}$$

This expression follows directly from the conservation of energy shown in (2.5).

2.1.2 Orbital Elements

The orbital elements are a set of parameters that uniquely characterize the shape, orientation, and position of a satellite along its trajectory (Curtis, 2020). The six classical elements that describe an unperturbed Keplerian orbit are presented below:

1. Semi-major axis (a): defines the size of the orbit. For elliptical orbits, $a > 0$; for parabolic orbits, $a \to +\infty$; for hyperbolic orbits, $a < 0$.

2. Eccentricity (e): measures the "stretching" of the orbit relative to the circular shape. For elliptical orbits, $0 \leq e < 1$; for parabolas, $e = 1$; for hyperbolas, $e > 1$. The eccentricity vector $\vec{e}$ is defined by equation 2.7.

$$\vec{e} = \frac{1}{\mu} \left(\dot{\vec{r}} \times \vec{h} \right) - \frac{\vec{r}}{r} \tag{2.7}$$

   In this case, one has e in the form $e = \|\vec{e}\|$.

3. Inclination (i): angle between the orbital plane and the Earth's equatorial plane. $0 \leq i \leq 180°$.

$$i = \cos^{-1} \left(\frac{h_z}{h} \right) \tag{2.8}$$

   Here, h_z is the z-component of $\vec{h}$.

4. Right ascension of the ascending node (Ω): angle measured in the equatorial plane, from the x-axis of the geocentric–equatorial frame to the line of nodes (intersection

between the orbital and equatorial planes), in the counterclockwise direction.

$$\Omega = \begin{cases} \cos^{-1}\left(\dfrac{N_x}{N}\right), & N_y \geq 0, \\ 2\pi - \cos^{-1}\left(\dfrac{N_x}{N}\right), & N_y < 0, \end{cases} \tag{2.9}$$

In this case, $\vec{N} = (N_x, N_y, 0) = \vec{k} \times \vec{h}$ is the node vector and $N = \|\vec{N}\|$.

5. Argument of perigee (ω): angle in the orbital plane, from the line of nodes to the perigee point (point of minimum distance).

$$\omega = \begin{cases} \cos^{-1}\left(\dfrac{\vec{N} \cdot \vec{e}}{N\, e}\right), & e_z \geq 0, \\ 2\pi - \cos^{-1}\left(\dfrac{\vec{N} \cdot \vec{e}}{N\, e}\right), & e_z < 0. \end{cases} \tag{2.10}$$

6. True anomaly (ν): angle in the orbital plane, from perigee to the current position of the satellite. Given the position vector $\vec{r}$ and the vector $\vec{e}$, the true anomaly is defined by equation 2.11.

$$\nu = \begin{cases} \cos^{-1}\left(\dfrac{\vec{e} \cdot \vec{r}}{e\, r}\right), & \vec{r} \cdot \dot{\vec{r}} \geq 0, \\ 2\pi - \cos^{-1}\left(\dfrac{\vec{e} \cdot \vec{r}}{e\, r}\right), & \vec{r} \cdot \dot{\vec{r}} < 0. \end{cases} \tag{2.11}$$

Formulation of the orbital elements: the orbit, in the perifocal frame ($\{\vec{p}, \vec{q}, \vec{w}\}$), is described by:

$$r = \frac{a\,(1 - e^2)}{1 + e\, \cos\nu} \tag{2.12}$$

$$\vec{r}_{\text{pf}} = r \begin{bmatrix} \cos\nu \\ \sin\nu \\ 0 \end{bmatrix}, \quad \dot{\vec{r}}_{\text{pf}} = \sqrt{\frac{\mu}{p}} \begin{bmatrix} -\sin\nu \\ e + \cos\nu \\ 0 \end{bmatrix}, \tag{2.13}$$

where $p = a\,(1 - e^2)$ is the semi-latus rectum. The local orbital velocity is

$$v = \sqrt{\mu\left(\frac{2}{r} - \frac{1}{a}\right)} \tag{2.14}$$

2.2 Coordinate System Transformations

To move from the perifocal frame $\{\hat{p}, \hat{q}, \hat{w}\}$ to the geocentric–equatorial frame $\{\hat{i}, \hat{j}, \hat{k}\}$, the following rotation matrix is used:

$$\vec{Q}_{\text{PE}} = R_3(-\Omega)\, R_1(-i)\, R_3(-\omega) \tag{2.15}$$

where

$$R_3(\alpha) = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad R_1(\beta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\beta & \sin\beta \\ 0 & -\sin\beta & \cos\beta \end{bmatrix}$$

(Curtis, 2020). Thus, the position and velocity in the geocentric–equatorial frame are

$$\vec{r}_{\text{GE}} = \vec{Q}_{\text{PE}}\, \vec{r}_{\text{pf}}, \qquad \dot{\vec{r}}_{\text{GE}} = \vec{Q}_{\text{PE}}\, \dot{\vec{r}}_{\text{pf}}. \tag{2.16}$$

In the perifocal frame:

- $\hat{p}$ points from the focus (Earth) to perigee;
- $\hat{q}$ lies in the orbital plane, orthogonal to $\vec{\hat{p}}$ in the direction of orbital motion;
- $\hat{w}$ coincides with the angular momentum vector $\vec{h}$.

2.3 Remote Sensing

Remote sensing is the technique of acquiring information about an object or phenomenon without direct physical contact, using sensors onboard satellites or airborne platforms (Campbell, 2002; Lillesand; Kiefer; Chipman, 2015). These sensors detect electromagnetic radiation reflected or emitted by the Earth's surface, enabling environmental analyses, land use and land cover mapping, natural resource monitoring, and applications in defense and security.

In the context of PESE, remote sensing is a fundamental pillar, divided into two main mission families: optical sensing (CARPONIS) and radar sensing (LESSONIA), both designed to operate in Low Earth Orbit (LEO) and to provide vital data for National Defense (Martins, 2021).

A remote sensing satellite is equipped with optical or microwave instruments capable of recording data in multiple bands of the electromagnetic spectrum. It is characterized by the presence of:

- Optics with high spectral and spatial resolution;
- Data collection and storage systems;
- Transmission antennas to send information to the ground station;
- Attitude control systems for precise sensor pointing;
- Power supply (solar panels and batteries) and thermal systems for satellite stabilization (Lillesand; Kiefer; Chipman, 2015).

2.3.1 Optical Sensing

Optical satellites capture solar radiation reflected by the Earth's surface. They use detectors sensitive to wavelengths in the visible spectrum, approximately in the interval 0.4–0.7 μm), and in the near-infrared, approximately in the interval 0.7–1.1 μm), recording reflectance levels that vary according to land cover (water, vegetation, bare soil, etc.) (Lillesand; Kiefer; Chipman, 2015).

This capability is the focus of the CARPONIS family of PESE. It consists of a set of high-resolution Optical Remote Sensing (SRO) satellites (Martins, 2021). The objective of the Carponis-1 satellite is to provide submetric and color imagery (Martins, 2021), enabling detailed identification of targets to support intelligence in military operations and civil inspection. Although Brazil already cooperates in the CBERS program, CARPONIS aims at a sovereign capability with superior resolution.

For optical sensing, the following are required:

- Focusing lenses or mirrors: direct light to the detectors;
- CCD/CMOS detectors: convert photons into electrical signals;
- Spectral filters: isolate specific spectral bands;
- Attitude stabilization systems: ensure stable sensor pointing, minimizing blur effects;
- Data processor: for digitization, compression, and temporary storage;
- Telecommunication antenna: for transmitting images to the ground station (Campbell, 2002).

2.3.2 Radar Sensing (SAR)

Synthetic Aperture Radar (SAR) emits microwave pulses, typically in the 1–10 GHz range, directed toward the Earth's surface. The reflected signal returns to the satellite, where it is captured by the receiver. The SAR image is formed from the phase and amplitude differences of the returned echoes, allowing the generation of images that are independent of solar illumination and weather conditions (Campbell, 2002).

Within PESE, this mission is carried out by the LESSONIA family, a fleet of Radar Remote Sensing (SRR) satellites (Martins, 2021). The Lessonia-1 system, composed of the satellites Carcará I and II, is already operational. The main tactical advantage of SAR is the ability to provide 24/7 surveillance, regardless of meteorological conditions (such as clouds or fog) or solar illumination, which is crucial for continuous monitoring of borders and the Amazon (Dual Use Defense/CENSIPAM) (Martins, 2021).

The basic equation for received power in a SAR system can be expressed as:

$$P_r = \frac{P_t \, G_t \, G_r \, \lambda^2 \, \sigma}{(4\pi)^3 \, R^4} \,, \tag{2.17}$$

where:

P_r: power received by the radar;

P_t: transmitted power;

G_t, G_r: gains of the transmitting and receiving antennas;

λ: carrier wavelength;

σ: radar cross section of the observed area;

R: distance between the satellite and the target (Campbell, 2002).

For radar sensing, the following are required:

- High-power radio-frequency transmitter: generates microwave pulses;
- Synthetic aperture antenna (SAR): performs azimuth scanning to synthesize a larger antenna;
- Radio-frequency receiver: captures and converts the returned echo into digital signals;
- Onboard processor: performs correlation of the received signal for image formation;
- Attitude and orbit control systems: ensure appropriate orientation and stable altitude for SAR, since the acquisition geometry directly affects resolution.

2.4 Communication

Communication satellites have as their main objective the retransmission of radio, television, telephony, Internet data, and other microwave services between different regions of the Earth's surface. These satellites act as active repeaters, receiving signals from a transmitting station, amplifying them, and retransmitting them to distant locations (Maral; Bousquet, 2011; Pratt; Bostian; Allnutt, 2003).

PESE addresses sovereign communications through a "system of systems" architecture with two distinct families that complement each other at the strategic and tactical levels:

- CALIDRIS: Focused on Strategic Communications from Geostationary Orbit (GEO) (Martins, 2021).
- ATTICORA: Planned for Tactical Communications in Low Earth Orbit (LEO) (Martins, 2021).

The essential characteristics of a communication satellite include:

- Transponders: sets of receivers, power amplifiers, and transmitters operating in specific frequency bands (C, Ku, Ka, L, S, X, etc.);
- High-precision antennas: typically fixed-beam antenna arrays to cover large geographic areas or spot beams;
- Frequency conversion systems: for translation from uplink (ground station to satellite) to downlink (satellite to ground station);
- Power supply: solar panels sized to support the power of the transponders, as well as batteries for continuity during eclipses;
- Attitude and orbit control system: ensures precise antenna pointing for geostationary orbit or another appropriate orbit (Maral; Bousquet, 2011).

A typical communication satellite mission includes:

1. Occupation of geostationary orbit (GEO) at about 35 786 km altitude, maintaining a fixed longitude position;
2. Reception of signals in the uplink and retransmission in the downlink, with amplification and filtering;
3. Distribution of signals to multiple ground stations, satellite TV operators, Internet providers via VSAT (*Very Small Aperture Terminal*), and mobile telephony systems;

4. Power budget management to ensure link quality even under adverse conditions (rain, interference) (Maral; Bousquet, 2011).

2.4.1 Forms of Communication

Satellite communications can be classified, among other criteria, by the bandwidth used in the radio link. The distinction between narrowband and broadband is fundamental for the selection of countless design parameters, such as data rate, modulation complexity, and hardware requirements. Narrowband is usually employed for telemetry, telecommand, and payloads with low data volume, whereas broadband supports high data-rate demands, such as image transmission and Internet access via satellite (Maral; Bousquet, 2011).

In narrowband systems, typical bandwidths range from a few tens of kilohertz up to, at most, a few hundreds of kilohertz. This reduced spectrum imposes limitations on the data rate according to the theoretical capacity given by Shannon's formula:

$$C = B\ \log_2\!\left(1 + \tfrac{S}{N}\right) \tag{2.18}$$

where C is the capacity in bits per second, B the bandwidth in hertz, and S/N the signal-to-noise ratio (SHANNON, 1948). In practice, narrowband systems in small satellites (for example, CubeSats in VHF/UHF) operate with $B \approx 25$–$100\,\mathrm{kHz}$ and achieve data rates of up to a few hundred kilobits per second, being suitable for platform telemetry and remote command (Pratt; Bostian; Allnutt, 2003).

This type of tactical communication corresponds to the mission of the ATTICORA family of PESE. As a low Earth orbit (LEO) constellation, its purpose is to provide voice communication and data collection in remote regions. The main advantage is to enable the use of "light and small terminals" by tactical units in the field, such as reconnaissance teams, which could not operate with the large GEO terminals (Martins, 2021).

In contrast, broadband communications exploit frequency ranges from the megahertz spectrum, such as L-band between 1–2 GHz and S-band between 2–4 GHz, up to multi-gigahertz bands such as Ku-band between 12–18 GHz and Ka-band between 26–40 GHz, with bandwidths of up to hundreds of megahertz per channel. These ranges support data rates of several gigabits per second, which are essential for the transmission of high-resolution imagery and broadband services for end users (RICHARIA, 2012).

This broadband, high-capacity profile defines the CALIDRIS family, represented by the Geostationary Defense and Strategic Communications Satellite (SGDC-1) (Martins, 2021). Operating in GEO, SGDC-1 provides dual-use strategic communications: it uses X-band, exclusively and securely for the Armed Forces, and Ka-band, of high capacity, for the civil National Broadband Program (PNBL) (Martins, 2021).

To maintain link quality over orbital distances, the free-space path loss model is applied:

$$\text{FSPL(dB)} = 20\,\log_{10}\!\left(\frac{4\pi d}{\lambda}\right) \tag{2.19}$$

where d is the satellite–Earth distance and λ the wavelength (SKOLNIK, 2008).

Regarding onboard devices, narrowband systems require low-complexity transceivers, consisting of band-pass filters with restricted bandwidth, low-noise amplifiers (LNA), modulators/demodulators with simple modulation schemes (for example, BPSK), and moderately directional antennas (helical or patch) (Maral; Bousquet, 2011). In broadband, satellites must incorporate high-capacity transponders with solid-state power amplifiers (SSPA) or traveling wave tube amplifiers (TWTA), wideband filters, modems with high-order modulation (QPSK, 16-QAM, or higher), and large reflector antennas or phased-array antennas capable of forming narrow, steerable beams (RICHARIA, 2012).

2.4.2 Two-Line Elements (TLE) and the SGP4 Model

Whereas the classical Keplerian orbital elements, described in Section 2.1.2, provide an ideal geometric description of an unperturbed orbit, the propagation of real satellites requires the inclusion of significant perturbations, such as the Earth's polar flattening (J2, J3, J4) and atmospheric drag. For this purpose, the most universally adopted data format for Earth-orbiting objects (LEO and GEO) is the TLE (*Two-Line Element*).

A TLE is not merely a data format; it is a set of input parameters inseparably linked to a specific family of analytical propagators, notably SGP4 (*Simplified General Perturbations 4*) and its derivatives (e.g., SGP, SGP8, SDP4, SDP8). SGP4 is an analytical propagator (and not a numerical one), which makes it computationally very fast. It was designed to include the secular and periodic effects of the main perturbations, allowing reasonably accurate trajectory predictions over several days.

The data in a TLE are not osculating orbital elements (i.e., the instantaneous geometric state of the satellite), but rather **mean elements**. These elements are the result of a complex fitting process of observational data (radar, optical) to an SGP4 model. In short, the mean elements of a TLE are those values which, when supplied to the SGP4 algorithm, produce a trajectory that best approximates the real observed trajectory of the object.

The TLE format consists of two lines of 69 characters each, in plain text, preceded by a title line. The structure, exemplified by the TLE of the SGDC-1 satellite used in this simulation, is as follows:

```
SGDC-1
```

```
1 42692U 17023B   25294.12501553 -.00000244  00000+0  00000+0 0  9999
2 42692   0.0439  67.7207 0003191 159.5177 132.7639  1.00272242 31015
```

The most relevant fields for propagation are:

- Line 1: Contains the TLE epoch (year and fraction of the day, e.g., `25294.12...`), which serves as the time t_0 of the element set, and the drag coefficient B* (a term derived from the ballistic coefficient), which models the effect of atmospheric drag.
- Line 2: Contains the six mean orbital elements required by SGP4: Inclination (i), Right Ascension of the Ascending Node (Ω), Eccentricity (e), Argument of Perigee (ω), Mean Anomaly (M), and Mean Motion (n, in revolutions per day).

In this work, orbital propagation is not performed by numerical integration of the classical equations of motion (Equation 2.3), but rather by the direct application of the SGP4 model, as implemented in the Python library `sgp4`. When instantiated, the `Satellite` class uses the provided TLE to initialize a `Satrec` object (via `Satrec.twoline2rv`). This object is the SGP4 propagator.

Subsequently, in the physical simulation phase (Section 3.2.3), the `propagate` method calls this object at each discrete time instant (via `satrec.sgp4`), obtaining the state vector (position $\vec{r}$ and velocity $\vec{v}$) of the satellite in the TEME (*True Equator Mean Equinox*) inertial frame. This state vector is the "truth" of the system at that instant, and serves as the basis for all subsequent geometric calculations, such as the verification of ground-station visibility and regions of interest.

3 Methodology

3.1 Materials

The development of the modeling and simulation algorithm will be carried out in the Python programming language (Foundation, 2020). Python is a high-level, interpreted, dynamically typed language, widely employed in scientific and engineering projects due to its clear syntax and large support community. Its native constructs, such as class definitions and function creation, allow for the modular structuring of entities in the simulation domain, such as satellites, ground stations, and orbital perturbation modules.

For orbital modeling and simulation in the scope of this work, the Poliastro library is employed, developed to provide interactive and programmatic orbital mechanics functionalities (León; Alarcón; Soto; Zuluaga, 2018). The main objective of its use lies in the ease of defining Keplerian elements, propagating trajectories, and applying maneuvers and perturbations through dedicated classes and methods. Among the fundamental classes, *Orbit*, responsible for representing orbits in the central-body frame, and *Maneuver*, which allows the definition of sequences of orbital impulses, stand out. For orbit creation, the *Orbit.from_classical* method makes it possible to instantiate an orbit object from the semi-major axis, eccentricity, inclination, right ascension of the ascending node, argument of perigee, and true anomaly. Orbit propagation can be performed using analytical and numerical integrators, accessible via the *orbit.propagate* method, which admits specification of propagation time and integrator type. The application of perturbations, such as the effect of polar flattening (J2), is enabled by defining perturbation elements in the numerical propagator, allowing a more accurate modeling of the real trajectories of satellites.

For the definition of the PESE satellite constellations, the library will be used in the programmatic generation of multiple orbits, repeating the structure of orbital parameters in successive planes according to the desired Earth coverage patterns. Replication of the *Orbit* object around the Earth will allow revisits and coverage times to be mapped, which are fundamental for simulating telecommands to be sent when the satellites are within the line-of-sight region of Brazilian ground stations.

Complementing Poliastro's functionalities, the Astropy library provides full support for the handling of physical units and the definition of coordinate systems, enabling consistent conversion of orbital parameters between different reference frames without the risk of scale errors (Astropy Collaboration, 2013). Together with this, NumPy offers a structure of multidimensional arrays and high-performance vector operations, facilitating batch execution of orbital calculations and the simultaneous evaluation of multiple constellation scenarios (Harris *et al.*, 2020). For numerical propagation and the treatment of differential equations that model perturbations such as the J2 effect, SciPy provides advanced integration routines and ODE solvers that are proven in scientific applications. Finally, Matplotlib enables the generation of two- and three-dimensional plots of orbital trajectories, as well as the plotting of coverage and revisit curves, supporting qualitative and quantitative analysis of the results (Hunter, 2007).

The pandas library plays a fundamental role in the handling and analysis of tabular data generated during PESE simulations, allowing records of positions, velocities, and coverage events to be organized into high-performance DataFrame structures (McKinney, 2018). With it, it is possible to import and export data in various formats (CSV, HDF5, Parquet), perform filtering, grouping, and aggregation operations to extract operational metrics such as the distribution of revisit times and dwell durations per ground station. In addition, advanced join (merge) and temporal resampling functionalities facilitate the combination of data from different constellations and the generation of standardized time series for subsequent statistical analysis. The integration of pandas with NumPy and Matplotlib ensures a cohesive workflow, in which DataFrames directly feed plotting functions, resulting in visual reports and summary tables that support the evaluation of the simulation results.

In practice, the simulation begins by defining Poliastro Orbit objects with classical elements, while Astropy ensures that all values are in consistent units, simplifying the composition of state vectors and transformations between angles and distances. Next, NumPy is employed to organize vectors of time instants and parameters of different orbital planes, allowing the creation of input matrices to be propagated in parallel. SciPy's integration functions are then called to compute the temporal evolution of positions and velocities, including additional perturbations, through programmatic interfaces that accept custom acceleration functions. Finally, Matplotlib will be used to generate detailed visualizations of orbits and coverage maps, integrating trajectory lines, geographic projections, and line-of-sight indicators for Brazilian ground stations, which facilitates the evaluation of PESE's operational metrics, such as revisit times and coverage durations.

3.2 Simulation

The simulator architecture was designed based on a separation-of-concerns philosophy, dividing the logic into distinct phases that will be detailed in the following subsections. The backbone of this architecture is the set of data structures used to define scenarios, manage the state of agents (satellite and stations), and record results.

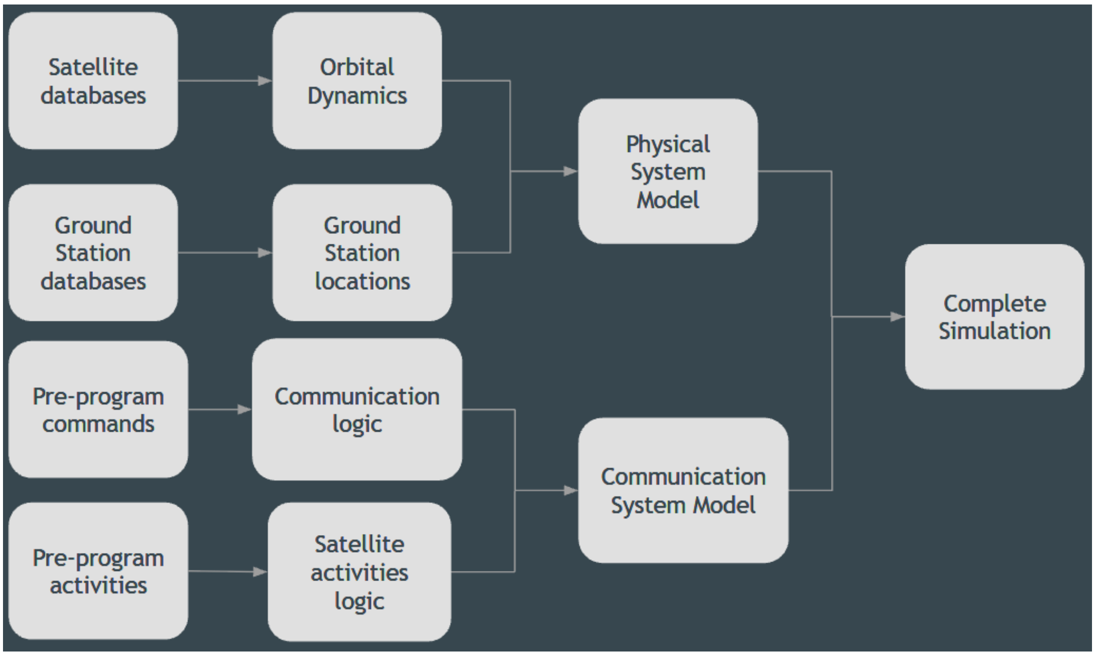

FIGURE 3.1 – Diagram describing the methodology adopted for the simulation.

3.2.1 Data Structures

The flow of information in the simulation is managed by two main categories of data structures: static configuration files, which define the scenario input parameters; and dynamic runtime objects, which maintain the state of the simulation and execute the operational logic.

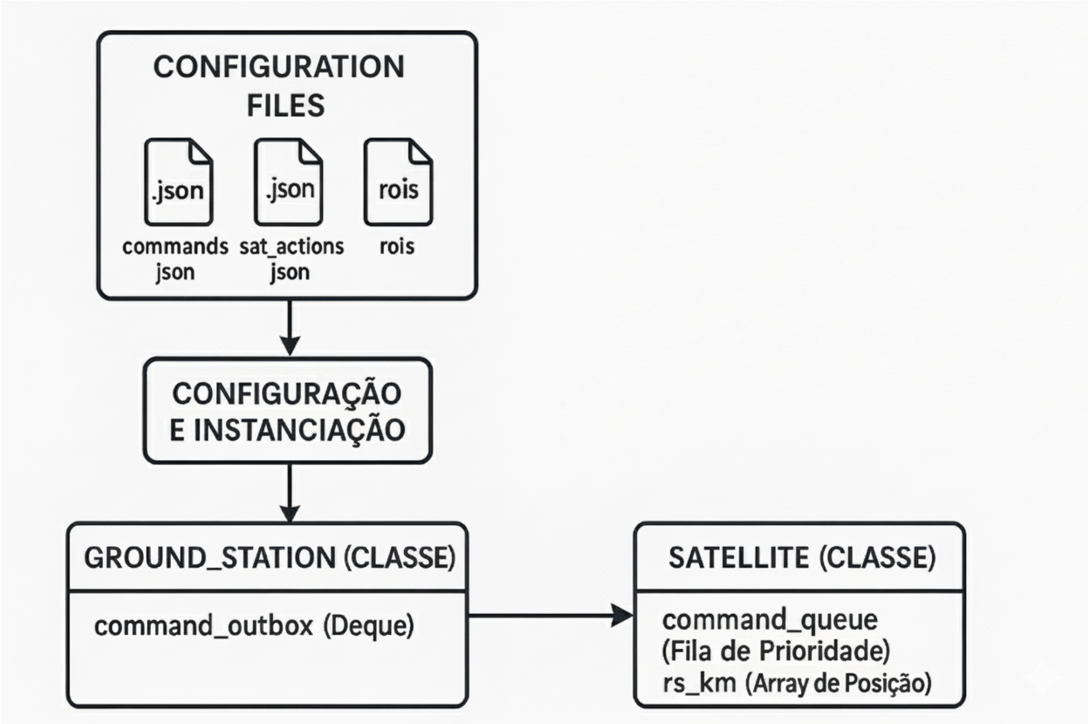

FIGURE 3.2 – Diagram of the main data structures and their interaction.

The simulation is parameterized by a set of configuration files in JSON format, allowing complex scenarios to be defined without changing the source code. The commands.json file serves as the initial mission plan, detailing the sequence of telecommands (TCs) that each ground station will attempt to transmit to the satellite.

Each command in this file references an id_atividade, whose fundamental properties, notably duracao_seg (duration in seconds), are cataloged in the sat_actions.json file. The commands also specify their execution prerequisites, such as visibilidade_area_alvo. The target areas (ROIs), in turn, are defined as geographic polygons in the rois.json file.

Finally, the timing of the communication protocol (duration of *beacon*, *uplink*, etc.) is governed by the parameters in comm_config.json.

At runtime, the static configuration parameters are deserialized to instantiate the agent classes that encapsulate the state and behavior of the system. The Satellite class is the central agent, storing the SGP4 propagator and, after the pre-computation phase, the complete trajectory results in the rs_km and vs_kms vectors. Critically, this class manages the command_queue, a priority queue (*min-heap* via *heapq*) that stores received TCs and ensures that the highest-priority activities (lowest numerical value) are evaluated first. The class also tracks the timing of key events, such as the receipt of TCs, in the event_timestamps dictionary.

In parallel, the Ground_Station class models the ground stations, storing their fixed geographic location and their TC queue for transmission, command_outbox, implemented as a *deque* (double-ended queue) for efficient loading and sending operations.

Finally, the essential bridge between physical simulation and event simulation is established by precomputed visibility vectors. These structures, typically boolean NumPy *arrays*, store the result of geometric calculations for each time step, allowing the main event loop to perform instantaneous state checks without the need for repeated orbital computations.

3.2.2 Configuration and Instantiation

The configuration and instantiation phase is the initialization process executed by the main module, main.py, before the start of the simulation loop. The first step is defining the simulation "clock." The epoch time, t0, is extracted directly from the satellite TLE (Two-Line Element), serving as the temporal zero mark. From this t0, a simulation horizon is defined (the variable periodo_ref, set to 24 hours) and discretized with a fixed time step (60 seconds). The result is the master time vector, a NumPy array named times, which dictates each instant at which the system state will subsequently be computed and evaluated.

Concurrently with the time definition, the system agents are instantiated. The Satellite class is used to create the sgdc1 object, which is initialized with the TLE lines and the sensor opening angle. During its initialization, the Satellite class stores the TLE and immediately uses the SGP4 library to create the Satrec propagator instance. This propagator is retained but not executed at this stage; propagation of the complete trajectory is a subsequent pre-computation step. Similarly, the ground stations are instantiated from the Ground_Station class (cope_station, cope_s_station). The Ground_Station constructor receives the geodetic parameters (latitude, longitude, altitude) and the line-of-sight cone opening angle, using them to create and store an EarthLocation object from the AstroPy library. This object will be essential for visibility calculations.

Once the agents are instantiated, their mission data are loaded. The main.py file reads and deserializes all JSON configuration files, such as rois.json, commands.json, sat_actions.json, and comm_config.json. The "data injection" step occurs when the load_commands method of each Ground_Station instance is invoked. This method filters the command dictionary (originating from commands.json) searching for a key that matches the name of the station itself ("COPE"). The commands found are then enqueued into the station's internal data structure, command_outbox (a deque), preparing them for future uplink. The data from the other configuration files, such as sat_actions_config and comm_config, are kept in the main scope to be injected into the state update functions at each simulation step. At the end of this phase, the system is fully configured and ready for the geometric pre-computation phase.

3.2.3 Physical Simulation

A fundamental architectural decision in this simulation is the complete separation between orbital propagation, which is computationally intensive, and event simulation. All mission geometry is precomputed in this phase, before the main event loop is started. This phase transforms the static input parameters into full time series of position and visibility for the entire simulation horizon.

The process begins with orbit propagation. The main.py module invokes the propagate method of the Satellite object, passing the master time vector (times) as an argument. Internally, this method iterates over each time instant and, using the Satrec library, computes the state vector (position $\vec{r}$ and velocity $\vec{v}$) in the TEME (True Equator Mean Equinox) inertial frame for that specific instant. It is important to note that the SGP4 model inherently includes the main orbital perturbations, such as the effects of polar flattening (J2) and atmospheric drag. The resulting state vectors for the entire simulation are stored directly in the Satellite object, in the rs_km and vs_kms arrays.

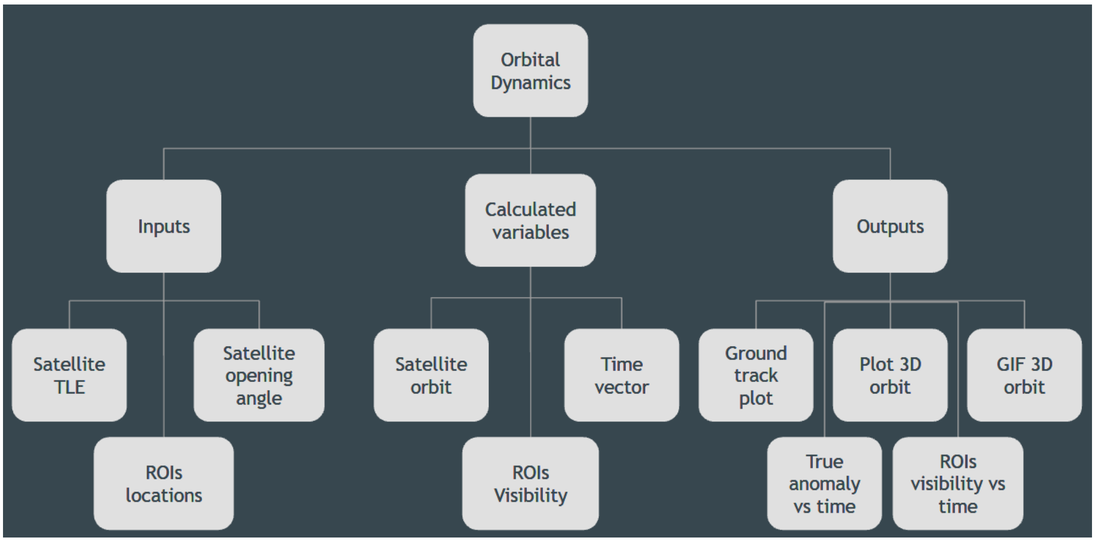

FIGURE 3.3 – Diagram describing the methodology for the physical model of orbital dynamics.

Once the complete trajectory is available, the next step is to compute visibility events. This calculation is delegated to the agents themselves. The main.py file invokes the calculate_visibility method of each Ground_Station instance. This method uses the physical parameters of the station: its geodetic location (an EarthLocation object from the AstroPy library) and its opening angle (opening_deg), which defines the line-of-sight cone from the zenith (e.g., an angle of 40° requires a minimum elevation of 50°). The method iterates over the entire rs_km trajectory, converting the satellite TEME position, at each step, to the station's local frame (Altitude/Azimuth). By comparing the computed elevation with the visibility threshold, the method generates a boolean vector (e.g., cope_visible) that maps "True" or "False" for each instant of the simulation.

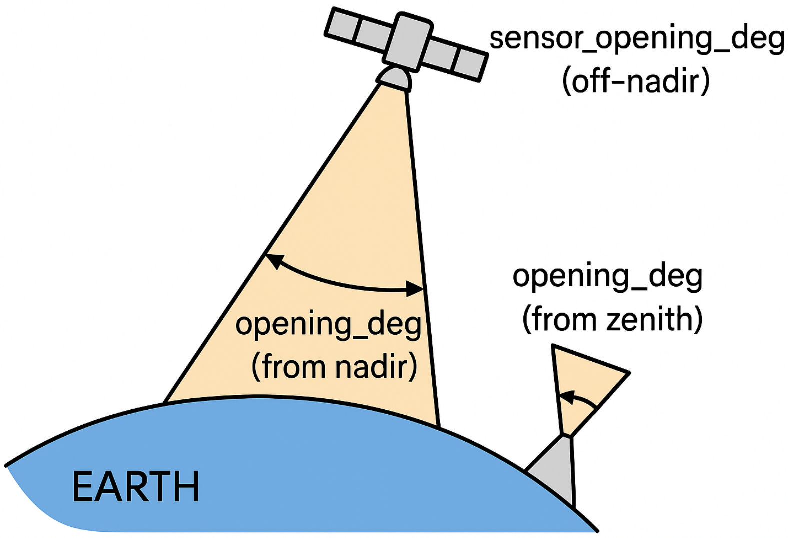

FIGURE 3.4 – Line-of-sight geometry for the satellite sensor and the ground station.

An analogous process is carried out for the regions of interest (ROIs). The main.py file invokes the check_polygon_visibility method of the Satellite object. This method uses the satellite sensor half-opening angle (sensor_opening_deg) and the polygon coordinates (read from rois.json). For each time step, the function computes the satellite subpoint (nadir) and the angular radius of its footprint on the surface, using the spherical trigonometry function sensor_ground_range. It then checks whether all vertices of the ROI polygon are contained within this footprint. The result is another boolean vector that defines the imaging opportunity windows.

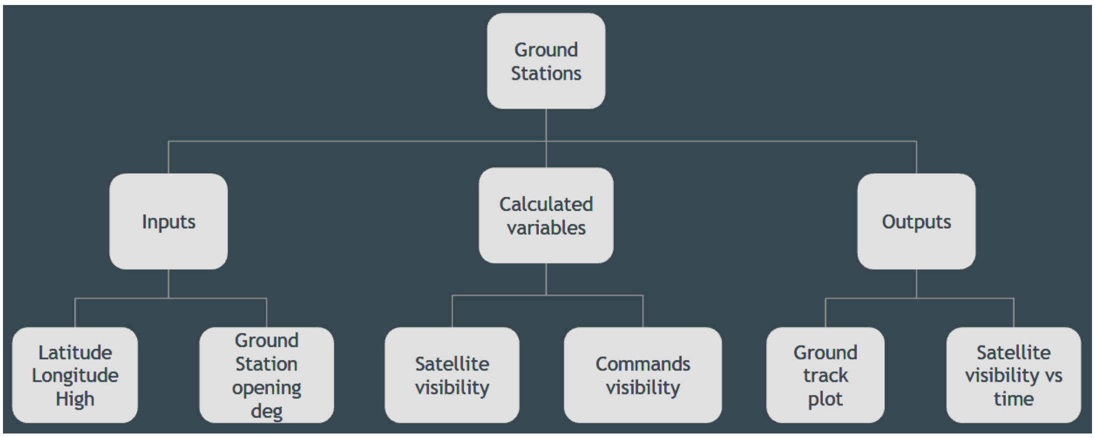

FIGURE 3.5 – Diagram describing the methodology for the physical model of the ground stations.

The visual consolidation of this phase is generated by the ground_track_plot function. This function processes the complete trajectory, converting the TEME positions to geodetic coordinates (latitude, longitude) and plotting them on a 2D world map using the Cartopy library. The resulting plot (saved as ground_track_plot.png) displays the satellite trajectory, the sensor coverage swath, and the ground-station visibility cones, providing a complete overview of the mission geometry.

3.2.4 Communication Simulation and Event Logic

After the physical simulation phase, the simulation enters its main event loop, iterating step by step over the entire time horizon. In this phase, the boolean visibility vectors act as triggers that activate the communication logic. Communication is not modeled as a continuous flow, but as a discrete protocol managed by Finite State Machines (FSMs), both on the satellite and on the ground stations.

The communication channel is simulated through *buffer* variables in the scope of the main loop. At each time step, the agents (satellite and stations) read and write to these *buffers*, simulating the exchange of data packets. The logic of which agent can speak or listen is governed by their state update methods.

When visibility between the satellite and a station is established, the ground station initiates the protocol. It transitions from the IDLE state to WAITING_BEACON, indicating that it is awaiting the first communication. The satellite, upon detecting this state in the station, responds by transitioning to DOWNLINKING_BEACON, sending a health packet and starting a timer. The duration of this and all communication stages is defined by the comm_config.json configuration file.

After the *beacon*, the satellite transitions to the UPLINKING state, signaling that it is ready to receive commands. The ground station, upon detecting this change, transitions to its own UPLINKING state, starting the duracao_uplink_comandos_seg timer. At the end of this timer, the station aggregates all pending commands from its command_outbox queue, loaded from commands.json, and places them in the transmission *buffer*, emptying its local queue. The station then switches to the DOWNLINKING state, awaiting confirmation telemetry.

The satellite, upon receiving the command packet in the *buffer*, performs one of the most critical actions in the simulation: it iterates over the received commands and inserts them into its internal priority queue, self.command_queue (a *min-heap*). At the same time, it records the arrival time of each command, which is crucial for checking delay-based prerequisites. The satellite then transitions to DOWNLINKING_DATA, where it sends its telemetry packets, including the queue state and the output files generated by completed activities, before returning to the IDLE state. The station, upon detecting that the satellite has returned to IDLE, considers the pass complete and also returns to its IDLE state. If visibility is lost at any time, the ground station's update_state method forces an immediate return to the IDLE state, recording a loss-of-signal event.

For managing multiple ground stations, the simulator implements an implicit priority. Within the main loop, the COPE station has its update_state method executed before COPE. Likewise, the satellite's update_state_communication method checks the state of COPE before checking that of COPE-S. In practice, this ensures that if both stations are simultaneously visible, the satellite will communicate with COPE.

3.2.5 Activity Execution Simulation

The core of the satellite's autonomy lies in its ability to manage and execute tasks. This logic is encapsulated in the update_state_activity method, operating independently of communication. Within the main simulation loop, activity is always updated first at each time step, ensuring that activity decisions are made before any new communication interaction.

When the satellite antenna receives a telecommand (TC) packet from the ground station, it does not execute them immediately. Instead, it inserts them into the satellite's internal command queue, command_queue. This data structure is implemented as a priority queue (a *min-heap* from the heapq library), which orders commands based on the prioridade field defined in the commands.json file. An insertion counter (command_insertion_counter) is used as a tiebreaker to ensure FIFO (*First-In, First-Out*) ordering for commands with identical priority.

The "brain" operates as a simple Finite State Machine (FSM) with two main states: IDLE and BUSY. If the satellite is in the IDLE state and its command_queue is not empty, it adopts an opportunistic execution logic: it extracts the highest-priority command from the queue (the top of the *heap*) and submits it for prerequisite checking via the auxiliary method _verificar_prerequisitos. This method is the gateway to execution. It validates the command against the current world state (world_state), checking conditions such as target-area visibility, whether a dependency command has already been completed, or whether a waiting period (*delay*) has already been satisfied.

If all prerequisites are satisfied, the satellite transitions to the BUSY state. At this moment, it stores the command under execution, queries the action catalog to determine the activity duration, and sets a completion timer. If the prerequisites are not satisfied, the command is reinserted into the priority queue and the satellite remains IDLE, ready to evaluate the next command in the following cycle.

While in the BUSY state, the satellite monitors two conditions at each time step. First, it checks whether the current time has exceeded the completion timer. Second, it continuously reevaluates all prerequisites of the command under execution. This implements an interruption logic: if a continuous prerequisite, such as target visibility, is lost during execution, the _verificar_prerequisitos method will fail. The satellite will record a "FAILURE," interrupt the task, and reinsert the failed command back into the priority queue, returning to the IDLE state.

If the completion timer is reached without interruptions, the activity is considered a "SUCCESS." The command identifier is added to the completed_commands set, and the output files defined in commands.json are recorded in the output_files dictionary. This dictionary, together with the current activity state, constitutes the activity telemetry. During the next communication pass, when the satellite antenna is in the DOWNLINKING_DATA state, this data packet will be sent to the ground station, closing the command cycle and allowing mission control to receive the results of the executed tasks.

3.2.6 Results Structure

At the end of the main simulation loop, the main.py module executes a final data collection and persistence routine, generating a set of output artifacts. These artifacts are divided into two categories: a graphical result for visual validation of mission geometry and a set of textual log files for detailed traceability of communication and activity events.

The graphical result consists of an image that, using the Cartopy library, renders the 2D projection of the satellite orbital trajectory (the *ground track*), the sensor coverage swath, and the ground-station visibility cones on a world map. Its main purpose is to provide visual and qualitative verification of the physical simulation phase, confirming whether the calculated visibility windows correspond to the expected orbital geometry.

The primary results of the simulation are the textual log files, which provide a complete chronicle of all events that occurred. The main.py module consolidates the log lists from all agent instances and writes them to individual text files.

The formatting of these files is standardized to facilitate reading. Each event recorded in the objects is a tuple containing a timestamp, an event type, and a descriptive message.

The content of these logs enables a multi-perspective analysis of the mission. The sgdc1_activity.txt file records the satellite's activity events, allowing the lifecycle of all satellite activity executions to be traced. The communication files sgdc1_comm.txt and cope_comm.txt record handshake protocol events from the perspective of both agents, detailing pass start, telecommand transmission, and interruptions due to loss of visibility. Finally, the cope_downlink_data.txt and cope_s_downlink_data.txt files are crucial, as they record not only *events* but the *telemetry payload* received from the satellite, including the final output_files dictionary, which confirms which data files were generated and are ready for *downlink*. Taken together, these files provide complete traceability, allowing failure debugging and verification of the simulated system's operational performance.

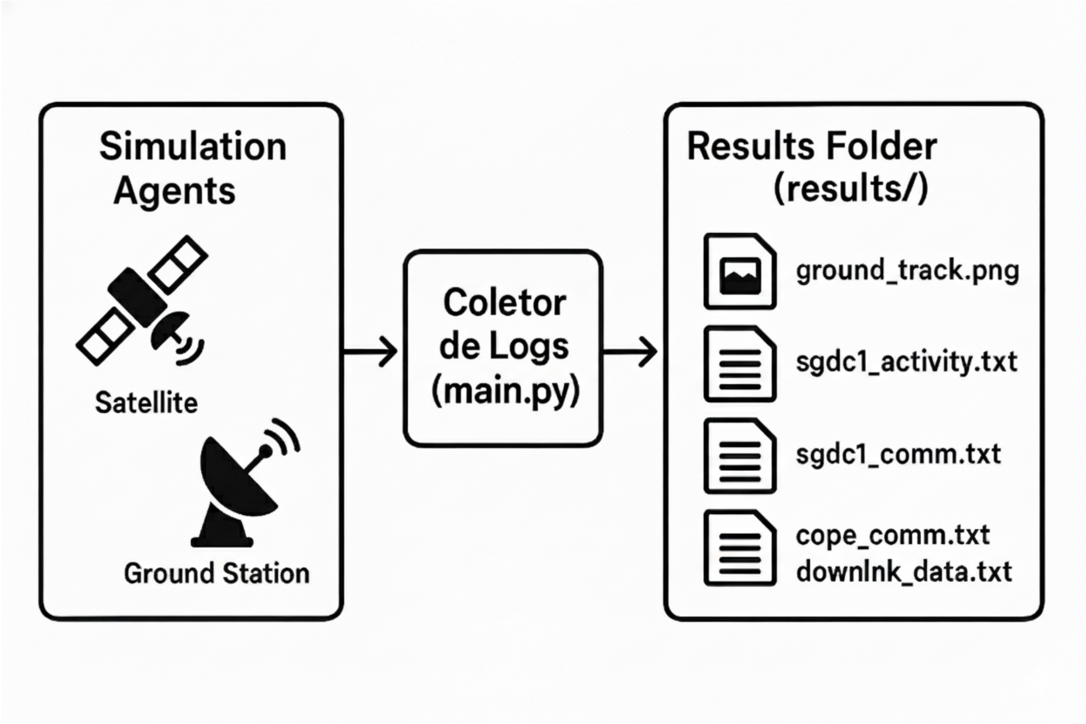

FIGURE 3.6 – Flowchart of the generation of simulation result artifacts.

3.3 State Machines

The execution logic of the simulation is governed by three distinct Finite State Machines (FSMs), which operate together within the main event loop. Two FSMs manage the communication protocol, while the third manages the satellite's autonomous task execution.

3.3.1 Ground Station Communication

The communication logic of each instance of the Ground_Station class is controlled by its internal comm_status attribute. This FSM is responsible for initiating contact, sending commands, and receiving telemetry. The states are:

- IDLE: The default state. The station is not in communication. It transitions to WAITING_BEACON at the instant when the geometric pre-computation vector indicates visibility with the satellite.
- WAITING_BEACON: The station is waiting for the satellite's initial signal. It remains in this state until it detects that the satellite's public state attribute has changed to UPLINKING.
- UPLINKING: Upon detecting that the satellite is ready to receive, the station transitions to this state. An internal timer is started. The station remains in this state

until the timer expires. Upon expiration, it groups all commands in its command_-outbox queue, places them in the transmission *buffer*, and empties its local queue. Immediately afterward, it transitions to DOWNLINKING.

- DOWNLINKING: The station has already sent its commands and is now waiting for the satellite's response. It remains in this state while the satellite transmits its data. The transition back to IDLE occurs when the station detects that the satellite has completed its cycle and its status has returned to IDLE.

A global failure transition overrides this logic: if the ground station loses visibility with the satellite at any instant, the station's update_state method forces an immediate return to the IDLE state, regardless of the current state, recording a loss-of-signal event.

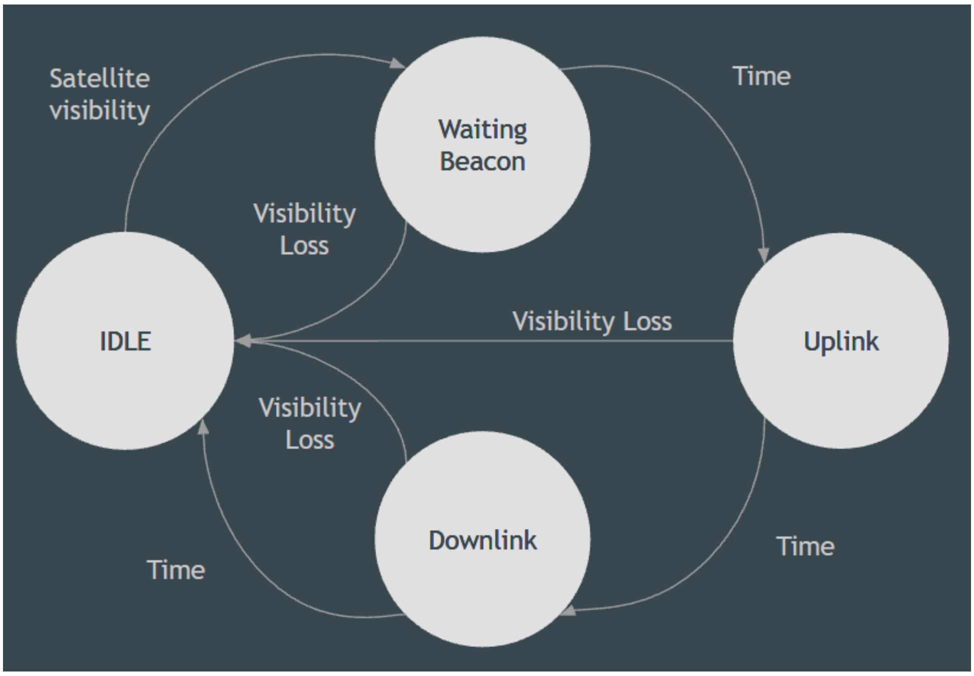

FIGURE 3.7 – Finite State Machine for the ground station communication logic.

3.3.2 Satellite Communication

The satellite has two independent FSMs. The first, its antenna, mirrors the ground station logic. The states are:

- IDLE: The default state. The antenna is not in communication. It transitions to DOWNLINKING_BEACON upon detecting that a visible station (with priority given to COPE) has changed its state to DOWNLINKING_BEACON.
- DOWNLINKING_BEACON: The satellite is transmitting its initial message to the ground station. A timer is started. Upon expiration, the state automatically changes to UPLINKING.

- UPLINKING: The most critical state of communication. The satellite is receiving commands from the ground station. A timer is started. During this state, the satellite monitors the input *buffer*. If commands are received, they are processed: each command is inserted into the internal activity priority queue via heapq.heappush, and its arrival time is recorded. At the end of the timer, the state transitions to DOWNLINKING_DATA.
- DOWNLINKING_DATA: The satellite aggregates its state telemetry and the results of completed activities and places them in the output buffer. A timer is started. Upon expiration, the communication is considered complete and the state returns to IDLE.

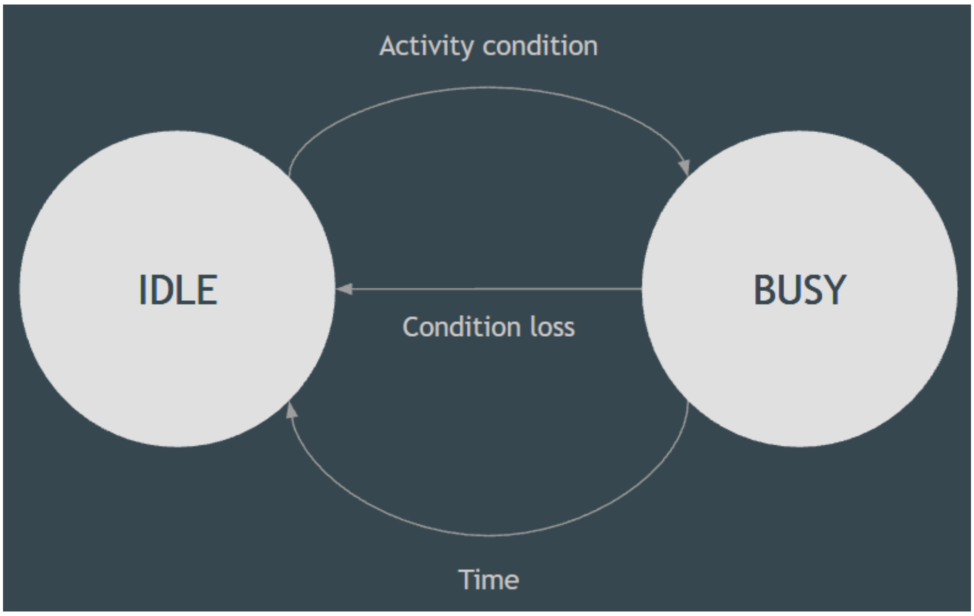

FIGURE 3.8 – Finite State Machine for the satellite communication logic.

3.3.3 Satellite Activity Execution

The second FSM of the satellite is its "brain." It operates independently of the antenna and is updated first at each time step. This FSM has only two states:

- IDLE: The satellite is not executing any internal task. In this state, at each time step, the satellite checks the top of its priority queue. It extracts the highest-priority command and submits it to the verification method. If the verification is successful, the state transitions to BUSY. Otherwise, the command is reinserted into the queue and the satellite remains IDLE.
- BUSY: The satellite is actively executing a task. Upon entering this state, a completion timer is set based on the activity duration, read from the sat_actions.json

file.

Two conditions can cause the satellite to leave the BUSY state:

1. Success: The current time reaches or exceeds the activity time. The task is marked as "SUCCESS," its ID is added to the set of completed tasks, its results are stored, and the state returns to IDLE.

2. Failure/Interruption: At each time step, while in BUSY, the continuous prerequisites are rechecked. If this verification fails, the task is immediately interrupted. A "FAILURE" event is logged, the command is reinserted into the queue, and the state returns to IDLE.

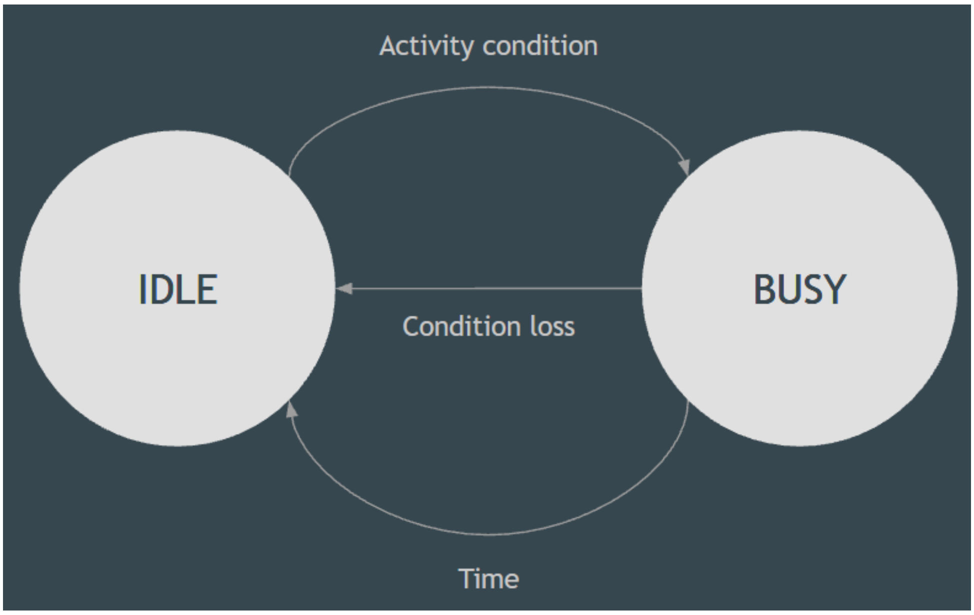

FIGURE 3.9 – Finite State Machine for the satellite activity execution logic.

4 Results

This section presents the results obtained from the execution of the simulation, according to the methodology detailed in Chapter 3. The test scenario is based on the input parameters defined in the configuration files, simulating a 24-hour period of operation of the SGDC-1 satellite. The objective is to validate the state machines, the prerequisite logic, and the interaction between the physical (orbital) simulation and the event simulation (communication and activities).

4.1 Simulation Description

The simulation was executed with a set of input parameters to validate the architecture of the model. The main agent is the SGDC-1 satellite, instantiated from its real TLE (Two-Line Element). The TLE, with an epoch of 2025, indicates a geostationary satellite, with a mean motion of 1.0027 revolutions/day and a very low inclination (0.0439°). The simulation horizon was configured for a period of 24 hours, with a discrete time step of 60 seconds.

Two ground stations were modeled: COPE (Brasília) and COPE-S (Rio de Janeiro), both with a line-of-sight cone of 40° from the zenith. The satellite was configured with an imaging sensor with a half-opening (off-nadir) angle of 6.0°, used to monitor the Region of Interest (ROI) São Paulo (SP), defined by a polygon over the state of São Paulo.

The mission plan was loaded via commands.json, assigning three telecommands (TCs) to the output queue of the COPE station:

1. TC001_MANUTENCAO: A low-priority (3) battery maintenance task, with a duration of 1800 seconds and a prerequisite delay of 3600 seconds after receipt of the command.

2. TC002_FOTO_SP: A high-priority (1) task to image the "SP" ROI, with a duration of 240 seconds and a prerequisite of continuous target visibility.

3. TC003_AJUSTE: A high-priority (1) orbital adjustment task, with a duration of

5000 seconds and no execution prerequisites.

The communication times were defined in comm_config.json, allocating 60 seconds for the command *uplink* and 60 seconds for the telemetry *downlink*.

4.2 Physical Simulation Analysis

The first stage of results focuses on validating the mission geometry. Since the SGDC-1 satellite is geostationary, the expected behavior is high positional stability with respect to the ground.

4.2.1 Ground Track and Visibility Analysis

Execution of the ground_track_plot function, which processes the complete trajectory of the satellite and converts it to geodetic coordinates (latitude, longitude), produces the trajectory plot. As expected for a GEO satellite with low inclination and eccentricity, the resulting *ground track* is not a long sinusoidal strip, but rather an almost stationary point over Brazil. The small inclination of 0.0439° and eccentricity of 0.0003191 result in minimal daily drift, known as an analemma, which is correctly captured by SGP4 propagation.

A direct consequence of this stability is constant visibility. The boolean vectors generated by the calculate_visibility method take the value *True* for all 1440 time steps of the simulation (24 hours). The COPE station has full and uninterrupted coverage of the satellite, since it remains fixed in its skies within its 40° line-of-sight cone, while for the COPE-S station it is not visible for this line-of-sight cone. Similarly, the 6° sensor *footprint* illuminates a fixed coverage area on the ground which, given the satellite longitude, also continuously covers SP. Therefore, the SP visibility vector also takes the value *True* for the entire simulation.

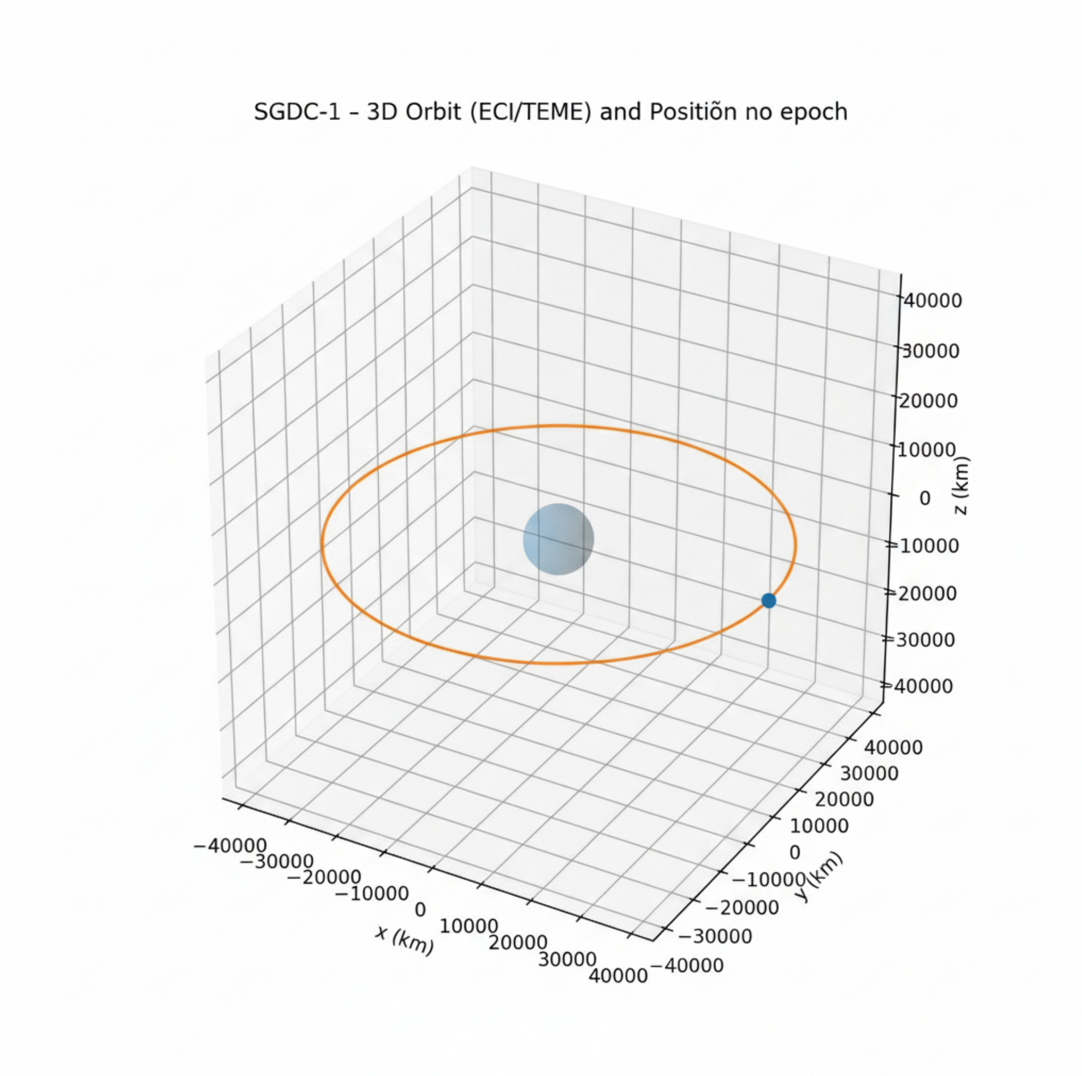

FIGURE 4.1 – 3D visualization of the satellite orbit.

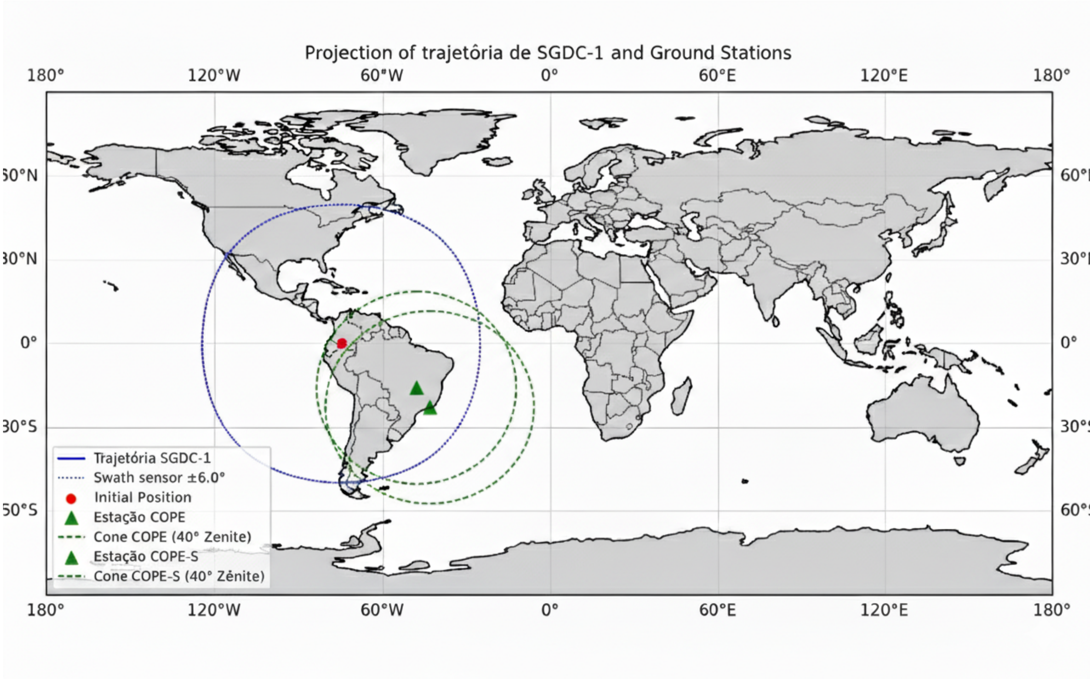

FIGURE 4.2 – Visualization of the geostationary ground track and station coverage.

4.2.2 Orbital Analysis

The SGP4 propagation worked as expected, generating a stable trajectory. Because this is a geostationary orbit, the true anomaly should behave approximately linearly, since it is an approximately circular orbit. This behavior was observed in the simulation, see Fig. 4.3.

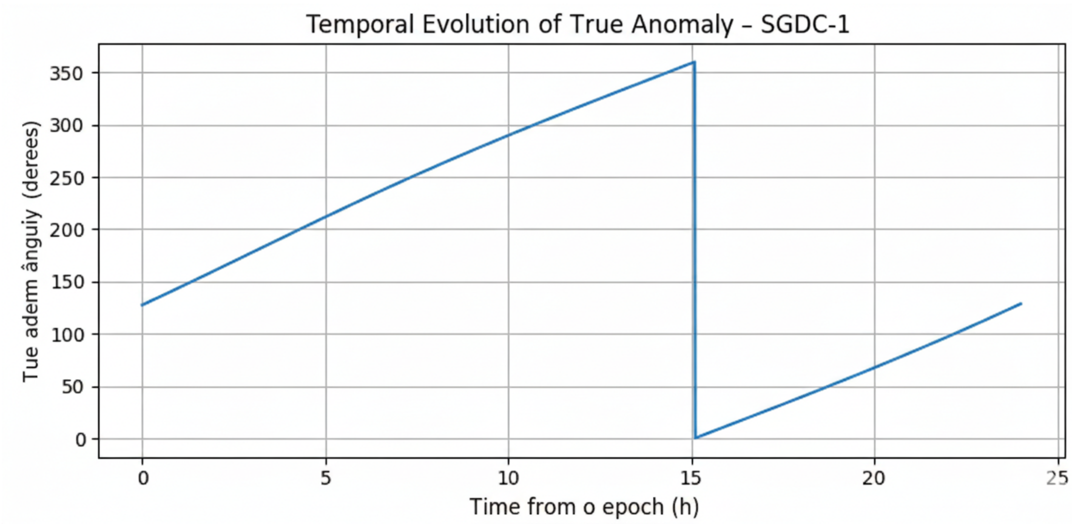

FIGURE 4.3 – Evolution of the true anomaly over time.

Furthermore, given the location of the ground stations and the opening angles of the ground-station and satellite sensors observed in Fig. 4.2, it is expected that, due to the low variation of the geostationary orbit, the COPE station will be available for communication throughout the entire simulation, while COPE-S will be unavailable for the entire simulation. In addition, the SP region should be visible to the satellite throughout the entire simulation.

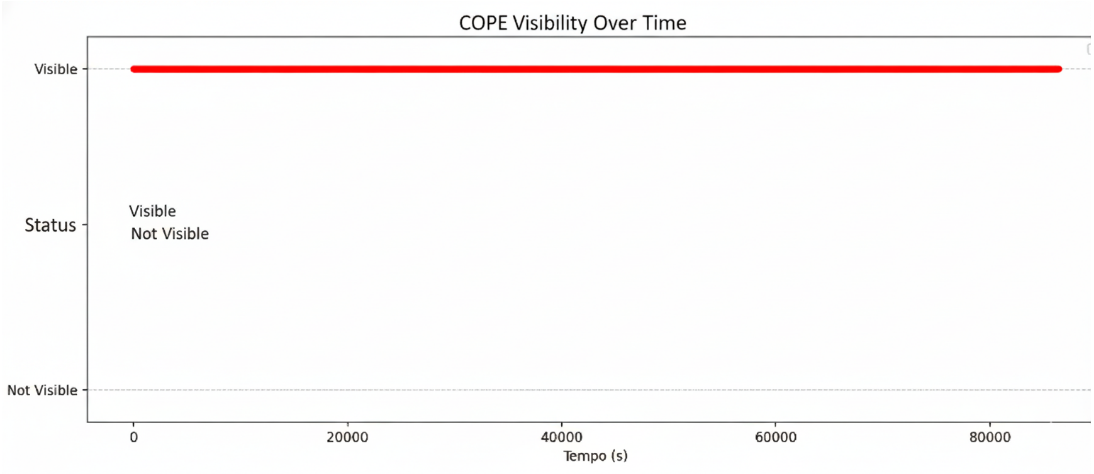

FIGURE 4.4 – Visibility of the COPE ground station over time.

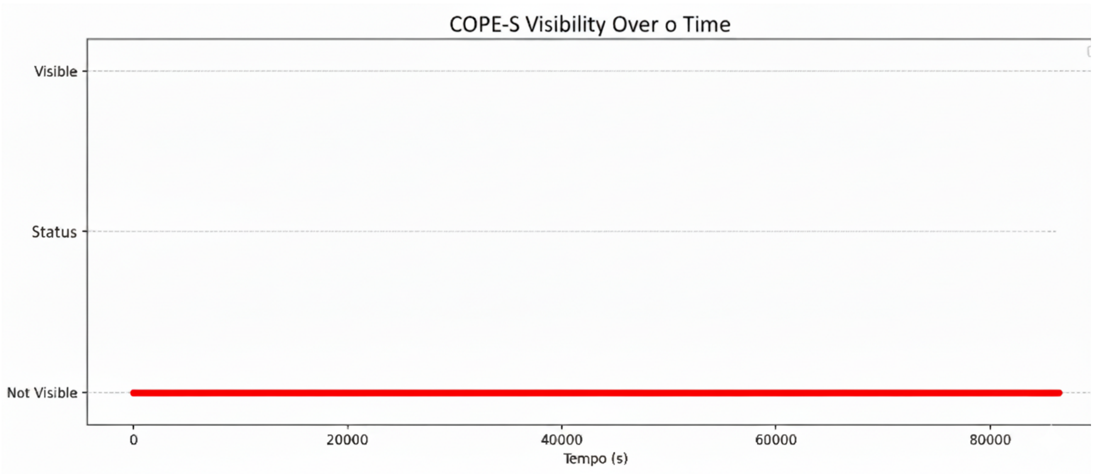

FIGURE 4.5 – Visibility of the COPE-S ground station over time.

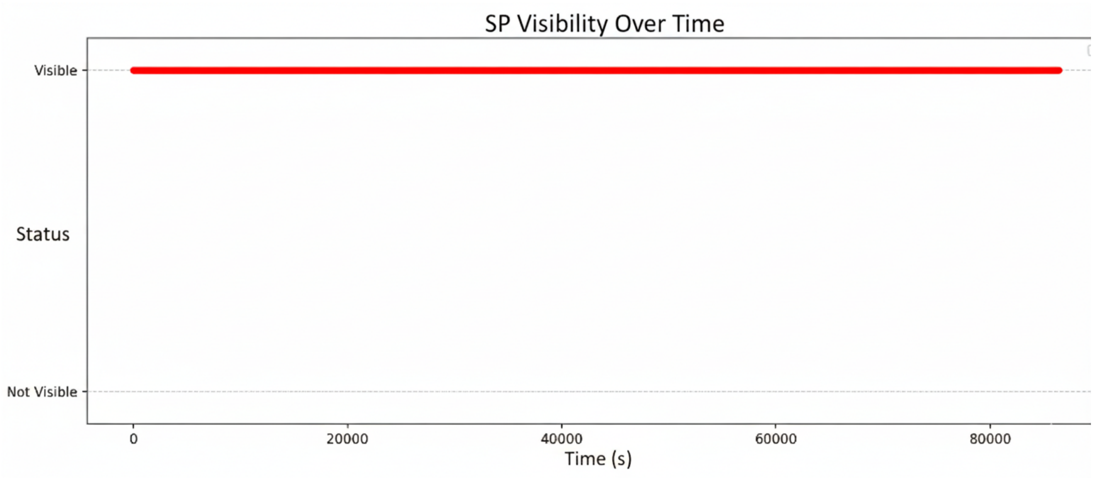

FIGURE 4.6 – Visibility of the SP region over time.

4.3 Communication and Activity Simulation Analysis

Analysis of the textual output logs enables validation of the state-machine logic. The GEO scenario, with its constant visibility, revealed a complex and successful emergent behavior that differs significantly from a simple LEO satellite pass.

4.3.1 Communication Logs Analysis

The expected behavior for a constant-visibility (GEO) scenario could be a single communication *handshake* at the beginning of the simulation. However, the cope_comm.txt and sgdc1_comm.txt logs show a much more dynamic behavior, revealing four distinct communication *handshakes* throughout the simulation.

The first contact occurs as expected, starting at t=0. The cope_comm.txt log shows the station transitioning through "PASS_START" (t=0), "UPLINK_START" (t=120), and registering "Pacote com 3 TCs enviado" in t=180. The satellite log, sgdc1_comm.txt, mirrors this interaction perfectly, recording "Recebidos 3 TCs" at t=240. The initial *handshake* is completed at t=300.

The most revealing aspect is the occurrence of subsequent *handshakes* (starting at t=540, t=5583, t=7385). The COPE station, even with its transmission queue (command_outbox) empty, re-initiates communication. Analysis of the sgdc1_comm.txt log shows that in these subsequent contacts, the satellite records "Nenhum comando recebido por UPLINK." This confirms that the ground-station logic is not purely reactive to command transmission. The ground station detects that the satellite has new data in its output_files buffer and proactively initiates a new *handshake* for the sole purpose of performing the data *downlink*.

This data "polling" behavior is the key to the success of the simulated mission. The COPE-S station logs, in turn, are empty, which is also an expected result. The priority logic and the constant visibility of COPE ensure that COPE-S never has the opportunity to communicate, since COPE is always served first.

4.3.2 Activity Logs Analysis

The sgdc1_activity.txt log is the satellite's activity record and validates correct operation of the priority queue and the prerequisite checking.

Temporal analysis of the events (Figure **??**) shows that immediately after completion of the first *handshake* (at t=300), the "brain" (which was IDLE) evaluates its queue. It receives the three commands: TC001 (Priority 3, *delay* 3600s), TC002 (Priority 1, SP visibility), and TC003 (Priority 1, no prerequisites).

The sgdc1_activity.txt log shows that the priority queue (heapq) operates as intended:

1. t=300: The satellite starts the first priority-1 task. The prerequisite of ROI SP visibility is satisfied.
2. t=540: The task (duration 240s) is completed. The satellite returns to IDLE and immediately evaluates the next item in the queue.
3. t=540: The next priority-1 task is started.
4. t=5583: The task (duration 5000s) is completed. The "brain" returns to IDLE.
5. t=5583: The last item in the queue is evaluated. Its *delay* prerequisite of 3600s, counted from receipt (t≈240s), has already been amply satisfied. The task is started.
6. t=7385: The final task (duration 1800s) is completed.

The execution sequence shows that the prioritization logic and the prerequisite verification (both visibility and time-based) were successfully implemented.

4.3.3 Data Log Analysis

The cope_downlink_data.txt log is the piece of evidence that connects the two FSMs (communication and activity). It records the exact *payload* received by the ground station in each communication. Analysis of this log closes the validation loop:

- Downlink 1 (t=240): The station receives telemetry during the first communication. The 3 TCs have just arrived, and the satellite has not yet processed them.

- Downlink 2 (t=780): Occurs during the communication initiated at t=540, triggered by the completion of TC002. This log proves that the ground station performed the *downlink* of the first image result while the satellite was executing the second task.
- Downlink 3 (t=5824): Occurs during the communication initiated at t=5583. The orbital adjustment result was downloaded.
- Downlink 4 (t=7625): Occurs during the final communication. The last result file is downloaded, and the satellite reports that there are no more commands in the queue and no ongoing execution.

4.3.4 Behavior Verification

The expected behavior was that the three commands would be sent, executed in order of priority, and that their prerequisites would be respected. The behavior actually observed, documented in the logs, not only met all these expectations but also demonstrated a non-trivial emergent capability.

The simulation validated that the ground station's *polling* logic of initiating contact upon detecting pending output files on the satellite is a highly effective data *downlink* strategy in a permanent-contact (GEO) scenario. The mission, as simulated, was a complete success: all commands were executed in the correct order, and all generated result files were subsequently downloaded by the ground station.

4.4 Future Improvements

The simulator developed, with its modular architecture and clear separation between physical propagation and event logic, has proven to be an effective tool for validating the sequencing of operations of a geostationary satellite. The current platform serves as a robust foundation for a series of extensions that would significantly increase its fidelity and scope, aligning it more closely with the complex objectives of PESE, such as constellation management.

An immediate usability improvement would be to refactor the output structure of the results. Currently, logs are generated as .txt files with formatting optimized for human reading. Migrating to a structured log format, such as JSON Lines (JSONL) or CSV, in which each event is recorded as a single line or object, would drastically simplify computational post-processing. This would allow the creation of more robust analysis *scripts* to automatically generate Gantt charts, resource utilization statistics, and performance validation.

Second, the network logic of the ground stations could be expanded. The current model implements simple communication prioritization based on the order of execution in the main loop (COPE before COPE-S). A natural extension would be to implement a failover logic, in which the COPE-S station (currently idle in the logs) would actively assume communications if the COPE station were marked as unavailable. Going further, inter-station communication could be modeled, whereby one station could route commands or telemetry to another through terrestrial networks, simulating a more resilient and distributed ground-control network.

The fidelity of the satellite simulation can also be deepened through the modeling of internal subsystems. Currently, activities have only a "duration." A significant evolution would be to add parameters of energy consumption (in Watts) and data generation (in Megabits) for each activity. This would require the Satellite class to manage two new finite resources: the state of charge of the battery (SOC) and the onboard data storage capacity. New prerequisite rules could then be created, allowing the satellite to postpone or reject commands not due to lack of opportunity, but due to internal resource constraints.

In addition, the current form of data *input* is simple, and makes use of a widely adopted data structure, JSON. Thus, it is possible to generate this type of file through other *software* and automatically produce inputs for running the simulation, facilitating integration with the CONCEPTIO laboratory, one of the goals of the project.

Finally, because the code was written in the simplest possible way with respect to data structures, modification, enhancement, and the addition of functions and methods are extremely straightforward, which provides the developer with considerable flexibility to create a highly customizable project.

5 Conclusions

This Undergraduate Thesis had as its general objective the development and validation of a computational simulator for the integrated modeling of the operation of space systems, with an initial focus on PESE scenarios. The proposed simulator aimed to unify orbital dynamics, telecommand logic, ground-station operations, and activity execution rules into a single analysis tool implemented in Python.

It is possible to conclude that this main objective, together with the large majority of its specific objectives, was fully achieved. The work resulted in a functional simulator whose core architecture exhibits a robust separation between physical simulation, which is computationally intensive, and event simulation, which is based on decision logic. The physical simulation phase demonstrated the correct implementation of SGP4 orbital propagation and the computation of visibility vectors for stations and regions of interest, as validated by the trajectory and visibility plots.

The central success of the project lies in the validation of the telecommand logging and queuing module, implemented through the satellite activity Finite State Machine (FSM). Analysis of the result logs provided unequivocal evidence that the satellite was able to:

1. Ingest telecommands received from the ground station;
2. Manage a priority queue (heapq), executing priority-1 tasks before priority-3 tasks;
3. Check and respect multiple types of prerequisites, successfully validating both continuous target visibility and a temporal delay.

Furthermore, the simulation was able to reveal an emergent operational behavior. Analysis of the communication logs demonstrated that, in a constant-visibility (GEO) scenario, the ground station's *polling* logic was a highly effective *downlink* strategy. The simulated mission was a complete success: all commands were executed in the correct order, and all generated result files were subsequently downloaded by the ground station, validating the end-to-end operational cycle.

The greatest value of this work, however, does not lie solely in the specific scenario that was validated, but in the foundation it establishes. The architecture proved to be inherently flexible and extensible. The modularity achieved through the division into classes and scenario parameterization via JSON files confers a high degree of adaptability to the simulator. Reconfiguration for a LEO mission, for example, would require only changes to the TLE and the ROIs, without deep modifications to the source code.

This flexibility points directly to future work. The class structure is ready to be expanded with the modeling of subsystem resources such as power and data storage. The network logic can evolve to simulate station failures (*failover*) and inter-station communication. This final integration step, facilitated by JSON-based data input, will allow the simulator to decouple from its internal execution *loop* and connect to external systems such as the CONCEPTIO laboratory, thereby fulfilling the final objective of the project.

This work, therefore, delivers not merely a program, but a validated and extensible simulation platform, capable of growing in fidelity and complexity, and serving as a valuable tool for the analysis of operations and validation of mission concepts within the scope of the Strategic Space Systems Program.

References

AGÊNCIA ESPACIAL BRASILEIRA. **Programa Nacional de Atividades Espaciais: PNAE: 2022–2031**. 4. ed. Brasília, DF: Ministério da Ciência, Tecnologia e Inovação e Agência Espacial Brasileira, 2022. Disponível em: `https://www.gov.br/aeb/pt-br/programa-espacial-brasileiro/programa-nacional-de-atividades-espaciais`. Acesso em: 31 maio 2025. Cit. on pp. 11, 12.

ASTROPY COLLABORATION. Astropy: A Community Python Package for Astronomy. **Astronomy & Astrophysics**, v. 558, a33, 2013. DOI: `10.1051/0004-6361/201322068`. Cit. on p. 27.

BRASIL. MINISTÉRIO DA DEFESA. ESTADO-MAIOR CONJUNTO DAS FORÇAS ARMADAS. **Programa Estratégico de Sistemas Espaciais (PESE)**. Brasília, DF, 2018. Disponível em: `https://www.gov.br/defesa/pt-br/arquivos/ajuste-01/legislacao/emcfa/publicacoes/doutrina/` (acesso em: 31 maio 2025). Cit. on pp. 11, 12.

CAMPBELL, J. B. **Introduction to Remote Sensing**. 3. ed. New York, NY: Guilford Press, 2002. ISBN 978-1572306401. Cit. on pp. 19–21.

CURTIS, H. D. **Orbital Mechanics for Engineering Students**. Fourth. Oxford, UK: Elsevier/Butterworth-Heinemann, 2020. Cit. on pp. 16, 17, 19.

D'AMATO, A. S. Alinhamento do Programa Estratégico de Sistemas Espaciais à Estratégia Nacional de Defesa. **Revista da UNIFA**, Rio de Janeiro, v. 30, n. 2, p. 24–33, July 2017. Disponível em: `https://www2.fab.mil.br/unifa/images/revista/pdf/v30n2/Art-74-Alinhamento-R3.pdf`. Acesso em: 31 maio 2025. Cit. on p. 12.

FOUNDATION, P. S. **Python 3 Documentation**. [*S.l.: s.n.*], 2020. `https://docs.python.org/3/`. Acesso em: 07 jun. 2025. Cit. on p. 26.

HARRIS, C. R.; MILLMAN, K. J.; WALT, S. J. van der; GOMMERS, R.; VIRTANEN, P.; COURNAPEAU, D.; WIESER, E.; TAYLOR, J.; BERG, S.; SMITH, N. J.; KERN, R.; PICUS, M.; HOYER, S.; KERKWIJK, M. H. van; BRETT, M.; HALDANE, A.; FERNÁNDEZ, J.; GARCÍA, I., *et al.* Array Programming with NumPy. **Nature**, v. 585, p. 357–362, 2020. DOI: `10.1038/s41586-020-2649-2`. Cit. on p. 27.

HUNTER, J. D. Matplotlib: A 2D Graphics Environment. **Computing in Science & Engineering**, v. 9, n. 3, p. 90–95, 2007. DOI: `10.1109/MCSE.2007.55`. Cit. on p. 27.

KLUEVER, C. A. **Spaceflight Dynamics**. Cambridge, UK: Cambridge University Press, 2018. Cit. on pp. 16, 17.

LEÓN, P. de; ALARCÓN, Á.; SOTO, J.; ZULUAGA, J. I. poliastro: An open source Python library for interactive Astrodynamics. **Journal of Open Source Software**, v. 3, n. 30, p. 1059, 2018. DOI: `10.21105/joss.01059`. Cit. on p. 26.

LILLESAND, T. M.; KIEFER, R. W.; CHIPMAN, J. **Remote Sensing and Image Interpretation**. 7. ed. Hoboken, NJ: Wiley, 2015. ISBN 978-1118343289. Cit. on pp. 19, 20.

MARAL, G.; BOUSQUET, M. **Satellite Communications Systems: Systems, Techniques and Technology**. 5. ed. Chichester: Wiley, 2011. ISBN 978-0470747452. Cit. on pp. 22–24.

MARTINS, P. R. P. As Famílias de Satélites do PESE e seu emprego no GptOpFuzNav. **Âncoras e Fuzis**, v. 50, p. 73–84, 2021. Available from: `https://portaldeperiodicos.marinha.mil.br/index.php/ancorasefuzis/article/view/2299`. Cit. on pp. 19–23.

MCKINNEY, W. **Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython**. 2. ed. [*S.l.*]: O'Reilly Media, 2018. ISBN 978-1491957660. Cit. on p. 27.

MIRANDA, R. D. C. **O Programa Estratégico de Sistemas Espaciais: uma questão de Defesa ou de Estado?** Brasília, 2019. Orientador: Cel R/1 Antônio Jorge Dantas de Oliveira. Cit. on p. 12.

MONTENBRUCK, O.; GILL, E. **Satellite Orbits: Models, Methods and Applications**. Second. Berlin, Germany: Springer, 2012. Cit. on p. 16.

PRATT, T.; BOSTIAN, C.; ALLNUTT, P. **Satellite Communications**. 2. ed. Chichester: Wiley, 2003. ISBN 978-0471340365. Cit. on pp. 22, 23.

RICHARIA, G. R. **Satellite Communication Systems: Design Principles**. 1. ed. London: Macmillan Press, 2012. Cit. on pp. 23, 24.

SHANNON, C. E. A Mathematical Theory of Communication. **Bell System Technical Journal**, v. 27, n. 3, p. 379–423, 1948. Cit. on p. 23.

SKOLNIK, M. I. **Radar Handbook**. 3. ed. New York: McGraw-Hill, 2008. Cit. on p. 24.

Annex A - main.py Python Code

```
# ============================================
# Arquivo main.py responsável pela orquestração da simulação
# ============================================

import numpy as np

import astropy.units as units

from poliastro.bodies import Earth

import orbit_funcs
import ground_station_funcs
from satellite import Satellite
from ground_station import Ground_Station

import json

run_comms = True

# ======= 1) Insira aqui o TLE ATUAL do SGDC-1 (NORAD 42692) =======
# Linhas exatas do TLE obtidas no CelesTrak ou de qualquer outra fonte

TLE_LINE1 = "1 25544U 98067A   23315.86477028  .00010878  00000+0  19782-3 0  9999"
TLE_LINE2 = "2 25544  51.6416 359.8887 0005703 147.2885 242.8710 15.49503463403306"


# Semiabertura do sensor (off-nadir), em graus
sensor_opening_deg = 6.0 * units.deg

# Definição do satélite de interesse na análise
sgdc1 = Satellite(TLE_LINE1, TLE_LINE2, sensor_opening_deg, name="SAT")

```

```
# Epoch do TLE em Julian Date (usaremos como t0 para as amostragens)

t0 = sgdc1.epoch_time
periodo_ref = 6 * 60 * 60 # 24H
passo_de_tempo_seg = 10.0
num_pts = int(periodo_ref / passo_de_tempo_seg)

relative_time = np.linspace(0.0, periodo_ref, num_pts)
times = t0 + relative_time * units.s
sgdc1_rs_km, sgdc1_vs_kms = sgdc1.propagate(times)

# ======= Cálculo do verdadeiro ângulo (true anomaly) a partir de r,v =======
# Faremos um cálculo "clássico" com o parâmetro gravitacional da Terra de poliastro.
mu_km3_s2 = Earth.k.to((units.km**3) / (units.s**2)).value

f_deg = np.zeros(num_pts)
e_list = np.zeros(num_pts)
for i in range(num_pts):
    f_deg[i], e_list[i] = orbit_funcs.true_anomaly_from_rv(sgdc1_rs_km[i], sgdc1_vs_kms[i], mu_km3_s2)

# ======= Plot 3D da órbita (ECI/TEME) com um ponto no satélite =======
orbit_funcs.plot_final_point_orbit(sgdc1_rs_km, "results/orbit.png")

# ======= Evolução temporal do verdadeiro ângulo (true anomaly) =======
orbit_funcs.true_anomaly_evolution(times, f_deg, "results/true_anomaly_evolution.png")

# ======= GIF do satélite em órbita =======
# orbit_funcs.trajectory_gif_generation(sgdc1_rs_km, "results/sgdc1_orbita.gif")

# Definição das estações de solo

lat_cope = -15.784 * units.deg # graus
lon_cope = -47.908 * units.deg # graus
alt_cope = 1.172 * units.km  # altitude aproximada

lat_cope_s = -22.821 * units.deg # graus
lon_cope_s = -43.187 * units.deg # graus
alt_cope_s = 0.01 * units.km  # altitude aproximada
```

```python

# Ângulo a partir do zênite
opening_deg = 85.0 * units.deg

# Agora criamos os OBJETOS Ground_Station

cope_station = Ground_Station(name="COPE",
                              lat=lat_cope,
                              lon=lon_cope,
                              height=alt_cope,
                              opening_deg=opening_deg)

cope_s_station = Ground_Station(name="COPE-S",
                              lat=lat_cope_s,
                              lon=lon_cope_s,
                              height=alt_cope_s,
                              opening_deg=opening_deg)

# ======= Análise de Visibilidade =======
# Pedimos à ESTAÇÃO para calcular a visibilidade do SATÉLITE
# Passamos a trajetória do satélite (rs_km) e os tempos (times)

cope_visible, cope_elevs = cope_station.calculate_visibility(
    sgdc1.rs_km, sgdc1.times
)

cope_s_visible, cope_s_elevs = cope_s_station.calculate_visibility(
    sgdc1.rs_km, sgdc1.times
)

orbit_funcs.plotar_status_simulacao(cope_visible, relative_time, cope_station.name, "results/cope_visibility.png")
orbit_funcs.plotar_status_simulacao(cope_s_visible, relative_time, cope_s_station.name, "results/cope_s_visibility.png")

# Pedimos à ESTAÇÃO para calcular a visibilidade das regiões de interesse (rois)

with open('rois.json', 'r') as f:
    rois = json.load(f)

visibilidade_rois = {}
```

```python

for roi_name, roi_data in rois.items():

    polygon = roi_data['polygon']
    vis_array = sgdc1.check_polygon_visibility(polygon_vertices_deg=polygon,
    ↪   polygon_name=roi_name)
    visibilidade_rois[roi_name] = vis_array

    orbit_funcs.plotar_status_simulacao(vis_array, relative_time, roi_name,
    ↪   f"results/{roi_name}_visibility.png")

# Importando comandos para as bases de solo

with open('commands.json', 'r') as f:
    commands_data = json.load(f)

cope_station.load_commands(commands_data, sgdc1.name)
cope_s_station.load_commands(commands_data, sgdc1.name)

with open('sat_actions.json', 'r') as f:
    sat_actions_config = json.load(f)

lista_estacoes = [cope_station, cope_s_station]

# Ground track plot
ground_station_funcs.ground_track_plot(sgdc1,lista_estacoes,
↪   "results/ground_track_plot.png")

if run_comms:

    # Carrega a configuração de comunicação
    with open('comm_config.json', 'r') as f:
        comm_config = json.load(f)

    # Variáveis de estado públicas que serão trocadas
    sat_public_state = {
        'antenna_status': sgdc1.status_antenna,
        'activity_status': sgdc1.status_activity,
        'output_files': None,
        'name': sgdc1.name
    }
```

```
    cope_public_state = {
        'comm_status': cope_station.comm_status
    }
    cope_s_public_state = {
        'comm_status': cope_s_station.comm_status
    }

    # Buffers de dados em trânsito
    data_from_sat_to_cope = None
    data_from_sat_to_cope_s = None
    data_from_cope_to_sat = None
    data_from_cope_s_to_sat = None

    print("Simulação de Eventos Iniciada.")

    # Loop principal da simulação de eventos
    for i in range(len(times)):

        current_time = int(times[i].to_value('unix') - t0.to_value('unix')) # Usar
        ↪ tempo em segundos
        is_cope_visible = cope_visible[i]
        is_cope_s_visible = cope_s_visible[i]

        # Constrói o 'world_state' dinamicamente
        world_state = {}
        for roi_name, vis_array in visibilidade_rois.items():
            flag_name = f"is_visible_{roi_name}"
            world_state[flag_name] = vis_array[i]

        # Adiciona visibilidade das estações (para pré-requisitos de downlink)
        world_state['is_visible_cope'] = is_cope_visible
        world_state['is_visible_cope_s'] = is_cope_s_visible

        # Atualiza as atividades do Satélite
        sat_public_state_activity = sgdc1.update_state_activity(
            current_time,
            world_state,
            sat_actions_config
        )

        # Atualiza o Satélite
```

```
        sat_public_state, data_from_sat_to_cope, data_from_sat_to_cope_s = sgdc1.update_state_communication(
            current_time,
            cope_public_state,
            is_cope_visible,
            cope_s_public_state,
            is_cope_s_visible,
            world_state,
            data_from_cope_to_sat,
            data_from_cope_s_to_sat,
            comm_config
        )

        # Atualiza a Estação COPE
        cope_public_state, data_from_cope_to_sat = cope_station.update_state(
            current_time,
            sat_public_state,
            is_cope_visible,
            data_from_sat_to_cope,
            comm_config
        )

        # Atualiza a Estação COPE-S
       cope_s_public_state, data_from_cope_s_to_sat = cope_s_station.update_state(
            current_time,
            sat_public_state, # Passa o estado PÚBLICO do satélite
            is_cope_s_visible,
            data_from_sat_to_cope_s,
            comm_config
        )

        # Atualiza o estado público geral
        sat_public_state['activity_status'] = sat_public_state_activity['activity_status']
        sat_public_state['output_files'] = sat_public_state_activity['output_files']


    log_infos = [cope_station.log_communication, cope_station.log_downlink_data, cope_s_station.log_communication, cope_s_station.log_downlink_data, sgdc1.log_communication, sgdc1.log_activity]
```

```python
log_filenames = ['cope_comm.txt', 'cope_downlink_data.txt', 'cope_s_comm.txt', 'cope_s_downlink_data.txt', 'sgdc1_comm.txt', 'sgdc1_activity.txt']

save_path = 'results/'

for log_filename in log_filenames:
    with open(save_path + log_filename, 'w', encoding='utf-8') as f:

        log_info = log_infos[log_filenames.index(log_filename)]

        # Pega o número total de tuplas para sabermos qual é a última
        num_tuplas = len(log_info)

        # Itera pela lista de tuplas usando enumerate
        for i, tupla in enumerate(log_info):

            # Itera por cada elemento *dentro* da tupla
            for elemento in tupla:
                # Escreve o elemento seguido de uma quebra de linha ('\n')
                f.write(f"{elemento}\n")

            # Adiciona as duas linhas em branco APÓS a tupla se NÃO estamos na última tupla
            if i < num_tuplas - 1:
                f.write('\n\n')

    print(f"Arquivo '{log_filename}' criado com sucesso!")

print("Simulação de Eventos Concluída.")
```

Annex B - sattelite.py Python Code

```python
# satellite.py
#
# Classe para armazenar e processar dados de um único satélite.

import numpy as np
from sgp4.api import Satrec
from astropy.time import Time
import astropy.units as units
from astropy.coordinates import TEME, CartesianRepresentation, ITRS, EarthLocation
from collections import deque
import heapq

# Importa as funções auxiliares que já criamos
import orbit_funcs

class Satellite:
    """
    Representa um único satélite propagado via SGP4.
    """

    def __init__(self, tle_line1: str, tle_line2: str,
    ↪   sensor_opening_deg:units.Quantity, name: str = None):
        """
        Inicializa o satélite a partir de suas duas linhas de TLE.
        """
        self.tle1 = tle_line1
        self.tle2 = tle_line2
        self.sensor_opening_deg = sensor_opening_deg

        # O 'name' é útil para gráficos e legendas
        if name:
            self.name = name
```

```
        else:
            # Tenta pegar o nome do satélite da linha 1 (colunas 2-6)
            self.name = tle_line1[2:6]

        # Criar o propagador SGP4 (antiga 'sat' em teste.py)
        self.propagator = Satrec.twoline2rv(self.tle1, self.tle2)

        # Calcular e armazenar o Epoch
        epoch_jd = self.propagator.jdsatepoch + self.propagator.jdsatepochF
        self.epoch_time = Time(epoch_jd, format="jd", scale="utc")

        # Calcular e armazenar o período orbital

        # n = Média de movimento em [radianos / minuto]
        n_rad_per_min = self.propagator.no_kozai

        # T = (2 * pi) / n --> nos dá o período em [minutos]
        period_min = (2 * np.pi) / n_rad_per_min

        # Convertendo para [segundos]
        self.period_sec = period_min * 60.0

        # Atributos para armazenar os resultados da propagação
        self.times = None
        self.rs_km = None
        self.vs_kms = None

        # Fila de comandos a executar
        self.command_queue = []
        self.command_insertion_counter = 0

        # Fila de telemetria/dados a enviar
        self.data_outbox = deque()

        # Log de operações
        self.log_communication = []  # Para eventos da antena
        self.log_activity = []       # Para eventos do "cérebro" (atividades)

        # Histórico de comandos
        self.completed_commands = set()
        self.output_files = {}
```

```python

        # Estado da Antena (Comunicação)
        self.status_antenna = "IDLE"
        self.antenna_task_end_time = None # Timer para tarefas de comunicação

        # Estado Interno (Atividade/Cérebro)
        self.status_activity = "IDLE"
        self.current_activity_item = None   # Qual atividade está rodando (ex: "foto")
        self.activity_task_end_time = None  # Timer para a atividade interna

        # Dicionário para rastrear tempos de eventos
        self.event_timestamps = {}

    def propagate(self, master_times_array: Time):
        """
        Propaga a órbita do satélite para cada instante no 'master_times_array'.

        Armazena os resultados em self.rs_km e self.vs_kms.
        """
        # Armazena o array de tempo que foi usado
        self.times = master_times_array
        num_pts = len(self.times)

        # Prepara os arrays de resultado
        self.rs_km = np.zeros((num_pts, 3), dtype=float)
        self.vs_kms = np.zeros((num_pts, 3), dtype=float)

        print(f"Propagando órbita para {self.name}...")
        for i, ti in enumerate(self.times):
            r_i, v_i = orbit_funcs.rv_teme_from_sgp4(self.propagator, ti)
            self.rs_km[i] = r_i
            self.vs_kms[i] = v_i

        print(f"Propagação de {self.name} concluída.")
        return self.rs_km, self.vs_kms

    def check_polygon_visibility(self, polygon_vertices_deg: list, polygon_name: str):
        """
        Verifica se um polígono na superfície está totalmente dentro
```

```
        do footprint (sensor cone) do satélite.

        :param polygon_vertices_deg: Lista de tuplas [(lat1, lon1), (lat2, lon2), ...]
        :param sat_sensor_opening_deg: Semi-abertura do sensor (off-nadir)
        :return: Array booleano (True se o polígono estiver totalmente visível)
        """
        if self.rs_km is None:
            raise RuntimeError(f"Você deve chamar 'propagate()' primeiro para o satélite {self.name}.")

        num_pts = len(self.times)
        polygon_visible_flags = np.zeros(num_pts, dtype=bool)

        #  Converte os vértices do polígono para objetos EarthLocation
        #  Assumimos altitude 0, a menos que você queira especificá-la.
        vertex_locations = []
        for lat, lon in polygon_vertices_deg:
            vertex_locations.append(
                EarthLocation(lat=lat*units.deg, lon=lon*units.deg, height=0*units.m)
            )

        print(f"Verificando visibilidade da região {polygon_name} para {self.name}...")

        # Itera em cada instante de tempo
        for i in range(num_pts):
            t_i = self.times[i]
            r_i_teme = self.rs_km[i] # Posição em TEME

            # Converte a posição do satélite (TEME) para ITRS e depois para (Lat, Lon, Alt)
            teme_coord = TEME(CartesianRepresentation(r_i_teme*units.km), obstime=t_i)
            itrs_coord = teme_coord.transform_to(ITRS(obstime=t_i))
            sat_location = EarthLocation.from_geocentric(itrs_coord.x, itrs_coord.y, itrs_coord.z)

            # Ponto exato na superfície abaixo do satélite (altitude 0)
```

```
            sat_subpoint = EarthLocation(lat=sat_location.lat, lon=sat_location.lon, height=0*units.m)
            sat_altitude_km = sat_location.height.to(units.km).value

            #  Calcula o raio (em graus) do footprint na superfície
            footprint_radius_deg = self.sensor_ground_range(sat_altitude_km).value

            # Verifica se TODOS os vértices estão dentro do footprint
            all_vertices_visible = True # Começa assumindo que sim

            for vertex_loc in vertex_locations:

                # Obter a representação ITRS de ambos os pontos no tempo t_i
                subpoint_itrs = sat_subpoint.get_itrs(obstime=t_i)
                vertex_itrs = vertex_loc.get_itrs(obstime=t_i)

                # Calcular a separação angular entre os dois frames ITRS
                separation = subpoint_itrs.separation(vertex_itrs)
                separation_deg = separation.to(units.deg).value

                if separation_deg > footprint_radius_deg:
                  # Este vértice está fora. O polígono não está totalmente visível.
                    all_vertices_visible = False
                    break # Para de verificar os outros vértices

            # Salva o resultado para este instante de tempo
            polygon_visible_flags[i] = all_vertices_visible

        return polygon_visible_flags

    def sensor_ground_range(self, altitude_km):
        """
        Converte a semiabertura do sensor (off-nadir, em graus) no ângulo central (, em graus)
        entre o subponto e a borda do footprint na superfície.
        Geometria exata (sem aproximação plana).
        """
        Re = 6371.0  # km
        Rs = Re + altitude_km
        a = np.radians(self.sensor_opening_deg)

```

```
        # cos() obtido da solução fechada do triângulo O-S-P
        # Escolha do ramo garante =0 quando a=0.
        k = (Rs / Re) * np.sin(a)
        # Proteção numérica: para aberturas muito grandes k poderia chegar >1
        # (não é o caso de a=2°, mas deixamos robusto).
        inside = 1.0 - k**2
        inside = np.where(inside < 0.0, 0.0, inside)

        cos_psi = - (Rs / Re) * (np.sin(a)**2) + np.cos(a) * np.sqrt(inside)
        cos_psi = np.clip(cos_psi, -1.0, 1.0)
        psi_deg = np.degrees(np.arccos(cos_psi))
        return psi_deg  # em graus

    # Dentro da classe Satellite, em satellite.py

    def update_state_communication(self, current_time,
                                   cope_state, is_cope_visible,
                                   cope_s_state, is_cope_s_visible,
                                   world_state, data_from_cope_to_sat,
                                   data_from_cope_s_to_sat,   comm_config):
        """
        Gerencia a "antena" do satélite e o protocolo de comunicação.
        """

        data_to_cope = None
        data_to_cope_s = None

        # Checa se o timer (self.antenna_task_end_time) terminou
        if self.status_antenna != "IDLE" and self.antenna_task_end_time is not None and current_time >= self.antenna_task_end_time:

            if self.status_antenna == "DOWNLINKING_BEACON":
                # Terminou envio de dados de saúde, agora ouve o Uplink

                self.status_antenna = "UPLINKING"
                self.antenna_task_end_time = current_time + comm_config['duracao_uplink_comandos_seg']
                self.log_communication.append((current_time, "BEACON_END", "Beacon enviado, pronto para uplink."))
                self.log_communication.append((current_time, "UPLINK_START", "Aguardando TCs"))
```

```

            elif self.status_antenna == "DOWNLINKING_DATA":
                # Terminou de enviar dados, comunicação encerrada

                self.status_antenna = "IDLE"
                self.antenna_task_end_time = None
                self.log_communication.append((current_time, "DOWNLINK_END", "Downlink de dados concluído."))
                self.output_files = None

        if self.status_antenna == "UPLINKING":

                if is_cope_visible and cope_state['comm_status'] == "DOWNLINKING":
                    # Terminou de receber TCs, agora envia dados/fila
                    self.status_antenna = "DOWNLINKING_DATA"

                    if data_from_cope_to_sat is not None:
                        comandos_recebidos = data_from_cope_to_sat
                    else:
                        comandos_recebidos = None
                       self.log_communication.append((current_time, "UPLINK_DATA", f"Nenhum comando recebido por UPLINK"))

                    if comandos_recebidos is not None:
                       self.log_communication.append((current_time, "UPLINK_DATA", f"Recebidos {len(comandos_recebidos)} TCs."))
                        for cmd in comandos_recebidos:
                            prioridade = cmd.get('prioridade', 99)
                          item = (prioridade, self.command_insertion_counter, cmd)
                            heapq.heappush(self.command_queue, item)
                           self.event_timestamps[cmd['id_comando']] = current_time
                            self.command_insertion_counter += 1

                    duracao_tm_dados = comm_config['duracao_downlink_saude_fila_seg']
                    self.antenna_task_end_time = current_time + duracao_tm_dados
                   self.log_communication.append((current_time, "UPLINK_END", "TCs recebidos, iniciando downlink de dados."))

                elif is_cope_s_visible and cope_s_state['comm_status'] == "DOWNLINKING":
```

```
                    # Terminou de receber TCs, agora envia dados/fila
                    self.status_antenna = "DOWNLINKING_DATA"

                    if data_from_cope_s_to_sat is not None:
                        comandos_recebidos = data_from_cope_s_to_sat
                    else:
                        comandos_recebidos = None
                       self.log_communication.append((current_time, "UPLINK_DATA", f"Nenhum comando recebido por UPLINK"))

                    if comandos_recebidos is not None:
                       self.log_communication.append((current_time, "UPLINK_DATA", f"Recebidos {len(comandos_recebidos)} TCs."))
                        for cmd in comandos_recebidos:
                            prioridade = cmd.get('prioridade', 99)
                          item = (prioridade, self.command_insertion_counter, cmd)
                            heapq.heappush(self.command_queue, item)
                            self.command_insertion_counter += 1

                    duracao_tm_dados = comm_config['duracao_downlink_saude_fila_seg']
                    self.antenna_task_end_time = current_time + duracao_tm_dados
                   self.log_communication.append((current_time, "UPLINK_END", "TCs recebidos, iniciando downlink de dados."))


        # Se a antena está IDLE, checa se alguma estação quer falar
        if self.status_antenna == "IDLE":
            # Prioritiza COPE
            if is_cope_visible and cope_state['comm_status'] == "WAITING_BEACON":
                # Estação COPE está ouvindo! Começa o Beacon.
                self.status_antenna = "DOWNLINKING_BEACON"
                self.antenna_task_end_time = current_time + comm_config['duracao_tm_beacon_seg']
                self.log_communication.append((current_time, "BEACON_START", f"Enviando beacon para COPE."))

            elif is_cope_s_visible and cope_s_state['comm_status'] == "WAITING_BEACON":
                # Estação COPE-S está ouvindo!
                self.status_antenna = "DOWNLINKING_BEACON"
```

```
                self.antenna_task_end_time = current_time + comm_config['duracao_tm_beacon_seg']
                self.log_communication.append((current_time, "BEACON_START", f"Enviando beacon para COPE-S."))

        if self.status_antenna == "DOWNLINKING_BEACON":
            tm_pacote_saude = {
                'status_saude': 'NOMINAL',
                'tipo_tm': 'BEACON_SAUDE'
            }
            # Descobre para quem enviar
            if is_cope_visible and cope_state['comm_status'] == "WAITING_BEACON":
                data_to_cope = tm_pacote_saude
            elif is_cope_s_visible and cope_s_state['comm_status'] == "WAITING_BEACON":
                data_to_cope_s = tm_pacote_saude

        # Se estamos no estado de enviar dados, preparamos o pacote de dados/fila
        elif self.status_antenna == "DOWNLINKING_DATA":
            tm_pacote_dados = {
                'comandos_na_fila': len(self.command_queue),
                'executando_comando': self.current_activity_item[2]['id_comando'] if self.current_activity_item is not None else False,
                'outputs': self.output_files
            }
            # Descobre para quem enviar
            if is_cope_visible and cope_state['comm_status'] == "DOWNLINKING":
                data_to_cope = tm_pacote_dados
            elif is_cope_s_visible and cope_s_state['comm_status'] == "DOWNLINKING":
                data_to_cope_s = tm_pacote_dados


        # Retorna o estado público atualizado
        public_state = {
            'antenna_status': self.status_antenna,
            'activity_status': self.status_activity,
            'output_files': self.output_files if self.output_files is not None else None,
            'name': self.name
        }
```

```python
        return (public_state, data_to_cope, data_to_cope_s)

    def update_state_activity(self, current_time, world_state, config_atividades):
        """
        Gerencia o "cérebro" do satélite (atividades internas).
        'world_state' é um dicionário com o estado do mundo agora,
        ex: {'is_sp_polygon_visible': True, 'is_cope_visible': False}
        'config_atividades' é o JSON 'atividades.json' carregado.
        """

        # --- LÓGICA SE ESTIVER "BUSY" ---
        if self.status_activity == "BUSY":

            # Pega o comando que está em execução
            comando_em_execucao = self.current_activity_item[2] # (prior, count, comando_obj)

            # LÓGICA DE INTERRUPÇÃO
            # Verifica se os pré-requisitos ainda valem
            prereqs_ok = self._verificar_prerequisitos(
                comando_em_execucao, current_time, world_state, check_continuous=False
            )

            if not prereqs_ok:
                # FALHA! O pré-requisito foi perdido no meio da tarefa.

                # Gerar log
                self.log_activity.append((current_time, "FALHA", f"Atividade {comando_em_execucao['id_comando']} interrompida (perda de pre-requisito)."))

                # Devolver para a fila
                heapq.heappush(self.command_queue, self.current_activity_item)

                # Retornar para "IDLE"
                self.status_activity = "IDLE"
                self.current_activity_item = None
                self.activity_task_end_time = None

```

```
                public_state = {'activity_status': self.status_activity, 'output_files': self.output_files if self.output_files is not None else None}
                return public_state # Termina o update deste "tick"

            # CHECAGEM DE TIMER
            # Se não fomos interrompidos, checamos se a tarefa terminou
            if current_time >= self.activity_task_end_time:
                # SUCESSO! Tarefa concluída.
                self.log_activity.append((current_time, "SUCESSO", f"Atividade {comando_em_execucao['id_comando']} concluída."))
                self.completed_commands.add(comando_em_execucao['id_comando'])
                if self.output_files is None:
                    self.output_files = {}
                self.output_files[comando_em_execucao['id_comando']] = comando_em_execucao['outputs']

                # Retorna para "IDLE"
                self.status_activity = "IDLE"
                self.current_activity_item = None
                self.activity_task_end_time = None

            # Se não foi interrompido e não terminou, continua BUSY (não faz nada)


        # --- LÓGICA SE ESTIVER "IDLE" ---
        if self.status_activity == "IDLE":

            # LÓGICA OPORTUNISTA
            # Tenta encontrar o trabalho de maior prioridade que PODE ser feito AGORA.

            comandos_rejeitados = []
            comando_para_executar_item = None

            while self.command_queue:
                # Pega o melhor item da fila
                item_da_fila = heapq.heappop(self.command_queue)
                comando = item_da_fila[2] # (prior, count, comando_obj)

                # Verifica os pré-requisitos para INICIAR
```

```python

                prereqs_ok = self._verificar_prerequisitos(
                    comando, current_time, world_state, check_continuous=False
                )

                if prereqs_ok:
                    # SUCESSO! Encontramos uma tarefa.
                    comando_para_executar_item = item_da_fila
                    break # Para de procurar na fila
                else:
                    # FALHA. Este comando não pode ser executado agora.
                    # Guarda na lista de rejeitados.
                    comandos_rejeitados.append(item_da_fila)

            # Devolve os comandos rejeitados para a fila
            for item in comandos_rejeitados:
                heapq.heappush(self.command_queue, item)

            # Se encontramos uma tarefa, inicia ela
            if comando_para_executar_item:
                comando = comando_para_executar_item[2]
                id_atividade = comando['id_atividade']
                # Pega a duração do catálogo de atividades
                duracao_seg = config_atividades[id_atividade]['duracao_seg']

                # Muda o estado para "BUSY"
                self.status_activity = "BUSY"
                self.current_activity_item = comando_para_executar_item
                self.activity_task_end_time = current_time + duracao_seg # Define o timer!
                self.log_activity.append((current_time, "INÍCIO", f"Iniciando atividade {comando['id_comando']}."))

        # Retorna o estado atual (para o motor do teste.py)

        public_state = {'activity_status': self.status_activity, 'output_files': self.output_files if self.output_files is not None else None}

        return public_state

```

```
    def _verificar_prerequisitos(self, comando, current_time, world_state, check_continuous):
        """
        Verifica se um comando pode ser executado ou continuar executando.
        """
        if 'pre_requisitos' not in comando or not comando['pre_requisitos']:
            return True

        for pre_req in comando['pre_requisitos']:
            is_continuous = pre_req.get('continuous', False)

            if check_continuous and not is_continuous:
                continue

            # --- CASO 1: Visibilidade de Região ---
            if pre_req['tipo'] == 'visibilidade_area_alvo':

                # 1. Pega o NOME da ROI do parâmetro do comando
                nome_roi = comando['parametros']['area_alvo']

                # 2. Constrói o nome da flag que esperamos no world_state
                flag_name = f"is_visible_{nome_roi}"

                # 3. Checa o world_state

                try:

                    if not world_state[flag_name]:
                        self.log_activity.append((current_time, "FALHA_ACTIVITY", f"Perda de visibilidade da área alvo '{nome_roi}'"))
                        return False # FALHA

                except:

                    self.log_activity.append((current_time, "FALHA_CONFIG", f"Área alvo '{nome_roi}' não definida em rois.json."))
                    return False # FALHA


            # --- CASO 2: Dependência (Outro Comando) ---
            elif pre_req['tipo'] == 'comando_concluido':
```

```
                # (Este só é checado no início)
                comando_id = pre_req['id_comando_dependencia']
                if comando_id not in self.completed_commands:
                    return False # FALHA

            # --- CASO 3: Atraso (Delay de Tempo) ---
            elif pre_req['tipo'] == 'delay':

                # Pega os parâmetros do bloco PRINCIPAL do comando
                params = comando['parametros']

                if 'delay_tempo_seg' not in params:
                    self.log_activity.append((current_time, "FALHA_CONFIG", f"Comando {comando['id_comando']} não tem parametros 'delay_tempo_seg'."))
                    return False # FALHA (JSON mal configurado)

                delay_seg = params['delay_tempo_seg']

                tempo_do_evento = self.event_timestamps[comando['id_comando']]
                if current_time < (tempo_do_evento + delay_seg):
                    return False # FALHA (ainda em espera)

            else:
                self.log_activity.append((current_time, "FALHA_CONFIG", f"Comando {comando['id_comando']} com tipo de pré-requisito não configurado."))
                return False

        # Se passou por todos os pré-reqs, está OK
        return True
```

Annex C - ground_station.py Python Code

```python
# ground_station.py
#
# Classe para armazenar dados e calcular visibilidade de uma estação de solo.

import numpy as np
import astropy.units as units
from astropy.time import Time
from astropy.coordinates import EarthLocation, AltAz, TEME,
↪  CartesianRepresentation
from collections import deque

class Ground_Station:

    def __init__(self, name: str, lat: units.Quantity, lon: units.Quantity,
                 height: units.Quantity, opening_deg: units.Quantity):
        """
        Inicializa a estação de solo.

        :param name: Nome da estação (ex: "COPE")
        :param lat: Latitude (astropy.units.Quantity)
        :param lon: Longitude (astropy.units.Quantity)
        :param height: Altitude (astropy.units.Quantity)
        :param opening_deg: Ângulo de abertura (cone a partir do ZÊNITE) em graus.
                            0° = cobertura mínima (só no zênite)
                            90° = cobertura máxima (até o horizonte)
        """
        self.name = name
        self.opening_deg = opening_deg

        # Cria e armazena o objeto EarthLocation
```

```
        self.location = EarthLocation(lat=lat, lon=lon, height=height)

        # Fila de comandos a serem ENVIADOS
        # Isto é carregado do telecomandos.json
        # Usaremos um dict mapeando satélite -> fila de comandos
        self.command_outbox = {}

        # Log de operações da estação (Requisito 5)
        # Será uma lista de tuplas: (timestamp, tipo_evento, detalhes)
        self.log_communication = []
        self.log_downlink_data = [] # Log separado para dados recebidos
        self.task_end_time = None

        # Estado de comunicação (para gerenciar a troca TM/TC)
        self.comm_status = "IDLE"

        self.last_comm_time = None


    def calculate_visibility(self, rs_km: np.ndarray, times: Time):
        """
        Calcula os flags de visibilidade e ângulos de elevação para
        uma trajetória de satélite.

        :param rs_km: Array (N, 3) de posições do satélite em TEME [km]
        :param times: Array (N) de instantes (astropy.time.Time)
        :return: (visible_flags, elevations_deg)
        """
        num_pts = len(times)
        visible_flags = np.zeros(num_pts, dtype=bool)
        elevations_deg = np.zeros(num_pts, dtype=float)

        print(f"Calculando visibilidade de {self.name}...")

        # Converte o opening_deg (Quantity) para um float simples em graus
        # para usar dentro do loop
        opening_angle_val_deg = self.opening_deg.to(units.deg).value

        for i in range(num_pts):
            r_i_km = rs_km[i]
            t_i = times[i]
```

```python

            # 1. Converte a posição TEME do satélite para um objeto AstroPy
            sat_coord = TEME(CartesianRepresentation(r_i_km * units.km),
                             obstime=t_i)

           # 2. Transforma para as coordenadas locais da estação (Altitude/Azimute)
            #    'self.location' é o EarthLocation criado no __init__
            altaz = sat_coord.transform_to(AltAz(obstime=t_i,
                                                 location=self.location))

            # 3. Extrai a elevação (0° = horizonte, 90° = zênite)
            elev = altaz.alt

            # 4. Aplica a sua lógica de "ângulo a partir do zênite"
            #    Ex: opening_deg = 10° -> elev.value deve ser >= 80°
            vis = elev.value >= (90.0 - opening_angle_val_deg)
            el = elev.to(units.deg).value

            # 5. Salva os resultados
            visible_flags[i] = vis
            elevations_deg[i] = el

        return visible_flags, elevations_deg

    def load_commands(self, all_commands_data, sat_name):
        """
        Carrega os comandos do JSON para a fila de saída desta estação.
        """
        if self.name in all_commands_data:
            # Pega a lista de comandos para este satélite
            commands_list = all_commands_data[self.name]

            # Inicializa a fila para este satélite
            self.command_outbox = deque(commands_list)

            print(f"Estação {self.name} carregou {len(commands_list)} comandos para {sat_name}.")
        else:
            # Inicializa uma fila vazia se não houver comandos
            self.command_outbox = deque()
            print(f"Estação {self.name} não tem comandos para {sat_name}.")
```

```python

    def update_state(self, current_time, sat_public_state, is_visible, data_received_from_sat, comm_config):
        """
        Gerencia a máquina de estados de comunicação da estação de solo.
        Recebe o 'sat_public_state' (um dict) do satélite.
        """

        data_to_broadcast = None

        # 1. Checagem Mestra: Perdemos a visibilidade?
        if not is_visible:
            if self.comm_status != "IDLE":
                # Se a comunicação estava ativa, ela é interrompida.
                self.comm_status = "IDLE"
                self.task_end_time = None
                self.log_communication.append((current_time, "LOSS_OF_SIGNAL", f"Visibilidade perdida com {sat_public_state['name']}."))

            public_state = {'comm_status': self.comm_status}
            return public_state, data_to_broadcast # Retorna o estado atual

        # --- Se chegamos aqui, is_visible == True ---

        if len(self.command_outbox) == 0 and self.last_comm_time is not None:
            if self.comm_status == "IDLE" and self.last_comm_time < 24*60*60 and sat_public_state['output_files'] is None:
                public_state = {'comm_status': self.comm_status}
                return public_state, data_to_broadcast # Retorna o estado atual

        # 2. Máquina de Estados

        # ESTADO: IDLE (Ocioso)
        if self.comm_status == "IDLE":
            # Se estamos ociosos e vemos o satélite,
            # iniciamos o protocolo indo para WAITING_BEACON.
            self.comm_status = "WAITING_BEACON"
            self.log_communication.append((current_time, "PASS_START", f"Visibilidade com {sat_public_state['name']} iniciada."))

        # ESTADO: WAITING_BEACON
```

```
        elif self.comm_status == "WAITING_BEACON":
            # Estamos "ouvindo". O satélite terminou seu "Olá" e
            # está pronto para nosso uplink?
            if data_received_from_sat is not None:
                # O satélite nos enviou dados!
                self.log_communication.append((current_time, "SAT_HEALTH", "Dados recebidos"))
                self.log_downlink_data.append((current_time, "SAT_HEALTH", data_received_from_sat))

            if sat_public_state['antenna_status'] == "UPLINKING":
                # Sim! Nossa vez de enviar TCs.
                self.comm_status = "UPLINKING"
                self.task_end_time = current_time + comm_config['duracao_uplink_comandos_seg']
                self.log_communication.append((current_time, "UPLINK_START", f"Enviando TCs para {sat_public_state['name']}."))


        # ESTADO: UPLINKING (Enviando TCs)
        elif self.comm_status == "UPLINKING":
            # Estamos ocupados enviando. Checamos nosso timer.
            if self.task_end_time is not None and current_time >= self.task_end_time:

                # --- A LÓGICA DE "ENTREGA" ---
                # Prepara o pacote de TCs para ser retornado
                if len(self.command_outbox) > 0:

                    # Esvazia a fila de saída para o pacote
                    data_to_broadcast = list(self.command_outbox)
                    self.command_outbox.clear()
                    self.log_communication.append((current_time, "UPLINK_DATA", f"Pacote com {len(data_to_broadcast)} TCs enviado."))

                # Nosso envio terminou.
                self.comm_status = "DOWNLINKING"
                self.task_end_time = None # Limpa o timer
                self.log_communication.append((current_time, "UPLINK_END", "TCs enviados."))

```

```python
        # ESTADO: DOWNLINKING (Recebendo Dados/Fila)
        elif self.comm_status == "DOWNLINKING":
            # Estamos "ouvindo" os dados.

            if data_received_from_sat is not None:
                # O satélite nos enviou dados!
                self.log_communication.append((current_time, "DOWNLINK_DATA", "Dados recebidos"))
                self.log_downlink_data.append((current_time, "DOWNLINK_DATA", data_received_from_sat))

            # Sabemos que terminou quando o satélite ficar IDLE.
            if sat_public_state['antenna_status'] == "IDLE":
                # Ele terminou! A comunicação foi um sucesso.
                self.comm_status = "IDLE"
                self.log_communication.append((current_time, "PASS_COMPLETE", "Comunicação concluída."))
                self.last_comm_time = current_time

        if self.comm_status == "UPLINKING":
            # Descobre para quem enviar (Adaptado para cope e cope_s)
            if is_visible and sat_public_state == "UPLINKING" and len(self.command_outbox) > 0:
                # Esvazia a fila de saída para o pacote
                data_to_broadcast = list(self.command_outbox)

        public_state = {'comm_status': self.comm_status}

        # Retorna o estado público atual da estação
        return public_state, data_to_broadcast
```

Annex D - orbit_funcs.py Python Code

```python
import numpy as np
import matplotlib.pyplot as plt
from sgp4.api import jday
from astropy.time import Time
from poliastro.bodies import Earth
import astropy.units as units
from matplotlib.animation import FuncAnimation, PillowWriter
from typing import List

def ground_station_plot(times, elevations, opening_deg):
    # =============================
    # Plot de elevação ao longo do tempo
    # =============================
    plt.figure(figsize=(8, 4))
    t_hours = (times - times[0]).to(units.hour).value
    plt.plot(t_hours, elevations, label="Elevação (graus)")
    plt.axhline(90 - opening_deg.value, color="r", linestyle="--", label="Limite de visada")
    plt.xlabel("Tempo desde epoch (h)")
    plt.ylabel("Elevação (graus)")
    plt.title("Passagens do SGDC-1 sobre INPE (SJC)")
    plt.grid(True)
    plt.legend()
    plt.tight_layout()
    plt.show()

def rv_teme_from_sgp4(satrec, when: Time):
    """
    Retorna r (km) e v (km/s) em TEME para o instante 'when' (astropy Time).
    """
    # Converte o instante para datetime UTC
    dt = when.utc.datetime
```

```

    # Converte para julian day (inteiro + fração)
    jd, fr = jday(dt.year, dt.month, dt.day,
                  dt.hour, dt.minute,
                  dt.second + dt.microsecond * 1e-6)

    # Propaga usando SGP4
    e, r, v = satrec.sgp4(jd, fr)   # <-- aqui a mudança

    if e != 0:
        raise RuntimeError(f"SGP4 retornou erro code: {e}")

    return np.array(r, dtype=float), np.array(v, dtype=float)

def true_anomaly_from_rv(r_vec, v_vec, mu):
    # Vetores numpy (km, km/s)
    r = r_vec
    v = v_vec
    r_norm = np.linalg.norm(r)
    h = np.cross(r, v)
    e_vec = (np.cross(v, h) / mu) - (r / r_norm)
    e = np.linalg.norm(e_vec)
    # f = arccos( (e · r) ), com correção de quadrante via sinal de r·v
    cosf = np.dot(e_vec, r) / (e * r_norm)
    # Numérica: limitar dom. de arccos
    cosf = np.clip(cosf, -1.0, 1.0)
    f = np.degrees(np.arccos(cosf))
    if np.dot(r, v) < 0:
        f = 360.0 - f
    return f, e

def plot_final_point_orbit(rs_km, save_path):

    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(111, projection='3d')

    # Desenha a Terra como uma esfera (raio médio WGS84 ~6371 km; Earth.R em poliastro)
    Re = Earth.R.to(units.km).value
    u_sphere = np.linspace(0, 2 * np.pi, 50)
    v_sphere = np.linspace(0, np.pi, 25)
```

```
    xs = Re * np.outer(np.cos(u_sphere), np.sin(v_sphere))
    ys = Re * np.outer(np.sin(u_sphere), np.sin(v_sphere))
    zs = Re * np.outer(np.ones_like(u_sphere), np.cos(v_sphere))
    ax.plot_surface(xs, ys, zs, alpha=0.2, linewidth=0)

    # Órbita (trajetória ao longo de um período)
    ax.plot(rs_km[:, 0], rs_km[:, 1], rs_km[:, 2], linewidth=1.5)

    # Ponto do satélite no epoch (primeira amostra)
    ax.scatter(rs_km[0, 0], rs_km[0, 1], rs_km[0, 2], s=50)

    # Eixos e enquadramento
    max_range = np.max(np.linalg.norm(rs_km, axis=1))
    lim = max(1.2 * Re, 1.05 * max_range)
    ax.set_xlim(-lim, lim)
    ax.set_ylim(-lim, lim)
    ax.set_zlim(-lim, lim)
    ax.set_box_aspect([1, 1, 1])

    ax.set_title("SGDC-1 - Órbita 3D (ECI/TEME) e posição no epoch")
    ax.set_xlabel("x (km)")
    ax.set_ylabel("y (km)")
    ax.set_zlabel("z (km)")
    plt.tight_layout()
    plt.savefig(save_path, dpi=200)

def true_anomaly_evolution(times, f_deg, save_path):

    # ======= Evolução temporal do verdadeiro ângulo (true anomaly) =======
    fig2 = plt.figure(figsize=(8, 4))
    t_hours = (times - times[0]).to(units.hour).value
    plt.plot(t_hours, f_deg, linewidth=1.5)
    plt.xlabel("Tempo desde o epoch (h)")
    plt.ylabel("Verdadeiro ângulo f (graus)")
    plt.title("Evolução temporal do verdadeiro ângulo - SGDC-1")
    plt.grid(True)
    plt.tight_layout()

    # plt.show()
    # plt.savefig(save_path, dpi=200)

```

```

def trajectory_gif_generation(rs_km, save_path):

    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(111, projection="3d")

    # ===== 1) Desenha a Terra =====
    Re = Earth.R.to(units.km).value
    u_sphere = np.linspace(0, 2 * np.pi, 50)
    v_sphere = np.linspace(0, np.pi, 25)
    xs = Re * np.outer(np.cos(u_sphere), np.sin(v_sphere))
    ys = Re * np.outer(np.sin(u_sphere), np.sin(v_sphere))
    zs = Re * np.outer(np.ones_like(u_sphere), np.cos(v_sphere))
    ax.plot_surface(xs, ys, zs, alpha=0.2, linewidth=0)

    # ===== 2) Órbita completa =====
    ax.plot(rs_km[:, 0], rs_km[:, 1], rs_km[:, 2], "k--", linewidth=1)

    # ===== 3) Ponto do satélite (que será animado) =====
    sat_point, = ax.plot([], [], [], "ro", markersize=6)

    # Configuração dos eixos
    max_range = np.max(np.linalg.norm(rs_km, axis=1))
    lim = max(1.2 * Re, 1.05 * max_range)
    ax.set_xlim(-lim, lim)
    ax.set_ylim(-lim, lim)
    ax.set_zlim(-lim, lim)
    ax.set_box_aspect([1, 1, 1])

    ax.set_title("Evolução orbital do SGDC-1")
    ax.set_xlabel("x (km)")
    ax.set_ylabel("y (km)")
    ax.set_zlabel("z (km)")

    # ===== 4) Função de inicialização =====
    def init():
        sat_point.set_data([], [])
        sat_point.set_3d_properties([])
        return sat_point,

    # ===== 5) Função de atualização a cada frame =====
```

```
    def update(frame):
        x, y, z = rs_km[frame]
        sat_point.set_data([x], [y])
        sat_point.set_3d_properties([z])
        return sat_point,

    # ===== 6) Cria a animação =====
    frames = len(rs_km)
    ani = FuncAnimation(fig, update, frames=frames,
                        init_func=init, blit=True, interval=100)

    # ===== 7) Salva como GIF =====
    ani.save(save_path, writer=PillowWriter(fps=20))
    # plt.show()

# === Função para converter ângulo de cobertura em raio de visada na superfície ===
def ground_station_range(opening_deg, altitude_km):
    """
    Retorna a distância angular (em graus) entre o subponto do satélite e o limite
    ↪  do cone de visada.
    """
    Re = 6371.0  # raio médio da Terra [km]
    elev = np.radians(90 - opening_deg)
    return np.degrees(np.arccos(Re / (Re + altitude_km) * np.cos(elev)) - elev)

# ============================
# Raio angular do footprint a partir do cone do sensor (off-nadir)
# ============================
def sensor_ground_range(opening_deg, altitude_km):
    """
    Converte a semiabertura do sensor (off-nadir, em graus) no ângulo central (,
    ↪  em graus)
    entre o subponto e a borda do footprint na superfície.
    Geometria exata (sem aproximação plana).
    """
    Re = 6371.0  # km
    Rs = Re + altitude_km
    a = np.radians(opening_deg)

    # cos() obtido da solução fechada do triângulo O-S-P
    # Escolha do ramo garante =0 quando a=0.
```

```
    k = (Rs / Re) * np.sin(a)
    # Proteção numérica: para aberturas muito grandes k poderia chegar >1
    # (não é o caso de a=2°, mas deixamos robusto).
    inside = 1.0 - k**2
    inside = np.where(inside < 0.0, 0.0, inside)

    cos_psi = - (Rs / Re) * (np.sin(a)**2) + np.cos(a) * np.sqrt(inside)
    cos_psi = np.clip(cos_psi, -1.0, 1.0)
    psi_deg = np.degrees(np.arccos(cos_psi))
    return psi_deg  # em graus


def plotar_status_simulacao(lista_booleanos: List[bool], vetor_tempo: List[float], visible_object: str, save_path: str):
    """
    Gera um gráfico de linha (degrau) a partir de uma lista de booleanos e um vetor de tempo.

    O eixo Y é 1 para True e 0 para False.
    O eixo X é o vetor de tempo fornecido.

    Args:
        lista_booleanos (List[bool]): A lista de status (True/False) por epoch.
        vetor_tempo (List[float]): A lista de valores de tempo (epochs).
    """

    # 1. Validação de entrada
    if len(lista_booleanos) != len(vetor_tempo):
        raise ValueError("A lista de booleanos e o vetor de tempo devem ter o mesmo tamanho.")

    # 2. Converter a lista de booleanos para 0s e 1s
    # (True vira 1, False vira 0)
    valores_y = [1 if status else 0 for status in lista_booleanos]

    # 3. Criar a figura e os eixos
    plt.figure(figsize=(12, 5)) # Define um bom tamanho para o gráfico

    # 4. Gerar o gráfico
    # Usamos 'plt.step' pois é melhor para visualizar mudanças de estado discretas.
    # 'where='post'' significa que o valor muda após o ponto de tempo.
```

```
    plt.step(vetor_tempo, valores_y)

    # Adiciona "bolinhas" em cada ponto de dados para clareza
    plt.plot(vetor_tempo, valores_y, 'o', color='red', markersize=4)

    # 5. Customizar o gráfico
    plt.title(f'Visibilidade de {visible_object} com o tempo')
    plt.xlabel('Tempo (s)')
    plt.ylabel('Status')

    # Define os "ticks" (marcas) do eixo Y para serem exatamente 0 e 1,
    # e coloca os rótulos "False" e "True" para clareza.
    plt.yticks([0, 1], labels=['Not Visible', 'Visible'])

    # Define os limites do eixo Y para dar um respiro visual
    plt.ylim(-0.1, 1.1)

    # Adiciona um grid (grade) horizontal para facilitar a leitura
    plt.grid(axis='y', linestyle='--', alpha=0.7)

    plt.legend()
    plt.tight_layout() # Ajusta o layout para evitar sobreposição de texto

    # 6. Mostrar o gráfico
    # plt.show()
    plt.savefig(save_path, dpi=200)

```

Annex E - ground_station_funcs.py Python Code

```python
import numpy as np
import matplotlib.pyplot as plt

import astropy.units as units
from astropy.coordinates import EarthLocation, TEME, CartesianRepresentation, ITRS
import orbit_funcs

import cartopy.feature as cfeature
import cartopy.crs as ccrs

try:
    from satellite import Satellite
    from ground_station import Ground_Station
except ImportError:
    # Evita falha se os arquivos ainda estiverem sendo movidos
    pass

# ============================
# Geodésica direta para deslocar ponto (lat,lon) por um ângulo  com azimute
# ============================
def fwd_geodesic(lat_deg, lon_deg, bearing_deg, delta_deg):
    """
    Avança sobre a esfera de raio unitário:
    - lat_deg, lon_deg: ponto de partida (graus)
    - bearing_deg: rumo (graus, 0=N, 90=E)
    - delta_deg: distância angular (graus)
    Retorna (lat2_deg, lon2_deg)
    """
    1 = np.radians(lat_deg)
    1 = np.radians(lon_deg)
```

```
      = np.radians(bearing_deg)
      = np.radians(delta_deg)

    sin1, cos1 = np.sin(1), np.cos(1)
    sin,  cos  = np.sin(),  np.cos()
    sin,  cos  = np.sin(),  np.cos()

    sin2 = sin1 * cos + cos1 * sin * cos
    2 = np.arcsin(np.clip(sin2, -1.0, 1.0))

    y = sin * sin * cos1
    x = cos - sin1 * sin2
    2 = 1 + np.arctan2(y, x)

    lat2 = np.degrees(2)
    lon2 = (np.degrees(2) + 540) % 360 - 180  # normaliza para [-180,180]
    return lat2, lon2

# Rumos do ground track (azimute geodésico ponto a ponto)
def track_bearings(lat_deg, lon_deg):
    lat = np.radians(lat_deg)
    lon = np.radians(lon_deg)
    dlon = np.diff(lon)
    # ajusta saltos de ±360°
    dlon = (dlon + np.pi) % (2*np.pi) - np.pi

    lat1, lat2 = lat[:-1], lat[1:]
    # Fórmula direta do azimute (bearing) ponto1->ponto2
    y = np.sin(dlon) * np.cos(lat2)
    x = np.cos(lat1)*np.sin(lat2) - np.sin(lat1)*np.cos(lat2)*np.cos(dlon)
    brg = (np.degrees(np.arctan2(y, x)) + 360) % 360
    # repete o último rumo para manter mesmo tamanho do vetor
    brg = np.append(brg, brg[-1] if brg.size>0 else 0.0)
    return brg

def ground_track_plot(satellite: Satellite,
                        ground_stations: list[Ground_Station],
                        save_path:str):
    """
    Plota o ground track de UM satélite e os cones de visada de
    VÁRIAS estações de solo.
```

```python

    :param satellite: Objeto Satellite (já propagado)
    :param ground_stations: Lista de objetos Ground_Station
    :param sat_sensor_opening_deg: Semi-abertura do sensor do satélite (off-nadir)
    """

    # === 1. Pega os dados de trajetória do objeto Satélite ===
    rs_km = satellite.rs_km
    times = satellite.times
    sensor_opening_deg = satellite.sensor_opening_deg

    if rs_km is None or times is None:
        print(f"Erro: Satélite {satellite.name} não foi propagado. Chame .propagate() primeiro.")
        return

    # === 2. Converte a órbita (TEME) para coordenadas terrestres (ITRS → geodésicas) ===
    lats = []
    lons = []
    alts = []

    for r_vec, t in zip(rs_km, times):
        teme_coord = TEME(CartesianRepresentation(r_vec * units.km), obstime=t)
        itrs = teme_coord.transform_to(ITRS(obstime=t))
        # Usamos EarthLocation.from_geocentric para converter ITRS (X,Y,Z) para (Lat,Lon,Alt)
        location = EarthLocation.from_geocentric(itrs.x, itrs.y, itrs.z)
        lats.append(location.lat.deg)
        lons.append(location.lon.deg)
        alts.append(location.height.to(units.km).value)

    lats = np.array(lats)
    lons = np.array(lons)
    alts_km = np.array(alts)

    # === 3. CALCULA A ALTITUDE MÉDIA (Conforme solicitado) ===
    avg_alt_km = np.mean(alts_km)
    print(f"Altitude média de {satellite.name} para plotagem: {avg_alt_km:.2f} km")

    # === 4. Raio de cobertura de visada do satélite (Swath) ===
```

```
    bearings = track_bearings(lats, lons)

    # Nota: O swath do sensor USA a altitude INSTANTÂNEA (alts_km),
    psi_sat_list = orbit_funcs.sensor_ground_range(
        sensor_opening_deg.value,
        alts_km  # Usa o array de altitudes instantâneas
    )

    lats_left,  lons_left  = [], []
    lats_right, lons_right = [], []
    for , , brg, psi in zip(lats, lons, bearings, psi_sat_list):
        latL, lonL = fwd_geodesic(, , (brg - 90.0) % 360, psi)
        latR, lonR = fwd_geodesic(, , (brg + 90.0) % 360, psi)
        lats_left.append(latL);  lons_left.append(lonL)
        lats_right.append(latR); lons_right.append(lonR)

    # === 5. Plot no mapa ===
    fig = plt.figure(figsize=(10, 6))
    ax = plt.axes(projection=ccrs.PlateCarree())
    ax.set_global()
    ax.add_feature(cfeature.LAND, zorder=0, edgecolor='black', facecolor='lightgray')
    ax.add_feature(cfeature.BORDERS, linewidth=0.5)
    ax.add_feature(cfeature.COASTLINE, linewidth=0.5)
    ax.gridlines(draw_labels=True, linewidth=0.3)

    # Plot da trajetória (usa o nome do satélite)
    ax.plot(lons, lats, color='blue', linewidth=1.2, transform=ccrs.PlateCarree(),
            label=f'Trajetória {satellite.name}')

    # Plot das bordas do swath
    sensor_deg_val = sensor_opening_deg.value
    ax.plot(lons_left,  lats_left,  linestyle=':', linewidth=1.0, color='blue',
            transform=ccrs.PlateCarree(), label=f'Swath sensor ±{sensor_deg_val:.1f}°')
    ax.plot(lons_right, lats_right, linestyle=':', linewidth=1.0, color='blue',
            transform=ccrs.PlateCarree()) # Sem label duplicado

    # Marca o subponto inicial
    ax.scatter(lons[0], lats[0], color='red', s=30, transform=ccrs.PlateCarree(),
               label='Posição inicial')
```

```python

    # === 6. Loop nas Estações de Solo (Refatorado) ===
    # Itera sobre a LISTA de objetos Ground_Station
    for station in ground_stations:
        # Pega os dados do objeto 'station'
        lat_val = station.location.lat.to(units.deg).value
        lon_val = station.location.lon.to(units.deg).value
        opening_deg_val = station.opening_deg.to(units.deg).value

        # Marca a estação de solo
        ax.scatter(lon_val, lat_val, color='green', s=60, marker='^',
                   transform=ccrs.PlateCarree(), label=f'Estação {station.name}')

        # Calcula o cone da estação usando a altitude MÉDIA do satélite (avg_alt_km)
        # e o opening_deg específico desta estação
        Rcov_deg = orbit_funcs.ground_station_range(opening_deg_val, avg_alt_km)

        # Desenha o cone de cobertura (círculo de visada)
        theta = np.linspace(0, 2 * np.pi, 200)
        circle_lat = lat_val + Rcov_deg * np.cos(theta)
        # Correção para longitudes em altas latitudes
        circle_lon = lon_val + Rcov_deg * np.sin(theta) / np.cos(np.radians(lat_val))

        ax.plot(circle_lon, circle_lat, 'g--', linewidth=1.0, transform=ccrs.PlateCarree(),
                label=f'Cone {station.name} ({opening_deg_val:.0f}° Zênite)')

    # Personalização do mapa (usa o nome do satélite)
    ax.set_title(f"Projeção da trajetória de {satellite.name} e Estações de Solo", fontsize=11)
    ax.legend(loc='lower left', fontsize='small')

    plt.tight_layout()
    plt.savefig(save_path, dpi=200)


```

Annex F - rois.json Input File


```json
{
  "SP": {
    "polygon": [ [-23.4, -46.5], [-23.4, -46.8], [-23.7, -46.8], [-23.7, -46.5] ]
  }
}
```

Annex G - commands.json Input File

```
{
  "COPE": [
    {
      "id_comando": "TC001_MANUTENCAO",
      "id_atividade": "manutencao_bateria",
      "prioridade": 1,
      "parametros": {
        "delay_tempo_seg": 3600
      },
      "pre_requisitos": [
        {
          "tipo": "delay",
          "continuous": false
        }
      ],
      "outputs": [
        "manutencao.csv"
      ]
    },
    {
      "id_comando": "TC002_FOTO_SP",
      "id_atividade": "foto",
      "prioridade": 2,
      "parametros": {
        "area_alvo": "SP"
      },
      "pre_requisitos": [
        {
          "tipo": "visibilidade_area_alvo",
          "continuous": true
        }
      ],
```

```
      "outputs": [
        "TC001_FOTO_SP.png"
      ]
    },
    {
      "id_comando": "TC003_AJUSTE",
      "id_atividade": "ajuste_orbita",
      "prioridade": 3,
      "parametros": { "direcao": "NORTE" },
      "pre_requisitos": [],
      "outputs": [
        "ajuste_orbita_norte.csv"
      ]
    }
  ]
}
```

Annex H - comm_config.json Input File


```json
{
  "duracao_tm_beacon_seg": 3,
  "duracao_uplink_comandos_seg": 10,
  "duracao_downlink_saude_fila_seg": 10,
  "duracao_downlink_foto_seg": 30
}
```

Annex I - sat_actions.json Input File

```json
{
  "foto": {
    "descricao": "Aponta o sensor para um alvo e captura uma imagem.",
    "duracao_seg": 240
  },
  "manutencao_bateria": {
    "descricao": "Executa ciclo de manutenção da bateria.",
    "duracao_seg": 1800
  },
  "ajuste_orbita": {
    "descricao": "Aciona propulsores para correção de órbita.",
    "duracao_seg": 5000
  }
}
```

FOLHA DE REGISTRO DO DOCUMENTO

1. CLASSIFICAÇÃO/TIPO	2. DATA	3. DOCUMENTO Nº	4. Nº DE PÁGINAS
TC	19 de novembro de 2025	DCTA/ITA/TC-119/2025	96

5. TÍTULO E SUBTÍTULO:

Modeling and Simulation of the Operation of PESE Space Systems program

6. AUTOR(ES):

Lucas Balen Cardozo

7. INSTITUIÇÃO(ÕES)/ÓRGÃO(S) INTERNO(S)/DIVISÃO(ÕES):

Instituto Tecnológico de Aeronáutica – ITA

8. PALAVRAS-CHAVE SUGERIDAS PELO AUTOR:

PESE; Modelagem; Simulação

9. PALAVRAS-CHAVE RESULTANTES DE INDEXAÇÃO:

Satélites; Monitoramento; Simulação computadorizada; Modelagem (processos); Planejamento de tarefas (robótica); Python (linguagem de ptogramação); Computação; Engenharia aeroespacial.

10. APRESENTAÇÃO: **(X) Nacional** ( ) **Internacional**

ITA, São José dos Campos. Curso de Graduação em Engenharia Aeroespacial. Orientador: Prof. Dr. Lucas Oliveira Barbacovi; Coorientador: Prof. Dr. Christopher Schneider Cerqueira. Publicado em 2025.

11. RESUMO:

O Programa Estratégico de Sistemas Espaciais (PESE) visa fortalecer a soberania brasileira, desenvolvendo constelações de satélites para comunicações seguras e monitoramento. Este trabalho propõe e valida o desenvolvimento de um simulador computacional em Python para modelar e simular a operação integrada desses sistemas. A arquitetura do simulador separa a simulação física, de caráter intensivo, da lógica de eventos. A dinâmica orbital é pré-calculada utilizando o propagador analítico SGP4, a partir de dados TLE (Two-Line Element), para estabelecer as trajetórias e determinar as janelas de visibilidade entre satélites, estações de solo e regiões de interesse. O núcleo da simulação operacional é gerenciado por Máquinas de Estado Finitas (FSMs) que governam a lógica de comunicação e a execução autônoma de atividades no satélite. O simulador implementa um módulo de registros e uma fila de prioridade (heapq) para telecomandos, capaz de validar pré-requisitos complexos, como visibilidade contínua do alvo, dependências entre tarefas e atrasos temporais. A validação foi realizada com um cenário de 24 horas do satélite geoestacionário SGDC-1. Os resultados comprovaram o sucesso do ciclo operacional: os comandos foram executados na ordem de prioridade correta e todos os pré-requisitos foram respeitados. A simulação também revelou um comportamento emergente de ”polling” da estação de solo, que re-iniciou a comunicação proativamente para realizar o downlink de dados gerados por tarefas concluídas. O trabalho entrega uma plataforma de simulação flexível, parametrizada por arquivos JSON, e validada, servindo como uma fundação robusta para análises de missão e futuros desenvolvimentos, como a modelagem de subsistemas e a integração com o laboratório CONCEPTIO.

12. GRAU DE SIGILO:

(X) OSTENSIVO ( ) **RESERVADO** ( ) **SECRETO**