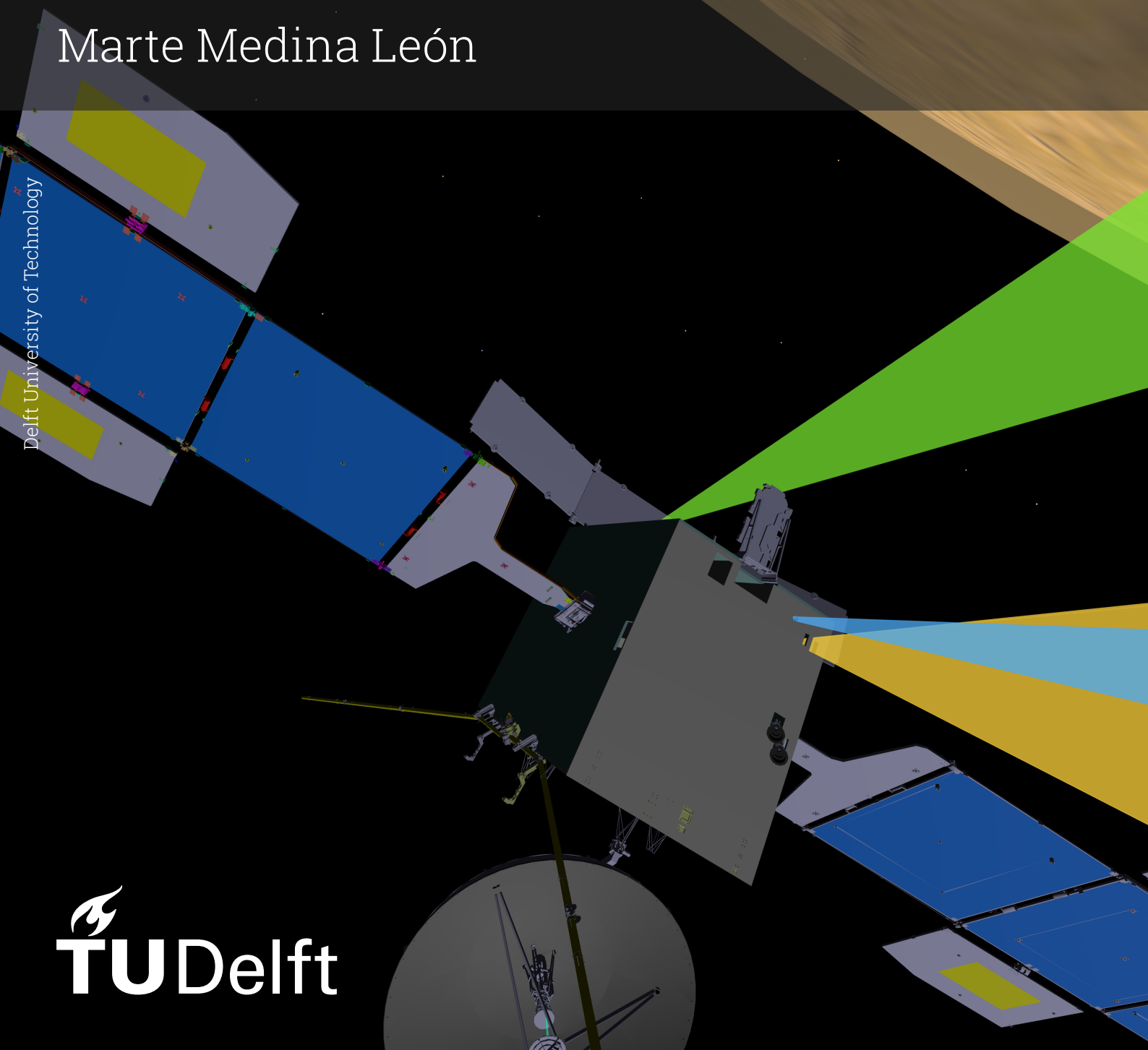
A Mission-Agnostic Spacecraft System Simulator Toolkit:

Development and Implementation for the ESA EnVision Mission

Marte Medina León



Delft University of Technology

TU Delft

A Mission-Agnostic Spacecraft System Simulator Toolkit:

Development and Implementation for the ESA EnVision Mission

by

Marte Medina León

to obtain the degree of Master of Science at the Delft University of Technology,
to be defended on the 16th of December, 2025

Student number:	5293022
Daily Supervisor:	Dr. A.E. (Ana) Rugina
TU Delft Supervisors:	ir. J. (Johan) Vennekens
	Dr.ir. J. (Jasper) Bouwmeester
Chair:	Dr. S. (Stefano) Speretta
External Examinor:	Dr.ir. P.P. (Prem) Sundaramoorthy
Project Duration:	February – December, 2025
Faculty:	Faculty of Aerospace Engineering, Delft
Cover:	WebVision application

An electronic version of this thesis is available at https://repository.tudelft.nl/

TU Delft esa

Preface

If I travelled back in time, and told 18-year-old me that we would finish our studies as Aerospace Engineer contributing to a mission to Venus, he'd probably laugh noting the irony of my name. Yet, it turns out that Venus, out of all planets, has become both the final destination of my TU Delft journey and the beginning of my career in space. After these challenging months, it is deeply rewarding to present an outcome I am proud of, and one that I see contribute in some way to Europe's deep-space exploration.

First, I would like to send my sincere thanks to Johan Vennekens for taking on this subject with me and supervising the first part of this thesis project. My gratitude also goes to Jasper Bouwmeester for taking over the torch and guiding me through to the finish line. Both of your guidance kept me grounded and led me on the right track for a successful completion of the project, I am grateful for your support.

I am thankful to my family for supporting me in pursuing these studies, and encouraging every opportunity that has come my way. To my friends, I thank all of you for providing the much-needed distraction during stressful times, pulling me out the thesis bubble now and then. My deepest thanks goes to Lauren, for always staying close no matter how far I ventured.

Naturally, this thesis would not have existed without the supervision of Ana Rugina. I would like to express my sincere gratitude, Ana, for accommodating the unforeseen roadblocks, calming my stressful moments, and putting your trust in me throughout this internship. Your support has been invaluable, and I am forever thankful for the opportunity in contributing to the EnVision mission. This gratitude extends to all colleagues at ESTEC, who have gone out of their way to help me find answers and provide guidance.

To the EnVision team: thank you for all the question-answering, the work-talk-free lunches, and generally making me feel welcome in the team from day one. It goes to show how a mission to the hottest planet in our solar system, creates the warmest team spirit. It is such a privilege to start my graduate traineeship in such an exceptional team.

To Venus!

Marte Medina León
Leiden, January 2026

Summary

Within the context of scientific planetary missions – characterized by multi-instrument systems with demanding operational timelines – there is a need for more detailed, system-level analyses grounded in simulation rather than simplified analytical sizing methods. Growing accessibility and proficiency in high-level programming are shifting the paradigm from monolithic software solutions to bespoke script-based analysis in the spacecraft systems engineering process.

This thesis presents the development, implementation and evaluation of **PAS3**: a **P**ython toolkit for **A**ccessible **S**PICE-based **S**pacecraft **S**imulations. PAS3 streamlines the creation of ad-hoc, simulation-backed models for discipline-specific analysis, for any planetary mission in its implementation phase. It integrates Andrew Annex's SpiceyPy, a procedural wrapper of NASA/JPL's NAIF SPICE toolkit, inside an object-oriented framework extending its accessibility to non-developers.

The PAS3 package provides mission-agnostic class blueprints to generate objects tied to a central simulation environment. These objects (as well as the simulation object) are to be imported in the user's analysis scripts, where custom (i.e. mission-specific) models can be developed and ran within the mission simulation context. The classes provided by the toolkit are summarized below:

`pas3.Simulation`:	Main object loading SPICE kernels, extracting coverage, and storing and controlling the simulation environment (time and global settings). Updates the assigned actors' states at time changes.
`pas3.Spacecraft`:	Spacecraft object representing a spacecraft actor in the simulation. Retrieves its state (and attitude if applicable) from SPICE kernels and provides state-driven functions for the user.
`pas3.CelestialBody`:	Celestial body object representing any celestial body defined in loaded SPICE kernels. Assigned as central body to spacecraft actors and provides utility functions (accessing body properties, relative states and body-fixed frame transformations).
`pas3.GroundStation`:	Parent class for ground station objects, with `pas3.SpiceGroundStation` (SPICE-based configuration) and `pas3.CustomGroundStation` (Custom configuration) subclasses. Assignable to individual spacecraft, providing visibility and relative positioning functions with configurable visibility masks.
`pas3.Geometry`:	Parent class for geometry objects that may represent infinite geometrical regions in space fixed to the spacecraft. Allows the creation of SPICE-based instrument FOV's (`pas3.SpiceInstrument`) or user-defined instrument FOV's (`pas3.CustomInstrument`), or an aggreggation of these geometries (`pas3.AssembledGeometry` & `pas3.AssembledInstrument`). The Geometry classes provide external incidence angle calculations, visibility and surface hit functions.

The toolkit was developed within the EnVision project team at ESA/ESTEC as part of a student internship to create an interactive 3D mission visualization framework of the scientific mission to Venus. The toolkit was integrated in models for EnVision's attitude, instrument and platform thermals, and power generation, supporting sizing analyses for the mission. Additionally, a web-based 3D mission visualization application – *WebVision* – was developed, using PAS3 as its simulation backend.

The division between mission-agnostic simulation traits and user-defined model development allows for separation of concerns, allowing the system engineer to focus on their application-specific models while relying on the toolkit's mission configuration and simulation management. This provides the user with agency over their required solutions, and generally increases the accessibility to system-level spacecraft simulations.

PAS3 is an open-source package under the Permissive European Space Agency – ESA Software Community Licence and is available at `https://essr.esa.int/project/pas3`.

Contents

List of Figures

List of Tables

Code Listings

Nomenclature

Abbreviations

Abbreviation	Definition
AIV/T	Assembly, Integration and Verification/Testing
AOCS	Attitude and Orbit Control System
AOI	Angle of Incidence
API	Application Programming Interface
AR	Acceptance Review
AU	Astronomical Unit
CaC	Cost at Completion
Cal/Val	Calibration and Validation
CDF	Concurrent Design Facility
CDR	Critical Design Review
CK	C-matrix Kernel
CNES	Centre National d'Études Spatiales
comms.	Communications
Config.	Configuration
CRA	Contamination Rejection Angle
CRR	Commissioning Results Review
deg	Degrees
DEM	Digital Elevation Model
DOM	Document Object Model
DSK	Digital Shape Kernel
ECSS	European Cooperation for Space Standardization
ELR	End of Life Review
EM	Engineering Model
EPS	Electrical Power System
EPS	Experiment Planning System
ESA	European Space Agency
ESAC	European Space Astronomy Centre
ESH	Equivalent Sun Hours
ESOC	European Space Operations Centre
ESTEC	European Space research and Technology Centre
ESTRACK	European Space Tracking
ET	Ephemeris Time
Excl.	Exclusion
FD	Flight Dynamics
FK	Frame Kernel
FM	Flight Model
FOV	Field of View
GMAT	General Mission Analysis Tool
GS	Ground Station
GUI	Graphical User Interface
H	Altitude
H	Horizontal
HGA	High Gain Antenna
HH	Horizontal transmit and Horizontal receive
HIL	Hardware-in-the-Loop
HV	Horizontal transmit and Vertical receive
IAU	International Astronomical Union
ICRF	International Celestial Reference Frame
IDE	Integrated Development Environment

Abbreviation	Definition
IDM–CIC	Integrated Design Model – Centre d'Ingénierie Concourante
IK	Instrument Kernel
ILS	Instrument Lead Scientists
IOC	In-Orbit Commissioning
IOV	In-Orbit Verification
IR	Infrared
ITRF	International Terrestrial Reference Frame
J2000	12:00 on January 1, 2000 Barycentric Dynamical Time (TDB)
J2000	SPICE name for the International Celestial Reference Frame (ICRF)
JPL	Jet Propulsion Laboratory
JS	JavaScript
LEOP	Launch and Early Operations Phase
LSK	Leap Seconds Kernel
M&S	Modelling and Simulation
MAPPS	Mission Analysis and Payload Planning System
MAR	Mission Adoption Review
MCR	Mission Close-out Review
MDR	Mission Definition Review
MK	Meta-Kernel
MLI	Multi-Layer Insulation
MOC	Mission Operations Centre
MSR	Mission Selection Review
NAIF	Navigation and Ancillary Information Facility
NASA	National Aeronautics and Space Administration
NIR	Near Infrared
NRA	Negative Roll Angle
occ.	Occultation
OEM	Orbit Ephemeris Message
OOP	Object-Oriented Programming
PAS3	Python toolkit for Accessible SPICE-based Spacecraft Simulations
PASEOS	PAseos Simulates the Environment for Operating multiple Spacecraft
PCK	Planetary Constants Kernel
PDR	Preliminary Design Review
PDU	Power Distribution Unit
PRA	Positive Roll Angle
PRR	Preliminary Requirements Review
PTB	Project Test Bed
QAR	Qualification and Acceptance Review
QM	Qualification Model
QR	Qualification Review
RAAN	Right Ascension of the Ascending Node
rad	Radians
RMSE	Root Mean Square Error
RoI	Region of Interest
RSE	Radio Science Experiment
s	seconds
SAA	Solar Array Assembly
SAR	Synthetic Aperture Radar
SciOps	Science Operations
SCLK	Spacecraft Clock Kernel
SCT	Spacecraft Control Toolbox
SE	System Engineering
SIL	Software-in-the-Loop
SOC	Science Operations Centre
SOLab	Solar System Science Laboratory
SPICE	Spacecraft Planet Instrument C-matrix Events
SPK	Spacecraft Planet Kernel

Abbreviation	Definition
SRR	System Requirements Review
SRS	Subsurface Radar Sounder
SS	System Simulator
SSC	System Simulation Concept
SSMM	Solid State Mass Memory
STD. SAR	Standard SAR observation mode
STK	Systems Tool Kit
STM	Structural Thermal Model
SWT	Science Working Team
Tbits	Terabits
TDB	Barycentric Dynamical Time
TRL	Technology Readiness Level
TT&C	Telemetry, Tracking and Command
TUDat	TU Delft Astrodynamics Toolbox
UFOV	Unobstructed Field of View
UI	User Interface
UNREQ	Unrequired pointing
UTC	Coordinated Universal Time
UV	Ultraviolet
V	Vertical
VenSAR	Venus Synthetic Aperture Radar
VenSpec	Venus Spectrometer
VESPA	Venus Science Planning Assistant
VR	Virtual Reality
VSO	Venus-centric Solar Orbital frame

Symbols

Symbol	Definition	Unit
A	area	m^2
A_{array}	solar array area	m^2
ΔA	surface area	m^2
ESH	equivalent sun hours	h
λ	longitude	$^\circ$
λ	separation angle between a boresight vector and the nadir vector	rad
ϕ	latitude	$^\circ$
ϕ	cross angle	rad
h	altitude	km
v	velocity	km/s
R	celestial body radius	km
r	orbit radius / distance	km
$\hat{r}$	direction unit vector	-
T_{sim}	simulation time	s
$d\Sigma$	surface element	m^2
ω_{orbit}	orbital angular velocity	rad/s
ω_{Earth}	Earth's rotation rate	rad/s
η_{min}	minimum elevation angle	rad
ϑ	reference angle	rad
ϑ	colatitude	$^\circ$
ϑ_{FOV}	angle of view of a FOV	rad
ϑ_{SRS}	SRS antenna tilt angle	rad
ϑ_{HGA}	HGA tilt angle	rad
ϑ_{VenSAR}	VenSAR ray tilt angle	rad
ϑ_{Panel}	VenSar reflectarray panel tilt angle	rad
ϑ_i	incidence angle	rad

Symbol	Definition	Unit
$\vartheta_{default}$	default VenSAR look angle	rad
ϑ_{max}	half-angle of the visible cap cone	rad
ϑ_{in}	inner cone half-angle	rad
ϑ_{out}	outer cone half-angle	rad
ξ	the angle between a local surface normal and a surface area element	rad
η	the angle between the boresight vector and the direction vector towards a surface element	rad
β	the angle between $\hat{n}$ and $\vec{v}_{sun}$	rad
β	orbit beta angle	rad
S	distance between spacecraft and a surface element	km
$\vec{v}_s$	vector pointing to surface element	-
$\vec{v}_{sun}$	Sun direction vector	-
$\hat{v}_{sun}$	unit vector pointing to the Sun (equivalent to $\vec{v}_{sun}$)	-
$\vec{v}_{earth}$	Earth direction vector	-
$\vec{v}_{nadir}$	nadir direction vector	-
$\hat{n}_{orbit}$	orbital plane normal vector	-
$\hat{n}$	Surface normal vector	-
$\hat{b}$	boresight direction vector	-
$\hat{b}_{in}$	inner cone boresight vector	-
$\hat{b}_{out}$	outer cone boresight vector	-
$\hat{v}_{proj}$	projected vector	-
q_{sol}	solar flux	W/m^2
q_{alb}	albedo flux	W/m^2
q_{IR}	planetary IR flux	W/m^2
I	solar Constant	W/m^2
α_L	apparent limb half-angle	rad
ϑ_s	elongation angle	rad
ρ	albedo coefficient	-
F_{12}	view factor	-
R^2	coefficient of determination	-
FF	fill factor	-
η_{cell}	solar cell efficiency	-
S_{shad}	shadow factor	-
P_{MPP}	maximum Power Point Power	W
P_{min}	minimum power	W
F_{aoi}	power reduction factor due to angle of incidence losses	-
$F_{sh.}$	power reduction factor due to shadowing losses	-
dt	time step	s

1
Introduction

The drive to explore our solar system has led to an array of ambitious planetary exploration missions dedicated to understanding our neighbouring planets. Still, there are many unanswered questions waiting to be uncovered. The spacecraft systems we send out to deep space to answer these questions are often multi-instrument platforms with complex operational profiles, required to meet the stringent scientific objectives in an often limited lifetime around these distant worlds. The process of simulating spacecraft systems is an essential part in the development of such missions. Being able to accurately model the operational environment and spacecraft behaviour allows the mission system engineers to evaluate design choices and assure a successful mission outcome.

Within the context of space system engineering activities for scientific interplanetary missions, there exists a need for a more accessible approach to system-level spacecraft simulation and mission contextualization starting at early implementation phase. The current status quo of Modelling & Simulation (M&S) practices of the system engineer is often characterized by either the use of monolithic, discipline-specific tools for subsystem analysis, or ad-hoc solutions implemented for specific analysis needs. These solutions often lack modularity and reusability, and are often not truly system-level due to the complexity of integrating a full simulation backend platform and cross-discipline model interactions. This results in limited flexibility to adapt to evolving mission analysis needs.

As the accessibility for non-developers to create own code-based solutions keeps increasing, the M&S paradigm is shifting towards this ad-hoc approach more and more. To follow this trend while addressing the needs and current limitations, this thesis presents the development and implementation of PAS3; a **P**ython toolkit for **A**ccessible **S**PICE-based **S**pacecraft **S**imulations. PAS3 aims to provide the system engineer with the necessary tools to build their own discipline-specific models, and easily integrate it in a baseline System Simulator (SS) environment for the mission. This provides the user with agency over the model development, while abstracting the underlying mission simulation work.

The development of PAS3 aims to address the following main research question:

"How can a Python-Based system simulator toolkit, support systems engineering activities of interplanetary scientific orbiter missions from the preliminary definition phase and onwards?"

To investigate this, a case study is performed for the European Space Agency (ESA) EnVision mission. This thesis project was conducted during a 9-month internship period at ESA's European Space Research and Technology Centre (ESTEC) within the EnVision project team. EnVision is an upcoming Venus orbiter mission, set to launch in 2031. The mission will study Venus from core to upper atmosphere, being the first mission to provide a truly holistic data stream of the planet dynamics. The mission aims to answer key questions about Venus's history, geological activity, and climate. The case study focused on scoping, designing, developing and testing the PAS3 toolkit; then deploying a general spacecraft configuration of EnVision, and using the toolkit to support the development of custom models for:

1. The spacecraft's attitude.
2. Thermal models for some of its instruments.
3. A thermal model to generate inputs for EnVision's MLI thermo-optical degradation models.
4. A Power model considering the spacecraft's attitude, panel steering and self-shadowing effects.

Additionally, a separate implementation of the EnVision PAS3 configuration is performed to address the need for increased contextualization and understanding of the mission by the spacecraft system engineers. This implementation focuses on creating a 3D visualization of the EnVision spacecraft in its orbital environment during the mission's science phase. The resulting visualisation web-application *WebVision* was developed, with

PAS3 as its simulation backend. This showcased the interfacing potential of the toolkit for a larger software ecosystem.

The thesis report is structured as follows: First, chapter 2 will give the necessary background surrounding scientific mission development within the agency. System engineering practices, mission phases and stakeholder roles are covered to provide necessary context for the work. It ends with the identification of specific needs in the field, which this research aims to address. Next, chapter 3 introduces the field of space M&S, and the current state-of-the-art of existing tools and practices. This chapter also defines the framework by which the resulting SS for EnVision will be evaluated (flowing down into an assessment of the underlying toolkit). Chapter 4 then presents the development methodology of the PAS3 toolkit. It covers the design philosophy, requirements and architecture of the toolkit first, ending with the verification approach.

Chapter 5 describes the actual implementation of the PAS3 toolkit for the EnVision mission case study. It first introduces the EnVision mission and spacecraft briefly, before going into the configuration approach of the SS. Then, the custom model development (enumerated on the previous page) is presented, numerical results are shown, and the models are verified. Afterwards, chapter 6 presents the original internship deliverable: the WebVision application. The chapter covers the requirements, architecture and implementation of the application, showcasing its functionalities, while highlighting the role of PAS3 as its simulation backend. Finally, chapter 7 establishes a requirements' verification round for the toolkit, and evaluates the results of the EnVision case study against the defined evaluation framework from chapter 3. This reflects the effectiveness of the toolkit in addressing the research goals, and allows the research question to be answered.

2

Background

As the title of this thesis suggests, the research will focus on the development of a toolkit that aims to help the creation of spacecraft system simulators. The development was started within the context of scientific planetary missions, more specifically of EnVision, the European Space Agency's (ESA) upcoming mission to Venus. This mission has recently entered its implementation phase, where the need for a system-level approach to sizing analysis for requirement generation and deeper mission contextualization was identified. The research in this thesis will aim to offer a method to support the systems engineering practices of scientific planetary missions in their implementation phases. This chapter will offer some fundamental insight into the broader scientific, technical, and institutional context to help clarify the motivation behind this research and its methodologies.

Section 2.1 will introduce the processes behind ESA scientific mission development. It covers the standard project phases from study to disposal, fundamental systems engineering practices and the segmentation of responsibilities among stakeholders throughout the mission development lifecycle. The chapter ends with the identification of the occurring needs for systems engineers in planetary project teams in section 2.2. The resulting need statement will form the foundation for the research objective and research questions, which will be synthesized in chapter 3.

2.1. ESA Scientific Mission Development

This section presents the general ESA development process, specifically for scientific planetary missions. In order to develop a method to support the systems engineering of such missions, it is essential to understand the context in which systems engineering activities are conducted first. subsection 2.1.1 covers the general project phases of ESA missions and what activities are performed in each phase. subsection 2.1.2 elaborates on the systems engineering practices during implementation. Finally, the different segments and stakeholders involved in a typical ESA scientific mission are discussed in subsection 2.1.3, providing an overview of the roles and responsibilities of the various entities involved in such a mission.

2.1.1. Project Phases

ESA's project development process is structured around a series of predefined phases, each with specific objectives and deliverables. These phases are designed to ensure that the mission is developed in a systematic and controlled manner, allowing for effective management of risks, costs, and schedules. This subsection will give a short overview of these phases.

ESA mostly follows the mission phases in conformance with systems engineering standards established by ECSS-E-ST-10C Rev.1, being divided into phase 0, A, B, . . . , F [1]. ESA groups these phases into two main categories: *Study Phase* (0/A/B1) and *Implementation Phase* (B2/C/D/E1). Afterwards, mission exploitation can begin [2, 3].

At the very start of every ESA scientific mission, an **expression of scientific need** is established. The scientific community is invited to propose desired experiments and mission types within the context of the agency's long-term science programs. These proposals are reviewed by discipline-specific committees and then refined by a top-level advisory committee into a set of mission concepts. During this mission concept phase (phase 0 and A), both industrial and internal studies assess technical feasibility and establish a strict cost envelope, as missions are funded through ESA's fixed mandatory programs budget. Concepts may be revised multiple times to balance scientific goals with financial and technical constraints [4].

Phase 0: Mission Definition – During Phase 0 (sometimes referred to as *Pre-Phase-A*), the system engineering function lays the groundwork for the mission. It supports the identification of customer needs and proposes possible system concepts. The goal is to ensure alignment with the *Mission Statement*, which captures the declared user needs.

In this phase, the mission statement document is also analysed and relevant contributions from lower-level suppliers is integrated into a mission description document. Additionally, it proposes initial technical requirements against the expressed user needs for agreement with the customer/stakeholders.

The phase concludes with the Mission Definition Review (MDR), where mission needs, science performance goals, safety, and operational constraints are identified, and initial technical requirements specifications are created. The feasibility is studied in a Concurrent Design Facility (CDF) campaign, generally lasting about 4 weeks [5], and resulting in a CDF Study Report to be used as a baseline for the next phase [6, 7].

Phase A: Mission feasibility – During Phase A, the needs identified in Phase 0 are refined and system solutions to meet stakeholder requirements are proposed. It supports the Mission Selection Review (MSR) preparation and validates requirements against expressed needs in collaboration with the stakeholders.

This phase involves producing initial technical designs, management plans, system engineering plans, and product assurance plans from usually two prime industrial contractors (referred to as *primes*) in a competitive fashion. Feasibility is assessed including implementation, programmatic aspects, cost, operations, organization, production, maintenance, and disposal, while risks are evaluated. In the context of a scientific mission, the scientific instruments and their respective provision by ESA member states are established, and the scientific requirements are formalized. The phase ends with the selection of the mission (in competition with other candidates) during the MSR [8, 9].

Phase B: Preliminary Definition – In Phase B, ESA takes a distinctive approach to the preliminary design phase, splitting it up into two sub-phases: B1 and B2:

- **Phase B1 –** This phase is still part of the *study phase*, but more focused on the system design. The scientific return requirements have been consolidated, and the preliminary development can start. During this phase, the system requirements and interfaces are prepared, subsystem and instrument design are defined and reviewed and a preliminary mission schedule and cost at completion is outlined. Industrial studies of essential technologies are being finalized to bring them to sufficient maturity before the implementation phase [10]. At the end, the prime contractor of the mission is selected, and the mission is officially adopted as the conclusion of the Mission Adoption Review (MAR) [3, 11].
- **Phase B2 –** This phase kicks off the *implementation phase* of the mission. It performs the preliminary design of the mission in close collaboration with the prime contractor. Within phase B2, the System Requirements Review (SRR) is performed, which functions as an assessment of preliminary performance based on conformance with system functional requirements. Evaluation of major plans concerning design & development, product assurance, and assembly integration and testing is performed. This phase ends with the Preliminary Design Review (PDR), which assesses analyses on system performance, confirms compatibility between the design and requirements, and establishes technological readiness. It also results in the approval of key project plans, and validates that the internal interfaces and verification methods are sufficiently defined to proceed to the detailed design phase [12].

Phase C: Detailed Definition – In Phase C, the detailed design of the system is finalized. The Engineering model (EM) and, if considered useful, the Structural Thermal Model (STM) are built, a user manual is set up, development testing is performed, and planning for Assembly, Integration, Verification, and Testing (AIV/T) is established. The phase concludes with the Critical Design Review (CDR). The assembly of the Qualification Model (QM) might also start throughout this phase in preparation for the next phase. The goal of this phase is to demonstrate that the system meets the technical requirements of the system technical requirements specification [8, 11].

Phase D: Qualification and Production – Phase D, Qualification and Production, involves finalizing the system design, building the Flight Model (FM), and integrating components from various European industry partners under the prime contractor. This phase includes qualification testing of the QM and acceptance testing of the FM, culminating in the Qualification Review (QR), Acceptance Review (AR), or a combined QAR. Electrical, software, navigation, and pointing tests are performed to ensure all systems function correctly and meet requirements. Additionally, preparations for operation and utilization must be finalized. Upon completion, the spacecraft is cleared for delivery to the launch site, marking readiness for operational deployment [8, 11].

Phase E: Operations and Utilization – This phase supports the operations and utilization of the system. Similarly to phase B, it is divided into two sub-phases: E1 and E2.

- **Phase E1 –** This phase includes the Launch and Early Operation Phase (LEOP), In Orbit Verification (IOV), In Orbit Commissioning (IOC), and calibration and validation campaign (Cal/Val). It ends with a Commissioning Result Review (CRR), concluding the *Implementation phase* of the project [13].

- **Phase E2 –** After the implementation phase is concluded, nominal mission operations activities of the spacecraft can start. The phase ends after End of Life Review (ELR), when mission disposal can start [8].

Phase F: Disposal – This phase is the final phase of the mission, where the spacecraft is disposed of in either a controlled, or uncontrolled manner (depending on the mission and destination). This may involve deorbiting the spacecraft or transferring it to a graveyard orbit. The phase – and consequently the mission – ends with a Mission Close-out Review (MCR).

2.1.2. Systems Engineering Practices in Implementation Phase

The research activities of this thesis will focus on supporting the systems engineering process in implementation phases of ESA scientific missions. The team overseeing the implementation phase, is called the *Project Team*. It manages the mission development, planning and top-level procurements. They assure the design conforms to its intended goals by monitoring and reviewing the development process. The systems engineering activity of the project team entails providing industry with system requirements based on the mission requirements that are the result of a consolidation of the scientific community's demands stemming from the study phase.

A systems engineer in a project team must derive a design-oriented technical solution from customer requirements through a top-down approach. This process involves multidisciplinary functional decomposition, where the system is broken down into lower-level products and resources are allocated according to established margin philosophies. At each level, verification activities are conducted to demonstrate that requirements are satisfied, providing evidence that the final product meets technical, cost, schedule, and quality objectives [1]. The systems engineering viewpoint encompasses all of these objectives. To quote [14] directly: "[The systems engineering viewpoint] seeks to look beyond the obvious and the immediate, to understand the user's problems, and the environmental conditions that the system will be subjected to during its operation". This last part is especially relevant for this thesis' goals (more in section 2.2).

ECSS-E-ST-10C Rev.1 lists the following systems engineering sub-functions [1]:

- **Requirement engineering:** requirement analysis and validation, requirement allocation, and requirement maintenance.
- **Analysis:** resolving requirements conflicts, decomposing and allocating requirements during functional analysis, and assessing system effectiveness.
- **Design and configuration:** resulting in a physical architecture, and its complete system functional, physical and software characteristics.
- **Verification:** demonstrating that the deliverables conform to the specified requirements, including qualification and acceptance.
- **Integration and control:** coordinating the various engineering disciplines and participants throughout all the project phases.

In phase B2, the start of the mission's implementation phase, the systems engineering focus lies on the preparation of the Preliminary Design Review (PDR). At this point, the general system design and interfaces of the spacecraft must be defined and ready to start its detailed development. While a general concept of the system architecture is already established in the study phase, as the subsystems and instruments are consolidated more exhaustively throughout B2, the system design is prone to changes and iterations. Implementing these subsystem and instrument inputs and evolutions into the system/mission level is the system engineering task of the project team in B2. This notion of flexibility in the spacecraft design and operations, must be kept in mind throughout the following sections and chapters, as it influences many of the design decisions discussed.

2.1.3. Mission Segments and Stakeholders

A typical space mission is divided into several segments. The spacecraft (the space segment), is launched by the launch provider (launch segment), and supported by a number of centres operating on ground (the ground segment) [1]. While the focus of this thesis lies mostly within the space segment, ignoring the interactions between the different elements could lead to uninformed decisions in the research process. Moreover, certain tools and methods discussed in section 3.1 are used/developed in different segments with their own particular applications. They might provide insight and functionalities for a project team as well. The segments and their interactions are illustrated in Figure 2.1, with the ESA EnVision mission as example. The different segments, composing a typical scientific space mission will be briefly discussed in this subsection.

The space segment consists of the spacecraft itself, which includes the platform and the scientific payloads. In a typical ESA scientific mission, the platform is provided by the prime contractor, integrating subsystem

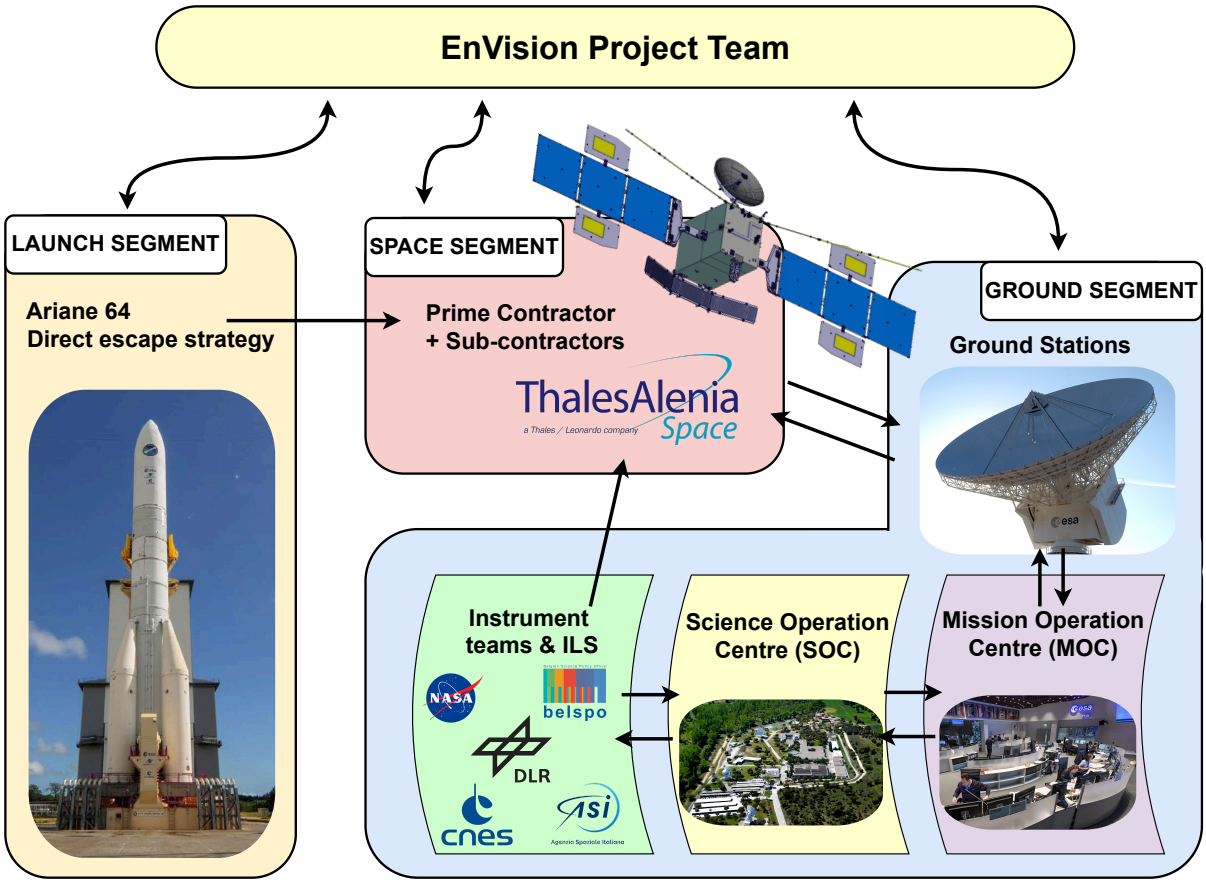


Figure 2.1: Mission segments of an ESA scientific mission (EnVision as example).

components from many different subcontractors from multiple member states [3]. The scientific instruments/ experiments are provided (and funded) by national institutes or consortia. The development of these instruments is performed in early stages of the mission development, scheduling PDR and CDR earlier than the platform's. They will eventually carry out the mission science objectives which the spacecraft platform must support. While the instruments collectively aim to reach the overarching scientific goals, they generally function independently, observing different science scopes. The consortia's interests are defended by the ILS's (Instrument Lead Scientists, previously: Principal Investigators) [15]. They must ensure that the platform and mission allocate sufficient resources and coverage to their respective instrument.

The ILS's and the science working team work closely with the Science Operations Centre (SOC) located at ESAC, Madrid. The SOC, as part of the Science Ground Segment, leads the science planning process in alignment with the mission's scientific objectives set by the Science Working Team (SWT). It coordinates the scheduling and execution of science observations. It receives predicted trajectories and ground station schedules inputs from the Mission Operations Centre (MOC), and uses these to plan science activities across long-, medium-, and short-term planning cycles. The SOC then delivers science planning products, including pointing requests and instrument telecommand files, back to the MOC for validation and uplink. The MOC, located at ESOC in Darmstadt, deals directly with the spacecraft; commanding and operating it through a network of ground stations, while providing flight dynamics data to the other centres and segments (both during operations and design phases). To summarize: the SOC acts as the central interface between the scientific community (instrument teams, ILS's and SWT) and spacecraft operations (MOC) [16, 17].

In Figure 2.1, the scientific mission's project team is in direct connection to all segments. The responsibility of the project team, is the management and coordination of the segments and their stakeholders to produce a successful mission. Scientific requirements from the ILS and SWT are translated by the project team into system requirements for the spacecraft prime contractor, system constraints from instruments and the platform are converted into launcher requirements and vice versa. While a big focus of the project team activities is on the space segment – being a one-off development generally not relying on reusable elements – the success of a scientific mission depends on the successful integration of all segments.

2.2. Identification of Needs

The design of a complex system such as a spacecraft is a challenging process. It requires a deep understanding from the systems engineer of the mission's objectives, the spacecraft's design constraints, and the operational environment. Within the context of scientific planetary missions, certain needs have been identified in the current systems engineering process. These stem from consulting with experts in the field, particularly (but not exclusively) from the EnVision project team, and from the broader need to situate the thesis within the wider field of planetary space missions. They are the following:

The need for a method allowing:

1. **Accessible system-level simulation for early analysis of sizing cases.**
2. **Mission contextualization and understanding.**

And this method should preferably be:

3. **Maintainable throughout the maturity of the mission implementation phase.**
4. **Compatible across different missions.**
5. **Usable across systems engineering disciplines.**

Referring back to the ECSS-defined sub-functions of the systems engineer in subsection 2.1.2, carrying out analysis is an essential part of the *requirements engineering* and *verification* practice. Often in the case of requirement engineering, this means the calculation of specific sizing cases to provide contractors and instrument teams with the necessary design margins, as well as verifying their analysis results in the system context (especially in early implementation phase). An effective strategy is to implement conservative assumptions in analysis calculations with high margins for generating design specifications. However, the nature of certain missions may require a more detailed modelling for sizing the spacecraft systems and mission. This is especially true for planetary missions. Their common mission objective is to maximize scientific return; this means having a platform with multiple instruments and applying heavy planning and coordination strategies to ensure sufficient coverage and data generation in a limited mission lifetime.

For example, the EnVision mission is characterized by a dynamic science operations profile, resulting in an attitude-intensive timeline (more in section 5.3) [18]. Bridging the gap between platform design and the combination of dynamic science operations and environment can be challenging without considering the mission and system in a more holistic simulation-based analysis. This is less critical for missions with a less intensive and periodic orientation, resulting in more stable calculations (take for example Earth Observation spacecraft with mostly near-nadir observation and communication), causing less uncertainty in conventional sizing analysis. However, setting up such system-level simulations in the early implementation phase (starting from B2) can be challenging, due to changing system architecture, changing operations timelines, and limited time resources for the project team systems engineers. An accessible method that allows for such analysis, already starting from early implementation, can help quantify alignment with predefined margins and help the requirement generation and verification process greatly.

When conducting system-level analyses, whether using specialized tools or general mission simulators, it can be very hard to relate temporal results within the mission context. The complexity of planetary missions, with their dynamic operational timelines and evolving spacecraft state (such as position, dynamical attitude, Sun/Earth geometry, visibility, eclipses, operational modes, and occultations), can make it difficult for systems engineers to intuitively grasp how these factors influence analytical outcomes. To support a deeper understanding of the mission, there is a need to complement less intuitive analysis results (graphs, documents, spreadsheets, . . .) with a 3D visualization of the mission. This would allow systems engineers to see the spacecraft and its environment in an intuitive way that relates analysis outputs (like environmental constraints violations) to specific mission events or configurations. This not only aids in interpreting complex results but also enhances communication among team members and stakeholders, supporting better-informed decision-making throughout the mission development process [19].

Applying an integrated (i.e. system-level) approach to modelling and simulation methods from the start of a project (preferably even before phase B2), allows a *working version* or dynamic model of the system to be set up. Initially simple, with low fidelity, and gradually increasing in complexity as the project matures [20]. However, what often times happens is that established tools' use drops off when the project enters a new design phase. This is often contributed to their limited usefulness to the new phase's needs, data compatibility issues (new phases require new inputs and outputs), and the high familiarization effort when the tool finds new users. One can argue that these issues stem from the ad-hoc approach to Modelling and Simulation (M&S) tools often

applied in longer design projects for SE analysis and verification purposes [21]. This behaviour will be further discussed in subsection 3.2.2. A sufficient level of continuity in the method's use is desirable.

This reusability can be extended to usability across different missions. The implementation and this thesis' result will be developed within the context of a single mission. However, in order to provide a relevant contribution to the field of space systems engineering of planetary missions, the method should be *mission-agnostic*. This means that the method should be applicable to other missions, not only EnVision. The method should be flexible enough to adapt to different mission requirements and constraints, while still providing the necessary functionalities for system-level analysis and verification. Furthermore, project teams are comprised of a diverse set of specialists, each with their own use cases in both system-level analysis and verification of their analysis results [1, 21]. Hence, the method should not only be compatible across phases, missions, but also across disciplines.

This chapter can be concluded in a single need statement:

> *"In the field of planetary space mission development, there exists a need for a mission-agnostic simulation method, allowing sizing analysis and visualization tailored for spacecraft systems engineers that can evolve and mature throughout a project development."*

This need statement will act as a starting point for the research activity presented in this thesis. chapter 3, will take this need statement and identify the state of the art in the field of system-level modelling and simulation methods. Shortcomings and trends will be identified within the current field, flowing into the research objective and research questions, this thesis will aim to answer.

3

Modelling & Simulation in Space Systems Engineering

The term System Modelling & Simulation (M&S) is described in ECSS-E-TM-10-21A [20] as follows: "M&S refers to the modelling and simulation activities carried out at system level, where the *system* is considered to be a spacecraft, the space segment and/or the ground segment. It covers simulation models and simulator infrastructure used to support specification, design, verification and operations of space systems." System M&S tools may support the systems engineering (SE) activities from the identification of mission concepts, early verification of feasibility, consolidation of mission and system design, to the qualification of the system prior to launch and operations. They help the iterative SE design process by enabling trade-offs, informing design decisions, and assisting in the verification and validation of mission and system designs through behavioural predictions [20, 21].

Before starting this chapter, a common understanding of the following terms must be established; while often being used interchangeably in literature, they hold different meanings in the context of this thesis.

- **Model**: A representation of a real-world system that can be used in a computational environment [22].
- **Modelling**: The construction of models as abstract representations of reality, describing (parts of) the system as a whole and demonstrating its behaviour [21].
- **Simulation**: The dynamic execution of a model, extracting the system's behaviour as a response to time-varying inputs and environmental conditions, studying the dynamic response modes [20, 21].
- **Tool**: A software application that performs a specific function or set of functions to assist in a task or process [22].
- **Toolkit**: A toolkit is a collection of tools or libraries designed to facilitate the development of software or to perform specific tasks [23].

At the end of the previous chapter, a general needs statement in planetary missions was identified. The need to analyse mission-/system-level events, calls for a system-level simulator which models the space segment (spacecraft and its environment) and its interaction with the ground segment (ground stations, data linking). Such a simulation must allow performing system analyses for the whole mission timeline and allow interfacing to a visualization framework (a distinct need from section 2.2).

Section 3.1 will present a state-of-the-art study of currently available tools which show potential in satisfying the needs identified in section 2.2. Section 3.2 will place these tools within the current landscape of system-level M&S, and investigate trends observed in the field, as well as guidelines in developing space system simulators from literature. Based on this assessment, section 3.3 will formulate the objective of the presented research, specify the research scope, and formulate the research questions.

3.1. State of the Art

Multiple tools exist that are used for simulation and visualization of spacecraft. Research was done in literature to assess the state of the art of such software and their usage in space projects. In order to provide a useful assessment of the current state of the art in M&S tools, the search domain is constrained to those showing potential in satisfying the needs identified in section 2.2, in particular allowing **3D visualization** of a spacecraft in the solar system, while showing potential functionality in performing **system analyses**. A selection is discussed in this section, with particular focus on their individual use cases. The resulting tools are clustered into five categories: Scientific data visualization, solar system renderers, science operation & planning tools, spacecraft system design tools, and Python-/Matlab-based tools.

3.1.1. Scientific Data Visualization Tools

Scientific data visualization tools are a large subset of developed software within the M&S field of scientific space missions. The tools summarized all share in common that they visualize both the spacecraft, solar system bodies, and scientific data in a 3D context. They are all open-source, which might allow the possibility of adding more analysis-focused functionality within the context of this thesis if necessary.

3DView is an interactive visualization tool developed by the French Plasma Physics Data Centre for rendering spacecraft trajectories, attitudes, and scientific data in a 3D heliospheric and planetary context. It uses SPICE kernels for positioning and orientation of spacecraft and planetary bodies, supports visualization of instrument fields of view, and integrates with large scientific databases via virtual observatory standards. Users can upload data, overlay time series, field lines, spectrograms, and simulation results, and explore analytic models of planetary boundaries and magnetic fields. Designed for both mission analysis and education, 3DView enables interpretation of space science data and is used in scientific analysis and higher education settings to interpret and visualize mission data [24]. Figure 3.1 illustrates a 3DView scene of the Venus Express spacecraft, showing the ASPERA4 instrument's spectrogram data in its spatial context.

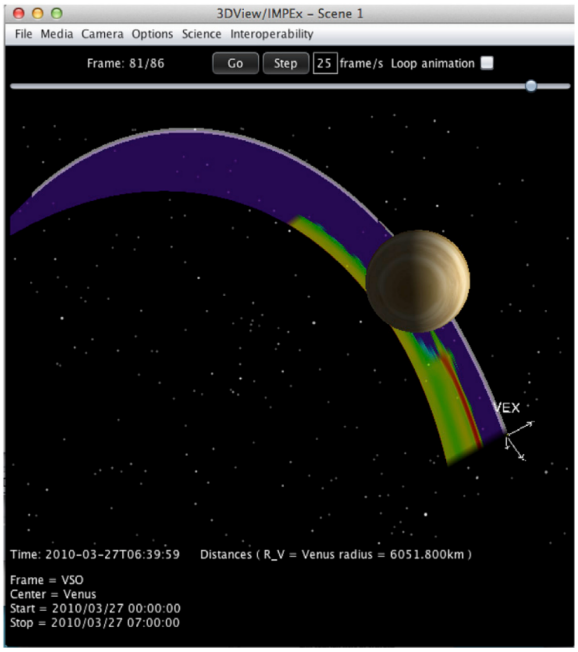


Figure 3.1: 3DView scene depicting spectrogram data by the ASPERA4 instrument aboard Venus Express. [24]

Gaia Sky is an interactive 3D visualization tool developed as part of ESA's Gaia mission Data Processing and Analysis Consortium. Gaia Sky's primary purpose is to provide an off-the-shelf visualization of the Gaia star catalogue, supporting both scientific analysis and outreach activities. It features a full simulation of the Solar System; including planets, moons, and asteroids. It enables exploration of Gaia star dataset releases. The tool does not only allow users to visualize Gaia and its orbit, but also other spacecraft. Additionally, it offers advanced features such as scripting interfaces (using a REST API access), stereoscopic and panorama modes, planetarium output, and VR support; offering interactive exploration and scientific visualization of complex astronomical data [25].

OpenSpace is an interactive data visualization framework designed to visualize the known universe and support dynamic, real-time rendering of scientific data. OpenSpace presents data from astronomical observations, simulations, and space mission planning and operations, spanning a vast range of spatial scales (spacecraft to universe scale). The software features an extensible architecture that supports high-resolution tiled displays, planetarium domes, and standard desktop computers, and allows for simultaneous global connections for shared audience experiences. The default dataset, automatically downloaded at startup, includes extensive Solar System data and space mission information (including spacecraft models and their trajectories), with support for high-resolution planetary surface views using virtual texturing and height maps. The tool offers functionality for both science communication and mission contextualization [25, 26].

3.1.2. Solar System Renderers

The previously mentioned scientific data visualization tools have as primary functionality the visualization of scientific data; the rendering of spacecraft and solar system bodies is a secondary feature, which supports the understanding of the data's context. The following tools were specifically designed to visualize spacecraft and the solar system. The selected tools provide much functionality when it comes to visualizing space missions, while being open-source (for potential addition of functionality).

Celestia is a highly flexible, open-source, real-time 3D astronomy and spaceflight simulation platform built to let users navigate through the Solar System, and Universe. Celestia uses the OpenGL process to render 3D graphics of celestial bodies, star catalogues and spacecraft models. Furthermore, Celestia is supported by a massive community-driven ecosystem of add-ons, offering high-resolution textures, virtual texturing, custom spacecraft models and SPICE kernels trajectory support. Celestia supports custom scripting by the user through LUA or its own CEL scripting language. Because of its large community and lightweight design,

Celestia is frequently used in educational settings, outreach activities, and by astronomy enthusiasts [25, 27].

The developer of Celestia, Chris Laurel, also developed another open-source solar system renderer called *Cosmographia*. Later, NASA's Navigation and Ancillary Information Facility (NAIF) group at JPL modified the software to include *SPICE* functionality, creating the SPICE-Enhanced Cosmographia software [28]. SPICE is an ancillary information system that assists scientists in planning and interpreting scientific observations from space-borne instruments, and to assist engineers involved in modelling, planning and executing activities needed to conduct planetary exploration missions [29]. The SPICE toolkit is now also widely adopted in ESA scientific missions, and it will be discussed in more detail in section 4.2.

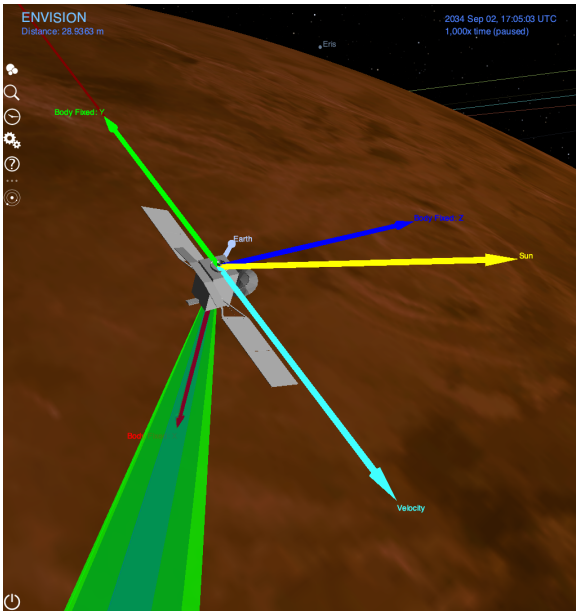


Figure 3.2: Cosmographia scene with loaded EnVision SPICE scenario

SPICE-enhanced Cosmographia offers interactive 3D representations of spacecraft trajectories, orientations, instrument fields of view, and celestial body positions using loaded SPICE kernel data. Users integrate mission-specific elements into Cosmographia through JSON catalogues, allowing visual navigation through detailed planning and mission activities of the spacecraft. Aside from education and outreach, SPICE-enhanced Cosmographia is also employed by planetary exploration teams and mission planners at space agencies. It supports tasks including visualization of mission sequences, spacecraft positioning analysis, and trajectory assessment [30]. Figure 3.2 shows a scene with an EnVision mission SPICE scenario loaded in Cosmographia, showing the spacecraft, instrument FOV, and vectors depicting reference frames and directions.

3.1.3. Science Operation & Planning Tools

During the assessment on usage of M&S simulators (with analysis functionality and a visual rendering interface) in ESA science missions, it became clear that the distribution of use is not even between centres managing the different mission segments. It became apparent that visual system simulators are more widely used within the SOC, compared to scientific project teams located at ESTEC. As was discussed in subsection 2.1.3, the SOC is responsible for the planning and execution of science operations. Two science operations and planning tools will be presented in this subsection. They showed high potential in meeting the needs identified in section 2.2.

In the past, the SOC used two interdependent tools in their science operations planning process: the Project Test Bed (PTB) and the Experiment Planning System (EPS). The PTB was developed for SOCs to simulate the spacecraft and its environment around the target body and mainly serves as a science opportunity identifier to help perform mission scenario studies. It can also provide both 2D and 3D graphical displays for visualizing mission geometry, based on the timeline created by the EPS. The EPS is a timeline management tool that schedules experiment operations from multiple instruments, ensuring there are no resource conflicts or operational constraints violations (power constraints, data volume limitations, instrument conflicts, etc.). Once it validates a conflict-free input timeline, it outputs a payload operations request for the MOC [31].

The **Mission Analysis and Payload Planning System (MAPPS)** is a software tool that was originally a 2D mapping tool derived from the PTB [31]. Presently, it is a stand-alone tool that has fully absorbed the PTB and EPS modules and is used by all solar system mission SOC teams at ESAC. MAPPS is a planning and validation tool that manages the science planning processes by verifying all input data (observation timelines, spacecraft configuration files, and operator inputs), simulating planning timelines to detect conflicts, checking for (spacecraft and experiment) constraint violations, and generating the necessary output files (spacecraft and experiment commanding files). It also supports auxiliary tasks such as producing simulation data for instrument teams and generating customizable data packs and event timelines [32]. In addition to MAPPS' simulation and planning capabilities, a 3D extension was integrated in 2014 for visualizing the complex operations of the Rosetta mission. The 3D interface is designed for high-speed rendering of mapping elements on the target body's surface, combining spatial and temporal domains to display geometric information such as spacecraft footprints, instrument swaths, 3D trajectories, instrument fields-of-view, star backgrounds, and solar system bodies [33]. Even though MAPPS is used as a planning tool in the SOC, its simulation and visualization

capabilities show major potential in addressing the predefined needs for project team systems engineers.

The **Solar System Science Laboratory (SOLab)** is a software tool developed at ESAC that is very similar to MAPPS, but with key differences. While MAPPS is more operational oriented, SOLab is development oriented. SOLab is a SPICE-based tool for optimization of the science goals and strategic assessment for long term and medium term planning. It was developed for the following functionalities [35]:

1. Identify observation opportunities for requested science quantities under specified geometrical constraints.
2. Quickly analyse and visualize the geometry of a given mission scenario.
3. Support science operations of planetary missions from study phase to day-to-day science operations.

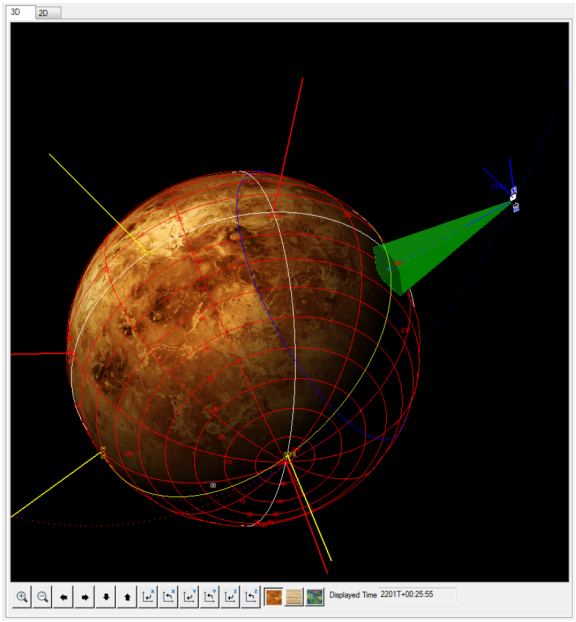


Figure 3.3: SOLab scene with Venus Express trajectory and instrument FOV visualization. [34]

The analogy between SOLab's functionalities and the identified needs in section 2.2 is quickly made. SOLab is designed for use throughout all mission phases, requiring a different user interaction. Simulations range from planning single observations (short time span with small time steps) to running full mission scenarios (long time span with large time steps). It allows users to define and analyse observation time windows, select spacecraft pointing and roll modes, and visualize instrument footprints, ground-tracks, and orbital elements in both 2D and 3D. Outputs can be stored, customized, and used for further analysis or integration into science operations [34]. A scene of SOLab, running an interactive SPICE scenario of the Venus Express mission, is shown in Figure 3.3.

3.1.4. Spacecraft System Design Tools

Another category of visual system simulators are those that are used for aiding the design of the spacecraft system itself. Many such tools exist; the subset presented in this subsection are selected based on compatibility with planetary missions, are system-level (not focused on a single subsystem), and offer a 3D visualization interface.

The **System Tool Kit (STK)** is a general-purpose modelling and simulation platform by Ansys used for Digital Mission Engineering and Systems Analysis. As a commercial software platform, it is used across the industry for mission design and system performance analysis. STK allows users to simulate satellite orbits, communication links, sensor coverage, and mission timelines by integrating physics-based models with real-world data. One of its major strengths is the high-fidelity 3D visualization it offers of the spacecraft system under simulation [36].

IDM–CIC (Integrated Design Model – Centre d'Ingénierie Concourante) is a Microsoft Excel-based macro built by the French national space agency (CNES) to support concurrent design practices in the early study phases of space projects. It enables collaborative work on a shared system model by storing and organizing data to generate essential budgets such as mass, power, and inertia at both subsystem and mission levels. The tool is applied in concurrent engineering activities in CNES, ESA and industry. IDM–CIC was designed to improve efficiency by reducing design iteration time and enhancing cross-disciplinary communication. Within CNES' IDM ecosystem, the application IDM-VIEW interfaces with the tool to provide a 3D visualization of the spacecraft system modelled in IDM–CIC [37].

3.1.5. Python-/Matlab-Based Tools

The final category of tools that were assessed, are not necessarily tools, but rather libraries or frameworks that are used to model spacecraft systems. Some are not particularly designed for 3D visualization, but offer elementary visualization, or could serve as a backend for a potential rendering framework. MATLAB and Python seem to be the most common programming languages used in script-based simulations for spacecraft systems. The previous tools are often distributed as executable applications, while the tools in this section are distributed as libraries or frameworks that can be used within the user's script.

Basilisk is an open-source, modular, and highly flexible spacecraft simulation framework designed to support mission design, algorithm development, validation, and post-launch analysis. Its architecture is built around modules, tasks, and task groups connected by a message-passing system that enables low coupling and high cohesion between the modules. Basilisk combines the speed of C++/C with the accessibility of a Python interfaces, allowing fast prototyping, Monte Carlo analysis, multiprocessing execution, and integration with external libraries. It supports fully coupled spacecraft dynamics, including structural flexing, fuel slosh, and environmental interactions. Importantly, Basilisk also provides visualization capabilities through Vizard, a 3D simulation and playback tool developed with Unity. It displays elements actively modelled in a simulation (planets, the Sun, or spacecraft devices) with corresponding lighting and interface features like HUDs or data panels. Users can interact with the simulation timeline through playback controls, including scrubbing, pause/resume, and speed adjustment, to explore the results dynamically [38].

PASEOS (PAseos Simulates the Environment for Operating multiple Spacecraft) is an open-source Python module containing built-in models for diverse physical and operational constraints encountered by spacecraft in various mission scenarios, including thermal balance, power, bandwidth, communications, and radiation effects. The package was primarily developed to study constellations in space for emerging operational scenarios, such as edge computing, edge and decentralized learning, and artificial intelligence in space. However, it presents itself as an application-agnostic framework that can be used for a wide range of space mission simulations and applications. It uses the Python package `pykep` to model Keplerian orbits or `orekit` for propagated orbits, with `skyfield` for ground station visibility geometries. PASEOS prioritizes rapid background execution over fidelity, making it useful for early technology demonstrations, hardware-in-the-loop (HIL) and software-in-the-loop (SIL) simulations, and enabling easy initial consideration of various operational aspects for spacecraft with low computational cost. The package ensures modularity with its activity paradigm and custom properties features. In its current state however, it models (and visualizes) spacecraft as point masses [39].

MATLAB/Simulink-based tools play a significant role in the M&S landscape for spacecraft projects, particularly in early design phases and modelling of dynamical systems. They are widely regarded as industry standards for developing custom simulation products, especially within specific disciplines such as Attitude and Orbit Control Systems (AOCS). The Aerospace Toolbox and other commercial toolboxes such as ASTOS and the Spacecraft Control Toolbox (SCT) exist, extending MATLAB/Simulink with predefined functionalities for spacecraft systems. However, MATLAB-based simulators are more commonly developed in-house, either from scratch or adapted from previous projects, shared among colleagues. MATLAB-based tools are widely used in the industry, offering high flexibility and performance in computing complex dynamical simulations [21].

3.2. Implementation of System Modelling and Simulation

This chapter started with the introduction of system M&S for space systems engineering. Particular needs within the implementation of planetary missions were identified in section 2.2, and a selection of promising tools, addressing these needs was presented in section 3.1. This section aims to study the implementation of system M&S by first assessing the current trends in system-level M&S in space systems engineering in subsection 3.2.1; How does M&S look in the current landscape of space systems engineering and is the state of the art from section 3.1 actually implemented to address the common needs? What are the M&S shortcomings based on the observed (lack of) implementation? subsection 3.2.2 will place the observed trends in an academic field of study and introduce the System Simulator Concept (SSC), an approach to M&S that aims to overcome the existing shortcomings in M&S for systems engineering in space projects.

3.2.1. Trends in System-Level M&S

The landscape of system-level M&S tools is very extensive. The selection of tools included in state-of-the-art reviews typically represents only a small fraction of what is available. Although numerous tools exist, the majority of systems engineers either rely on highly specialized, discipline-specific software or develop their own in-house scripts tailored to mission-specific analysis needs. These scripts are developed and maintained within fragmented repositories, specifically designed to meet the requirements of the mission they support.

There are several reasons why in-house scripts are often used over existing tools. First, some tools lack essential analytical capabilities. This is especially apparent for the solar system renderers category. Although they are often open-source, modifying them to add missing features requires significant development effort. Contrarily, some tools offer too much functionality, resulting in cluttered interfaces and a steep learning curve for the new user. These tools are often designed with specific applications in mind, which may be irrelevant for users with different missions, disciplines, or mission phases. This can hinder usability and discourage its adoption. Examples of this are STK, IDM-CIC, and 3DView. Their extensive functionality (which makes them attractive in their own domain), limits their usability for systems engineers in project teams who may not require all

features they offer [21].

There are cases where tools intentionally limit their scope to remain usable and effective in their specific use case. For instance, in the early development of MAPPS, the tool showed great promise of supporting design practices, but its scope was deliberately narrowed to focus on planning and operations. According to one of its developers, this decision helped preserve its usability and relevance, avoiding the dilution of its core functionality. Tools that attempt to "do it all" often lose coherence and ultimately fall out of use.

A large obstacle when it comes to tool adoption is the time investment required to familiarize with a new tool. Familiarization is not only needed in order to use or customize the tool, but also to assess its potential use for the project. This is one of the main reasons for simulation scripts fragmentation between missions. A system simulator developed in a legacy mission, evolved throughout this mission's development. This means that at the time a new project team want to reuse it, a more specialized product that ended in a more mature project phase is received [20]. Usually, instead of committing valuable time to familiarize and customize the legacy tool, the systems engineer decides to start from scratch and start the development of new mission-specific scripts.

The profound shift towards the use of in-house scripts is very noticeable in the case of SOLab. In subsection 3.1.3, its functionalities seemed to align greatly with the predefined needs in section 2.2. However, in a video call, SOLab's developer M. Costa clarified that the software had been fully deprecated in favour of analysis with Python Notebooks with a heavy SPICE usage and visualization with Cosmographia. Over the years, the accessibility to script-based M&S for non-developers has increased tremendously (especially with the large adoption of Python and SpiceyPy, more in section 4.2). The paradigm has changed completely; developing a mission-general tool or application is out of date, such tools inherently cannot cover all the specific (and growing) needs of a mission.

The adoption of existing frameworks such as those described in subsection 3.1.5 appears to remain limited in the context of system-level modelling and simulation (M&S) for space missions. Several possible reasons may contribute to this, some of which are similar to those discussed above. Take for example Basilisk – being a simulation framework instead of a library – might let it be perceived as complex or unintuitive, particularly by users more familiar with the straightforward interfaces of standard Python libraries. Even basic tasks, such as plotting or setting up a minimal simulation, may require considerable configuration effort. MATLAB-based tools, tend to be more discipline-specific, especially used in the domains of orbit and attitude dynamics and control. In the context of the mission's systems engineer, the dynamics and control aspects are often covered by specialized teams and used as inputs for requirement verification on a higher system level. Moreover, the built-in models provided by packages such as PASEOS and Basilisk may not always align with the fidelity requirements of a given mission. PASEOS prioritizes computational efficiency at the expense of detail (e.g., omitting attitude modelling), whereas Basilisk implements high-fidelity models in C++, which might reduce transparency and ease of modification from the systems engineer's perspective. Taken together, these factors may help explain why system engineers frequently choose to build mission-specific simulations from scratch, allowing for greater control, flexibility, and alignment with the specific objectives of their analysis. These observations are supported by findings covered in the next subsection.

3.2.2. The System Simulator Concept

T. Nemetzade in her PhD dissertation, *Characterization and Application of User-Centred System Tools as Systems Engineering Support for Satellite Projects* [21], identifies the same shortcomings of M&S tools in the space systems engineering discipline as in subsection 3.2.1. She states that modelling and simulation tools are developed using an ad-hoc approach. This means: modelling and simulation tools are mostly discipline-specific, created from scratch, self-made, and employed in a short time period, rather than being system-oriented, reused, or used over a longer period of time. The trend is usually due to factors like a tight time schedule (not allowing for familiarization time) and limited usefulness of existing tools for their specific desired task. This leads to the development of tools that have limited usefulness to other disciplines, do not allow for knowledge transfer between phases and missions, and have a limited system-level application.

The dissertation introduces the *System Simulator Concept (SSC)*, a novel approach to M&S aimed at overcoming these existing weaknesses in projects. The core idea of the SSC is to integrate the user into the design and development process of software solutions, following a user-centred design approach. The SSC serves as a guideline for projects that aim to create and apply a system simulator to support Systems Engineering activities. It systematizes the successful development approach and necessary characteristics of the simulator end product.

Nemetzade's study will act as a guideline in assuring the results of this thesis will be an effective tool in

supporting SE activities across the mission development phases and disciplines (as specified in the project's needs in section 2.2). The SSC is structured around three main principles which the simulator must follow to be considered an effective SSC-based simulator:

Principle I: Characteristics: the simulators built according to the SSC are designed to be: lean, adaptive, transparent, and easily manageable; they focus on simulation of the system, rather than being limited to single disciplines. Only essential models and functionalities are implemented; functionality is added based on growing user needs throughout the mission lifecycle.

Principle II: Context of Use: the primary users of SSC-based simulators are systems engineers, with secondary users being discipline experts. It is expected that users have a fundamental astronautical background, but limited experience with M&S tools. The tool must be suited for a highly dynamic project environment, where simulator development and familiarization time is limited.

Principle III: Implementation Principles:

- **Principle III – a: Simulator Development Approach:** the SSC advocates an iterative and user-centric development approach. In short, this means assuring user satisfaction, while keeping employment barriers as low as possible. This implies a need for continuous implementation of user needs, gradual addition of user-specific, user-defined content, which does not overburden the users. Clear and easy interfaces are key for manageable familiarization. Change requests and development activities (specification, implementation and validation) must be streamlined to be handled in a timely manner.
- **Principle III – b: Simulator Scope:** The simulator must act as a complement to advanced discipline-specific simulators. It must provide a system-level perspective by integrating essential elements from multiple disciplines, modelling their cross-connections. Focusses on fundamental but comprehensive system-level models instead of detailed but single discipline modelling.
- **Principle III – c: Scheduling of SSC Implementation:** It is recommended to start SSC implementation in the early life cycle stages, to allow for evolution together with the project.

During the development of the proposed work in this thesis, the SSC principles will be used as guidelines. They will be covered in the evaluation of the end product in the concluding chapter 7.

3.3. Objective & Research Questions

In this section, the previously stated needs of the systems engineer from chapter 2 are translated into a research goal, informed by the observed shortcomings and established guidelines in the field of space system M&S in chapter 3. Based on the trends observed in subsection 3.2.1 and supported by subsection 3.2.2, it is clear that established tools with System Simulator (SS) capabilities, are discarded in favour of ad-hoc mission-specific scripts. While the ambition to attempt the development of a single mission-agnostic SS can be compelling, a different approach is pursued in this thesis. The chosen approach is to develop a generic toolkit that supports the development of mission-specific SS's, taking advantage of the observed preference in M&S instead. It shifts the focus from developing a SS *tool* that is useful for all missions, to developing a generic *toolkit* that is useful for developing mission-specific SS's, facilitating the current status quo of in-house ad-hoc developments.

This approach is not novel; Von Hippel and Katz [23] formulated this approach as a way to transfer design capabilities to users, by offering "toolkits for user innovation". It shifts away from attempting to satisfy a market's ever-changing need, to outsourcing need-related innovation tasks to the users themselves, after equipping them with the appropriate toolkit. This has shown to speed up the development of custom products at a lower cost. The process was studied within the general field of product design, but in the field of space systems engineering, which is saturated with effective application-specific toolkits (mostly employed in the SOC and MOC), it seems to be missing in high level systems engineering in the scope of a project team.

In this work, it is theorized that a toolkit supporting the – presupposed – inevitable ad-hoc approach of M&S activities of the systems engineer, could offer a practical and contemporary approach to the development of mission-specific system simulators (SS's). *Instead* of aiming to develop a SS that is useful for all missions. The toolkit will be developed and used in a case study of a mission-specific SS development. The development of the toolkit, its use in creating a SS, and its evaluation will be performed within the scope of the EnVision project, starting from B2 implementation phase.

While the new approach may assure the method is applicable to multiple missions, a question remains as to how the toolkit can offer simulators that remain relevant throughout the mission lifecycle, and effectively support the end user (systems engineers). This is where the guidelines established by the SSC in subsection 3.2.2 come

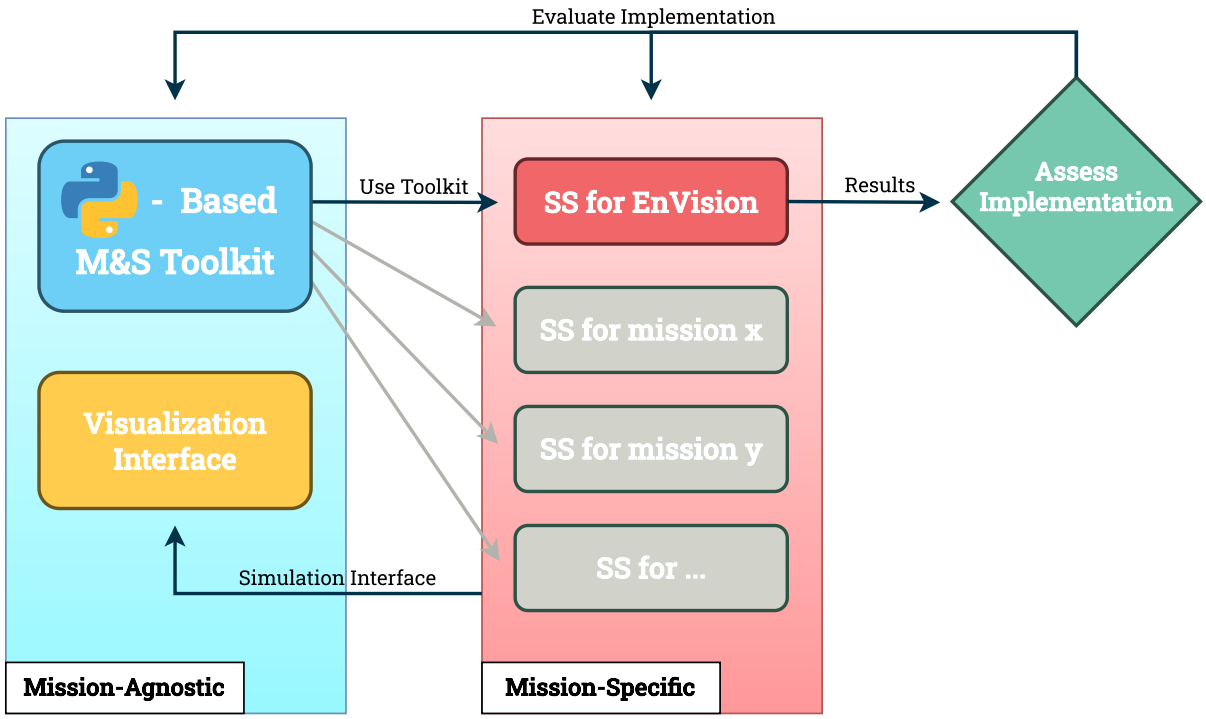


Figure 3.4: Graphical representation of the toolkit usage, implementation and evaluation.

into play. By making informed design decisions that follow the SSC principles, these properties can be more confidently ensured and evaluated.

The chosen approach is illustrated graphically in Figure 3.4. To rephrase the approach in a more structured way, the following three main components are defined:

1. The **Toolkit** will aim to be *mission-agnostic* by nature, providing only the most essential functionalities in aiding the design and verification of a custom, mission-specific system simulator.
2. The **System Simulator (SS)** will be custom-built using the toolkit, and is *mission-specific* (in this case, EnVision). The system simulator will house a configuration of the spacecraft system and allow simulating models of its components and their interactions.
3. To **Evaluate** the approach in its ability to stay relevant and support the end user, the SS and the underlying toolkit will be assessed on their adherence to the principles of the SSC.

The decision was made to constrain the development to a Python-based toolkit and SS's. The majority of existing ad-hoc system-level tools in ESA interplanetary projects are already developed in Python. This shows the growing preference in using the language as a growing familiarity with Python is already common among systems engineers. The descriptive nature of the language offers increased transparency on how models are built. Also, the fact that Python is a programming language rather than a pre-packaged application, allows for the maximum amount of flexibility and control. Note that transparency, flexibility and control, are key principles of the SSC guidelines from subsection 3.2.2. The latter does also have a downside; flexibility is often in a direct trade-off with accessibility and ease-of-use [21].

However, Python is not known for its high performance. A SS, by nature, must include multiple models into a representation of the system as a whole. This often means performing many calculations in a short time span, which can be computationally expensive. Careful design choices must be made to overcome this, limiting flexibility and functionality, to benefit usability and performance. Besides, a necessary consideration is to assess whether or not Python *can* offer the necessary functionality to support a SS development.

The Research Objective is as follows:

"Supporting spacecraft system-level analysis and verification, by developing a mission-agnostic Python-based toolkit, designed to support the development of custom System Simulators; using EnVision as a hands-on case study of its implementation in producing a viable System Simulator that adheres to the SSC."

To tackle the research objective, the following research questions and sub-questions have been formulated. Answering these questions will provide a structured approach to the research, and will ensure the research objective is met. The main research question is:

"How can a Python-based system simulator toolkit, support systems engineering activities of interplanetary scientific orbiter missions from the preliminary definition phase and onwards?"

This research question is encompasses a broad development scope. To ensure the research remains focused, the following sub-questions are defined. They will be answered in the course of the research, and will help to structure the development of the toolkit and its evaluation:

SubQ – 1: *What general functionalities must the toolkit offer, in order for systems engineers to develop effective system simulators, and which ones should it aim to exclude?*

SubQ – 2: *What are the criteria for assessing the effectiveness of a system simulator developed with the toolkit?*

SubQ – 3: *What design choices are key in allowing the toolkit to be implemented in multiple interplanetary mission projects?*

SubQ – 4: *What design choices are necessary to ensure support across multiple mission phases and disciplines?*

This concludes the research orientation phase. A solid baseline on ESA systems engineering process and the current state of the art is set up. The objective of the research is defined, with the supporting research questions in place. The approach taken in reaching the objective is scoped, based on a study on current trends observed in the field of planetary space systems engineering, and backed by guidelines and principles established in literature. In the next chapters, the methodology of the toolkit development, and its implementation in the EnVision case study will be discussed.

4

PAS3 Development

This thesis proposes a newly developed toolkit called PAS3 (a **P**ython toolkit for **A**ccessible **S**PICE-based **S**pacecraft **S**imulations). The toolkit is designed to support the development of space mission simulators, with a high focus on planetary missions. The toolkit is built in Python, and can be used as a package import to be used in the user's own script when they require a simulation backend of the system.

PAS3 wraps the SpiceyPy library, which provides a Python interface to the SPICE Toolkit, into a more accessible object-oriented, declarative interface. This provides the user with an intuitive and Pythonic way to access SPICE data and other system/environmental attributes in their own system models. Moreover, PAS3 extends the SPICE functionalities with built-in custom classes, omitting the need for the user to manually set up SPICE-equivalent objects.

Figure 4.1: PAS3 repository logo.[1]

Section 4.1 discusses the design philosophy of the toolkit, its intended use, its scope and the solution space it aims to cover. Section 4.2 introduces the SPICE Toolkit, its use and role within the toolkit. Within this work, many SPICE jargon will be used, this section serves as an introduction to the SPICE Toolkit and its terminology. In section 4.3, the functional and non-functional requirements of the toolkit are established. The resulting toolkit's architecture is presented in section 4.4, listing every module, their main functionalities and mutual interactions. The chapter concludes with descriptions of verification activities of the toolkit in section 4.5.

4.1. Design Philosophy

In the previous chapter (more specifically section 3.3), the reasoning behind a toolkit approach, rather than a stand-alone tool, was discussed. The supporting work on this approach also highlights the importance of the so-called *solution space* in the development of a toolkit [23]. The solution space refers to the range of possible product or service designs that users can create using a given toolkit. It represents the boundaries of what is possible given the capabilities and constraints. The toolkit must be intentionally constrained to balance user freedom and the complexity and performance of the toolkit itself. This subsection will establish these constraints in functionality of the toolkit.

In the case of a Python-based toolkit for space mission simulations, constraining the solution space is especially important. Python is a high-level scripting language, interpreted at runtime and not natively allowing for multithreading. This makes Python a very accessible language, but also one that is not known for its performance at runtime. Most simulator tools (section 3.1) are written in lower-level languages, such as C, C++ or JAVA, which allow for quicker execution of mission analysis. Selectively decreasing functionality of the toolkit, in favour of optimizing the performance of the functionalities in the predefined solution space, is a key design consideration to overcome performance concerns. The decision must be made based on the context in which the toolkit aims to be used.

Figure 4.2 divides two distinct features of the toolkit: the left side shows the models that are integrated within the toolkit, while the right side represents the models that are (deliberately) left out of the toolkit. The main difference between these two is their *mission-agnosticism*.

[1] AI-generated using OpenAI ChatGPT

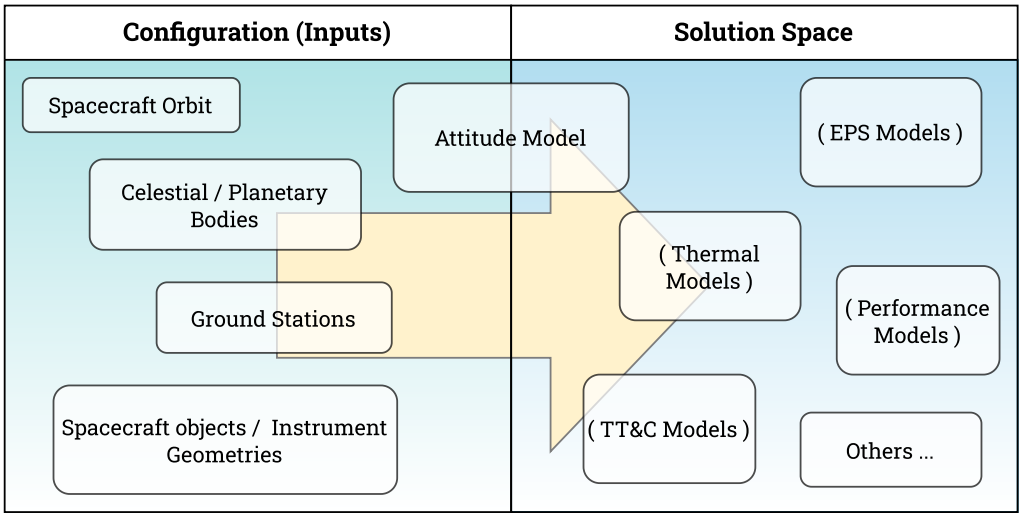


Figure 4.2: Toolkit configuration (inputs) and solution space (desired "outputs"). The left side shows the models integrated within the toolkit, configured by the user. The right side shows the models left out of the toolkit, which are built by the end user.

Integrated models are deemed generic; their inherent model architecture is common across space systems, and to provide the correct behaviour for a space system, the end user must only *configure* them correctly. General models include:

- **Spacecraft orbit**.
- **Celestial bodies**: the spacecraft's central body, the Sun, Earth, . . .
- **Ground stations** on Earth.
- **Geometries**: spacecraft geometrical regions (side panels, solar arrays, antennae) or instrument fields of view (more in subsection 4.4.5).

These models are generic in the sense that they have different configurations (e.g. orbits, central bodies and instruments change between missions), but inherently are modelled the same way across different space systems. They are integrated natively within the toolkit.

The models on the right are left out on purpose. Including them would make the toolkit not mission-agnostic by design. These models are characterized by having an application-specific architecture; be it due to different system/mission specifications (every spacecraft is different) or the required fidelity of the model (low fidelity in early design, higher fidelity in more mature phases). These models must not only be configured, but are expected to be *built* by the end user.

For example: a spacecraft's thermal model can start off as a simple point-mass system, taking solar, albedo and IR fluxes depending on the spacecraft's position and computing the elementary thermal balance. This model can be extended to include spacecraft attitude, individual faces, their view factors and thermo-optical properties. Even more in-depth models could include event-based activity of other subsystems and their heat dissipation, active/passive thermal control systems, and so on. It is clear that one single thermal model cannot truly be mission-agnostic, as it must be tailored to the specific spacecraft design and development phase requirements.

Requiring the custom development of such models, can be quite demanding from the end user. For this exact reason, the toolkit was developed. The toolkit provides an intuitive and Pythonic interface to access space system attributes and methods to ease the design of these custom models. Standardizing a select set of system models within a simulation, requiring only their configuration, provides a balance between flexibility and familiarization effort for the end user (the systems engineer). This approach is a direct interpretation of the principles established by the SSC. It must be emphasized that the toolkit solution space is intended to include the mission-specific system models it helps create, not their numeric outputs[1].

To illustrate the model interaction between native toolkit models and custom models the toolkit aims to achieve, a hypothetical Electric Power Subsystem (EPS) model is illustrated in Figure 4.3. The toolkit provides a simulation environment with spacecraft and environmental attributes (positions, attitudes, etc.). The general

[1] Refer to Figure 3.4: the toolkit helps to make system simulators (SS) and their mission-specific models. The SS provides the deeper system analysis functionality.

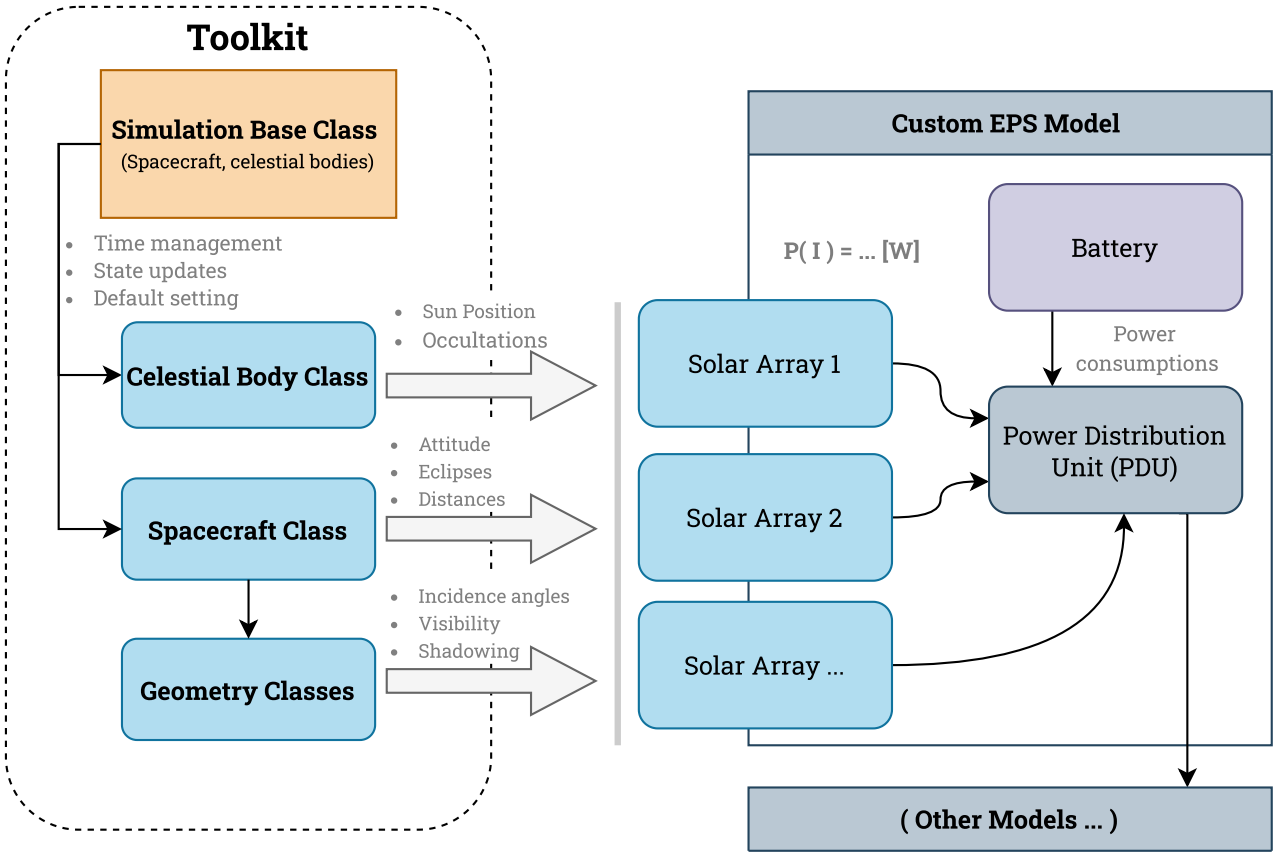


Figure 4.3: Example of the Toolkit's built-in Object classes and their interaction with a hypothetical custom Electrical Power Subsystem (EPS) model of the spacecraft.

Geometry abstractions allow the user to define the spacecraft's solar panels, with built-in methods to compute the solar flux I received by the geometry. The number of solar arrays, their behaviour (how the solar flux is transformed to power), and how the power distribution and storage is handled, is left to the end user to implement in their mission-specific SS. The nature, architecture and required complexity is inherently mission non-agnostic.

As mentioned before, the solution space of the toolkit must be constrained in advance for performance reasons. The toolkit has been deliberately designed to exclude orbit propagation from its internal models. This can be seen in Figure 4.2, where spacecraft orbit is treated as a configuration input, not a model. This decision stems from both performance considerations and the specific context in which the toolkit is intended to be used. In the context of the implementation phase of an ESA scientific mission, detailed orbit trajectories are established by the Mission Operations Centre (MOC) and Flight Dynamics (FD) teams at ESOC. These are distributed via standardized data formats such as: OEM (Orbit Ephemeris Message) files, and SPICE kernels. Recomputing or propagating these orbits within the toolkit would not only be redundant for its intended application space but also an inefficient use of Python's computational resources.

Similarly, the toolkit shall not be intended for tasks like parametric optimization of orbits trajectory or science return. These tasks are better handled by specialized tools/teams like the FD team, the mission performance team and SOC respectively. While in the future, an orbit propagation module interface could be added to the toolkit (e.g., GMAT, TUDat, Orekit), and the toolkit itself could be used to run observation windows and science return optimizations, the current focus is to take these timelines as inputs, providing only the robust framework for system-level simulations. The toolkit's primary purpose is to support synchronous simulations of the space system, extract design sizing parameters, verify system-level requirements, and interface with other analysis and visualization tools.

In line with this design philosophy, state variables of surrounding celestial bodies are not computed dynamically during simulation. Instead, they are extracted from SPICE kernels, which provide high-fidelity data on planetary positions, spacecraft states, reference frames, and time systems. These kernels serve as the primary input to the toolkit's configuration (see the left side in Figure 4.2).

A special case within this configuration approach is the spacecraft attitude (divided between being an input and

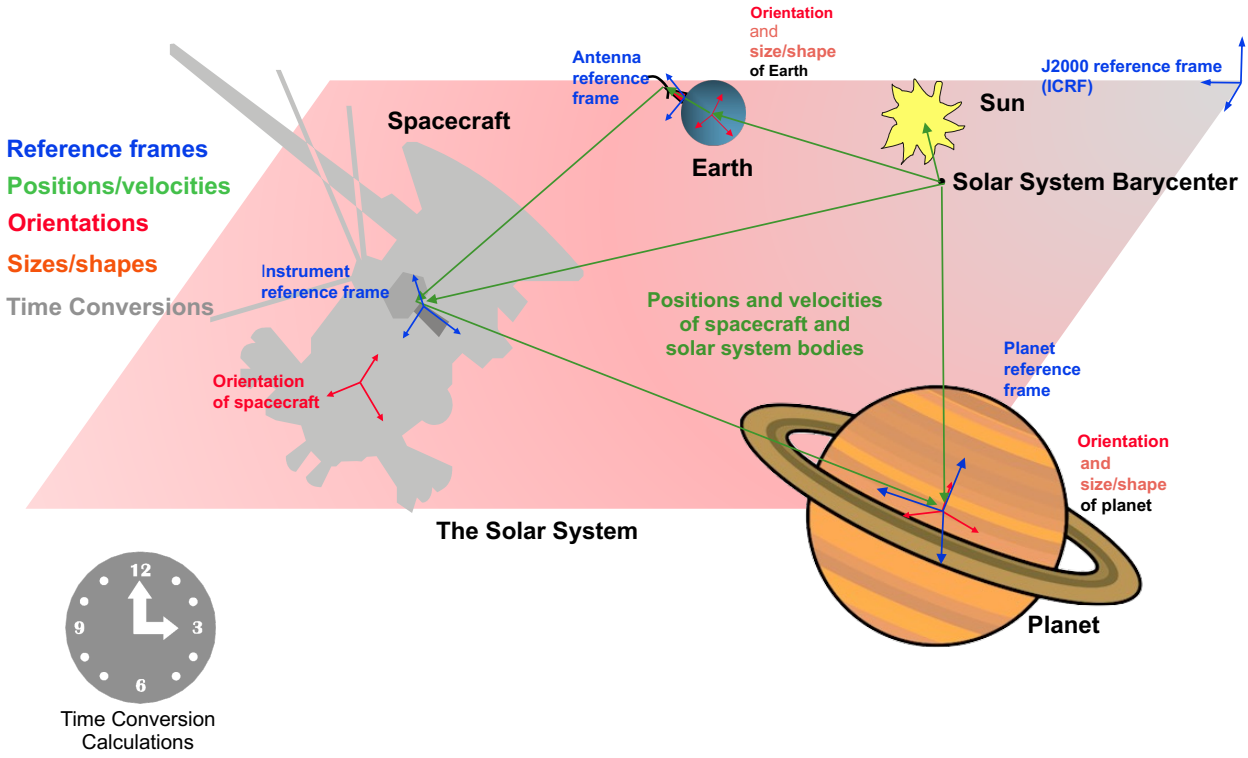


Figure 4.4: Graphical overview by NAIF of "ancillary data" stored in SPICE kernels [41]

solution model in Figure 4.2). While SPICE kernels can define attitude profiles, their availability and fidelity depend heavily on the mission phase, and the mission itself. In early development stages, attitude information may not yet be defined due to ongoing planning of science observations and communication windows. In later stages, a reference attitude profile may be delivered via SPICE kernels. To accommodate this variability, the toolkit supports both configuration from existing SPICE kernels and user-defined attitude modelling using its internal tools. This dual approach ensures agile flexibility in early phases, and the possibility to have a more optimized runtime with SPICE kernels in the more mature phases of the mission. A more detailed discussion on SPICE kernels and their role in the toolkit is provided in the following section.

4.2. SPICE Toolkit

As briefly introduced in previous sections, the SPICE Toolkit is a software system developed by NASA's Navigation and Ancillary Information Facility (NAIF) to assist in the planning and analysis of space science missions. It is a software system that provides a framework for computing observation geometry data, such as positions, orientations, lighting angles, and field-of-view projections over time. This type of data is referred to as "ancillary data", which essentially represents supplementary information required to characterize the geometry of an observation. While the main purpose of the SPICE Toolkit is to support planning and analysis of space science observations, it is also very powerful serving as a back-end configuration for spacecraft system simulators [40].

The SPICE Toolkit provides a framework to compute geometrical quantities (like occultation events, or line-of-sight conditions) based on time-tagged ancillary data stored in kernels (standardized data files). Ancillary data are typically sourced from spacecraft telemetry, mission control, hardware specifications, and scientific teams as well as international organizations like the International Astronomical Union (IAU), which standardizes certain reference frames and planetary constants. Different SPICE kernel types are used to store different kinds of ancillary data. The file types are summarized below, and are also illustrated in Figure 4.4.

- **SPK** (Spacecraft Planet Kernel) files contain ephemerides (position and velocity) for everything that moves. This includes: spacecraft, planets, satellites, comets, asteroids, and also moving or fixed spacecraft and instrument structures.
- **PCK** (Planetary Constants Kernel) files contain physical, dynamical and cartographic constants for target bodies like: size, shape, orientation of the spin axis and prime meridian. Orientation models for natural bodies can also be stored in binary PCK files.

- **IK** (Instrument Kernel) files contain instrument parameters like its shape and FOV, and internal timing parameters.
- **CK** (C-matrix Kernel) files contain time-varying orientations for spacecraft, spacecraft structures, and articulating science instruments.
- **DSK** (Digital Shape Kernel) files contain detailed shape models for extended objects, such as planets, moons, asteroids, or spacecraft components. DSKs are used for higher-resolution surface and topography data.
- **LSK** (Leap Seconds Kernel) files contain the leap seconds and the values of other constants required to perform a transformation between Universal Time Coordinated (UTC) and Ephemeris time (ET), also known as Barycentric Dynamical Time (TDB).
- **SCLK** (Spacecraft Clock Kernel) files contain on-board clock calibration data required to perform a transformation between Ephemeris time (ET) and spacecraft on-board time.
- **FK** (Frame Kernel) files define reference frames and their relationships within the SPICE system. They provide the necessary information to establish how different frames are oriented and connected, including mounting alignments for spacecraft and instrument frames.
- **MK** (Meta-Kernel) files list sets of related SPICE kernels that should be used together. Instead of having to load individual kernels in the right order, a meta-kernel allows users to load a single file that specifies all the necessary kernels for a particular analysis or mission phase.

The SPICE Toolkit is available in Fortran 77 and C, with NAIF official wrappers for MATLAB, IDL, and Java Native Interface. Community-contributed interfaces extend SPICE functionalities to other environments like Python, Ruby, Swift, Julia and even the Unreal Engine 5 game engine. The emergence of the open-source Python SPICE wrapper *SpiceyPy* [42] a few years ago, has significantly increased the accessibility of SPICE for Python developers, likely explaining the shift towards SPICE-based Python notebooks observed in subsection 3.2.1. This wrapper will be used heavily in the toolkit environment.

Although SPICE is mature and well-documented, it has a steep learning curve. Users must be familiar with spacecraft coordinate systems, time systems, SPICE's numerous file formats and terminologies. The Toolkit's semantics are not inherently descriptive, and certain methods and outputs rely on the user's background in the conventions established in SPICE's required reading material. Multiple SPICE training courses are provided by agencies like ESA and NASA for this exact reason.

Additionally, SPICE follows a procedural approach, requiring users to manually integrate routines into their programs. This provides ample functionalities for the Toolkit's intended use, but can be cumbersome for users who are more accustomed to object-oriented approaches of integrating libraries. While SPICE excels as a robust backend for established missions with fixed parameters, the kernels are not inherently agile in use. They are not intended to be modified easily (serving as a standard to be used as input, not necessarily as variable) and changing parameters like attitude or instrument configurations dynamically during runtime is not supported.

SPICE offers a robust foundation for the toolkit under development, enabling fast and high-fidelity geometric computations. However, for increased flexibility in analysis (particularly in early-implementation design activities) an object-oriented architecture in Python is desirable to support more agile, system-level simulations. This architectural consideration should be incorporated into the development process of the resulting toolkit.

4.3. Toolkit Requirements

To ensure the toolkit effectively supports the systems engineering process for planetary space missions, a clear set of requirements has been defined. They flow down from the objective established in section 3.3 and the design philosophy outlined in section 4.1. Many of these requirements also stem from guidelines established by the SSC in subsection 3.2.2, which should ensure the end product can be adopted as an effective system simulator toolkit. In subsection 4.3.1, the functional requirements of the toolkit are summarized, while subsection 4.3.2 describes the non-functional requirements.

4.3.1. Functional Requirements

Functional requirements specify the essential capabilities and behaviours that the toolkit must provide to fulfil the needs and expectations of its users. They define what the system should do, describing its functions, features, and interactions with users or external systems. Clearly defined functional requirements are crucial for guiding the development process, ensuring that the software delivers the intended value and meets its operational objectives, following predefined guidelines. The Functional requirements and their reasoning of the toolkit are summarized in Table 4.1.

Table 4.1: Functional toolkit requirements

ID	Description	Reasoning
	SPICE Kernel Management:	Choice of using SPICE kernels in section 4.2 as an ephemerides-based method state determination. Need for making data more accessible and user-friendly for the end-user (SSC principle II).
TK-F-1.1	The toolkit shall provide a way to load and use SPICE kernels' data.	
TK-F-1.2	The toolkit shall provide methods to list kernels and their stored ephemerides bodies or frames.	
TK-F-1.3	The toolkit must assign time coverage of loaded kernels to a simulation instance.	
	Simulation environment:	Ensures the possibility to create a mission-specific system simulator (section 3.3)
TK-F-2.0	The toolkit shall provide instances of configurable simulation environments to run synchronous simulations of assigned actor's states.	
	Simulation Modularity:	In adherence to SSC principle I and the toolkit solution space in section 4.1.
TK-F-3.0	The toolkit shall allow the user to add custom callable models to the simulation environment, providing an interface to pass arguments to integrated custom models at time changes.	
	Actor Configuration:	Use of SPICE in configuring standard space system simulator objects (section 4.1). A system simulator must aim to model the system as a whole, calling for interdependency between simulated instances.
TK-F-4.0	The toolkit shall provide **SPICE**-configurable object instances of (multiple): • Spacecraft • Celestial bodies • Ground stations • Instruments Inheriting state-dependent attributes and methods.[1]	
	Configuration Agility:	Need for agility and flexibility in early design-centric phases (adherence to SSC principles I and III-c). SPICE toolkit may offer ground station, attitude and instrument geometrical descriptions, but kernels might not be present (yet). Also, more general spacecraft properties like geometries and subsystems are not supported and need to be configured separately (section 4.1)
TK-F-5.0	The toolkit shall provide the user with the necessary tools to configure **user**-defined objects compatible with the SPICE-configured simulation instances for: • Ground stations • Spacecraft Attitude • General spacecraft geometries like panels (radiators, spacecraft faces, solar panels, antennae, etc.) and (unconventional) instrument fields of view (SAR, coronagraphs, etc.).	

[1] Example: an instrument instance is associated to a parent spacecraft instance; its FOV visibility methods' outputs are heavily dependent on the state of the parent spacecraft's and loaded celestial bodies' states in the simulation.

ID	Description	Reasoning
	Actor State Management:	The toolkit must provide an accessible way of extracting an actor's state throughout time (section 4.1). SPICE's state extractions require input parameters like epoch, observers, aberration correction, frames of reference, which can be standardized in the simulation environment.
TK-F-6.1	The toolkit shall provide methods to update the state (i.e. relative positions, velocities, orientation, visibilities, distances, angular separations, orbital parameters) of the actors in the simulation environment.	
TK-F-6.2	The toolkit shall provide a method to configure global settings of each simulation environment in the calculation of these actor's states.	

4.3.2. Non-Functional Requirements

The non-functional requirements of the system are defined as the quality constraints that the system must satisfy to enhance the overall user experience. These requirements focus on how the system performs rather than what it does.

The non-functional requirements of the toolkit are summarized in Table 4.2. These requirements are derived from the SSC principles, the current trends in system-level modelling and simulation, and the needs of the end users as discussed in section 2.2.

Table 4.2: Non-functional toolkit requirements

ID	Description	Reasoning
	Usability:	Requirements to decrease familiarization time in utilizing the SPICE toolkit in Python scripts and notebooks (high importance for SSC Principle II and III-a).
TK-NF-1.1	The toolkit shall provide comprehensive documentation.	
TK-NF-1.2	The toolkit shall provide descriptive attributes, methods, properties and error messages.	
TK-NF-1.3	The toolkit shall limit required input parameters for integrated SPICE functions by providing configurable defaults.	
TK-NF-1.4	The toolkit shall provide examples and tutorials.	
	User-Agency:	Integrated loops decrease transparency and control. Aim for user-driven architecture to be used in the current scripts and notebooks (subsection 3.2.1) and giving the user control in addressing their needs (SSC principle I).
TK-NF-2.0	Toolkit simulations must be integrable in external event loops, using stepwise execution by the user, rather than self-contained, internal simulation loops.	
	Object-Oriented:	subsection 3.2.1 and SSC Principle I.
TK-NF-3.0	The toolkit shall be developed in Python, following object-oriented programming (OOP) principles, and provide a Pythonic interface to the user.	
	Mission Agnostic:	Ensures adaptation and reusability to other planetary missions as covered in section 2.2 and SSC Principle III-a.
TK-NF-4.0	The toolkit shall not be tailored to a specific mission or spacecraft design and does not include spacecraft-specific models.	

4.4. Toolkit Architecture

The toolkit will be developed as a Python package with a structured architecture, distributing its functionalities across multiple modules. Within these modules, the toolkit stores class blueprints to be used as object-oriented building blocks for the user's mission system simulator. Six main types of class blueprints are provided by the toolkit; they are listed below, as well as their use case.

- The **Simulation** instance – provides functionality in loading and interpreting SPICE kernels, managing simulation time in the user's execution loops, updating the state of the actors in the simulation, and configuring default simulation parameters.
- The **Spacecraft** actor – main focus of the toolkit, offering extensive functionalities in extracting the spacecraft's time-dependent state within its environment. Does not inherently offer models of the spacecraft subsystems, only geometrically-linked parameters.
- The **Celestial Body** actor – offers state variables of celestial objects like planets, asteroids, comets, moons or the Sun. One must be configured as the spacecraft's central body.
- The **Ground Station** actor – provide methods for determining and configuring visibility with one or more spacecraft. Use for power and communication models.
- The **Geometry** objects – An abstract class representing "regions" of space moving with the spacecraft actor including different shapes of instrument fields of view (FOV's), rays and hemispherical regions (like a single face of a plane). Used for extracting orientation-coupled visibility conditions.

Notice that these blueprints do not offer a complete architecture of a space mission as a whole. This stems from the need to make this toolkit mission-agnostic. The objects above offer generic models, usable to all missions as they aim to only provide unambiguous object representations. The blueprints are building blocks to speed up the development of more analysis-driven models. They must be viewed as inputs rather than outputs. For example: a solar panel might be considered a generic attribute of a space mission; however, different methods with different fidelities exist in modelling the panel. Implementing one model would impose assumptions and limitations that may not align with the needs of a specific mission, reducing flexibility and reusability. Rather, an unambiguous model of a geometrical plane can be a useful input to this custom solar array model. This way, the user can focus on modelling the solar panel's behaviour, and let the toolkit offer the needed time-dependent inputs like: spacecraft orientation, eclipse windows, solar angle of incidence on the plate, and the solar flux at the panel location.

The package's architecture is illustrated in Figure 4.5. Notice that the modules are divided into three categories: the Simulation module, modules containing spacecraft system description classes, and modules containing auxiliary information about the environment around the spacecraft (celestial bodies and ground stations). To set up a simulation, the minimum required configuration is to create a simulation instance and a spacecraft actor. The other modules are optional, but can provide useful functionalities for the system simulator.

The modules containing the actor classes from the list above are connected directly to the simulation instance (spacecraft, celestial bodies, and ground stations). Their states are automatically updated when the simulation time is changed. Within the spacecraft system, the geometry module is connected to the spacecraft actors directly, allowing the user to configure spacecraft-specific geometries and help generate models of the spacecraft subsystems.

In this chapter, each of the modules and classes will be discussed in more detail, covering their purpose, functionalities, and how they fit into the overall toolkit architecture. The simulation base class is covered in subsection 4.4.1, celestial bodies in subsection 4.4.2, ground stations in subsection 4.4.3, the spacecraft class in subsection 4.4.4, and geometrical objects in subsection 4.4.5.

4.4.1. Simulation Base Class

The simulation base class is the core utility class of the toolkit for usage in the user's execution loops. It is responsible for managing the simulation time, which is inherited by all actors associated to the simulation instance.

The utility of the simulation base class can be divided in three categories:

1. **SPICE Kernel Management:** loading SPICE kernel and automatically extracting useful metadata, helping the user to configure their simulation scripts.
2. **Simulation Time Management:** managing the simulation time across all associated actors, updating their states when the time is changed.

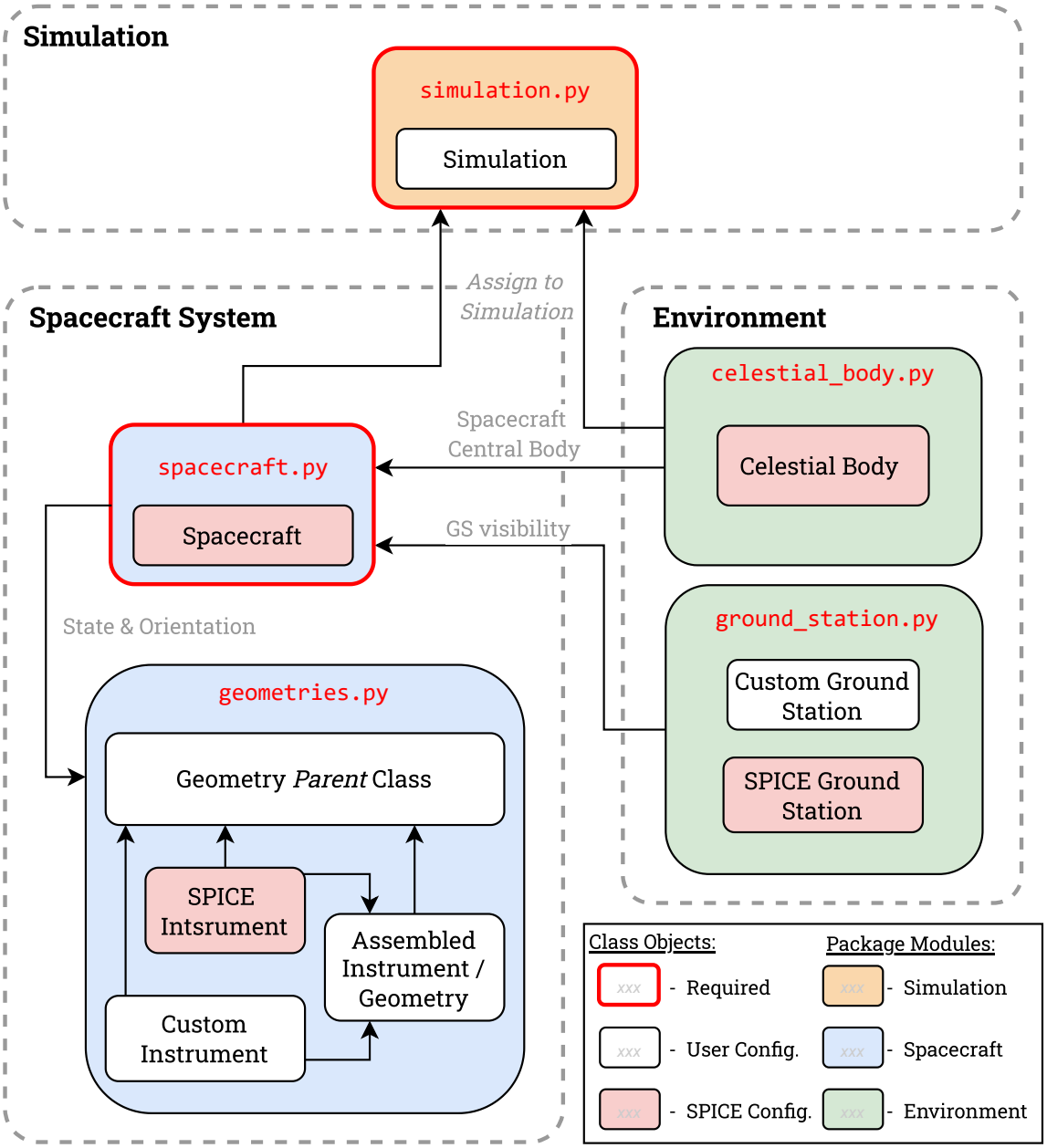


Figure 4.5: Toolkit module architecture

3. **Simulation Configuration:** providing methods to configure default simulation parameters to be used within the methods of actors associated to the simulation instance.
4. **Model Wrapping:** providing an interface to add custom callable models to the simulation environment.

A typical script with PAS3 usage starts with creating a simulation instance and loading the necessary SPICE kernels of the mission. Listing 1 shows an example of how to create a simulation instance with the (optional) input arguments of the SPICE *kernel paths*, *start time* and *end time*. Time formats accepted are various Coordinated Universal Time (UTC), Julian, calendar strings (like "YYYY-MM-DDTHH:MM:SS" or "YYYY month DD, HH:MM:SS"), Barycentric Dynamical Time (TDB) seconds past J2000, or a Python datetime object. This uses the underlying SPICE (SpiceyPy) routines `str2et`, `utc2et` and `datetime2et` to convert the input time formats to Ephemeris Time (ET), TDB seconds past J2000 which is the time format used internally in the simulation and its actors.

It is recommended to initialize separate simulation instances if simulating multiple actors with different time coverages in their state data, using different time steps in the same script or different default configuration parameters. Loading SPICE kernels in one simulation instance does not prevent other simulation instances from using the same kernels. SPICE kernels are loaded in a global kernel pool in your system's memory; if at some other point in the user's script, another simulation instance is created, the same kernel pool will be

```python
import pas3 as p3

# ================================== PAS3.Simulation ====================================
sim = p3.Simulation(
    kernel_paths=["path/to/lsk_file.tls", "path/to/spk_file.bsp", ...], # paths to kernels
    start_time="2002-02-04T08:30:00",    # (optional) simulation start time
    end_time="2025-08-21T12:17:15",      # (optional) simulation end time
)
```

Listing 1: Example of a simulation instance creation and SPICE kernel loading

used. However, the simulation instances have internal logic to store time coverages as utility for the user's simulation loops. It is therefore recommended to use separate simulation instances with separate SPICE kernels if different time coverages are required, or if different default configuration parameters are needed.

If the optional input arguments *start time* and *end time* are not specified, the simulation instance will automatically set the start and end times to the shortest coverage of all loaded SPK actors in the simulation instance. The start and end times are attributes to control the simulation's behaviour in the user's execution loops. When the simulation time is outside the configured start and end times interval, the simulation instance will be set to `active = False`, and actors' states will stop updating. This prevents the script from raising SPICE coverage errors, and results in more robust and clean simulation loops.

An example of the minimal setup of a simulation instance (i.e. providing a SPICE MK and no start and end time) is shown in Listing 2. When a new simulation instance is created, the loaded SPICE kernels are listed, and the simulation instance is automatically configured with the earliest start time and latest end time of the loaded SPK data.

```python
import pas3 as p3

# ===================== PAS3.Simulation - minimal setup using SPICE MK ====================
sim = p3.Simulation("envision_kernels/mk/envision_study_et1_2031_north_voi_ml014.tm")
```

Outputs:

```
Total number of loaded kernels: 18
-----------------------------------------------------------
Loaded SPICE kernels:
-----------------------------------------------------------
KERNEL:         TYPE:           NAME:
Kernel 1        Text    MK      envision_study_et1_2031_north_voi_ml014.tm
Kernel 2        Text    FK      envision_v03.tf
Kernel 3        Text    FK      estrack_v04.tf
...             ...     ...     ...
Kernel 17       Binary  SPK     EnVision_ET1_2031_NorthVOI_ML014_340611_380817_v01.bsp
Kernel 18       Binary  CK      EnVision_ET1_2031_NorthVOI_ML014_v01.bc
-----------------------------------------------------------

The simulation is temporally bound by the loaded SPK data of 'ENVISION' (NAIF CODE: -668).
Coverage: 2031-11-21T04:05:34.060 to 2038-08-17T13:14:28.003.

No start time given.
Setting the simulation start time to 2031-11-21T04:05:34.060

No end time given.
Setting the simulation end time to 2038-08-17T13:14:28.003
```

Listing 2: Example of setting up an PAS3 simulation instance for the EnVision mission, loading a SPICE MK file without providing start and end time.

The simulation base class provides a set of methods to let the user more easily extract data from the simulation instance. This is useful to help the user configure the other actors covered in the next sections. The method `list_loaded_spk_actors()` lists all the loaded SPICE actors with positional (and possibly velocity) data in the loaded kernels (see Listing 3). The method `get_spk_coverage(. . . )` provides a way to extract the coverage of each loaded actor (see Listing 4). Similar functionality is also provided for reference frames and actors with attitude stored in CK files. Using these methods, the user can easily find the NAIF code and name of the actors, check if they are loaded, and use them in configuring their PAS3 representations.

Conventional SPICE metadata extraction methods for binary kernels conventionally require command-line executable programs (like brief and ckbrief), or multiple SpiceyPy functions to run in sequence. The built-in SPICE kernel management methods of the simulation base class provide a more user-friendly way to understand the loaded SPICE data and use it.

```
sim.list_loaded_spk_actors()
```

Outputs:

```
--- Listing Actors stored in SPK files: ---
NAIF CODE       NAME:                 SPK FILE:
        1       MERCURY BARYCENTER    de432s.bsp
        2       VENUS BARYCENTER      de432s.bsp
        3       EARTH BARYCENTER      de432s.bsp
        4       MARS BARYCENTER       de432s.bsp
        5       JUPITER BARYCENTER    de432s.bsp
        6       SATURN BARYCENTER     de432s.bsp
        7       URANUS BARYCENTER     de432s.bsp
        8       NEPTUNE BARYCENTER    de432s.bsp
        9       PLUTO BARYCENTER      de432s.bsp
       10       SUN                   de432s.bsp
      199       MERCURY               de432s.bsp
      299       VENUS                 de432s.bsp
      301       MOON                  de432s.bsp
      399       EARTH                 de432s.bsp
   398990       NEW_NORCIA            estrack_v04.bsp
   398991       NEW_NORCIA_2          estrack_v04.bsp
   399500       KIRUNA1               estrack_v04.bsp
      ...       ...                   ...
     -668       ENVISION              EnVision_ET1_2031_eON_LPO_ML014_311121_330522_v01.bsp
     -668       ENVISION              EnVision_ET1_2031_NorthVOI_ML014_340611_380817_v01.bsp
```

Listing 3: Example of listing the loaded SPK actors in the simulation instance (truncated for readability).

```
sim.get_spk_coverage(-668)
```

Outputs:

```
--- SPK coverage for 'ENVISION' (NAIF CODE: -668): ---

'EnVision_ET1_2031_eON_LPO_ML014_311121_330522_v01.bsp' coverage:
  Start: 2031-11-21T04:05:34.060
  Stop:  2033-05-22T03:00:31.375

'EnVision_ET1_2031_NorthVOI_ML014_340611_380817_v01.bsp' coverage:
  Start: 2034-06-11T13:22:36.569
  Stop:  2038-08-17T13:14:28.003
```

Listing 4: Example of coverage extraction for a loaded SPK actor (EnVision in this case).

Besides loading SPICE kernels, the simulation base class provides methods designed to be used within

simulation loops. The actors in the following sections must always be assigned to a simulation instance, as they inherit the simulation time and configuration parameters from this instance. Every spacecraft and celestial body actor has internal functions to update their state (`update_state()`) to store default state variables as class attributes. At every simulation time change, these functions are called for every assigned actor. The method `advance_time(dt)` advances the simulation time by a time step `dt` (in seconds), and the method `set_time(time)` sets the simulation time to a specific time. These methods are intended to be used in the user's execution loops. Additionally, a progress bar from the `tqdm` library can be added to follow the simulation progress. An example is shown in Listing 5.

```
# Optional: set the simulation time before starting the simulation loop.
sim.set_time(some_start_time)

# Optional: add a progress bar
sim.add_progress_bar(dt)

while sim.active:
    # ======================
    # User's execution loop.
    # Add analysis code here.
    # ======================

    # Advance the simulation time by a time step dt:
    sim.advance_time(dt)
    # Loop will stop when the simulation time exceeds the configured end time.
```

Listing 5: Example of a simulation instance's use in a user's execution loop.

As mentioned in section 4.2, the SPICE toolkit is a procedural toolkit; this means it provides the user with functions to extract geometrical state data from globally loaded kernels. These functions often require multiple input parameters. Take for example the CSPICE `occult_c` routine (in SpiceyPy: `spiceypy.occult`), which takes as input arguments:

`target1`: name or NAIF ID of the first target body.

`shape1`: shape model of the first target body.

`frame1`: name of the body-fixed reference frame of the first target body.

`target2`: name or NAIF ID of the second target body.

`shape2`: shape model of the second target body.

`frame2`: name of the body-fixed reference frame of the second target body.

`abcorr`: aberration correction flag to be applied.

`observer`: name or NAIF ID of the observer body.

`et`: observation epoch in ET (seconds past J2000).

The routine outputs an integer between -3 to 3, indicating the occultation state between the two target bodies as seen from the observer at the specified epoch. -3 represents a total occultation of target 1 by target 2, -2 an annular occultation, -1 a partial occultation, 0 no occultation. Flipping the signs flips the targets. This routine offers a very procedural approach to extracting occultation information; it requires some knowledge of SPICE conventions, namings, frames shape models, and aberration corrections. Often, in system-level mission analysis, the interest is simply on whether the spacecraft is eclipsed or its line of sight with a ground station is not obstructed. PAS3 aims to provide the user with a more object-oriented approach with methods like `sees(<object>)`, automatically using the given object's attributes, spacecraft's occulting bodies' attributes as input parameters, and the simulation's default configuration parameters (like aberration correction and occultation tolerance) to provide a more user-friendly solution. PAS3 automatically checks the number of occulting bodies to consider (Earth-eclipses around a Jovian moon would consider both the host moon, Jupiter and the Sun as occulting bodies), assigns their body-fixed frames and shape models (ellipses or points) based on distance. The user can configure the tolerance that defines an "eclipse" using the simulation base class `set_occultation_tolerance(...)` method.

Similarly, the user can configure parameters to be used simulation-wide, like SPICE's aberration correction (light time and stellar aberration corrections for apparent positions) and a default inertial reference frame for state outputs, and the solar constant to be used. This is done using the methods `set_aberration_correction(. . . )`, `set_inertial_frame(. . . )` and `set_solar_constant(. . . )`. Configuring these default parameters reduces the number of input parameters required, making the object-oriented state extraction methods more straightforward and less procedural.

The final use of the simulation base class is the `add_custon_model(. . . )` method. This method provides a way to add a callable function (i.e. a user-defined model) to the simulation instance. The custom model will be called at every time change and is intended to update dynamic attributes assigned to the simulation's actors or custom class objects in the user's script. This provides a way to include project-specific models in the base configuration file of the simulation, without the need for other project contributors to manually call these models in their own execution loops in their analysis scripts. Required input arguments of the custom models, once added, are inherited by the simulation instance's `advance_time(. . . )` and `set_time(. . . )` methods. The inclusion of custom models will be explored in more detail in chapter 5.

All methods and properties of the simulation base class are summarized in Table A.1 in Appendix A. For class attributes, the code documentation must be consulted.

4.4.2. Celestial Body Object

The *CelestialBody* class represents a celestial body in the simulation. This can be a planet, moon, asteroid, comet, or the Sun. When initializing a spacecraft, one celestial body must be assigned as the spacecraft's central body. PAS3 automatically initializes its central body as a *CelestialBody* object as well (host planet for moons, the Sun for planets). Ephemeris data, reference frames and constants for the celestial bodies are stored in SPK, FK and PCK kernels. The standard kernels provided by JPL NAIF (like "development ephemerides" deNNN.bsp and outerplanets_vNNN.bsp for planetary ephemeris data with N being the version number) cover all major celestial bodies in the solar system and are regularly updated. This makes the initialization of celestial bodies in PAS3 very straightforward, as long as the necessary kernels are loaded in the simulation instance. In Essence, the *CelestialBody* class is a wrapper around SpiceyPy functions to extract the body's state and physical parameters straightforwardly.

The *CelestialBody* class stores general attributes (mass, gravitational parameter, shape) and provides methods to extract the body's position and velocity relative to its parent body, the spacecraft, or any other body in the simulation. It also provides methods to extract the body's orientation (body-fixed rotation matrix) relative to the default frame of the simulation or a custom one. Methods for extracting the body's apparent limb from an observer distance are also provided. It mostly serves as a utility class to provide the *Spacecraft* class with the necessary state data of its central body and other occulting bodies in the simulation, for more spacecraft-oriented methods (ground track, albedo, body-fixed position, etc.).

An example of initializing a *CelestialBody* instance of the Jovian moon Ganymede is shown in Listing 6. The only required input argument is the name or NAIF ID of the body and the simulation instance it is associated to (for automatic state updates). The necessary SPICE kernels (LSK for time conversions, SPK for inner solar system and outer solar system ephemeris data, FK for non-default Jovian moon SPICE reference frames, and PCK for planetray constants) must be loaded in the simulation instance.

All methods and properties of the *CelestialBody* class are summarized in Table A.2 in Appendix A. For class attributes, the code documentation must be consulted.

```python
import pas3 as p3

# ==================================== PAS3.Simulation ====================================
sim = p3.Simulation(["naif0012.tls", "de432s.bsp", "jup365_19900101_20500101.bsp",
    "rssd0002.tf", "pck00010.tpc","gm_de431.tpc"])

# ================================== PAS3.CelestialBody ===================================
Ganymede = p3.CelestialBody(
    naif_name_or_code="GANYMEDE",    # Ganymede's NAIF name or code (503)
    simulation=sim                   # PAS3 Simulation object to assign the body to
)
```

Outputs:

```
Total number of loaded kernels: 6
-----------------------------------------------------------
Loaded SPICE kernels:
-----------------------------------------------------------
KERNEL:         TYPE:           NAME:
Kernel 1        Text    LSK     naif0012.tls
Kernel 2        Binary  SPK     de432s.bsp
Kernel 3        Binary  SPK     jup365_19900101_20500101.bsp
Kernel 4        Text    FK      rssd0002.tf
Kernel 5        Text    PCK     pck00010.tpc
Kernel 6        Text    PCK     gm_de431.tpc
-----------------------------------------------------------
...
Loading central body JUPITER (599) for GANYMEDE (503).
Loading central body SUN (10) for JUPITER (599).
--- Initialized the Sun ---
--- Initialized the Planet 'JUPITER' as celestial body with central body 'SUN' ---
--- Initialized the Satellite 'GANYMEDE' as celestial body with central body 'JUPITER' ---
```

Listing 6: Example of setting up an PAS3 Celestial Body instance of the Jovian moon Ganymede, automatically initializing additional occulting bodies (truncated for readability).

4.4.3. Ground Station Objects

Another utility class in the PAS3 toolkit is the *GroundStation* class. This class represents an Earth-based ground station in the simulation. The SPICE toolkit offers purely geometric ground station functionality, specifying the station's position in SPK files and frame transformations in FK files. There is no inherent "visibility" function, the SPICE toolkit offers a set of low-level functions to extract the position of an SPK-based actor relative to a ground station's topocentric frame of reference. The topocentric reference frame is a non-inertial local horizon frame, with the z-axis pointing upwards (normal to the geodetic surface), the x-axis pointing towards the local north, and the y-axis completing the right-handed frame (see Figure 4.6) [43].

PAS3 offers two class blueprints for setting up ground-stations. The parent *GroundStation* class is inaccessible for users to configure; instead, the user must choose between initializing a ground station using SPICE kernels (using the *SpiceGroundStation* child class) or a custom ground station using geodetic coordinates as input parameters (using the *CustomGroundStation* child class). The custom ground station is partially SPICE-based, using Earth's shape, relative positioning and rotation with the help of SPICE routines. Positioning the ground station, topocentric frame transformations and local horizon calculations are performed internally.

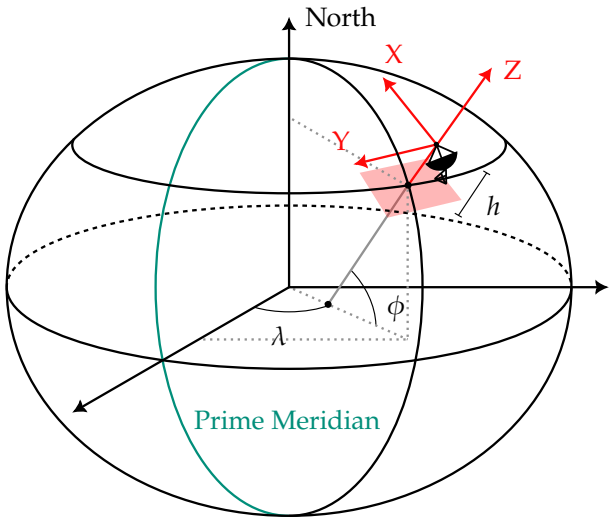


Figure 4.6: Ground station's topocentric reference frame positioned on an ellipsoidal Earth with geodetic longitude λ and latitude ϕ, positioned at an altitude h above the reference ellipsoid.

Both *SpiceGroundStation* and *CustomGroundStation* classes provide the same methods to extract visibility information with one or more spacecraft (or even celestial bodies). The main method is `groundstation.is_visible(<object>)` which returns a boolean indicating whether or not the ground station has line of sight with the given object. Local azimuth and elevation angles of objects w.r.t. the station can be extracted, as well as its topocentric direction vector. As the SPICE toolkit does not provide inherent visibility functions, PAS3 automatically sets a minimum elevation angle of 5° [44] over the local horizon for the visibility flag as a conical visibility mask. However, a custom visibility mask may be configured by the user as a function taking azimuth and elevation angles as input parameters, returning a boolean visibility flag. This can be done using the method `set_visibility_mask(<callable>)`. This way, the user can implement custom visibility masks based on local terrain, obstacles, or other site-specific constraints.

A ground station's reference frame has a fixed position and orientation w.r.t. the Earth's body-fixed rotating frame. SPICE offers built-in body-fixed reference frames for major Solar System bodies defined by the International Astronomical Union (IAU), like "IAU_EARTH" for Earth and "IAU_VENUS" for Venus. However, these frames do not account for polar motion, true sidereal time, precession and nutation effects. It is generally recommended to use high-accuracy terrestrial SPICE rotation models when working with the Earth [45].

The default body-fixed frame used in the *GroundStation* classes is the "EARTH_FIXED" frame, which is an alias for the body-fixed terrestrial frame for either the high-accuracy "ITRF93" or lower-accuracy "IAU_EARTH" frame, depending on the loaded FK alias kernel. The *GroundStation* class has the optional input parameter `earth_fixed_frame` to specify the body-fixed frame to be used. This allows the user to prioritize either accuracy or execution speed. When using a high-accuracy orientation frame, this frame must be loaded as a binary SPICE PCK file before initializing a ground station object.

```python
import pas3 as p3

# =================================== PAS3.Simulation ====================================
sim = p3.Simulation(["naif0012.tls", "de432s.bsp", "estrack_v04.bsp","estrack_v04.tf",
 ↪  "pck00010.tpc", "earth_000101_251120_250825.bpc"])

# ====================== PAS3.SpiceGroundStation - (2 input arguments) ====================
NewNorcia = p3.SpiceGroundStation(
    naif_name_or_code="NEW_NORCIA",     # New Norcia's NAIF name or code (398990)
    simulation=sim                      # PAS3 Simulation object to assign the station to
)

# ==================== PAS3.CustomGroundStation - (6 input arguments) ====================
CustomNewNorcia = p3.CustomGroundStation(
    name="Custom New Norcia station",   # Name of the ground station
    simulation=sim,                     # PAS3 Simulation object to assign the station to
    longitude=116.19,                   # Geodetic longitude [°]
    latitude=-31.05,                    # Geodetic latitude [°]
    altitude=0.252,                     # (optional) Altitude above the Earth ellipsoid [km]
    earth_fixed_frame="ITRF93"          # (optional) Earth body-fixed SPICE frame to use
)
```

Outputs:

```
Total number of loaded kernels: 7
-----------------------------------------------------------
Loaded SPICE kernels:
-----------------------------------------------------------
KERNEL:         TYPE:           NAME:
Kernel 1        Text    LSK     naif0012.tls
Kernel 2        Binary  SPK     de432s.bsp
Kernel 3        Binary  SPK     estrack_v04.bsp
Kernel 4        Text    FK      estrack_v04.tf
Kernel 5        Text    FK      earthfixeditrf93.tf
Kernel 6        Text    PCK     pck00010.tpc
Kernel 7        Binary  PCK     earth_000101_251120_250825.bpc
-----------------------------------------------------------
...
--- Initialized a SPICE-defined ground station 'NEW_NORCIA' at longitude 116.19146780000001°
 ↪  and latitude -31.0482254° inside EARTH_FIXED frame ---
--- Initialized a custom ground station 'Custom New Norcia station' at longitude 116.19° and
 ↪  latitude -31.05° inside ITRF93 frame ---
```

Listing 7: Example of setting up PAS3 ground station instances for the New Norcia deep space ground station using both the *SpiceGroundStation* and *CustomGroundStation* classes (truncated for readability).

In Listing 7, an example of initializing a ground station for the New Norcia deep space ground station is shown. The first instance uses the *SpiceGroundStation* class, using the SPICE-defined name "NEW_NORCIA" (NAIF ID: 398990) as input parameter. The second instance uses the *CustomGroundStation* class, specifying the geodetic coordinates of the station and using the high-accuracy ITRF93 terrestrial frame (equivalent to the default "EARTH_FIXED" frame alias used in the first instance). The necessary SPICE kernels are loaded in the simulation instance (LSK for time conversions, SPK for planetary ephemeris data and ESTRACK ground station positions, FK for ground station topocentric frames and the "EARTH_FIXED" association alias, and PCK files for Earth's shape and "ITRF93" high-accuracy binary orientation model).

All methods and properties of the *GroundStation* classes are summarized in Table A.3 in Appendix A. For class attributes, the code documentation must be consulted.

4.4.4. Spacecraft Object

The main focus of the PAS3 toolkit is the *Spacecraft* class. This class represents a spacecraft in the simulation. It holds the main functionalities of the toolkit, aiming to serve as a foundation in system modelling and simulator scripts. The class blueprint holds ... main functionalities:

1. **State accessing:** providing methods and properties to access the spacecraft's state and derived orbital parameters.
2. **Attitude handling:** providing methods to configure the spacecraft attitude and access spacecraft body-fixed frame transformations.
3. **Visibility analysis:** providing methods to check visibility with ground stations and celestial bodies for occultation and eclipse events.

The *Spacecraft* class is designed to provide descriptive methods to access the spacecraft's state and derived parameters. This is to improve accessibility to integrate a spacecraft mission simulation as a solid backend of system and mission analysis scripts. The object-oriented approach, together with informed initial configurations of default parameters, frames and celestial bodies provide minimal input parameters to the user, reducing the procedural complexity of using the SPICE toolkit directly.

When it comes to accessing the spacecraft's state, its position is naturally the most basic attribute. In PAS3, the spacecraft's position (and velocity) is based on its ephemeris data stored in SPK files, accessed with the help of the SpiceyPy routine: `position = spiceypy.spkpos(targ, et, ref, abcorr, obs)[0]`, the *Spacecraft* class wraps this routine in the method `get_position(frame)`, internally filling input arguments with existing object attributes (like targ=spacecraft's NAIF name, et=simulation time, abcorr=simulation's aberration correction setting). The reference frame input argument is optional, defaulting to the simulation's inertial reference frame (equivalent to the simple `spacecraft.position` attribute). Within PAS3, the spacecraft position and velocity are always expressed with respect to its central body. The central body can be reconfigured using the method `set_central_body(<CelestialBody>)`.

For classical orbital parameters, the *Spacecraft* class calls the SpiceyPy routine `spiceypy.oscltx(state, et, mu)` internally, which outputs an extended set of classical conic elements (11 output elements) using internal energy and momentum calculations. To stay in line with the minimal OOP approach for PAS3 while minimizing the effect on execution performance, every conical element gets assigned its own property and the underlying routine is called once when the user accesses a conical property. The conical elements are only recalculated when the simulation time (and thus the spacecraft's state) advances and the user calls a conical property. In order to get orbital elements, the spacecraft's central body must have a gravitational parameter initialized in its PCK or manually (with `set_gravitational_parameter(. . . )`).

Additionally, the class provides methods to access relative positions, directions and distances to other actors of the simulation instance. This can be done with respect to other *Spacecraft*, *CelestialBody* or *GroundStation* classes in any given SPICE reference frame (internal methods solve *CustomGroundStation*'s SPICE compatibility). Some positions, directions and distances are standardized and can be accessed directly without specifying the target object, like: the Sun, Earth and spacecraft's central body (i.e. nadir direction). These standard vectors are expected to be called frequently within implementation interfaces of the class instance and are therefore called and stored in their respective class attributes whenever the spacecraft's state is updated. Effort was put in storing some frequently used variables as attributes and others as properties or methods; calling spice routines multiple times in a single time step can be avoided this way and help improve execution performance.

In section 4.1, it is mentioned how attitude is not configured using SPICE by default. The *Spacecraft* class enables CK attitude functionality using the method `use_spice_attitude(<bool>)`. When set to `True`, the spacecraft's attitude is extracted from loaded CK files SpiceyPy routines. When set to `False`, the spacecraft's

attitude is initialized to the identity matrix (aligned with the *Simulation* default frame) and can be reconfigured manually using the method `construct_body_rotation_matrix(<x, y, z>)` with the x, y, z axes expressed in the configured default frame as input parameters. This way, the user can implement custom attitude models without the need to create a CK file. In later versions, quaternions and Euler angles compatibility should be added. This SPICE disconnect in attitude determination complicates accessing previous state variables expressed in the body-fixed frame, as no standard SPICE routines can be used for this. The methods `inertial_to_body(<vector>)` and `body_to_inertial(<vector>)` provide a way to express vectors between the inertial default frame and the spacecraft's body-fixed frame. These methods are used internally for accessing body-fixed directions, like the spacecraft's nadir, Sun, Earth and PAS3 actor directions and positions.

Using these state-relative descriptions, the *Spacecraft* class provides methods to check visibility with other actors in the simulation. The main method is `is_visible(<object>)` which returns a boolean indicating whether or not the spacecraft has line of sight with the given object. Similarly to positional and directional methods, Earth and Sun visibility are standardized and can be accessed directly in their own methods (`is_earth_visible()` and `is_sun_visible()`). The *Spacecraft* class does internal automatic method dispatching to select the correct number of occulting bodies to check depending on the configured central body type of the spacecraft. The output visibility flag depends on the simulation's occultation tolerance setting (see subsection 4.4.1).

When it comes to ground stations, every *Spacecraft* instance can be assigned to one or more *GroundStation* instances using the method `assign_ground_station(<GroundStation>)`. The ground stations are stored in a dictionary with their names as keys. Using the previous occultation handling methods, the user can check if the spacecraft is in a position where it can have an unobstructed line of sight with the ground station (satisfying its custom visibility mask) using the method `is_in_communication_window(<GroundStation>)`; accessing the visible ground stations with `list_visible_ground_stations()`.

The final type of built-in class methods for the *Spacecraft* class are methods related to its central body. The spacecraft's central body is given as an input argument directly (as *CelestialBody* instance) or indirectly (as NAIF name of the body). At every simulation time, the user can access the spacecraft's ground track coordinates w.r.t. its central body's fixed reference frame in either planetocentric or planetodetic latitude, longitude and altitude or as a plain vector expressed in an arbitrary SPICE reference frame. Additionally, the user can access properties of the apparent limb of the central body as seen from the spacecraft's position, useful for field-of-view, albedo and observations analysis. The `get_limb_ellipse(...)` outputs the limb ellipse parameters (semi-major axis, semi-minor axis, centre position) using the `SpiceyPy.edlimb()` routine; the half-angle of the cone encircling the limb disk can be accessed using the `approx_limb_half_angle_in_rad()` for the central body and `sun_half_angle_in_rad()` for the Sun disk. These are mainly used for field-of-view analyses for the upcoming *Geometry* class (see subsection 4.4.5).

An example of initializing a *Spacecraft* instance of the SMART-1 lunar orbiter is shown in Listing 8. The only required input arguments are the name or NAIF name or code of the spacecraft, the simulation instance it is associated to (for automatic state updates), and its central body as a *CelestialBody* instance or a NAIF name or code. In the example, a standard MK file (SMART1_OPS.tm) is loaded to initialize all required kernels. For a *Spacecraft* class, the minimal required kernels are LSK for time conversions and SPK for ephemeris data (of spacecraft and celestial bodies). For kernel-based attitude functionality, a CK and accompanying FK kernel must be loaded to specify the dynamic body-fixed reference frame of the spacecraft.

```python
import pas3 as p3

# ==================================== PAS3.Simulation ====================================
sim = p3.Simulation("SMART1_OPS.tm", "2004-11-13T00:00:10")

# ================================== PAS3.CelestialBody ===================================
Moon = p3.CelestialBody("MOON", sim)

# ==================================== PAS3.Spacecraft ====================================
Smart1 = p3.Spacecraft(
    naif_name_or_code="SMART1",     # SMART-1's NAIF name or code (-238)
    central_body=Moon,              # SMART-1's central body as CelestialBody instance.
    simulation=sim                  # PAS3 Simulation object to assign the spacecraft to
)
```

Outputs:

```
Total number of loaded kernels: 29
-------------------------------------------------------------
Loaded SPICE kernels:
-------------------------------------------------------------
KERNEL:         TYPE:           NAME:
Kernel 1        Text    MK      SMART1_OPS.tm
...             ...     ...     ...
Kernel 29       Binary  SPK     ESTRACK_V01.BSP
-------------------------------------------------------------
...
Loading central body EARTH (399) for MOON (301).
Loading central body SUN (10) for EARTH.
--- Initialized the Sun ---
--- Initialized the Planet 'EARTH' as celestial body with central body 'SUN' ---
--- Initialized the satellite 'MOON' as celestial body with central body 'EARTH' ---

Found SPICE 'CK-based' body-fixed frame 'SMART1_SPACECRAFT' (ID: -238000) for the spacecraft
 ↪  'SMART1' (ID: -238)
--- Initialized Spacecraft actor 'SMART1' with central body 'Moon' ---
```

Listing 8: Example of setting up an PAS3 Spacecraft instance of the SMART-1 lunar orbiter, automatically initializing its central body and additional celestial bodies (truncated for readability).

4.4.5. Geometry Objects

The *Geometry* class blueprint represents spatial regions that are either infinite (e.g. half-spaces) or conical (e.g. fields of view). These are unbounded regions in space (no bounded geometries like cubes and spheres) that are physically attached to a specific spacecraft actor within the simulation. The *Geometry* class is the parent class of the many geometry types with inherently the same unbounded, infinite or conical nature. Together, they are all bounded in the `geometries.py` module of the PAS3 package. The class inheritance hierarchy is shown in Figure 4.7. The module provides the user with five class blueprints:

- *Geometry*: Superclass for all geometry types, accessible to the user to configure non-instrument geometries.
- *SpiceInstrument*: subclass for SPICE-based instrument geometries.
- *CustomInstrument*: subclass for user-defined custom instrument geometries.
- *AssembledGeometry*: subclass for combining multiple geometries in one geometry.
- *AssembledInstrument*: subclass of the *AssembledGeometry*, combining multiple geometries in one instrument.

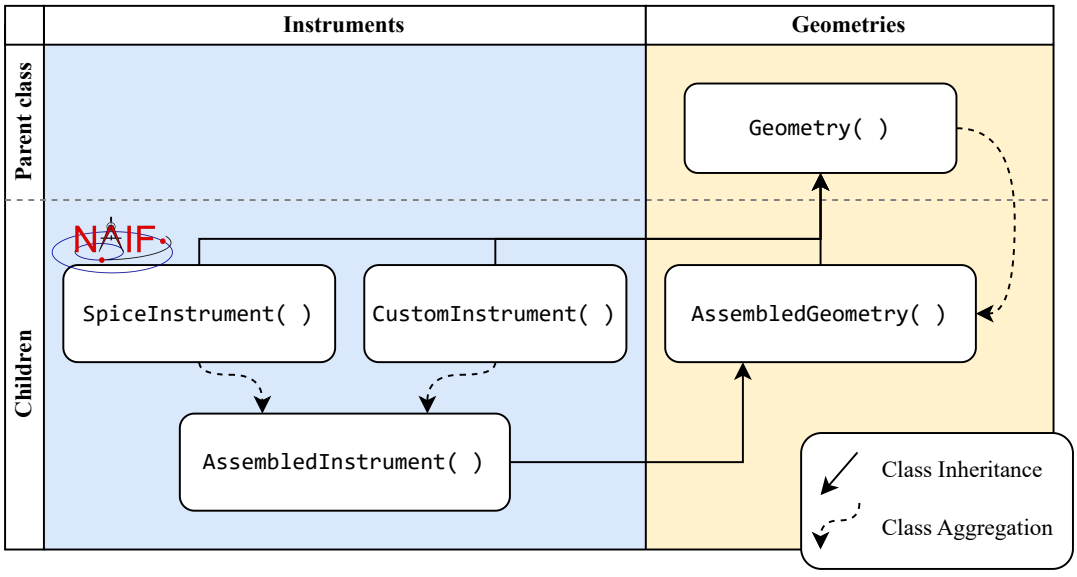


Figure 4.7: Class structure in the PAS3 geometries module. Superclass (parent) and subclass (child) inheritance is depicted, with assembled geometries being composed of class aggregations.

Every *Geometry* type class must provide a *Spacecraft* instance as input argument, to which the geometry is physically attached. The geometry's position and orientation are inherited from the spacecraft's body-fixed frame. The *Spacecraft* instance will hold two dictionary attributes (`geometries` and `instruments`) to store the geometries based on their type. Within PAS3, the only difference between a geometry and an instrument is the dictionary the geometry is stored in within the spacecraft instance; this allows the user to more easily loop through instruments and geometries separately (depending on the use case).

The main functionality of the *Geometry* classes is to provide visibility methods to check if a given target object in the simulation (another spacecraft or celestial body) is inside the geometry's spatial region. This is provided by predicate methods returning boolean flags if the object is inside or intersects the geometry region. Celestial bodies as target objects can be treated as spheres (using their mean radius as a simplification of their shape) with the method `sees_celestial_body(<CelestialBody>)` or built-in: `sees_central_body()` for the spacecraft's central body and `sees_the_sun()` for the Sun. These methods use the given *CelestialBody* instance's apparent limb from the spacecraft's perspective (see subsection 4.4.2) and computes built-in intersection algorithms with the circular cone encapsulating the limb disk. Internally, this method uses the `sees_circular_cone(<vector>, <half_angle>)` function, which checks if a given circular cone (defined by its apex direction in body-fixed frame and its half-angle) intersects or is contained by the geometry's spatial region. Spacecraft, ground stations and far-away celestial bodies can be treated as points with the method `sees_point_object(<object>)`. This method is less computationally expensive, internally computing encapsulation of the position vector within the geometry's spatial bounds using the `sees_ray(<vector>)` function. All these visibility conditions depend on the *Spacecraft* instance's occultation checks of the associated spacecraft of the geometry.

The predicate visibility methods of PAS3's geometry classes are a good example of the internal interdependency of methods between the PAS3 classes. An example code snippet of the `sees_celestial_body(<CelestialBody>)` method is shown in Listing 9 to illustrate this. First, it is checked if the parent *Spacecraft* instance is not occulted by other celestial bodies in the simulation (using the `spacecraft.is_object_visible(<CelestialBody>)` method). The limb of the target celestial body is required to create the encapsulating cone for the visibility check. The target body, being of the type *CelestialBody*, provides the necessary method to extract its apparent limb half-angle from the spacecraft's distance (a `Spacecraft` method). Finally, the geometry's own method is called to check (purely mathematically) if the encapsulating cone intersects or is contained by the geometry's spatial region. The `Simulation` involvement instance is not directly apparent, but stores the occultation tolerance setting used in the spacecraft's occultation checks, the relative positions of actors with light time correction flags, and provides the internal simulation time for all actor state updates.

```python
def sees_celestial_body(self, target_object: CelestialBody) -> bool:
    """Function to check if the geometry sees a planetary body. The planetary body is assumed to be spherical (encapsulating the elliptical limb).
    Args:
        target_object (CelestialBody): target body.
    Returns:
        bool: True if target body approximate limb intersects / is contained by the geometry.
    """
    # Check if the target object is visible from the spacecraft (not occulted).
    if not self.spacecraft.is_object_visible(target_object):
        return False

    # Get the direction vector from the apex of the cone to the target object
    vector = normalize(self.spacecraft.get_object_position_in_body_fixed_frame(target_object) - self.body_location)

    # Get the half-angle of the cone encapsulating the body's limb seen from the spacecraft.
    half_angle = target_object.get_approximate_limb_half_angle_in_rad(
        observer_distance=self.spacecraft.get_object_distance(target_object=target_object)
        )

    return self.sees_circular_cone(vector, half_angle)
```

Listing 9: Code snippet of the celestial body visibility method of the *Geometry* classes, showing the internal interdependency between the PAS3 classes' functions.

Five geometry shapes are supported in the current version:

"CIRCLE":	A circular infinite viewing cone, defined by its axis (boresight) and half-angle.
"RECTANGLE":	A rectangular viewing cone, defined by its four boundary corner vectors.
"POLYGON":	A polygonal viewing cone, defined by an arbitrary number of boundary corner vectors.
"HALF-SPACE":	An infinite half-space, defined by its normal vector.
"RAY":	A vector ray, defined by its direction vector.

The "CIRCLE", "RECTANGLE" and "POLYGON" shape names are taken from the SPICE instrument geometry definitions [46], the SPICE shape "ELLIPSE" is not yet supported by PAS3. The "HALF-SPACE" and "RAY" shapes are custom PAS3 implementations; they have no SPICE-instrument equivalent.

The base *Geometry* class, as well as the *CustomInstrument* subclass, are initialized fully by the user. The user must provide the geometry's name, shape type, and necessary shape parameters, like half-angle, (corner) vertices or a boresight / axis vector. An example of initializing a custom instrument is shown in Listing 10. The exact same input parameters are used in the *CustomInstrument* subclass; only difference being its classification.

```
geometry = PAS3.Geometry(   # PAS3.CustomInstrument(...) equivalent.
    spacecraft,             # Spacecraft instance the geometry is attached to.
    name,                   # Name of the geometry.
    vertices,               # List of vertices defining the geometry shape.
    shape,                  # "CIRCLE", "RECTANGLE", "POLYGON", "HALF-SPACE", or "RAY".
    half_angle_in_rad=None  # Half-angle in radians (required for "CIRCLE" shape).
    body_location=[0,0,0],  # Location of the geometry in the spacecraft body-fixed frame.
)
```

Listing 10: *Geometry* (and the equivalent *CustomInstrument* subclass) input parameters.

To initialise a SPICE-based instrument geometry, the user must provide the NAIF name or code of the instrument as defined in a loaded IK file. The *SpiceInstrument* subclass will internally extract the necessary shape parameters from the IK file using SpiceyPy routines. An example of initializing a SPICE-based instrument geometry is shown in Listing 11.

```
spice_instrument = PAS3.SpiceInstrument(
    spacecraft,             # Spacecraft instance the geometry is attached to.
    naif_name_or_code,      # instrument NAIF name or code defined in a loaded IK file.
    body_location=[0,0,0]   # Location of the geometry in the spacecraft body-fixed frame.
)
```

Listing 11: Example of initializing a SPICE-based instrument geometry using the *SpiceInstrument* class.

Both methods include an optional input parameter for the geometry's position in the spacecraft's body-fixed frame. This allows visibility functions to use relative positions of target objects w.r.t. the geometry's apex. The default position is the origin of the spacecraft's body-fixed frame.

The last subclasses of the *Geometry* module are the *AssembledGeometry* and *AssembledInstrument* classes. These classes provide a way to combine multiple geometries into one; creating more complex spatial regions around the spacecraft. The boolean output of their predicate visibility methods depends on the combination of the individual geometries' visibility, as well as their underlying logical operator (AND, OR, NOT).

Aside from the visibility methods, every geometry class provides a function to extract the angle of incidence of a given incoming ray (expressed in the spacecraft's body-fixed frame) w.r.t. the geometry's boresight or normal vector. This is done with the method `angle_of_incidence_in_<. . .>(<vector>)`. Also, during runtime, the geometry's position and orientation w.r.t the spacecraft can be altered dynamically using the methods `move_to(<vector>)` and `rotate(<axis>, <angle>)`. All methods and properties of the *Geometry* classes are summarized in Table A.8 in Appendix A. For class attributes, the code documentation must be consulted.

4.5. Toolkit Verification

This section will describe the verification tests performed to ensure the correct implementation of the PAS3 functionalities. Many of these functionalities are SPICE-based, and as the SPICE toolkit is already a well-established toolkit with verified results, the verification activities mainly consist of checking if the SPICE integration is done correctly and if additional custom functionalities give expected results.

The verification campaign focused on 4 main aspects: setting up and testing a verification configuration, testing occultation events, testing ground station visibility, and testing geometry functionalities. The choice of focusing on ground stations and geometries is made as these are the main modules that offer either non-SPICE configurations of similar SPICE actors (custom ground stations vs SPICE-defined ground stations) or use visibility algorithms that are not SPICE-based (the geometries module). The success of the *Geometry* classes' methods depends heavily on the correct implementation of *Spacecraft* and *CelestialBody* class methods, and a failure in *Geometry* verification tests can indicate issues in both. Much emphasis is therefore put on verifying the geometry functionalities, as it consolidates the verification of many other PAS3 functionalities. This also explains the choice of verifying the occultation check implementations; visibility checks of geometries depend both heavily on spacecraft and celestial body states, and also on the correctness of general occultation of the visibility targets.

Section 4.5.1 describes the configuration of the two sample simulations used for executing the verification tests. This orbit configuration is also briefly checked if it conforms with the expected orbit parameters of the verification spacecraft. Section 4.5.2 describes the verification tests performed to check the correct implementation of SPICE-based occultation checks. In subsection 4.5.3, Spice-based and custom ground stations' mutual results and calculated visibility intervals are compared. The implementation of a custom visibility mask to the ground station objects is also verified. Finally, subsection 4.5.4 describes the extensive verification tests performed to compare the numerous *Geometry* subclasses with SPICE-based results, as well as mutual comparisons and calculations.

4.5.1. Verification PAS3 Setup

As described in section 4.1, PAS3 class blueprints are composed largely from the SPICE toolkit (SpiceyPy package) under the surface. In order to run verification tests of the PAS3 implementation, SPICE kernels must be generated for a predictable, stable sample orbit (and instruments) to compare the outputs of PAS3 with pure SPICE outputs or own calculations. Two simulation configurations were set up for this purpose: a Venus-centred orbit (in conformance with PAS3's intent of aiding interplanetary missions), and an Earth-centred orbit (for better ground station visibility verification). Their orbital parameters are shown in Table 4.3. Both orbits are circular, low altitude orbits with a rounded orbit radius for easier calculations. They were created using the `spkw15` SpiceyPy routine, generating a type 15 SPK file using a precessing conic propagation model. The orbits do not consider any perturbations (like the J2 effect) and are therefore stable over time. The orbits were generated spanning 10 days starting from the J2000 epoch (01/01/2000 11:58:55.8 UTC), commonly referred to as ET = 0 (zero seconds TDB past J2000).

Table 4.3: Verification spacecraft orbit parameters expressed in the J2000 (ICRF) reference frame at 0 TDB (01/01/2000 11:58:55.8 UTC).

		Venus Verification Orbit	**Earth Verification Orbit**
Position Vector	[km]	x: -6482.76; y: -417.63; z: 222.38	x: 1162.75; y: -6496.77; z: 0.0
Orbit Radius	[km]	6500.0	6600.0
Velocity Vector	[km/s]	v_x: 0.31; v_y: -6.43; v_z: -2.91	v_x: 7.65; v_y: 1.37; v_z: 0.0
Velocity Magnitude	[km/s]	7.07	7.77
Orbit Period	[hh:mm:ss]	01:36:17	01:28:56
Eccentricity	[-]	0.0	0.0
Inclination	[°]	24.43	0.0

The Venus verification orbit is coincident with Venus's orbital plane, which is inclined 3.39° w.r.t. the ecliptic plane and 24.43° w.r.t. the J2000 (ICRF) reference frame's XY plane. To construct the Venus orbit, use was made of the Venus-centric Solar Orbital frame (VSO) with its +X axis pointing from Venus towards the Sun

and +Y axis in the direction of the Sun's velocity w.r.t. Venus[2]. The verification spacecraft (henceforth referred to as *Verisat*) will be oriented coincident with the VSO frame at the initial start time for many *Geometry* tests in subsection 4.5.4 to ensure initial alignment of the spacecraft's body-fixed frame; pointing X-axis to the Sun (and nadir at ET=0), and pointing the Z-axis in the opposite direction of the orbit plane's normal vector. The Earth verification orbit will be used only in one test, namely the ground station visibility verification comparing visibility durations with hand calculations. The orbit is circular with a radius of 6600 km. The inclination shown in Table 4.3 is 0° w.r.t. the J2000 (ICRF) reference frame, coinciding with the Earth's equatorial plane.

After creation of the SPK files, Python assertion statements were used within the Venus and Earth orbit PAS3 configuration files to ensure the correct implementation of the orbit parameters in Table 4.3. The verification configurations are included in section B.3 and the SPK file creation scripts in section B.1 in Appendix B.

4.5.2. Occultation Verification

Checking the state of occultation for celestial bodies and spacecraft at specific epochs, is an important functionality which the PAS3 toolkit aims to streamline. Establishing a line of sight with other bodies is an essential input for several analysis use cases: target observation (central body visibility), thermal and power models (Sun visibility), and communication links (Earth, ground station, or additional spacecraft visibility). As mentioned in subsection 4.4.1, the *Simulation* class provides a way to configure the tolerance of occultation checks with the `set_occultation_tolerance(<tolerance>)` method, ensuring a stepwise and less procedural approach to occultation checks. SPICE distinguishes three types of occultation: total, partial and annular occultation. When the target body is the Sun, a total occultation is interpreted as the spacecraft being in umbra, a partial occultation as a penumbra state, and an annular occultation as the antumbra state. A graphical representation of the occultation types is shown in Figure 4.8.

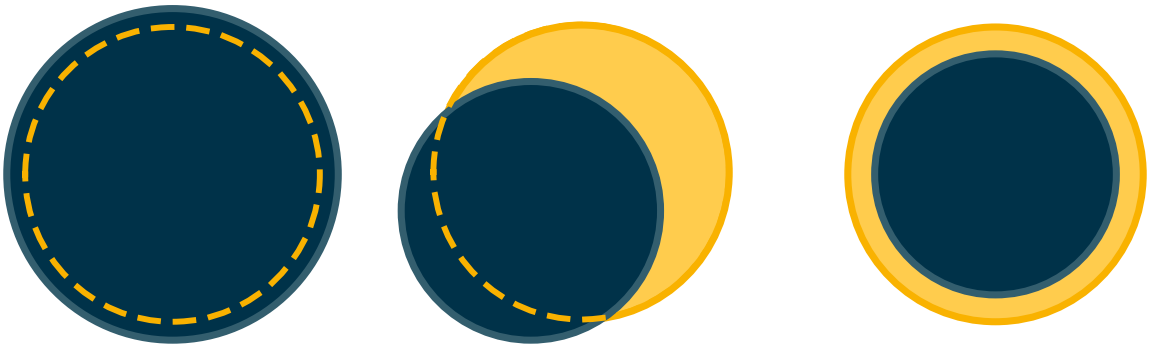


Figure 4.8: Occultation types. Left to right: total, partial, and annular occultation of the Sun (yellow) by another celestial body (dark blue).

The first check performed was to verify the correct implementation of the occultation checks in the *Spacecraft* class, comparing it to the `SpiceyPy.gfoclt()` routine. This function extracts occultation intervals between two bodies as seen from an observer over a set period of time. It takes the following arguments: the occultation type ("ANY", "FULL", "PARTIAL" or "ANNULAR"), the front body (name, shape and body fixed frame), the back body (name, shape and body fixed frame), the aberration correction flag, the observer name, the timestep for the search, and spice windows to store the output intervals and define the search period. Two PAS3 Verisat simulations were run, computing the spacecraft's eclipse state for 50 orbits. One with occultation tolerance set to 1 (visibility only if there is no occultation at all) and one with tolerance set to 3 (visibility allowing both partial and annular occultations). Calling `Verisat.is_sun_visible()` at every time step, a `False` flag for occultation tolerance 1 corresponds to any type of occultation ('ANY' setting for `gfoclt()`); a `False` flag for occultation tolerance 3 corresponds to a total occultation ('FULL' setting for `gfoclt()`).

The SPICE interval finder uses a convergence searching method to extract very precise intervals; the PAS3 comparison execution loop uses a fixed time step and extracts the intervals stepwise. This means that the extracted intervals must have a maximum difference of the time step size. The resulting start and end time differences are shown in Figure 4.9 for a timestep of 2 seconds. Note that the difference is always positive, as the PAS3 simulation results will always flag an interval start or end at a time step right after the actual event. Three assertions are set up in the verification test scripts: ensuring the difference in occultation intervals is not greater than the time step and always positive, ensuring the *full* occultation intervals always fall within the *any*-type occultation intervals, and ensuring the number of intervals found is the same for both methods.

It must be noted that the native SPICE `gfoclt()` routine is significantly quicker than the PAS3 stepwise

[2] Acquired from the RSSD0002.tf Generic Frame Definition Kernel File for ESA Planetary Missions: https://naif.jpl.nasa.gov/pub/naif/pds/data/mex-e_m-spice-6-v2.0/mexsp_2000/DATA/FK/ (last accessed on 16/09/2025)

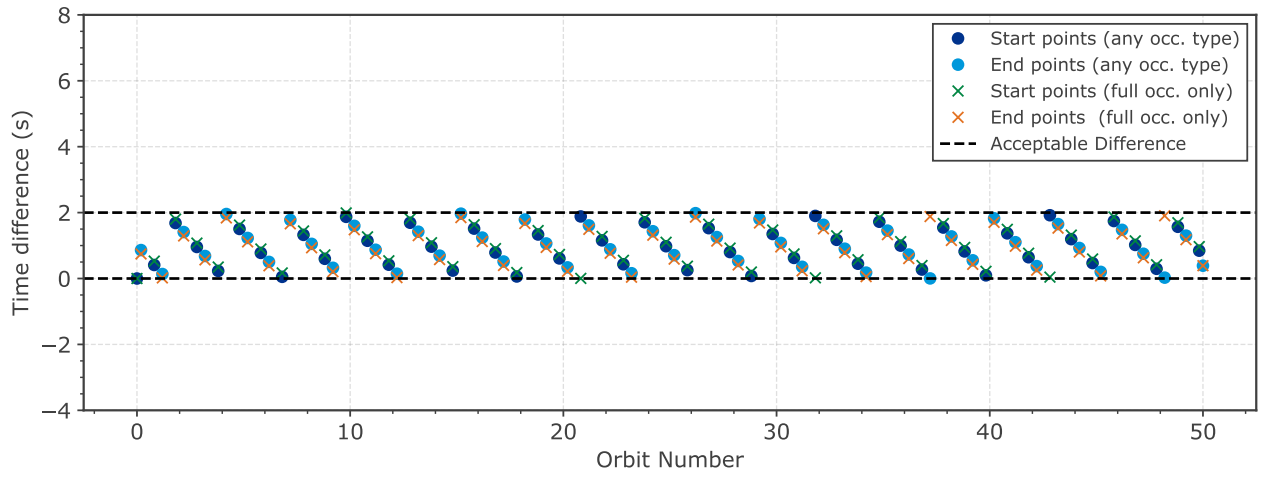


Figure 4.9: Interval time differences between PAS3 occultation checks (timestep = 2 sec) and SPICE `gfoclt()` results for 50 orbits of the Verisat spacecraft for full eclipses and any type of eclipse. Differences are computed as PAS3 time – SPICE time.

approach (about 15×), as it uses a C-based search algorithm. Within an PAS3 simulation loop, a lot of overhead is created by the multiple class method calls for states, orientations, and occultations. This highlights the difference in intended use cases: the SPICE routine is great for focussed analysis of specific events (or intervals), while PAS3 aims to provide a more accessible holistic simulation environment, at the cost of more overhead.

4.5.3. Ground Stations Verification

In this subsection, the *SpiceGroundStation* and *CustomGroundStation* classes from subsection 4.4.3 will be tested. There are three main reasons why the ground station functionalities get their own individual verification tests:

1. SPICE does not offer native ground station visibility checks. Ground stations are represented as fixed-offset or dynamic frames, potentially assigned to ephemeris state data (SPK). Implementing a ground station within a system simulation calls for expressing the target spacecraft object in the topocentric reference frame of the ground station and determining visibility with additional algorithms (occultations and visibility masks).
2. PAS3 offers the possibility to define a ground station without the need to create a specific SPICE FK. This method must be compared to SPICE ground stations to ensure correct implementation.
3. Both ground station classes have the possibility to get a custom visibility mask assigned, which must be verified as well.

Additionally, the verification tests give a nice way to also check the correct implementation of positions of celestial bodies and spacecraft in different reference frames (checking correct wrapping of the SPICE routines). Similar to the upcoming geometry verification tests, the success of these tests depends heavily on the correct implementation of *Spacecraft* and *CelestialBody* class methods, largely verifying their SPICE implementation by proxy.

Two PAS3 simulations were run for the ground station verification tests: the Venus verification orbit with two (real) SPICE-defined stations and their custom counterparts, and the Earth verification orbit with a fictional ground station. The Venus orbit will be used to compare the outputs of the SPICE-defined and custom ground stations, while the Earth orbit will be used to verify the correctness of the visibility mask implementation. In the Venus orbit configuration (see Listing 40 in Appendix B), The Cebreros and Svalbard[3] ground stations are initialized based on their SPICE frame and ephemeris data. Custom counterparts are initialized using their geodetic coordinates and altitude above the Earth ellipsoid.

The Venus orbit is simulated for one Earth-day (±15 orbits) with a time step of 1 second. Assertions are set up to ensure that both ground station types have the same Earth-fixed positions, the same topocentric rotation matrices, do not deem the spacecraft visible when the Earth is not visible, and give the exact same visibility intervals over the simulation period, regardless of being configured using SPICE frames or custom geodetic coordinates. Additionally, the method `get_object_topocentric_position(<PAS3 Object>)` is called at every step and the relative position of the Verisat spacecraft w.r.t. the ground stations are compared to ensure the

[3]Venus – Svalbard communication is not realistic as it is not part of the deep space network, it was only chosen for its very high latitude.

exact same results. The resulting visibility intervals of both ground stations (and their class types) are shown in Figure 4.10.

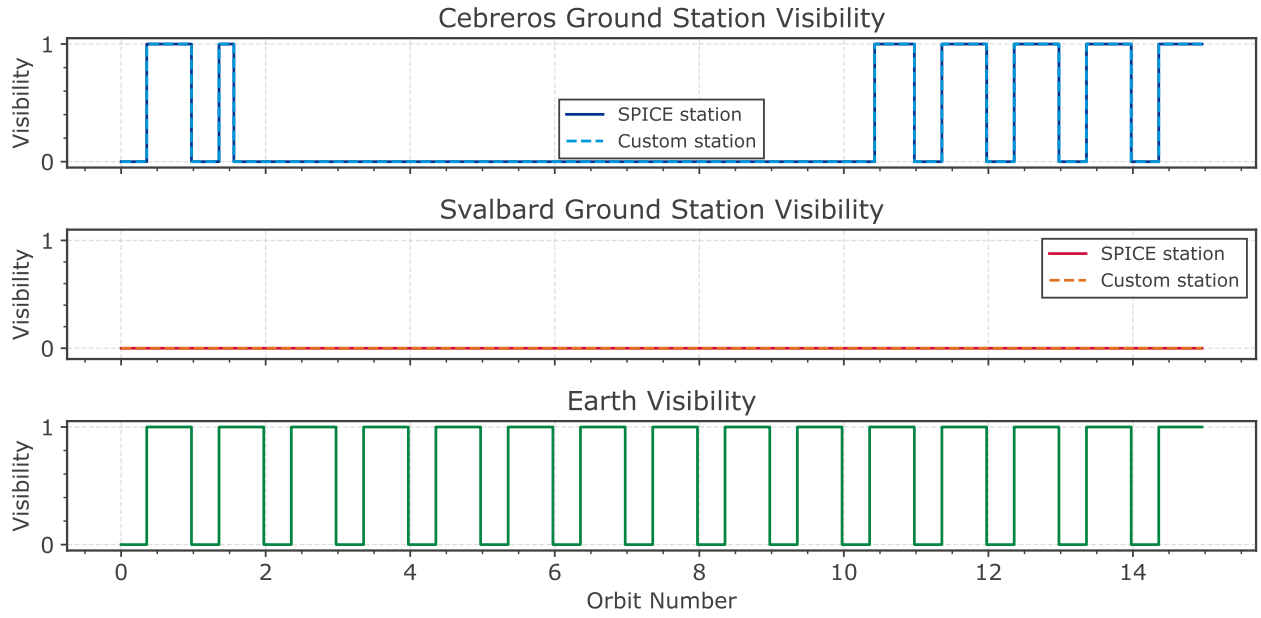


Figure 4.10: Venus verification orbit ground station visibility intervals for Cebreros and Svalbard over one Earth-day, including Earth occultations. Both SPICE-defined and custom ground stations give the same visibility intervals.

Note that the visibility masks for these ground stations were manually set to a minimum elevation angle of 0°. Due to the high latitude of Svalbard, the ground station is not able to see the spacecraft for a full Earth rotation. The Cebreros ground station, being closer to the equator, does have periodic visibility with the spacecraft. From the Cebreros and Earth Visibility plots, it can be seen that the visibility intervals correspond to the times when the Earth is not occulted by Venus. The relationship between ground station latitudes and the relative position of Venus is shown graphically in Figure 4.11; it is clear that Venus will always have a negative elevation angle w.r.t. the Svalbard station (red horizon plane) in one earth rotation (the duration of the verification simulation).

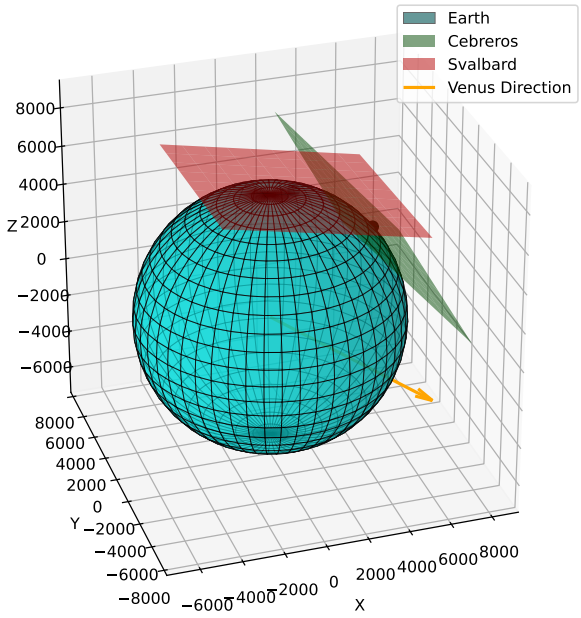


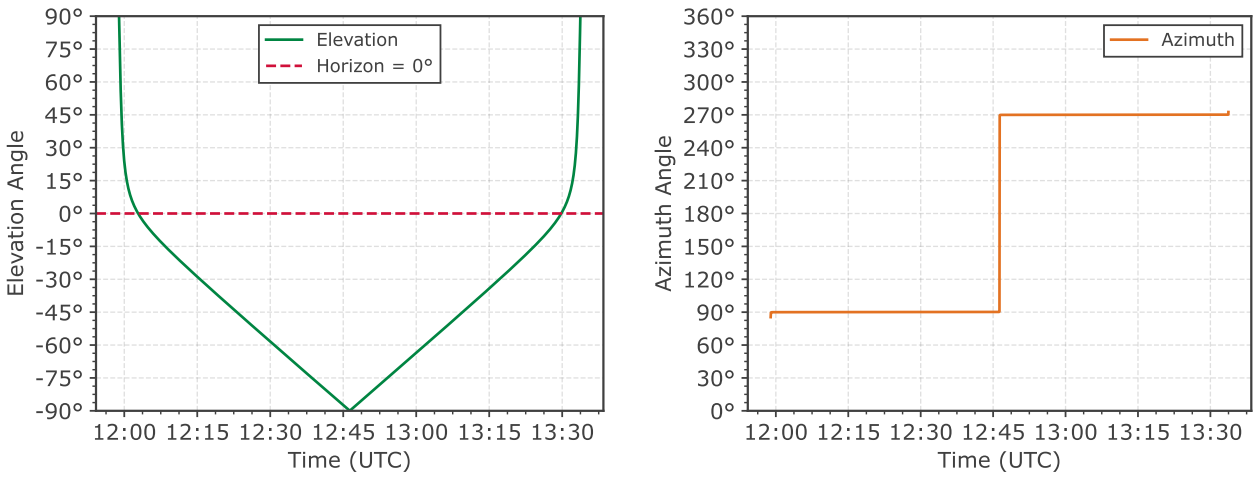
Figure 4.11: Verification ground stations positioned on Earth with their local horizons shown at ET = 0.

In order to verify the correct numerical results derived from ground station visibility masks, the Earth verification orbit is used. This makes comparison between simulated results and hand calculations easier. As mentioned in subsection 4.4.3, *GroundStation* objects start with a default visibility mask of a minimum elevation angle of 5°. The minimum elevation angle can be changed using the method `set_minimum_elevation_angle(<angle in degrees>)`. This results in conical regions, with the apex at the ground station location, where the spacecraft must be located to be visible. However, more complex visibility masks are also allowed, taking the spacecraft azimuth angle into account as well. The mask may be configured by the user and passed into the ground station object using the method `set_custom_visibility_mask(<callable(elevation, azimuth)>)`.

A fictional ground station is placed at the equator (0° latitude, 0° longitude) at sea level. The verification spacecraft is set on an equatorial orbit with a radius of 6600 km with a start position right above the station. The simulation is run for one orbit pass over the station, calculated as follows:

$$T_{sim} = \frac{2\pi}{\omega_{Orbit} - \omega_{Earth}} \approx 5688 \text{ seconds} \tag{4.1}$$

With ω_{Orbit} the orbital angular velocity of the spacecraft and ω_{Earth} the Earth's rotation rate. With this duration, the spacecraft should end up at the same relative location right above the fictional ground station at the end of the simulation. The simulation is run with a time step of 1 second. The Spacecraft position in the ground station's topocentric frame (i.e. elevation and azimuth) is presented in Figure 4.12.



(a) Elevation angle of the spacecraft w.r.t. the ground station's local horizon.

(b) Azimuth angle of the spacecraft w.r.t. the ground station's local North.

Figure 4.12: Topocentric position of the spacecraft w.r.t. the fictional ground station over one orbit pass.

From Figure 4.12a, it can be seen that the spacecraft is only briefly visible above the local horizon (elevation angle > 0°) of the ground station. The pass was repeated for a minimum elevation angle of 0°, 15°, 30°, 45°, 60°, 75° and 89° (close to 90°), and the resulting pass durations stored. Analytically, due to the geometrical properties of the circular, equatorial orbit, the expected percentage of the orbit period that the spacecraft is visible above a certain minimum elevation angle ϵ_{min} can be calculated as follows:

$$Orbit_{\%} = \frac{1}{2} - \frac{\epsilon_{min} + arcsin\left(\frac{R}{r \cdot cos(\epsilon_{min})}\right)}{\pi} \tag{4.2}$$

With R the radius of the Earth, and r the orbit radius of the spacecraft. This equation holds for default (conical) visibility masks. To test the implementation of custom visibility masks, a mask was created that only allows visibility for azimuth angles between 0° and 180° (i.e. the spacecraft must be on the Eastern side of the station). This was implemented as follows:

```python
def azimuth_mask(azimuth, elevation):
    """A Visibility mask that only returns true when the Spacecraft is East of the station
    ↪ and above a minimum elevation angle.
    """
    nonlocal min_elevation_angle # defined in outer scope

    if elevation < np.radians(min_elevation_angle) or azimuth > np.pi:
        return False
    return True

EquatorialGroundStation.set_visibility_mask(azimuth_mask)
```

Listing 12: Custom visibility mask allowing visibility only for azimuth angles between 0° and 180° and above a specified minimum elevation angle.

The expected duration of the custom visibility mask should be half of the default mask's duration, as the space

of allowed azimuth angles is halved. The results of the visibility duration tests are shown in Figure 4.13 and conform with the expected results.

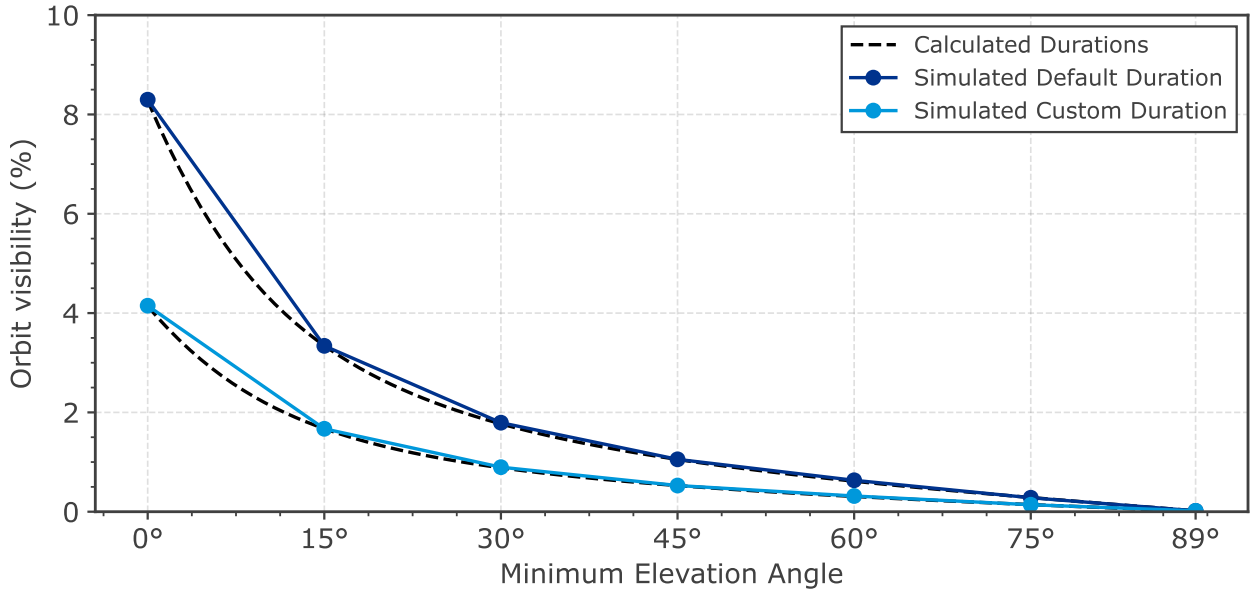
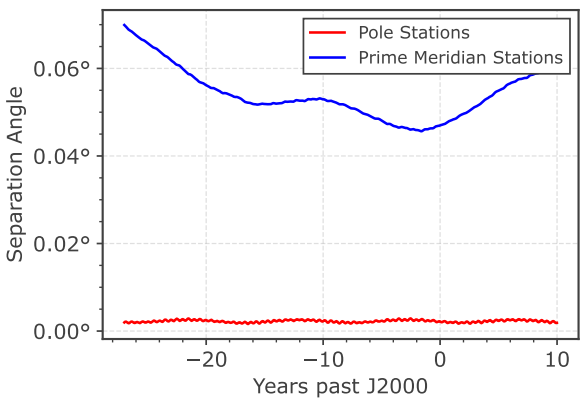
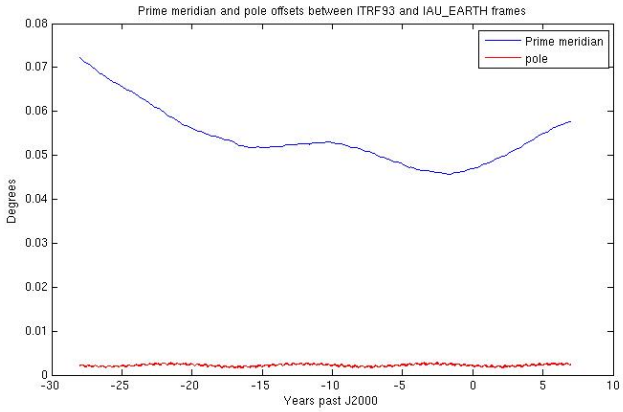


Figure 4.13: Percentage of the orbit pass that the spacecraft is visible above the ground station's local horizon for different minimum elevation angles (and a custom visibility mask).

An additional check performed for *GroundStation* class is the correct implementation of the Earth-fixed reference frames the ground stations are defined in. As mentioned in subsection 4.4.3, an earth-fixed reference frame must be specified as input parameter at initialization. This allows for very precise positioning of the ground station w.r.t. an arbitrarily complex representation of the Earth orientation. To check if configuring the ground station w.r.t. the high-accuracy "ITRF93" or low-accuracy "IAU_EARTH" has any effect, ground stations were initialized on the Earth North Pole and the intersection between the equator and prime meridian in both frames. A long simulation (50 years) was run and the evolution of angular separation between both types of initializations was compared with the "*IAU_EARTH vs ITRF93 Comparison Plot*" from the NAIF SPICE documentation [41]. If the angular separation between the two ground stations conforms with the given NAIF plot, the earth-fixed frames are correctly implemented. The results of the angular separation test are shown in Figure 4.14, conforming with the NAIF documentation.



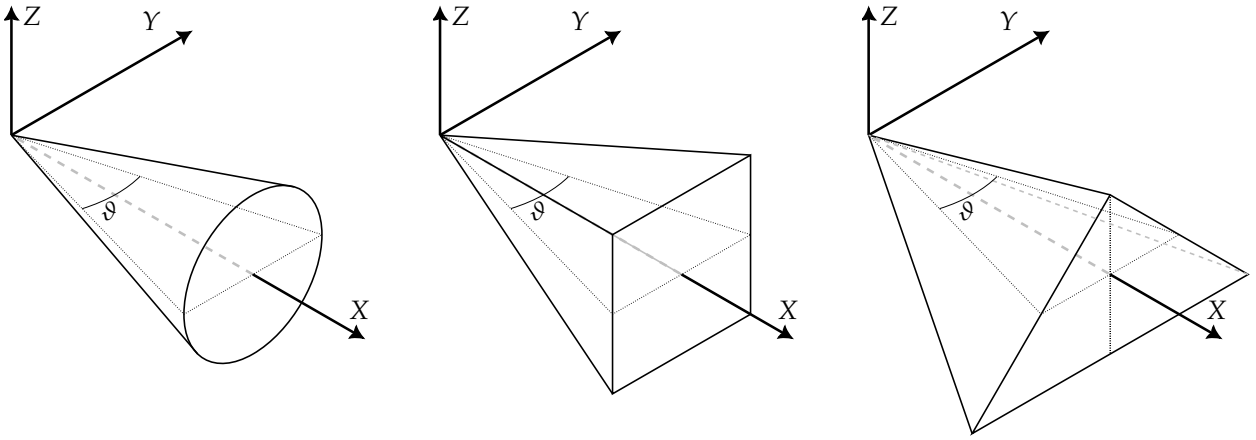
(a) PAS3 ground stations separation.



(b) Reference plot from NAIF SPICE documentation [41].

Figure 4.14: Comparison between high- and low-accuracy Earth-fixed frame definitions, showing deviations in.

This concludes the *GroundStation* class verification tests. It was verified that the SPICE-defined and custom ground stations give the exact same results, as long as the SPICE-defined ground stations are fixed w.r.t an Earth-fixed frame. The correct offset with such (dynamic) Earth-fixed frames was also verified, as well as expected interplanetary and Earth-bound visibility intervals.



(a) Circular FOV (CircFOV). **(b)** Rectangular FOV (RectFOV). **(c)** Triangular FOV (TriFOV).

Figure 4.15: Verification geometries defined in SPICE IK files with shapes: (a) Circular FOV, (b) Rectangular FOV, and (c) Triangular FOV. All with angle $\vartheta = 45°$.

4.5.4. Geometries Verification

A large part of the PAS3 toolkit's verification campaign is the verification of the *Geometry* classes. This is due to the choice of fully omitting the use of built-in SPICE visibility functions for instrument FOV's. This choice was made for the following reasons:

1. SPICE visibility functions require a SPICE-based attitude profile of the spacecraft. PAS3 does not require a SPICE-based attitude profile (see the solution space in section 4.1), and therefore the SPICE visibility functions are not applicable.
2. The *Geometry* classes offer a more accessible way to define instrument FOV's, without the need to create a specific SPICE IK file. This allows for quick prototyping and testing of different instrument configurations. SPICE instrument visibility routines always require IK files.
3. The *Geometry* classes offer non-SPICE native infinite geometries (half-spaces and rays), which are not possible to implement using IK files[4].

Therefore, unlike verified SPICE-based functionalities, the *Geometry* methods must be verified more extensively. Similarly to *GroundStation* class verification cases, first results from the geometry / instrument classes initialized using SPICE IK data (*SpiceInstrument*) are compared to the custom geometry / instrument classes (*CustomInstrument* / *Geometry*). Their results must be identical, as they represent the same geometrical shape and orientation. Secondly, these results are then compared to pure SPICE visibility routines (tied to a SPICE-based spacecraft attitude profile). Lastly, the non-SPICE geometries (half-space and ray) are verified using hand calculations, as well as rotation and movement methods of the geometries.

For all verification cases, the Venus verification orbit from Table 4.3 is used. Within the verification simulation configuration file, three instruments are initialized: a rectangular FOV (`RectFOV`), a circular FOV (`CircFOV`), and a polygonal FOV (initialized as a triangular FOV[5], `TriFOV`). All instruments have their boresight aligned with the spacecraft's +X axis and angle of view $\vartheta = 45°$ in the XY plane. They are presented graphically in Figure 4.15. In Listing 39, the SPICE IK used to initialization of the *SpiceInstrument* objects is shown. The same geometries are initialized using the *CustomInstrument* and *Geometry* classes as well in Listing 40.

Three verification orbits were run, each executing a different visibility check. The first orbit runs a visibility check of the Sun, the second orbit runs a visibility check of Venus (central body), and the third orbit runs a visibility check of the nadir vector. The first two check if a celestial body is within the instrument FOV, modelling the target as a circular cone encapsulating the (spherical) body. The nadir check verifies if the nadir vector (a direction vector) is within the instrument FOV. During these three test orbits, all geometry / instrument objects call their visibility methods and compare their results. The same shapes (different object types) should always have the same result. Moreover, due to the symmetry of the orbit, it is expected that all geometries will have visibility of the Sun and nadir at the exact same moments in time, while for the Venus visibility check, the triangular FOV should have longer visibility intervals due to the larger base edges (see Figure 4.15c). This is reflected in the visibility intervals shown in Figure 4.16.

[4]SPICE planes and rays do exist and can be used for intersection checks, but these are not inherently fixed to a spacecraft body. DSK surfaces are also mostly used for tessellated planetary surfaces, not simple infinite geometries.

[5]A triangular FOV is not a very realistic FOV shape, it is purely set up to ensure compatibility with "POLYGON" SPICE FOV shapes.

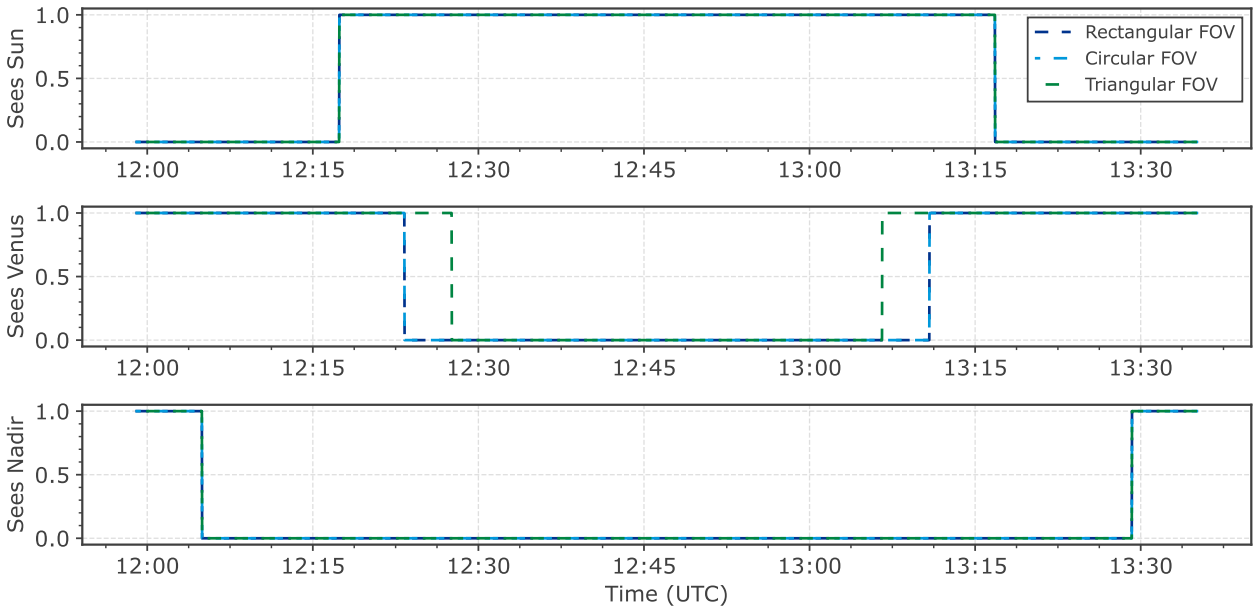


Figure 4.16: Visibility intervals of all geometries (*Geometry*, *CustomInstrument*, and *SpiceInstrument*) for the Sun, Venus, and nadir checks over one Venus verification orbit.

After the same results between all geometry / instrument types were verified, the numerical results can be compared to SPICE FOV visibility routines. The functions `SpiceyPy.fovtrg()` and `SpiceyPy.fovray()` are used to check if a target body or vector is within a specified instrument FOV respectively. As mentioned above, for these routines to work, The *Verisat* spacecraft must have a SPICE-based attitude description. For simplicity, it was decided to map the spacecraft's body-fixed frame to the VSO frame at every time step, ensuring the +X axis always points to the Sun. `SpiceyPy.fovtrg()` was called to check the Sun and Venus visibility, making sure to call `SpiceyPy.occult()` for the Sun visibility checks to ensure occulted states by Venus are filtered out[6]. `SpiceyPy.fovray()` was called to check the nadir visibility, passing the nadir vector expressed in the VSO frame. The results of these SPICE visibility checks were then compared to the PAS3 geometry visibility results (with SPICE attitude enabled using `Verisat.use_spice_attitude(True)`), ensuring the exact same results.

The non-SPICE geometries (half-space and ray) cannot be compared to SPICE visibility routines, as these shapes are not natively supported by SPICE. Therefore, their verification must be done using hand calculations. To verify the half-space visibility, the geometry is initialized with its normal vector pointing in the +X direction of the spacecraft body frame. The orbit uses inertial pointing (no dynamic sun-pointing VSO orientation), starting off with the spacecraft's +X axis pointing to the Sun direction. The plane must always see the sun when the spacecraft is not in eclipse.

To verify the half-space visibility function correctly models the sun as a sphere, the spacecraft is rotated 90° around the +Z (out of the page) axis, pointing the plane's normal vector orthogonally to the Sun direction. Due to the spherical Sun model, the plane should still see the Sun for the full orbit when not eclipsed. Lastly, the spacecraft is rotated just far enough for the edge of the infinite plane of the half-space to be tangent to the Sun sphere, which must result in no visibility for the full orbit. The orientations are shown in Figure 4.17.

For the ray geometry, the ray is initialized with its direction vector pointing in the +X direction of the spacecraft body frame. The orbit again uses inertial pointing, starting off with the spacecraft's +X axis pointing to the Sun direction. The assertion

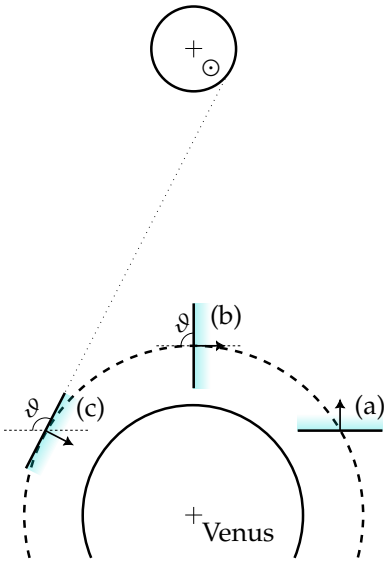
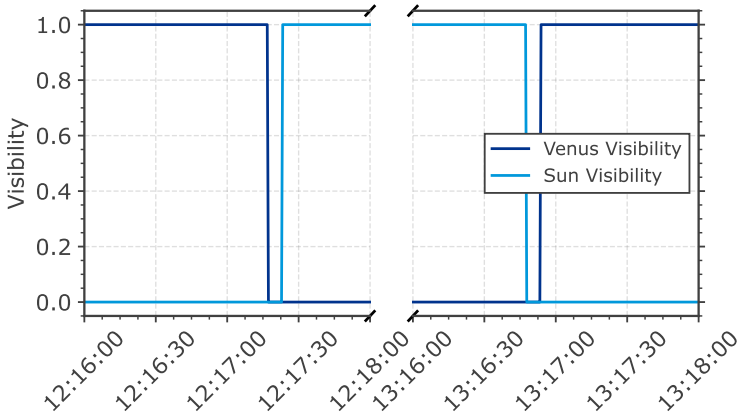


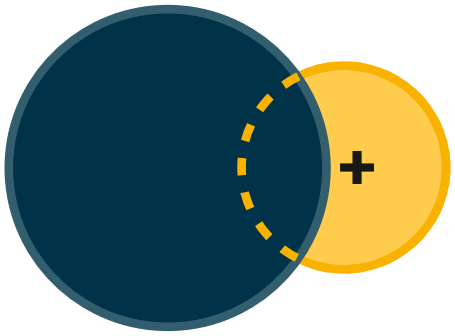
Figure 4.17: Half-space verification setup in three orientations: (a) normal vector in Sun direction, (b) normal vector orthogonal to Sun direction, and (c) edge case of no visibility for a full orbit.

[6]SPICE visibility does not by default consider occultations; they are purely geometrical by default.

is set up to ensure when the ray intersects with the sun, it does not intersect with Venus (and vice versa). Simultaneous visibility would mean the ray intersects both spheres, and for the visibility function, only the first intersection is considered. One special case is observed when the spacecraft undergoes partial occultation (i.e. located in penumbra). In this case, the ray may intersect with the sun only, but returns no visibility due to the configured occultation tolerance of the simulation. When PAS3 determines the target body is occulted from the point of view of the geometry's host spacecraft, the geometry visibility function will always return no visibility, regardless of the ray intersection results. This behaviour must be patched in upcoming work. The result is shown in Figure 4.18.



(a) Occultation tolerances causing no visibility during partial occultation (penumbra).



(b) Ray intersection of the Sun during partial occultation.

Figure 4.18: Short cases of no visibility during partial occultation (penumbra) in the verification ray geometry simulation due to occultation tolerances of the sim.

Now that all geometry types have been verified, additional tests were performed to verify the rotation and movement methods of the geometries (i.e. `rotate(<axis>, <angle in radians>)` and `move_to(<new position>)`). These methods are intended to allow the user to change the geometry's orientation and position during a simulation without changing that of the spacecraft.

The rotation method was tested by rotating the rectangular and circular geometries one full rotation ($d\vartheta = 0.1°$) around the +Z axis of the spacecraft body frame without advancing the simulation time (all actors remain stationary). Due to their symmetry in the XY plane, the circular and rectangular geometries should have the same visibility intervals of Venus. The cutoff angles for visibility conditions were hand calculated using the following equation:

$$\begin{cases} \phi_{los} & = \frac{\vartheta_{FOV}}{2} + \arcsin\left(\frac{R}{r}\right) \approx 91.1° \\ \phi_{aos} & = 360° - \phi_{los} \approx 268.9° \end{cases} \tag{4.3}$$

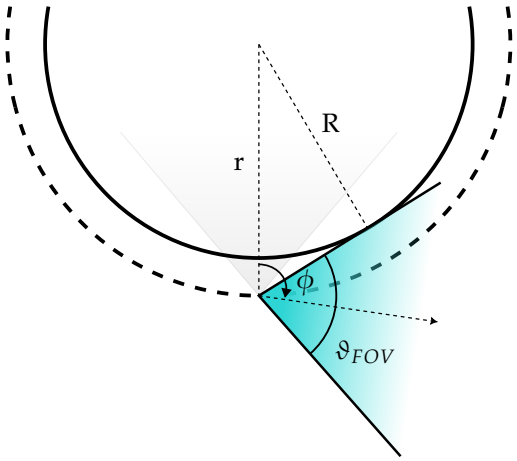


Figure 4.19: Rotation verification test setup, rotating (angle ϕ) the geometries around the +Z axis of the spacecraft body frame.

With ϕ_{los} the angle where loss of sight occurs, ϕ_{aos} the angle where acquisition of sight occurs, ϑ_{FOV} the geometry's angle of view (see Figure 4.15), R the radius of Venus, and r the orbit radius of the spacecraft. These symbols are also shown in Figure 4.19. The resulting visibility intervals are shown in Figure 4.20, conforming with the analytical solution from Equation 4.3.

For the movement verification, a slightly different approach was considered. Due to the vast scales and distances of celestial bodies, moving a geometry by a few metres has very little effect on most visibility results. The case in which this can make an observable difference is when the geometry is very narrow and is looking at the edge case of visibility (for example the edge of a planet).

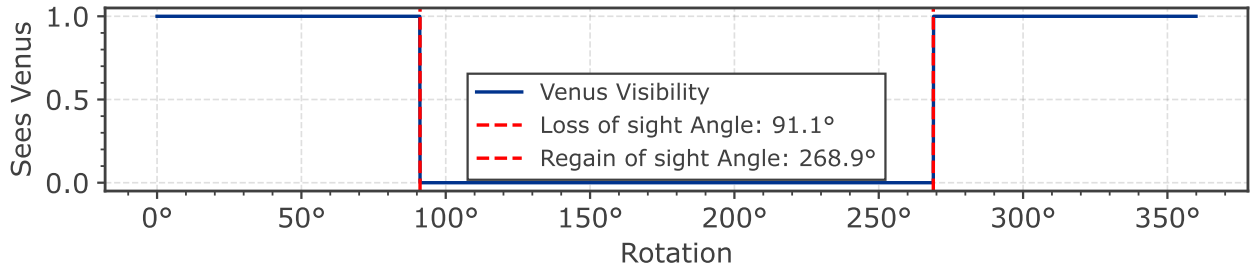


Figure 4.20: Visibility interval of the circular and rectangular geometries during a full 360° rotation around the +Z axis of the spacecraft body frame without advancing simulation time. The visibility cutoffs conform with analytical solutions.

A very narrow rectangular geometry ($\vartheta_{FOV} = 0.01°$) was initialized in the same fashion as the other geometries. The spacecraft body was then rotated by 68.6° around the +Z axis of the spacecraft body frame, pointing the narrow geometry's boresight right at the edge of Venus's limb. Then, the geometry was moved by 1 metre increments along the +Y axis until the geometry no longer intersected Venus. As with the rotation checks, the simulation time is not advanced either. The expected offset distance to achieve loss of sight was hand calculated as follows:

$$\Delta Y = sin\left(\frac{\vartheta_{FOV}}{2}\right)\sqrt{r^2 - R^2} \approx 414 \text{ m} \qquad (4.4)$$

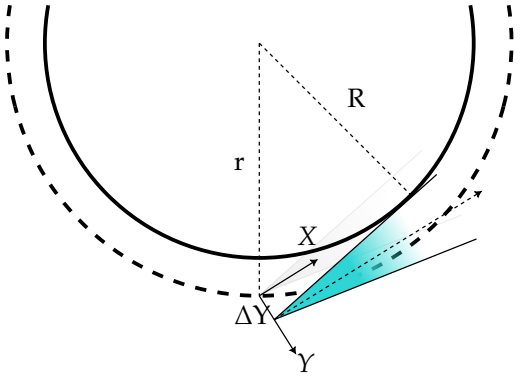


Figure 4.21: Movement verification test setup, moving the geometry along the +Y axis of the spacecraft body-fixed frame.

With ΔY the offset distance along the +Y axis of the spacecraft body frame, ϑ_{FOV} the geometry's angle of view, R the radius of Venus, and r the orbit radius of the spacecraft. These symbols are also shown in Figure 4.21. The resulting visibility vs. offset relationship is shown in Figure 4.22, conforming with the analytical solution from Equation 4.4.

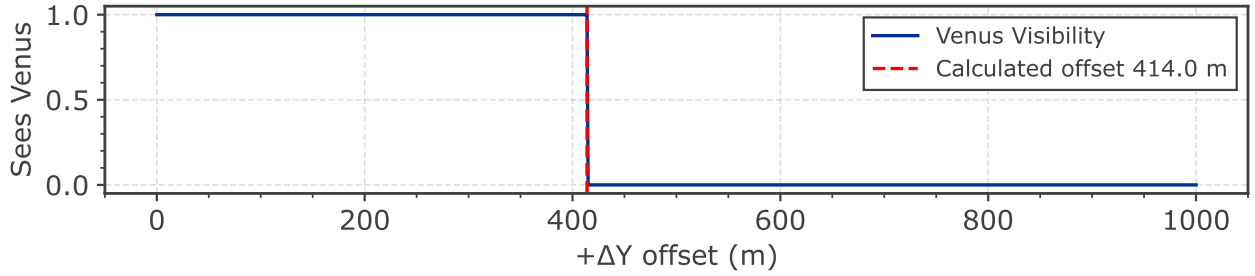


Figure 4.22: Visibility of a narrow rectangular geometry ($\vartheta_{FOV} = 0.01°$) when offset along the +Y axis of the spacecraft body frame while pointing at the edge of Venus's limb. The visibility cutoff conforms with the analytical solution.

This concludes the *Geometry* class verification tests. It was verified that the custom geometry / instrument classes give the exact same results as their SPICE IK counterparts, as well as pure SPICE visibility routines when a SPICE-based attitude profile is used. The non-SPICE geometries (half-space and ray) were verified using hand calculations, as well as the rotation and movement methods of the geometries. Similar to the *GroundStation* class verification tests, the working of the visibility methods rest on the correct implementation of *Spacecraft* and *CelestialBody* class methods. Incorrect results in *Spacecraft* attitude methods like `use_spice_attitude()`, default frame relationships and transformations and descriptions of positions expressed in the body-fixed frame (`get_<...>_in_body_fixed_frame()`) would lead to incorrect geometry visibility results. Most of the previous tests rely heavily on the correct workings of relative limb calculations of celestial bodies (Venus and Sun), which are implemented in the *CelestialBody* class, as well as automatic allocations of frame, shape and name attributes for encapsulated SPICE routines. Therefore, these verification tests also serve as an indirect verification of many *Spacecraft* and *CelestialBody* class methods and their correct SPICE implementation.

5

EnVision System Simulator Implementation

The goal of the proposed toolkit package is to create a mission-specific system simulator for space missions. As discussed in chapter 2, the toolkit development was started in the context of the EnVision mission's project team at the European Space Research and Technology Centre (ESTEC). While the toolkit is designed to be generic and applicable to a wide range of planetary missions, this chapter will focus on its implementation in creating custom models to be used on a system simulator for EnVision. The goal of this chapter, however, will not be to create a fully-fledged system simulator including models for all subsystems. Rather, the focus will be on demonstrating the toolkit's capabilities by developing a set of subsystem models, depicting their development process and their mutual interactions. This chapter should be viewed as a proof of concept, showcasing how the toolkit can be used to create system models for a specific mission, aiding requirement generation, sizing analysis and verification activities.

This chapter will first establish a solid foundation of the EnVision mission. In section 5.1, the mission's context is introduced, including its history, scientific goals, and a description of the spacecraft platform & payloads. Afterwards, the science operations and pointing profile is discussed in section 5.3, linking EnVision's scientific goals, instruments and platform constraints to a representative model of the spacecraft's attitude. Section 5.4 and section 5.5 will describe the process of building models for thermal and power analysis for the EnVision spacecraft.

Before diving into this chapter, it is essential to understand that this part of the thesis aims to show the process of model development using the PAS3 toolkit. An early distinction must be made, as the *results* of this thesis are not the quantitative results of the models, rather the models themselves are. The presented analysis results are demonstrative of PAS3's capability of providing a practical framework for supporting the necessary system engineering analyses within a real mission development context.

5.1. EnVision

EnVision is the latest proposed medium-class mission by the European Space Agency (ESA) in its Cosmic Vision 2015–2025 programme. Scheduled for launch in 2031, the mission aims to characterize Venus's interior, (sub)surface and atmosphere, studying their interactions and providing a holistic understanding of the planet.

The contents of the thesis were conducted within the EnVision project team at the European Space Research and Technology Centre (ESTEC). The presented research aims to support the systems engineering activities of ESA planetary science missions. Results will be assessed by implementation within the EnVision context. This section aims to establish a solid foundation of the mission at hand. In subsection 5.1.1, EnVision's place within ESA's long-term planning cycles and its history, from proposal to adoption is presented, as well as why the mission was proposed by the scientific community in the first place. Section 5.1.2 gives a short overview of the spacecraft platform, its design choices and scientific instruments/experiments. This description of the spacecraft platform and instruments will be used throughout to contextualize the models developed in this chapter.

5.1.1. Mission History & Scientific Goals

By the time of writing, the ESA science directorate is within its third long-term scientific planning cycle: Cosmic Vision 2015-2025. This programme builds on the success of its predecessors (Horizon 2000 and Horizon 2000 plus) and, through consultation of the broad scientific community, aims to address the following fundamental questions [47]:

- What are the conditions for planet formation and the emergence of life?

- How does the Solar System work?
- What are the fundamental physical laws of the Universe?
- How did the Universe originate, and what is it made of?

Following this overarching scientific framework, proposals for new missions are evaluated based on their scientific merit, feasibility, and alignment with the programme's goals. The Cosmic Vision programme is structured around four mission classes: large (L-class), medium (M-class), small (S-class) and fast (F-class).

L-class missions are European-led complex flagship missions with significant budgets, demanding novel technology developments and long development timescales. They are planned to have a launch frequency of one every six to seven years. Three have been selected: L1 - JUICE (launched in 2023), L2 - Athena (recently rescoped to NewAthena [48]) and L3 - LISA [49].

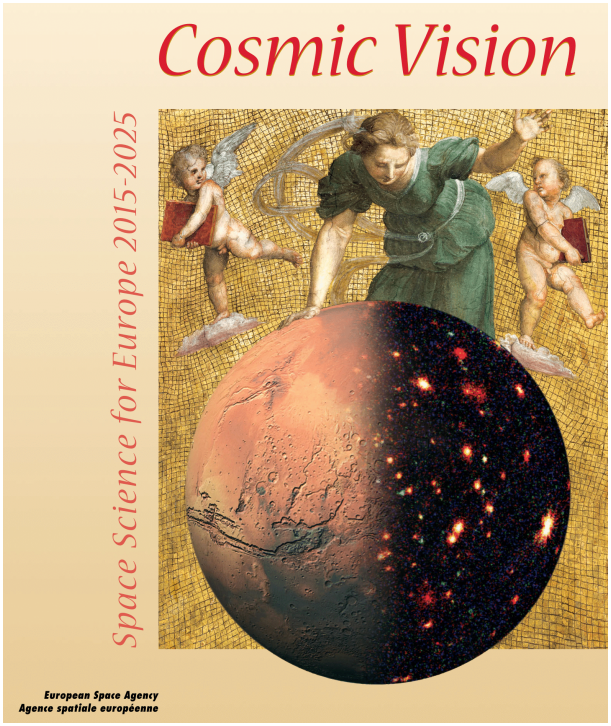


Figure 5.1: Cosmic Vision 2015-2025 programme brochure cover [47]

S- and F-class missions are smaller missions with lower budgets, shorter development times and relying on fully demonstrated technologies. S-class missions – currently: S1 - CHEOPS (launched 2019) and S2 - SMILE [50] – focus more on allowing member states to take a leading role, with limited financial ESA-contribution. F-class missions rely on rapid development of innovative implementations. They are ESA-led missions with a total development duration from selection to launch readiness of less than 10 years (F1 - Comet Interceptor and F2 - ARRAKHIS are two F-class missions under development) [49, 51].

EnVision is the 5th M-class mission in the Cosmic Vision programme (M5)[1]. Medium-class missions allow flexibility to the programme in responding to evolving challenges in scientific research. They are aimed to launch every two to three years, with typically two launches in between large-size mission launches, [51]. The main distinction between the L- and M-class missions is the design-to-cost strategy adopted for the M-class missions, which limits the ESA Cost at Completion (CaC) to 550 M€ [53] (compared to L-class 1.05 B€ CaC [54]). The design-to-cost influence on the mission and spacecraft platform will be discussed in later sections.

Within the scope of Cosmic Vision 2015-2025, EnVision will address the question:

"What are the conditions for planet formation and the emergence of life?"

By studying Venus's interior, surface, and atmosphere, the mission seeks to understand the planet's geological and climatic evolution. Providing insights into why Venus and Earth, despite their similarities, have followed such different evolutionary paths. This investigation will contribute to a better understanding of planetary habitability and the processes that shape terrestrial planets [18].

From 2006 to 2014, ESA's Venus Express mission marked a significant milestone in Venus exploration, establishing European leadership in the field. While primarily focused on atmospheric research, Venus Express uncovered evidence of geological activity on Venus [55]. However, many questions remain unanswered, particularly regarding the divergence of Venus's evolutionary path from that of the Earth. Venus shares a similar size, bulk composition, and proximity to the Sun, and may have possessed water-rich environments early in its history. Yet, while Earth evolved into a habitable world, Venus underwent a dramatic transformation – possibly due to a runaway greenhouse effect – that led to its current inhospitable state [18].

Throughout history there have been a total of 15 successful missions with Venus as destination. Of which, 8 were orbiters [56]. Arguably one of the most significant of these orbiter missions was NASA's Magellan. The

[1] The other 4 medium-size adopted missions are: M1 - Solar Orbiter (launched 2020), M2 - Euclid (launched 2023), M3 - PLATO (launch planned in 2026), M4 - Ariel (launch planned in 2029) [52]

thick Venusian atmosphere prohibits conventional optical methods of mapping its surface; therefore, we must resort to indirect methods like microwave remote sensing [57]. Magellan's synthetic aperture radar (SAR) was able to map between 89%–98%[2] of the surface of Venus from 1990 to 1994. This mission provided a wealth of data on Venus's surface geology, revealing a planet dominated by volcanic and tectonic processes [58, 59].

Magellan provided a static survey of Venus's surface, but recent advancements in radar technology revived interest in dynamic surface studies, leading to the proposals of EnVision. After an initial M3 proposal in 2010 [60], a second M4 proposal submission in 2015 [61], and a successful M5 proposal in 2016 [62], the mission passed concurrent design campaign in 2018 [63], and the mission was selected in 2021 and officially adopted in 2024 [64].

EnVision's main goal is to perform a radar geophysics study, in synergy with atmospheric, subsurface and interior investigations. EnVision will build on the legacy of missions like Venus Express, Magellan, Mars Express, Cassini, and Sentinel-1, aiming to revolutionize our understanding of Venus's geology, atmosphere, and interior [55]. The scientific objectives are categorized in 3 main themes [18]:

History – How have the surface and interior of Venus evolved?

- Understanding Venus's magmatic history
- Understanding Venus's tectonic history
- Assessing Venus's surface modification processes
- Understanding how Venus's interior and surface have evolved

Activity – How geologically active is Venus?

- Understanding Venus's volcanic activity in the present era
- Assessing Venus's aeolian activity and mass wasting

Climate – How are Venus's atmosphere and climate shaped by geological processes?

- Understanding the role of geological activity in Venus's climate evolution
- Assessing temporal variations of the Venus atmosphere

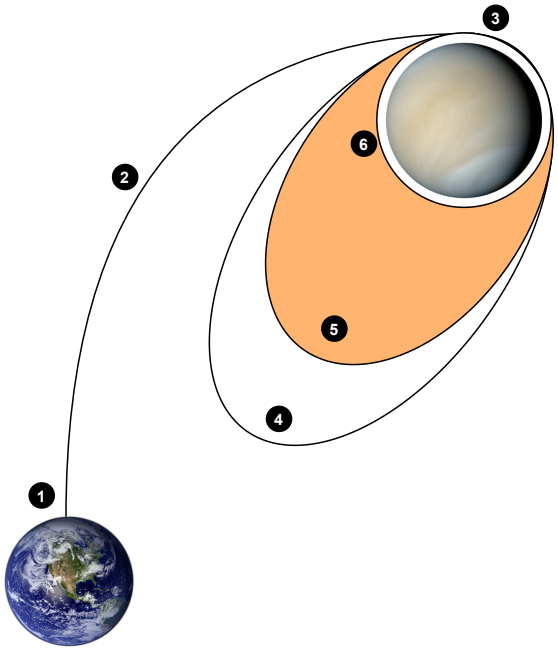


Figure 5.2: Planned mission phases of EnVision:
1) Launch, 2) Interplanetary cruise, 3) Venus orbit insertion,
4) Apoapsis lowering, 5) 11 months aerobraking, 6) Science orbit.

EnVision shall achieve its mission objectives within 7.4 years after launch with an Ariane 64 in a direct escape strategy planned November 2031. The interplanetary cruise phase will last up to 18 months ending with a Venus orbit insertion manoeuvre. After an apoapsis lowering manoeuvre, the 11-month aerobraking phase in Venus's atmosphere starts. The nominal scientific phase is planned to start in 2034 and observations will last 6 Venus-cycles[3] [65]. The planned mission phases are illustrated in Figure 5.2.

During the science phase, EnVision will conduct subsurface sounding, surface SAR (stereo) imaging, altimetry, radiometry, polarimetry, spectroscopy, radio-occultation experiments (measuring temperature and pressure structure of the atmosphere) and gravity experiments (mapping of the Venus gravity field) [18]. The instruments and general spacecraft platform will be discussed in more detail in the following subsection.

5.1.2. Spacecraft Platform & Instruments

Throughout this chapter, references will be made to architecture and instruments of the EnVision spacecraft system. It is therefore beneficial to establish an understanding of the spacecraft platform and payload and to contextualize their use reaching the mission objectives. Note that most of the information in this subsection is

[2] Coverage percentage depends on type of SAR observation.

[3] A Venus cycle is defined as 243.02 Earth days, or 1 Venus sidereal day, the time needed for Venus to spin 360 degrees on its axis in a celestial reference frame, in retrograde motion. This Venus "sidereal day" lasts longer than the revolution of Venus around the Sun (or Venus "year", 224.70 days) [18]

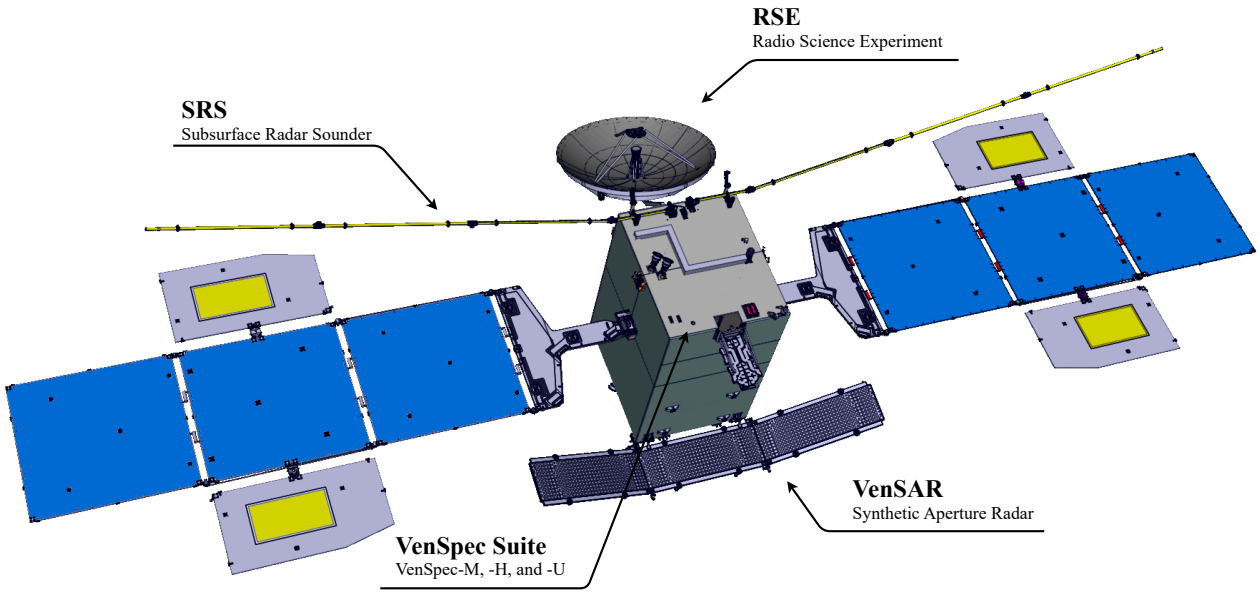


Figure 5.3: EnVision CAD model rendering of the latest design iteration (model produced by Thales Alenia Space Italia)[1]

based on the publicly available EnVision definition study report [18], but some details may differ from this report based on newer design iterations.

EnVision will be a three-axis stabilized orbiter, roughly 2.5 x 4 x 4 meters in its stowed configuration with a dry mass of approximately 1,700 kg (margins included). The spacecraft is equipped with a 2.5-metre deployable high-gain antenna using X- & Ka-band communication for downlinking a total of 189 Tbits of science data over the mission lifetime. The platform will be powered by two deployable solar arrays, delivering up to 2.8 kW of power (margins included). The system includes chemical propulsion, designed for orbit transfer, and reaction wheel offloading.

EnVision holds 6 scientific payloads (5 instruments and one experiment). Three spectrometers, one subsurface radar sounder (SRS), a synthetic aperture radar (SAR), and a radio science experiment (RSE). An illustration of the EnVision spacecraft platform and its payloads is shown in Figure 5.3. Each payload instrument, and its science target, will be discussed briefly on the next pages.

The **VenSpec Suite** holds three spectrometers, each targeting different spectral ranges and scientific objectives:

- **VenSpec-M** (multispectral NIR): pushbroom imaging spectrometer designed to measure near-global compositional data on rock types, weathering, and crustal evolution. It operates within 14 spectral bands at the nightside of Venus.
- **VenSpec-H** (high-resolution IR): an atmospheric spectrometer with 4 spectral bands. Its main objective is to quantify SO_2, H_2O and HDO in the atmosphere, below (nightside) and above (dayside) the clouds, characterizing gas exchanges and searching for volcanic activity.
- **VenSpec-U** (UV): will map distribution and spatial and temporal variations of sulphur-bearing gases (SO, SO_2) and unknown particulate absorbers at the cloud tops. It aims to support volcanic activity detection by constraining atmospheric dynamics variability.

VenSar, EnVision's synthetic aperture radar, is the main driver of the mission's scientific return when it comes to characterizing the surface of Venus. VenSAR will be designed and built by NASA's Jet Propulsion Laboratory (JPL) in an ESA–NASA partnership. As a dual-polarization S-band SAR, it can operate in three operating modes:

- The **SAR Imaging Mode** images the surface of Venus through its thick atmosphere. With a resolution of 10–30 m/pixel, EnVision will provide surface imaging data at an order of magnitude better than the currently available Magellan data at 100 m/pixel and will provide unprecedented detail of preselected

[1]Note that the SRS configuration has changed to the orientation shown in Figure 5.6a. This CAD model was produced before the change.

Regions of Interest (RoI's) on Venus's surface. For these types of observations, the radar must be pointed at the surface of Venus with an incidence angle of 20°–40°. Stereo imaging (observations of the same RoI from two different incidence angles) will offer topographic information at 300 m spatial and 20 m vertical resolution in conjunction with altimetry mode.

- In **Altimetry Mode**, VenSAR is pointed towards nadir and will measure the surface topography of Venus with a vertical resolution of 2.5 m. This will allow for the generation of high-resolution Digital Elevation Models (DEM's) of the surface, which can be used to study surface deformation, tectonics, and volcanic processes.
- The **Radiometry Mode** will measure the surface emissivity and temperature of Venus, by measuring microwave emissivity with a precision < 1.0 K and an absolute radiometric accuracy < 2.0 K. It is able to observe in both horizontal (H) and vertical (V) polarizations passively, and supports active dual-polarimetry in HH, and HV modes.

The **Subsurface Radar Sounder (SRS)** is the first instrument to profile Venus's subsurface, studying buried geological structures (like fractures, layering, and flow features) and subsurface materials up to 1 km deep. It complements VenSAR by using much longer wavelengths, allowing deep penetration and additional surface characterization. The 16-metre long deployable sounder is built on heritage from the JUICE mission's RIME instrument [66], which like the SRS, is designed to characterize the subsurface of the Jovian moon Ganymede.

The **Radio Science Experiment (RSE)** will aim to extract scientific data from the spacecraft's radio link with Earth. The experiment is divided in two specific experiments:

- **Gravity Experiment:** measuring Venus's gravity field with a 140–200 km spatial resolution by tracking the spacecraft's orbital velocity, comparing Doppler shifts of the received frequency with respect to the transmitted frequency.
- **Radio-occultation Experiment:** Uses the received signal from the spacecraft's radio link through the atmosphere of Venus to extract information about its atmospheric temperature, pressure structure, and abundance of sulphuric acid.

Together, the six scientific payloads aim to provide a holistic understanding of Venus. Their scientific return is not meant to be taken as stand-alone data, but rather as a synergistic approach to studying the planet from core to upper atmosphere. Their distinct roles in EnVision's scientific synergy is illustrated in Figure 5.4 by positioning each instrument, and their measurement objectives, in its spatial context.

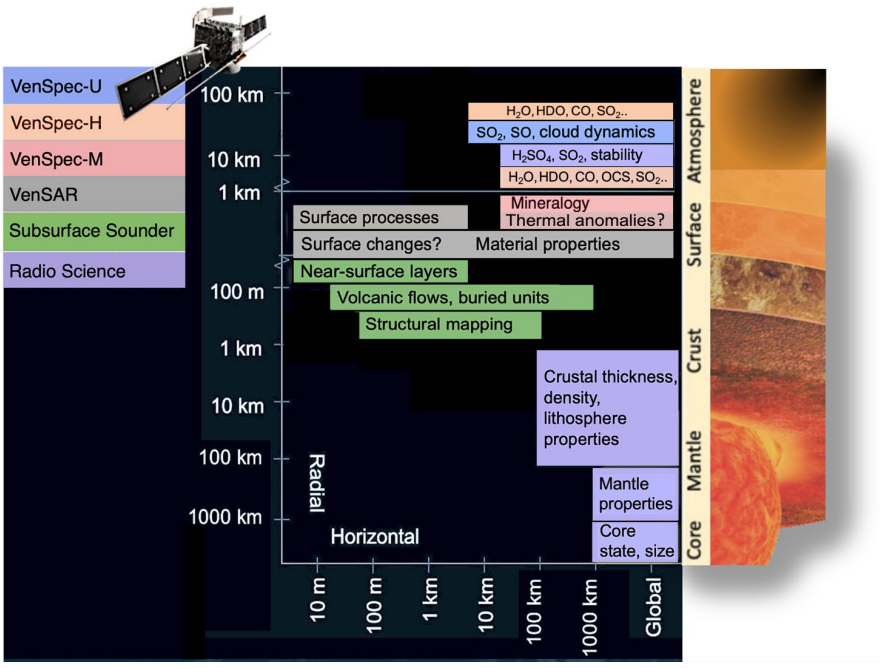


Figure 5.4: 6 scientific payloads' science targets, and their spatial scope on Venus. [18]

As an M-class mission, EnVision is designed with a 'design-to-cost' approach. This imposes certain constraints on the mission architecture and design choices, to keep its CaC within bounds. This design philosophy results in the following key design choices [18]:

- A body-fixed dual-band High Gain Antenna (HGA) is used over a steerable HGA, to limit the on-board solid state mass memory size, and to rely on mature, high TRL technologies for all subsystems.
- Electric propulsion solutions are discarded (studied in phase 0/A), which would also fulfil the mission requirements but at a significantly higher cost.
- The use of aerobraking becomes mandatory to reach the desired science orbit with the limited (propellant) mass budget in place. The science orbit itself can only be coarsely controlled as well.
- The complexity and duration of ground operations is limited to save on costs. The mission duration is adjusted to six Venus cycles to confidently fulfil all science requirements.

The number of instruments, their position on the platform (see Figure 5.3), the fixed HGA, and limited observation opportunities all contribute to the need for careful planning of the mission's science operations. The impact will be covered in the first system model of the spacecraft's attitude in section 5.3.

5.2. EnVision PAS3 Configuration

Having a solid understanding of the EnVision mission, spacecraft platform and instruments, the next step is to configure the PAS3 toolkit for the specific mission case. The intended workflow of implementing the PAS3 toolkit in a mission analysis context is to first create a *mission configuration file*. This configuration file (`config.py`) contains the mission-specific PAS3 objects, relevant parameters, simulation settings and additional baseline custom models. The intent is to then import this configuration file into separate *sub-scripts*, where the models for the required mission/system analysis are created and executed. This modular approach has the following advantages:

+ **Separation of concerns:** The mission configuration file focuses solely on defining the mission objects, parameters and settings, while the analysis scripts can focus on the specific modelling and analysis tasks. This separation makes it easier to manage and update each component independently.
+ **Consistency & reusability:** By having a single mission configuration file, all analysis scripts can use the same baseline mission parameters and settings. This ensures consistency across different analyses and reduces the risk of discrepancies due to different configurations. When changes occur in the system (design iterations) or mission (change in planning), only the configuration file needs to be updated, rather than modifying each analysis script individually. The users have full control over the version control of the file and scripts; be it via Git, scheduled team updates, or a non-destructive "add-only" approach.
+ **Efficiency:** Having a centralized configuration file eliminates the need to duplicate simulation and mission setup code in each analysis script. In essence, once the config file is created, the only concern of the analysis script developer (simulation-wise) is the straightforward implementation of the simulation loop in Listing 5. No SPICE know-how is required by the analysis script developer; they can solely focus on the model, not the simulation development.

The setup of a PAS3 configuration file is illustrated in Figure 5.5. The file contains the PAS3 objects (*Simulation*,

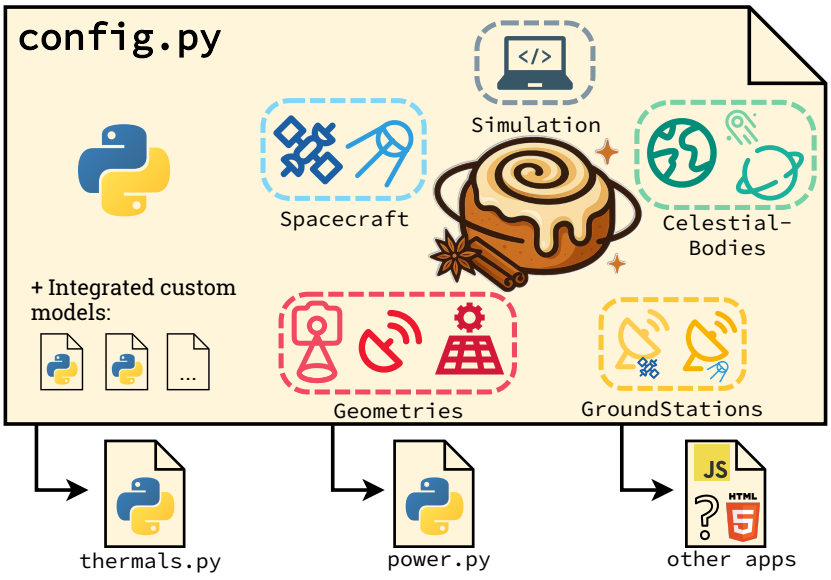


Figure 5.5: PAS3 Configuration file setup and usage.

Spacecraft, etc.), as well as integrated custom models that are not part of an analysis script, yet are necessary for the other models to function correctly. In the case of EnVision, this includes models of the science operations and attitude profile. This will be discussed in section 5.3. In essence, the collection of the configuration file and its sub-scripts form the desired **EnVision System Simulator (SS)** platform from section 3.3.

Within the EnVision configuration file, the first PAS3 object created is the *Simulation* object. The start of EnVision's `config.py` file is shown in Listing 13. Immediately, the mission-specific nature of the configuration file becomes apparent with the custom integrated model "VespaData" imported from a separate module and used to configure the right start and end times. This class contains the planned science operations data for EnVision and will be covered in more detail in subsection 5.3.1. The necessary SPICE kernels are given in a list of file paths and loaded at initialization. The automatic simulation coverage settings are overwritten by the custom science operations data. Finally, some default simulation settings are applied: the default inertial frame is set to J2000, the occultation tolerance is set to 1 (full visibility) and the solar constant is set to 1362 W/m^2.

```
# ============================================================================================
# -------------------------------- Configure Simulation --------------------------------
# ============================================================================================
import pas3 as p3
from config.vespa_pointing_tool import VespaData

# Setup VESPA data for correct SciOps coverage
SciOps = VespaData(...)

sim = p3.Simulation(
    kernel_paths=["envision_v03_custom.tf", "estrack_v04.tf", "earth_topo_050714.tf",
    ↪  "earthfixeditrf93.tf", "envision_vensar_v00.ti", "envision_venspec_v02_custom.ti",
    ↪  "naif0012.tls", "envision_200715_fict.tsc", "pck00010.tpc", "de-403-masses.tpc",
    ↪  "earth_200101_990628_predict.bpc", "de432s.bsp", "estrack_v04.bsp",
    ↪  "earthstns_itrf93_050714.bsp", "EnVision_ET1_2031_NorthVOI_v01.bc",
    ↪  "EnVision_baseline_ET1_2031_e0_NorthVOI 1.bsp"],
    start_time=SciOps.cycle_start_times[0],         # Start of first science cycle
    end_time=SciOps.cycle_end_times[-1],            # End of last science cycle
)
sim.set_default_frame("J2000")                      # Default inertial frame to use
sim.set_occultation_tolerance(1)                    # visibility True with full visibility
sim.set_solar_constant(1362)                        # W/m² at 1 AU
```

Listing 13: Simulation setup in the EnVision PAS3 configuration file.

Next, the celestial bodies and spacecraft are configured in Listing 14. The Earth and Venus objects are created first, followed by the EnVision spacecraft object, which is associated with Venus as its central body. By default, the celestial bodies' fixed reference frames are set to "IAU_EARTH" and "IAU_VENUS", for deomonstrative purposes, the Earth body-fixed frame is changed to "ITRF93" in this example (which is a high-fidelity dynamic SPICE frame). The EnVision spacecraft is set to not use SPICE CK data for its attitude, as a custom integrated attitude model will be implemented in subsection 5.3.2. If no CK kernel was loaded, PAS3 would have defaulted

```
# ============================================================================================
# ---------------------- Configure Celestial Bodies & EnVision ---------------------------
# ============================================================================================
Earth = p3.CelestialBody("EARTH", sim)
Venus = p3.CelestialBody("VENUS", sim)

EnVision = p3.Spacecraft("ENVISION", Venus, sim)

Earth.set_body_fixed_frame("ITRF93")  # Set Earth body-fixed frame to ITRF93
EnVision.use_spice_attitude(False)    # Disable SPICE CK, use custom attitude model instead
```

Listing 14: Earth, Venus and EnVision setup in the EnVision PAS3 configuration file.

to not using SPICE attitude data anyway.

With EnVision set up as a spacecraft object, we can proceed to configure its ground stations. Three ESTRACK deep-space ground stations are used for communication with EnVision: New Norcia, Cebreros and Malargue. These are created as *SpiceGroundStation* objects in Listing 15 and assigned to the EnVision spacecraft.

```
# ============================================================================
# ---------------------- Configure Ground Stations --------------------------
# ============================================================================
New_Norcia = p3.SpiceGroundStation("NEW_NORCIA", sim)
Cebreros   = p3.SpiceGroundStation("CEBREROS",   sim)
Malargue   = p3.SpiceGroundStation("MALARGUE",   sim)

EnVision.assign_ground_stations(New_Norcia, Cebreros, Malargue)
```

Listing 15: Ground station setup in the EnVision PAS3 configuration file.

Now, the configuration file will create objects of the payload instruments in Listing 16. The VenSpec suite instruments were initialized using their SPICE IK files. Note that the body locations of the instruments are not included in these snippets. Because the VenSpec FOV angles remain relatively fixed throughout the mission development, they are defined in the SPICE IK files instead of using *CustomInstrument* objects.

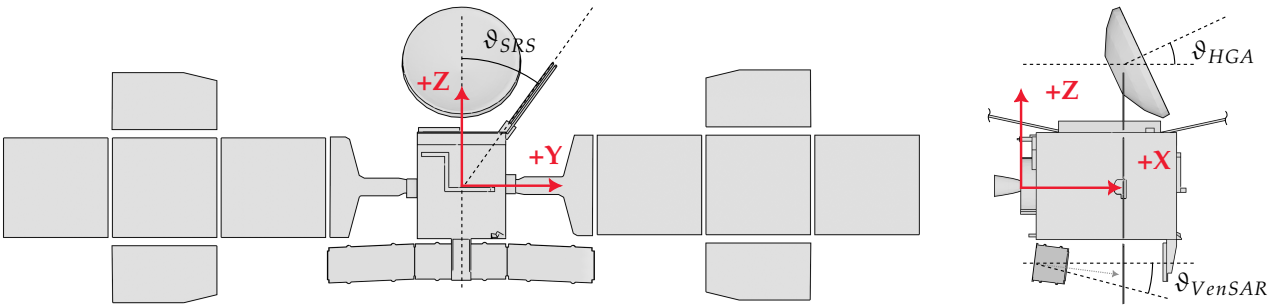
```
# ============================================================================================
# ----------------------- Configure VenSpec SPICE Instruments ----------------------------
# ============================================================================================
VenSpec_M = p3.SpiceInstrument(EnVision,  "ENVISION_VENSPEC_M")
VenSpec_H = p3.SpiceInstrument(EnVision,  "ENVISION_VENSPEC_H")
VenSpec_U = p3.SpiceInstrument(EnVision,  "ENVISION_VENSPEC_U")
```

Listing 16: VenSpec instruments setup in the EnVision PAS3 configuration file (body locations not included).

The remaining instruments: VenSAR, SRS and RSE (which is the HGA), are configured in Listing 17 by the use of non-SPICE initialization methods. Their boresight vectors are defined by rotating a unit vector along one of the spacecraft axes by a certain tilt angle. The VenSAR reflectarray is tilted 7.14° from the –X axis around the +Y axis, resulting in a reflected ray tilt of 14.28°. The SRS is tilted 36° from the +Z axis aroud the +X axis.

```
# ============================================================================================
# ------------------------ Configure Additional Instruments ----------------------------
# ============================================================================================
from math import radians, cos, sin
# VenSAR:
VENSAR_TILT     = radians(...)                                      # Reflectarray tilt
VENSAR_RAY_TILT = VENSAR_TILT * 2                                   # Reflected beam tilt
VENSAR_vec      = [cos(VENSAR_RAY_TILT), 0, -sin(VENSAR_RAY_TILT)]  # Boresight vector
VenSAR = p3.CustomInstrument(EnVision, "ENVISION_VENSAR", [VENSAR_vec], "CIRCLE", 5.0)

# SRS:
SRS_TILT = radians(...)                                             # SRS boresight tilt
SRS_vec  = [0, sin(SRS_TILT), cos(SRS_TILT)]                        # Boresight vector
SRS = p3.CustomInstrument(EnVision, "ENVISION_SRS", [SRS_vec], "CIRCLE", 5.0)

# HGA (RSE):
HGA_TILT = radians(...)                                             # HGA LOS tilt
HGA_vec  = [cos(HGA_TILT), 0, sin(HGA_TILT)]                        # Boresight vector
HGA = p3.Geometry(EnVision, "High Gain Antenna", [HGA_vec], "RAY")
```

Listing 17: VenSAR, SRS and RSE instrument PAS3 setup in configuration file (body locations not included).



(a) Front view **(b)** Side view

Figure 5.6: Envision instrument tilt angles: **(a)** SRS tilt angle ϑ_{SRS}, **(b)** VenSAR tilt angle ϑ_{VenSAR} and HGA tilt angle ϑ_{HGA}.

The HGA is tilted 25° from the +X axis around the -Y axis. Note that the half angles of the circular FOV's of VenSAR and SRS are only configured for visualization purposes in chapter 6. More realistic beamwidths can be defined when the analysis requires it. The tilt angles and boresight vectors of these three instruments are illustrated in Figure 5.6.

In addition to the instruments, some more general spacecraft geometries are configured in Listing 18. The 6 spacecraft panels are defined as half-space geometries, aligned with the spacecraft body axes (see Figure 5.6). The –Z panel is defined as the cold face of the spacecraft. Additionally, a sun impingement constraint zone during Earth-communications is also configured. These constraint zones will be used (and explained further) in the attitude model in subsection 5.3.2.

```
# ============================================================================================
# -------------------------- Configure EnVision Geometries -------------------------------
# ============================================================================================
# +X and -X Panels
PX = p3.Geometry(EnVision, "+X Panel", [[ 1, 0, 0]], "HALF_SPACE")
NX = p3.Geometry(EnVision, "-X Panel", [[-1, 0, 0]], "HALF_SPACE")
# +Y and -Y Panels
PY = p3.Geometry(EnVision, "+Y Panel", [[0,  1, 0]], "HALF_SPACE")
NY = p3.Geometry(EnVision, "-Y Panel", [[0, -1, 0]], "HALF_SPACE")
# +Z and -Z Panels
PZ = p3.Geometry(EnVision, "+Z Panel", [[0, 0,  1]], "HALF_SPACE")
NZ = p3.Geometry(EnVision, "-Z Panel", [[0, 0, -1]], "HALF_SPACE")

# Constraint zones:
ColdFace = NZ # The -Z panel is the cold face
n_vec = [cos(HGA_TILT), 0, -sin(HGA_TILT)]
Comms_sun_exclusion_zone = p3.Geometry(EnVision, "Comm. excl. zone", [n_vec], "HALF_SPACE")
```

Listing 18: Spacecraft geometries setup in the EnVision PAS3 configuration file.

This concludes the basic EnVision PAS3 configuration file setup. More PAS3 geometries and integrated models are expected to be initialized in the `config.py` file for the upcoming model sections. The intention is to build upon this configuration file throughout mission design and storing the EnVision setup in a repository accessible to team members, to be imported as a submodule in their own scripts.

5.3. Science Operations & Attitude Model

This whole chapter aims to build models utilizing the PAS3 toolkit as a proof of concept of its intended use. For the EnVision mission case study, the first model to be built is an attitude model. EnVision carries instruments and an antenna that require a specific attitude of the whole spacecraft to operate correctly. Combine this with the nature of a Venus-centred medium-class mission, where the spacecraft has a fixed HGA, limited lifetime, high thermal constraints, limited Solid State Mass Memory (SSMM) capacity for heavy SAR data, limited repeat observation opportunities, and varying link budgets with Earth. The combination of these factors does not only give rise to a pointing-intensive attitude profile, but also a highly aperiodic one. This makes sizing analysis on attitude-dependent parameters of the spacecraft more challenging. The factors influencing the

spacecraft's aperiodic attitude profile are summarized in Table 5.1.

Table 5.1: Pointing considerations for the EnVision spacecraft: mission properties and their influence on spacecraft aperiodic attitude [18]

Origin	Factors	Attitude Consequences
Instruments	VenSAR imagery requires incidence angle between 20° – 40°.	Off-Nadir pointing required with varying off-nadir angles.
	Standard Nadir-pointing differs between VenSpec spectrometers, VenSAR altimetry and SRS operational modes.	Nadir-pointing attitude roll angle differs between instrument operational modes.
	VenSpec instruments need regular dark- and sun-calibration.	Specific occasional pointing profiles.
	VenSAR produces large volumes of data, while SSMM is limited (design-to-cost limitations).	Necessary to allocate many communication timeslots.
	Steady-State sun impingement must be avoided in certain VenSpec units Field Of Views (FOV's).	Pointing constraints.
Platform	The spacecraft has one face (with VenSpec radiators) that must avoid steady state solar illumination (called the *cold face*).	Pointing constraints.
	Design-to-cost approach results in a fixed high-gain antenna (HGA) for communication and the RSE, rather than a steerable one.	Requires the whole spacecraft to point towards the Earth.
Environment	Variability of communication link budget and visibility with Earth.	Earth-pointing communication modes are not distributed evenly.
	Limited mission lifetime due to high thermal loads in the Venus environment.[1]	Need for careful planning of observations (and pointing).
	Slow rotation rate of Venus[2] makes SAR repeat observation opportunities limited.	Careful planning of SAR pointing based on RoI passes is necessary.

Note that for the current status of the project, and the desired activities of the project team, the attitude is decided to be taken as an input for the sizing analysis, rather than an output. The attitude profile is therefore built based on a reference science operations timeline, and does not consider slew times, dynamical perturbations or transition profiles. The attitude model is therefore a steady-state representation of the spacecraft's pointing profile. Throughout the mission, EnVision's near-polar orbit will have a period of approximately 90 minutes. The currently used reference operational timeline (introduced in subsection 5.3.1) divides the operations in 45-minute intervals; with EnVision's agility requirements and Sun-avoiding slew capabilities, it was decided internally this approximation would be sufficient for the intended sizing analysis.

An overview of the science operations is described subsection 5.3.1. The procedure of translating this operations profile into a spacecraft attitude model for the EnVision PAS3 configuration is given in subsection 5.3.2 (using one mode as example). The model is then verified in subsection 5.3.3.

5.3.1. SciOps Overview

Coordinated effort between the mission performance team and the SOC has resulted in a repository of software tools and scripts to aid the Science Operations (SciOps) planning. The main tool used in the presented work will be VESPA (Venus Science Planning Assistant), which is an ecosystem of tools assisting the analysis of EnVision mission's performance. VESPA automates the assignment of science operations profiles, maximizing data return and science phase performance. After discussion with the EnVision project team, it was decided the VESPA reference science operations timeline will be used as input for the attitude model. This gives the SciOps team within the EnVision project the opportunity to link the mission performance output directly to a system-analysis platform, while having the flexibility to change operations and planning. Using this approach

[1] A Venus orbit can significantly reduce mission lifetime. Venus Express had a lifetime of 8 years, compared to the 20+ years of its 'sibling' Mars Express [67]

[2] The rotation period of Venus is about 243 days [68]. In EnVision's near-polar orbit and planned lifetime, this equates to 6 opportunities (6 Venus cycles) to make repeat SAR observations.

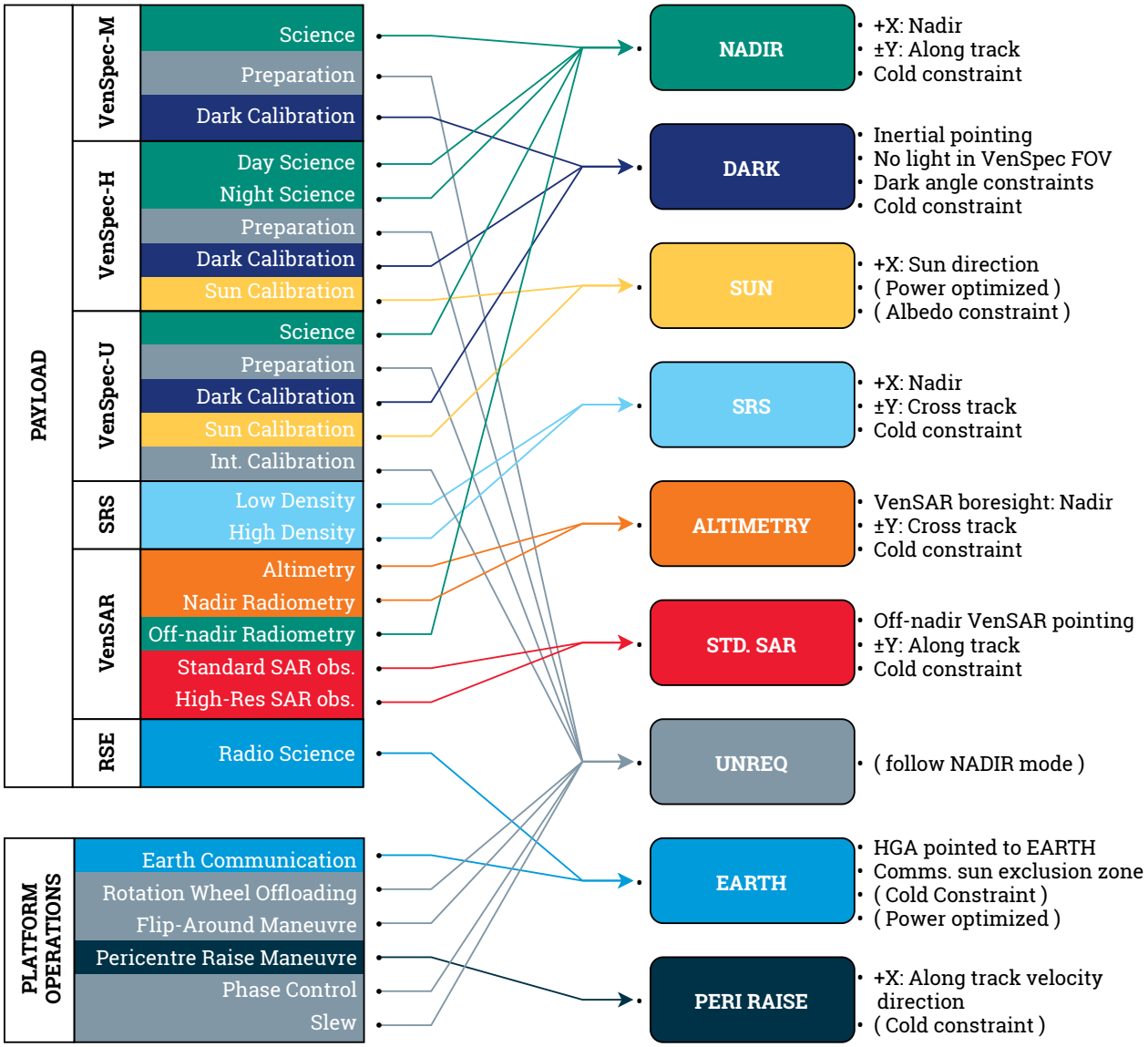


Figure 5.7: Payload and platform operational modes and their nominal pointing requirements and constraints. Text between brackets indicates preferred pointing conditions for modelling purposes.

provides more agility in the design and analysis process compared to a pre-defined SPICE CK kernel approach.

Vespa allocates the science operations into half-orbits called *arcs*, these arcs get a number associated to it based on the science or general operational activity taking place. Figure 5.7 illustrates the different operational modes of the spacecraft; VESPA takes these modes and assigns them to the arcs based on the science return requirements, data volume, and communication opportunities. Operational modes that can be combined are grouped together, creating an extensive timetable of distributed operational modes. The interest of this section lies in extracting an attitude profile from the operational modes. In total 9 pointing modes are defined; they are:

- Nadir pointing (NADIR)
- Dark calibration pointing (DARK)
- Sun calibration pointing (SUN)
- Nadir SRS orientation (SRS)
- VenSAR altimetry mode (ALTIMETRY)
- VenSAR off-nadir imaging mode (STD. SAR)
- Earth-pointing communication mode (EARTH)
- Orientation for a propulsive pericenre-raising manoeuvre (PERI RAISE)
- Unspecified/Unrequired pointing (UNREQ)

Each mode has its own set of pointing requirements and constraints, which are summarized in Figure 5.7[1]. Note that the UNREQ mode is assigned to operations dat do not require a specific orientation. For modelling purposes, the default NADIR mode is mapped onto these operations.

In order to orient the spacecraft in the correct steady attitude during an operational mode, at least two body axes must be defined, the whole body-fixed frame can then be completed following the right-hand rule. In most cases, one axis must point towards a specific target (nadir, Earth, Sun), the spacecraft then has a degree of freedom around this axis (necessary roll angle). The second axis is then either set by specific observation orientation requirement, a pointing constraint, or a combination of both. For example, in the NADIR mode, the +X body axis must point towards nadir (pointing requirement), then the body is rolled to ensure the Y-axis is in the along track direction (orientation requirement), then the direction of the +Y axis along track is selected to ensure the cold face is not illuminated by the Sun (cold constraint). Some constraints are:

- Cold constraint: No steady state sun impingement on the -Z panel of the spacecraft.
- Dark angle constraint: during dark VenSpec calibration, the angle between the nadir and VenSpec line of sight must be > 90°.
- Comms. Exclusion zone: during Earth-pointing communication, the cold constraint cannot always be satisfied. A new region of prohibited sun impingement is defined (referred to as communication exclusion zone).

In some modes, hard orientation requirements or constraints are not sufficient to define the second axis, therefore a preferred orientation is defined (between brackets in Figure 5.7). These are:

- (Power optimized): Orientation minimizes the angle of incidence of the Sun on the solar arrays, maximizing power generation. With EnVision's rotatable arrays on the Y panels, this means the sun vector is in the body XZ-plane of the spacecraft.
- (Albedo constraint): Similar to the cold constraint, but instead of sun impingement, albedo reflections from Venus must be avoided on the -Z panel.
- (follow NADIR mode): As explained earlier, modes without a specific pointing requirement are mapped onto the NADIR mode.
- (Cold Constraint): during Earth pointing communication, the cold constraint is preferred to be satisfied if possible.

Within the EnVision SS repository, a helper class called *VespaData* was configured which reads a VESPA-generated timeline Excel sheet, reads a pointing description file (like Figure 5.7), and outputs a similar timeline but with the corresponding pointing modes with every arc as a serialized `pickle` file of the corresponding `pandas` data frame. This ensures the timeline can be easily read and processed in Python's own binary format, without having to read a spreadsheet before every simulation. Even with an "operations resolution" of half-orbits (about 45 minutes), for 6 Venus-cycles, this results in about 45,000 modes to read. Before the attitude model is initialized, the timeline is *unpickled*, and the pointing modes are extracted at arbitrary time steps using the `numpy.searchsorted()` binary search method using the *VespaData* class method `get_pointing(<ephemeris time>)`. Similarly, the `get_activity(<ephemeris time>)` method is used to extract the science / operational activity taking place during an arc. The data objects have additional useful attributes like SciOps coverages (start and end times), operational activity and pointing modes data frames. Because the SciOps timeline and provided SPICE kernels do not overlap perfectly, the SciOps coverage is used manually in EnVision's PAS3 simulation configuration to ensure correct attitude profiles are used during analysis (see the EnVision PAS3 simulation instance setup in Listing 13).

Using this class, simulations can be easily run using different VESPA timelines or operational modes. Within the EnVision SS configuration, a *VespaData* object is created, either by reconfiguring a new SciOps timeline using the `reconfigure=True` flag (taking a bit more than a minute), or by reading an existing (serialized) timeline with `reconfigure=False` (about 20 ms on the same machine). Setting up SciOps objects is shown in Listing 19.

Following the definition of the 9 operational modes, together with the SciOps timeline provided by the VESPA tool and the *VespaData* mode accessing, the next step is to create the attitude model using the PAS3 toolkit. This will be shown in the next subsection.

[1]Note that VenSAR Nadir radiometry in Figure 5.7 is performed in the spacecraft ALTIMETRY mode and Off-Nadir radiometry in spacecraft NADIR mode. This is due to the VenSAR reflectarray tilt (see Figure 5.6b) looking nadir in altimetry mode and off-nadir when the spacecraft looks nadir.

```
SciOps1 = VespaData("configured_vespa_timeline", reconfigure=False) # Pickled data (20 ms)
Sciops2 = VespaData("new_vespa_timeline", reconfigure=True)         # New timeline (1 min)
```

Listing 19: Loading EnVision SciOps timeline using the custom VespaData class.

5.3.2. Attitude Model Setup

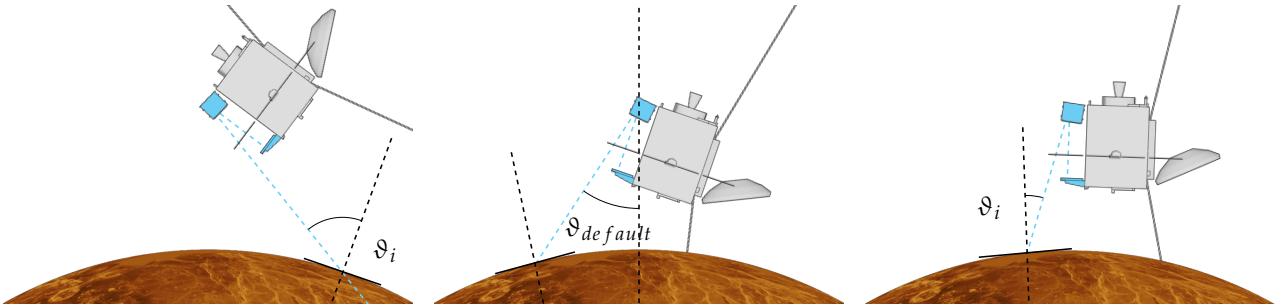
This subsection will briefly showcase the approach taken to model the pointing of the spacecraft following the VESPA operational timeline from subsection 5.3.1 and the PAS3 EnVision configuration from section 5.2. Covering all 9 operational modes would be too extensive for this report; therefore, only the EARTH mode will be covered in detail, as it nicely illustrates the use of the toolkit. The other modes are modelled in a similar fashion, taking into account their specific pointing requirements and constraints.

Within the EnVision SS, the module `attitude_generator.py` was created. It includes a function dispatching dictionary, connecting the operational modes to their corresponding attitude generation functions.

```
POINTING_METHODS = {
    "NADIR":            point_nadir,
    "EARTH":            point_earth, # This one is explained in detail below
    "ALTIMETRY":        point_altimetry,
    "SAR_NRA_MAX":      point_nra_max,
    "SAR_PRA_MIN":      point_pra_min,
    "SAR_PRA_DEFAULT":  point_pra_default,
    "SUN":              point_sun,
    "PERI_RAISE":       point_peri_raise,
    "DARK":             point_dark,
    "UNReq":            point_nadir, # Assume NADIR for unrequired pointing
    "SRS":              point_srs_config_3b, # Result of different SRS configurations
    "FLIP":             point_nadir  # Flipping manoeuvre assumes default NADIR (for now)
}
```

Listing 20: Pointing scripts dispatching dictionary in the EnVision attitude generator.

Note that the SAR imaging pointing is divided into three different functions; this is because the off-nadir pointing requires different incidence angles depending on the type of SAR imagery. Also, during such off-nadir pointing, the spacecraft can be rolled around the Y-axis in a positive roll angle (PRA) or negative roll angle (NRA) direction. While standard SAR observations can happen between incidence angles of minimum 20° and maximum 40°, two specific cases are modelled: the maximum off-nadir angle (40°) in the NRA orientation (worst case for albedo impingement on cold face and Sun impingement in VenSpec FOV's), and the minimum off-nadir angle (20°) in the PRA orientation (worst case for albedo impingement in the VenSpec FOV's). The DEFAULT function points to a baseline off-nadir look angle (not incidence angle) of 29° in PRA orientation. The choice of including both NRA_MAX and PRA_MIN (instead of assuming only default SAR orientations) will become apparent in section 5.4. The three variations are shown in Figure 5.8.



(a) Maximum off-nadir incidence angle ($\vartheta_i = 40°$) in NRA orientation.

(b) Default off-nadir look angle ($\vartheta_{default} = 29°$) in PRA orientation.

(c) Minimum off-nadir incidence angle ($\vartheta_i = 20°$) in PRA orientation.

Figure 5.8: Three variations of EnVision's VenSAR standard off-nadir pointing modes modelled in the current attitude model (angles not to scale).

The dispatching in Listing 20 is used in a general attitude update function in Listing 21. The function extracts the pointing mode from the SciOps timeline, checks if the mode is recognized, and calls the corresponding function to update the attitude of the spacecraft.

```python
from envision_config import EnVision, SciOps

def update_attitude(pointing_id: str | None =None) -> None:
    # Get pointing mode from SciOps timeline or input argument (manually setting possible)
    if pointing_id is None:
        pointing_id = SciOps.get_pointing(sim.et)
    # Check if pointing mode is recognized:
    if pointing_id in POINTING_METHODS:
        POINTING_METHODS[pointing_id]() # Call corresponding pointing function
    else:
        print(f"Warning: Pointing mode ({pointing_id}) not recognized.")
        point_nadir() # Default to NADIR

    # Store pointing mode
    EnVision.pointing_mode = pointing_id
```

Listing 21: Attitude update function in the EnVision SS attitude generator.

To point EnVision in the EARTH communication pointing mode (`point_earth()` in Listing 20), first an initial orientation is calculated. Following the power-optimized constraint in Figure 5.7, a plane is defined between the Sun (`EnVision.sun_direction`) and Earth (`EnVision.earth_direction`). The +Y body axis is aligned perpendicular to this plane. To complete the initial orientation, the +X (or +Z) body axis is defined by rotating the Earth direction vector by the HGA tilt angle around the +Y axis using Rodrigues' rotation formula (see Equation 5.1), where $\vec{Y}$ is the +Y body axis, $\vec{X}$, the +X body axis, and $\vec{v}_{sun}$ and $\vec{v}_{earth}$ are the Sun and Earth direction vectors respectively. ϑ_{HGA} is the HGA tilt angle w.r.t the +X body axis. The right-handed body-fixed frame can then be completed.

Figure 5.9 illustrates the geometry of the initial orientation. EnVision is positioned in the power-optimized plane (the page), with the HGA pointing towards Earth. The figure shows four cases of the Sun's position that contribute to the second step in defining the final orientation; the constraint handling. The initial setup has two potential orientations satisfying the Earth-pointing requirement, as well as a power-optimized orientation, namely one that flips the spacecraft by 180° around the HGA axis (transparent in Figure 5.9). Selecting whether the spacecraft should be flipped is done by following a constraint handling procedure. As mentioned before, the cold face (in blue) of the spacecraft should desirably not have any sun impingement; case **c)** in Figure 5.9 would naturally need a flipped spacecraft to satisfy the cold constraint. However, in case **d)**, the cold constraint cannot be satisfied in both orientations. For this reason, a communication exclusion zone (in red) is defined, which ensures a preferred orientation is always possible during Earth communication modes. Case **b)** clearly does not violate any constraints, and can remain in the initial orientation. Case **a)** shows the Sun is in the communication exclusion zone, but the cold constraint can be satisfied. In this case, the spacecraft is flipped to satisfy both.

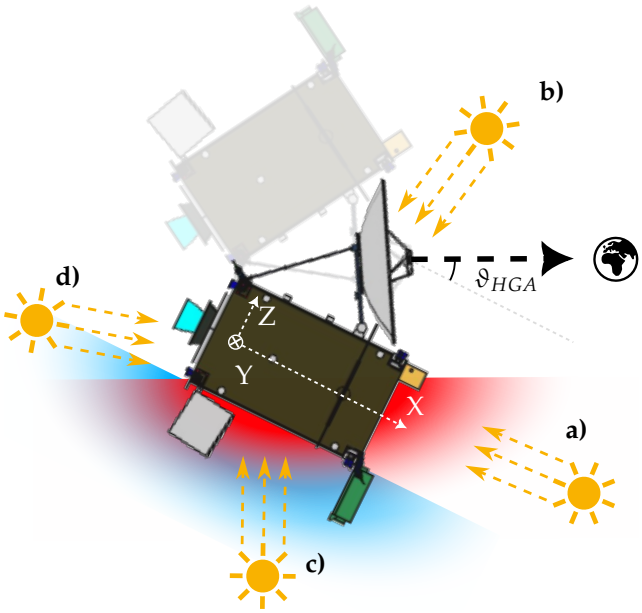


Figure 5.9: Two power-optimized orientations of EnVision during Earth-pointing communication, with four different Sun positions. The cold face is shown in blue, the communication exclusion zone in red; with ϑ_{HGA} the tilt angle of the HGA w.r.t the +X body axis.

$$\vec{X} = \vec{v}_{earth} cos\left(\vartheta_{HGA}\right) + \left(\vec{Y} \times \vec{v}_{earth}\right) sin\left(\vartheta_{HGA}\right) + \vec{Y}\left(\vec{Y} \cdot \vec{v}_{earth}\right)\left[1 - cos\left(\vartheta_{HGA}\right)\right] \tag{5.1}$$

The constraint checking procedure is shown in the flowchart in Figure 5.10. The procedure first checks if the cold constraint can be satisfied, if not, flips the spacecraft and checks if in the new orientation, both constraints are violated. If so, the spacecraft is flipped back to the initial orientation, having violated the cold constraint, but satisfying the communication exclusion zone constraint. When the cold constraint can be satisfied in the initial orientation, the procedure checks if the Sun is in the communication exclusion half-space. If this is the case, the spacecraft is flipped to satisfy both constraints, if not, it remains in the initial orientation. The cases illustrated in Figure 5.9 are indicated in the flowchart as well.

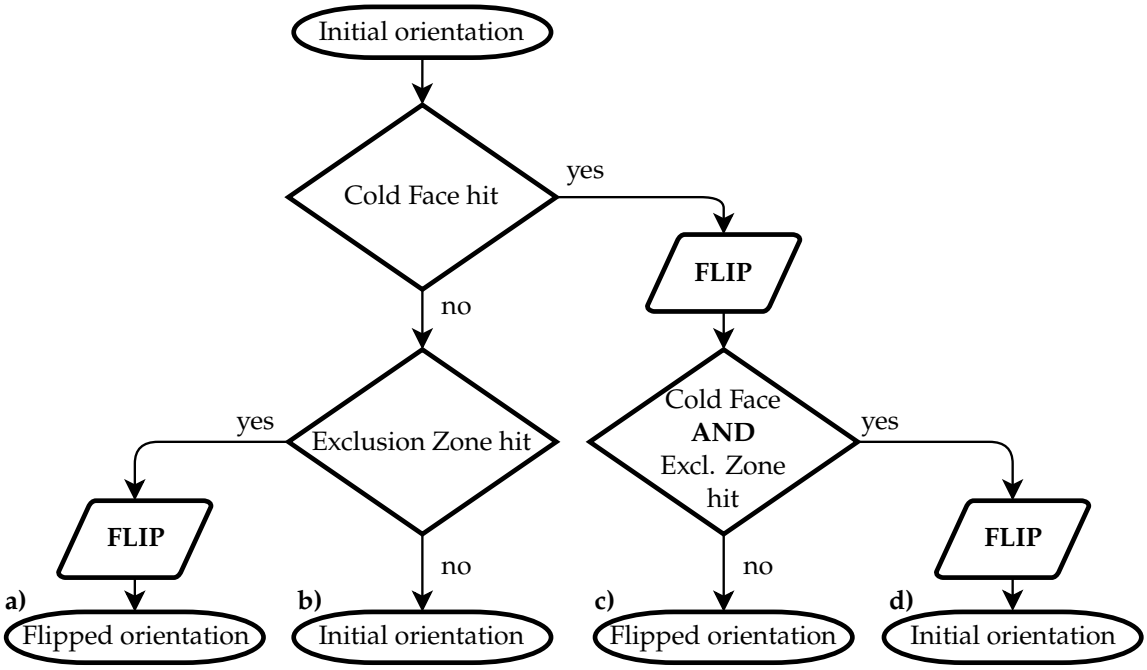


Figure 5.10: Constraint handling flowchart for EnVision Earth-pointing communication mode. Four sun-constrained cases (a, b, c, d) are illustrated in Figure 5.9.

PAS3 offers some convenient functions to support vector operations to set up body rotation matrices. Examples are: `rotate_around_axis(<v, axis, angle>)` to rotate a vector around an arbitrary axis, `normalize(<v>)` to normalize a vector, and `get_orthogonal_coplanar_vector(<fixed vector, reference vector>)` to get a vector orthogonal to a fixed vector, but as close as possible to a reference vector (used to get along-track directions). operations present in `numpy` are used directly in the scripts (like `np.cross()` and `np.linalg.norm()`). Some `SpiceyPy` functions are replaced by utility functions as they have a slightly lower performance (slight improvement in simulation time). Examples are `spiceypy.vhat(<v>)` replaced by `normalize(<v>)`, and `spiceypy.vsep()` replaced by `angle_between(<v1, v2>)`.

Using the same procedure as above (i.e. setting up an initial orientation, then handling constraints), the other pointing modes are modelled as well; using the PAS3 utils functions and*Spacecraft* and *CelestialBody* state attributes. It is recognized that the attitude model can be improved by the use of quaternions, and/or a more sophisticated analytical method of linking pointing and constraints, instead of using conditional constraint checks of the spacecraft orientation. For example: it was later realized, using the visualization results in chapter 6, that current pointing constraints can be reached by defining the +Y body axis as the cross product between the Sun and Earth direction vectors ($\vec{Y} = \vec{v}_{sun} \times \vec{v}_{earth}$), having a dynamic initial orientation and automatically satisfying the sun exclusion zone. This is only possible due to the symmetry of Sun exclusion zone w.r.t the Earth direction. The more stepwise constraint approach from Figure 5.9 was used first to increase readability and ease of understanding of the pointing logic. If at a later point, the exclusion zone becomes asymmetric or changes shape (see subsection 4.4.5), the constraint handling procedure can be reintroduced. The importance of this section is to illustrate the approach taken, as an analogy to the pointing and constraint handling in other modes.

Within the project, a more precise reference attitude profile is being built in the form of SPICE CK kernels by the SOC, which will replace the current steady-state attitude model in the future. However, the current model can still be used when operational timelines are changed, instrument accommodation (and thus pointing requirements) changes, or the pointing mode needs to be altered during runtime (for example safe mode injections).

5.3.3. Attitude Model Verification

In order to verify the attitude model, every pointing mode from Figure 5.7 was verified individually by running short simulations (one orbit) in a fixed pointing mode. To check the correctness of the modelled spacecraft attitude, the *Geometry* classes were used, together with their `get_angle_of_incidence(<direction>)` method, and the environmental (body-fixed) direction vector attributes from the *Spacecraft* actor. As shown in the EnVision PAS3 configuration in section 5.2, half-space geometries were configured for every panel of EnVision using the body-fixed axes as their plane normals. The choice of using the geometry classes for verification is made because their method outputs depend on the correct implementation of the spacecraft attitude. This ensures the verification does not simply reverse-engineer the steady pointing logic described in section 5.3, but rather checks for the correct implementation of attitude within the overall toolkit.

The verification orbit performs three consecutive checks. First, the pointing requirement is checked (for example: does the axis of the instrument point towards the desired target?). If the correct axis points towards the desired direction, the spacecraft has a rotational degree of freedom around this axis. The second check verifies if the correct roll angle is implemented (for example: is the along-track direction aligned with the +Y body axis?). Finally (optionally), the constraints are checked (for example: is the cold face not illuminated by the Sun?). The checks per pointing mode are summarized in Table 5.2.

Table 5.2: Verification checks per pointing mode.

	Pointing requirement	**Roll definition**	**Constraint checking**
NADIR	$\angle \vec{x}_+ \vec{v}_{nadir} = 0°$	$\angle \vec{y}_+ \vec{n}_{orbit} = 90°$	$\vec{v}_{sun}$ out of Cold Face half-space
EARTH	$\angle \vec{z}_+ \vec{v}_{earth} = 90° - \vartheta_{HGA}$ $\angle \vec{x}_+ \vec{v}_{earth} = \vartheta_{HGA}$	$\angle \vec{y}_+ \vec{v}_{sun} = 90°$	No Sunlight in the Sun-exclusion zone Ray *Geometry* for the HGA intersects the Earth *CelestialBody* actor.
ALTIMETRY	$\angle \vec{x}_+ \vec{v}_{nadir} = \vartheta_{VenSAR}$	$\angle \vec{y}_\pm \vec{n}_{orbit} = 0°$	$\vec{v}_{sun}$ out of Cold Face half-space
NRA_MAX	$\vartheta_i = 40°$	$\angle \vec{y}_+ \vec{n}_{orbit} = 90°$	$\angle \vec{z}_+ \vec{v}_{nadir} > 90°$
PRA_MIN	$\vartheta_i = 20°$	$\angle \vec{y}_+ \vec{n}_{orbit} = 90°$	$\angle \vec{z}_- \vec{v}_{nadir} > 90°$
PRA_DEFAULT	$\angle \vec{x}_+ \vec{n}_{nadir} = 29° - \vartheta_{VenSAR}$	$\angle \vec{y}_+ \vec{n}_{orbit} = 90°$	$\angle \vec{z}_- \vec{v}_{nadir} > 90°$
SUN	$\angle \vec{x}_+ \vec{v}_{sun} = 0°$	$\angle \vec{y}_+ \vec{v}_{nadir} = 90°$ $\angle \vec{y}_+ \vec{v}_{sun} = 90°$	$\angle \vec{z}_- \vec{v}_{nadir} > 90°$
DARK	$\angle \vec{x}_+ \vec{v}_{nadir} = \angle \vec{x}_+ \vec{v}_{sun}$	$\angle \vec{y}_+ \vec{v}_{sun} = 90°$	$\angle \vec{y}_+ \vec{v}_{nadir} > 90°$ $\angle \vec{y}_+ \vec{v}_{sun} > 90°$ No VenSpec FOV Sun impingement
SRS	$\angle \vec{z}_+ \vec{v}_{nadir} = \vartheta_{SRS}$	$\angle \vec{x}_\pm \vec{n}_{orbit} = 0°$	

For every iteration, first the all the body axes ($\vec{x}_+$, $\vec{y}_+$, $\vec{z}_+$) expressed in the J2000 inertial frame are checked for orthogonality and right-handedness as per Equation 5.2. The **+** or **-** suffixes below the axes vectors means the positive or negative direction respectively (the ± suffix in Table 5.2 represents the axis itself). Then, external vectors like nadir direction ($\vec{v}_{nadir}$), Sun direction ($\vec{v}_{sun}$), Earth direction ($\vec{v}_{earth}$), and the orbit normal vector ($\vec{n}_{orbit}$) are extracted from the *Spacecraft* actor (EnVision) expressed in its body-fixed frame using methods like: `EnVision.get_sun_direction_in_body_fixed_frame()`, `EnVision.to_body_fixed_frame( <normal vector>)`. The angles between the body axes and these vectors are calculated and checked with the expected values from Table 5.2. For example, checking if the +X body axis points towards nadir, the incidence angle on the +X panel geometry is calculated using `PX.get_angle_of_incidence(< -nadir direction >)`, with `PX` the +X panel half-space *Geometry*, attached to the PAS3 *Spacecraft* representation of EnVision. The expected angle is 0°. Constraint checking is done by either checking visibility of instruments / geometries towards celestial bodies, or more abstract angle checks; for example, $\angle \vec{z}_- \vec{v}_{nadir} > 90°$ is equivalent to checking if the -Z panel (or cold face) is pointed away from Venus. The three STD. SAR attitudes are verified by checking the VenSAR boresight incidence angle (ϑ_i) on the surface on Venus (using the PAS3 VenSAR plane abstraction *Geometry*

function `get_surface_aoi()`) for all but the default mode. The default mode is defined by a look angle w.r.t. the nadir direction (as seen in Figure 5.8) and VenSAR pointing can be verified more directly by checking if the +X body axis makes the correct angle with the nadir direction vector (taking into account the VenSAR tilt angle). Standard VenSAR observations are done with the +Y axis along-track (which is the roll definition) and the PRA and NRA constraints are checked by verifying if the correct Z panel is pointed away from nadir. Common roll definitions are $\angle \vec{y}_+ \vec{n}_{orbit} = 90°$ for +Y along-track, and $\angle \vec{y}_\pm \vec{n}_{orbit} = 0°$ for +Y pointing cross-track.

$$\begin{cases} \vec{x}_+ \cdot \vec{y}_+ = 0 \\ \vec{y}_+ \cdot \vec{z}_+ = 0 \\ \vec{x}_+ \cdot (\vec{y}_+ \times \vec{z}_+) = 1 \end{cases} \tag{5.2}$$

These verification orbits with fixed pointing modes were run for 100 uniformly distributed random start epochs within the SciOps coverage period (August 2034 – August 2038) with a time step fidelity of 5 seconds. While most could be verified using the incidence angle methods of the side panels (VenSpecs have the same LOS as the +X axis), some instruments are not aligned with the body axes (VenSAR, SRS and the HGA). Their tilt angles (ϑ_{VenSAR}, ϑ_{HGA} and ϑ_{SRS}) are used in combination with body axes in the verification assertions. In the constraints column, representations of the cold face and communication exclusion zone constraints (see Figure 5.9) are used, as well as the VenSpec FOV geometries to check for Sun impingement during DARK calibration mode. The relevant tilt angles and body axes are illustrated in Figure 5.6, while the constraining half-spaces can be seen in Figure 5.9.

With this verified attitude model, the attitude generation can be implemented in the EnVision PAS3 configuration as a custom model using the `add_custom_model()` method of the *Simulation* class (shown in subsection 4.4.1). The attitude update function from Listing 21 is called at every time step, updating the spacecraft's attitude based on the SciOps timeline. With this attitude representation, the consequent subsystem models automatically receive the correct spacecraft orientation during simulation runtime, without the need of importing this attitude module and calling it manually. The model is added in `config.py` as follows:

```
from attitude_generator import update_attitude
sim.add_custom_model(update_attitude)
```

Outputs:

```
Registered custom model: 'update_attitude'
Additional keyword arguments: ['pointing_id']
```

Listing 22: Adding the attitude update function as a custom model in the EnVision PAS3 configuration.

The keyword argument `pointing_id` is an optional string that can now be inserted into the update functions (`advance_time` and `set_time`) to manually overwrite the pointing mode inside the simulation loops (this will mainly be used in thermal models in section 5.4 and the visualization application in chapter 6). If not provided, the function extracts the pointing mode from the SciOps timeline as described in Listing 21.

5.4. Thermal Modelling

The aperiodic pointing profile of the multi-instrument EnVision mission brings uncertainties in the modelling of sizing cases that are highly dependent on the spacecraft attitude. In the proof-of-concept implementation of the PAS3 toolkit for EnVision, it was chosen to build thermal and power models which can extract full-mission attitude-dependent analysis. With the verified steady-state attitude model from section 5.3 added to the EnVision PAS3 configuration in section 5.2, the creation of these subsystem models can start.

This section will describe the thermal modelling efforts performed. For the EnVision mission, two thermal analyses were requested:

1. **VenSpec (& VenSAR) fluxes** — The EnVision project team requested an update on previous thermal analysis for the VenSpec instruments' accumulated fluxes within their FOV's over the mission lifetime as a requirement for the instrument teams. Previous analysis was done with a more analytical approach. A simulation-based analysis was desired.
2. **Thermo-optical degradation of MLI** — The spacecraft thermal department requested an analysis of the accumulated flux on the spacecraft panels over the mission lifetime. This analysis is to be used to verify

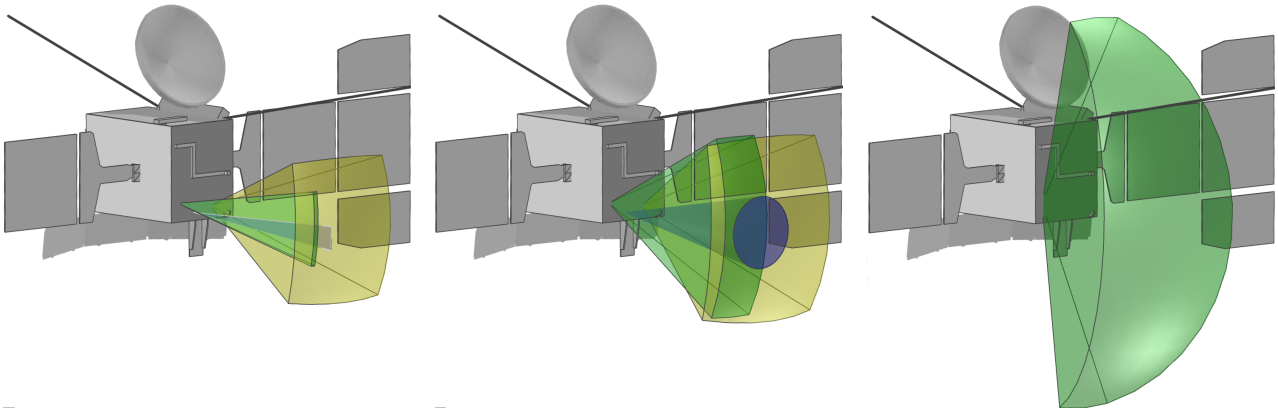
the current assumptions made on the expected thermo-optical degradation of the MLI blankets on the spacecraft bus's side panels.

Note that for both these cases, the desired output of the analyses are the accumulated Equivalent Sun Hours (ESH), which is defined as the equivalent number of hours a surface is exposed to 1365 W/m^2 of solar radiation [69]. This measure is commonly used in spacecraft thermal design to estimate the degradation of thermo-optical properties of surface materials (or optical instruments) over time due to solar exposure (both direct and albedo) [70].

Two different approaches were taken for both of these thermal analyses. For the VenSpec analysis, the established equations and assumptions from the previous analytical ESH analysis were implemented, but with a PAS3/VESPA–backed simulation. This is a lower-fidelity model, but offers a conservative approach to the ESH sizing, while keeping the simulation time manageable (this will become apparent in subsection 5.4.1). For the spacecraft panel flux analysis, a more complex numerical thermal model was built. A requirement on fidelity was not provided directly by the thermal department, giving the opportunity to implement a more complex, numerical model as a proof-of-concept of the PAS3 toolkit capabilities. Both models will be described in the following subsections.

5.4.1. VenSpec ESH Analysis

EnVision's VenSpec suite consists of three spectrometers (VenSpec–U, –H, and –M) each with a different FOV. The FOV's are rectangular cone-shaped with their axes aligned with the +X body axis of the spacecraft. These FOV's were already initialized following their design specifications stored in the EnVision SPICE kernels in EnVision's PAS3 configuration in Listing 16. They are also illustrated in Figure 5.11a. Additionally, each instrument has an Unobstructed Field of View (UFOV), which is a larger region representing the maximum possible view angles of the instrument without any physical obstructions. Besides, VenSpec-U also has a region defined by a Contamination Rejection Angle (CRA), which is an even larger FOV representing the area in which light contamination from stray light sources can enter the instrument.



(a) VenSpec FOV's **(b)** VenSpec UFOV's **(c)** VenSpec-U CRA

Figure 5.11: Current VenSpec FOV's, UFOV's and CRA rendered on the EnVision spacecraft model. Smallest to largest cones in **(a)**: VenSpec-H (blue), VenSpec-U (green), VenSpec-M (yellow).

These regions were added manually to the EnVision IK file. They could have been initialized as a PAS3 `CustomInstrument`, but as the FOV angles are not expected to change often, manually adding the UFOV and CRA descriptions inside the SPICE IK was preferred (keeping clarity in the `config.py` file). Their PAS3 initialization in the EnVision configuration file is shown in Listing 23:

```
# UFOV's:
VenSpec_M_UFOV = p3.SpiceInstrument(EnVision, "ENVISION_VENSPEC_M_UFOV")
VenSpec_H_UFOV = p3.SpiceInstrument(EnVision, "ENVISION_VENSPEC_H_UFOV")
VenSpec_U_UFOV = p3.SpiceInstrument(EnVision, "ENVISION_VENSPEC_U_UFOV")
# CRA:
VenSpec_U_CRA  = p3.SpiceInstrument(EnVision, "ENVISION_VENSPEC_U_CRA")
```

Listing 23: VenSpec UFOV's and CRA initialization.

As was mentioned before, the same equations for solar and albedo fluxes from earlier ESH analysis was

implemented. The maximum solar flux is calculated very straightforwardly in Equation 5.3, with I the solar constant at 1 AU (1365 W/m²), and r the current Sun–spacecraft distance in AU. Note that the incidence angle of the sun inside the FOV is not considered in this very conservative approach.

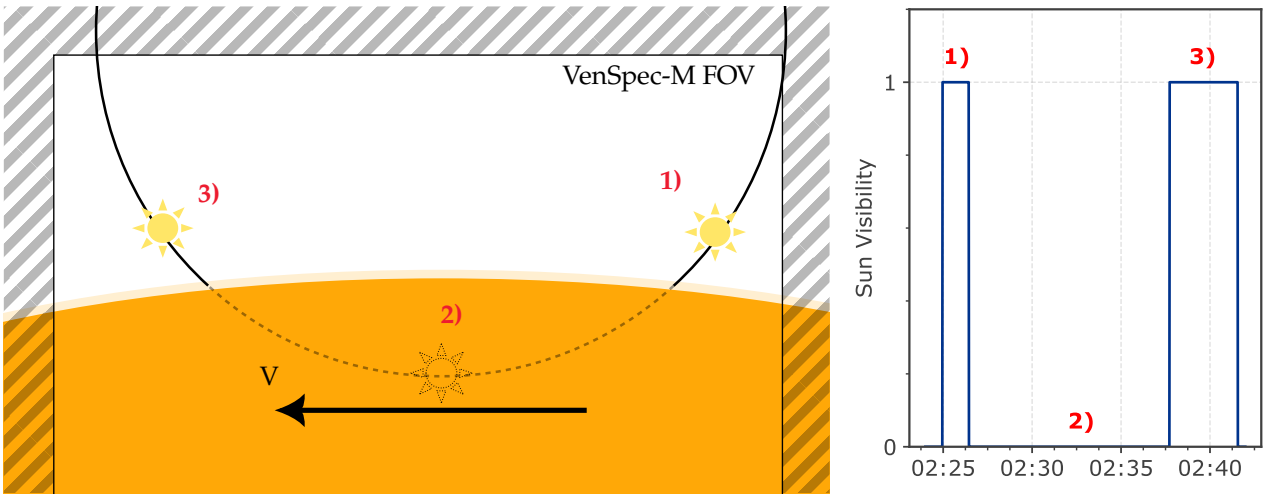
$$q_{sol} = I \left(\frac{1AU}{r} \right)^2 \tag{5.3}$$

Within the PAS3 *Spacecraft* object, this can be accessed as the property: `EnVision.solar_flux`. The simulation-wide solar constant I is configured in the *Simulation* object as seen in Listing 13. The solar flux equation is standardized enough to be implemented in the base *Spacecraft* class of the PAS3 toolkit without violating the mission-agnostic philosophy.

With all VenSpec FOV's initialized, they can be imported into the analysis script and a full-mission simulation can be run very straightforwardly as seen in Listing 24. At every time step, the solar ESH is calculated by normalizing the current solar flux w.r.t. the solar constant, multiplying it by the time step in hours. Then, for every VenSpec instrument, it is checked whether the instrument sees the Sun using the `sees_the_sun()` method of the *Instrument* class. If so, the solar ESH is stored.

```python
from config import sim, EnVision

# Store accumulated solar ESH per VenSpec instrument
solar_esh_dict = {name: [] for name in EnVision.instruments if name not in
 ↪  ["ENVISION_VENSAR", "ENVISION_SRS"]}

# Main simulation loop
DT = 60*2.5 # Time step of 2.5 minutes in seconds
while sim.active:
    sun_esh = EnVision.solar_flux / sim.solar_constant * DT/3600  # ESH
    for instrument in EnVision.instruments.values():
        if instrument.sees_the_sun(): # Add solar ESH
            solar_esh_dict[instrument.name].append(sun_esh)
        else:                         # No sun in FOV
            solar_esh_dict[instrument.name].append(0.0)

    sim.advance_time(DT)
```

Listing 24: Solar ESH simulation



(a) VenSpec-M FOV in "NRA_MAX" mode with occasional Sun impingement (not to scale).

(b) VenSpec-M PAS3 Sun visibility output.

Figure 5.12: Maximum SAR incidence angle in NRA orientation can cause sun impingement for large FOV's.

Remember that the spacecraft attitude is fully encapsulated by the `sim` object in the previous section. Below the surface, EnVision's attitude follows the VESPA operations, and this attitude (as well as the *Spacecraft* occultation calculations) directly determines if the VenSpec objects' Sun visibility functions return `True` or `False`. By default, the standard SAR pointing mode is configured as "NRA_MAX" (see Figure 5.8a). In this orientation, the likelihood of Sun impingement in the VenSpec FOV's by the sun during a VenSAR observation is maximized due to the FOV potentially exceeding the limb horizon. This is shown in Figure 5.12a. For albedo flux calculations, the worst case scenario is the "PRA_MIN" mode (see Figure 5.8c), as this orients the VenSpec FOV's more towards the planet, maximizing the albedo reflection. This will be covered below.

Modelling the albedo flux is a lot more ambiguous; there exist many ways to model it with different levels of fidelity. Therefore, no standard albedo model is included in PAS3. The user is expected to create the model using the PAS3 functionalities. The ESH analysis used a simplified albedo model, expressing the albedo flux as a function of the solar flux, a (constant) albedo coefficient of Venus, a view factor, and the elongation angle of the Sun w.r.t. Venus as seen from the spacecraft. This results in Equation 5.4, with ρ the albedo coefficient of Venus (0.76), q_{sol} the solar flux on Venus, $F_{12}(\lambda, \alpha_L)$ the view factor, and ϑ_s the elongation angle. They are also illustrated in Figure 5.13.

$$q_{alb}(\vartheta_s, \lambda, \alpha_L) = q_{sol} \rho F_{12}(\lambda, \alpha_L) cos(\vartheta_s) \tag{5.4}$$

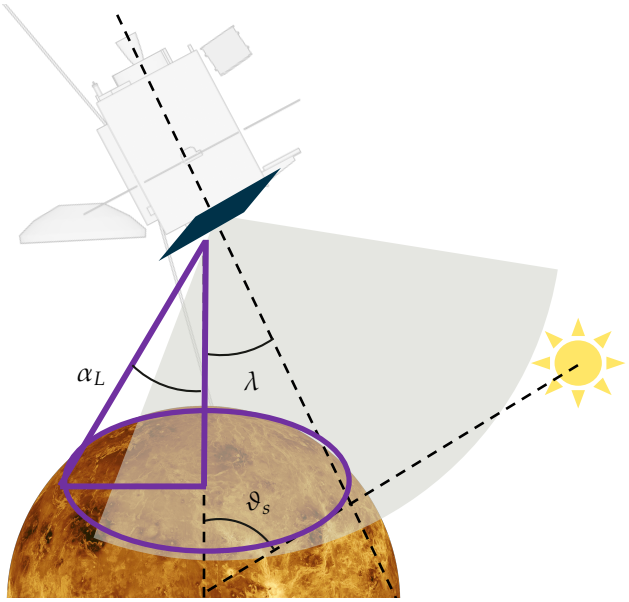


Figure 5.13: Geometry for albedo flux calculation, with λ the angle between nadir and boresight, ϑ_s the solar elongation angle, and α_L the limb half-angle.

The view factor was newly added, as this attitude-dependent factor could be calculated straightforwardly using the PAS3 simulation functionalities. An analytical model for a planar to spherical view factor from *ECSS-E-HB-31-01* was implemented to account for self-shielding of the visible cap of Venus at low altitudes (like that of EnVision's) [71, 72]. Note that this approach is still a simplification, as the FOV geometries are not planar, but conical. However, this approach is very conservative (partial shielding of a planar geometry is always less than that of the infinite conical geometry of the FOV) while ensuring no wide overestimations of albedo fluxes at low altitude.

$$F_{12} = B_0 + B_1 cos(\lambda) + B_2 cos^2(\lambda) + B_3 cos^3(\lambda) + B_4 cos^6(\lambda) \tag{5.5}$$

with the coefficients defined as:

$$B_0 = \frac{2}{7\pi}\left[\frac{577}{105} - 7cos(\alpha_L) + \frac{4}{3}cos^3(\alpha_L) - \frac{2}{5}cos^5(\alpha_L) + \frac{4}{7}cos^7(\alpha_L)\right] \tag{5.6}$$

$$B_1 = \frac{1}{2}sin^2(\alpha_L) \tag{5.7}$$

$$B_2 = \frac{8}{7\pi}\left[cos(\alpha_L) - 2cos^3(\alpha_L) + 4cos^5(\alpha_L) - 3cos^7(\alpha_L)\right] \tag{5.8}$$

$$B_3 = \frac{4}{7\pi}\left[-cos(\alpha_L) + \frac{40}{3}cos^3(\alpha_L) - \frac{91}{3}cos^5(\alpha_L) + 18cos^7(\alpha_L)\right] \tag{5.9}$$

$$B_4 = \frac{8}{35\pi}\left[5cos(\alpha_L) - 35cos^3(\alpha_L) + 63cos^5(\alpha_L) - 33cos^7(\alpha_L)\right] \tag{5.10}$$

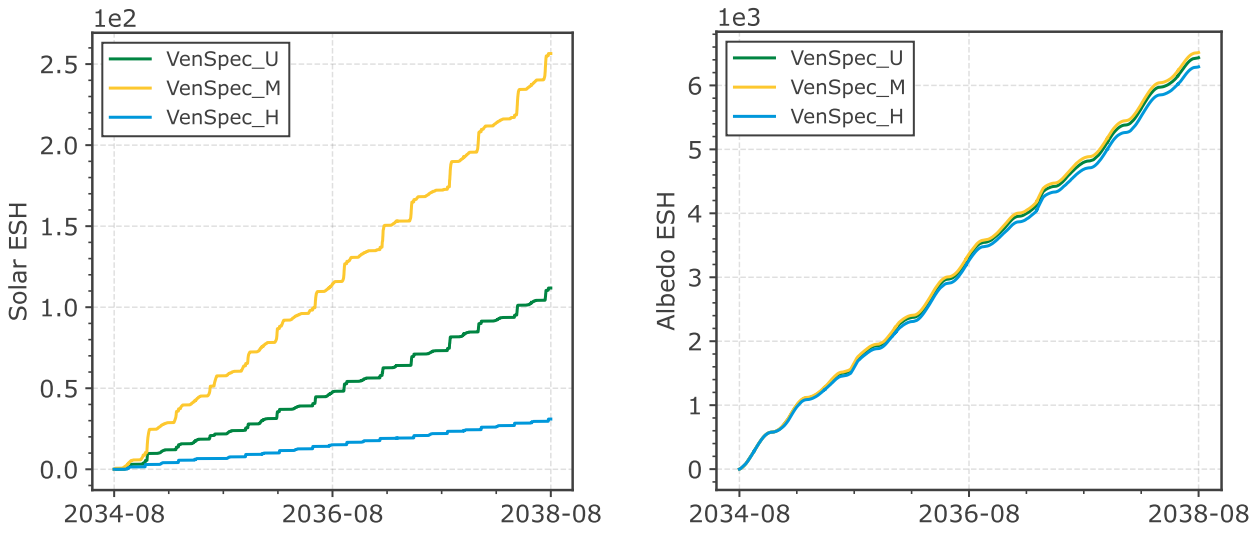
The VenSpec FOV's, as well as their UFOV and CRA representations can then be imported into the thermal model script, as well as the EnVision *Spacecraft* and *Simulation* objects. Built-in geometrical properties from the PAS3 *Spacecraft* class are used to fill the necessary parameters in Equation 5.4, like ϑ_s with `EnVision.sc_sun_elongation_angle`, λ calculated with accessing the instrument's boresight and the property `EnVision.nadir_direction_in_body_fixed_frame`, and α_L with `EnVision.get_limb_half_angle()`. The solar flux q_{sol} is accessed the same way as in the solar flux model, but only centred at Venus using `Venus.solar_flux`.

Equation 5.4 was turned into a function `albedo_flux(coeff)`, taking the albedo coefficient ρ as input. Inside, Equation 5.5–5.10 are implemented, taking their variables from the PAS3 actors. Unlike the solar flux case, the worst case albedo flux inside the VenSpec FOV's during SAR observations is not during "NRA_MAX" mode, but during "PRA_MIN" mode (see Figure 5.8c). Therefore, the simulation loop checks if the current pointing mode is a SAR observation, and if so, temporarily overwrites the attitude to "PRA_MIN" using the keyword argument functionality of the attitude update function from Listing 22. This ensures that the worst-case albedo fluxes are captured during SAR observations. This is shown in Listing 25.

```python
from config import sim, EnVision, SciOps
from albedo_model import albedo_flux

# Store accumulated solar ESH per VenSpec instrument
albedo_esh_dict = {name: [] for name in EnVision.instruments if name not in
↪ ["ENVISION_VENSAR", "ENVISION_SRS"]}

# Main simulation loop
DT = 60*2.5 # Time step of 2.5 minutes in seconds
while sim.active:
    # Overwrite attitude to PRA_MIN during SAR observations for worst-case albedo
    if SciOps.get_pointing(sim.et) == "SAR_NRA_MAX":
        sim.set_time(sim.et, pointing_id="SAR_PRA_MIN") # update the sim
    albedo_esh = albedo_flux(coeff=0.76) / sim.solar_constant * DT/3600 # ESH
    for instrument in EnVision.instruments.values():
        if instrument.sees_central_body(): # Add solar ESH
            albedo_esh_dict[instrument.name].append(albedo_esh)
        else:                              # No Venus in FOV
            albedo_esh_dict[instrument.name].append(0.0)

    sim.advance_time(DT)
```

Listing 25: Albedo ESH simulation

In this case, the albedo ESH is stored only if the instrument sees Venus using the `sees_central_body()` method of the *Instrument* class. With both solar and albedo ESH simulations implemented, a full-mission simulation can be run to extract the accumulated ESH values for all VenSpec instruments over the mission lifetime. The results for the VenSpec UFOV's accumulated solar and albedo ESH are shown in Figure 5.14.



(a) Solar ESH

(b) Albedo ESH

Figure 5.14: VenSpec UFOV solar **(a)** and albedo **(b)** ESH simulation results.

With the aim of writing a general requirement, the team requested the accumulated ESH as a function of the

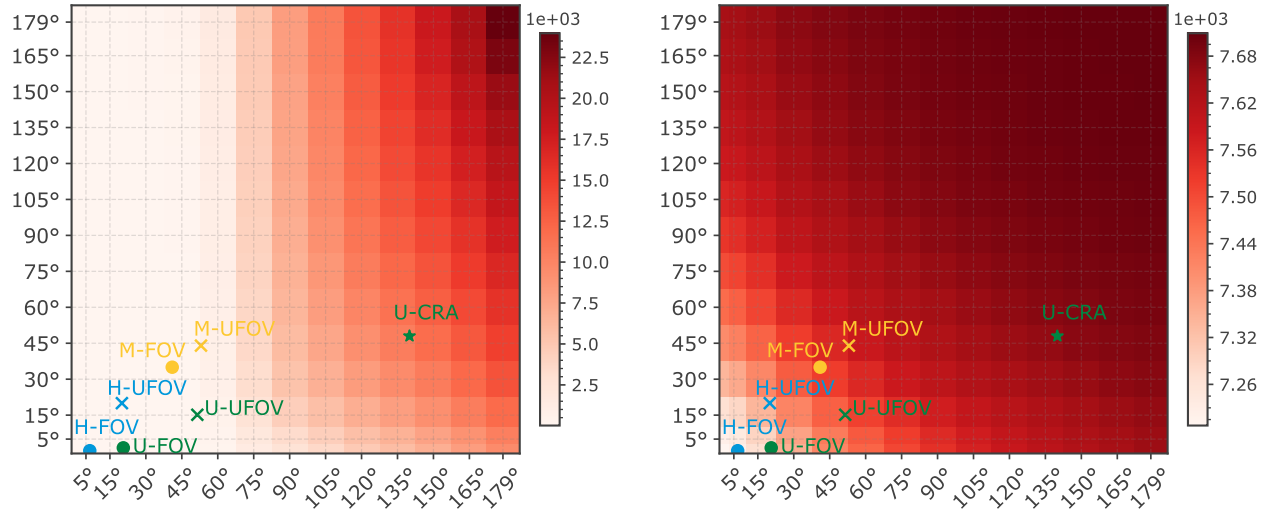
```python
import pas3 as p3
import itertools
from config.envision_config import EnVision

def rectangular_cone_vertices(ref, cross):
    """Returns unit vertices binding the rectangular FOV cone given referenc & cross angle."""
    ...

# Get combinations of reference and cross angles
angles = [1, 5, 15, 30, 45, 60, 75, 90, 105, 120, 135, 150, 165, 179]
angle_combinations = list(itertools.product(angles, repeat=2))

for reference_angle, cross_angle in angle_combinations:
    name = f"FOV_{reference_angle}_{cross_angle}"
    vertices = rectangular_cone_vertices(reference_angle, cross_angle)

    p3.CustomInstrument(EnVision, name, vertices, "RECTANGLE")
```

Listing 26: Creation of 196 custom instruments with varying rectangular cone FOV angles.

FOV angles. The rectangular FOV cone shape can be defined by two angles: the reference angle and the cross angle. In SPICE terminology, the reference angle is the half-angle around a chosen reference axis, with the cross angle perpendicular to it. The rectangular cone is then defined in the direction of a given boresight. In our case, the reference axis is the +Y body axis, the cross axis the +Z body axis (with the boresight pointing in the +X direction). Henceforth, when referring to the reference and cross angles, the full angle is meant (so a reference angle of 30° means a half-angle of 15° around the +Y axis in the +X direction).

To extract the relationship between reference and cross angles and the accumulated ESH, custom instruments were created with a grid combination of reference and cross angles varying from 1° to 179°. This results in 196 virtual VenSpec instruments attached to the EnVision *Spacecraft* object. They are initialized as per Listing 26.

The same simulations as before can be run with these custom instruments, only storing the total accumulated ESH at the end of the simulation. This results in heatmaps of accumulated ESH as a function of reference and cross angles, shown in Figure 5.15.



(a) Solar ESH

(b) Albedo ESH

Figure 5.15: accumulated VenSpec solar **(a)** and albedo **(b)** ESH as a function of FOV reference (x-axis) and cross (y-axis) angles for the full EnVision mission lifetime.

Using a polynomial least squares regression of degree 3 with the obtained summed total results, a general ESH expression can be generated as a function of reference (ϑ) and cross angle (ϕ). The resulting Equation 5.11 has a coefficient of determination R^2 of 0.993 and a root mean square error (RMSE) of 566 ESH, which is acceptable

given the total ESH values in the order of 10^4 ESH.

$$ESH_{total}(\vartheta,\phi) \approx a_0 + a_1\phi + a_2\phi^2 + a_3\phi^3 + a_4\vartheta + a_5\vartheta\phi + a_6\vartheta\phi^2 + a_7\vartheta^2 + a_8\vartheta^2\phi + a_9\vartheta^3 \tag{5.11}$$

With coefficients:

$$\begin{array}{llllll} a_0 = & 7.6\times10^{3} & a_5 = & 3.8\times10^{-1} & a_1 = & 2.2\times10^{-1} \quad a_6 = \quad -1.3\times10^{-3} \\ a_2 = & -3.4\times10^{-1} & a_7 = & 1.8\times10^{0} & a_3 = & 1.35\times10^{-3} \quad a_8 = \quad 1.6\times10^{-3} \\ a_4 = & -8.4\times10^{-1} & a_9 = & 6.0\times10^{-3} & & \end{array}$$

This allows the EnVision project team to provide the instrument manufacturers with a design constraint depending on their FOV specifications. This section showcased nicely how existing analyses can be implemented in a simulation-based approach once a PAS3 configuration of the mission is built. The modular approach of the configuration file, allows for quick reiteration of models like this ESH analysis, updating the acquired sizing results. A future change in the operational timeline (VESPA Data) or spacecraft attitude model (future SPICE CK files) would be automatically reflected in the ESH results without touching the analysis script itself.

5.4.2. VenSAR ESH Analysis

As an addition to the VenSpec ESH analysis, the project team requested a similar analysis on the VenSAR reflectarray as well. Due to uncertainties in EnVision's SAR acquisition, a clear ESH estimation is required as a relationship of the array's tilt angle. This subsection will be brief, as the model is very similar to the VenSpec ESH analysis.

The reflect-array is a panel with its normal vector at a tilt angle ϑ_{VenSAR} w.r.t. the +X body axis. Running a similar simulation as in subsection 5.4.1 with only a *half-space* VenSar panel geometry would ignore the shielding effect of the Spacecraft bus on the panel. As covered in subsection 4.4.5, more complex geometrical regions can be created as PAS3 objects by using the `AssembledGeometry` / `AssembledInstrument` classes. This way, illumination conditions on more complex geometries can account for self-shielding effects of other physical parts of the spacecraft. Therefore, an `AssembledInstrument` representing the VenSAR panel was created using a half-space for the panel itself as the reference geometry (MAIN geometry) and the -Z panel added as AND geometry. This basically abstracts illumination on the array as only being possible if the illumination is present on the panel AND on the -Z spacecraft panel. This is illustrated in Figure 5.16. The `AssembledInstrument` is created in Figure 27.

```python
import pas3 as p3
from math import cos, sin

# VenSAR panel normal vector:
boresight = [cos(PANEL_TILT), 0, -sin(PANEL_TILT)]

Panel = p3.Geometry(
    spacecraft=EnVision,
    name="REFLECT_ARRAY",
    vertices_in_body_fixed_frame=[boresight],
    shape="HALF_SPACE"
)

AssembledSAR = p3.AssembledInstrument(
    MAIN_geometry=Panel,    # Assembled reference boresight is from MAIN geometry
    AND_geometries=[NZ],    # Add -Z panel to account for self-shielding
)

```

Listing 27: VenSAR assembled instrument creation in `config.py`.

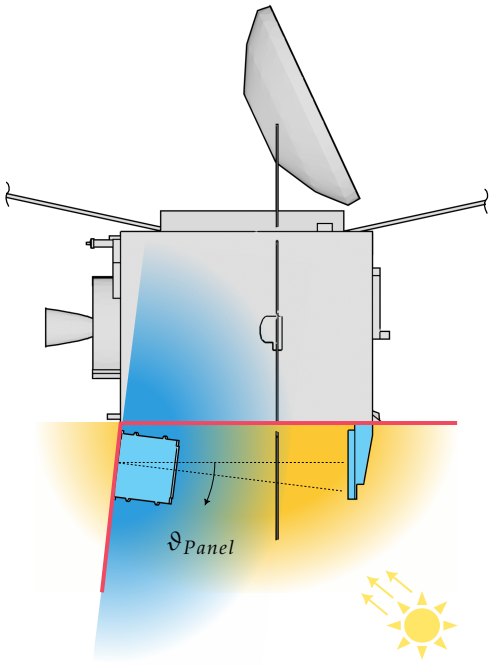


Figure 5.16: Two half-space geometries combined for VenSAR illumination conditions.

Using the *AssembledInstrument* class, more geometries (like the side panels of the reflectarray, visible in Figure 5.3) could be added to the VenSAR panel visibility region (as OR or NOT geometries). In this case,

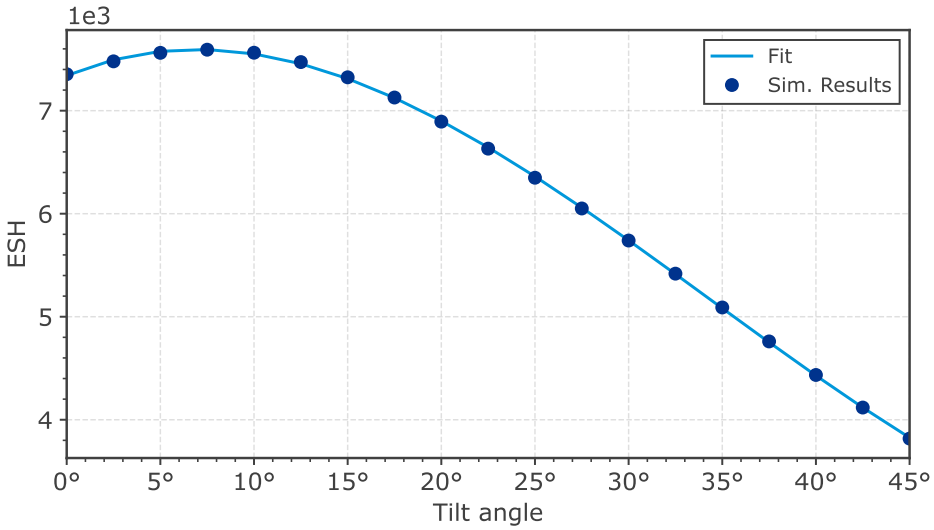


Figure 5.17: Accumulated VenSAR ESH as a function of reflectarray tilt angle over the full EnVision mission lifetime.

they were not included to keep the model simple and conservative. In the same fashion as the VenSpec ESH analysis, a full-mission simulation was run, rotating the reference panel normal vector using the *Geometry* method `rotate(<axis>, <angle>)` to rotate the panel by a new tilt angle and updating the attitude accordingly. The accumulated ESH on each tilt angle was stored. The results are shown in Figure 5.17.

Fitting the results with a polynomial regression of the third degree results in the following general ESH expression as a function of panel tilt angle ϑ_{Panel}:

$$ESH(\vartheta_{Panel}) = \left(5.9 \times 10^{-2}\right) \vartheta_{Panel}^{3} - (6.1)\, \vartheta_{Panel}^{2} + (7.6 \times 10)\, \vartheta_{Panel} + 7.34 \times 10^{3} \tag{5.12}$$

The assembled instrument, while representing a more complex geometry, is a subclass of the simple geometries and thus, when initialized correctly, uses the exact same methods and attributes as those simpler objects. The straightforward object-oriented initialization of geometries that abstract regions of interest for the spacecraft instruments, allowed for quick reuse of previously built models with more complex geometries without changing the underlying model structure. It also allowed for a fast setup of analysis in the context of a VenSAR-related CDF study, studying different SAR allocations on the spacecraft. The effect of different SAR orientations was quickly studied by reusing the same thermal models as before.

5.4.3. MLI ESH Analysis

For the second thermal analysis, the spacecraft thermal department requested an estimation of the accumulated fluxes on the spacecraft panels over the mission lifetime. The accumulated ESH is used to have a better idea of the MLI's thermo-optical degradation over time in the harsh thermal environment around Venus. Unlike the previous VenSpec and VenSAR thermal analyses, a more complex numerical thermal model was built to serve as a proof of concept of how PAS3 can help build higher fidelity models as well.

In the previous analyses, a straightforward analytical approach was taken. The solar fluxes will remain pretty similar (the directional light source of the Sun requires no further complexity), the difference lies within the modelling assumptions made for the Venus albedo. While the VenSpec ESH model did implement a finite polynomial (Equation 5.5) to form a view factor attempting to account for self-shielding, it fails to account for more complex geometrical effects like: variability in albedo reflection on the visible cap of Venus, and the lack of Sun reflections behind the terminator line. This approach sufficed as a conservative analysis and kept the computational load (for the 196 custom instruments) manageable. Creating two external flux models, a straightforward analytical and a more complex numerical model, will showcase the intended versatility of PAS3 and its agnostic nature as toolkit for model creation.

To calculate reflected albedo flux on a planar surface of the spacecraft at arbitrary orientation and position, a solar reflection model from a NASA technical note (D-1842) was consulted [73]. Although this report was published in 1963, it is still referenced in modern thermal design literature [74]. The model divides the visible cap of Venus into a polar grid of small surface elements, each reflecting a portion of the incoming solar flux towards the spacecraft panel.

The reflected albedo flux received by the panel can be calculated by integrating the reflected solar flux from infinitesimal surface elements ($d\Sigma$) over the visible cap of Venus at the current spacecraft position. The visible cap is illustrated in orange Figure 5.18, right below the spacecraft panel with its boresight vector $\hat{b}$. The integral is shown in Equation 5.13, with assumed constants: q_{sol} the solar flux at Venus, ρ a constant albedo coefficient, and R Venus's radius. ϑ and ϕ the local colatitude and longitude of the surface elements in an arbitrary reference frame aligned with the spacecraft position (ijk in Figure 5.18).

$$q_{alb} = q_{sol}\rho \int_0^{\vartheta_{max}} \int_0^{2\pi} \frac{cos\left(\eta\right) cos\left(\xi\right) cos\left(\beta\right)}{\pi S^2} R^2 sin\left(\vartheta\right) d\phi d\vartheta \tag{5.13}$$

Three cosines are included; they account for reduction in solar reflection / absorption at different incidence angles. The angle ξ is the angle between the local surface normal ($\hat{n}$) of the infinitesimal surface element and the direction towards the spacecraft panel, β the angle between $\hat{n}$ and the direction vector towards the Sun ($\vec{v}_{sun}$), and η the angle between the panel boresight vector $\hat{b}$ and the direction vector towards the surface element.

Finally, S is the distance between the surface element and the spacecraft panel. It follows the unit vector $\hat{v}_S$ that points towards the surface element from the panel. ϑ_{max} is the half-angle of the visible cap cone centred at Venus's centre; it is therefore also the upper limit of the outer integral in Equation 5.13.

In order to numerically solve the integral, the visible cap is discretized into a polar grid of $N_\vartheta \times N_\phi$ surface elements, with uniform angular segments:

$$\Delta\vartheta = \vartheta_{max}/N_\vartheta \tag{5.14}$$
$$\Delta\phi = 2\pi/N_\phi \tag{5.15}$$

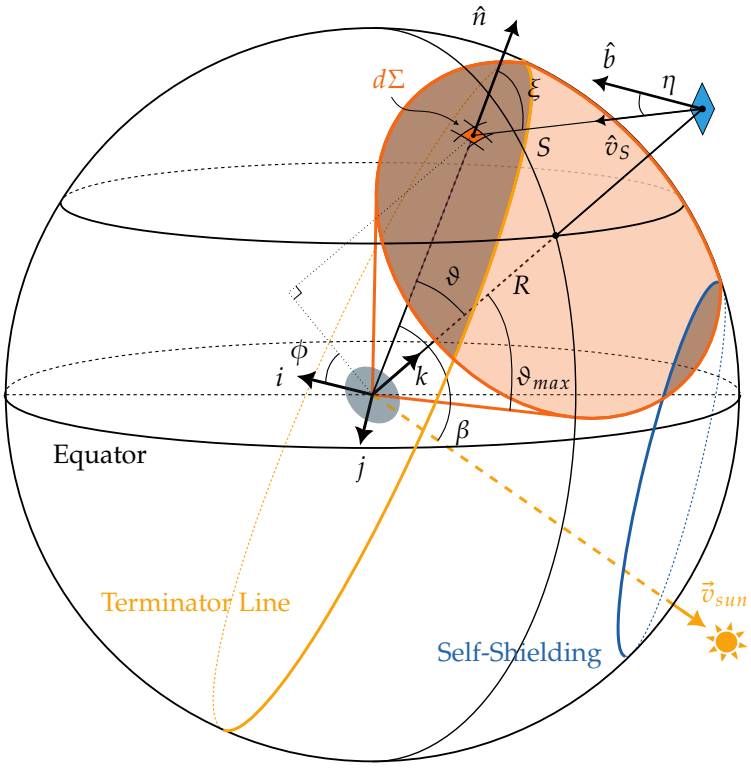


Figure 5.18: Geometry of the albedo flux model with variables for integration.

The integral can then be approximated as a double sum over all surface elements:

$$q_{alb} \approx q_{sol}\rho \sum_{i=1}^{N_\vartheta} \sum_{j=1}^{N_\phi} \frac{cos\left(\eta_{i,j}\right) cos\left(\xi_{i,j}\right) cos\left(\beta_{i,j}\right)}{\pi S_{i,j}^2} \Delta A_i \tag{5.16}$$

with the area of every surface element defined as:

$$\Delta A_i = R^2 sin(\vartheta_i)\Delta\phi\Delta\vartheta \tag{5.17}$$

The area of the surface element at grid position (i, j). The angles $\eta_{i,j}$, $\xi_{i,j}$, and $\beta_{i,j}$, as well as the distance $S_{i,j}$ are calculated for every surface element. Additionally, only surface elements that are illuminated by the Sun and visible from the spacecraft panel are considered in the sum. Depending on the direction of the Sun and the panel orientation (i.e. spacecraft attitude), some integrands should be zeroed out (otherwise the cosines might become negative). The Sun shadowing on Venus behind the terminator line and self-shielding of the panel is also illustrated in Figure 5.18 by the darker areas on the visible cap. Recognizing that all cosines can be expressed as dot products between the unit vectors involved, the following conditions apply:

$$cos(\beta_{i,j}) = \hat{v}_{sun} \cdot \hat{n} > 0 \quad \text{(surface element illuminated by Sun)} \tag{5.18}$$
$$cos(\eta_{i,j}) = \hat{v}_S \cdot \hat{b} > 0 \quad \text{(panel illuminated by surface element)} \tag{5.19}$$

It should be noted that this model also applies for IR fluxes, replacing the solar flux q_{sol} and albedo coefficient ρ with the IR flux at Venus $q_{IR} = \sigma T^4$, with σ the Stefan-Boltzmann constant and T Venus's blackbody temperature (226.6 K). In this case, the Sun reflection factor ($cos\,(\beta)$) is omitted [74].

Now, how does PAS3 help in implementing this geometrically complex model? We see that the essential equations are already set up, they can be easily integrated within a custom thermal model of the EnVision panels. The non-trivial part is accessing all necessary geometrical and dynamic variables. The approach of integrating PAS3 objects in the custom thermal model will be covered below.

The first step is to create a custom spacecraft *Face* class inside the `thermal_model.py` sub-script. For pure ESH analysis, only the PAS3 *Geometry* of the outward-pointing half-space is strictly necessary. In the custom *Face* class initialization shown in Listing 28, additional properties of the panel are also included, like area, mass, specific heat, absorptivity, emissivity and starting temperature. These properties will be useful when extending the model to a full thermal balance calculation.

```
import pas3 as p3

class Face:
    def __init__(self, geometry, area, mass, cp, absorptivity, emissivity, starting_temp):
        self.geometry     = geometry: p3.Geometry
        self.spacecraft   = geometry.spacecraft: p3.Spacecraft
        self.central_body = geometry.spacecraft.central_body: p3.CelestialBody
        ...
```

Listing 28: Custom *Face* class for EnVision bus faces in the thermal model sub-script using the PAS3 *Geometry* class as building block.

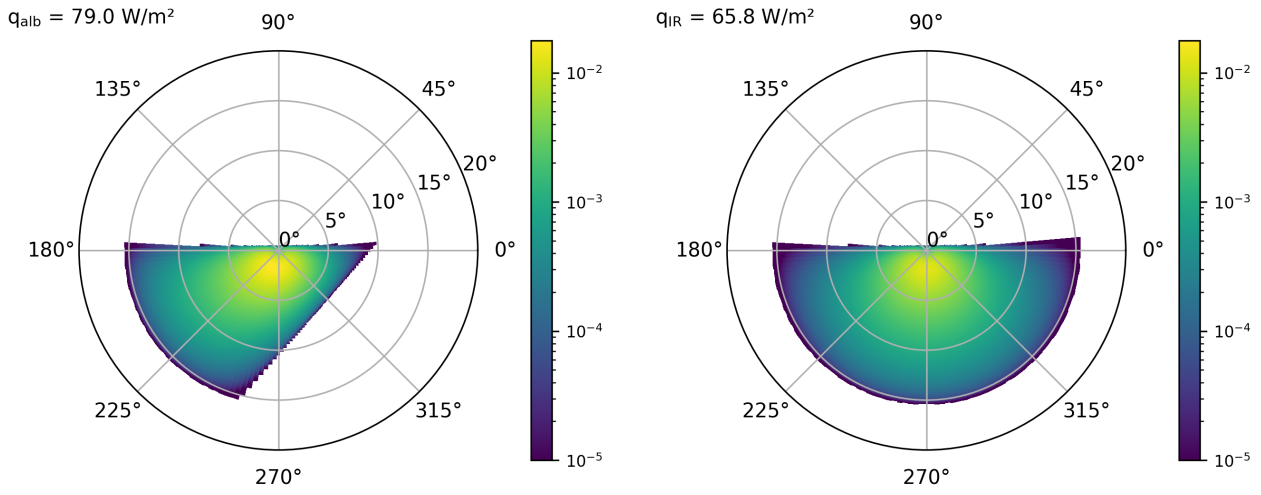
Within this custom *Face* class, a method can be created which will compute the numerical equation in Equation 5.16. First and foremost, the PAS3 *Geometry* method `sees_cetral_body()` is called to check if Venus is within the half-space geometry of the panel. If not, the albedo flux is zero and the method returns early (omitting unnecessary calculations).

If Venus is visible, the polar grid of surface elements on the visible cap is created. The uniform angular segments $\Delta\vartheta$ and $\Delta\phi$ are defined by taking the number of angular divisions (N_ϑ and N_ϕ) as input parameters, and accessing the maximum colatitude ϑ_{max} from the PAS3 *Spacecraft* method: `central_body_limb_half_angle()`. Using `numpy.meshgrid`, the polar grid of colatitude and longitude values is created. Converting from spherical to Cartesian coordinates gives the unit normal vectors $\hat{n}$ of all surface elements in the mesh grid. For the grid definition, PAS3 only helps in accessing the maximum colatitude ϑ_{max}; dividing the grid and accessing the surface element vectors (`v_surface_element` below) and acquiring the element area (ΔA in Equation 5.17) is done with standard `numpy` operations. The remaining variables from Equation 5.16, 5.18 and 5.19 were accessed using the PAS3 toolkit as per Table 5.3:

Table 5.3: PAS3 variable access to complete the numerical albedo flux model.

Variable	PAS3 access method in the `albedo_flux()` *Panel* class function
ϑ_{max}:	`pi/2 - Panel.spacecraft.central_body_limb_half_angle()`
$\hat{v}_S$:	`pas3.utils.normalize(v_surface_element - (Panel.spacecraft.position * 1000))`
S:	(norm of $\hat{v}_S$)
$\hat{b}$:	`Panel.spacecraft.from_body_fixed_frame(Panel.geometry.boresight)`
$\hat{v}_{sun}$:	`self.central_body.sun_direction`

To get an idea of the numerical integration applied, a snapshot was taken of the visible cap mesh and the resulting integrands for every surface element at 2034/09/03 17:58:20 (UTC) during an EnVision Earth communication attitude for the +X panel. Both Albedo and IR integrands were computed and are shown in Figure 5.19. Both show the upper half without any integrand values due to self-shielding of the panel. The albedo integrands in Figure 5.19a clearly show an additional shadowed region behind the terminator line, where no solar illumination is present on Venus's surface elements to be reflected to the panel. This snapshot, however, only gives a qualitative idea of the integration process. A full verification campaign of the model was performed, comparing results with literature values in subsection 5.4.4.



(a) Albedo integrands

(b) IR integrands

Figure 5.19: Albedo **(a)** & IR **(b)** model integrands for EnVision's +X panel distributed over the visible cap mesh at 2034/09/03 17:58:20 (UTC) in Earth communication attitude.

Inside the custom *Face* class, the solar flux is modelled using PAS3 methods as well. First, the sun visibility is checked using the `sees_the_sun()` method, then received solar flux is calculated using the solar irradiance at EnVision's position and the panel incidence angle (`angle_of_incidence_in_rad()` method). The resulting method is shown in Listing 29.

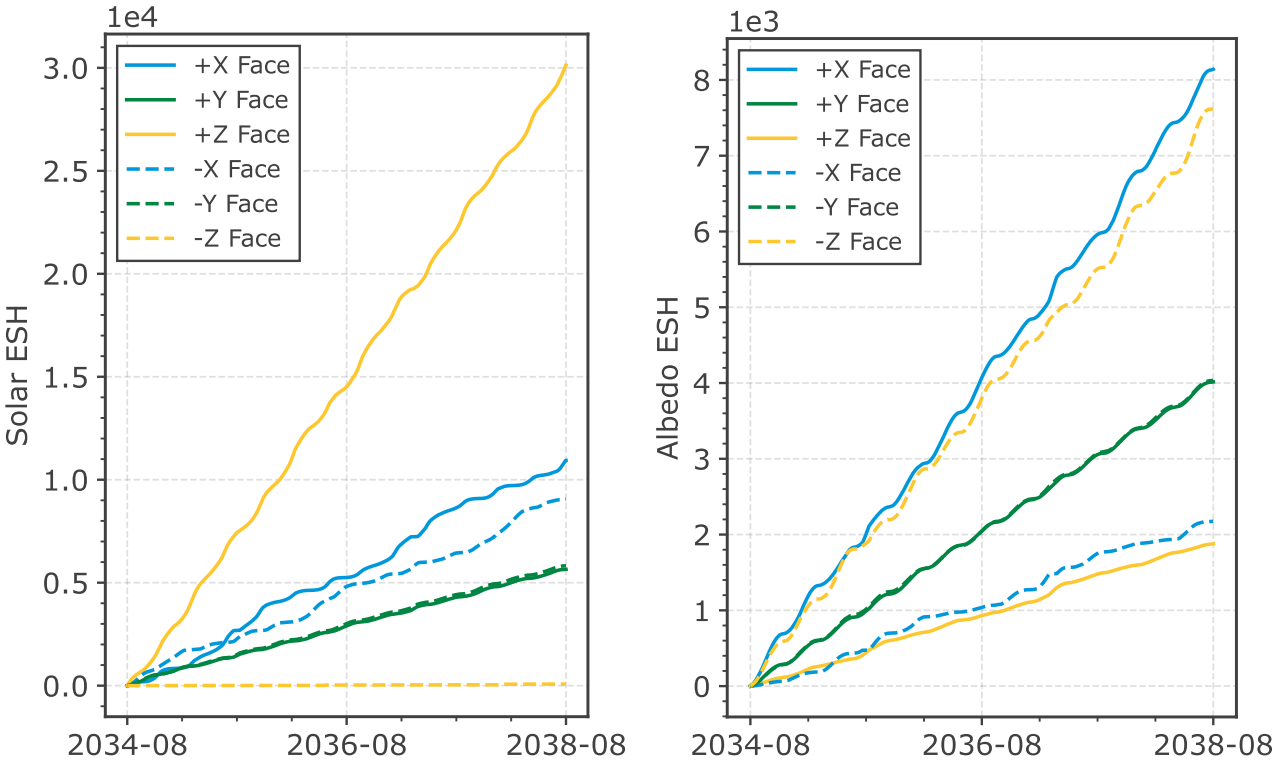
```
def solar_flux(self): # Watt/m²
    if not self.geometry.sees_the_sun():
        return 0.0
    else:
        incidence = self.geometry.angle_of_incidence_in_rad(
            -self.spacecraft.get_sun_direction_vector_in_body_fixed_frame()
        )
        solar_flux = self.spacecraft.solar_flux
        return solar_flux * max(0, np.cos(incidence))
```

Listing 29: Custom *Face* class method to calculate solar flux on the panel using PAS3 methods.

Eventually, the goal of this analysis was to get a full-mission accumulated ESH on every spacecraft panel. Therefore, a PAS3 simulation loop was performed, similar to the ones in subsection 5.4.1 and 5.4.2. 6 *Face* objects were created for every main bus panel, each with their own half-space *Geometry* from the EnVision PAS3 configuration (`PX, NX, PY, ...` in Listing 18). Inside the simulation loop, every face's solar and albedo flux is calculated and converted to an ESH value, automatically updating the spacecraft attitude and position with every time step.

This analysis resulted in the following ESH accumulation plots in Figure 5.20. These plots result from a full-mission simulation with 5-minute time steps, using a polar grid of 300x180 surface elements for the albedo flux integration. The plots show that the cold face (-Z) nicely avoids direct solar illumination due to the cold-constraint handling in subsection 5.3.2; however, it still receives a significant amount of albedo flux over the mission lifetime. This is to be expected, as the simulation ran with default SAR orientation of "NRA_MAX", which exposes the -Z panel towards Venus during all standard SAR observations. The +Z face receives the highest ESH of all panels, as in the attitude model, it is often the face which is deliberately pointed towards the Sun to keep the cold face shadowed.

While the presented approach focuses mostly on implementing the albedo & IR flux model as described in literature, the mission-level dependencies (such as time management, orbital ephemerides, spacecraft attitude, and reference frame transformations) are handled by the PAS3 toolkit. This abstraction allows variables that depend strongly on the simulation state (see Table 5.3) to be accessed directly, without the need for spacecraft state modelling inside the thermal script. Consequently, the effort required to implement a given model is determined entirely by the model's own intrinsic complexity rather than by the surrounding



(a) Solar ESH **(b)** Albedo ESH

Figure 5.20: Accumulated ESH on EnVision bus panels over the full mission lifetime.

simulation framework. This way, PAS3 removes the underlying barriers of mission simulation (by offering an object-oriented configuration) and enables the system engineer to focus exclusively on implementation of the model itself.

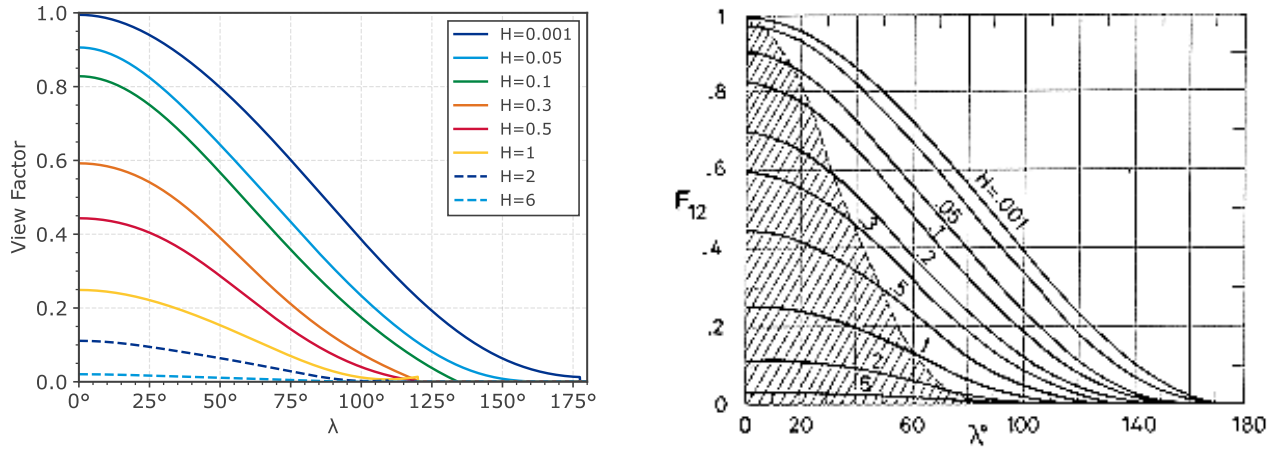
5.4.4. Thermal Models Verification

This subsection will describe the verification campaign that was performed to verify the thermal models presented in subsection 5.4.1, 5.4.2 and 5.4.3. Two types of models were implemented for two different use cases. The first two models were straightforward analytical models to get a quick estimation of accumulated fluxes on the instruments, while the last model was a more complex numerical model. Both will be verified separately below.

The first thermal flux model was taken from an established analysis document, the only addition being the implementation of a geometric view factor F_{12}. As the view factor implementation is the only new addition to the model, a verification run was performed to check if the implemented view factor behaves as expected. This is done with the knowledge that the used *Spacecraft* and *Geometry* FOV visibility methods have been verified extensively in previous chapter (section 4.5), as well as the underlying attitude model (subsection 5.3.3). The ECSS thermal handbook, where Equation 5.5 is taken from, provides reference plots of the view factor as a function of the angle between nadir and the panel boresight vector λ (see Figure 5.13) for different non-dimensional altitudes (H), which is the altitude divided by the central body radius [71]. As errors can be made quite easily in the expressions for the coefficients in Equation 5.6–5.10, resulting view factors from the implemented model were compared with the provided plots. The results are shown in Figure 5.21. They match closely, validating the implementation of the view factor in the VenSpec ESH model.

The second thermal flux model, used for the EnVision panel ESH analysis, was newly implemented based on literature [73, 74]. Therefore, a more complete verification campaign was performed. The NASA technical note included the model to calculate albedo fluxes incident on arbitrarily oriented spinning plates around the Earth. It concludes with numerous plots of albedo fluxes as a function of the actual altitude of the spacecraft and the same λ angle as above. Every plot is assigned to a different solar elongation angle ϑ_s (see Figure 5.13), which is the angle between the Sun direction vector and the spacecraft position vector.

In order to produce the same plots with the custom *Face* object albedo function from subsection 5.4.3, Faces were initialized in the correct rotation around the nadir vector (on the *Spacecraft* vector) with Earth as *CelestialBody*. Then, albedo fluxes were accessed for different relative Sun directions. For the case $\vartheta_s = 0$ (i.e. the spacecraft



(a) View factor verification run.

(b) ECSS view factor reference [71]

Figure 5.21: Verification of the implemented view factor in the VenSpec ESH model against ECSS reference plot.

is between the Earth and the Sun), the process is quite straightforward; the albedo is computed at different altitudes, resulting in the comparative plots in Figure C.1 in Appendix C.

The verification for other elongation angles required an additional step. The shadowing symmetry on the visible cap at $\vartheta_s = 0°$ dos not apply at different sun orientations. As the reference paper only considers results for arbitrarily oriented spinning plates only, the orientation of the custom *Face* object had to be rotated around the nadir vector (i.e. changing the direction of the boresight vector $\hat{b}$) for every altitude. Averaging the albedo fluxes obtained at discretized steps of a full 360° rotation, resulted in the comparative plots shown in Figure C.4 for $\vartheta_s = 90°$. The results match closely again, verifying that not only self-shielding effects, but also the Sun shadowing on Venus behind the terminator line are implemented accurately.

All verification plots (from $\vartheta_s = 0°$ to 90°) are shown in Appendix C. This concludes the verification campaign of the Create appendix thermal flux models. Two distinct thermal models were developed to address different use cases within the project team. The results demonstrate that PAS3 is not only capable of implementing both simple and complex models, but also of producing accurate and verifiable outcomes.

5.5. Power Model

After developing the thermal models for EnVision's side panels and instruments, the mission performance team requested to perform a sizing analysis on the spacecraft power system. As seen in section 5.3, the currently used SciOps timeline of the mission performance team plans observations in *arcs* of different types (e.g. SAR, VenSpec science, etc.); some of which, require a very high power consumption (e.g. high resolution SAR imaging). EnVision's current solar arrays are oversized and assumed to handle worst-case power consumption scenarios. However, the mission performance team desires an estimation of the maximum possible generated power during every arc to compare the power margins in place when accounting for the power consumptions of the observations.

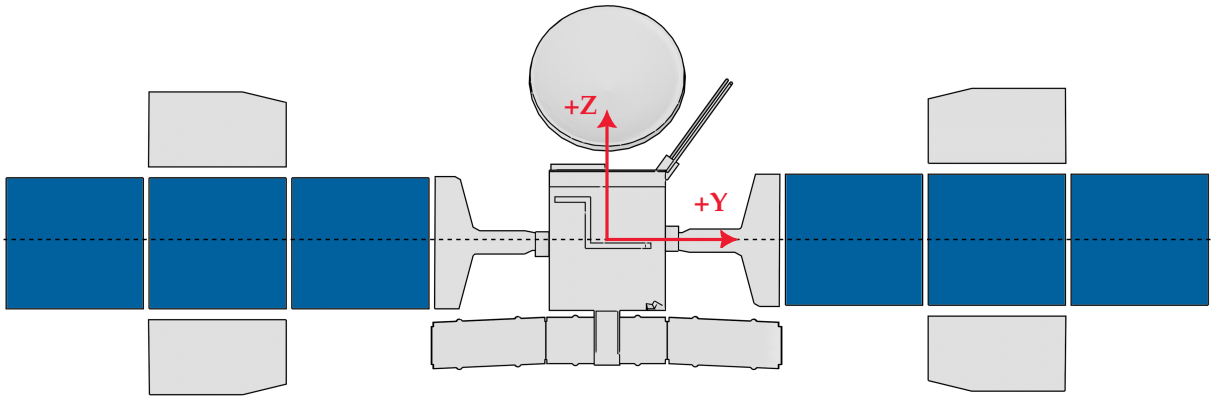


Figure 5.22: EnVision's Solar Array Assembly (SAA).

There is a high interest in the following effects on the power generation capabilities of the spacecraft:

- **Observation attitude** effect on solar array pointing.
- **Solar panel steering** during observations, being in a sun-tracking or fixed state.
- **Partial shadowing** of the solar arrays by the spacecraft body in certain attitudes / Sun directions.

All of these effects are highly attitude (thus science operations) dependent, but not directly modelled in the current mission performance team tool chain. This prompted an excellent case study to employ PAS3 in making a power generation model for EnVision. The power model implementation will be described in the following subsections. First, a steerable solar panel, and its steering is modelled in subsection 5.5.1. Then, methods determining partial shadowing of the spacecraft body on the steerable arrays is described in subsection 5.5.2 and results are presented. Finally, a verification campaign is outlined in subsection 5.5.4.

5.5.1. Panel Steering Model

EnVision's solar arrays are steerable around the Y-axis of the spacecraft body. They rotate symmetrically, meaning that both arrays rotate with the same angle. The Solar Array Assembly (SAA) is illustrated in Figure 5.22, with the solar cells coloured in dark blue.

From the previous sections, it was already showcased that PAS3 can initialize panel geometries which will automatically follow the modelled spacecraft attitude (be it custom, or CK-based attitude), and change incidence angle calculation results accordingly. However, due to the steering ability of the arrays, an additional degree of freedom is introduced, which should be modelled. Similar to the ESH analysis on EnVision's side panels in subsection 5.4.3, a custom *SteerableSolarArray* object class was set up:

```
class SteerableSolarArray:
    def __init__(self,
                 panel_geometry: p3.Geometry,   # PAS3 half-space Geometry of the solar array
                area: float,                    # Solar array area [m²]
                fill_factor: float,             # Solar cell fill factor [-]
                efficiency: float | Callable,   # Cell efficiency [-] (constant or function)
                rotation_axis: np.ndarray,      # Rotation axis in body-fixed frame [x,y,z]
                shadowing_function: Callable ): # Function to calculate shadowing % [-]
    self.geometry   = panel_geometry
    self.spacecraft = panel_geometry.spacecraft
    ...
```

Listing 30: Custom *SteerableSolarArray* class initialization, taking the PAS3 *Geometry* blueprint as main building block.

The power function in Equation 5.20 shall be used to extract power generation of one solar array. Note that this is the maximum possible power, assuming perfect power conditioning at the cell's Maximum Power Point (MPP). This assumption was discussed and agreed upon with the power system engineer.

$$P_{MPP} = I\left(A_{array}FF\right)\eta_{cell}(...)\cos(\vartheta_i)(1 - S_{sh}) \tag{5.20}$$

Input parameters for power calculations are provided in the *SteerableSolarArray* class initialization: the solar array area (A_{array}) and fill factor (FF) to get total cell area, the cell efficiency (η_{cell}), either constant or as a function

```
def sun_hits(self):
    """Returns True if the solar array is illuminated by the sun."""
    # Sun modeled as disk, self.geometry.sees_the_sun() might be True even if AOI is < 0°
    return (not self.spacecraft.eclipsed and (self.solar_aoi <= pi/2))

def solar_aoi(self):
    """Returns the solar angle of incidence (AOI) on the solar array in radians."""
    return self.geometry.angle_of_incidence_in_rad(
        -self.spacecraft.get_sun_direction_vector_in_body_fixed_frame() )
```

Listing 31: Solar Array Sun incidence methods using PAS3 *Spacecraft* and *Geometry* objects.

of temperature from a thermal model, or lookup table, the rotation axis of the arrays in body-fixed frame, and (S_{sh}) a shadowing function to calculate partial shadowing percentage (to be described in subsection 5.5.2). The class uses the PAS3 *Geometry* and associated *Spacecraft* objects to access relevant simulation state parameters. For example, to check if the panel is illuminated by the sun and to access the solar incidence angle (ϑ_i), the following PAS3-object-based properties were created in the class in Listing 31.

Note that the spacecraft attitude and position (i.e. eclipses) are fully encapsulated in the *Spacecraft* and *Geometry* objects. As long as the dynamic *Simulation* object is updated correctly, the associated objects in the static power model will automatically reflect the current simulation state.

But as mentioned, the panel *Geometry* orientation is not only attitude-dependent, but also depends on the steering angle of the arrays. Therefore, its normal vector (the "boresight" of the half-space *Geometry*) must be updated while sun-tracking, and stay fixed. Two steps were taken:

1. **Planning** of sun-tracking and fixed intervals: an inherent SciOps timeline property. Dependent on instrument observation, disturbances by solar array actuation may not be acceptable.
2. **Steering** of the panel geometry based on the planned intervals. Orienting the panel normal in a fixed power-optimized direction while not actively tracking.

We start with the planning of the intervals. The mission performance team provided an arc – steering constraint mapping table, a simplified version is shown in Table 5.4. This shows which arc types require fixed solar array angles (SAA fixed *true*) and fixed duration, as well as which arcs can be sun-tracking (SAA fixed *false*).

Table 5.4: SciOps arc types and associated solar array steering constraints.

Arc Number	Pointing Mode	Description	SAA Fixed	Fixed duration
1, 3, 5, 7	**NADIR**	VenSpec observations	true	Full Arc
2, 4, 6	**Std. SAR**	VenSpec x VenSAR observations	true	Full Arc
8	**UNREQ**	Instrument Cool-Down	false	–
9	**SUN**	Sun Calibration	false	–
10	**DARK**	Dark Calibration	false	–
11–20	**Std. SAR**	Different VenSAR observation types	true	112–488 sec
21	**ALTIMETRY**	VenSAR Altimetry observation	true	During Eclipse
22	**SRS**	SRS observation	true	During Eclipse
23	**EARTH**	Radio Science Experiment	true	488 sec
24	**EARTH**	Earth Communication	false	–
...	...	*– miscelaneous arc types –*	false	–
31	**UNREQ**	Rotation Wheel off-loading	true	Full Arc
32	**UNREQ**	Flip-around manoeuvre	false	–
33	**PERI RAISE**	Pericentre raise manoeuvre	true	600 sec
...	...	*– additional manouevre arc types –*	false	–

Recall from subsection 5.3.1 that pointing modes are assigned to all arc types and accessed every ephemeris time using the *VespaData* class. Using this same architecture, the *VespaData* class was extended to map the solar array steering constraints from Table 5.4 to the SciOps timeline. Similarly to its `get_pointing()` method, a `get_saa_fixed_interval(<p3.Spacecraft>)` method was created to extract the fixed interval start and end times for fixed steering arcs. Within the method, three cases are considered:

- **Full arc fixed**: returns the start and end times of the full arc.
- **Fixed duration**: Accesses the duration from Table 5.4, takes the midpoint of the arc, and returns start and end times around the midpoint.
- **Fixed during eclipse**: Uses PAS3's `spacecraft.get_orbit_eclipse_intervals()` method to get the next eclipse interval. Returns this interval.

Note that an assumption was made for fixed durations to centre them around the arc midpoint. As the current SciOps timeline has arc-fidelity, and does not implement exact observation moments yet. During the fixed-array interval, it was decided to put the solar arrays in a fixed orientation that maximizes the time of optimal power generation at the average relative Sun position during this interval. The inertial position of the Sun during this (relatively short) interval stays essentially fixed, but the spacecraft attitude changes, and thus the average relative Sun direction in body-fixed frame must be considered in the interval. These assumptions were discussed and accepted with the mission performance team. The midpoint duration interval is illustrated in Figure 5.23

With the planning of fixed and sun-tracking intervals in place, the steering of the panel geometry can be implemented. A *SteeringManager* model was put in place. Knowing the steering is dependent on the spacecraft and the Sciops timeline, and must rotate (multiple) *SteerableSolarArray* objects, it takes the *Spacecraft*, *VespaData* and list of *SteerableSolarArray* objects as input parameters:

```python
class SteeringManager:
    def __init__(self,
                 spacecraft: p3.Spacecraft,
                 SciOps: VespaData,
                 solar_arrays: list[SteerableSolarArray]):
        self.sim = spacecraft.simulation # Access the Spacecraft's PAS3 Simulation object
        ...
```

Listing 32: *SteeringManager* initialization.

The task of accessing the time-averaged relative Sun direction during fixed intervals w.r.t a spacecraft with dynamic attitude, sounds relatively complex. Luckily, PAS3 makes this much more accessible. Listing 33 shows how PAS3's encapsulation of spacecraft state, attitude and simulation time, makes static functions like this very straightforward to implement. Requiring (only) 6 lines of actual code to calculate the required result.

```python
def get_average_sun_direction(self, start, end, n):
    """Calculates the average Sun direction vector in
    ↪ body-fixed frame over a fixed interval."""

    # Store time start time:
    store_et = self.sim.et

    # List to store sun directions:
    sun_directions = []

    # Divide fixed interval in uniform steps:
    for et in np.linspace(start, end, n):
        # set time (updating position and attitude):
        self.sim.set_time(et)

        # store sun direction in body-fixed frame:
        sun_directions.append(self.spacecraft.get_sun_
        ↪ direction_vector_in_body_fixed_frame())

    # reset simulation to the time before function call:
    self.sim.set_time(store_et)

    # Return normalized average Sun direction:
    return normalize(np.mean(sun_directions, axis=0))
```

Listing 33: PAS3 object implementation of averaging n body-fixed Sun directions over a fixed interval.

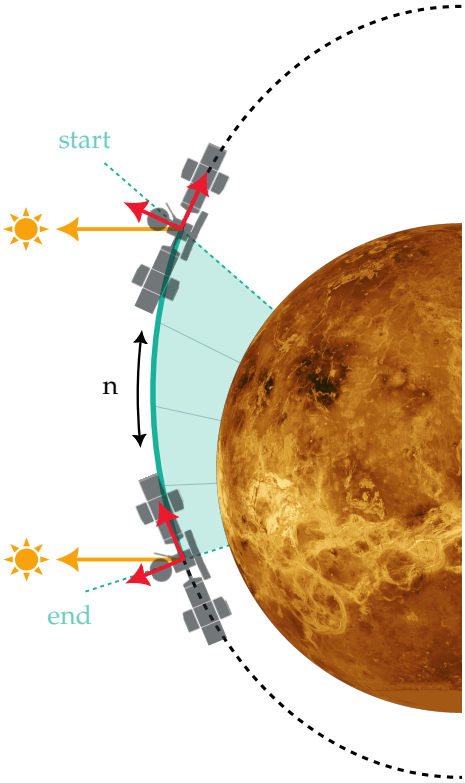


Figure 5.23: Midpoint fixed interval illustration with n different spacecraft states.

Having obtained an average relative Sun direction vector, the panel geometry can be rotated as close as possible to this direction around the rotation axis of the arrays; either during sun-tracking intervals, or fixed intervals.

Within the *SteerableSolarArray*, a steering method was created, taking an arbitrary (Sun) direction as input. For power-optimized orientation, the normal vector of the panel geometry must align with the projection of the Sun direction onto the rotation plane of the arrays. This follows Equation 5.21, with $\hat{v}_{sun}$ the Sun direction unit vector and $\hat{r}$ the rotation axis.

$$\hat{v}_{proj} = \hat{v}_{sun} - (\hat{v}_{sun} \cdot \hat{r})\,\hat{r} \tag{5.21}$$

When obtaining a projected direction $\hat{v}_{proj}$, The PAS3 built-in *Geometry* `set_vertices(`$\hat{v}_{proj}$`)` is called, essentially resetting the boresight vector of the panel's half-space to the power-optimized orientation. Within the *SteeringManager*, it is only a matter of integrating all dependencies in an `update()` function, which checks the simulation time, determines the rotation constraint, averages the sun direction (if necessary), and steers all assigned *SteerableSolarArray* objects accordingly. This `update()` function can then be called every simulation time step in the main simulation loops. These EnVision-specific models can then be integrated into the EnVision `config.py` file.

The correct global panel orientation of the SAA can now be obtained at every simulation time. This means, ϑ_i from Equation 5.20 can now be accessed correctly, accounting for both spacecraft attitude and solar array steering. The missing S_{sh} shadowing factor will be described in the next subsection.

5.5.2. Partial Panel Shadowing

Having modelled the correct orientation of the solar arrays, both spacecraft attitude- and steering-wise, the next step is to determine whether the arrays are partially shadowed by the spacecraft body in certain spacecraft attitudes. This section will show how creative use of general PAS3 blueprints can be used to implement such a shadowing function.

To estimate the effect of shadowing on the solar arrays by the spacecraft bus, two circular cone-shaped PAS3 *Geometry* objects were created for every panel: One outer cone and one inner cone. They can be initialized as per Listing 10. Both encapsulate the outer edges of the spacecraft bus, but differ in their half-angles (ϑ_{in} and ϑ_{out}) and positions. The inner cone is scaled in such a way that a Sun incidence vector outside of it, will always result in the full panel to be illuminated. The outer cone is scaled such that if the Sun incidence angle is contained by it, the full panel is shadowed by the Spacecraft bus. These cases define the extreme cases: either total illumination ($S_{sh} = 0$) or total shadowing ($S_{sh} = 1$). Any Sun incidence angle in between these two extremes will result in partial shadowing. Both cones are illustrated in Figure 5.24.

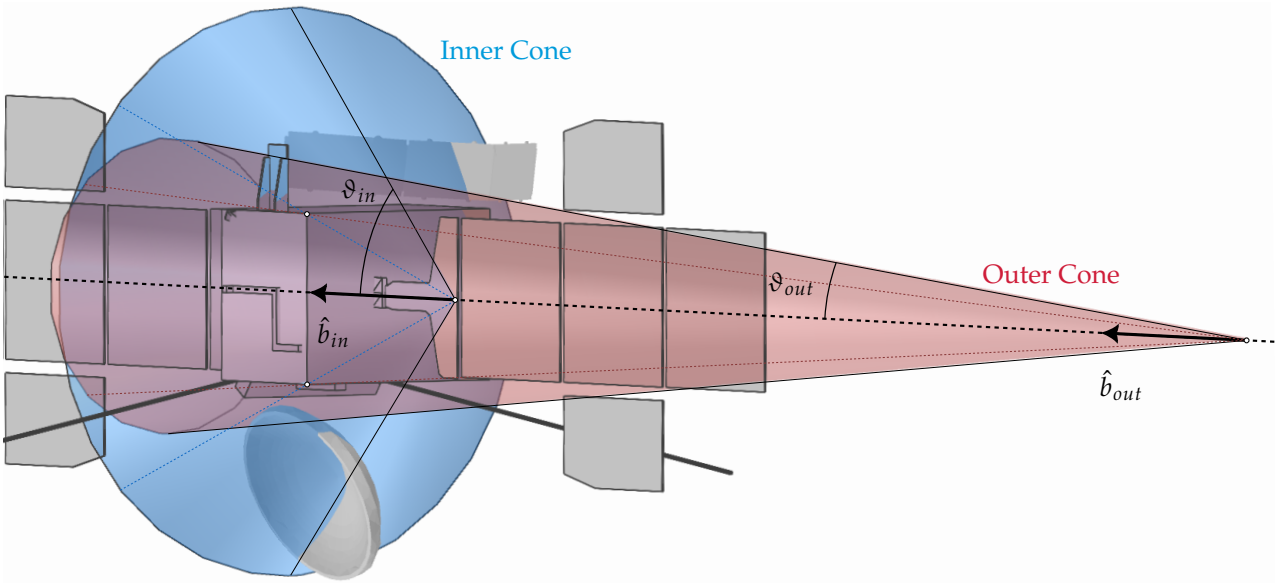


Figure 5.24: Inner and outer shadowing cones for EnVision's solar array shadowing model, with ϑ the cone half-angles and $\hat{b}$ their boresight vectors.

Note that the conditional statements in Equation 5.22 can be reduced to only the partial shadowing case (the equation includes extrema); The approach aims to showcase the full logic of implementing the PAS3 objects. In higher-detail shadowing models, the cones may be initialized as non-circular polygonal cones, or even *AssembledGeometry* objects to better determine full shadowing cases.

To determine the partial shadowing factor S_{sh} between these two cases, a linear relationship is assumed between both extrema. The Solar incidence angle ϑ_i is analogous to the angle between the Sun direction vector and the boresight vector of either cone ($\hat{b}_{in}$ or $\hat{b}_{out}$). And can be accessed using the *Geometry* method `angle_of_incidence_in_rad()`. This results in the following equation:

$$S_{sh} = \begin{cases} 1.0 & (\hat{v}_{sun} \text{ in outer cone}) \\ 0.0 & (\hat{v}_{sun} \text{ not in inner cone}) \\ (\vartheta_i - \vartheta_{in}) / (\vartheta_{out} - \vartheta_{in}) & (\text{partial shadowing}) \end{cases} \tag{5.22}$$

Initializing these two *Geometry* objects for every solar array panel (four cones in total), ensures they remain fixed w.r.t. the spacecraft body, while the Sun incidence angle changes dynamically with spacecraft attitude. Recall how the power model integrated in the *SteerableSolarArray* requires a callable shadowing function to be provided in its initialization in Listing 30, which should output a S_{sh} value for its integrated power model. Using the two shadowing cones and the linear approximation described above, such a function can be implemented as shown in Listing 34.

```
THETA_OUT = OuterCone.half_angle_in_rad # ϑ_out
THETA_IN  = InnerCone.half_angle_in_rad # ϑ_in

def shadowing_function():
    if OuterCone.sees_the_sun():
        return 1.0 # fully shadowed
    if not InnerCone.sees_the_sun():
        return 0.0 # no shadowing

    # partial shadowing:
    angle = OuterCone.angle_of_incidence_in_rad(
        -EnVision.get_sun_direction_vector_in_body_fixed_frame() )

    return (angle - THETA_IN) / (THETA_OUT - THETA_IN) # linear approximation for S_sh
```

Listing 34: Shadowing function using PAS3 cone *Geometry* objects.

This approach to partial shadowing ensures a conservative result; it overestimates shadowing as the circular cones extend to the outer corner points of the spacecraft bus (and therefore have larger regions of partial / full shadowing). This was discussed and accepted internally with the lead power systems engineer. Shadowing by the HGA, VenSAR array and SRS were not considered. The intended use case is a high-level power sizing analysis; complex shadowing effects are better determined using dedicated ray tracing software.

5.5.3. Power Sizing Simulation

With the Panel properties initialized in Listing 30 (panel area, & fill factor), the solar irradiance accessed by the *Spacecraft* PAS3 object, and custom models set up for steering and shadowing (resulting in ϑ_i and S_{sh}), the only mission variable in Equation 5.20 left is the solar cell efficiency η_{cell}. A choice could be made to use a constant efficiency value, or a temperature-dependent function. Initially, it was planned to integrate the panel thermal model from subsection 5.4.3 to extract panel temperatures, and use a temperature-dependent efficiency function. However, due to time constraints, and with the agreement of the power systems engineer, the model was simplified to consider a sub-optimal power scenario for this first sizing iteration. Equation 5.20 was divided in two pieces: a constant power level at optimal irradiance (i.e. fully lit, at 0° AOI), and a geometric scaling factor dependent on incidence angle and shadowing. The constant power level is set at the worst-case efficiency; coinciding with hot panels right before entering eclipse (panel efficiency is much higher when cold, coming out of eclipse). This results in the following modified power equation:

$$P_{MPP} \approx \underbrace{I \left(A_{array} FF\right) \eta_{cell}}_{P_{min}=5400 \text{ [W]}} \underbrace{cos\left(\vartheta_i\right)\left(1 - S_{sh}\right)}_{\text{geometric scaling factor}} \tag{5.23}$$

Using a constant worst-case power level P_{min} of 5400 W at optimal irradiance, the power model can now focus on the geometric scaling factor only (which was the driving focus of the analysis). The power model is however

set up in such a way that integrating the thermal panel model (and a callable efficiency / degradation function) is compatible in future iterations.

With the power model set up, it can be implemented in a PAS3 simulation loop. The main goal of the power sizing simulation was to add an extra layer of power budget to the existing SciOps timeline. This was performed successfully, but unfortunately, due to the nature of the results, they are not very presentable to be included in this work. As the power is simulated in relatively small time steps, over 4 years (6 Venus cycles), with numerous arcs (±45 minutes), at different attitudes, and periodical eclipses, the full result has the appearance of noise, rather than clear trends. The inclusion of a battery storage model and power consumption at every arc was not initialized yet, so critical power deficits could not yet be identified as clear results for this report.

However, During the development of the visual tool of chapter 6, and visual identification of the implemented shadow zones, the power lead systems engineer noted that the shadowing cases might not have been considered in current low-level array modelling. This might lead to an underestimation of generated power for high-consumption VenSAR arcs at low beta-angles in the current SciOps timeline. This gave the opportunity to perform a power analysis on an arc-to-arc level, resulting in more presentable results.

```python
from envision_config import EnVision, SciOps, PY_halfspace, NY_halfspace
from thermal_model   import SteerableSolarArray, SteeringManager

# Setup custom solar array & steering manager objects (see Listing 30, Listing 32):
PY_SAA = SteerableSolarArray(..., PY_halfspace, [0, 1, 0], PY_shadowing_function) # +Y panel
NY_SAA = SteerableSolarArray(..., NY_halfspace, [0, 1, 0], NY_shadowing_function) # -Y panel
Steering = SteeringManager(EnVision, (PY_SAA, NY_SAA), SciOps)

# Main simulation loop over VenSAR high-res arcs at low beta-angles:
DT = 1
for et in arc_start_times:
    sim.set_time(et)
    arc_type = SciOps.get_arc_number(sim.et)
    if arc_type == 19 and abs(EnVision.beta_angle_in_deg) < 1: # ONLY VenSAR high-res, low β

        _, end_of_this_arc = SciOps.get_arc_start_end_time(sim.et)
        while sim.et < end_of_this_arc:
            Steering.update() # Steer panels

            PY_power = PY_SAA.approx_power_generated(min_power=5400)
            NY_power = NY_SAA.approx_power_generated(min_power=5400)

            # Store data:
            ...

            sim.advance_time(DT) # Advance time with small timestep
```

Listing 35: Focused PAS3 simulation loop for power generation on VenSAR high-resolution observation.

Note how Listing 35 shows a focussed simulation loop over only VenSAR high-resolution arcs (arc type 19) at low beta-angles ($< 1°$)[1], taking small time steps ($dt = 1$ second) using the *Simulation* `advance_time()` method. This allows the simulation to capture only what's relevant for this analysis. On the other hand, the `set_time()` method allows for "discontinuous" time jumps (even backwards in time if needed like listing 33) to very specific times. This highlights the deliberate decision of not including dedicated "run simulation" methods in PAS3, giving the user full flexibility and agency in setting up the architecture of their simulation. The approach taken in Listing 35 drastically reduces the total simulation time; what could have taken many hours with fixed time step, took about a minute with selective time stepping.

In Figure 5.25, the power generation of arc no. 22,111 at minimum beta-angle of –0.03° (2036-07-31) is shown over the whole arc duration. Both individual panel powers are shown, as well as their sum. Due to the uniform rotation of both panels, their solar incidence angles are identical, and thus their power profiles overlap

[1]The upper bound of the beta-angle is arbitrary; partial shadowing will occur in "Std. SAR" pointing at $|\beta| < \vartheta_{in}$. 1° was taken for the purpose of plotting the worst-case scenario.

when not shadowed. The plot can be more easily interpreted by looking at Figure 5.23, which shows the along-track orientation of the panels during a "Std. SAR" arc. The plot starts at the very top (shadowing one panel completely), and ends at the very bottom (shadowing the other panel completely). The small edge discrepancies at the start and end of the arc are due to the arc start and end times not perfectly aligning with the geometrical "top" and "bottom" orbit positions.

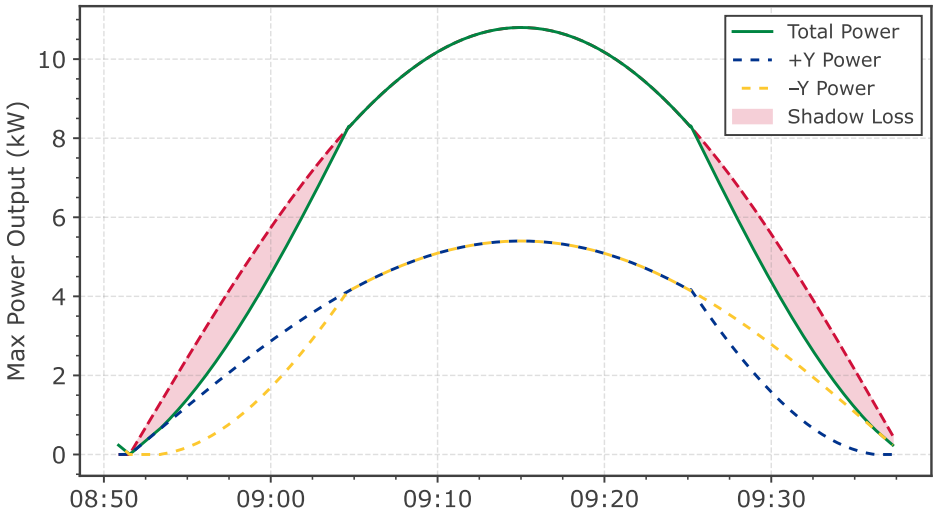


Figure 5.25: Maximum power generation with shadow losses from both solar panels during VenSAR high-resolution at a beta-angle of −0.03° (2036-07-31).

Note that in a realistic scenario, the total power throughput of the panels will be capped at a fixed power conditioning limit; this is not portrayed here. Nevertheless, the plot clearly shows the effect of partial shadowing on both panels during this arc. The power loss due to shadowing for the whole arc is about 7% (369 kWh). However, the losses are greatest near the poles, possibly affecting VenSAR high-resolution imaging of polar regions of interest. The significance will be further investigated by the team in the future.

5.5.4. Power Model Verification

In order to verify the implemented power model, a verification campaign was started to check the correct functioning of the solar panel's steering and shadowing functions. To obtain predictable results, the same circular orbit around Venus from section 4.5 was used, only with a non-disturbed Keplerian orbit perpendicular to Venus's orbital plane. This orbit will be refered to as the "Polar" orbit (even though strictly speaking, it is not polar w.r.t Venus itself). This orbit setup ensures conditions where the spacecraft's orbit beta-angle shall become exactly 0° and 90° at predictable times. The orbit parameters are shown in Table 5.5. The orbit is initialized to start at a 90° beta-angle (i.e. perpendicular to the Sun-Venus line) at 0 TDB (01/01/2000 11:58:55.8 UTC).

Table 5.5: Power verification orbit parameters expressed in the J2000 (ICRF) reference frame at 0 TDB (01/01/2000 11:58:55.8 UTC).

		Venus Verification *Polar* Orbit
Position Vector	[km]	x: 374.52; y: -2662.38; z: 5917.89
Orbit Radius	[km]	6500.0
Velocity Vector	[km/s]	v_x: 0.31; v_y: -6.43; v_z: -2.91
Velocity Magnitude	[km/s]	7.07
Orbit Period	[hh:mm:ss]	1:36:17
Eccentricity	[-]	0.0
Inclination	[°]	91.96

For the verification setup, two *SteerableSolarArray* objects were initialized equivalent to EnVision's solar arrays, with rotation axis along the spacecraft Y-axis; the shadowing cones were also included, but with more rounded half-angles of ϑ_{in} = 50° and ϑ_{out} = 25° to better identify partial shadowing modelling. The *SteeringManager*

was initialized with both panels and a custom *VespaData* object was created to handle the fixed/sun-tracking intervals (EnVision's timeline did not apply here).

For spacecraft attitude, an *along-track* nadir-pointing mode (equivalent to EnVision's "NADIR" mode) and a *cross-track* nadir-pointing mode (similar to EnVision's "ALTIMETRY" mode, but without VenSAR's tilt considered) were implemented. These essentially point the spacecraft body X-axis towards nadir, with the Y-axis (the solar panels' rotation axis) either *along-track* or *cross-track*. Combining these two attitudes with an orbit at 0° and 90° beta-angles, results in four distinct symmetrical cases allowing to cleanly verify both panel orientation and shadowing factors separately. This verification campaign will verify the geometric scaling factor from Equation 5.23 only, the power in the results shown in subsection 5.5.3 are generated with a constant worst-case power level, meaning the only variability is due to geometry. The coming verification plots will show these factors only, with:

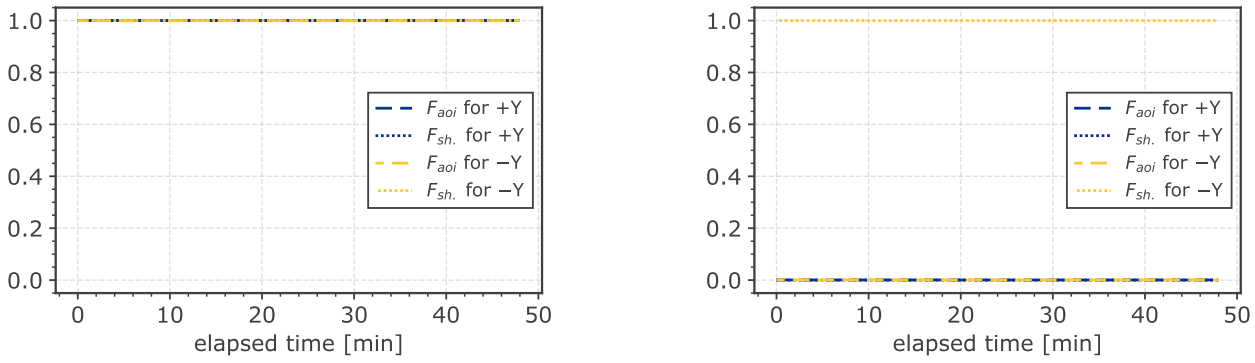
$$F_{aoi} = cos\,(\vartheta_i) \quad \text{with } (0 \leq F_{aoi} \leq 1) \tag{5.24}$$

$$F_{sh.} = (1 - S_{sh}) \quad \text{with } (0 \leq F_{sh.} \leq 1) \tag{5.25}$$

The plots will indicate results for the +Y panel in dark blue, and the –Y panel in light yellow. The solar incidence factor F_{aoi} will be shown with solid dashed lines, and the shadowing factor $F_{sh.}$ with dotted lines. The x-axis will indicate elapsed arc time in minutes.

The verification started with the 90° beta-angle case. This meant that for the full arc, The Sun direction vector ($\hat{v}_{sun}$) is perpendicular to the orbital plane of the spacecraft (with orbital plane normal vector $\hat{n}_{orb}$). The following condition applies: $\angle \hat{v}_{sun} \hat{n}_{orb} = 0°$. Of course, while the arc is running, the beta-angle will slightly deviate from 90° due to the motion of Venus; small disturbances are to be expected. In this orientation, it is expected that both panels are fully illuminated in *along-track* nadir-pointing mode, while one being fully shadowed in *cross-track* nadir-pointing mode. In the following cases, both panels are sun-steering during the full arc.

The results for the 90° beta-angle case are shown in Figure 5.26. As expected, these plots are very monotonous; in the *along-track* attitude, the spacecraft solar panels are perfectly perpendicular to the Sun, resulting in both panels being fully illuminated at optimal incidence angle without any shadowing (all factors equal to 1 in Figure 5.26a). The opposite is true for the *cross-track* attitude; Now the panels are parallel to the Sun direction, resulting in the +Y panel being fully shadowed by the spacecraft bus, and the –Y panel being fully illuminated, but at a 90° incidence angle. This means that all factors but the shadowing factor of the –Y panel are equal to zero, as shown in Figure 5.26b.



(a) *Along-track* nadir-pointing mode, sun-tracking panels.

(b) *Cross-track* nadir-pointing mode, sun-tracking panels.

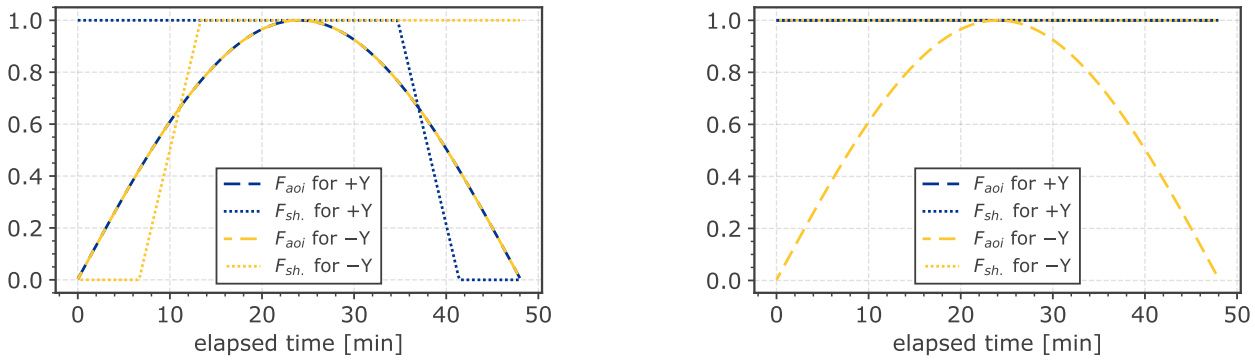
Figure 5.26: Geometric scaling factors for AOI (F_{aoi}) and shadowing ($F_{sh.}$) during 90° beta-angle orbit arc for *along-track* **(a)** and *cross-track* **(b)** orientation of the panels.

However, solar incidence factors were not exactly equal to 1 or 0, but rather oscillated slightly near these values. This is due to the small deviations from the perfect 90° beta-angle during the arc. Assertions in the verification window were placed to assure that these deviations stayed below the recorded beta-angle deviations from 90°, ensuring no modelling mistakes of the sun-tracking behaviour.

Disabling and enabling sun-tracking of the panels during this beta-angle will not show any interesting results. Due to the perpendicular Sun direction, the average relative Sun direction will stay fixed in both attitudes.

This also causes the shadowing factors to remain constant. To have a better verification case on the steering and shadowing models, the 0° beta-angle case was considered next.

For the 0° beta-angle case, again, both *along-* and *cross-track* attitudes were considered. However, it was decided to only make the +Y panel sun-tracking, while the –Y panel was assumed defective with a fixed alignment with the –X axis. This allows to see the effect of sun-tracking vs (partial) fixed panel orientations more clearly. The initial results for both attitudes are shown in Figure 5.27.



(a) *Along-track* nadir-pointing mode, sun-tracking +Y panel.

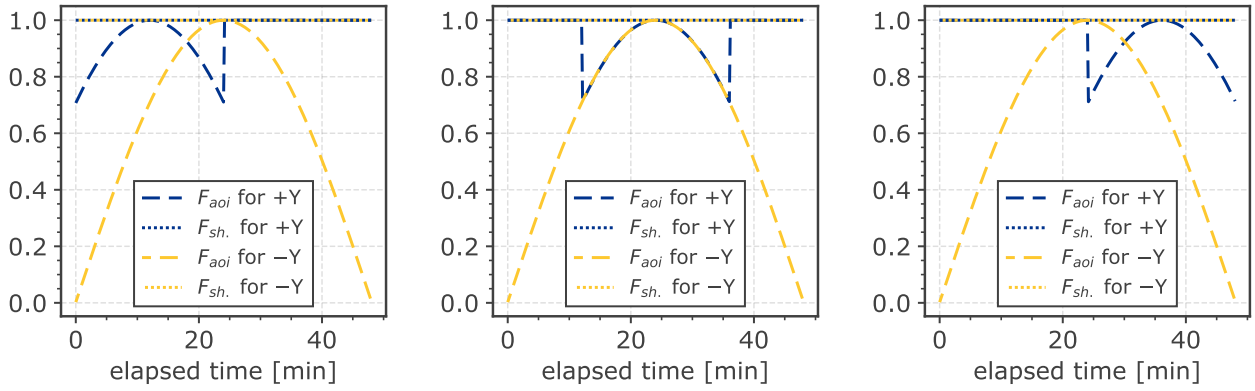
(b) *Cross-track* nadir-pointing mode, sun-tracking +Y panel.

Figure 5.27: Geometric scaling factors for AOI (F_{aoi}) and shadowing ($F_{sh.}$) during 0° beta-angle orbit arc for *along-track* **(a)** and *cross-track* **(b)** orientation of the panels.

Figure 5.27a is analogous to the illustration in Figure 5.23. At first, the –Y panel is fully shadowed ($F_{sh.} = 0$), while the +Y panel is fully illuminated ($F_{sh.} = 1$). Later on (after about 9 minutes), the sun exits the outer cone of the –Y panel, resulting in partial shadowing until exiting the inner cone after about 11 minutes. Nearing the other pole, the situation reverses. Even if both panels are steering, the *along-track* attitude prevents optimal solar incidence angles (due to the Y body axis being aligned with the sun direction near both poles).

In Figure 5.27b, the *cross-track* attitude allows the +Y panel to achieve optimal solar incidence angles (i.e. $F_{aoi} = 1$) when steering, while the fixed –Y panel experiences similar incidence losses as in the *along-track* case. Also, in this orientation, both panels experience full illumination throughout the arc (the sun is in the spacecraft's XZ plane). As this orientation nicely shows the effect of the sun-tracking algorithm, this exact condition will be used to verify the partial tracking behaviour of the *SteeringManager*.

Three cases were set up, each with different fixed-interval durations for the +Y panel. First, an observation (i.e. fixed panels) is planned in the first half of the arc, then a second observation in the middle of the arc, and finally an observation in the last half of the arc. For each fixed interval, the *SteeringManager* calculates the average relative sun direction over the interval, and freezes the panel orientation accordingly. The results of these three cases are shown in Figure 5.28.



(a) Fixed-panel interval in first half of the arc.

(b) Fixed-panel interval in the middle of the arc.

(c) Fixed-panel interval in last half of the arc.

Figure 5.28: Geometric scaling factors for AOI (F_{aoi}) and shadowing ($F_{sh.}$) during 0° beta-angle orbit arc in *cross-track* nadir-pointing mode with the +Y panel fixed for the first **(a)**, middle **(b)** and last **(c)** half of the arc. (–Y panel permanently fixed along –X spacecraft axis)

Note in the figures how the maximum solar incidence factor F_{aoi} aligns with the average Sun direction during the fixed intervals (i.e. the midpoint of the interval). The fact that the average Sun direction coincides with

the midpoints of the intervals is expected, as verification orbit is circular (and therefore symmetric on every interval). In Figure 5.28b, the average Sun direction in the Spacecraft body frame is the –X direction, resulting in the +Y panel being fixed with the (defective) –Y panel during its fixed interval. The average Sun direction during the other two fixed intervals is $< cos(45°), 0, -cos(45°) >$ for the first half, and $< -cos(45°), 0, -cos(45°) >$ for the last half (as expected with the circular orbit). The instantaneous jumps in fixed- and tracking-mode are present because the *SteeringManager* does not consider maximum rotation speed limitations yet; this was accepted internally for the first implementation, and might be added in future iterations.

Lastly, the shadowing functions were verified in more detail as well. This was done with a slightly different approach. Instead of running the PAS3 *Simulation* of the verification spacecraft, the spacecraft was fixed in space and rotated manually to check the shadowing factors at different attitudes. The spacecraft was set to the *along-track* nadir-pointing mode, again with +Y sun-tracking and –Y fixed in the –X direction. The orbit was set during a 0° beta-angle arc. The spacecraft was then rotated around its X-axis up to 180° using the PAS3 utility function `rotate_frame()`. The results are shown in Figure 5.29.

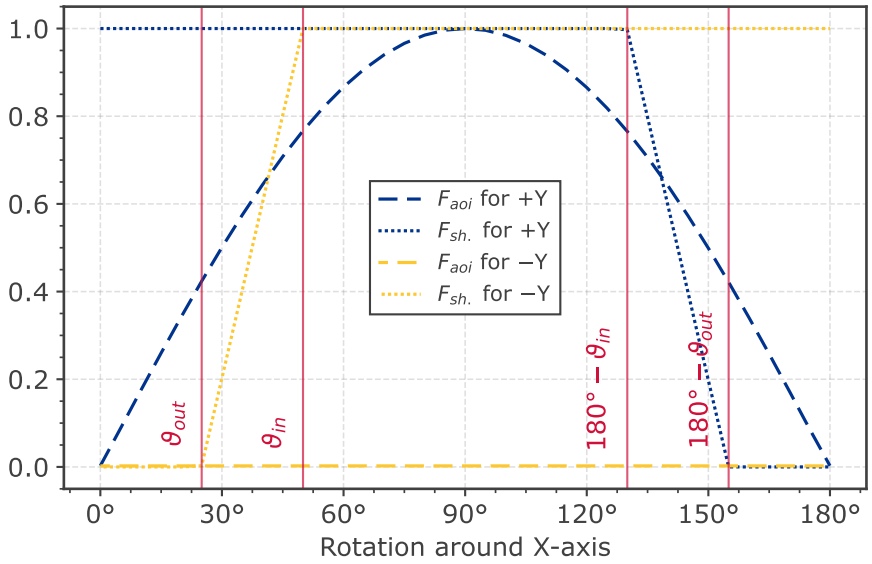


Figure 5.29: Geometric scaling factors for AOI (F_{aoi}) and shadowing ($F_{sh.}$) for both solar panels at 0° beta-angle with spacecraft in *along-track* nadir-pointing mode, while manually rotating the spacecraft around its X-axis.

As the rotation was started at the pole of the orbit, the –Y panel (being fixed in the –X direction, which is nadir pointing) is looking directly at Venus with the Sun perfectly in-plane (i.e. 90° incidence angle). The +Y panel is sun-tracking, but due to the spacecraft rotation, cannot maintain a 0° incidence angle, resulting in increasing / decreasing behaviour of its F_{aoi}. The interesting part is the shadowing factors. Their behaviour matches perfectly the expectations from Figure 5.26a, but now the underlying model is clearly visible; the linear partial shadowing relationship falling perfectly between the initialized inner and outer cones' half angles.

This verification campaign successfully verified the correct functioning of both the steering and shadowing models implemented for EnVision's power sizing analysis. It is recognized that the instantaneous steering behaviour and the linear shadowing approximation are simplifications, but their behaviour shows to be correctly implemented. While the thermal modelling chapters in section 5.4 proved PAS3's application in mission-wide (sub)system analysis (spanning years), this section proved the toolkit's ability to zoom into analysis on a minutes-scale as well.

6
Mission Visualization

Within the context of the internship, during which this thesis is written, the primary deliverable was a mission visualization tool to support contextualization of the complex EnVision mission during design and analysis activities.

This was identified as a general need in the development of interplanetary missions in section 2.2. The state-of-the-art analysis in section 3.1 initially placed much emphasis on this use case, evaluating existing tools based on their simulation *and* visualization capabilities. Eventually, it was decided to separate the visualization aspects from the simulation toolkit itself in section 3.2; the outdated approach of monolithic applications was identified and rejected in favour of a modular toolkit/package that users can employ within their own specific implementations.

Figure 6.1: WebVision repository logo.[1]

Although the majority of this thesis revolves around the simulation toolkit package itself, the visualization tool is an important deliverable to address the identified needs of the system engineer. It also serves as an opportunity to demonstrate the extensive flexibility in implementation of the agnostic simulation toolkit; showing it is not limited to pure numerical analyses in chapter 5, but is also capable of supporting a wider range of external integrations in new applications.

This chapter will cover the developed visualization tool. First, section 6.1 establishes the requirements of the application, gathered through interviews with the end-users and the internship's needs description. Then, section 6.2 discusses the development of the web application and its architecture (frontend and backend). Finally, section 6.3 presents the resulting application, discussing its features and verifying its requirements' coverage.

The resulting application is a web-based 3D mission visualization tool called **WebVision**. It was developed using modern web technologies (HTML, JavaScript, FastAPI, Three.js) to ensure cross-platform compatibility, accessibility, and customizability. The tool interfaces with a PAS3 simulation backend, and is set up to support interactive visualization of the EnVision mission fully inside the user's web browser.

6.1. Requirements

The main objective of the application, literally taken from the job description, was:

> *"The objective of this tool is to allow engineers from the various engineering disciplines to have a better understanding of the external environment of the spacecraft."*

In this section, the requirements of the visualization tool are established. These requirements were gathered through the internship's needs description, and were further refined with interviews with potential end-users (i.e. the EnVision project team members). Following the software requirements, both functional and non-functional, ensures the developed application meets the expectations and needs of its users. The functional requirements are listed in Table 6.1, they sum up what the application *should do*.

[1] AI-generated using Microsoft Copilot

Table 6.1: Functional requirements of the 3D mission visualization app.

ID	Description	Reasoning
	Display:	Agreed upon elements to be included in a 3D rendering of the spacecraft. This ranges from physical model meshes (Spacecraft, Sun, Venus, FOV's) and UI indicators (time, ground stations, operations).
VIS-F-1.1	The application shall display the EnVision CAD model in orbit at the correct planned position and orientation.	
VIS-F-1.2	The application shall display Venus as central body at correct orientation.	
VIS-F-1.3	The application shall display the Sun at correct relative direction.	
VIS-F-1.4	The application shall indicate which ground stations are visible.	
VIS-F-1.5	The application shall display the simulation time of the provided scene.	
VIS-F-1.6	The application shall display planned operational modes of the mission.	
VIS-F-1.7	The application shall display sensor FOV's	
VIS-F-1.8	The application shall display pointing vectors in the direction of: • Earth • The Sun • Spacecraft Velocity • Nadir	
VIS-F-1.9	The application shall display rendered models in true-scale.	
	Interactivity/Controls:	These requirements cover the desired interactivity of the user. Time, camera movement and rotation allow the user to view desired spacecraft/mission elements. Anchoring to either EnVision or Venus gives a spacecraft- and orbit-centred view to the simulation.
VIS-F-2.1	The application shall allow the user to change the simulation time and speed.	
VIS-F-2.2	The application shall allow the user to rotate and pan around the following anchor points: • EnVision body frame origin • The central body origin (Venus)	
VIS-F-2.3	The application shall allow the user to change the spacecraft attitude to different pointing viewpoints.	Requirement from internship description.
VIS-F-2.4	The application shall allow the user to toggle vectors and FOV displays.	Stems from **VIS-NF-3.1**.
	Data Integration:	
VIS-F-3.1	The application shall provide an interface to display simulation data from the rendered simulation scene.	This sets the application apart from visualization-only tools and more tailored to the design process.

The functional requirements are complemented by non-functional requirements, which define the quality attributes of the application. These are listed in Table 6.2. In essence, they describe what the application *should be*.

Table 6.2: Non-functional requirements of the 3D mission visualization app.

ID	Description	Reasoning
	Accessibility:	
VIS-NF-1.1	The application shall limit the setup/installation steps needed to run, preferably be a press-and-play type application.	It was determined that having to take many building steps (i.e. cloning repositories, downloading runtime environments, installing packages, running code in an IDE) will drastically limit accessibility and hinder usage.
	Extendability & Customizability:	
VIS-NF-2.1	The application shall be modular in its architecture, allowing future additions and extensions.	Requirement from internship description.
VIS-NF-2.2	The application shall be compatible with other missions than EnVision.	Requirement stemming from the generalized applicability of the thesis work and the existing interest from other ESA projects.
	Additional Use Cases:	
VIS-NF-3.1	In addition to supporting mission contextualization in a design context, the application shall be tailored to support outreach activities and interdisciplinary knowledge transfer.	Additional requirement that appeared during the later stages of development, recognizing the outreach potential for a scientific mission like EnVision, and potential to complement technical documents/presentations.

The aim for extendability & customizability in **VIS-NF-2.1** is an important non-functional requirement before the start of development. It ensures the developed application can be adapted to future needs in different mission development phases. There is future interest in interfacing the tool with external API's, such as telemetry or live test data streams. Additionally, the modular and customizable architecture falls in line with the overall design approach of this thesis work, described in section 3.3 (referring to the approach of "toolkits for user innovation").

With the requirements established, the next section discusses the development of the web application and its architecture. The requirements in Table 6.1 and 6.2 directly influence the decisions made in the development tools (e.g. programming language, deployment platform, distribution method).

6.2. Visualization Application

This section discusses the development of the visualization application. In order to satisfy the established requirements in section 6.1, the correct development tools and architecture needs to be chosen in advance. The initial clear choice was to create an executable application using established game engines such as Unity, Godot or Unreal Engine. These engines provide powerful 3D rendering capabilities, user interactivity, and cross-platform compatibility. However, considering requirement **VIS-NF-2.1**, concerns were raised on the maintainability and extendability of such engines in the project's context. Games running on these engines are written in C# (Unity), C++ or built-in scripting languages (Unreal Engine & Godot), which would require team members to have knowledge of these languages. Additionally, the roadblocks of even installing the engines on ESA machines due to security policies, made these engines less favourable when considering the requirement in question.

While orienting in existing 3D tools, it was discovered that the thermal department at ESTEC had developed web-based visualization tools for spacecraft thermal analysis using the Three.js JavaScript library. These tools showed spacecraft temperature distributions on a mesh model from both ESATAN-TMS results and telemetry data. This led to the decision to develop a web application using Three.js for 3D mission rendering. The prospect of implementing the established thermal (telemetry) visualization tools was added value to this

choice. The choice of developing a web application had the following benefits:

+ Increased accessibility to user base as no installation is required; the application backend can be hosted on a server and accessed on the intranet through a web browser. (**VIS-NF-1.1**)
+ High customizability and extendability by using a popular web development stack (JavaScript, HTML, CSS, . . .), with established expertise and tools within ESA. (**VIS-NF-2.1**)
+ More straightforward implementation within different missions. No monolithic application to adapt and customize for the mission needs, but rather a code repository to change configuration files and assets. (**VIS-NF-2.2**)
+ Increased flexibility in design (bottom-up), by coding directly outside established engine frameworks (allowing integration with the PAS3 simulation backend through API calls).

After deciding on the approach, the *WebVision* application was developed. Figure 6.2 shows its architecture diagram, with the front end and back end components. The backend is responsible for handling the simulation data through a PAS3 simulation instance and exposing the relevant data through a REST API using the FastAPI library. Serialized data describing the spacecraft state, environment and mission elements are fed to the front end to be displayed (**VIS-F-1.1 – VIS-F-1.8**). The front end is responsible for rendering the 3D scene using Three.js. The user may control the simulation time, camera position and orientation, toggle visual elements and request custom attitudes through the Graphical User Interface (GUI) controls (**VIS-F-2.1 – VIS-F-2.4**). The front end sends the scene's time and attitude requests to the backend, which returns the updated spacecraft state for rendering.

Inside Figure 6.2, the relevant software tools' logos are indicated in the bottom; from left to right: JavaScript, HTML, CSS, Three.js, Node.js & Vite (for development and package management), FastAPI (API framework), Python and the SPICE Toolkit (using SpiceyPy integrated in PAS3). FastAPI was chosen for its high performance, ease of use, and growing community support. With the intended client size being small (only the EnVision team), scalability was not a primary concern, and an approach of serialized HTTP requests was sufficient to meet the current performance needs.

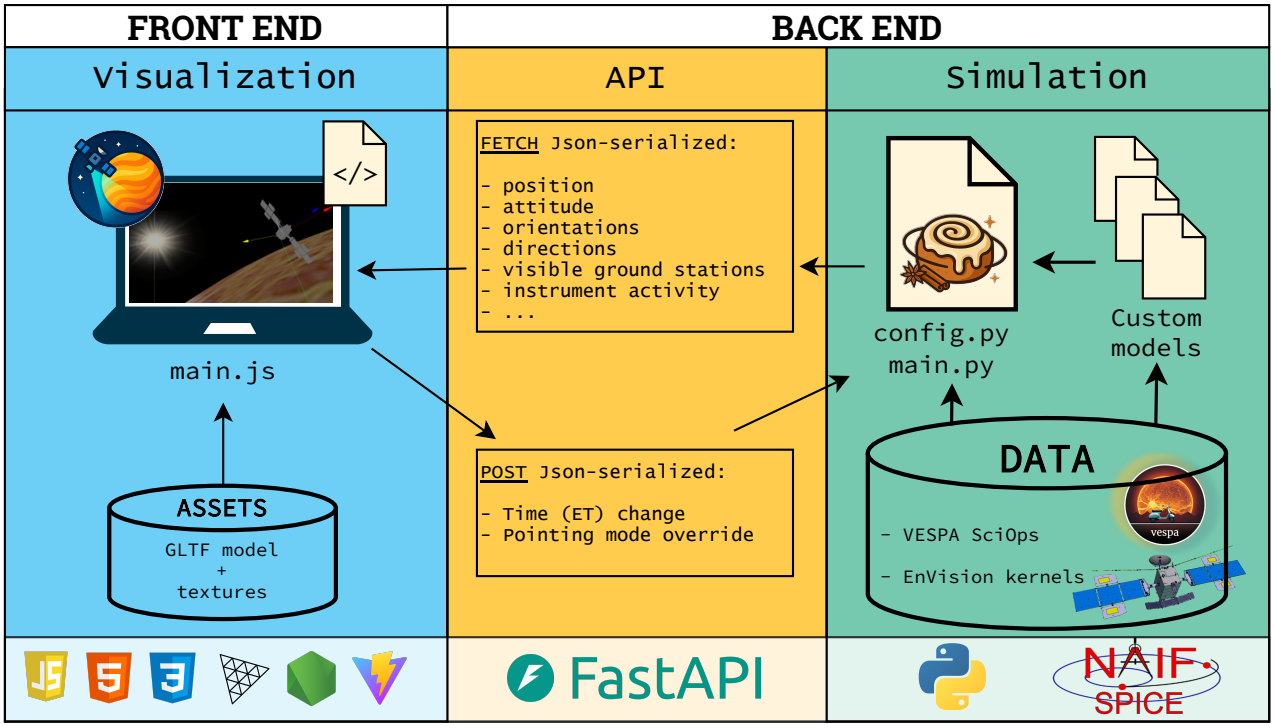


Figure 6.2: WebVision application architecture diagram.

6.2.1. Back End PAS3 Integration

In order to support running multiple simulations and users, one must ensure the same PAS3 object is not assigned to multiple users at the same time. The same EnVision configuration from chapter 5 is used, but wrapped in a `SimSession` data class to keep track of each session's objects. Instead of importing the same object as before, initialization functions were set up to dynamically create new instances per client (or browser tab),

instead of reusing them (e.g. "`import sim`" is replaced by a "`sim = create_sim()`" function). Each session gets a simulation, EnVision object, Venus object, SAA geometry with associated steering manager (to render rotating panels), and attitude generator (to provide custom attitudes on request). An asyncio lock is used to ensure thread-safe access to the PAS3 objects (FastAPI is multi-threaded). Additionally, a timestamp is stored to delete inactive sessions after a certain timeout. The initialization of the `SimSession` data class is shown in Listing 36.

```python
from dataclasses import dataclass, field
import asyncio
import time

import pas3 as p3
from attitude_model import AttitudeGenerator
from power_model import SteeringManager

@dataclass
class SimSession:
    sim: p3.Simulation                                  # PAS3 simulation instance
    spacecraft: p3.Spacecraft                           # PAS3 spacecraft instance
    central_body: p3.CelestialBody                      # PAS3 celestial body instance
    saa: p3.Geometry                                    # PAS3 SAA half-space instance
    steering: SteeringManager                           # Custom SAA steering manager
    attitude_generator: AttitudeGenerator               # Custom attitude generator
    lock: asyncio.Lock = field(default_factory=asyncio.Lock) # Asyncio lock
    last_used: float = field(default_factory=time.time)      # Timestamp of last usage
```

Listing 36: Session data class initialization for PAS3 simulation instances in WebVision.

The session object is created upon client connection and stored in a dictionary with a unique session ID as key. Every request from the front end includes this session ID, preventing other clients from interfering with other simulations being run. The backend API then exposes endpoints to get the current spacecraft, central body, Sun, and visibility states, depending on the user's requests for time (and attitude). The relevant data is serialized by FastAPI into JSON format and sent to the front end for rendering.

In Table 6.1, requirements are put to display the Sun direction, Earth direction, ground station visibility and instrument fields of view. However, *GroundStation* objects and *CelestialBody* objects for the Sun and Earth were not directly provided in the session object. This is because these attributes and objects are linked to the PAS3 using methods like: `Spacecraft.get_sun_direction()`, `Spacecraft.instruments`, or `Spacecraft.list_visible_ground_stations()`. Accessing these attributes and objects is done inside the API endpoints automatically.

It can be noted that the SciOps manager from subsection 5.3.1 is not integrated in the session. This is because this class is a read-only manager that does not hold any stateful data per client. The SciOps manager is initialized once on the server side, and used by all clients to retrieve planned operational modes at requested epochs (**VIS-F-1.6**).

6.2.2. Front End Visualization

The front end of the WebVision application is responsible for rendering the 3D scene and providing the user with a GUI. The Three.js library is used for 3D rendering. Three.js is a WebGL framework that simplifies the process of rendering 3D graphics in web browsers using JavaScript [75]. The library provides a number of built-in features (like renderers, cameras, lights, geometries, materials, ...) that lets developers focus on the application instead of low-level WebGL programming. The option of using *React Three Fiber* was considered, but due to inexperience with the React framework, it was decided to use plain Three.js for the initial implementation. The front end follows a standard folder structure as shown in Figure 6.3. The `index.html` file is the main entry point of the application, loading the necessary JavaScript and CSS files. The `main.js` file contains the core logic and imports all modules from the other directories in the `src\` folder.

First, inside `main.js`, the modules from the `scene\` directory are imported and used to set up the Three.js renderer, camera, scene, directional/omnidirectional lighting (from the sun). The API client module (getSimData.js) handles the communication with the backend API, sending requests and receiving serialized data. An

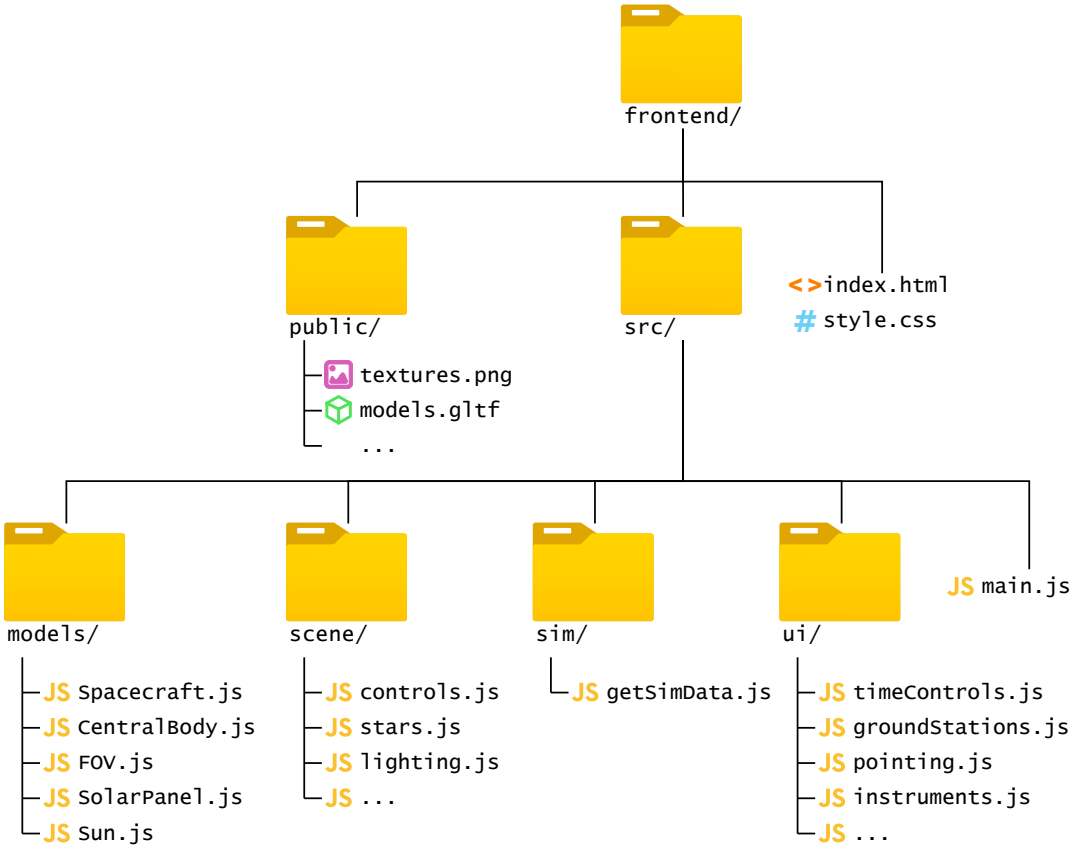


Figure 6.3: Folder structure of the WebVision front end.

initialization fetch request is sent to create a new simulation session, and return the relevant parameters to set up the scene. The initialization parameters fetched from the backend are summarized in Table 6.3.

Using the classes from the `models\` directory, Three.JS *Object3D* instances are created with associated meshes and materials. For example, the `Spacecraft` class creates a Three.js *Group* object containing the EnVision CAD model mesh (from the assets in the `public/` folder in `.gltf` format), helper vectors and instrument FOV's. The `CelestialBody` class creates a Three.js *Mesh* object with an ellipsoid geometry representing Venus, scaled to true size using the shape parameters fetched from the backend, and textures for surface details from the

Table 6.3: Initialization parameters fetched from the backend API to set up the 3D scene.

Parameter	Back End PAS3 Method	Initialization Use
Simulation start time	`Simulation.et`	To set the initial state of the scene.
position	`Spacecraft.position`	Initial spacecraft position vector.
attitude	`Spacecraft.body_rotation_matrix`	Initial spacecraft orientation.
pointing mode	Custom SciOps method	Planned operational mode at start time.
nadir vector	`Spacecraft.nadir_direction`	Render helper vector.
velocity vector	`Spacecraft.velocity_direction`	Render helper vector.
sun vector	`Spacecraft.sun_direction`	Render helper vector + Sun sprite.
earth vector	`Spacecraft.earth_direction`	Render helper vector.
central body rotation	`CelestialBody.body_rotation_matrix`	Set the initial orientation of Venus.
central body shape	`CelestialBody.shape`	Ellipsoid parameters for the central body mesh.
groundstations	`Spacecraft.ground_stations.keys()`	List assigned ground stations
visible ground stations	`Spacecraft.list_visible_ground_stations()`	To highlight visible ground stations at start time.
Earth visibility	`Spacecraft.is_earth_visible()`	Indicate Earth occultations.
active instruments	Custom SciOps method	Show/hide instrument FOV's at start time.

`public/` folder. A separate fetch is done, iterating through the spacecraft's instrument objects to retrieve their field of view specifications, names, locations and orientations on the spacecraft body. A Conical mesh using Three.JS *ConeGeometry* and *EdgesGeometry* in the `FOV.js` module is set up accordingly. Additionally, a sun sprite is created with the `Sun.js` module, overlaying a lens flare shader for visual effect. The scene is then populated with these objects, and the Hipparcos starfield catalogue is used to create a background of stars.

To provide the desired interactivity to the user, the `ui/` directory holds modules to dynamically create DOM (Document Object Model) elements depending on the backend initialization parameters. For example, the `instruments.js` module creates toggle buttons for each instrument based on the list of instruments fetched from the backend. It also indicates which instruments are active during the simulated arc. The `timeControls.js` module creates a time picker similarly to the one in Cosmographia, allowing the user to set the simulation time, or control the simulation with play/pause and speed controls. This sends a front-end stored simulation time to the backend API to update the PAS3 simulation state.

Importantly, `pointing.js` provides a dropdown menu to select different spacecraft attitudes supported by the backend's attitude generator. By default, the SciOps planned attitude is displayed; however, with this dynamic DOM element, the user may request custom attitudes that are sent to the backend API as a pointing override in the attitude generation function (see Listing 21).

While some of the individual modules in the front end only handle specific (and sometimes small) tasks, the modular architecture allows for future extensions and additions to be made more easily. UI elements can be added or modified without having to change the core module. Replacing textures or 3D models is supported and with the correct backend configuration, different missions can be visualized as well.

6.3. Results

With both the front end and back end developed, the final step was to build the application and distribute it to the end-users. The front end was built using *Vite*, bundling all JavaScript modules and assets into static files (HTML, CSS, JS) that can be served by any web server in production. Within the SCI directorate IT infrastructure, the process of requesting and setting up a virtual machine (VM) to host the application was started. This process was not pursued as it required significant development effort, and it did not fall within the thesis's scope. Intranet hosting will be considered in future work.

Instead, it was decided to package the entire application into a standalone executable using *PyInstaller*. This approach ensures that the end-users can download the application in the team's collaborative environment without the need of any git cloning, package installations or Python setups. The executable was tested on Windows and macOS systems. When clicking the executable, a local web server is started on the user's machine, and WebVision opens in the default web browser.

To allow collaboration and future development, the entire codebase was pushed on a repository within ESA's GitLab. To ease development, multiple batch scripts were created for starting a development server, running a preview server, building the front end, and packaging the application with PyInstaller. Documentation on how to set up the development environment and use the application is provided in the repository's README file and package requirements are added in a `requirements.txt` file.

Figure 6.4 shows a screenshot of the WebVision GUI, rendering the EnVision spacecraft in orbit around Venus, with GUI elements annotated in red. In the top left, view controls are provided; these support changing the anchor point between EnVision and Venus (spacecraft- or orbit-views), as well as the camera following modes. The camera can be set to "inertial" (i.e. non-auto-rotating) or body-fixed (i.e. rotating with EnVision's/Venus's orientation). Some options are provided to show/hide visual elements. Below the view controls, the visible ground stations are indicated with a red or green dot. Earth occultation is indicated by greying out the text.

In the top right, the time controls are shown. Similar to Cosmographia, each time unit (YYYY, MM, DD, hh, mm, ss) can be changed individually by the arrows above, allowing the user to navigate to a specific epoch. The play/pause button and speed control buttons are placed below the time picker. The Simulation time scaling buttons scale the simulation speed by a factor of ±10x. Below the time controls, the instrument toggles are shown. Each instrument FOV can be shown or hidden by clicking the checkboxes. A toggle is present to automatically show/hide instruments based on the planned operational modes from the SciOps manager; their activity (like the ground stations) is indicated by green/red dot. Below the instrument toggles, the pointing mode dropdown is shown. The user may select automatic (SciOps planned) pointing, or select a custom one. On the right, the planned operational mode is indicated regardless of the overwritten mode.

Additional elements were added for outreach and practical purposes. On the bottom right, a button is added to generate a high-resolution render of the current scene. This allows users to create images at specific

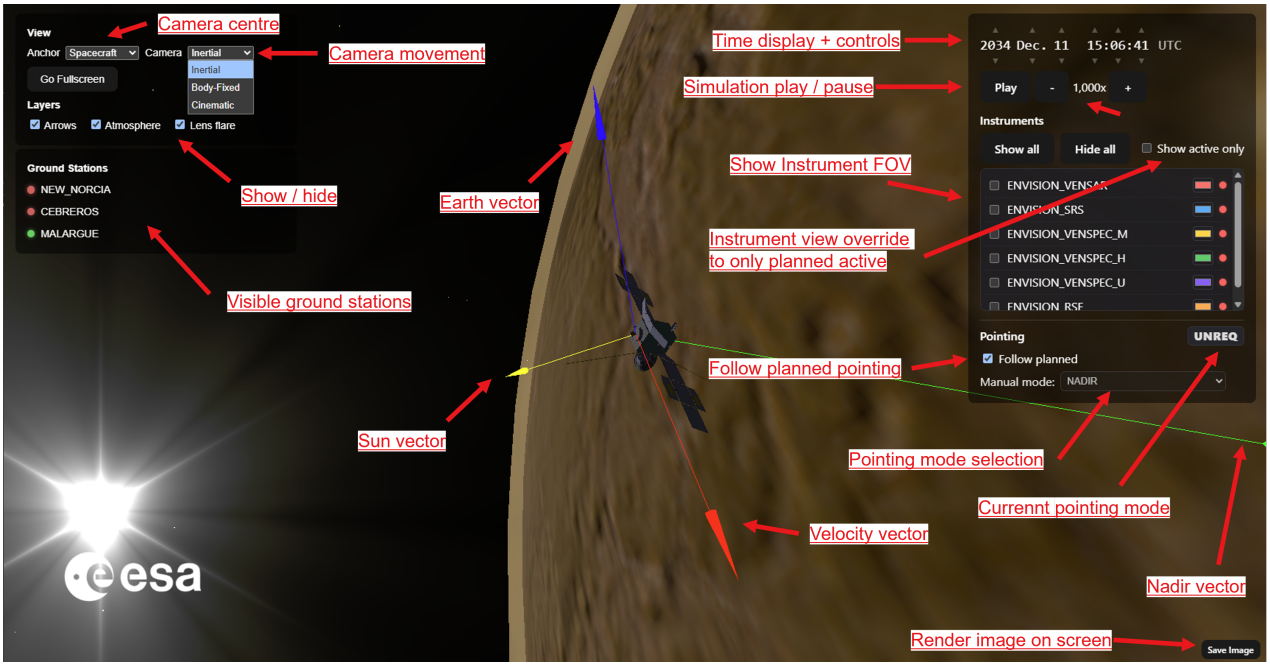


Figure 6.4: WebVison GUI showing the EnVision spacecraft in orbit around Venus.

observations/cases for presentations, documentation or outreach activities. Some additional outreach elements added were: user customization of the FOV colours, a "cinematic" camera mode to slowly pan around the spacecraft, the ability to hive all clutter and only show the render. The simulation runtime is only limited by the mission's duration; the WebVision application was run for a full day during the ESTEC open day, helping to explain different mission phases, instruments and timescales to visitors. No memory overloads or performance issues were observed during this continuous runtime.

In the following subsections, the requirements compliance of the application is discussed, followed by an assessment of the application and potential future work.

6.3.1. Requirements Compliance

In this subsection, all established requirements from section 6.1 will be evaluated against the developed WebVision application. Every requirement from Table 6.1 and 6.2 will get a "COMPLIANT", "PARTIALLY COMPLIANT" or "NON-COMPLIANT" rating, with a brief reasoning. This requirements' compliance coverage is included in this chapter, rather than in the Conclusions of this thesis (chapter 7). In this report, the WebVision application serves as a deliverable to address an identified need, as well as a demonstration of PAS3's implementation flexibility; it is treated as a separate entity from the simulation toolkit itself (its requirements from section 4.3 *will* be covered in the including chapter).

Table 6.4: Requirements compliance coverage tabe for the WebVision application.

ID	Compliance	Justification
VIS-F-1.1	COMPLIANT	Displayed in Figure 6.4, verified attitude model in subsection 5.3.3.
VIS-F-1.2	COMPLIANT	Displayed in Figure 6.4, SPICE frame correctly implemented with PAS3.
VIS-F-1.3	PARTIALLY COMPLIANT	Spacecraft-anchored views have correct sun direction from PAS3-backed API, but same relative orientation is given when anchored at Venus. Not patched due to time constraints.
VIS-F-1.4	COMPLIANT	Displayed in Figure 6.4.
VIS-F-1.5	COMPLIANT	Displayed in Figure 6.4.
VIS-F-1.6	PARTIALLY COMPLIANT	WebVision displays the current operational mode, but a timeline of past and future operations is not included yet.

ID	Compliance	Justification
VIS-F-1.7	COMPLIANT	Displayed in Figure 6.4.
VIS-F-1.8	COMPLIANT	Displayed in Figure 6.4.
VIS-F-1.9	COMPLIANT	Logarithmic depth buffer was applied in the Three.JS renderer to support true scales of rendered objects.
VIS-F-2.1	COMPLIANT	Time controls displayed in Figure 6.4.
VIS-F-2.2	COMPLIANT	View controls displayed in Figure 6.4.
VIS-F-2.3	COMPLIANT	Pointing controls displayed in Figure 6.4.
VIS-F-2.4	COMPLIANT	Instrument controls displayed in Figure 6.4.
VIS-F-3.1	*REJECTED*	The need for this requirement was filled by the development of EnVision subsystem models in chapter 5. The analysis and visualization concerns were separated.
VIS-NF-1.1	COMPLIANT	Executable application was distributed and launched on user's machines without the need for additional supporting software installations.
VIS-NF-2.1	COMPLIANT	Modular approach shown in section 6.2.
VIS-NF-2.2	PARTIALLY COMPLIANT	Activities in the Earth-Observation directorate are ongoing to implement WebVision for the Sentinel-2 mission. Effort was put in compatibility, but hidden technical debt appeared, showing EnVision-specific elements in the application. While the application does not disallow adaptation to other missions, a more organized agnostic interface is desired to satisfy this requirement fully.
VIS-NF-3.1	COMPLIANT	DOM elements were added to produce specific views, screen renders and visuals for outreach and documentation purposes (see introduction of section 6.3).

As seen in Table 6.4, WebVision application meets most of the established requirements from section 6.1. The functional requirements are mostly satisfied, with only two partially compliant requirements (**VIS-F-1.3** & **VIS-F-1.6**), easily fixable in existing features. One requirement (**VIS-F-3.1**), which covered the inclusion of simulation data plots in the GUI, was rejected. This feature was deprioritized during development of the EnVision subsystem models in chapter 5. Mission analysis and sizing is preferably done over a longer period in fast succession of time steps. The resulting plots can then be contextualized by navigating to the points of interest in time. Optionally, the dynamic generation of plots could be added in future work when more robust interfacing is implemented. The partially compliant requirement (**VIS-NF-2.2**) arise due to the lack of interfacing abstractions in the application. This will be addressed in future development work.

6.3.2. Application Assessment and Future Work

The WebVision application was successfully deployed in the EnVision project, giving the systems engineers a valuable tool to understand the mission and its operational environment more intuitively. Its deployment was received positively, with team members being able to see what was previously only described in documents and presentations. In the short time after deployment, WebVision showed its practical use by showing particular spacecraft orientations in certain mission phases, calling for a deeper analysis for mitigating potential power concerns (described in subsection 5.5.3). The interactivity in time selection and spacecraft pointing enabled not only the identification of such occurrences, but also supported discussions surrounding it between payload and power management.

The application showed its outreach potential during the ESTEC open day event, as well as interest from an EnVision documentary being produced. Additionally, other departments have shown interest in adapting the application for their own applications (Sentinel-2 mission telemetry visualization and CDF study mission design support). The accessibility of web-based 3D visualization makes the WebVision framework very attractive for these use cases. The current progress of the Sentinel-2 adaptation is shown in Figure 6.5, where the Sentinel-2 spacecraft model is rendered in orbit around Earth.

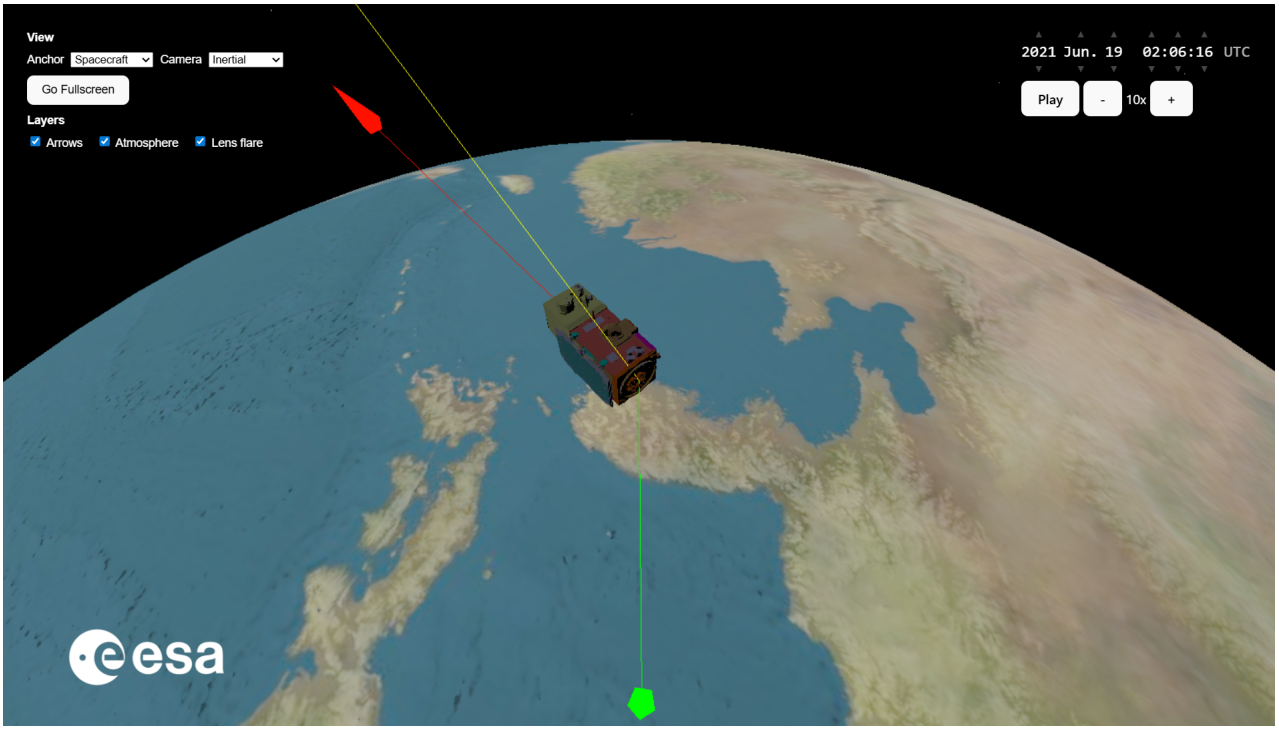


Figure 6.5: WebVision adapted for Sentinel-2 mission visualization (work in progress by Marta Rozycka).

There are several improvements and extensions that can be made to the WebVision application in future work. The following list summarizes some of these potential improvements:

- Implement the timeline of planned operational modes in the GUI (**VIS-F-1.6**).
- Reworking the WebVision repository to a bare framework, with mission-specific configuration files and assets separated from the core application code with clear interfaces (**VIS-NF-2.2**).
- Creating an agnostic interface for simulation data plots (**VIS-F-3.1**).
- Integrating WebSockets for real-time data streaming from PAS3 for multi-client serving.
- Reworking the *SpiceyPy* package to support asynchronous calls, improving performance and stability when multiple clients are connected.

The takeaway from this chapter is the proof of concept of using PAS3 as a generalized simulation backend toolkit for a wide range of applications. While chapter 4 and 5 focused on the standard methodologies of building analysis scripts and models in a code environment, this chapter showed how PAS3 integration can be extended deeper into custom applications. This shows the approach of "toolkit for user innovation" described in section 3.3, where PAS3 does not offer the tool, but rather the building blocks to create the custom tool needed by the user. While Python packages exist with built-in mission analysis models and visualization (e.g. *PASEOS* [39] and *Basilisk* [38]), PAS3's approach is to offer developers the tools to create their own applications instead. While this requires more initial effort, and for many projects existing tools may be sufficient, the toolkit approach fills a niche for projects with specific needs that cannot be satisfied by off-the-shelf software.

7

Evaluation & Conclusions

The PAS3 toolkit was developed over a 9-month period inside the EnVision project team at the European Space Agency's ESTEC. During this time, an assessment was made on the contemporary needs of the systems engineer in interplanetary scientific missions in the agency. Based on this, a research objective and a set of research questions were formulated to guide the development of the solution (chapters 2 and 3). Based on current Modelling & Simulation (M&S) paradigm shift from monolithic software applications to user-made ad-hoc solutions, a methodology was proposed to develop a toolkit that would enable the systems engineer to more easily create their solutions. The toolkit development methodology was described in chapter 4, defining its scope, requirements, architecture and verification approach.

Considering PAS3's *solution space* being the creation of models in an ad-hoc System Simulator (SS) context, rather than their numerical results, the process of implementing the toolkit with such approach was covered in chapters 5 and 6, with ESA's EnVision mission as a case study. The PAS3 toolkit is to be evaluated by its implementation process, and the resulting SS adherence to the defined System Simulator Concept (SSC) described by T. Nemetzade [21].

This concluding chapter aims to reflect on the developed PAS3 toolkit and evaluate its results. The compliance of the toolkit with the defined requirements in section 4.3 is assessed in section 7.1. section 7.2 evaluates the resulting implementation in the EnVision case study (both the resulting SS and Visualization app), and answers the research questions from section 3.3. Finally, section 7.3 provides an overview of the limitations of the developed toolkit, and suggests possible future work to extend its functionality and usability.

7.1. Requirements Compliance

In this section, the requirements set in section 4.3 will be reviewed. The assessment of whether they have been fulfilled is justified by referencing to the relevant chapters and sections that display their compliance. Based on the assessment, each requirement is classified as "COMPLIANT", "PARTIALLY COMPLIANT", or "NON-COMPLIANT". The resulting compliance table is shown on the next page, in Table 7.1.

PAS3 provides an object-oriented framework to implement SPICE-based data (and custom models) into a user-controlled simulation environment. The following class blueprints were created:

`pas3.Simulation`:	Main object loading SPICE kernels, while storing and controlling the simulation environment (time and global settings). Updates the assigned actors' states at time changes.
`pas3.Spacecraft`:	Spacecraft object representing a spacecraft actor in the simulation. Retrieves its state (and attitude if applicable) from SPICE kernels.
`pas3.CelestialBody`:	Celestial body object representing any celestial body defined in loaded SPICE kernels.
`pas3.GroundStation`:	Parent class for ground station objects, with `pas3.SpiceGroundStation` and `pas3.CustomGroundStation` subclasses. Assignable to individual spacecraft.
`pas3.Geometry`:	Parent class for geometry objects representing infinite geometrical regions in space fixed to the spacecraft. Allows for the creation of SPICE-based instruments (`pas3.SpiceInstrument`) or user-defined geometries (`pas3.CustomInstrument`, `pas3.AssembledGeometry`, `pas3.AssembledInstrument`).

Table 7.1: PAS3 toolkit requirements compliance table.

ID	Compliance	Justification
TK-F-1.1	**COMPLIANT**	Covered in subsection 4.4.1 with *Simulation* class initialization in Listing 13 and demonstrated thoroughly in chapter 5.
TK-F-1.2	**COMPLIANT**	Covered in subsection 4.4.1 with provided *Simulation* class methods listing: • loaded SPICE kernels (`list_loaded_kernels()`, or automatically at *Simulation* initialization) • loaded SPICE SPK actors (`list_loaded_spk_actors()`) • loaded SPICE CK actors (`list_loaded_ck_actors()`) • loaded SPICE reference frames (`list_loaded_frames()`) demonstrated in Listing 3.
TK-F-1.3	**PARTIALLY COMPLIANT**	Covered in subsection 4.4.1 with provided *Simulation* class method `get_spk_coverage()` extracting simulation coverage automatically at initialization of the *Simulation* object. Partial compliance due to CK and dynamic frame (PCK) coverage not being considered yet.
TK-F-2.0	**COMPLIANT**	Covered in subsection 4.4.1 with the *Simulation* class and demonstrated in chapter 5.
TK-F-3.0	**COMPLIANT**	PAS3 supports configurable user-made models in the simulation environment by the use of the *Simulation* object's `add_custom_model()` method to be used in user's configuration architecture (see section 5.2). Demonstrated in Listing 22.
TK-F-4.0	**COMPLIANT**	Supported SPICE-defined PAS3 objects are: • *Spacecraft* (subsection 4.4.4) • *CelestialBody* (subsection 4.4.2) • *SpiceGroundStation* (subsection 4.4.3) • *SpiceInstrument* (subsection 4.4.5) All objects' usage demonstrated in chapter 6 and chapter 5 (except *SpiceGroundStation*). All are verified in section 4.5.
TK-F-5.0	**COMPLIANT**	Supported non-SPICE-defined PAS3 objects are: • *CustomGroundStation* (subsection 4.4.3) • *Geometry* (subsection 4.4.5) • *CustomInstrument* (subsection 4.4.5) • *AssembledGeometry* (subsection 4.4.5) • *AssembledInstrument* (subsection 4.4.5) Usage for all is demonstrated in chapter 5 (except *CustomGroundStation*) and verified in section 4.5.
TK-F-6.1	**COMPLIANT**	PAS3 *Simulation* object updates assigned object's state, attitude, visibility, and orbital parameters automatically using the core `advance_time()` and `set_time()` methods (see subsection 4.4.1). Implemented across chapters 5 and 6 and verified in sections 4.5 and 5.3.3.

ID	Compliance	Justification
TK-F-6.2	**PARTIALLY COMPLIANT**	Supported (optional)post-initialization configuration settings: *Spacecraft* object: • NAIF body-fixed-frame ID (`set_spacecraft_platform_id()`) • its central body (`set_central_body()`) *CelestialBody*: • gravitational parameter (`set_gravitational_parameter()`) • body-fixed reference frame (`set_body_fixed_frame()`) *GroundStation*parent class: • visibility mask (`set_visibility_mask()`, `reset_visibility_mask()` and `set_minimum_elevation_angle()`). *Geometry* parent class: • direction/shape (`set_vertices()`, `set_half_angle()`, `rotate()`) • position (`move_to()`) Global *Simulation* settings: • default reference frame (`set_default_frame()`) • solar constant (`set_solar_constant()`) • occultation tolerance (`set_occultation_tolerance()`) • start time (`set_start_time()`) • end time (`set_end_time()`) Partial compliance for missing global SPICE aberration and light time correction settings and adding additional occultating bodies other than the flow down of central bodies.
TK-NF-1.1	**NON-COMPLIANT**	Non-compliance: absence of package wiki or "Read-the-Docs" page.
TK-NF-1.2	**COMPLIANT**	Provided class properties are expressed in the simulation's default frame, their equivalent getter functions require a reference frame as input argument. Methods are long, but descriptive by design (see Appendix A). All initialization functions of classes have Error/Warning messages when configuration fails. Input parameters of methods in Appendix A are type-hinted.
TK-NF-1.3	**COMPLIANT**	Input arguments are minimized, using the global configuration methods in **TK-F-6.2**.
TK-NF-1.4	**COMPLIANT**	Tutorial notebooks are provided inside the "examples" directory in the PAS3 git repository. A summary of all object methods is provided in Appendix A.
TK-NF-2.0	**COMPLIANT**	No integrated simulation run methods are provided. Simulation runtime is controlled fully by the user and requires considerable involvement from the user. Most apparent in Listing 35.
TK-NF-3.0	**COMPLIANT**	PAS3 is written in Python. Its OOP architecture is described in section 4.4 and demonstrated in section 5.2. The interface is visible throughout chapter 5.
TK-NF-4.0	**COMPLIANT**	No EnVision-specific models were included. The solution space from Figure 4.2 was followed. Tutorial notebooks in the repository use different missions' SPICE kernels as examples. Verification campaigns in sections 4.5 and 5.5.4 were performed with custom-tailored spice kernels.

Their attributes, properties and methods allow for straightforward and elegant integration of a PAS3 mission configuration file within the mission-specific SS architecture, complying with most of the defined functional and non-functional requirements.

From Table 7.1, it is clear that some requirements were only partially fulfilled, and one was not fulfilled at all. The partial compliance of **TK-F-1.3** (SPICE kernel coverage assignment utility) is due to CK- and PCK-equivalent coverage functions to the present SPK one were not yet implemented. This could cause the *Simulation* object to not fully restrict the simulation time to shorter configured intervals of CK (attitude) or PCK (dynamic planetary body frame) data. **TK-F-6.2** (configuration settings) is partially compliant due to two absent settings: configuring the aberration and light time correction flag to be used in the encapsulated SPICE functions, and adding additional occulting bodies to a spacecraft's object line-of-sight checks. Strictly speaking, requirement **TK-NF-1.1** (documentation provision) is non-compliant, as no dedicated user-manual or wiki was included yet. One could argue its necessity, considering this report's Appendix A and the provided tutorials in notebook format (**TK-NF-1.2**).

7.2. Discussion

The PAS3 toolkit was implemented in a System Simulator (SS) for ESA's EnVision mission as a case study. Its architecture was described in section 5.2. The intended use of PAS3 is to construct a main configuration file of the mission, which holds the relevant configured PAS3 objects, as well as mission-specific custom models (for example: SciOps manager for EnVision). This file is not intended to perform any simulations or analyses by itself, but rather to be imported in the discipline-specific sub-scripts that perform the actual simulations and analyses. For the EnVision case study, three sub-scripts were created: a thermal modelling module (implementing two thermal models), a power sizing module (implementing a single power model), and an attitude model (modelling a steady-state approximation of the spacecraft attitude during science operations). Each model can be imported to the main configuration file itself to support cross-disciplinary connectivity (as done for the attitude model).

As mentioned earlier, the PAS3 toolkit will be evaluated based on its ability to create SS's that adhere to the System Simulator Concept (SSC) principles. This evaluation aims to eliminate subjective bias in the assessment, acknowledging that the toolkit was developed and implemented in one mission (EnVision), in one mission phase (B2). Note that when referring to "the SS", the EnVision System Simulator platform (i.e. the collection of configuration file and analysis sub-scripts) from chapter 5 is meant. Below, every SSC principle is revisited, and the compliance of the resulting SS to each principle is discussed.

Principle I: Characteristics. *"The simulators built according to the SSC are designed to. . .*

- . . . *be lean & adaptive"* – Additional PAS3 objects and custom models can be added as needed to the established configuration file structure.
- . . . *be transparent"* – Imported PAS3 classes are fully built in Python, allowing the user to inspect the underlying logic from the IDE they are using. (no inherent C++ or Fortran API black-boxes except for the SPICE toolkit itself). Transparency in the sub-scripts itself is increased by the readability of their integrated models and simulation loops (a common baseline of mission objects is imported).
- . . . *be easily manageable"* – The configuration file structure allows for straightforward addition/removal of objects and models, managing the down-flow of objects in the sub-scripts.
- . . . *be system-focussed, not limited by single disciplines"* – The PAS3 objects are generic and can be implemented in any discipline's model needing a mission simulation backend (demonstrated for thermal and power sizing in chapter 5).
- . . . *include only essential models and functionalities"* – The developer of the SS is in full control of which models and functionalities to include.
- . . . *add functionality based on growing user needs"* – The modular approach in Figure 5.5 ensures models and configuration can grow as the mission matures, without the need to change framework.

Principle II: Context of Use. *"The primary user of the SCC-based simulator is a systems engineer who . . .*

- . . . *has limited experience with M&S tools"* – Separation of concerns: once the configuration is set up, the developer is limited by their Python experience in implementing their discipline-specific models. The developer does not need to have experience modelling spaceflight dynamics, frame transformations, or SPICE usage.
- . . . *works in a highly dynamic environment, with limited familiarization and development time"* – Descriptive OOP architecture and example tutorials allow for quick familiarization and the separation of concerns allows

for development time only going into the discipline-specific model.

Principle III – a: Simulator Development Approach. *"The simulator development must allow for. . .*

. . . *continuous implementation of user needs"* – The modular sub-script architecture inherently supports this.

. . . *clear and easy interfaces"* – The user is in full control of the applied interfaces of their models.

Principle III – b: Simulator Scope. *"The SSC-based simulator must provide. . .*

. . . *a complementing function to advanced detailed discipline-specific simulators"* – Demonstrated in the thermal model (ESH is modelled for thermo-optical degradation estimations to be implemented in detailed thermal balance models) and power model (geometrical factors and impact are sized to complement detailed SAA ecosim models).

. . . *allow for modelling cross-connections of subsystem models"* – The custom attitude model in section 5.3 showcases cross-connectivity with the sub-script models by being configured as an inherent custom model inside the PAS3 *Simulation* (any model can be cross-configured like this).

. . . *focus on fundamental system-level models instead of detailed single-discipline models"* – While the models in the sub-scripts are discipline-specific, they are considered more system-level due to their reliance on the whole mission configuration. Integrating a detailed single-discipline model is not strictly forbidden, but would not necessarily reflect the PAS3 configuration use case.

Principle III – c: Scheduling of SSC Implementation. "It is recommended to start SSC implementation in the early life cycle stages" – not applicable to the EnVision case study.

It shows how the resulting SS adheres to the SSC principles, showcasing the toolkit's projected ability to develop effective system simulators. This deemed the research objective successfully achieved. With the newly gained insights from the development and evaluation of the PAS3 toolkit, the **research questions** from section 3.3 can now be confidently answered:

> *"What general functionalities must the toolkit offer, in order for systems engineers to develop effective system simulators, and which ones should it aim to exclude?"*

The answer to this question is dependent on the context of implementation for the toolkit. In the case of PAS3, the context is in ESA interplanetary science missions' implementation phase; more specifically, within the mission's project team. At this stage, the scientific orbit is mostly defined, and is provided by the MOC and SOC in SPICE kernel format. This orbit is taken as-is to perform system-level platform and mission performance analyses. SPICE kernels also hold additional data for celestial bodies (ephemerides, reference frames), spacecraft (attitude, reference frames), instrument FOV geometries and ground station reference frames. These are also taken as reference input data for the system simulator.

Following the SSC principles, the toolkit must allow for creating SS's with accessible agility and flexibility. While SPICE data can be altered, the SPICE utility tools and kernel formats need considerable familiarization (non-conformance with SCC). Therefore, for all but the *Spacecraft* and *CelestialBody* objects, custom counterparts are provided to allow the user to create their own objects (instruments, geometries, ground stations) circumventing SPICE definitions entirely. Planetary bodies' ephemerides are not suspected to be changed by the end user and are accessible on NAIF's servers[1] while custom spacecraft trajectories (for verification) can be created with SPICE's `spkwrite` utility (keeping its customizability optional, but not natively supported). Spacecraft attitude is an optional SPICE-based input depending on the mission development maturity.

With the aforementioned objects initialized, the question becomes which methods and functionalities they should (not) provide. The design philosophy taken was to support ad-hoc model development; the toolkit should therefore provide only the essential simulation building blocks, while leaving subsystem models to be implemented by the user. All higher-level models that may be open to interpretation (for example: albedo models, link budgets, power balances, attitude steering, etc.) are not included. Only strictly unambiguous[2] geometrical and physical properties are provided by the objects, with the view of providing simulation-tied parameters offering separation of concerns in the SS architecture. Examples are: spacecraft position/velocity, Keplerian orbit parameters, vector frame transformations, instrument intersections on planetary surfaces, visibility of ground stations and celestial bodies, incidence angles on geometries, as well as global simulation settings (solar constant, occultation tolerances, default reference frame) and simulation time settings.

[1] https://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/

[2] Unambiguous in the sense that the context of implementation allows certain inputs to be taken as-is (like planetary ephemerides, planetary shapes, instrument geometries, spacecraft orbits and possibly attitude).

"What are the criteria for assessing the effectiveness of a system simulator developed with the toolkit?"

It was recognized that the proposed toolkit would be implemented and evaluated in a single mission case study in B2 phase. The effectiveness of PAS3 and its resulting SS would be apparent in this specific context, but not necessarily generalizable to other missions or phases. Therefore, a more generalized evaluation approach was required (hence the research question).

The chosen approach follows directly from the literature study, which identified the System Simulator Concept (SSC) as a promising reference framework for assessing system-level M&S tools in space projects. Nemetzade's SSC formulates a structured set of simulator characteristics, context of use considerations, and implementation principles that distinguish effective system-oriented simulators [21]. Because the SSC was explicitly designed to evaluate whether a simulator supports interdisciplinary systems engineering activities across phases, it provides a mission-agnostic basis for assessing the effectiveness of the toolkit in supporting the creation of SSC-adhering system simulators. The principles are listed in subsection 3.2.2 and the evaluation was discussed above.

"What design choices are key in allowing the toolkit to be implemented in multiple interplanetary mission projects?"

The key design choice in assuring multi-mission applicability is following the "toolkit for user innovation" strategy. This strategy focuses on providing the necessary building blocks for the user to create their own ad-hoc solutions, rather than providing built-in solutions that may not fit every mission's needs. The following design choices were key in following this strategy:

- Providing a standardized initialization data standard: in this case, SPICE kernels were used, which are commonly supported across scientific planetary missions.
- Providing custom initialization methods when (SPICE) data is not yet available.
- Avoiding the inclusion of built-in analysis models (see first research question).
- Including unambiguous object methods that inherently do not offer analysis, but offer model inputs instead.

In essence, the *mission-agnosticism* of the toolkit was ensured by not assuming the analysis needs of one mission translate to others. Many object methods (in Appendix A) were implemented in PAS3 that were unused in the EnVision case study, but are sufficiently unambiguous to be conceivably useful in future models. Following this approach, additional class methods can be added in future toolkit versions as the need arises, without breaking existing implementations.

"What design choices are necessary to ensure support across multiple mission phases and disciplines?"

In order to ensure multiphase and multidisciplinary support, the resulting SS of the toolkit must be inherently modular. This allows for a growing *working version* of the SS with (new) discipline-specific models evolving as the mission matures. The toolkit may not assume the necessary modelling methods and fidelities, and only support their creation. Failing in this will lead to a rigid SS architecture that may not be adaptable to emerging needs.

The approach shown in Figure 5.5 of creating a main configuration file, holding the mission's objects and custom models, that can be imported in discipline-specific sub-scripts, supports this. Analysis sub-scripts can be created depending on the needs of the discipline, at the required fidelity of the development phase. These sub-scripts can grow, new ones can be added, and new objects can be configured in the configuration without having to change the SS framework itself. The modular nature of the configuration file approach allows for even broader implementations outside Python scripts or notebooks in external applications (demonstrated in chapter 6), showing the flexibility the design choice offers in its solution space.

The multidisciplinary support of the proposed configuration/sub-script strategy is apparent in chapter 5. Generalized objects like the half-space objects were able to support the creation of a thermal model (spacecraft faces), a power model (solar panels), and an attitude model (pointing constraints' identification).

7.3. Future Work

While the PAS3 toolkit shows promising results, some limitations were identified in the current version of the package. These limitations could be addressed in future work to further enhance the toolkit's functionality and usability. The following list summarizes the main limitations and suggests possible future work:

- **SPICE DSK Functionality:** Currently, the PAS3 toolkit does not support Digital Shape Kernel (DSK) functionality for detailed surface representations of celestial bodies. Future work could involve integrating DSK support to enhance the accuracy of surface-related analyses.
- **Generalized SPK Actor Class:** The currently provided SPK-based actor classes are the *Spacecraft* and *CelestialBody*. A generalized SPK actor class could be implemented to allow for moving objects to be defined that do not fit this category (for example: rovers, landers, regions of interest, . . .).
- **Space-Based Telescopes:** The toolkit has not been assessed for missions involving Lagrange-point orbiting spacecraft, such as space-based telescopes (possibly breaking when no shape is identified for the central body). Telescopes are a large part of ESA's scientific missions, and future work could focus on adapting the toolkit to accommodate such mission profiles.
- **Ground Station Visibility Masks:** the current implementation of ground stations' visibility masks does not differentiate between uplink and downlink visibility. Acquisitions of signal for downlink may start earlier than uplink windows. Future work could involve implementing separate visibility masks for uplink and downlink scenarios, and update the *Spacecraft* object's methods accordingly.
- **Spherical Approximations:** Currently, most *Geometry* object intersection methods assume spherical approximations of celestial bodies. Future work could involve implementing ellipsoidal or DSK-based intersection methods to increase accuracy for missions where this is critical.
- **Earth-visibility:** Currently, Earth visibility is determined based on occultation conditions, while in reality, Earth-communication is often not possible during periods nearing solar conjunctions. This should be a configurable setting in the *Spacecraft* object in future work.
- **Additional Simulation Settings:** Some simulation settings, such as aberration corrections and occulting bodies, are currently not configurable in the *Simulation* object.
- **Occultations:** In the current version of PAS3, boolean occultation checks are provided based on a single tolerance value. Future work could involve implementing a fraction-based output for percentage of (partial/annular) occultations. This could increase some custom models' accuracy.
- **Documentation:** While the package includes method summaries and tutorials, a dedicated wiki would increase accessibility significantly.
- **Automatic Simulation Coverage Extraction:** CK and PCK coverage extraction methods could be implemented to complement the existing SPK coverage method in the *Simulation* object.
- **Asynchronous Simulations:** Right now, SpiceyPy inherently does not support asynchronous calls. Future work could investigate the possibility of implementing thread-safe asynchronous calls to SPICE functions to run multiple simulations in parallel.
- **Better Mission Phase assessment:** The toolkit was only implemented in the EnVision B2 phase. While the SSC principles gave an indication of the toolkit's effectiveness in future phases, a more thorough assessment in later mission phases would be needed to say this with certainty.
- **Logging:** In the current PAS3 version, print statements are used to inform the user of configuration steps and warnings/errors. Future work could switch over to a logging module to allow better information control.

Interest was expressed from different ESA projects, ranging from assessing full SS implementations using PAS3 to simply using it as a more accessible interface to SPICE data. It is expected that more shortcomings and possible improvements will be identified as more users apply the toolkit in their projects.

References

[1] *Space engineering: System engineering general requirements*, ECSS-E-ST-10C Rev.1, European Cooperation for Space Standardization, Noordwijk, The Netherlands, 2017.

[2] J. Asquier et al., 'CHEOPS-first ESA small scientific mission,' in *7th European Conference for Aeronautics and Space Sciences (EUCASS)*, 2017. DOI: 10.13009/EUCASS2017-592

[3] H. Joumier, C. Freaner, B. Bitten and D. Emmons, 'Comparison of ESA and NASA acquisition approaches and their potential effects on science mission development duration and schedule change,' in *2012 ISPA/SCEA Joint International Conference & Training Workshop*, 2012. Accessed: 16th Jun. 2025. [Online]. Available: https://ntrs.nasa.gov/api/citations/20120009487/downloads/20120009487.pdf

[4] Z. Szajnfarber and A. L. Weigel, 'Innovation dynamics of large, complex, technological products in a monopsony: The case of ESA science missions,' in *2007 Atlanta Conference on Science, Technology and Innovation Policy*, 2007, pp. 1–13. DOI: 10.1109/ACSTIP.2007.4472887

[5] J. L. Alvarez, H. P. de Koning, D. Fisher, M. Wallum, H. Metselaar and M. Kretzenbacher, 'Towards a definition of best practices for model based systems engineering in European space agency projects,' in *2018 AIAA SPACE and Astronautics Forum and Exposition*, American Institute of Aeronautics and Astronautics Inc (AIAA), 2018. DOI: 10.2514/6.2018-5327

[6] European Space Agency, 'ATHENA: Assessment of an X-ray telescope for the ESA Cosmic Vision program,' CDF Study Report CDF-150(A), 2014.

[7] M. Bandecchi, B. Melton and F. Ongaro, 'Concurrent engineering applied to space mission assessment and design,' in *ESA Bulletin number 99*, B. Battrick, D. Guyenne and D. Danesy, Eds. Noordwijk, The Netherlands: European Space Agency, 1999, pp. 34–40.

[8] V. Cripps. 'Phases of an ESA hardware project explained,' Accessed: 16th Jun. 2025. [Online]. Available: https://www.space.irfu.se/seminars/IRFU_seminars_20181004.html

[9] D. Naylor et al., 'SPICA: the next observatory class infrared space astronomy mission,' in *Canadian Long Range Plan for Astronomy and Astrophysics White Papers*, vol. 2020, Oct. 2019, p. 49. DOI: 10.5281/zenodo.3825568

[10] I. Ferreira et al., 'ATHENA reference telescope design and recent mission level consolidation,' SPIE – International Society for Optics and Photonics, 2021, p. 31. DOI: 10.1117/12.2594443

[11] European Space Agency. 'Building and testing spacecraft,' Accessed: 16th Jun. 2025. [Online]. Available: https://www.esa.int/Science_Exploration/Space_Science/Building_and_testing_spacecraft

[12] *Space project management: Organization and conduct of reviews*, ECSS-M-30-01A, European Cooperation for Space Standardization, Noordwijk, The Netherlands, 1999.

[13] F. D. Bruin, 'Aeolus-CalVal in the context of the operational phase E1,' in *Aeolus Cal/Val Workshop*, 2015.

[14] A. Kossiakoff, W. N. Sweet, S. J. Seymour and S. M. Biemer, *Systems Engineering Principles and Practice*, 2nd ed., A. P. Sage, Ed. John Wiley & Sons, 2011.

[15] A. Parmar et al., *ESA Science Programme Missions: Contributions and Exploitation* (ISSI Scientific Report Series), 1st ed., A. Parmar, Ed. Bern, Switzerland: Springer Cham, 2024, vol. 18. DOI: 10.1007/978-3-031-69004-4

[16] M. Ashman et al., 'Science planning implementation and challenges for the ExoMars trace gas orbiter,' in *15th International Conference on Space Operations*, Marseille, France: American Institute of Aeronautics and Astronautics (AIAA), 2018. DOI: 10.2514/6.2018-2580

[17] M. Warhaut and A. Accomazzo, 'Venus Express ground segment and mission operations,' in *ESA Bulletin 124*, B. Battrick and B. Warmbein, Eds. Noordwijk, The Netherlands, 2005, pp. 34–37. Accessed: 11th Jul. 2025. [Online]. Available: https://www.esa.int/About_Us/ESA_Publications/ESA_i_Bulletin_i_124_November_2005

[18] A. G. Straume et al., 'EnVision definition study report,' Red Book ESA-SCI-DIR-RP-003, 2023. Accessed: 11th Jun. 2025. [Online]. Available: https://www.cosmos.esa.int/web/envision/links

[19] B. St-Aubin, G. Wainer and F. Loor, 'A survey of visualization capabilities for simulation environments,' in *2023 Annual Modeling and Simulation Conference (ANNSIM)*, 2023, pp. 13–24. Accessed: 22nd Jun. 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10155362

[20] *Space engineering: System modelling and simulation*, ECSS-E-TM-10-21A, European Cooperation for Space Standardization, Noordwijk, The Netherlands, 2010.

[21] T. Nemetzade, 'Characterization and application of user-centred system tools as systems engineering support for satellite projects,' Ph.D. dissertation, Universität der Bundeswehr München, 2019. Accessed: 19th Jun. 2025. [Online]. Available: https://athene-forschung.unibw.de/doc/131535/131535.pdf

[22] R. Blommestijn and C. Honvault, 'European space technology harmonisation dossier: System modelling and simulation tools,' IPC – Technology Harmonisation Advisory Group (THAG), Tech. Rep. ESA/IPC/THAG(2023)11, 2024.

[23] E. von Hippel and R. Katz, 'Shifting innovation to users via toolkits,' *Management Science*, vol. 48, no. 7, pp. 821–833, 2002. DOI: 10.1287/mnsc.48.7.821.2817

[24] V. Génot et al., 'Science data visualization in planetary and heliospheric contexts with 3DView,' *Planetary and Space Science*, vol. 150, pp. 111–130, 2018. DOI: 10.1016/j.pss.2017.07.007

[25] A. Sagristà, S. Jordan, T. Müller and F. Sadlo, 'Gaia Sky: Navigating the gaia catalog,' *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 1070–1079, 2019. DOI: 10.1109/ACSTIP.2007.447288710.1109/TVCG.2018.2864508

[26] A. Bock et al., 'OpenSpace: An open-source astrovisualization framework,' *Journal of Open Source Software*, vol. 2, no. 15, p. 281, 2017. DOI: 10.21105/joss.00281

[27] F. Gregorio. 'Celestia user's guide for version 1.6.1,' Accessed: 2nd Jul. 2025. [Online]. Available: http://www.celestiamotherlode.net/catalog/documentation.html

[28] C. Acton, N. Bachman, B. Semenov and E. Wright, 'An update on NAIF's package of "SPICE" astrodynamics tools,' in *6th International Conference on Astrodynamics Tools and Techniques (ICATT)*, Darmstadt, Germany, 2016. Accessed: 2nd Jul. 2025. [Online]. Available: https://indico.esa.int/event/111/attachments/package

[29] NASA Navigation and Ancillary Information Facility. 'The SPICE concept,' Accessed: 2nd Jul. 2025. [Online]. Available: https://naif.jpl.nasa.gov/naif/spiceconcept.html

[30] M. Costa Sitjà, A. Cardesín Moinelo, D. Frew and C. Arviset, 'Solar system geometry tools with SPICE for ESA's planetary missions,' in *6th International Conference on Astrodynamics Tools and Techniques (ICATT)*, Darmstadt, Germany, 2016. Accessed: 2nd Jul. 2025. [Online]. Available: https://indico.esa.int/event/111/contributions/353/

[31] R. M. Hoofs, D. Koschny and P. van der Plas, 'Planning strategy and supporting tools for the science operations of ESA's planetary science missions,' in *Space OPS 2004 Conference*, American Institute of Aeronautics and Astronautics (AIAA), 2004. DOI: 10.2514/6.2004-196-66

[32] P. van der Plas, B. García-Gutiérrez, F. Nespoli and M. Pérez-Ayúcar, 'MAPPS: A science planning tool supporting the ESA solar system missions,' in *14th International Conference on Space Operations*, Daejeon, South Korea: American Institute of Aeronautics and Astronautics, 2016. DOI: 10.2514/6.2016-2512

[33] M. Pérez-Ayúcar, F. Nespoli, M. Costa and P. van der Plas, 'MAPPS 3D tool: Use for Rosetta science operations,' in *15th International Conference on Space Operations*, Marseille, France: American Institute of Aeronautics and Astronautics (AIAA), 2018. DOI: 10.2514/6.2018-2581

[34] M. Almeida, M. Costa, A. Cardesín and N. Altobelli, 'Solar system operations lab for constructing optimized science observations,' in *SpaceOps 2012 Conference*, Stockholm, Sweden: American Institute of Aeronautics and Astronautics (AIAA), 2012. DOI: 10.2514/6.2012-1295153

[35] M. Costa, N. Altobelli, M. Almeida and A. C. Moinelo, 'The Solar System Science Operations Laboratory: a planetary science lab simulator supporting the JUpiter Icy moons Explorer (JUICE) science operations development,' in *13th International Conference on Space Operations*, Pasadena, CA, USA: American Institute of Aeronautics and Astronautics (AIAA), 2014. DOI: 10.2514/6.2014-1912

[36] Ansys. 'Ansys STK: Software for digital mission engineering and systems analysis,' Accessed: 3rd Jul. 2025. [Online]. Available: https://www.ansys.com/products/missions/ansys-stk

[37] F. J. Hernández González, 'Concurrent engineering methodologies applied to LISA satellite sizing through system and sub-system trade off analyses and system budgets definition,' M.S. thesis, Aerosp. Eng., PoliTo, Torino, Italy, 2019. Accessed: 3rd Jul. 2025. [Online]. Available: https://webthesis.biblio.polito.it/10358/

[38] P. W. Kenneally, S. Piggott and H. Schaub, 'Basilisk: A flexible, scalable and modular astrodynamics simulation framework,' *Journal of Aerospace Information Systems*, vol. 17, pp. 496–507, 9 2020, ISSN: 23273097. DOI: 10.2514/1.I010762

[39] P. Gomez, J. Ostman, V. M. Shreenath and G. Meoni, 'PAseos Simulates the Environment for Operating Multiple Spacecraft,' *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 17, pp. 17 398–17 411, 2024, ISSN: 21511535. DOI: 10.1109/JSTARS.2024.3445506

[40] C. H. Acton, 'Ancillary data services of NASA's Navigation and Ancillary Information Facility,' *Planetary and Space Science*, vol. 44, pp. 65–70, 1 1996. DOI: 10.1016/0032-0633(95)00107-7

[41] Navigation and Ancillary Information Facility (NAIF). 'An overview of SPICE: NASA's observation geometry system for space science missions,' Accessed: 19th Jul. 2025. [Online]. Available: https://naif.jpl.nasa.gov/naif/aboutspice.html

[42] A. Annex et al., 'SpiceyPy: A pythonic wrapper for the SPICE toolkit,' *Journal of Open Source Software*, vol. 5, p. 2050, 46 Feb. 2020. DOI: 10.21105/joss.02050

[43] Navigation and Ancillary Information Facility JPL. 'An overview of reference frames and coordinate systems in the SPICE context,' Accessed: 26th Aug. 2025. [Online]. Available: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/Tutorials/pdf/individual_docs/17_frames_and_coordinate_systems.pdf

[44] D. Giggenbach, 'Mobile optical high-speed data links with small terminals,' in *Unmanned/Unattended Sensors and Sensor Networks VI*, E. M. Carapezza, Ed., International Society for Optics and Photonics, vol. 7480, SPIE, 2009, p. 74800I. DOI: 10.1117/12.831141

[45] Navigation and Ancillary Information Facility JPL. '"high accuracy" orientation and body-fixed frames for the Moon and Earth,' Accessed: 26th Aug. 2025. [Online]. Available: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/Tutorials/pdf/individual_docs/23_lunar-earth_pck-fk.pdf

[46] Navigation and Ancillary Information Facility JPL, 'Instrument Kernel: IK,' 2023. Accessed: 3rd Sep. 2025. [Online]. Available: https://naif.jpl.nasa.gov/pub/naif/toolkit_docs/Tutorials/pdf/individual_docs/25_ik.pdf

[47] G. Bignami, P. Cargill, B. Schutz and C. Turon, *Cosmic Vision, Space Science for Europe 2015–2025*, A. Wilson, Ed. ESA Publications Division, 2005.

[48] Directorate of Science - European Space Agency. 'NEWATHENA,' Accessed: 9th Jun. 2025. [Online]. Available: https://www.cosmos.esa.int/web/athena

[49] L. Colangeli. 'Policy for missions of opportunity in the ESA science directorate,' Accessed: 9th Jun. 2025. [Online]. Available: https://sci.esa.int/web/cosmic-vision/-/59977-missions-of-opportunity

[50] C. Signorini. 'Overview of ESA science missions,' Accessed: 9th Jun. 2025. [Online]. Available: https://agenda.infn.it/event/14775/contributions/26309/attachments/18726/21225/Signorini.ESA.Vulcano2018.pdf

[51] A. Heske, 'Science and payloads for the next decades of ESA's Cosmic Vision program,' *2016 IEEE Aerospace Conference*, pp. 1–13, 2016. DOI: 10.1109/AERO.2016.7500853

[52] European Space Agency. 'Missions in the Cosmic Vision 2015–2025 programme,' Accessed: 16th Jun. 2025. [Online]. Available: https://sci.esa.int/web/home/-/51459-missions

[53] European Space Agency, *Call for a medium-size mission opportunity in ESA's science programme (M5)*, 2016. Accessed: 9th Jun. 2025. [Online]. Available: https://www.cosmos.esa.int/documents/960972/961021/Call+for+M5+missions.pdf/482b4406-e370-41b5-8b8e-ae3b95ba0fd2

[54] European Space Agency, *Call for mission concepts for the large-size "L3" mission opportunity in ESA's science programme*, 2016. Accessed: 9th Jun. 2025. [Online]. Available: https://www.cosmos.esa.int/documents/1137049/1137100/Call_for_L3_Gravitational+waves_missions.pdf/1f0a7dc8-b615-4d09-ba02-9bcda992e4d5

[55] R. Ghail et al., 'EnVision assessment study report,' European Space Agency (ESA), Yellow Book ESA/SCI(2021)1, 2021. Accessed: 11th Jun. 2025. [Online]. Available: https://sci.esa.int/web/cosmic-vision/-/envision-assessment-study-report-yellow-book

[56] D. R. Williams. 'Chronology of Venus exploration,' Accessed: 15th Jun. 2025. [Online]. Available: https://nssdc.gsfc.nasa.gov/planetary/chronology_venus.html

[57] F. J. Meyer and D. T. Sandwell, 'SAR interferometry at Venus for topography and change detection,' *Planetary and Space Science*, vol. 73, no. 1, pp. 130–144, 2012, Solar System science before and after Gaia. DOI: 10.1016/j.pss.2012.10.006

[58] G. G. Schaber et al., 'Geology and distribution of impact craters on Venus: What are they telling us?' *Journal of Geophysical Research: Planets*, vol. 97, no. E8, pp. 13 257–13 301, 1992. DOI: 10.1029/92JE01246

[59] NASA. 'Magellan,' Accessed: 12th Jun. 2025. [Online]. Available: https://science.nasa.gov/mission/magellan/

[60] R. Ghail et al., 'EnVision: Taking the pulse of our twin planet,' *Experimental Astronomy*, vol. 33, no. 2, pp. 337–363, 2012. DOI: 10.1007/s10686-011-9244-3

[61] R. Ghail et al., 'EnVision: Understanding why our most Earth-like neighbour is so different,' M4 Mission Proposal, 2015.

[62] R. Ghail et al., 'EnVision: Understanding why our most Earth-like neighbour is so different,' M5 Mission Proposal, 2017. arXiv: 1703.09010 [astro-ph.EP]. Accessed: 12th Jun. 2025. [Online]. Available: https://arxiv.org/abs/1703.09010

[63] European Space Agency. 'EnVision phase 0 CDF study – internal final presentation,' Accessed: 12th Jun. 2025. [Online]. Available: https://sci.esa.int/web/future-missions-department/-/61027-envision-phase-0-cdf-study-internal-final-presentation

[64] European Space Agency. 'EnVision factsheet,' Accessed: 12th Jun. 2025. [Online]. Available: https://www.esa.int/Science_Exploration/Space_Science/Envision/Envision_factsheet

[65] T. Voirin, 'EnVision reference mission timeline,' ESA, Tech. Rep. ESA-ENVIS-EST-MIS-LI-001, 2024.

[66] L. Bruzzone et al., 'Envision mission to Venus: Subsurface radar sounding,' in *International Geoscience and Remote Sensing Symposium (IGARSS)*, Institute of Electrical and Electronics Engineers Inc., 2020, pp. 5960–5963. DOI: 10.1109/IGARSS39084.2020.9324279

[67] C. O. A. Semprimoschnig, S. Heltzel, M. R. J. v. Eesbeek, J. R. Williamson, A. P. Tighe and A. Polsak, 'The ESA Venus Express mission – from a materials engineering perspective,' in *10th ISMSE & the 8th ICPMSE*, 2006. Accessed: 19th Jun. 2025. [Online]. Available: http://esmat.esa.int/Materials_News/SP-616/poster_sessions/P07_semprimoschnigrev.pdf

[68] B. A. Campbell et al., 'The mean rotation rate of Venus from 29 years of Earth-based radar observations,' *Icarus*, vol. 332, pp. 19–23, 2019. DOI: 10.1016/j.icarus.2019.06.019

[69] B. Spivey, A. Zinecker and J. Smith, 'Using thermal desktop to determine equivalent solar hours on spacecraft surfaces,' NASA Johnson Space Center, Tech. Rep. 20240011881, 2021.

[70] *Space engineering: Thermal design handbook – Part 5: Structural Materials: Metallic and Composite*, ECSS-E-HB-31-01 Part 5A, European Cooperation for Space Standardization, Noordwijk, The Netherlands, 2011.

[71] *Space engineering: Thermal design handbook – Part 1: View factors*, ECSS-E-HB-31-01 Part 1A, European Cooperation for Space Standardization, Noordwijk, The Netherlands, 2011.

[72] *Space engineering: Thermal design handbook – Part 3: Spacecraft Surface Temperature*, ECSS-E-HB-31-01 Part 3A, European Cooperation for Space Standardization, Noordwijk, The Netherlands, 2011.

[73] F. G. Cunningham, 'Earth reflected solar radiation incident upon an arbitrarily oriented spinning flat plate,' Goddard Space Flight Center, NASA Technical Note D-1842, 1963.

[74] K. Sasaki, 'Simplified Earth infrared and albedo coefficient models for spacecraft thermal analysis based on the CERES data products,' *Advances in Space Research*, vol. 75, pp. 1597–1615, 2 Jan. 2025. DOI: 10.1016/j.asr.2024.11.024

[75] B. Danchilla, 'Three.js framework,' in *Beginning WebGL for HTML5*. Berkeley, CA: Apress, 2012, pp. 173–203. DOI: 10.1007/978-1-4302-3997-0_7

A

PAS3 Class Methods and Properties

This appendix provides an overview of the methods and properties of the main classes in the PAS3 toolkit. The descriptions are kept brief; for more detailed information, please refer to the class docstrings in the code documentation at `https://essr.esa.int/project/pas3`. Class attributes are omitted for brevity. They can also be found in the code repository.

A.1. Simulation Class

Table A.1: Simulation base class methods and properties.

Class Function	Description
`load_kernels(. . . )`	Loads one or more SPICE kernels into the kernel pool, lists them, and unpacks their data into the simulation instance. *Input*: path to SPICE kernel.
`is_kernel_loaded(. . . )`	Checks if a given SPICE kernel file is already loaded in the simulation instance. *Input*: file name. *Returns*: boolean.
`list_loaded_kernels()`	Callable to list the loaded spice kernels' types and file names.
`list_loaded_spk_actors()`	Callable to list the loaded SPK actors' NAIF codes and names stored in the simulation instance's SPK data.
`list_loaded_ck_actors()`	Callable to list the loaded CK actors' NAIF codes and names stored in the simulation instance's CK data.
`list_loaded_frames()`	Callable to list the available reference frames' NAIF codes, names and type of reference frame stored in the loaded FK files.
`get_spk_coverage(. . . )`	Returns the time coverage of a given SPK actor loaded in the simulation instance's SPK data. *Input*: NAIF name or code of the loaded SPK actor.
`add_custom_model(. . . )`	Adds an arbitrary function (user-defined model) to the simulation instance, which will be called at each time change. The necessary argument's keyword (if any) is stored and arguments can be included at time changing functions. *Input*: Custom model as a callable function.
`set_aberration_correction_flags(. . . )`	Sets a SPICE abberation correction flag for embedded SPICE state evaluation methods within the simulation instance.
`set_default_frame(. . . )`	Sets the default reference frame for expressing states of *Spacecraft*, *CelestialBody*, *GroundStation* instances assigned to this Simulation object. *Input*: NAIF name or code of the frame.
`set_solar_constant(. . . )`	Sets the solar constant (in W/m^2) for the simulation instance, used in calculating solar flux at distances from the Sun for objects assigned to the simulation. When not configured, the solar constant is set to the default value of 1362 W/m^2. *Input*: Solar constant in W/m^2.

Class Function	Description
`add_progress_bar(...)`	Helper function to add a progress bar using the `tqdm` library based on a given timestep and (optional) end time. Only works with a fixed timestep using the `advance_time(...)` method in the user's external execution loop loop. *Input*: timestep in seconds, optional end time and description for the progress bar.
`remove_progress_bar()`	Removes the progress bar if one is configured.
`set_<...>_time(...)`	Sets the start/end time of the simulation instance. When not specified explicitely, the start and end times of the simulation are set to the shortest coverage covered by all loaded SPK actors. The simulation instance will be set to `active = False` when the time attribute is outside of the configured time coverage. Choose between: `start` or `end`. *Input*: start or end time.
`advance_time(...)`	Advances the simulation time by a given number of seconds (dt), and automatically updates the state of all *Spacecraft*, *CelestialBody* and *GroundStation* instances, assigned to the simulation. Also calls added custom models, by passing their keyword arguments. *Input*: time step in seconds, optional additional arguments for added custom models.
`set_time(...)`	Sets the simulation time to a given epoch given in ephemeris seconds past J2000, and automatically updates the state of all *Spacecraft* and *CelestialBody* instances assigned to the simulation. Also calls custom models, by passing their keyword arguments. *Input*: time to set the simulation to, optional additional arguments for added custom models.

A.2. CelestialBody Class

Table A.2: CelestialBody methods and properties.

Class Function	Description
`update_state()`	Updates the body's state.
`set_gravitational_parameter(...)`	Sets the gravitational parameter (μ or GM) of the body in km^3/s^2. If not set manually, the gravitational parameter is automatically configured from the loaded PCK kernel data (if loaded). *Input*: the gravitational parameter of the body.
`set_body_fixed_frame(...)`	Sets the body-fixed reference frame of the celestial body to a loaded SPICE frame defined in loaded FK kernel data. If not set manually, the body-fixed frame is automatically configured to its default IAU frame. *Input*: NAIF name or code of the body-fixed frame.
`get_position(...)`	Returns the position vector of the body in the specified reference frame centred at its central body at the current simulation time. *Input*: The reference frame to express the position in. *Returns*: The position vector in the given reference frame.
`get_velocity(...)`	Returns the velocity vector (relative to its central body) of the body in the specified reference frame at the current simulation time. *Input*: The reference frame to express the velocity in. *Returns*: The velocity vector in the given reference frame.
`get_position_wrt_sun(...)`	Returns the position vector of the body in the specified reference frame centred at the Sun at the current simulation time. *Input*: The reference frame to express the position in. *Returns*: The position vector in the given reference frame.

Class Function	Description
`get_sun_direction(...)`	Returns the direction vector from the body to the Sun in the specified reference frame at the current simulation time. *Input*: The reference frame to express the direction in. *Returns*: The direction vector in the given reference frame.
`get_orbital_plane_normal_vector(...)`	Returns the normal vector of the body's orbital plane in the specified reference frame at the current simulation time (same direction as the angular momentum vector). *Input*: The reference frame to express the normal vector in. *Returns*: The normal vector in the given reference frame.
`get_spin_axis(...)`	Returns the body's spin axis direction vector in the specified reference frame at the current simulation time. *Input*: The reference frame to express the spin axis in. *Returns*: The spin axis vector in the given reference frame.
`get_object_distance(...)`	Returns the distance to a given *CelestialBody*, or *Spacecraft* object w.r.t. the body's centre in kilometres, at the current simulation time. *Input*: A *CelestialBody* or *Spacecraft* actor. *Returns*: The distance in km.
`from_planetodetic(...)`	Converts planetodetic coordinates (longitude, latitude, altitude) to rectangular coordinates (x, y, z) in the specified reference frame. *Input*: planetodetic longitude (deg), latitude (deg), altitude (km), optional reference frame (if none, default body-fixed frame is used). *Returns*: The position vector in the given reference frame.
`to_planetodetic(...)`	Converts rectangular coordinates (x, y, z) in the specified reference frame to planetodetic coordinates (longitude, latitude, altitude). *Input*: position vector (x, y, z) in the given reference frame, optional reference frame (if none, default body-fixed frame is used). *Returns*: planetodetic longitude (deg), latitude (deg), altitude (km).
`from_planetocentric(...)`	Converts planetocentric coordinates (longitude, latitude, radius) to rectangular coordinates (x, y, z) in the specified reference frame. *Input*: planetocentric longitude (deg), latitude (deg), radius (km), optional reference frame (if none, default body-fixed frame is used). *Returns*: The position vector in the given reference frame.
`to_planetocentric(...)`	Converts rectangular coordinates (x, y, z) in the specified reference frame to planetocentric coordinates (longitude, latitude, radius). *Input*: position vector (x, y, z) in the given reference frame, optional reference frame (if none, default body-fixed frame is used). *Returns*: planetocentric longitude (deg), latitude (deg), radius (km).
`get_approximate_limb_half_angle_in_<...>(...)`	Returns the angle between the vector pointing towards the body's centre and apparent limb of the body viewed from an arbitrary observer position in radians or degrees. "Approximate" because the limb is assumed to be circular (encircling the real elliptical limb of the body). Choose between: `rad` or `deg`. *Input*: The observer distance. *Returns*: The limb half-angle in radians or degrees.
`get_elongation_in_<...>(...)`	Returns the elongation angle of a target body as seen from this Celestial Body in radians or degrees. The elongation angle is defined as the angle between the Sun direction and the target body direction, as seen from this body. Choose between: `rad` or `deg`. *Input*: A target *CelestialBody* (or its NAIF name or code). *Returns*: The elongation angle in radians or degrees.
`body_rotation_matrix`	(property) Returns the body's rotation matrix of the body-fixed frame expressed in the configured default reference frame of the simulation instance at the current simulation time.
`solar_flux`	(property) Returns the solar flux at the body's distance (centre) from the Sun in W/m^2 at the current simulation time.

Class Function	Description
`spin_axis`	(property) Returns the body's spin axis direction vector in the configured default reference frame of the simulation instance at the current simulation time.
`sun_direction`	(property) Returns the direction vector from the body to the Sun in the configured default reference frame of the simulation instance at the current simulation time.

A.3. GroundStation Class

Table A.3: GroundStation object methods and properties.

Class Function	Description
`is_visible(...)`	Returns `True` if the given *Spacecraft* object is visible (based on the configured visibility condition and Earth's visibility from the *Spacecraft* object's state) at the current simulation time, otherwise returns `False`. *Input*: A *Spacecraft* object.
`set_visibility_mask(...)`	Sets a user-defined visibility mask function for the ground station. Input arguments must be spacecraft azimuth and elevation angles, and return a boolean operator. Useful for more complex terrain around the ground station, where default minimum elevation angle condition does not suffice. *Input*: A callable function that takes azimuth and elevation angles as input arguments and returns a boolean.
`reset_default_visibility_mask()`	Resets the visibility mask to the default visibility mask, which is based on the minimum elevation angle of the spacecraft above the ground station horizon.
`set_minimum_elevation_angle(...)`	Sets the minimum elevation angle of the spacecraft for the defaul ground station visibility mask. The default value is 5 degrees when method is not called. *Input*: Minimum elevation angle in degrees.
`get_object_azimuth_elevation_in_<...>(...)`	Returns the local azimuth and elevation angles of the given *Spacecraft* or *CelestialBody* object from the ground station perspective in radians or degrees, at the current simulation time. Choose between: `rad` or `deg`. *Input*: A *Spacecraft* or *CelestialBody* object. *Returns*: azimuth and elevation angles in radians or degrees.
`get_object_topocentric_<...>(...)`	Returns the topocentric position or direction vector of the given *Spacecraft* or *CelestialBody* object in the ground station's topocentric frame of reference at the current simulation time. Choose between: `position` or `direction`. *Input*: A *Spacecraft* or *CelestialBody* object. *Returns*: position or direction vector in the ground station's topocentric frame.
`get_position(...)`	Returns the position vector of the ground station in the specified reference frame centred at the Earth centre at the current simulation time. *Input*: The reference frame to express the position in. *Returns*: The position vector in the given reference frame.

A.4. Spacecraft Class

Table A.4: Spacecraft object – state methods and properties.

Spacecraft State	
Class Function	**Description**
`update_state()`	Updates the *Spacecraft*'s state and other relevant attributes.
`get_position(. . . )`	Returns the position vector of the *Spacecraft* in the specified reference frame centred at its central body at the current simulation time. *Input*: The reference frame to express the position in. *Returns*: The position vector in the given reference frame.
`get_velocity(. . . )`	Returns the velocity vector of the *Spacecraft* in the specified reference frame centred at its central body at the current simulation time. *Input*: The reference frame to express the velocity in. *Returns*: The velocity vector in the given reference frame.
`get_<. . . >_distance(. . . )`	Returns the distance to the Sun, Earth or given *CelestialBody*, *GroundStation* or *Spacecraft* object in kilometres at the current simulation time. Choose between: `earth`, `sun`, `object`. If `object` chosen, the target object instance must be provided as input argument. *Input*: A target object instance (if `object` chosen) and optional reference frame name. *Returns*: The distance in km.
`get_<. . . >_position(. . . )`	Returns the relative position vector to the Sun, Earth or given *CelestialBody*, *GroundStation* or *Spacecraft* object in the specified reference frame with respect to the spacecraft at the current simulation time. If no reference frame is specified, the default frame of the simulation instance is used. Choose between: `sun`, `earth`, `object`. If `object` chosen, the target object instance must be provided as input argument. *Input*: A target object instance (if `object` chosen) and optional reference frame name. *Returns*: The position vector in the given reference frame.
`get_<. . . >_direction(. . . )`	Returns the relative direction vector to the Sun, Earth or given *CelestialBody*, *GroundStation* or *Spacecraft* object in the specified reference frame with respect to the spacecraft at the current simulation time. If no reference frame is specified, the default frame of the simulation instance is used. Choose between: `nadir`, `sun`, `earth`, `object`. If `object` chosen, the target object instance must be provided as input argument. *Input*: A target object instance (if `object` chosen) and optional reference frame name. *Returns*: The direction vector in the given reference frame.
`is_<. . . >_visible(. . . )`	Returns `True` if the *Spacecraft* can obtain a line of sight to the Sun, Earth or given *CelestialBody*, *GroundStation* or *Spacecraft* object at the current simulation time. Visibility depends on the occultation tolerance set in the simulation instance and, in the case of *GroundStation* objects, their configured visibility mask. Choose between: `sun`, `earth`, `object`. If `object` chosen, the target object instance must be provided as input argument. *Input*: A target object instance (if `object` chosen). *Returns*: boolean.
`get_<. . . >_vector_in_body_fixed_frame()`	Returns vectors expressed in the body-fixed frame of the spacecraft at the current simulation time. Choose between: `sun_direction` – unit vector pointing towards the Sun. `earth_direction` – unit vector pointing towards Earth. `nadir` – unit vector pointing in the nadir direction. `velocity` – unit vector pointing towards the sun *Returns:* The direction unit vector in the *Spacecraft*'s body-fixed frame.

Class Function	Description
`get_object_<. . . >_in_body_fixed_frame(. . . )`	Returns the relative position or direction to a given *Spacecraft*, *GroundStation* or *CelestialBody* object in the body-fixed frame of the spacecraft at the current simulation time. Choose between: `position`, `direction_vector`.
`get_<. . . >_in_body_fixed_frame(. . . )`	Returns additional vectors expressed in the body-fixed frame of the spacecraft at the current simulation time. Choose between: `apoapsis` – apoapsis vector of the spacecraft's orbit. `periapsis` – periapsis vector of the spacecraft's orbit. `eccentricity_vector` – eccentricity vector of the spacecraft's orbit. `limb_ellipse` – parameters of the limb ellipse of the central body as seen from the spacecraft (semi-major axis, semi-minor axis, tilt angle in radians). *Returns:* The requested vector or parameters in the *Spacecraft*'s body-fixed frame.
`get_<. . . >(. . . )`	General getter functions to access specific orbital vectorial elements of the spacecraft's orbit at the current simulation time in the specified reference frame. Choose between: `apoapsis` – apoapsis vector of the spacecraft's orbit. `periapsis` – periapsis vector of the spacecraft's orbit. `eccentricity_vector` – eccentricity vector of the spacecraft's orbit. `orbital_plane_normal_vector` – normal vector of the spacecraft's orbital plane. *Input*: optional reference frame to express the vector in (if none, uses the simualtion's default frame). *Returns*: The requested vector in the given reference frame.
`get_orbit_eclipse_intervals(. . . )`	Returns the start and end times of eclipse intervals (when the spacecraft is in the shadow of its central body, depending on the configured occultation tolerance in the simulation instance). *Input*: number of requested intervals and precision in seconds. *Returns*: List of start and end times of eclipse intervals in ephemeris seconds past J2000.
`altitude`	(property) Returns the altitude of the spacecraft above the surface of its central body at the current simulation time in km.
`solar_flux`	(property) Returns the solar flux at the spacecraft's position in W/m^2 at the current simulation time. Value is based on the configured solar constant in the *Simulation* instance with `set_solar_constant(. . . )` and the spacecraft's distance to the Sun.
`orbital_plane_normal_vector`	(property) Returns the unit vector normal to the orbital plane of the spacecraft at the current simulation time in the default frame of the simulation instance.
`eccentricity`	(property) Returns the eccentricity of the spacecraft's orbit at the current simulation time.
`orbital_period`	(property) Returns the orbital period of the spacecraft in seconds at the current simulation time.
`periapsis`	(property) Returns the periapsis vector of the spacecraft's orbit in the default frame of the simulation instance at the current simulation time.
`periapsis_distance`	(property) Returns the periapsis distance of the spacecraft's orbit in km at the current simulation time.
`apoapsis`	(property) Returns the apoapsis vector of the spacecraft's orbit in the default frame of the simulation instance at the current simulation time.
`apoapsis_distance`	(property) Returns the apoapsis distance of the spacecraft's orbit in km at the current simulation time.
`inclination_in_<. . . >`	(property) Returns the inclination of the spacecraft's orbit in radians or degrees at the current simulation time. Choose between: `rad` or `deg`.

Class Function	Description
raan_in_<. . . >	(property) Returns the right ascension of ascending node (RAAN) of the spacecraft's orbit in radians or degrees at the current simulation time. Choose between: rad or deg.
sc_sun_elongation_angle_in_<. . . >	(property) Returns the elongation angle between the Sun and the spacecraft as seen from the central body in radians or degrees at the current simulation time. Choose between: rad or deg.
argument_of_periapsis_in_<. . . >	(property) Returns the argument of periapsis of the spacecraft's orbit in radians or degrees at the current simulation time. Choose between: rad or deg.
mean_anomaly_in_<. . . >	(property) Returns the mean anomaly of the spacecraft's orbit in radians or degrees at the current simulation time. Choose between: rad or deg.
true_anomaly_in_<. . . >	(property) Returns the true anomaly of the spacecraft's orbit in radians or degrees at the current simulation time. Choose between: rad or deg.
beta_angle_in_<. . . >	(property) Returns the beta angle of the spacecraft's orbit in radians or degrees at the current simulation time. Choose between: rad or deg.
solar_raan_in_<. . . >	(property) Returns the solar right ascension of ascending node (RAAN) of the spacecraft's orbit in radians or degrees at the current simulation time. Choose between: rad or deg.
phase_angle_in_<. . . >	(property) Returns the phase angle between the Sun, spacecraft and central body in radians or degrees at the current simulation time. Choose between: rad or deg.

Table A.5: Spacecraft object – attitude methods and properties.

Spacecraft Attitude	
Class Function	**Description**
set_spacecraft_platform_id(. . .)	Sets the NAIF code of the spacecraft vehicle structure platform to which its instruments' attitude is referenced. By default PAS3 follows SPICE convention by adding three zeroes behind the spacecraft NAIF ID (as its CK body-fixed frame), but other codes are supported by this method. *Input*: NAIF code of the spacecraft platform.
use_spice_attitude(. . .)	If set to True, the *Spacecraft* will use the loaded CK (of FK) file attitude data (if present) to update its attitude at each time change. If set to False, the attitude will not be updated and the user must provide a custom attitude description in the form of a custom model using the consruct_body_rotation_matrix(. . .) method. The default value is True. *Input*: boolean value.
consruct_body_rotation_matrix(. . .)	Constructs and stores the spacecraft's body-fixed rotation matrix in the default frame of the simulation instance at the current simulation time based on given axes. *Input*: Three orthogonal unit vectors defining the body-fixed frame axes in the default simulation frame.
get_body_rotation_matrix(. . .)	Returns the spacecraft's body-fixed rotation matrix expressed in the specified reference frame at the current simulation time. *Input*: optional reference frame to express the rotation matrix in (if none, uses the simualtion's default frame). *Returns*: The rotation matrix in the given reference frame.
from_body_fixed_frame(. . .)	Converts a vector from the spacecraft's body-fixed frame to the specified reference frame at the current simulation time. *Input*: the vector in the body-fixed frame and an optional reference frame (if none, uses the simualtion's default frame). *Returns*: The vector in the given reference frame.

Class Function	Description
`to_body_fixed_frame(. . . )`	Converts a vector from the specified reference frame to the spacecraft's body-fixed frame at the current simulation time. *Input*: the vector in the given reference frame and an optional reference frame (if none, uses the simualtion's default frame). *Returns*: The vector in the body-fixed frame.

Table A.6: Spacecraft object – central body methods and properties.

Spacecraft & its Central Body	
Class Function	**Description**
`set_central_body(. . . )`	Set the central body of the spacecraft to a given *CelestialBody* object. This will be the body used to express the spacecraft's orbit and position with. The *CelestialBody* object must be assigned to the same *Simulation* instance as well. *Input*: A *CelestialBody* object.
`get_groundtrack_vector(. . . )`	Returns the vector pointing from the spacecraft's central body origin to the spacecraft's ground track on its surface in the specified reference frame at the current simulation time. If no reference frame is specified, the default frame of the simulation instance is used. *Input*: optional reference frame to express the vector in (if none, uses the simualtion's default frame). *Returns*: The ground track position vector in the given reference frame.
`get_limb_ellipse(. . . )`	Returns the parameters of the limb ellipse of the central body as seen from the spacecraft in the specified reference frame at the current simulation time. *Input*: optional reference frame to express the limb ellipse in (if none, uses the simualtion's default frame). *Returns*: semi-major axis, semi-minor axis, tilt angle in radians.
`central_body_approximate_limb_half_angle_in_<. . . >`	(property) Returns the angle between the vector pointing towards the body's centre and apparent limb of the body viewed from the spacecraft in radians or degrees. "Approximate" because the limb is assumed to be circular (encircling the real elliptical limb of the body). Choose between: `rad` or `deg`.
`sun_half_angle_in_<. . . >`	Returns the angle between the vector pointing towards the Sun and apparent solar limb as viewed from the spacecraft in radians or degrees at the current simulation time. Choose between: `rad` or `deg`.
`groundtrack_lon_lat`	(property) Returns a tuple of the longitude and latitude of the spacecraft at the current simulation time, in the central body's body-fixed reference frame.
`groundtrack_vector`	(property) Returns the vector pointing from the spacecraft's central body origin in the simulation's default reference frame to the spacecraft's ground track on its surface at the current simulation time.

Table A.7: Spacecraft object – Ground station methods and properties.

Spacecraft & Ground Stations	
Class Function	**Description**
`assign_ground_stations(. . . )`	Assigns one or more *GroundStation* objects to the *Spacecraft* instance. The *GroundStation* objects must be assigned to the same *Simulation* instance as well. *Input*: One or more *GroundStation* objects.
`get_ground_station(. . . )`	Returns assigned *GroundStation* object by its name. *Input*: Name of the ground station. *Returns*: The corresponding *GroundStation* object.

Class Function	Description
`is_in_communication_window()`	Returns `True` if at least one assigned *GroundStation* is visible from the spacecraft's perspective at the current simulation time.
`list_visible_ground_stations()`	Returns a list of *GroundStation* names that are visible from the spacecraft's perspective at the current simulation time.

A.5. Geometry Class

Table A.8: Geometry methods and properties.

Class Function	Description
`sees_central_body()`	Checks if the Geometry intersects the central body (assumed to be a sphere) *Returns*: boolean.
`sees_the_sun()`	Checks if the Geometry intersects the sun sphere *Returns*: boolean.
`sees_celestial_body(...)`	Checks if the Geometry intersects an arbitrary instance of type *CelestialBody* (assumed to be a sphere) *Input*: A *CelestialBody* object. *Returns*: boolean.
`sees_point_object(...)`	Checks if the *Geometry* "sees" an arbitrary instance of type *Spacecraft* or *CelestialBody*, modeled as a point in space. *Input*: A *Spacecraft* or *CelestialBody* object. *Returns*: boolean.
`sees_ray(...)`	Checks if an arbitrary vector (or ray) points inwards of the geometry. *Input*: A direction vector expressed in the spacecraft's body-fixed frame. *Returns*: boolean.
`rotate(...)`	Rotates the geometry around a given axis by a given angle in radians. This method will update the relative orientation of the geometry in its assigned *Spacecraft* body-fixed frame. *Input*: rotation axis (3D unit vector) and rotation angle in radians.
`move_to(...)`	Moves the geometry's position to a given location in the *Spacecraft*'s body-fixed frame. This method will update the relative position of the geometry in its assigned *Spacecraft* body-fixed frame. *Input*: position vector (3D) in the *Spacecraft*'s body-fixed frame.
`set_half_angle(...)`	Sets the geometry's half-angle (in radians) if applicable (e.g., for conical geometries). *Input*: half-angle in radians.
`set_vertices(...)`	Sets the geometry's vertices (boresight for half-space and conical geometries, and corner vertices for polygonal geometries) in the *Spacecraft*'s body-fixed frame. *Input*: list of 3D position vectors in the *Spacecraft*'s body-fixed frame.
`get_boresight()`	Returns the geometry's boresight direction vector in the simulation's default reference frame at the current simulation time. *Returns*: The boresight direction vector.
`get_surface_hit(...)`	Returns the intersection point of the geometry's boresight with its spacecraft's central body surface expressed in the given reference frame at the current simulation time. *Input*: reference frame to express the position in (if none, uses the simualtion's default frame). *Returns*: The position vector of the surface hit point in the given reference frame.

Class Function	Description
`get_surface_hit_lon_lat()`	Returns the longitude and latitude of the intersection point of the geometry's boresight with its spacecraft's central body surface in the central body's body-fixed frame at the current simulation time. *Returns*: longitude (deg) and latitude (deg).
`get_surface_aoi_in_<. . . >()`	Returns the Angle of Incidence between the geometry's boresight and an imaginary unit plane in the origin of the geometry, orthogonal to its boresight, in radians or degrees. Choose between: `rad` or `deg`. *Returns*: The angle of incidence in radians or degrees.
`angle_of_incidence _in_<. . . >(. . . )`	Returns the Angle of Incidence between an incoming ray and an imaginary unit plane in the origin of the geometry, orthogonal to its boresight. Choose between: `rad` or `deg`. *Input*: A direction vector expressed in the spacecraft's body-fixed frame. *Returns*: The angle of incidence in radians or degrees.

B

Toolkit Verification Orbit Configurations

This appendix shows the approach taken to set up the verification scenarios for the PAS3 toolkit in section 4.5. In order to verify the toolkit functionalities, two orbits were defined around Venus and Earth. Section B.1 shows the scripts used to create the orbit SPK files. Section B.2 shows the custom SPICE IK file created to define the circular, rectangular and triangular verification instruments'FOV's from subsection 4.5.4. Section B.3 shows the PAS3 configuration files (similarly to section 5.2) for the tests inside the verification campaign in section 4.5.

B.1. Verification SPK Files

The following code listings show the scripts used to create the verification orbit SPK files for Venus and Earth. In both cases, a circular orbit is created at low altitude. Listing 37 shows the script to create the Venus orbit SPK file, while Listing 38 shows the Earth orbit SPK creation script. PAS3 does not natively support orbit creation,

```python
import numpy as np
import spiceypy as spice

### Load necessary SPICE kernels
spice.furnsh('kernels/lsk/naif0012.tls')        # Leapseconds kernel
spice.furnsh('kernels/spk/de432s.bsp')          # Planetary ephemeris SPK
spice.furnsh('kernels/pck/pck00010.tpc')        # Planetary constants kernel
spice.furnsh('kernels/fk/RSSD0002.tf')          # Frame kernel for VSO frame

### Venus as central body
verisat = -1000                                 # NAIF ID for Verisat
center = 299                                    # NAIF ID for Venus
frame = "J2000"                                 # Reference frame
start_et = 0.0                                  # Start time [seconds past J2000]
end_et = 10 * 24 * 3600                         # end time [10 days past J2000]
segid = "Circular VSO Test Orbit Venus"         # Segment identifier
peri_et = 0.0                                   # Epoch of the periapse [seconds past J2000]
### Parameters
pole = spice.pxform("VSO", frame, start_et) @ np.array([0, 0, 1]) # Trajectory pole vector
sun_position = spice.spkpos("SUN", start_et, frame, "NONE", "VENUS")[0] # Sun position vector
r = 6500                                        # Orbit radius [km]
peri_vector = -np.array(sun_position) / np.linalg.norm(sun_position) * r # Periapsis vector
p = float(np.linalg.norm(peri_vector))          # Semi-latus rectum
ecc = 0.0                                       # Eccentricity
j2flg = 0                                       # J2 processing flag
GM = 0.32486e6                                  # Gravitational parameter [km^3/s^2]
j2 = 0.0                                        # Central body J2
R = spice.bodvrd("VENUS", "RADII", 3)[1][0]     # Equatorial radius of central body [km]
handle = spice.spkopn("verisat_venus_test_orbit.bsp", str(verisat), 0) # Create SPK file
### Populate SPK file
spice.spkw15(handle, verisat, center, frame, start_et, end_et, segid, peri_et, pole,
↪  peri_vector, p, ecc, j2flg, pole, GM, j2, R)
```

Listing 37: Creating Venus verification orbit SPK file.

hence these scripts use SpycePy's spkw15 routine to create a type 15 SPK defining a continuous ephemeris for an undisturbed Keplerian orbit at the J2000 epoch covering 10 days.

```python
import numpy as np
import spiceypy as spice

### Load necessary SPICE kernels
spice.furnsh('kernels/lsk/naif0012.tls')        # Leapseconds kernel
spice.furnsh('kernels/spk/de432s.bsp')          # Planetary ephemeris SPK
spice.furnsh('kernels/pck/pck00010.tpc')        # Planetary constants kernel

### Earth as central body
verisat = -1000                                 # NAIF ID for Verisat
center = 399                                    # NAIF ID for Venus
frame = "J2000"                                 # Reference frame
start_et = 0.0                                  # Start time [seconds past J2000]
end_et = 10 * 24 * 3600                         # end time [10 days past J2000]
segid = "Circular equatorial Test Orbit Earth"  # Segment identifier
peri_et = 0.0                                   # Epoch of the periapse [seconds past J2000]
### Parameters
pole = np.array([0, 0, 1])                      # Trajectory pole vector
sun_position = spice.spkpos("SUN", start_et, frame, "NONE", "VENUS")[0] # Sun position vector
r = 6600                                        # Orbit radius [km]
peri_vector = peri_vector = np.array([r, 0, 0]) # Periapsis vector
peri_vector_j2000 = spice.pxform("IAU_EARTH", "J2000", 0.0) @ peri_vector # Periapsis in
↪ J2000 (ICRF)
p = float(np.linalg.norm(peri_vector))          # Semi-latus rectum
ecc = 0.0                                       # Eccentricity
j2flg = 0                                       # J2 processing flag
GM = 3986004418e6                               # Gravitational parameter [km^3/s^2]
j2 = 0.0                                        # Central body J2
R = spice.bodvrd("EARTH", "RADII", 3)[1][0]     # Equatorial radius of central body [km]
handle = spice.spkopn("verisat_earth_test_orbit.bsp", str(verisat), 0) # Create SPK file
### Populate SPK file
spice.spkw15(handle, verisat, center, frame, start_et, end_et, segid, peri_et, pole,
↪ peri_vector_j2000, p, ecc, j2flg, pole, GM, j2, R) # Populate SPK file
```

Listing 38: Creating Earth verification orbit SPK file.

B.2. Custom SPICE IK File defining Verification Instruments

The following code listing shows the custom SPICE IK text file created to define the circular, rectangular and triangular verification instruments'FOV's from subsection 4.5.4. They are set up in such a way that their FOV's make a 45-degree angle at their apex at the intersection of the XY-plane (see Figure 4.15).

```
\begindata

INS-1000100_NAME                  = 'VERISAT_RECTANGULAR_INSTRUMENT'
INS-1000100_BORESIGHT             = ( 1.0, 0.0, 0.0    )
INS-1000100_FOV_FRAME             = 'VERISAT_INSTRUMENT'
INS-1000100_FOV_SHAPE             = 'RECTANGLE'
INS-1000100_FOV_CLASS_SPEC        = 'ANGLES'
INS-1000100_FOV_REF_VECTOR        = ( 0.0, 1.0, 0.0    )
INS-1000100_FOV_REF_ANGLE         = ( 22.5             )
INS-1000100_FOV_CROSS_ANGLE       = ( 22.5             )
INS-1000100_FOV_ANGLE_UNITS       = 'DEGREES'

INS-1000200_NAME                  = 'VERISAT_CIRCULAR_INSTRUMENT'
INS-1000200_BORESIGHT             = ( 1.0, 0.0, 0.0    )
INS-1000200_FOV_FRAME             = 'VERISAT_INSTRUMENT'
INS-1000200_FOV_SHAPE             = 'CIRCLE'
INS-1000200_FOV_CLASS_SPEC        = 'ANGLES'
INS-1000200_FOV_REF_VECTOR        = ( 0.0, 1.0, 0.0    )
INS-1000200_FOV_REF_ANGLE         = ( 22.5             )
INS-1000200_FOV_ANGLE_UNITS       = 'DEGREES'

INS-1000300_NAME                  = 'VERISAT_TRIANGULAR_INSTRUMENT'
INS-1000300_BORESIGHT             = ( 1.0, 0.0, 0.0    )
INS-1000300_FOV_FRAME             = 'VERISAT_INSTRUMENT'
INS-1000300_FOV_SHAPE             = 'POLYGON'
INS-1000300_FOV_BOUNDARY_CORNERS  = ( 0.92387953, 0,             0.38268343,
                                      0.73365688, 0.60778126,    -0.30389063,
                                      0.73365688, -0.60778126,   -0.30389063 )

NAIF_BODY_NAME += ('VERISAT_RECTANGULAR_INSTRUMENT', 'VERISAT_CIRCULAR_INSTRUMENT',
 ↪  'VERISAT_TRIANGULAR_INSTRUMENT')
NAIF_BODY_CODE += (-1000100, -1000200, -1000300)

```

Listing 39: Verification instruments SPICE IK file for the rectangular, circular and triangular instrument FOV.

B.3. PAS3 Verification Configuration Files

The following code listings show the PAS3 simulation configuration files used for the verification campaign in section 4.5. Listing 40 shows the configuration for the Venus verification orbit, while Listing 41 shows the configuration for the Earth verification orbit. The PAS3 objects created are directly used inside the verification tests scripts by importing them.

```python
import pas3 as p3

sim = p3.Simulation([
    "kernels/fk/vso_frame.tf",                    # VSO frame definition
    'kernels/fk/verisat_coincident_with_vso.tf',  # Verisat frame coincident with VSO
    'kernels/fk/earthfixeditrf93.tf',             # EARTH_FIXED alias FK for ITRF93
    'kernels/fk/estrack_v04.tf',                  # Estrack station ID's and names + frames
    'kernels/ik/verisat_instruments.ti',          # IK for instrument verification
    'kernels/lsk/naif0012.tls',                   # Leapseconds
    'kernels/pck/EARTH_000101_140824_140602.BPC', # Binary PCK for ITRF93 frame
    'kernels/pck/pck00010.tpc',                   # Planetary constants
    'kernels/spk/de432s.bsp',                     # Planetary ephemerides
    'kernels/spk/verisat_venus_test_orbit.bsp',   # VERISAT Verification orbit  around Venus
    'kernels/spk/estrack_v04.bsp',                # Estrack station ephemerides
])

# Bodies and Spacecraft
Venus = p3.CelestialBody(299, sim)
Venus.set_gravitational_parameter(0.32486e6) # km^3/s^2
Verisat = p3.Spacecraft(-1000, Venus, sim)
Verisat.use_spice_attitude(False)

# SPICE Instruments
RectFOV = p3.SpiceInstrument(Verisat, "VERISAT_RECTANGULAR_INSTRUMENT")
CircFOV = p3.SpiceInstrument(Verisat, "VERISAT_CIRCULAR_INSTRUMENT")
TriFOV = p3.SpiceInstrument(Verisat, "VERISAT_TRIANGULAR_INSTRUMENT")

# Custom Instrument counterparts
Custom_RectFOV = p3.CustomInstrument(Verisat, "CUSTOM_VERISAT_RECTANGULAR_INSTRUMENT",
↪  RectFOV.vertices, "RECTANGLE")
Custom_CircFOV = p3.CustomInstrument(Verisat, "CUSTOM_VERISAT_CIRCULAR_INSTRUMENT",
↪  [CircFOV.boresight], "CIRCLE", CircFOV.reference_angle_in_deg)
Custom_TriFOV = p3.CustomInstrument(Verisat, "CUSTOM_VERISAT_TRIANGULAR_INSTRUMENT",
↪  TriFOV.vertices, "POLYGON")

# Custom Geometry counterparts
Geometry_RectFOV = p3.Geometry(Verisat, "GEOMETRY_VERISAT_RECTANGULAR_INSTRUMENT",
↪  RectFOV.vertices, "RECTANGLE")
Geometry_CircFOV = p3.Geometry(Verisat, "GEOMETRY_VERISAT_CIRCULAR_INSTRUMENT",
↪  [CircFOV.boresight], "CIRCLE",CircFOV.reference_angle_in_deg)
Geometry_TriFOV =
↪  p3.Geometry(Verisat,"GEOMETRY_VERISAT_TRIANGULAR_INSTRUMENT",TriFOV.vertices,"POLYGON")

# Geometries
Plane = p3.Geometry(Verisat, "verification_plane", [[1,0,0]], "HALF_SPACE")
Ray = p3.Geometry(Verisat, "verification_ray", [[1,0,0]], "RAY")

# SPICE Ground Stations:
Cebreros = p3.SpiceGroundStation("CEBREROS", sim)
Svalbard = p3.SpiceGroundStation("SVALBARD", sim)

# Custom Ground Stations:
CustomCebreros = p3.CustomGroundStation("CUSTOM_CEBREROS", sim, Cebreros.longitude,
↪  Cebreros.latitude, Cebreros.altitude)
CustomSvalbard = p3.CustomGroundStation("CUSTOM_SVALBARD", sim, Svalbard.longitude,
↪  Svalbard.latitude, Svalbard.altitude)

Verisat.assign_ground_stations(Cebreros, Svalbard, CustomCebreros, CustomSvalbard)

# Set minimum elevation to 0 degrees for all ground stations
for station in sim.groundstations.values():
    station.set_minimum_elevation_angle(0.0) # degrees
```

Listing 40: Venus PAS3 simulation configuration for verification.

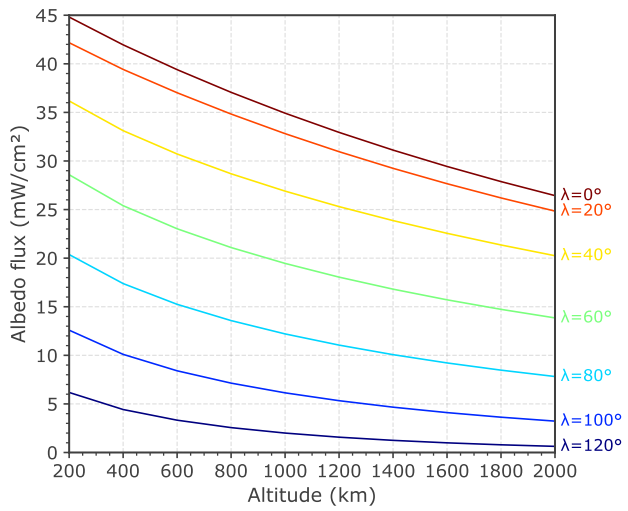
```python
import pas3 as p3
from math import pi, sqrt
import numpy as np

# Configure the simulation for Earth-based VERISAT
Earthsim = p3.Simulation([
    'kernels/fk/verisat.tf',                    # Link Veriisat ID to Name + body fixed frame
    'kernels/fk/earthfixeditrf93.tf',             # EARTH_FIXED alias FK for ITRF93
    'kernels/fk/estrack_v04.tf',                  # Estrack station ID's and names + frames
    'kernels/lsk/naif0012.tls',                   # Leapseconds
    'kernels/pck/pck00010.tpc',                   # Planetary constants
    'kernels/pck/EARTH_000101_140824_140602.BPC', # Binary PCK for ITRF93 frame
    'kernels/spk/de432s.bsp',                     # Planetary ephemerides
    'kernels/spk/verisat_earth_test_orbit.bsp',   # Verification orbit for VERISAT around
    ↪ Earth
    'kernels/spk/estrack_v04.bsp',                # Estrack station ephemerides
])

# Configure the Earth
Earth = p3.CelestialBody(399, Earthsim)
Earth.set_gravitational_parameter(0.3986004418e6) # km^3/s^2

# Configure the verification orbit and spacecraft
EarthVerisat = p3.Spacecraft(-1000, Earth, Earthsim)

# Add a theoretical ground station at the intersection between the equator and prime meridian
EquatorialGroundStation = p3.CustomGroundStation("EQUATORIAL_GS", Earthsim, 0.0, 0.0, 0.0)
EarthVerisat.assign_ground_stations(EquatorialGroundStation)

```

Listing 41: Earth PAS3 simulation configuration for verification.

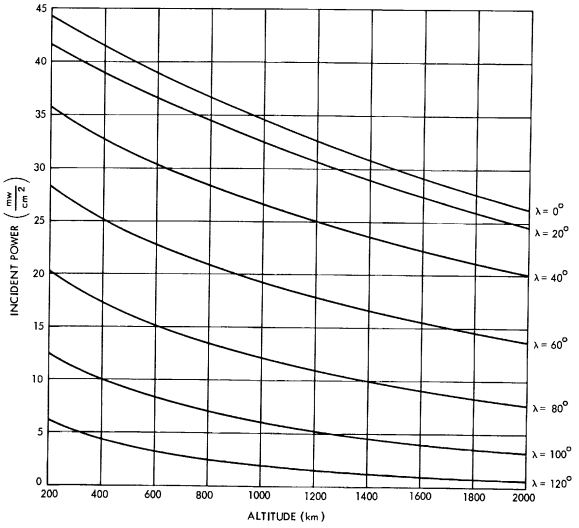
C

Thermal Verification Plots

This appendix presents the verification plots for the albedo flux model developed in subsection 5.4.3. The model is verified against reference plots from NASA [73] for four different solar elongation angles: $\vartheta_s = 0°, 30°, 60°, 90°$. The verification runs were performed as described in subsection 5.4.3, and the resulting incidence flux on a rotating panel is plotted against the altitude and panel tilt w.r.t the nadir vector (λ). The results from the PAS3-based model runs are shown to the left, while the reference plots are shown to the right for comparison.

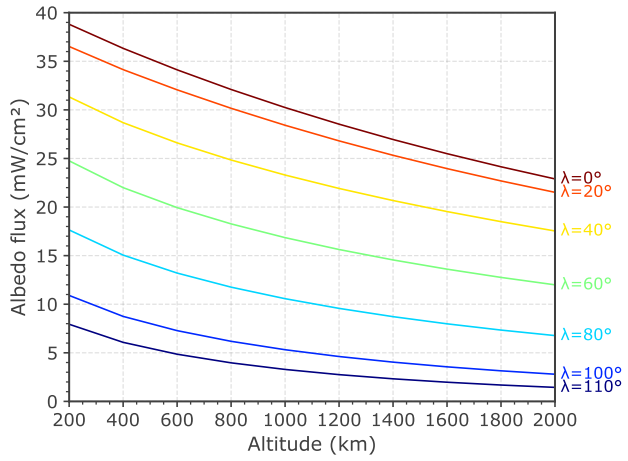


(a) Panel albedo verification run.

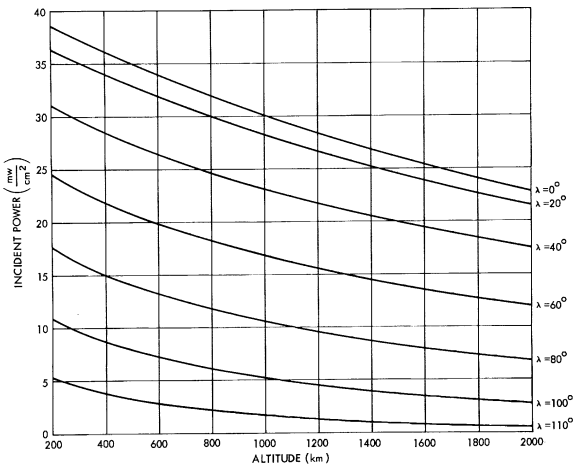


(b) Literature reference [73].

Figure C.1: Verification of the PAS3-based albedo flux model from subsection 5.4.3 against NASA reference plot for solar elongation angle $\vartheta_s = 0$.

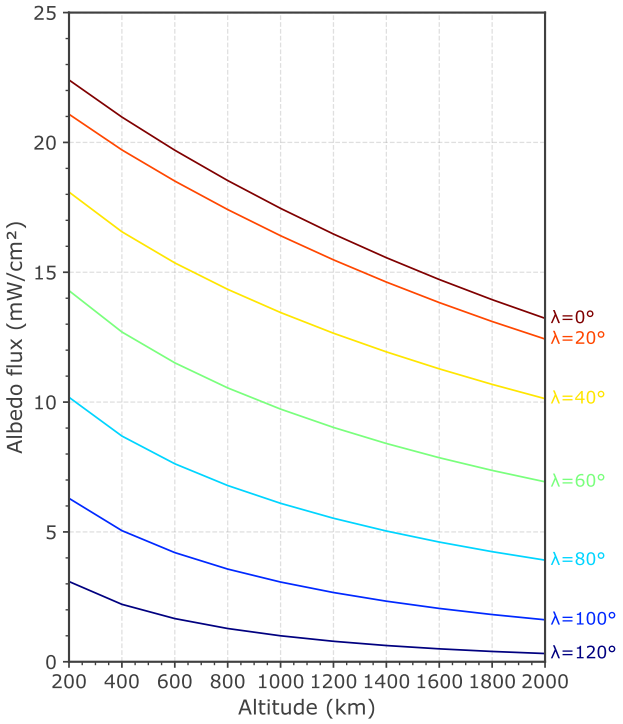


(a) Panel albedo verification run.

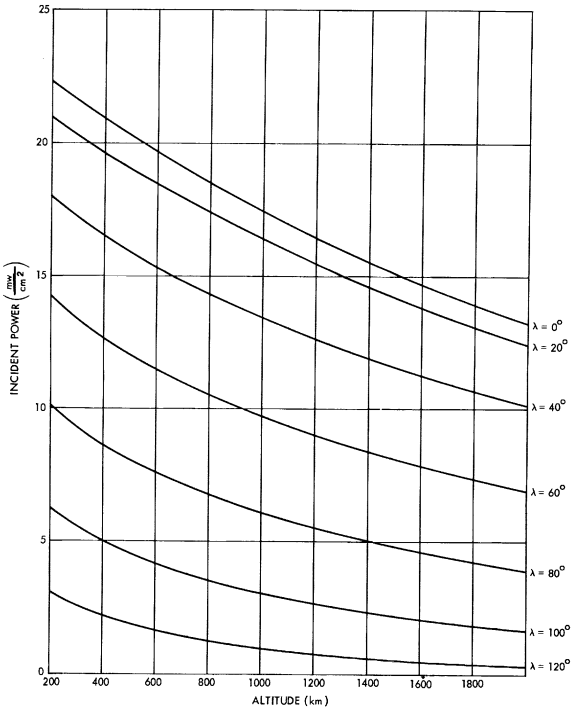


(b) Literature reference [73].

Figure C.2: Verification of the PAS3-based albedo flux model from subsection 5.4.3 against NASA reference plot for solar elongation angle $\vartheta_s = 30$.

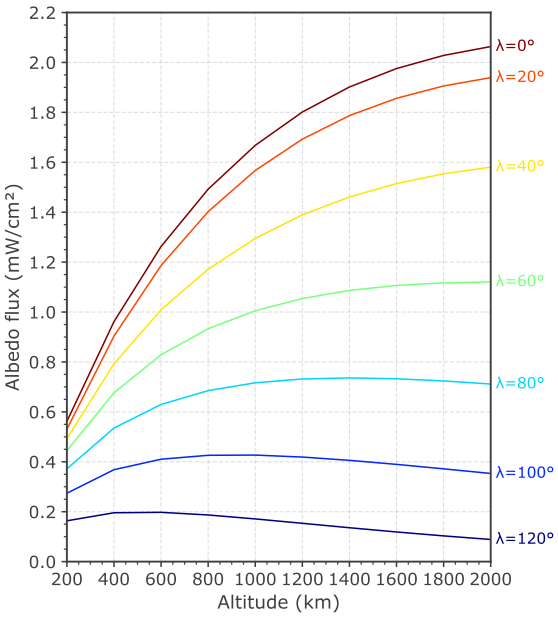


(a) Panel albedo verification run.

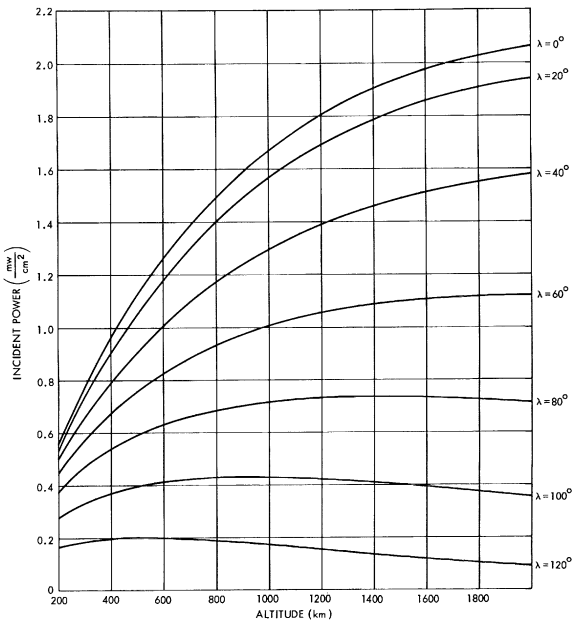


(b) Literature reference [73].

Figure C.3: Verification of the PAS3-based albedo flux model from subsection 5.4.3 against NASA reference plot for solar elongation angle $\vartheta_s = 60$.



(a) Panel albedo verification run.



(b) Literature reference [73].

Figure C.4: Verification of the PAS3-based albedo flux model from subsection 5.4.3 against NASA reference plot for solar elongation angle $\vartheta_s = 90°$.