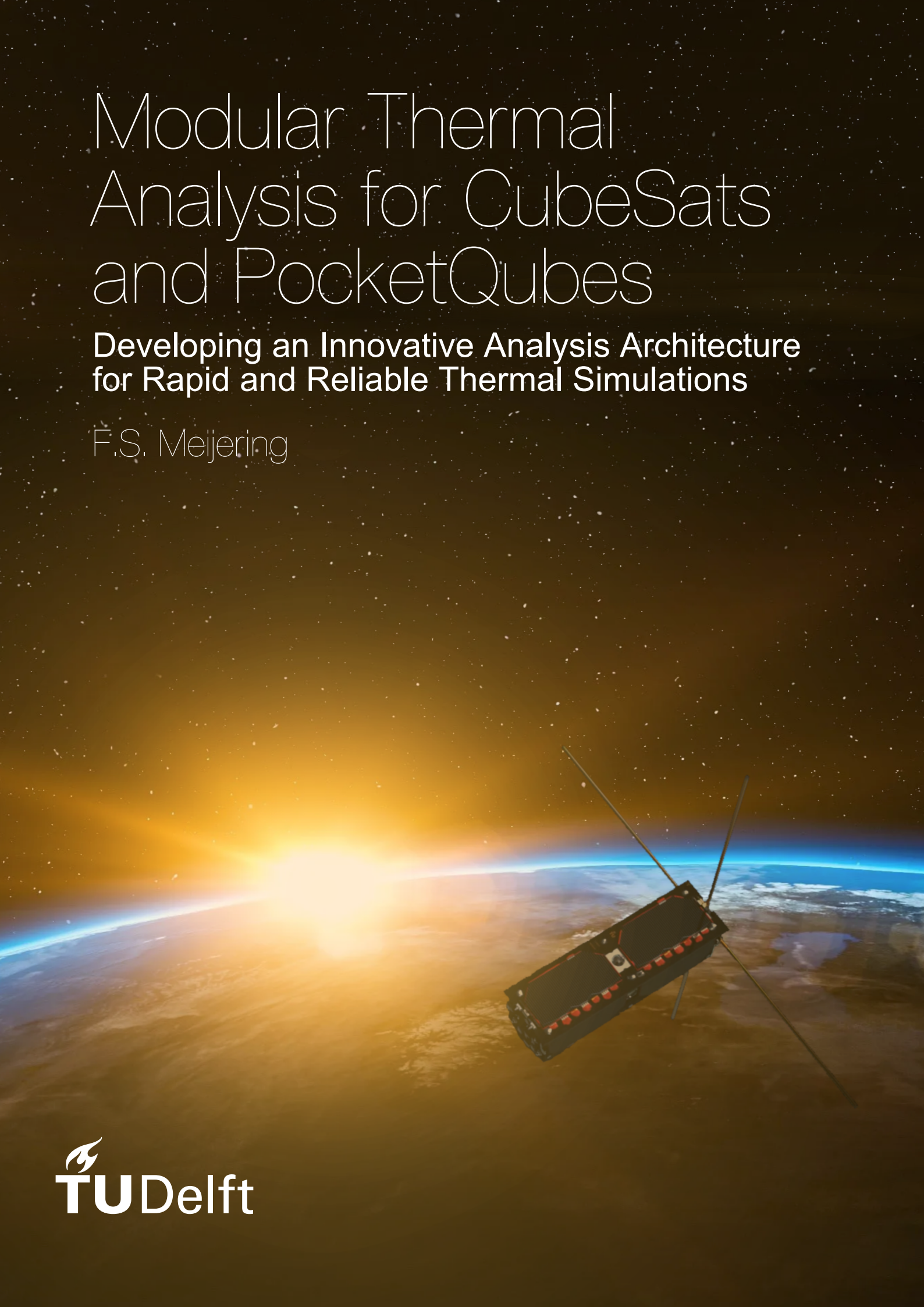
Modular Thermal Analysis for CubeSats and PocketQubes

Developing an Innovative Analysis Architecture for Rapid and Reliable Thermal Simulations

F.S. Meijering



TU Delft

Modular Thermal Analysis for CubeSats and PocketQubes

Developing an Innovative Analysis Architecture for Rapid and Reliable Thermal Simulations

MSc Thesis

by

F.S. Meijering

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on October 14, 2024 at 13:30

Thesis committee:

Chair:	J. Bouwmeester
Supervisor:	M.Ş. Uludağ
External examiner:	B.V.S. Jyoti
Place:	Faculty of Aerospace Engineering, Delft
Project Duration:	February, 2024 - October, 2024
Student number:	5040175

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

Cover Image: adapted from `https://www.youtube.com/watch?v=aboU5R8RycE` and `https://starinastar.com/where-is-the-sun/`

Faculty of Aerospace Engineering · Delft University of Technology

Abstract

The thermal analysis of small satellites (SmallSats) is often given a low priority compared to other sub-systems, due to the short timelines and small budgets associated with SmallSats. However, the satellite's temperature is crucial for preventing mission failures. In this thesis, a new, open-source, modular software for thermal analysis of SmallSats is created with the Python coding language, to simplify and speed up the thermal analysis process. This modularity is achieved by: ensuring that any thermal model can be connected to any other thermal model with a single connect function, allowing a hierarchical model structure that is easily mutable and re-usable; integrating a material database into the software, allowing frequent re-use of common materials, coatings, and contact connections; and including a dedicated sensitivity analysis tool in the software to allow the user to assess the sensitivity of the model's temperatures on various input parameters. To assess the quality of the software's computations, they were verified with the commercial ESATAN-TMS software, and validated with flight data from the FUNcube-1 CubeSat. The validation showed Root-Mean-Square Errors ranging from 1.4°C to 7.1°C for the outside panels of the satellite. With a sensitivity analysis performed on FUNcube-1, it was found that the solar absorptivity of the spacecraft demonstrates the greatest influence on simulation outputs: the largest risk may be its degradation throughout time, due to ultra-violet and atomic oxygen exposure. If the satellite would have had white paint, a year-long degradation could lead to a 17.1°C temperature increase of the maximum achieved temperature in the satellite, depending on the type of paint used. Further development into this software should aim to validate the model even further with different satellites, which will be an important step towards making this tool a standard for SmallSat thermal analysis.

Preface

Just five years after moving to Delft to start my bachelor's in Aerospace Engineering, I am here, writing this master's thesis at the Space Engineering department of TU Delft. This work marks the end of a long journey during which I have learnt more than I could have imagined. Working on one project for numerous months is a totally different game than studying for courses and taking exams, and it has taught me a lot about myself.

Undertaking such an important project would have never been possible just by myself. I have been my own engineer, manager, and adviser, but without the help of many others, I would have not been where I am right now. My thesis supervisor Şevket Uludağ has helped me steer this thesis in the most useful direction, and I highly appreciate his pragmatic and optimistic approach. I also want to thank Stefano Speretta for being my unofficial second supervisor, providing me with satellite flight data, and giving me a lot of insight and feedback as well. It was also very interesting to be part of the bi-weekly Delfi satellite meetings, and actively learn more about small satellite design in general.

I will also thank my study friends with whom I have spent countless hours working together at university, and my gym friends, who helped me forget about the thesis sometimes and work on myself. My friends at home and abroad have always supported me even from hundreds and even thousands of kilometres away. Finally, my family has always supported me through thick and thin. I could not have achieved anything without them, and I would not have been the same person as I am today.

Frank Meijering
Delft, October 2024

Contents

Nomenclature

List of Abbreviations

ABS	Solar Absorptivity
AD&C	Attitude Determination & Control
ALB	Albedo
ALT	Altitude
AO	Atomic Oxygen
AU	Astronomical Unit
BETA	Solar Declination Angle (Beta Angle)
CAP	Heat Capacity
CCHP	Constant Conductance Heat Pipe
CON	Thermal Conductivity
COTS	Commercial-Off-The-Shelf
CSV	Comma-Separated Values
DAY	Day of the Year
DOA	Dead-On-Arrival
DOP853	Explicit Runge-Kutta 8, using 7th order error estimations
DSMC	Direct Simulation Monte Carlo
DT	Time Step
ECSS	European Cooperation for Space Standardization
EMI	Emissivity
EOL	End-Of-Life
EPS	Electrical Power System
FDIR	Fault Detection, Isolation, and Recovery
GMM	Geometrical-Mathematical Model
IDE	Integrated Development Environment
IR	(Earth) Infrared
M&S	Mechanisms & Structures
MBSE	Model-Based Systems Engineering

MCRT	Monte Carlo Ray Tracing
MPPT	Maximum Power Point Tracker
NCG	Non-Condensable Gas
OBC	On-Board Computer
PCM	Phase Change Material
PG	Pyrolytic Graphite
PGS	Pyrolytic Graphite Sheet
POW	Internal Power
PQ	PocketQube
RAAN	Right Ascension of the Ascending Node
RAD	Internal Radiation
RK45	Explicit Runge-Kutta 5, using 4th order error estimations
RMSE	Root-Mean-Square Error
ROT	Rotating
SOL	Solar Heat
SSO	Sun-Synchronous Orbit
TCS	Thermal Control System
TEC	Thermo-Electric Cooler
TMM	Thermal-Mathematical Model
TT&C	Telemetry, Tracking, & Command
UV	Ultra-Violet
VCHP	Variable Conductance Heat Pipe
VEM	Variable Emissivity Material
VLEO	Very Low Earth Orbit

List of Symbols

α	Accommodation coefficient [-]
α	Solar absorptivity [-]
β	Angle between surface normal and line between two surfaces [rad]
ε	Emissivity (in infrared) [-]
ε	Obliquity of the ecliptic [rad]
η	Angle between the zenith vector and $\vec{n}$ [rad]
Γ	Ecliptic true solar longitude [rad]

γ	Angle between orbiting surface normal and the sun vector [rad]
λ	Angle between the nadir vector and $\vec{n}$ [rad]
μ_E	Gravitational parameter of Earth [m^3/s^2]
Ω	Right Ascension of the Ascending Node (RAAN) [rad]
ω	Angular velocity [rad/s]
ϕ	Azimuth angle [rad]
ψ_{ecl}	Eclipse angle [rad]
ρ	Density [kg/m^3]
σ	Stefan-Boltzmann Constant = $5.670374419 \cdot 10^{-8}$ [$W/(m^2K^4)$] [1]
τ	Polar angle [rad]
τ	Thermal time constant [s]
θ	True anomaly [rad]
θ_{ecl}	True anomaly of the onset of the eclipse [rad]
ξ	Solar zenith angle [rad]
A	Surface area [m^2]
a	Albedo coefficient [-]
a	Semi-major axis [m]
a_L	Angle between the nadir vector and the tangent line from the spacecraft to the Earth's surface [rad]
AU	Astronomical Unit = $1.495978707 \cdot 10^{11}$ m [2]
B_{ij}	Gebhart factor from body j to j [-]
C_{cap}	Heat capacity [J/K]
c_{cap}	Specific heat capacity [J/(kg·K)]
$C_{con,contact}$	Conductance from node i to j via a contact interface [W/K]
$C_{con,through}$	Conductance from node i to j through a single material [W/K]
C_{con}	Conductance [W/K]
e	Orbit eccentricity [-]
F_{ij}	View factor from body i to j [-]
h	Heat transfer coefficient [$W/(m^2K)$]
h	Orbital altitude [m]
i	Orbit inclination [rad]
k	Thermal conductivity [W/(m·K)]

L	Distance between nodes [m]
m	Mass [kg]
N	Number of rays [-]
$\vec{n}$	Orbiting surface normal vector [-]
$\dot{q}_{aero}$	Aerothermal heat flux [W/m^2]
$\dot{Q}_{albedo}$	Albedo heat flow [W]
$\dot{Q}_{Earthshine}$	Earth infrared heat flow [W]
$\dot{q}_{ij_{contact}}$	Heat flux from node i to j via a contact interface [W/m^2]
$\dot{q}_{ij_{through}}$	Heat flux from node i to j through a single material [W/m^2]
$\dot{Q}_{ij}$	Heat flow from body i to j [W]
$\dot{Q}_{internal}$	Spacecraft internal heat flow [W]
$\dot{Q}_{out}$	Radiative heat flow emitted by spacecraft [W]
$\dot{Q}_{solar}$	Solar heat flow [W]
r	Distance between a central and orbiting body [m]
R_E	Earth radius [m]
R_S	Sun radius [m]
R_{ij}	Radiative exchange factor from body i to j [-]
S	Distance between two surfaces [m]
$\vec{s}$	Sun-Earth vector [-]
S_0	Solar constant [W/m^2]
T	Orbital period [s]
T	Temperature [K]
t	Time [s]
T_{ecl}	Time in eclipse [s]
V	Velocity [m/s]
x	Zenith direction from the spacecraft's point of view [-]
y	Right-hand side compared to velocity vector of the spacecraft [-]
z	Velocity vector of the spacecraft [-]

List of Figures

List of Tables

1 Introduction

The thermal subsystem of small satellites (SmallSats[1]) is often given low priority by engineers, even though it is critical for the functioning of the satellite [3, 4]. The goal of the thermal subsystem is to keep the satellite's temperature within predefined bounds, thereby ensuring optimal performance of all electronic components. Since the main goal of SmallSats often is to perform Earth-observation measurements, the thermal design in itself is not a mission objective, but serves the rest of the satellite. As a result, the thermal design chosen by engineers often lacks sophistication, which shows in the fact that at least 5% of all SmallSat failures is due to a flawed thermal design [5]. This is touched upon in more detail in the literature review in Section 2.2.

As the thermal subsystem is highly dependent on the design of the rest of the spacecraft, the thermal design process is case-specific and time-consuming. Furthermore, thermal analysis tools such as ESATAN-TMS require specialised engineers, who are not always available in SmallSat design teams. Attempts have been made at developing modular thermal design tools for SmallSats [4, 6, 7]. Such modular tools employ the similarities between all SmallSats to simplify the thermal design process, but they compromise on quality. These compromises include: neglecting radiative heat transfer [7]; assuming optical properties instead of determining them experimentally [6]; and lacking system-level verification [3].

This thesis project aims to bridge the gap between the complexity of thermal design and the simplicity that is desired for SmallSats. An open-source software is developed, with an orbital module based on the work by C. Ruiz [7], and a newly created thermal (nodal) module. Modularity is the key to this software: the code is object-oriented, and allows to copy-paste different models and connect them to each other as desired. Furthermore, a significant amount of effort is dedicated to improving the user-friendliness of this software; part of that includes extensive documentation.

1.1. Research Formulation

The literature review will show that there are still numerous fields of improvement regarding thermal analysis specialised for SmallSats. The main challenge is that existing tools are complicated to learn and use, which can act as a barrier when choosing the thermal analysis method. Furthermore, SmallSats are different compared to traditional large satellites, due to their different requirements, tendency towards modularity, and the difference in resources of the designers. The thermal analysis procedure could be greatly simplified if it is specialised for SmallSats specifically, thereby discarding its use case for large satellites. The research question is therefore:

[1] SmallSats are all satellites with a mass below 180 kg (NASA: `https://www.nasa.gov/what-are-smallsats-and-cubesats/`). In this thesis, the term SmallSats is used because CubeSats and PocketQubes are differently sized satellites, and they may not fall under the same satellite category (nanosatellites (1-10 kg) or picosatellites (0.01-1 kg)). Since the terms CubeSats, PocketQubes, nanosatellites, and picosatellites all refer to slightly different satellite types, the term SmallSats is preferred for this thesis.

Research Question

How can thermal analysis tools be optimised for SmallSats, while maximising user-friendliness and maintaining the reliability of results?

This is a high-level question, and can be divided into a number of more detailed questions, which are stated below:

1. Since modularity is such an important aspect of SmallSats, it should be investigated how it could be included in analysis tools, and what the advantages and disadvantages are, compared to traditional analysis methods. This aspect is captured in Research Sub-Question 1.
2. It was also seen in the literature review that experimental and flight data are often missing in low-budget SmallSat missions. This makes it difficult to predict of which order of magnitude the modelling errors might be. This issue also has a relationship with modularity, as standardised/simplified modules or values might have been verified and validated for one case, but not for the other. What would be the overall impact of all these factors, when comparing an example spacecraft thermal model with real flight data?
3. The last topic to discuss is the sensitivity of thermal analysis to assumptions. For example, internal radiation in the satellite could be neglected, but such an assumption must be supported properly. Additionally, if radiation can indeed not be neglected, would Monte Carlo ray tracing be necessary, or are simplified view factor methods sufficient? Such questions are to be answered within this research sub-question.

Research Sub-Question 1

What is the impact of implementing modularity in a SmallSat thermal modelling software on the user experience and reliability of results?

Research Sub-Question 2

What level of accuracy can be achieved when validating a simplified, modular SmallSat thermal model with flight data?

Research Sub-Question 3

Which elements within a SmallSat thermal model demonstrate the greatest influence on simulation outputs?

1.2. Research Methodology Overview

In this section, a brief overview of the research methodology is given; a more extensive documentation of the methodology is provided in Chapter 3.

A significant part of this thesis is spent on developing all the computations for the thermal analysis, and developing the software's architecture around it. However, besides developing the software, its performance will also be evaluated using flight data from a 1U CubeSat: FUNcube-1. With this data, a thorough sensitivity analysis is also performed. All of this is further explained in the chronological list below:

1. The first month consists of reviewing existing literature; the focus was set on SmallSat failures, a high-level overview of spacecraft thermal control, and a more in-depth review of types of thermal analysis computational schemes. This literature review was used as the guideline for defining the research questions and providing the direction of this thesis, as well as allowing the formulation of requirements.
2. Since C. Ruiz [7] has developed a MATLAB tool for the thermal analysis of SmallSats, her tool is thoroughly investigated and checked for errors or room for improvement.

3. The methods by C. Ruiz are an inspiration for many computations within the new Python software, but some computations were improved, and especially, the software's architecture is vastly different. The new Python tool is focused on modularity, and hence takes an object-oriented approach. Developing ideas on which types of objects would be useful, how a user should be able to construct a model, and other questions are discussed with professors who have experience the field.
4. A "draft" version of the software is developed, and has the aim of performing all thermal computations correctly. The object-oriented approach is integrated, since that forms the backbone of the tool, but extra features related to the user experience are not yet included. The computations are verified by comparing the results with literature and the commercial ESATAN-TMS software. The FUNcube-1 spacecraft is modelled, and the results are compared with flight data.
5. Extra features are implemented, such as a material database and a sensitivity analysis tool. Constructing the material database consists of a small literature study that acquires physical properties of materials that are commonly used in SmallSats. Developing the sensitivity analysis tool is done while keeping user-friendliness in mind: not only does the sensitivity analysis on FUNcube-1 provide estimations on FUNcube-1-specific sensitivities, but the sensitivity analysis tool should also be easily accessible by future users. A user should be able to make their own thermal model, import it into the sensitivity analysis tool, and immediately generate results.
6. Supplementary tasks are performed such as creating an extensive user manual (Appendix C), uploading the software on GitLab and GitHub [8, 9], and improving the user-experience aspect of the software. The combination of these tasks make it significantly easier for new users to get started, or for more interested users to deeply understand the software. The latter is important in case other people aim to improve this software (recommendations are present in Chapter 7).

1.3. Document Overview

The structure of this thesis report is as follows. Chapter 2 is a literature review concerning small satellites, their failure mechanisms, how they relate to thermal analysis, and some general theory on thermal analysis. Next, Chapter 3 shows the methodology behind the software, including the overall software structure, as well as detailed descriptions of the computations that the software performs. Chapter 4 presents the verification and validation performed on this software. The validation is done with the FUNcube-1 satellite. Furthermore, a sensitivity analysis is presented in Chapter 5. Not only is this a general sensitivity analysis of an example satellite, it also is part of the software, where the user can apply the sensitivity function on their own thermal model. A conclusion is given in Chapter 6 and recommendations on future work in Chapter 7. For future users of the software, a detailed user's manual is provided in Appendix C, and the source code is available on GitLab and GitHub [8, 9].

2

Literature Review

This literature review investigates the current status of thermal analysis for small satellites (SmallSats[1]). This helps in understanding the big picture, and identifying knowledge gaps in the field. An extension of this review is provided in Appendix A, showing common thermal control technologies used in SmallSats. This information is not directly applied in this thesis, but it serves as a general understanding of spacecraft control, which is a must when working with thermal analysis software.

2.1. The Popularity Rise of SmallSats

There has been a rapid rise in of SmallSats since the early 2010s, which can be seen in Figure 2.1. Traditional spacecraft development has always been based on highly complex vehicles, optimised for performance and reliability [3]. Most spacecraft were application-specific, designed for a single use case, and therefore there is a limited amount of design that can be transferred directly to new missions. All these large missions are costly, and require much expertise and development time to be successful.

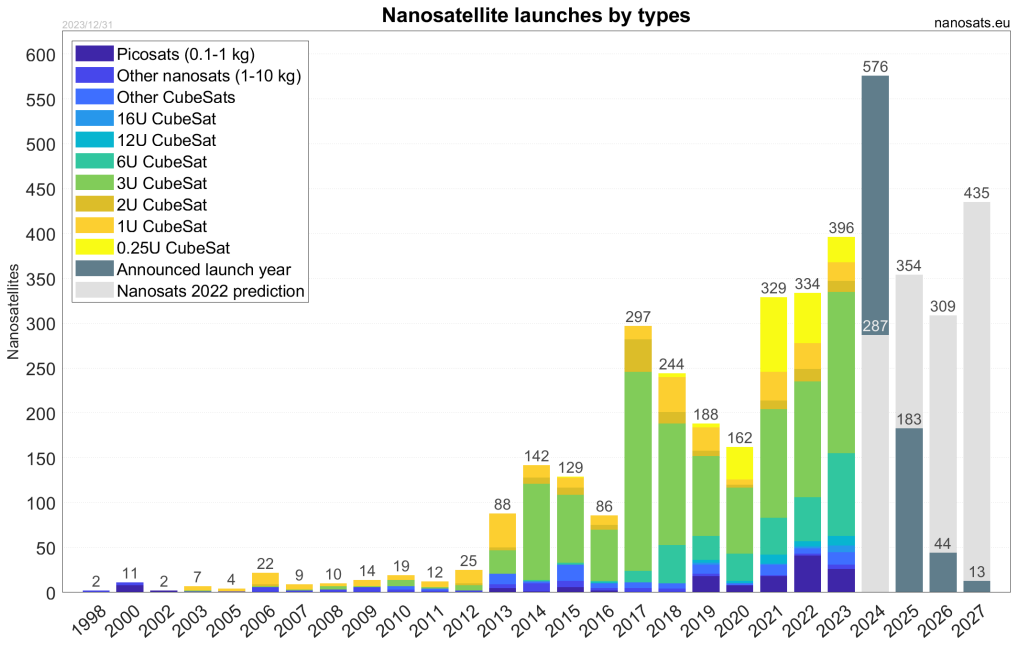


Figure 2.1: Increase in popularity of small satellites, grouped by type [10].

[1]SmallSats are all satellites with a mass below 180 kg (NASA: `https://www.nasa.gov/what-are-smallsats-and-cubesats/`). In this thesis, the term SmallSats is used because CubeSats and PocketQubes are differently sized satellites, and they may not fall under the same satellite category (nanosatellites (1-10 kg) or picosatellites (0.01-1 kg)). Since the terms CubeSats, PocketQubes, nanosatellites, and picosatellites all refer to slightly different satellite types, the term SmallSats is preferred for this thesis.

Recently, smaller companies and institutes such as universities have shown interest in launching their own satellites, but the traditional approach was simply not feasible, due to the associated high costs and development times [11]. SmallSat developments pushed towards these requirements, by employing a modular design philosophy, and using Commercial-Off-The-Shelf (COTS) components [11]. SmallSats are also much smaller than conventional spacecraft, generally standardised in size by means of CubeSat units (1 U = 10x10x10 cm) or PocketQube units [11]. This small form factor, combined with the modular design, results in a reduction of mass, cost, and development time of spacecraft; overall, the consequence of mission failure is significantly reduced compared to traditional spacecraft. This is of great benefit for technology demonstrations, educational purposes, and constellations/swarming.

2.2. SmallSat Failures

As anticipated, the simplicity of SmallSats comes at a cost: low reliability. Although the consequences of a mission failure are much lower than for traditional missions, the likelihood is much higher. In fact, it is estimated that university-built CubeSats have a likelihood near 50% of failing within the first six months after launch [3]; note that industry-built CubeSats are more reliable [12].

2.2.1. CubeSat Mission Fulfilment Status

Figure 2.2 from [13] shows the mission status for CubeSats launched between 2000 and 2015, after their End-of-Life, where the subcategories are:

- Prelaunch: the mission has been aborted before launch;
- Launch fail: the mission has launched, but not successfully inserted into orbit;
- Dead-On-Arrival (DOA): the spacecraft has undergone successful orbit insertion, but no communication link can be established between the spacecraft and ground station;
- Early Loss: the spacecraft can be communicated with, but did not perform the commissioned tasks;
- Partial Mission: the spacecraft has been able to perform a number of operations;
- Full Mission: the minimum number of requirements has been met to deem the mission a success;
- Unknown: the status is unknown to the developers, or the developers did not announce it publicly.

[14] show a similar type of mission status figure, see Figure 2.3, but up until 2018. Furthermore, the categories are slightly different. In the legend, for example, *launched* means that the spacecraft has successfully been launched, but no further information has been received afterwards. Similarly, *deployed* means that deployment was confirmed, but that no telemetry data has been available yet for the phases afterwards.

Figure 2.2 only looks at missions which have come to an end, while Figure 2.3 also includes all CubeSats still in operation (”Primary Operations”). Nevertheless, a similar trend in both figures is that near 40-50% of the missions are essentially failed, of which the majority of failures occur early in the mission.

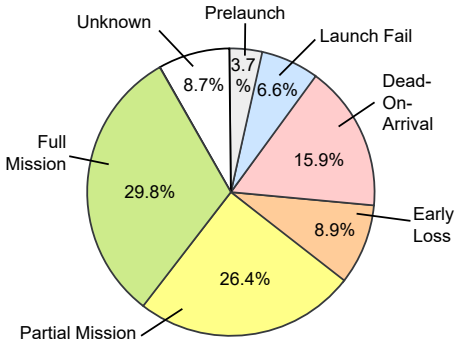


Figure 2.2: Distribution of CubeSat mission fulfilment throughout 2000-2015 (adapted from [13]).

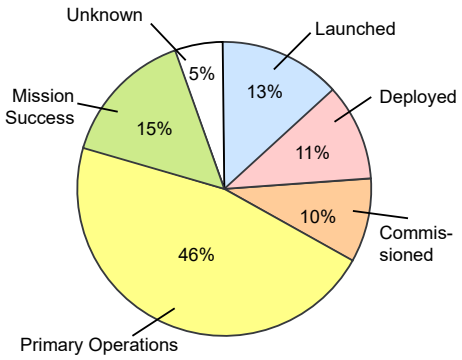


Figure 2.3: Distribution of all CubeSat mission statuses as of 31 May 2018 (adapted from [14]).

A more general trend on SmallSat failures is shown by [12] and [14]. [12] compare missions from 1990-2009 with 2010-2019; Table 2.1 clearly shows an improvement in mission success rates. [14] agree with this statement, see Figure 2.4. However, it is also clear that, throughout time, it will take longer to keep improving these success rates (the curve flattens).

Table 2.1: Failures of ”small” satellites between 40-500 kg, where *censored* indicates that the spacecraft monitoring was stopped or that the spacecraft was turned off before it failed [12].

Launch year	Launched	Failed	Censored
1990-2009	441	228	213
2010-2019	425	56	369

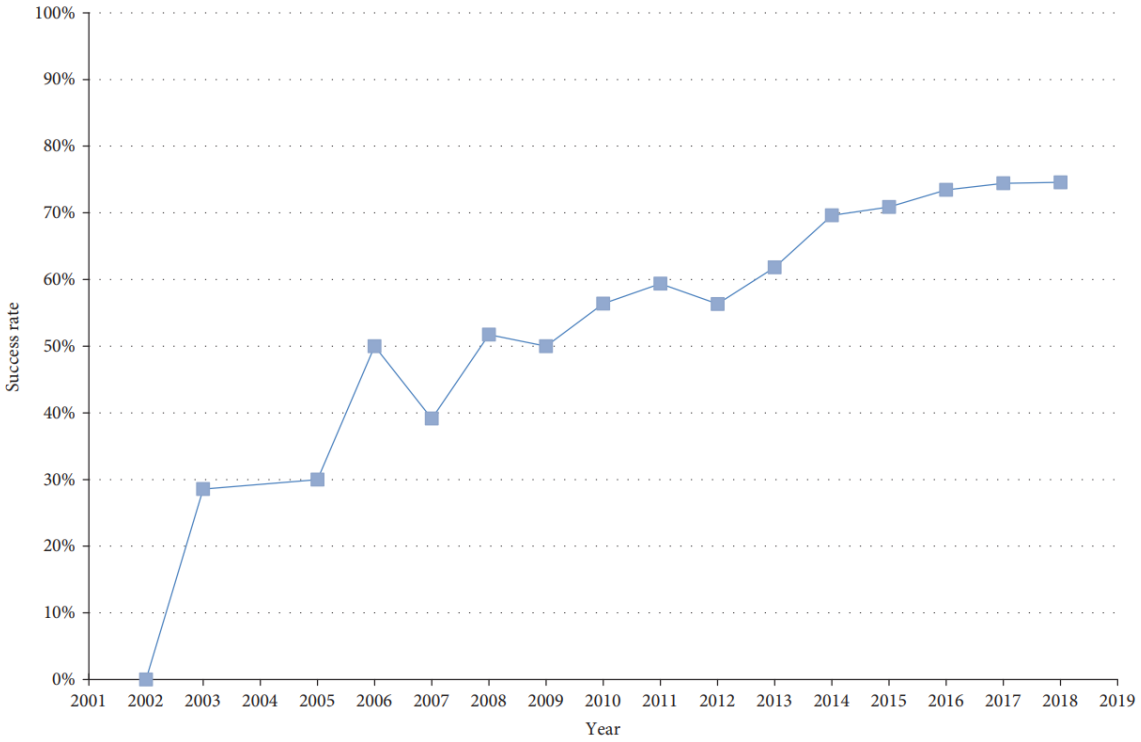


Figure 2.4: Increasing CubeSat success rate as a function of launch year [14].

2.2.2. Causes of Failure

Several authors [3], [12], and [15] dove deeper into the technical causes of mission failures has been done, although the exact cause of failure is not always known. Some important conclusions are:

- According to [3] and [15], the smaller the satellite, the more quickly after launch the reliability drops; for example, Figure 2.5 shows that picosatellites (such as PocketQubes) are almost half as reliable as nanosatellites (such as CubeSats), for a given time after launch. However, if the mission objectives of picosatellites are based on shorter lifetimes, they can still be met.

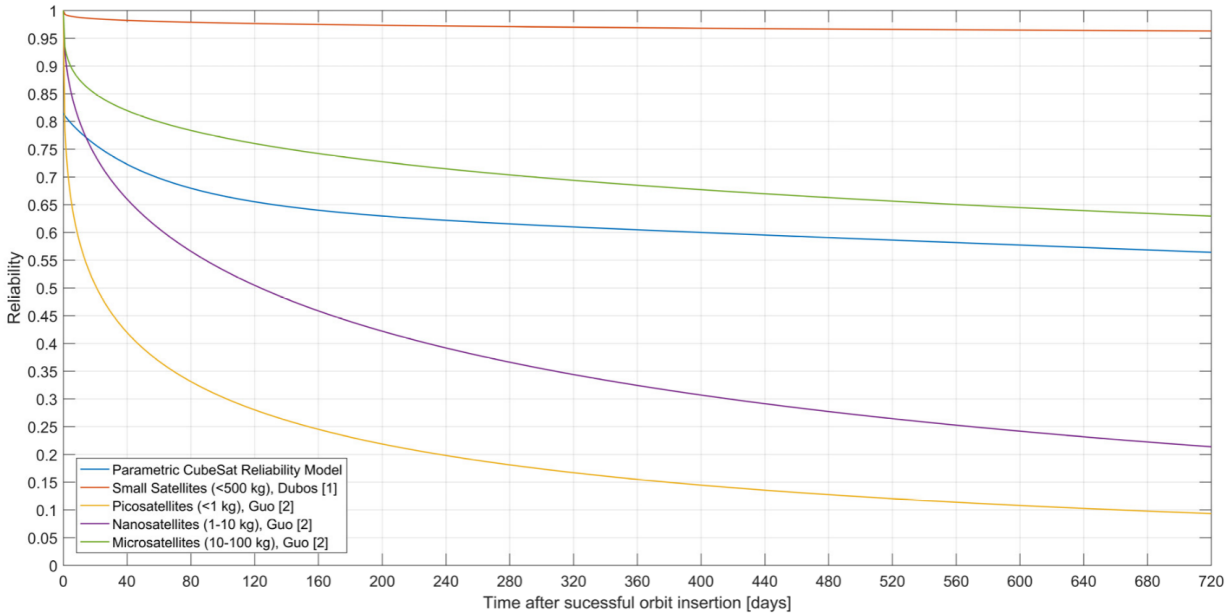


Figure 2.5: Reliability of different SmallSat form factors [3].

- There are often not enough system-level functional tests to verify proper functioning of the integrated system [3]. This is related to the high costs associated with testing campaigns, and such budgets are often not available. On one hand, it can be worth it to invest in developing novel testing/qualifying strategies [4]. On the other hand, the focus could be shifted to improving the reliability of analysis software, e.g. by more extensive validation with flight data of either a predecessor spacecraft, or similar other spacecraft.
- The distributions of the reliability of several subsystems are shown in Figure 2.6. Overall, the most failures occur in the Electrical Power System (EPS), On-Board Computer (OBC), and communications [3, 15]. It should be noted that the majority (40-80%, depending on the lifetime) of the failures has an unknown cause [3]. In all cases, the reliability drops the most severely at the beginning of the mission, which is known as the 'infant mortality' [15]. The telecommand (TT&C), mechanical (M&S), and thermal (TCS) subsystems relatively show the fastest drop, corresponding to the most severe infant mortality [15]. In other words, if these subsystems survive the initial phase after launch, they are unlikely to cause more issues.

  This is understandable for the mechanical and thermal subsystem: firstly, if the mechanisms survive the launch and any possible deployments are successful, the mechanical system remains rather passive during the rest of the flight; secondly, most SmallSat thermal systems are passive [16], so they are likely to experience cyclic behaviour once the initial steady-state has been reached after deployment from the launcher. The latter was seen with Delfi-PQ, where the actual temperature was 15°C lower than expected, after which the temperature remained cyclic [17]. This resulted indirectly in a performance reduction of other subsystems, as the performance of electronics heavily depends on temperature [5].

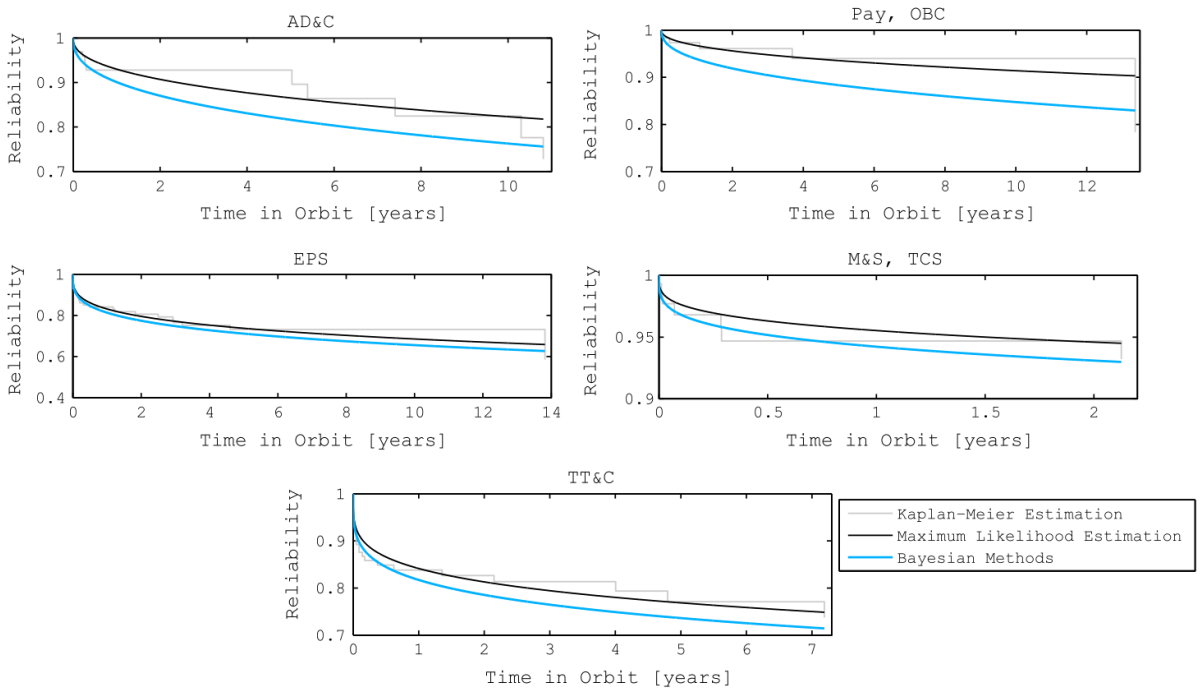


Figure 2.6: Reliability of different subsystems on CubeSats over time. Notice that the x-axes of the subfigures are not identical [15]. AD&C is Attitude Determination & Control; Pay is payload; OBC is On-Board Computer; EPS is Electrical Power System; M&S is Mechanisms & Structures; TCS is Thermal Control System; and TT&C is Telemetry, Tracking, & Command.

Limited information on the specific failures due to a dysfunctional thermal subsystem has been found, besides the information shown before. A reason for this could be that the thermal subsystem often has an indirect impact on the spacecraft's performance: e.g., if the temperature is too high, the batteries might fail, so the failure is attributed to the electrical power system [18]. The study by [18] indeed categorises failures in this manner, by including the thermal subsystem indirectly in all other subsystems. This makes it difficult to know to what extent the thermal subsystem really is the cause of failure.

A final comment can be made on the failed battery example. [19] presents a database of failed missions between 2001-2016. Newer, similar information on such a database could not be found. Regardless, there are three instances where failed thermal management was directly related to issues with the battery. They are mentioned below. This suggests that the batteries could be a sensitive element on the spacecraft when it comes to temperature discrepancies.

- QuakeSat, 3U CubeSat, Stanford University: *"both batteries were lost, allowing the mission to continue on solar power only. Loss of batteries thought due to high battery temperatures (120 degrees Fahrenheit) which may have caused the electrolyte to bake out since the batteries were not sealed beyond the normal factory packaging."* [19, p.20].
- SpriteSat (Rising) Tohoku University of Sendai, Japan: *"The battery charging system allowed the temperature of the battery to reach critical levels."* [19, p.24].
- TET-1 (DLR): *"The temperature within the satellite was higher than predicted, causing the battery voltage to be slightly exceeded. Re-orienting the attitude of the satellite remedied this problem."* [19, p.31].

2.2.3. Delfi-PQ Example

The Delfi-PQ (PocketQube) is a picosatellite launched in January 2022 by Delft University of Technology (TU Delft) [10]. After approximately two years of operation, the spacecraft de-orbited [20]. An image of Delfi-PQ in the lab is shown in Figure 2.7.

The thermal behaviour of the spacecraft was more than 15°C below the expected values [7, 17, 21], and this was immediately visible after launch [21]. This thermal anomaly is indeed a case of infant mortality, since the issue manifested itself immediately after launch. This indicates a severe shortcoming in the thermal balance analysis which was done pre-flight, and it is possible that these shortcomings are due to a lack of validation/experimental data. Although TU Delft has launched other satellites before this, Delfi-PQ was the first PocketQube format sent by the university. A lack of familiarity with the PocketQube format and hence a lack of experimental data could be the reason for the unexpected behaviour in orbit.

The acquired thermal data is discontinuous, due to unforeseen circumstances, and looks like Figure 2.8. The difference between eclipse and in sunlight is clearly seen, but since the data has long gaps in between the acquired points, this data by itself would be insufficient to correlate the analytical model to, and hence, [17] proposed machine learning methods to fill in the gaps in this data.

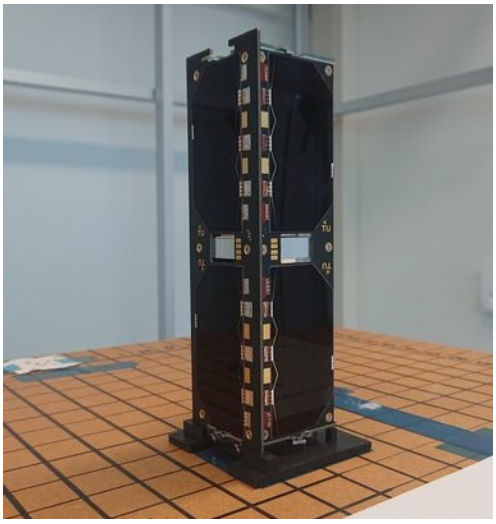


Figure 2.7: Delfi-PQ PocketQube by TU Delft [10].

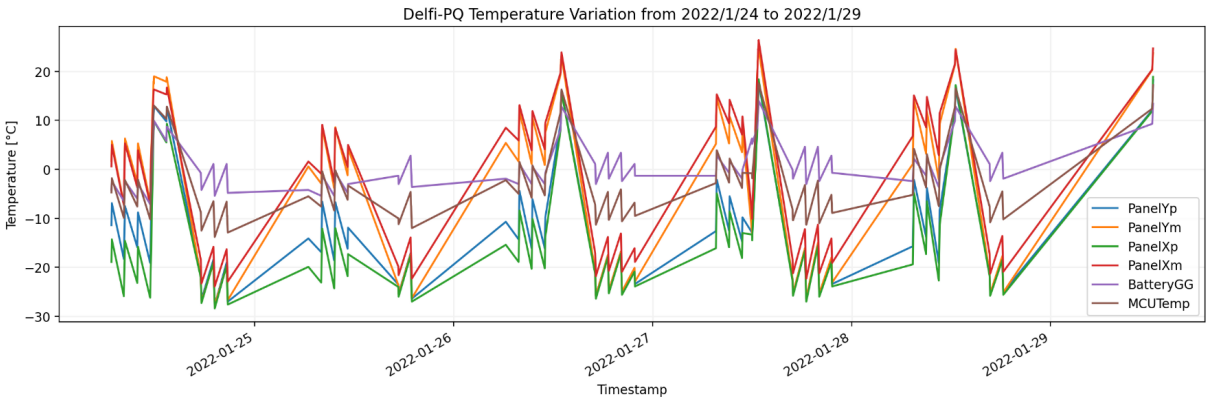


Figure 2.8: Delfi-PQ flight data of the spacecraft temperatures over the course of 5 days ([17] adapted from [21]).

2.2.4. Fault Detection, Isolation, and Recovery (FDIR)

Failures can also happen when there is an incorrect reading of sensory data, which is called a fault [22, 23]. Sensory data with respect to the thermal control entails temperatures and power flows; if any of those telemetry data are wrong, it becomes extremely difficult to use the flight data for monitoring and validation purposes.

FDIR is used on spacecraft since there is no opportunity for maintenance or manual check-ups [22]. To solve this issue, an automated system with detectors and monitors is used to perform regular check-ups on the system health. Properties of these detectors and monitors are [23]:

- Detectors: evaluate the value of a specific measurement on-board the spacecraft as either a True or False; for example, a detector could be a thermistor on a critical spacecraft component such as the battery, where the boolean value is determined by a pre-defined temperature range. When the temperature is within the range, the detector returns True, and otherwise, it reports False. Data filters, such as a low-pass filter [23], can be applied to avoid unstable responses.
- Monitors: identify patterns in the detector outputs. For example, in Figure 2.9, monitor A trips if detector 1 and either 2 or 3 return True. The activity of monitors can be defined by ground control, in the states depicted in Figure 2.10: in the bypass state, the monitor will never trip; in the check state, the monitor is active and will trip if a pattern, as mentioned before, is identified; lastly, the command state is entered after a trip, with the aim of mitigation/recovery.

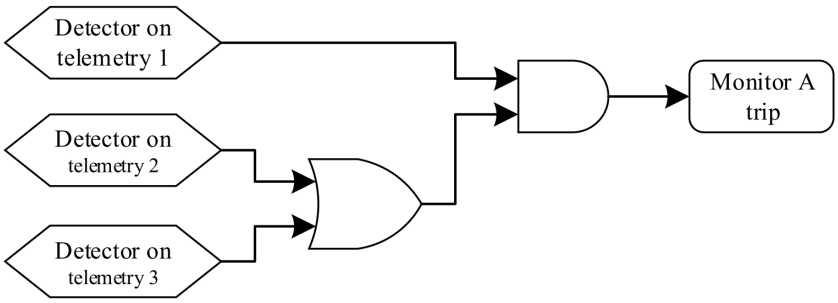


Figure 2.9: Example of the relationship between FDIR detectors and monitors [23].

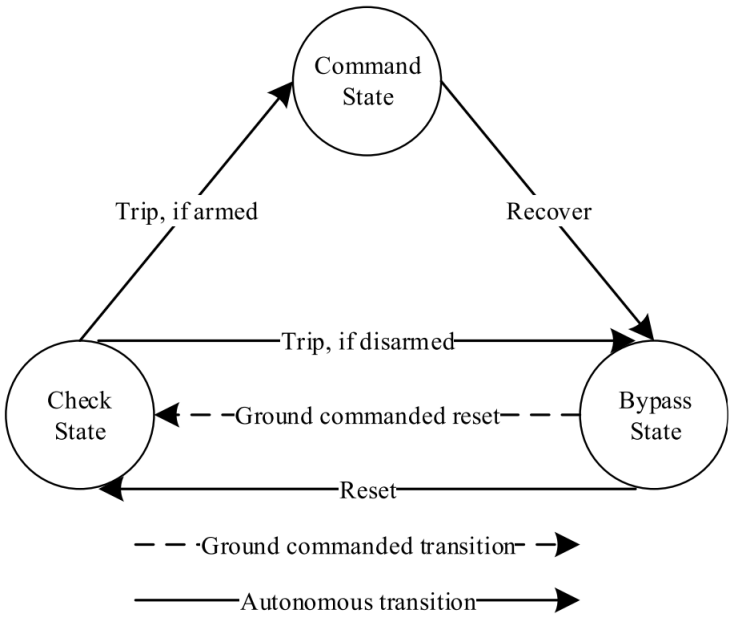


Figure 2.10: Possible states of FDIR monitors [23].

The connections between detectors and monitors can be visualised in a configuration matrix, such as the one in Figure 2.11. The detectors are given an ID and a pre-defined range (which could be temperature, for example). The filter coefficient is related to a low-pass filter, if this is decided to be applied. The coloured columns at the top are the monitors, which are given a checking period and a persistence. The persistence means that the same fault should be detected multiple times before the monitor trips. Finally, ”O1” and ”A1” in the matrix indicate OR and AND relationships between the multiple detectors, similarly to what was shown in Figure 2.9. For example, monitor Unit A Health trips if either Unit A Temp 1 OR Unit A Temp 2 is False (out of the defined bounds), AND Unit A Msg Index is True (constant, where the change is 0).

	ID	Upper Bound	Lower Bound	Filer Coefficient (k)	Detector Period (s)	Unit A Health	Unit B Health
Period (s)						30	60
Persistence						3	5
Logic						A	O
Detector							
Unit A Temp 1	87	5	35	0	2	F:O1	
Unit A Temp 2	88	10	40	0	2	F:O1	
Unit A Msg Index	89	0	0	-1	4	T	
Unit B On/Off	90	On	On	0	2		T:A1
Unit B Error	91	OK	OK	0	2		F:A1
Unit B Temp	92	10	40	0	2		F

Figure 2.11: FDIR configuration matrix with the detectors on the left rows and the monitors on the top columns [23].

2.3. Spacecraft Thermal Control

As seen in the Delfi-PQ example, the thermal subsystem is a crucial part of designing a functional spacecraft. The harsh space conditions result in large temperature gradients if the thermal subsystem is not designed properly: the lack of an atmosphere results in strong solar irradiation, heating up the spacecraft; the darkness of an eclipse results in severe temperature drops; the lack of convective heat flow in space limits the way in which the spacecraft temperature can be controlled, as the only heat exchange with the environment is radiative. The combination of these conditions makes the thermal control of spacecraft a unique challenge.

This section functions as a general introduction to spacecraft thermal control, but for the interested reader, some example thermal control technologies are mentioned in Appendix A.

2.3.1. Simplified Heat Balance

The heat exchange of the spacecraft with the space environment consists of the following aspects [24, p.202][25, pp.21-33], which are also depicted in Figure 2.12:

- Solar heat ($\dot{Q}_{solar}$): the solar spectrum spans a wide range of wavelengths, but the intensity peak is around the visible spectrum. The spacecraft’s surface properties related to sunlight are therefore tailored to visible light, and are notated as the solar absorptivity α.
- Albedo (reflected solar heat, $\dot{Q}_{albedo}$): a fraction of the incoming sunlight reflects off the Earth, towards the spacecraft. The albedo factor a ranges from 0-1, where 0 means that all radiation is absorbed by

Earth, and 1 means that all radiation is reflected. Again, the solar absorptivity α is used to denote the spacecraft's absorptivity of albedo.

- Earthshine (infrared, $\dot{Q}_{Earthshine}$): infrared (IR) radiation emitted by the Earth itself. These are longer wavelengths than Sunlight (and albedo), so the absorptivity α does not apply, but instead the IR emissivity ε is used as a spacecraft surface parameter. It is therefore possible to have a material that emits/absorbs well in the Solar spectrum, but poorly in the IR, or vice versa.
- Spacecraft internal heat ($\dot{Q}_{internal}$): internal electronics generate heat, depending on the activity of the satellite.
- Spacecraft outgoing heat ($\dot{Q}_{out}$): IR radiation emitted by the spacecraft itself. These are a similar wavelength compared to Earthshine, so also the emissivity ε applies.

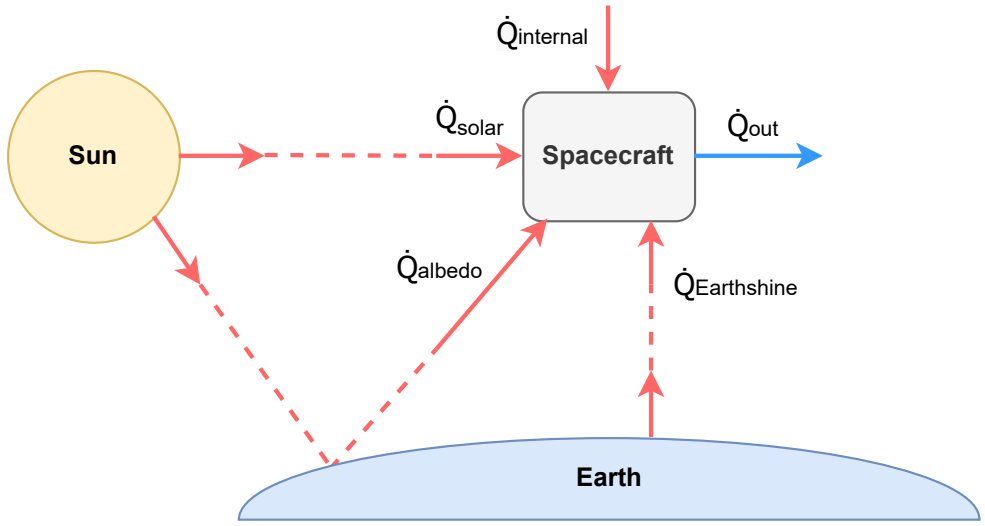


Figure 2.12: Incoming and outgoing heat for a spacecraft orbiting Earth.

The balance between the incoming and outgoing heat determines the change in temperature of the spacecraft. The net heat flux is related to the temperature change via the spacecraft's heat capacity: mass (m) times specific heat capacity (c_{cap}). Therefore, the heat balance equation can be formulated as shown in Equation 2.1 follows (adapted from [24, p.202]). Here, all heat $\dot{Q}$ has the unit [W], the mass m is in [kg], the specific heat capacity c_{cap} is in [J/(kg·K)], and the temperature change dT/dt is in [K/s] or [°C/s].

$$\left(m \cdot c_{cap} \cdot \frac{dT}{dt}\right)_{spacecraft} = \dot{Q}_{solar} + \dot{Q}_{albedo} + \dot{Q}_{Earthshine} + \dot{Q}_{internal} - \dot{Q}_{out} \tag{2.1}$$

This model is denoted as "simplified" since there are a few other components that could have a (small) effect. These components are:

- Free molecular (aerothermal) heating due to the atmosphere [25, pp.32-33]. This could be modelled by an analytical equation such as [25, p.32], or by using Direct Simulation Monte Carlo (DSMC)[2].

$$\dot{q}_{aero} = \alpha \left(\frac{1}{2}\right) \rho V^3 \tag{2.2}$$

where $\dot{q}_{aero}$ is the heat flux in [W/m^2], α is the accommodation coefficient (which is related to gas-surface interaction [26]), ρ is the surrounding gas density in [kg/m^3], and V is the relative velocity between the atmosphere and spacecraft in [m/s].

[2]Wikipedia: `https://en.wikipedia.org/wiki/Direct_simulation_Monte_Carlo`

For example, at an altitude of 350 km (such a low altitude is common for PocketQubes [27]), the mean density is $8.33 \cdot 10^{-12}$ kg/m^3 [28, p.1031]. In the worst-case scenario where $\alpha = 1$ [25, p.32], the aerothermal heating would be approximately 1.9 W/m^2. This value greatly increases for lower altitudes, since the atmospheric density and spacecraft velocity both increase. Hence, this simple analysis should be done in every case to avoid neglecting this heat flux without a solid justification.

- Charged-particle heating [25, pp.34-36]. Charged particles are trapped in the Van Allen belts, and their abundance also depends on solar activity. According to [25, pp.34-36], these particles mainly cause heating when the spacecraft contains cryogenic parts; the lower the temperature, the more charged-particle heating occurs. For instance, a radiator at 100 K at 3200 km altitude can heat up by 2 K due to this effect [25]. If the spacecraft is near room temperature, the heating is negligible [25].

2.3.2. The Difference Between SmallSats and Large Satellites

The temperature band in which the spacecraft must stay is usually dependent on the sensitivity of the components on board. These can be high-end solar cells, or sensitive payloads. Generally, ”better” payloads require a stricter temperature range. This is related to the complexity of the payload; optical components may have nanometre precision, which would be ruined by inconsistent temperatures. Up to now, SmallSats have not had any advanced temperature control, which means that their temperature fluctuates a lot, thereby drastically limiting the payload capabilities [29]. More advanced thermal control concepts from conventional satellites could be adapted for SmallSats, or completely new technologies could be developed. This would all be beneficial in increasing the scientific potential of SmallSats.

Some major differences between the thermal control of SmallSats and that of large satellites are:

- SmallSats have smaller masses than large satellites, which leads to them having smaller thermal capacities. This means that their temperatures fluctuate more when the thermal fluxes change. As a result, it is more difficult to maintain a constant temperature without active thermal control [16] .
- SmallSats have a limited external surface area, making it more difficult to have radiators to reject heat to outer space [16]. In the case of Delfi-PQ, the solar cells take up so much space that there are no dedicated radiators in the current design [6], but only small areas in between the solar cells act as radiators. Since the emissivity of the solar cells cannot be changed for thermal control purposes, it severely restricts the possibilities of sophisticated thermal control.
- Most SmallSats do not have sufficient power to employ active thermal control techniques [16]. This is especially true for satellites with body-mounted solar panels [30], such as Delfi-PQ, since these panels are not pointed continuously at the Sun.
- Most SmallSats do not have a stable attitude, but are freely tumbling. This tumbling complicates the use of radiators; radiators reject heat from the spacecraft to space, but they ideally point away from the Sun, to limit incoming heat. If the radiator points at the Sun unexpectedly, it starts absorbing heat instead of rejecting heat, which is the opposite effect of what is desired.
  Another negative effect of freely tumbling spacecraft is that the exact attitude cannot be predicted in advance; it is determined by the separation of the satellite from the mother-vehicle, which is difficult to predict. An average expected angular velocity may be available during the analysis/design phase [6], but it still limits the reliability of the temperature predictions.

2.4. Thermal Analysis

The very basics of thermal analysis were explained in Section 2.3: the net difference between incoming and outgoing heat results in a temperature change of the satellite, and the internal heat transfer occurs through radiation and conduction. However, a lot more complexity arises when implementing the theory into a computational program.

The entire thermal analysis procedure consists of multiple steps, which are mostly universally agreed upon by thermal engineers. This section describes this analysis architecture, closely following ECSS guidelines [31] and the Spacecraft Thermal Control Handbook (Gilmore) [25]. An overview of the whole thermal analysis process is shown in Figure 2.13. The main importance is the separation between the left and right branches of the diagram. The left branch represents the Geometrical-Mathematical Model (GMM), and the right branch represents the Thermal-Mathematical Model (TMM).

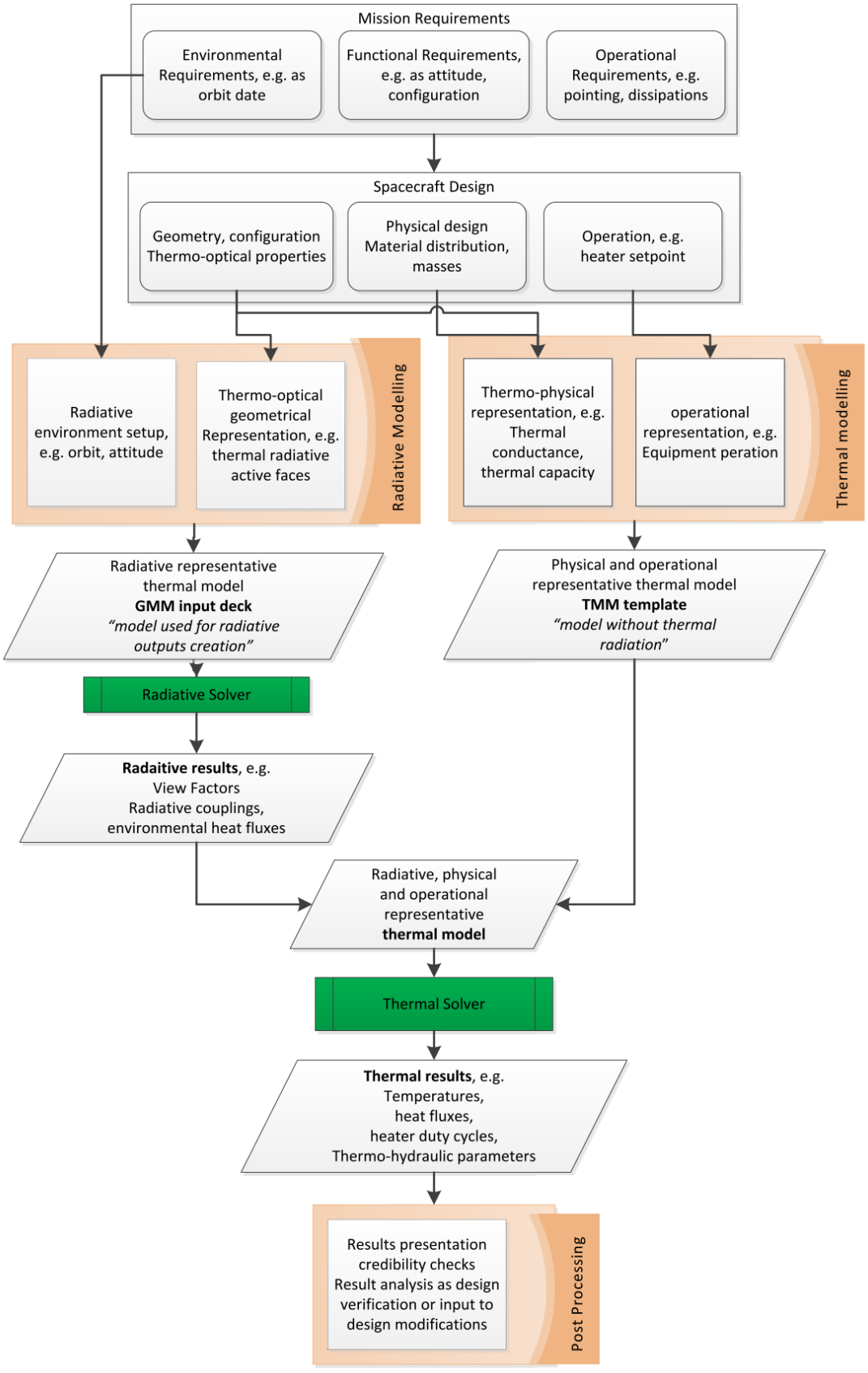


Figure 2.13: Thermal modelling process example by the ECSS [31].

2.4.1. Geometrical-Mathematical Model (GMM)

The GMM is a type of model that consists of geometrical shapes and surfaces, each having their own radiative properties. The goal of this model is to calculate radiative exchange factors between all surfaces (both the external radiation from the Sun and Earth, and the internal radiation). Therefore, this model does not contain inner conductive couplings within the spacecraft, and does not predict any temperatures [25, pp.536-552] [31].

Within the GMM, an orbital model must be included, which is also seen in Figure 2.13. The orbital model calculates the spacecraft's attitude with respect to the Sun and Earth throughout the mission lifetime, hereby also simulating eclipse times. This is not further discussed here, but details can be found in [7, 17, 25]. Furthermore, a method for internal radiative exchange factors must be included, which is generally either using (simplified) view factor calculations from the ECSS standard [32], or using numerical Monte Carlo ray tracing (MCRT) [31, 33].

Internal Radiation

Not only is radiation the single heat transfer phenomenon by which heat enters and leaves the spacecraft, it also plays a large role internally. The smaller the satellite, the more important radiation becomes, up to 80% for PocketQubes, which is shown in Figure 2.14 [4].

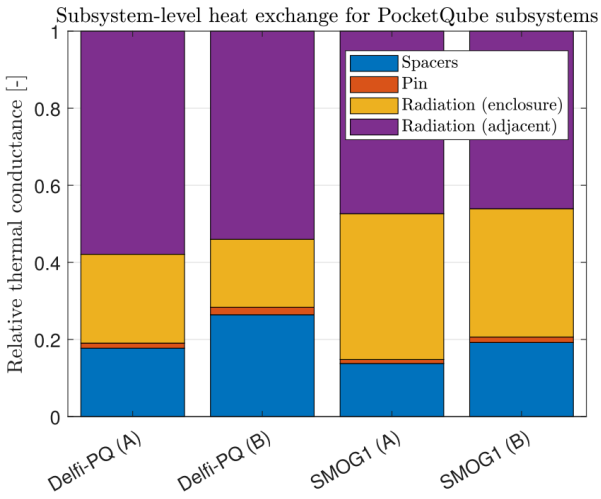


Figure 2.14: Worst-case scenario for relative thermal conductance of the internal heat transfer within Delfi-PQ and SMOG-1 [4, p.44].

The radiation exchange between two objects can analytically be defined using view factors (F_{ij}) and Gebhart factors (B_{ij}). The former describes what fraction of the total field of view of one surface the other surface occupies. The latter is similar, but also takes multiple reflections into account. The heat flow from body 1 to body 2 can then be defined as [34]:

$$\dot{Q}_{12} = \varepsilon_1 A_1 B_{12} \left(\sigma T_1^4 - \sigma T_2^4\right) = R_{12} \left(\sigma T_1^4 - \sigma T_2^4\right) \tag{2.3}$$

where $\dot{Q}_{12}$ is the heat flow from body 1 to body 2 in [W], ε_1 is the emissivity of body 1, A_1 is the surface area of body 1 in [m^2], B_{12} is the Gebhart factor from body 1 to 2, σ is the Stefan-Boltzmann constant ($\sigma = 5.670374419 \cdot 10^{-8}$ W/(m^{2}K^4) [1]), T_1 is the temperature of body 1 in [K], T_2 is the temperature of body 2 in [K], and R_{12} is commonly referred to as the radiative exchange factor. The Gebhart factor is in turn defined as [34]:

$$B_{ij} = F_{ij}\varepsilon_j + \sum_{k=1}^{n} \left[\left(1 - \varepsilon_k\right) F_{ik} B_{kj}\right] \tag{2.4}$$

where F_{ij} is the view factor from surface i to surface j. The index i is the emitting surface, the index j is the receiving surface, and the indices k are all surfaces in the system off which the rays can reflect. The emitting surface itself is also included in the summation over k, since a ray can reflect off its original surface when that surface is concave. The exact definition of the view factor is [32]:

$$F_{12} = \frac{1}{A_1} \int_{A_1} \int_{A_2} \frac{cos\,(\beta_1)\, cos\,(\beta_2)}{\pi S^2} dA_2 dA_1 \tag{2.5}$$

where β are the angles in [rad] between the normal of the surface and the line between the two surfaces, and S is the distance between the two surfaces in [m].

The view factor F_{ij} is coupled to the view factor F_{ji} via the reciprocity relation [34]:

$$F_{ij} A_i = F_{ji} A_j \tag{2.6}$$

Similarly, the Gebhart factors G_{ij} and G_{ji} are related to each other via another version of the reciprocity relation [34]:

$$\varepsilon_i A_i B_{ij} = \varepsilon_j A_j B_{ji} = R_{ij} = R_{ji} \tag{2.7}$$

Another property of view factors and Gebhart factors is that, for a given emitting surface, they must add up to unity, since all emitted energy is conserved [34]. These summations include $j = i$.

$$\sum_{j=1}^{n} F_{ij} = 1 \tag{2.8}$$

$$\sum_{j=1}^{n} G_{ij} = 1 \tag{2.9}$$

The aforementioned can be implemented in software in two ways:

1. **Simplified method using analytical view factors**: as described before, and only using a standard set of view factor expressions from ECSS. This limits the types of geometries that can be used, but also greatly simplifies the calculations. For example, it could be assumed that all surfaces within the spacecraft are parallel or perpendicular [33]. These are the two cases from ECSS shown below: Figure 2.15 and Equation 2.10 are for two perpendicular plates, and Figure 2.16 and Equation 2.11 are for two identical parallel plates.

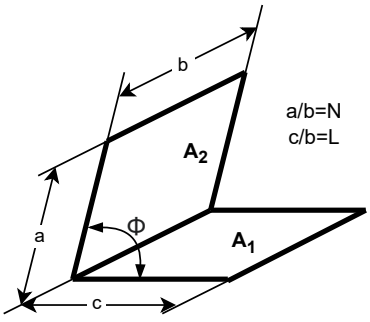


Figure 2.15: View factor geometry for two angled plates (adapted from [32, p.37]). $\Phi = 90°$C is the case for perpendicular plates.

$$F_{12} = \frac{1}{\pi L}\left\{\begin{array}{l} Larctan\left(\frac{1}{L}\right) + Narctan\left(\frac{1}{N}\right) - \sqrt{N^2+L^2}arctan\left(\frac{1}{\sqrt{N^2+L^2}}\right) + \\ +\frac{1}{4}ln\left[\frac{(1+N^2)(1+L^2)}{(1+N^2+L^2)}\cdot\left(\frac{L^2(1+N^2+L^2)}{(1+L^2)(N^2+L^2)}\right)^{L^2}\cdot\left(\frac{N^2(1+N^2+L^2)}{(1+N^2)(N^2+L^2)}\right)^{N^2}\right] \end{array}\right\}, \text{ where: } N = \frac{a}{b}; L = \frac{c}{b} \tag{2.10}$$

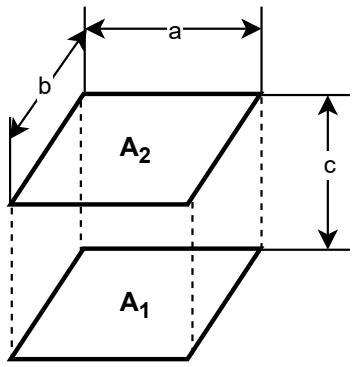


Figure 2.16: View factor geometry for two identical parallel plates (adapted from [32, p.35]).

$$F_{12} = \frac{2}{\pi XY}\cdot\left\{\begin{array}{c} ln\left[\frac{(1+X^2)(1+Y^2)}{1+X^2+Y^2}\right]^{1/2} + X\sqrt{1+Y^2}arctan\left(\frac{X}{\sqrt{1+Y^2}}\right) + \\ +Y\sqrt{1+X^2}arctan\left(\frac{Y}{\sqrt{1+X^2}}\right) - Xarctan\left(X\right) - Yarctan\left(Y\right) \end{array}\right\}, \text{ where: } X = \frac{a}{c}; Y = \frac{b}{c} \tag{2.11}$$

2. **Monte Carlo Ray Tracing (MCRT)**: ESATAN-TMS uses Monte Carlo ray tracing to approximate the radiative exchange between all surfaces. A Monte Carlo method is a stochastic computation technique to numerically solve problems. Ray tracing is simply to follow a beam of radiation as it reflects off of different surfaces. Hence, Monte Carlo ray tracing refers to firing random rays from a surface, checking which other surfaces it hits, and hence determining the view factors / Gebhart factors between surfaces [35]. The number of reflections of each beam can be determined by the user, or could be based on a minimum beam energy threshold (e.g., the beam could be stopped when the energy is below 1% of its original value). A sample algorithm is shown in Figure 2.17; this algorithm does not compute the view factor separately, but immediately assesses the heat transfer. In case only the view factor is to be calculated, Equation 2.12 is used. Here, N_{ij} is the number of emitted rays from surface i hitting surface j, and N_i is the total number of rays emitted from surface i. Depending on whether multiple reflections are taken into account, the view factor would instead be the Gebhart factor.

for each $i \in Particles$

do
- generate random point on particle
- generate random direction (pointing outside of particle i)
- calculate heat flux using Stefan Boltzmann's law
  $\dot{Q}_i \leftarrow \varepsilon_i \sigma A_i T_i^4$
- decrease heat flux of particle i by $\dot{Q}_i$
- trace ray for intersections with other particles
- **if** intersection with particle j
  - **then**
    - transfer absorbed part of heat: $\dot{Q}_i \varepsilon_j$
    - generate random direction (that points outside of particle j
    - shoot new reflection ray with $\dot{Q}_j \leftarrow (1-\varepsilon_j)\dot{Q}_i$
      (recursively, up to depth N)
  - **else** (background radiation) increase heat flux of particle i by $\varepsilon_i \sigma A_i T_{\text{background}}^4$

return

Figure 2.17: Example of a Monte Carlo ray tracing algorithm [36].

$$F_{ij} = \frac{N_{ij}}{N_i} \tag{2.12}$$

A comparison of the two methods for radiative calculations is shown in Table 2.2. The judgements for the scores on analytical view factors are speculative, since [33] appears to be one of the few to elaborate on the use of the ECSS equations [32] for view factor determination.

- **Simplicity** is considered highly important, especially when the analysis tool is to be developed by students. Ray tracing is complicated and a lot of effort can be spent optimising the algorithms[3]. The view factor approach simply uses a standardised set of equations from ECSS [32].
- **Computational Cost** has a low priority, as long as the computation times are not unreasonably long (which could entail waiting hours for a single radiative case). From the author's experience using ESATAN-TMS, the ray tracing approach does not exceed a few minutes per entire orbital analysis. Custom-built algorithms are likely to be less efficient, and it could be something to investigate in more detail.
- **Reliability** is moderately important, since SmallSat missions have less strict tolerances [29, 7], but it is still suggested by [4] that radiation could account for up to 80% of all internal radiation in a CubeSat. The view factor approach is less reliable than Monte Carlo ray tracing, since it is based on numerous assumptions, which were explained in the previous section [33]. It is also expected (but not yet proven) that the reliability of the view factor approach depends on the conditions, such as satellite size and distance between panels.
- **Flexibility** is considered highly important, and is essentially referring to modularity. The view factor approach would only be valid under a fixed number of circumstances for which the equations [32] are defined, which could restrict the satellite's design, while the ray tracing approach does not depend on such assumptions. However, the ray tracing approach is not highly flexible, since a simple non-geometric approach such as the one from [7] would not suffice, and a completely different geometrical model definition would have to be implemented.

[3] Wikipedia: `https://en.wikipedia.org/wiki/Path_tracing`

Table 2.2: Comparison of simplified view factor analysis and Monte Carlo ray tracing, from the point of view of SmallSats. The column width represents their importance for SmallSats; ”L”, ”M”, and ”H” are for ”low”, ”medium”, and ”high”; and the cell colours represent whether the property is desirable (green), neutral (yellow), or undesirable (red).

Radiative computation	Simplicity	Computational Cost	Reliability	Flexibility
View factors	H	L	M	L
Monte Carlo ray tracing	L	H	H	M

2.4.2. Thermal-Mathematical Model (TMM)

After the GMM has produced the external heat loads and the internal radiative exchange factors, those results can be fed into the TMM [31]. The TMM is a lumped parameter model, which means that the spacecraft is discretised into a finite number of nodes, where each node has its own thermal capacitance [J/K] and conductance [W/K] to other nodes [31] [25, pp.536-552]. The nodal model can be drawn as a thermal network, which has similar properties and calculation rules as electrical resistance-capacitance networks [37] [25, pp.536-552].

The result of the TMM calculation is the expected temperatures of all nodes [31]. For one set of input/output heat values, one temperature distribution results. As also shown in Figure 2.13, a numerical solver is needed for complex networks (although hand calculations can be done on simplified networks [37]).

Conduction

Conduction is the heat transfer method for TMM networks, and there are two types of conduction: through a single material, and contact interface conductance between two materials. The following formulas are adapted from [34].

The heat flux from point 1 to point 2 through a single material is calculated with Equation 2.13, and the heat flux through a contact interface is calculated with Equation 2.14:

$$\dot{q}_{12_{through}} = k \cdot \frac{T_1 - T_2}{L} \tag{2.13}$$

$$\dot{q}_{12_{contact}} = h \cdot (T_1 - T_2) \tag{2.14}$$

where $\dot{q}_{12_{through}}$ is the heat flux from 1 to 2 in [W/m^2], k is the thermal conductivity of the material in [W/(m·K)], T is the temperature in [K], L is the distance between the nodes in [m], $\dot{q}_{12_{contact}}$ is the contact heat flux from 1 to 2 in [W/m^2], and h is the heat transfer coefficient in [W/(m^2·K)].

The conductance C_{con} in [W/K] can also be defined for these two types of conduction:

$$C_{con,through} = \frac{k \cdot A}{L} \tag{2.15}$$

$$C_{con,contact} = h \cdot A \tag{2.16}$$

where A is the cross-sectional/contact area in [m^2].

2.4.3. Alternative Analysis Architectures

Traditionally, the GMM and TMM are separate software, and have separate models, even though they represent the same mission. However, there is a trend towards other architectures, where data flow between the GMM and TMM could be improved, or where the data flow between other subsystem analyses

could be improved, such as:

- **Combined GMM & TMM**: [31] claims that the line between the GMM and TMM blurs in some cases, when the TMM is immediately calculated during the GMM calculation. Not much literature is available on this topic, but it could be valuable to investigate this option experimentally. One argument for a single integrated software could be that the user experience improves, since one model would be sufficient for the entire analysis.
- **Modularity** approach: modularity is highly relevant for SmallSats, considering that SmallSats are often built with COTS and are standardised in terms of size [11, 33, 6]. In that sense, the thermal analysis process can also be very different from large missions, since those are much more specialised and require specialised analysis. However, several authors [6, 33, 7] display their concern for this complexity of traditional thermal analysis, since university SmallSats are often worked on by students or engineers who are not specifically experts in thermal analysis.

  Firstly, a modular analysis tool would include a component database [6, 33]: many parts in SmallSats are commonly used, such as the PCB type, structural rods, solar cells, etc. If each SmallSat developer would have to experimentally verify all the thermal capacitance and conductance values of all components, it would complicate and slow down the thermal analysis. Instead, if the thermal properties of those commonly used parts are included in the software, it provides a much simpler solution, which could be good enough depending on the application. Previous authors [38, 4] have conducted such experiments (or invented methods to simplify those experiments) to determine the thermal properties of common SmallSat components.

  Secondly, a modular tool would include some capability from the software side to easily connect components together. [33] uses ”potential connectors”, which are pre-defined on the database components. They correspond to the physical connections in a SmallSat.
- **Digital twin** approach: in the context of spacecraft, a digital twin is virtual copy of the physical spacecraft (”physical twin”), that exchanges data with the physical twin [39, 40], and it is intrinsically related to Model-Based Systems Engineering (MBSE). It is not specifically related to the thermal subsystem, but more to the entire spacecraft. Nevertheless, a similar architecture could be used to improve the thermal design by means of this one-on-one data flow between the physical and digital twin. Furthermore, it would allow to easily spot discrepancies between the model and reality during operation, since the flight data is immediately fed into the digital twin.

  As seen in Figure 2.18, a digital twin can already be created during the conceptual phase. Since not all spacecraft data is known yet at that stage, it is important to make the model robust [39], preventing singularities and other errors. Furthermore, it is crucial that the digital twin and physical twin have a number of fixed reference points which are monitored [40, 39], to allow for one-on-one data comparison.

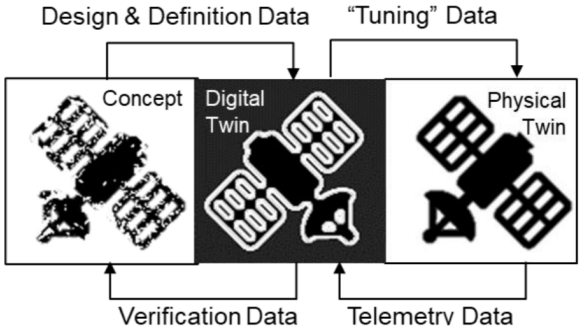


Figure 2.18: Digital twin concept for a spacecraft [39].

A summary of the aforementioned thermal analysis architectures is shown in Table 2.3. The importance (width) of each column and the scores in all cells, for SmallSats specifically, are judged based on common remarks from several authors:

- **Simplicity** has a high priority, since it is the design philosophy of SmallSats [11], and since the complexity of commercial thermal analysis tools is disliked by non-experts in the thermal analysis field [6, 33, 7].

  The simplicity of the modular approach is the highest of all, since it is essentially a traditional approach, but with a number of simplifications such as the component database [6, 33]. The combined GMM&TMM and digital twin approaches are complex, since they both rely on large all-encompassing models.
- **Heritage** has a low priority, since it would mostly be an advantage for expensive, large satellites, but SmallSats generally have less strict tolerances [29, 7] and a lower consequence associated with failure [11].

  The heritage is the worst for the digital twin, due to its novelty [40], and moderate for the modular approach. The traditional GMM&TMM approach is by far the most commonly used [31].
- **Data flow** refers to data management within the thermal model, and with data from other subsystems. It is moderately important, since good data flow results in greater simplicity for the user of the tool.

  The data flow of the traditional approach is relatively poor, since multiple models are needed (GMM & TMM separately). The digital twin has the best data flow, since it is possible to represent the entire spacecraft (to some extent) as a single digital entity; N2-charts can be used to represent the data flow between all subsystems [39].
- **Verification and validation** has a high priority, since this is an area where previous projects lack resources or experience [3, 4].

  The verification and validation is the worst for the modular approach, since many assumptions are made in the process, which may not always be perfectly justifiable [6, 33]. The digital twin has the best verification and validation potential, due to the great data flow with the physical twin.

Table 2.3: Summary of thermal analysis architectures. The column width represents their importance for SmallSats; ”L”, ”M”, and ”H” are for ”low”, ”medium”, and ”high”; and the cell colours represent whether the property is desirable (green), neutral (yellow), or undesirable (red).

Architecture	Simplicity	Heritage	Data Flow	Verification/ Validation
Traditional GMM&TMM	M	H	L	M
Combined GMM&TMM	L	M	M	M
Modular Approach	H	M	M	L
Digital Twin	L	L	H	H

2.4.4. Example Thermal Analysis Software

The most commonly used commercial thermal analysis tools are ESATAN-TMS[4], Airbus Thermica[5], Ansys Thermal Desktop[6], and COMSOL Heat Transfer Module[7]. There have also been open-source alternatives such as the MATLAB code developed by C. Ruiz [7], and the MATLAB app developed by B. Mattos [41], which are free to use (although MATLAB is paid). Limitations of commercial tools are that they can feel like a black box: it is never fully known what happens exactly when you run the simulation, and, if even possible, adding personalised features requires a very deep dive into the software. On the other hand, limitations of the open-source software are often that they lack full functionality:

[4]ESATAN-TMS: `https://www.esatan-tms.com/`

[5]Airbus Thermica/Thermisol: `https://www.airbus.com/en/products-services/space/customer-services/systema/thermica-and-thermisol`

[6]Ansys Thermal Desktop: `https://www.ansys.com/products/fluids/ansys-thermal-desktop`

[7]COMSOL: `https://www.comsol.com/heat-transfer-module`

- The tool by Ruiz calculates the environmental radiation and internal conduction, but does not include internal radiation inside the spacecraft. This could be a severe issue, as it was previously suggested (Section 2.4.1) that internal radiation can be up to 80% of the total internal heat flow. If radiation is to be added in this tool, the nodal definitions should also be changed, because they are currently non-geometric, only having a total surface area and distance to other nodes;
- The tool by Mattos calculates the environmental radiation and has a few additional features such as deployable solar panels (including the shadows they cast). However, the tool only supports standard CubeSat dimensions, and the model of the CubeSat itself is only the nodes of the outer surfaces, and a single inner node.

Advantages and disadvantages of the aforementioned types of tools are summarised below in Table 2.4. It should be noted that some scores depend on the exact tool, but the scores given here are what is generally expected from either commercial or open-source software:

- **Cost** is moderately important for university projects. The main reason why it is not highly important is that licenses for these tools are in some instances already available for other purposes within the university.

  Open-source software are free, while commercial software are not. However, some software such as ESATAN-TMS are already available at the university or other institutes.
- **Learning Curve** is highly important, since the timelines for SmallSats are rapid, and the engineers may not be experts in the field [3, 11].

  The learning curve of commercial software is generally really high [6], and the black box type of workflow can make the understanding of the program even more challenging. Open-source code is usually simpler and can be made for a specific purpose [41].
- **Modularity** is moderately important, as it does influence simplicity, but it is not as crucial as the tool having a reasonable learning curve.

  Open-source tools allow for modularity, since the possibilities are literally endless when the tool can be completely modified to your liking [41]. Commercial tools can have custom-made plug-ins, but they must comply with all the software's interfaces, still limiting the user's freedom.
- **Reliability** has a low priority, since it would mostly be an advantage for expensive, large satellites, but SmallSats generally have less strict tolerances [29, 7] and a lower consequence associated with failure [11]. Still, the reliability can be optimised by performing adequate verification and validation.

  Commercial tools have the main advantage of a high reliability. ESATAN-TMS has been extensively used in previous mission analysis [31], and has undergone years of refinement by experts. Open-source tools have less development time and fewer developers, while also often lacking "flight heritage", not having been widely used in the past [7, 41].

Table 2.4: Summary of spacecraft thermal analysis tools, from the point of view of a university project. The column width represents their importance for SmallSats; "L", "M", and "H" are for "low", "medium", and "high"; and the cell colours represent whether the property is desirable (green), neutral (yellow), or undesirable (red).

Type of software	Cost	Learning Curve	Modularity	Reliability
Commercial	M	H	M	H
Open-source	L	M	H	L

3

Methodology

This chapter aims to provide a complete overview of the computations behind the thermal analysis software, starting from the high-level computation architecture, to the low-level algorithms and formulas. The overview of the thesis methodology was explained in the introduction, in Chapter 1.

First, Section 3.1 presents the requirements and their rationale. Second, a broad overview of the software and a number of functional block diagrams are given in Section 3.2. Third, Section 3.3 explains how all external heat sources are computed in the orbital model. The computations of the external heat sources are based on the Matlab code from C. Ruiz [7], but there are some differences, which are indicated clearly. Moreover, the overall software architecture of the tool by Ruiz is vastly different from the newly developed Python tool, mainly because the Python tool is more modular and object-oriented. Fourth, the computation of all the heat flows in the nodal model is explained in Section 3.4. Finally, Section 3.5 gives an example case of the FUNcube-1 satellite, where the most important aspects of the software are shown from the user's perspective.

3.1. Requirements

The thermal analysis software is designed specifically for CubeSats and PocketQubes, and hence, the requirements for such a software are vastly different from larger satellites. Some requirements are related to the accuracy of the computations, while others are related to the user experience. For REQ-TH-06, Figure 3.1 provides a rationale, showing that most SmallSats operate in LEO.

Table 3.1: Requirements for the thermal analysis software.

Requirement ID	Description	Rationale
	High-Level Requirements	
REQ-TH-01	The software shall be centered around modularity.	This means that it is easy to copy-paste existing nodal models, change them on the fly, attach certain models to others, etc. This all speeds up and simplifies the making of a thermal model.
REQ-TH-02	The software shall be verified with ESATAN-TMS.	ESATAN-TMS is the industry standard thermal analysis tool (in Europe), and the TU Delft has licenses for this software.
REQ-TH-03	The software shall be validated with flight data of at least one small satellite.	More validation data can always be compared with the software at a later stage, but for this thesis, validating the software with just one set of flight data shall be sufficient for the prove-of-concept of this software.

REQ-TH-04	The software's orbital module and thermal (nodal) module shall be able to function independently of each other.	To model in-orbit thermal behaviour of a satellite, the orbital and thermal module should evidently be smoothly interfacing with each other. However, it should not be forgotten that on-ground validation tests do not require an orbital model, but only function as a thermal (nodal) model. Hence, for validation purposes, it is crucial that the orbital and thermal models can function independently of each other.
REQ-TH-05	The software shall contain easy-to-use functionalities for non-specialists.	In many cases, SmallSat design teams do not have dedicated thermal engineers, so there should be features present that allow intuitive model building.
REQ-TH-06	The software shall be able to perform orbital analysis for circular Low-Earth Orbits (LEO).	Most SmallSats orbit in LEO (see Figure 3.1), and hence the code can be kept simple by avoiding complex orbit propagators. One should look at the sensitivity analysis in Chapter 5 to find the consequences of the LEO assumption. The maximum allowed altitude is likely limited by de-orbiting guidelines; these may be checked for the mission, after which the LEO assumption could be re-assessed.
REQ-TH-07	The source code shall have fully documented functions and classes with the standard docstring definition.	This is extremely important to ensure an easy use of the software without always having to refer to external user manuals and examples, or worse, if the user does not have access to the user manual.
REQ-TH-08	The source code shall have a documentation on GitLab as a wiki page.	The GitLab wiki page is similar to a readme file, where the project is described and a file overview is given. Information can be easily accessed and changed, not being limited to local PC files.
REQ-TH-09	The material database shall be available online, open-source, and users must be able to easily add their own data to the database.	The database can be an Excel sheet included in the GitLab repository of the code; such a sheet is easily modifiable by the user. Ideally, different users could share their findings such that the entire scope of the database can be increased beyond that of a single user. However, a control group or mechanism would be required to ensure that no faulty data is added.
REQ-TH-10	The software shall contain a sensitivity analysis module that can be applied to an arbitrary nodal model.	The sensitivity analysis should not only be performed once for this thesis, but it is a type of analysis that is useful for any SmallSat developer. Hence, it should be possible to apply the software for the sensitivity analysis to any custom-made nodal model.
	Low-Level Requirements	
REQ-TH-11	The software shall be written in the Python coding language.	One of the main goals of this software is to be easily accessible to SmallSat engineers (amongst which many are students), in terms of coding language and the need for licenses. Python is extremely popular among engineers and students, it is open-source, and completely free. Furthermore, it is specifically taught at the Aerospace Engineering faculty at TU Delft. This is why Python is preferred over similar alternatives such as MATLAB; it has a larger audience and no licenses are needed.

REQ-TH-12	The software shall use predefined default parameters for variables that the user does not enter.	Detailed parameter definition should be optional for the user, such that the user can make the model as simple or complex as they desire. For example, if a node has no input power, the code should use zero power as a default parameter, instead of requiring the user to manually define it as zero. A downside of this is that the user should read the docstrings in detail to be aware of these assumptions.

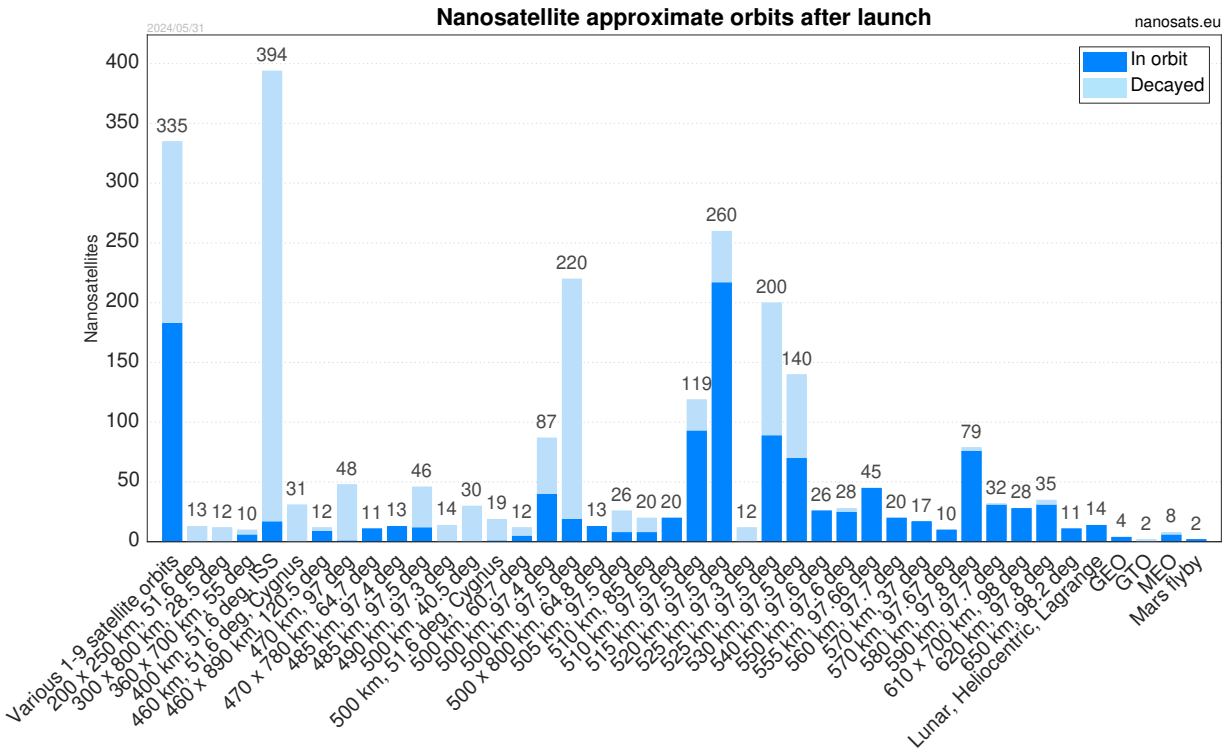


Figure 3.1: Number of small satellites launched per type of orbit [10]. As seen, most orbits are in Low-Earth Orbit, below approximately 600 km.

3.2. Software Overview & Block Diagrams

The code is object-oriented, and most of the functionalities and computations are embedded in these objects. Block diagrams of the three largest objects are shown in Figure 3.2, Figure 3.3, and Figure 3.4. Many features in the software were greatly simplified in these block diagrams, so it is recommended to visit the source code for more details. The code contains elaborate docstrings that explain what each function does and what the inputs and outputs are.

Furthermore, the ”external functions” mentioned in the diagrams are not given a detailed diagram, but the computations are simply explained in the remaining part of this chapter. One exception is the `heat_received` function, which is written in a separate Python file due to its complexity and functional-programming-style. This function is visualised in Figure 3.5.

Details about the actual usage of all classes and functions are given in the user’s manual in Appendix C, and Section C.6 specifically shows the meaning of all classes, functions, and their inputs and outputs.

3.2.1. Class OrbitalModel

The `OrbitalModel` class (Figure 3.2) defines the orbital parameters of the spacecraft, and provides a function to compute the Solar flux, albedo flux, and Earth infrared flux on the given surface or collection of surfaces (for the orbital model, a SmallSat is essentially a collection of 6 surfaces, see Figure 3.7 and Figure 3.8). The following functions are available:

- `__init__`: initialises an OrbitalModel object. Not all input parameters are mandatory to be entered by the user, but that is specified in more detail in the user manual (Appendix C).
- `modify`: allows the user to modify any parameter of the orbital model. If any parameter related to the time array or surfaces is changed, any previously computed heat fluxes (outputs of `compute`) will be erased. The modify function is mainly useful for the sensitivity analysis, when an existing model must be changed frequently.
- `compute`: computes the external heat on the surface(s) throughout time, and the results are stored in the OrbitalModel object itself. If the inputs for the surfaces are actual Node objects (instead of only strings such as 'x+'), the heat fluxes are automatically stored in there as well.
- `animate_attitude`: shows a 3D animation of the attitude of the surface(s) throughout time. This is useful to verify whether the angular rates are as desired, and it can be a useful visual feedback on why the external heat fluxes are a certain way. For example, if the 'x+' face is much hotter than the other faces, even though it is rotating, it might be that the eigenaxis of the rotation is approximately through the 'x+' face, hence it faces the Sun a lot more than the other plates.
- `get_heat`: after the `compute` function has been run, this returns the external heat fluxes. This can be used to assign the heat fluxes to the appropriate Node objects. If the Node objects were directly entered in the OrbitalModel, this is done automatically within the compute function.

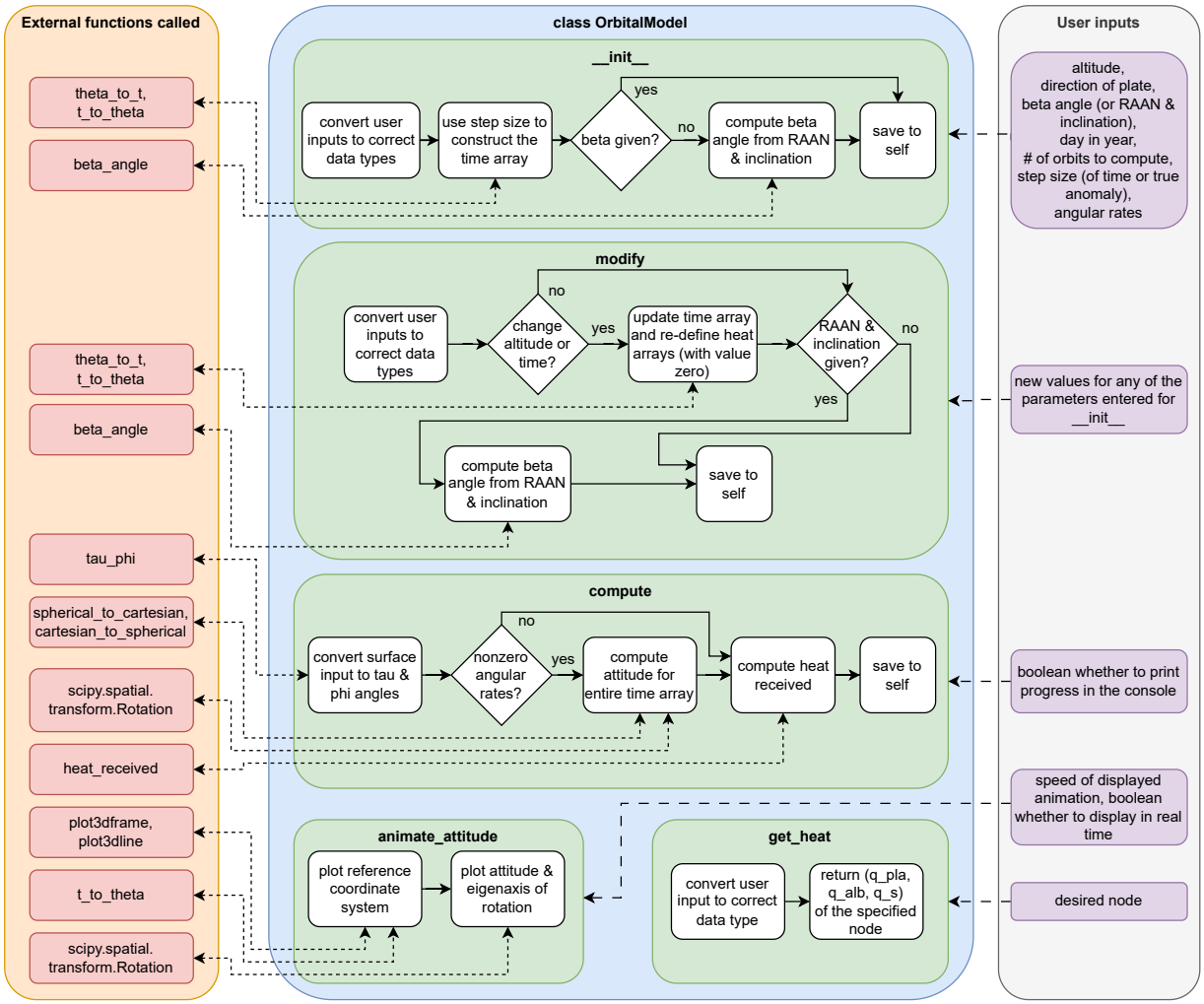


Figure 3.2: Overview of the OrbitalModel Python class, showing its functional flow, and the external functions or user parameters used. Refer to Section C.6 for details on all the functions and parameters.

3.2.2. Class Node

Class `Node` (Figure 3.3) is used to define a single node (lumped mass), with a reference name, thermal mass, optical properties, etc. (see Figure 3.3, 'User inputs') to fully define that one node. The node itself cannot be solved on its own, but has to be assigned to a NodalModel first. Then, it can be solved for a given time and for given heat inputs.

- `__init__`: initialises a Node object. The powers (external heat fluxes and internal power) may be a constant value, or varying throughout time (a NumPy array of values). Not all parameters are mandatory to be given by the user, but that is specified in more detail in the user manual (Appendix C).
- `modify`: allows the user to modify all parameters of a pre-existing Node object. The modify function is mainly useful for the sensitivity analysis, when an existing model must be changed frequently.

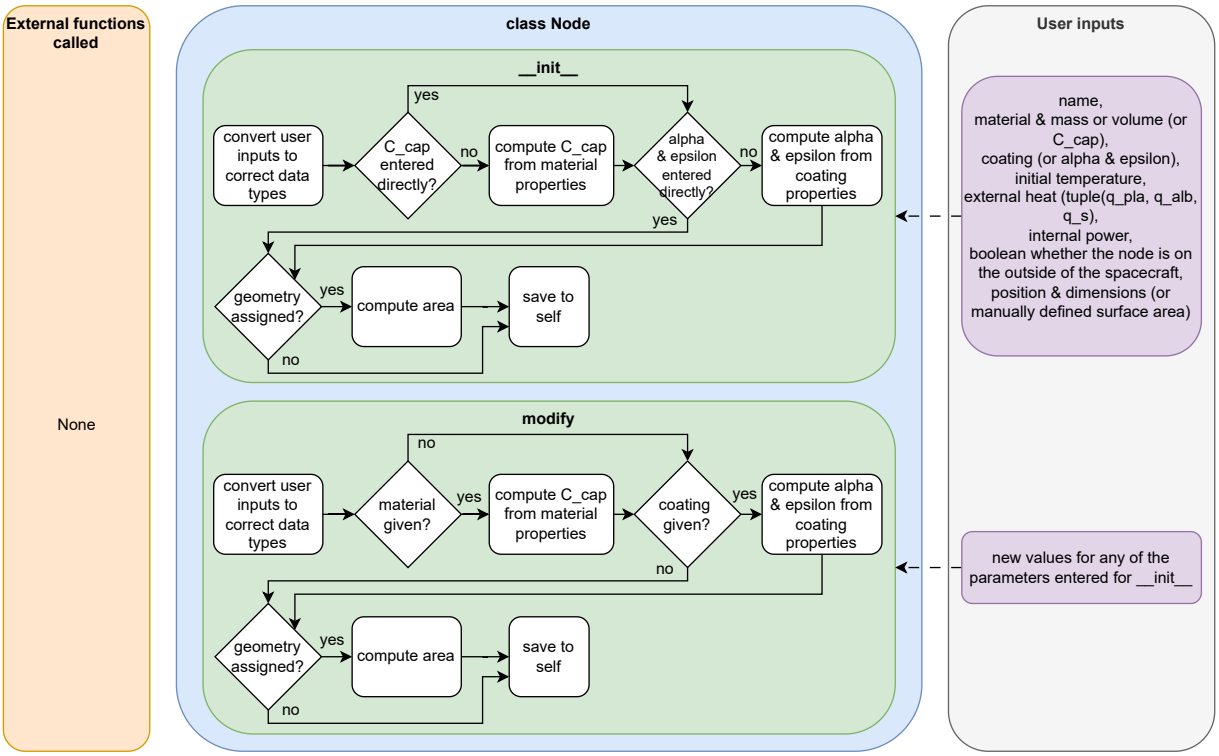


Figure 3.3: Overview of the Node Python class, showing its functional flow, and the external functions or user parameters used. Refer to Section C.6 for details on all the functions and parameters.

3.2.3. Class NodalModel

The class `NodalModel` (Figure 3.4) is a collection of Nodes or other NodalModels, with all connections between the nodes. The NodalModel object is the foundation of the thermal simulation, and can be used completely independently from the OrbitalModel. In that case, custom power inputs can be used.

In principle, a simple NodalModel can have a number of Node objects assigned, and that can be sufficient. However, to achieve a more modular approach, it is possible to add a NodalModel (e.g., a PCB with numerous nodes) to another NodalModel (e.g., the full satellite). While adding one NodalModel to another, it keeps all the properties of the models and copies them to one big matrix used for the calculations. In this manner, a hierarchical tree is created within the NodalModel, and this can be of arbitrary depth.

For example, it is perfectly possible to have a PCB NodalModel (including e.g. 5 nodes for the PCB and 1 node for a battery), embedded in a stack NodalModel (all PCBs and spacers in between them), embedded in the spacecraft NodalModel (the stack and outer panels etc.). This keeps the model modular and also makes it easier to keep track of all the nodes. The hierarchical tree would be of the following form (such a tree is always printed with the show_plots function):

- `NodalModel` Spacecraft
  - `Node` Structural ring A
  - `Node` Structural ring B
  - ...
  - `NodalModel` Stack
    - `Node` Spacer A
    - `Node` Spacer B
    - ...
    - `NodalModel` PCB A
      - `Node` PCB A centre
      - `Node` PCB A corner 1
      - `Node` PCB A corner 2
      - `Node` PCB A corner 3
      - `Node` PCB A corner 4
      - `Node` PCB A battery
    - `NodalModel` PCB B
      - `Node` PCB B centre
      - ...
    - ...

The NodalModel object contains the following functions:

- `__init__`: initialises a NodalModel object. Not all parameters are mandatory, but that is specified in more detail in the user manual (Appendix C).
- `add_node`: add another Node or NodalModel to the main NodalModel. Adding the node does not yet assign any thermal connections between nodes; that is done with the connect function. It should always be ensured that, if the Nodes or NodalModels have a time array of a certain length, that array must be identical to the model to which it is added. If they are different (e.g., one array ends at t=100 seconds and another ends at t=200 seconds), they are not compatible, and the software will print a warning in the console.
- `set_time`: define a time array in case it does not exist yet. Usually, the OrbitalModel is defined first, which generates a time array. That array can then be assigned to the NodalModel. However, it is also possible to define the NodalModel without a time array. Then, it cannot be solved, but it makes it easier to exchange different models between each other, since there are no incompatible time arrays assigned.
- `connect`: connect multiple nodes with a conductive and/or radiative connection. There are many ways to define a connection, for example, using the material database, or manually defining the number of Watts per Kelvin. More details on this are shown in the user manual (Appendix C).
- `modify_node`: allows the user to modify all parameters of a pre-existing Node in the NodalModel. The Node object also has its own modify function, but since the NodalModel contains many matrices with the information of all nodes, simply updating the Node itself is not sufficient. The modify_node function takes this into account, and changes both the Node itself, as well as all the matrices in the NodalModel.
- `solve`: solves the NodalModel for temperatures and heat flows throughout time. It stores all results in the NodalModel itself. It can also perform a steady-state computation to use as initial condition, if no initial temperature is given.
- `show_plots`: shows a drawing of the NodalModel, and plots of all temperatures, heat inputs, and heat exchanges between all nodes.

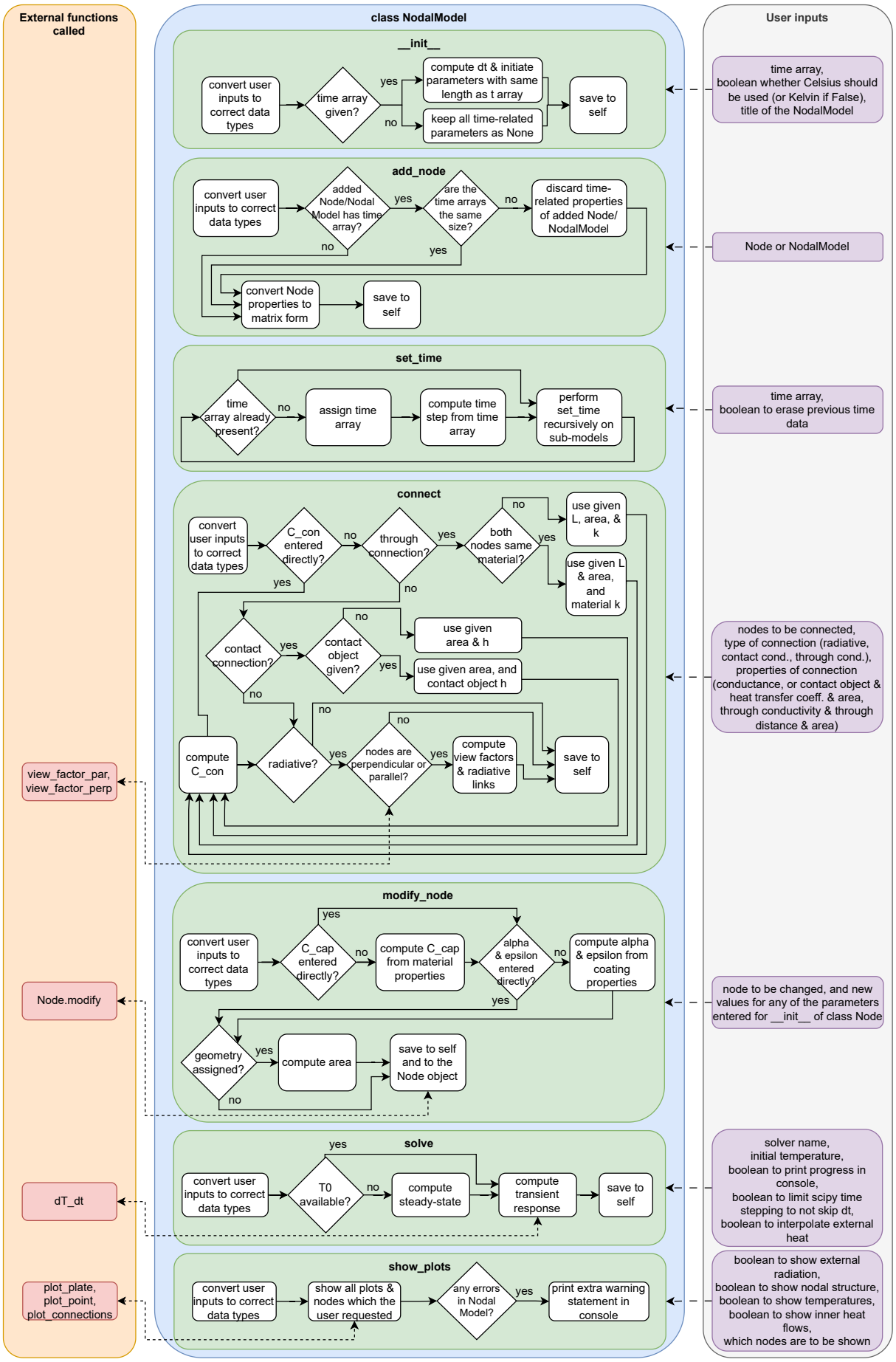


Figure 3.4: Overview of the NodalModel Python class, showing its functional flow, and the external functions or user parameters used. Refer to Section C.6 for details on all the functions and parameters.

3.3. Computation of External Heat Sources

The first step of a thermal analysis model is to compute the external heat sources on the spacecraft. For now, this will be done for a flat oriented plate, but the plate's surface area and properties are not considered yet. An overview of the calculation scheme is presented in the N2-chart in Figure 3.5. This methodology builds upon the previous work by C. Ruiz [7], but a number of adjustments and improvements have been incorporated, which will be explicitly noted where relevant.

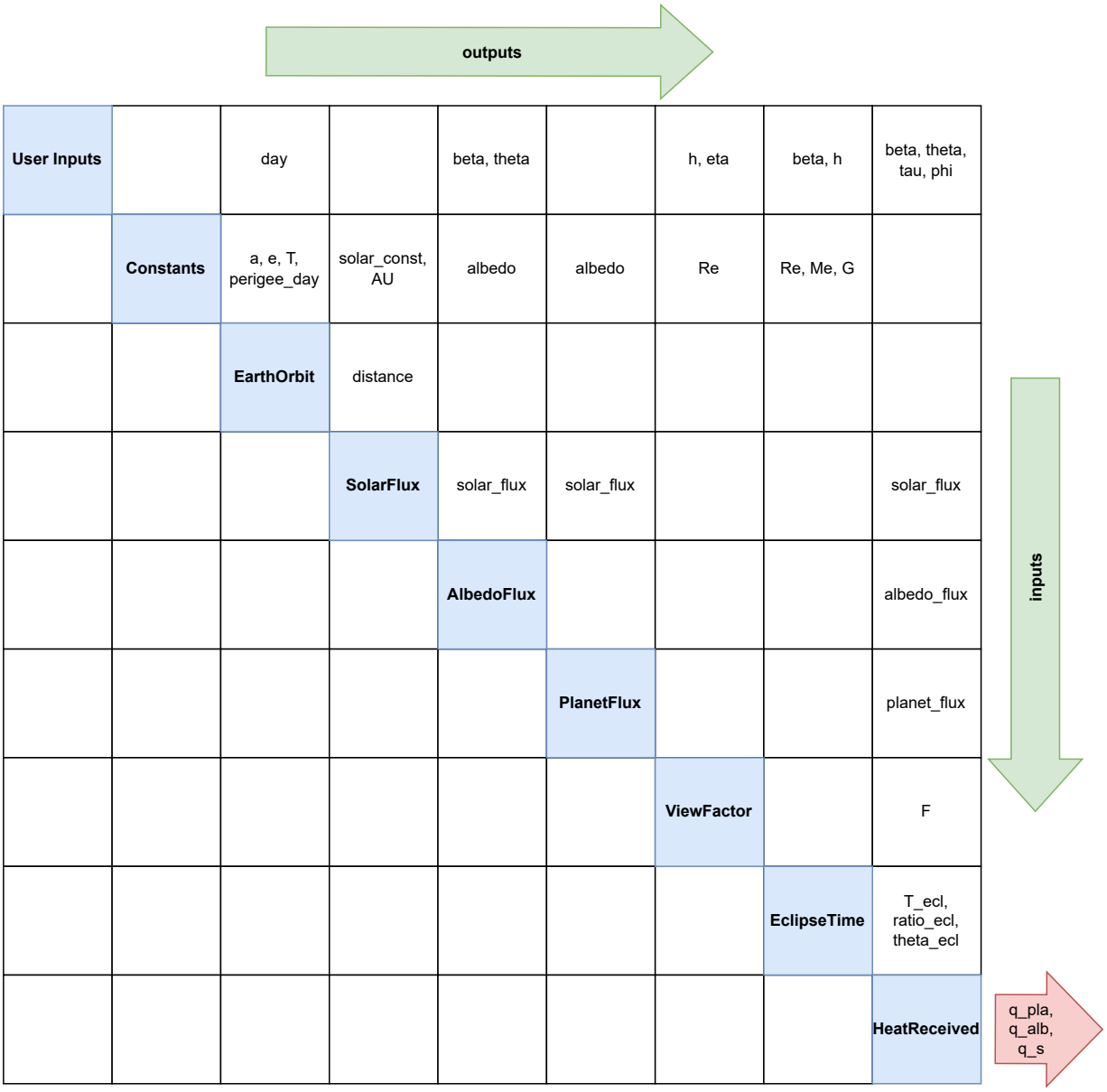


Figure 3.5: Python functions N2 chart of the external heat flux calculation `EnvironmentRadiation.py`. The functions are used starting from the top left to the bottom right.

3.3.1. Reference Frames and Assumptions

The reference frames used for this analysis are depicted in Figure 3.6, Figure 3.7, and Figure 3.8. Furthermore, the following assumptions are applied (the validity of these assumptions is thoroughly assessed in Section 5.3):

- The satellite is in Low-Earth Orbit (LEO). This results in the assumption that the Sun-Earth vector is

equal to the Sun-satellite vector.

- The Sun rays are assumed to be parallel at Earth, also resulting in the eclipse volume being a cylinder.
- The entire Earth surface has a uniform, constant outgoing infrared flux.
- The entire Earth surface has a uniform, constant albedo.
- There is no aerothermal heating and charged-particle heating present (see Section 2.3.1).

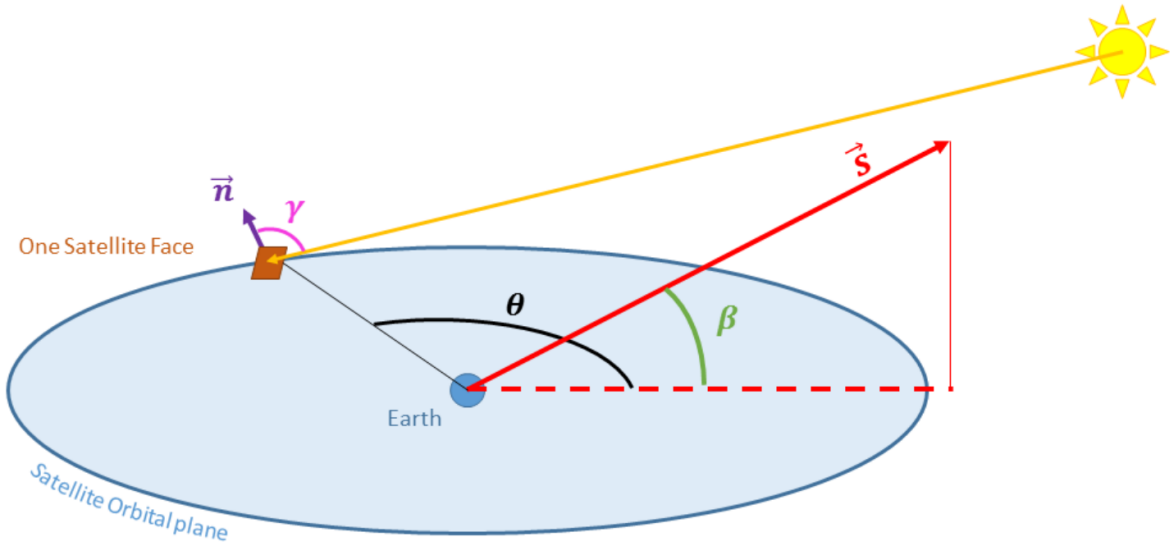


Figure 3.6: Reference system for a flat plate in orbit [7]. θ is the true anomaly in [rad], β is the solar angle (also referred to as beta angle) in [rad], $\vec{s}$ is the Sun-Earth vector, $\vec{n}$ is the orbiting surface's outward normal vector, and γ is the angle between the surface normal and Sun vector in [rad].

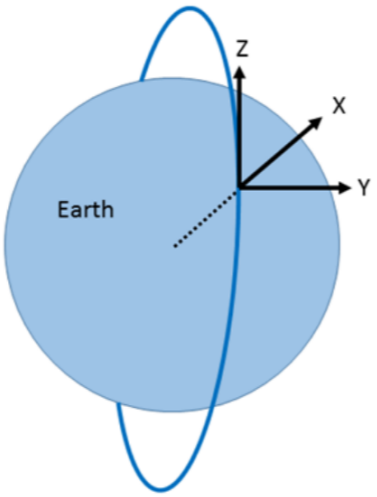


Figure 3.7: Reference system for the satellite position, where x points zenith and z points in the flight direction [7].

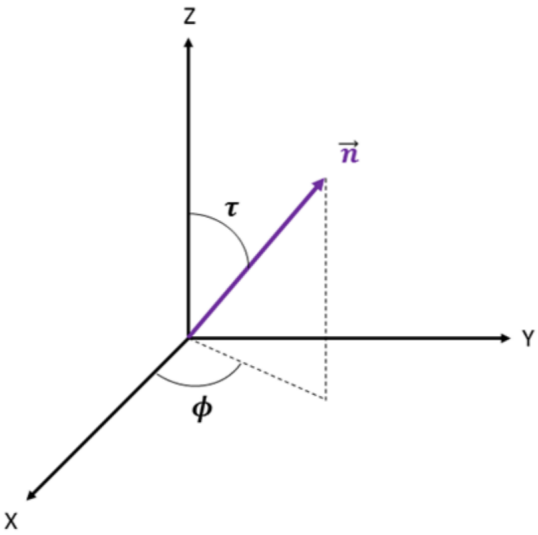


Figure 3.8: Reference system for the orientation of the plate [7].

3.3.2. Earth-Sun Distance

First, the Earth's distance with respect to the Sun throughout the year must be calculated. A day in the year (1-365) is given by the user, and the distance in metres is returned by the function. The orbit is assumed

to be Keplerian, which has the following equation for distance [42, p.165]:

$$r = \frac{a \cdot \left(1 - e^2\right)}{1 + e \cdot cos(\theta)} = \frac{a \cdot \left(1 - e^2\right)}{1 + e \cdot cos(\frac{day - day_{perigee}}{T} \cdot 2\pi)} \tag{3.1}$$

where the variables are (for the Earth-Sun obit):

- r is the distance in [m] between the central and orbiting body.
- a is the semi-major axis in [m]. For the Earth-Sun orbit, a = 1.00 AU = 1.495978707·10^{11} m [2].
- e is the orbit eccentricity. For the Earth-Sun orbit, $e = 0.01671$ [43].
- θ is the true anomaly [rad]. This notation will not be used further for the Earth-Sun orbit, but it will be used to indicate the true anomaly of the spacecraft's orbit.
- day is the day of interest starting at 1 on January 1st.
- $day_{perigee}$ is the day of perigee, which is on average January 3rd for the Earth-Sun orbit [44].
- T is the orbital period, which is approximately 365.25 days for Earth [43].

In reality, the orbit is not perfectly Keplerian, due to external perturbations such as the Moon and other planets. The effect of this is seen in the stochastically varying day of perigee by about 1-2 days [44], and in the different possible reference systems for the orbital period (sidereal/tropical) [43]. The effect of this will be studied in the sensitivity analysis in Chapter 5.

3.3.3. Solar Flux at Earth

Based on the Earth-Sun distance, the solar flux at Earth can be estimated. The flux at 1 AU (*solar constant* S_0) is currently (April 2024) accepted as 1361 W/m^2 [43, 45], but since it is not an absolute constant of physics, it may be subject to change based on changes throughout time, or after acquiring newer, better data.

Assuming the Sun radiates as a point source, the emitted power is spread uniformly as a sphere through space; hence, the flux at any distance can be estimated from the known solar constant at 1 AU:

$$\dot{q}_{solar} = S_0 \left(\frac{1\,AU}{r}\right)^2 \tag{3.2}$$

where $\dot{q}_{solar}$ is the solar heat flux in [W/m^2], S_0 is the solar constant in [W/m^2], and r is the distance from the Sun in [m]. Logically, r is retrieved from the Earth-Sun distance calculation, where the satellite-Sun distance is approximated as the Earth-Sun distance.

The total solar heat flow to the spacecraft also depends on the spacecraft's surface area, absorptivity α in the solar spectrum, and view factor F:

$$\dot{Q}_{solar} = \dot{q}_{solar} \cdot A_{sat} \cdot F_{Sun \to sat} \cdot \alpha_{sat} \tag{3.3}$$

The view factor is a simple case when the solar rays are assumed parallel, and hence:

$$F_{Sun \to sat} = max(cos(\gamma),\, 0) \tag{3.4}$$

where the max function is applied to account for the case where the plate faces away from the Sun (otherwise the cosine will turn negative). The satellite and Earth are so close that the Sun-Earth vector and Sun-satellite vector are almost parallel (e.g., for a 500 km LEO orbit, the angle between the vectors would only be $arctan((R_E + h)/AU) = 0.0026$ degrees), that the following approximation is valid:

$$\begin{aligned} cos(\gamma) \approx \vec{s} \cdot \vec{n} &= \begin{bmatrix} sin(\tau)cos(\phi) \\ sin(\tau)sin(\phi) \\ cos(\tau) \end{bmatrix} \cdot \begin{bmatrix} cos(\beta)cos(\theta) \\ -sin(\beta) \\ -cos(\beta)sin(\theta) \end{bmatrix} \\ &= cos(\beta)cos(\theta)sin(\tau)cos(\phi) - sin(\beta)sin(\tau)sin(\phi) - cos(\beta)sin(\theta)cos(\tau) \end{aligned} \tag{3.5}$$

It is important to note the minus sign in the second entry of $\vec{n}$, which was not present in the definition by Ruiz [7]. It appears because a positive β angle (Sun 'above' the orbit) is in the negative y-direction (y points down in Figure 3.6 when Figure 3.7 is used).

3.3.4. Planet Infrared Flux

The planetary heat flux is the infrared radiation emitted from Earth due to its temperature. The temperature can either be assumed constant throughout the year, or it can be assumed that the Earth is always in thermal equilibrium ($\dot{Q}_{in} = \dot{Q}_{out}$). In the latter case, the Earth's temperature would change throughout the year due to the changing Earth-Sun distance. This method was implemented by C. Ruiz [7], and the corresponding IR flux of Earth would be as demonstrated in Figure 3.9. The radiation is higher in northern winter (day 1 is January 3rd) and lower in summer, corresponding to the changing Earth-Sun distance.

It is debatable how accurate this assumption is, because the locally emitted temperature depends a lot on the local surface temperature (hence on the solar flux) and cloud coverage [25, p.24] [46]. Furthermore, flight data from 2007-2011, see Figure 3.10 [46], indicates a slight opposite trend compared to Figure 3.9, showing a higher average IR flux in northern summer compared to winter. The average flux throughout the year is 239 W/m^2, similar to the prediction by Rickman [47].

Another variation is the diurnal cycle, as it was just explained that the emitted IR also depends a lot on the Earth's surface temperature. [48]. The variance between the day and night heat flux is shown in Figure 3.11. The ocean has an almost constant outgoing IR flux, while continental areas deviate more [48]. The variation in W/m^2 is in the same order of magnitude as the seasonal changes from Figure 3.10.

Since the seasonal changes from Ruiz [7] conflicts with the measurements from CERES [46], a constant 239 W/m^2 is assumed for now. The diurnal variations are neglected for now, since the majority of Earth's surface is ocean, which shows the lowest diurnal temperature variations. In Chapter 5, the effects of these assumptions will be evaluated.

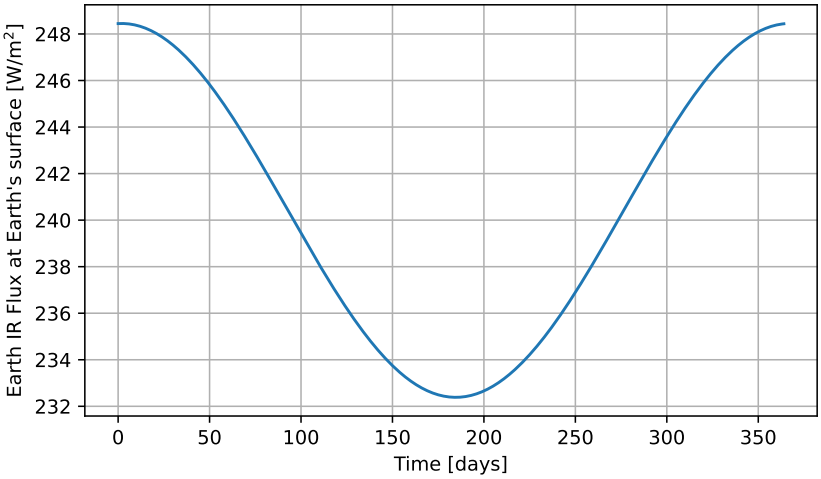


Figure 3.9: Earth infrared radiation through the year based on a heat balance with the incoming solar and outgoing infrared radiation.

Average Latitude	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	*Ave. over 2007*	*Ave. over 2008*	*Ave. over 2009*	*Ave. over 2010*	*Ave. over 2011*	Average over 5-year
85°	166 W	175 W	173 W	190 W	210 W	226 W	229 W	220 W	205 W	188 W	177 W	171 W	*194 W*	*193 W*	*194 W*	*194 W*	*194 W*	**194 W**
75°	173 W	177 W	180 W	196 W	213 W	226 W	232 W	224 W	211 W	194 W	183 W	175 W	*199 W*	*198 W*	*198 W*	*200 W*	*198 W*	**199 W**
65°	179 W	181 W	190 W	204 W	219 W	231 W	237 W	230 W	216 W	201 W	188 W	180 W	*205 W*	*204 W*	*205 W*	*206 W*	*204 W*	**205 W**
55°	192 W	193 W	202 W	214 W	223 W	234 W	239 W	237 W	227 W	214 W	201 W	193 W	*214 W*	*214 W*	*213 W*	*215 W*	*214 W*	**214 W**
45°	204 W	206 W	212 W	222 W	231 W	242 W	252 W	254 W	246 W	230 W	217 W	206 W	*228 W*	*227 W*	*226 W*	*226 W*	*227 W*	**227 W**
35°	226 W	225 W	230 W	237 W	245 W	257 W	267 W	268 W	262 W	251 W	239 W	229 W	*246 W*	*245 W*	*243 W*	*244 W*	*245 W*	**245 W**
25°	262 W	264 W	266 W	266 W	268 W	268 W	267 W	266 W	269 W	271 W	266 W	262 W	*266 W*	*266 W*	*266 W*	*267 W*	*265 W*	**266 W**
15°	275 W	280 W	279 W	272 W	262 W	252 W	244 W	239 W	244 W	254 W	264 W	271 W	*262 W*	*261 W*	*261 W*	*262 W*	*260 W*	**261 W**
5°	252 W	255 W	250 W	241 W	235 W	235 W	235 W	238 W	239 W	241 W	243 W	249 W	*244 W*	*241 W*	*244 W*	*243 W*	*242 W*	**243 W**
-5°	241 W	239 W	240 W	244 W	257 W	263 W	264 W	266 W	263 W	257 W	251 W	246 W	*253 W*	*252 W*	*252 W*	*253 W*	*253 W*	**253 W**
-15°	246 W	246 W	254 W	265 W	274 W	279 W	281 W	283 W	280 W	269 W	260 W	251 W	*266 W*	*266 W*	*266 W*	*266 W*	*266 W*	**266 W**
-25°	264 W	265 W	266 W	264 W	262 W	261 W	266 W	269 W	268 W	262 W	261 W	261 W	*264 W*	*264 W*	*264 W*	*264 W*	*264 W*	**264 W**
-35°	259 W	257 W	254 W	247 W	240 W	235 W	238 W	239 W	240 W	242 W	245 W	250 W	*246 W*	*245 W*	*245 W*	*246 W*	*246 W*	**246 W**
-45°	235 W	237 W	232 W	227 W	221 W	217 W	215 W	217 W	219 W	224 W	228 W	232 W	*225 W*	*225 W*	*224 W*	*225 W*	*226 W*	**225 W**
-55°	218 W	218 W	214 W	208 W	205 W	200 W	199 W	199 W	201 W	207 W	212 W	217 W	*208 W*	*208 W*	*208 W*	*208 W*	*208 W*	**208 W**
-65°	211 W	207 W	201 W	192 W	186 W	180 W	178 W	177 W	181 W	190 W	200 W	208 W	*194 W*	*192 W*	*193 W*	*192 W*	*193 W*	**193 W**
-75°	199 W	187 W	173 W	160 W	152 W	146 W	143 W	142 W	148 W	162 W	181 W	197 W	*167 W*	*165 W*	*167 W*	*165 W*	*167 W*	**166 W**
-85°	196 W	177 W	157 W	143 W	138 W	131 W	127 W	124 W	130 W	149 W	173 W	194 W	*154 W*	*152 W*	*154 W*	*152 W*	*154 W*	**153 W**
Average	**236 W**	**236 W**	**237 W**	**238 W**	**239 W**	**241 W**	**243 W**	**243 W**	**241 W**	**239 W**	**236 W**	**235 W**	*239 W*	*238 W*	*238 W*	*239 W*	*239 W*	**239 W**

Figure 3.10: Earth infrared heat flux measurements (NOTE: values in W/m^2) between 2007-2011 by the CERES spacecraft [46].

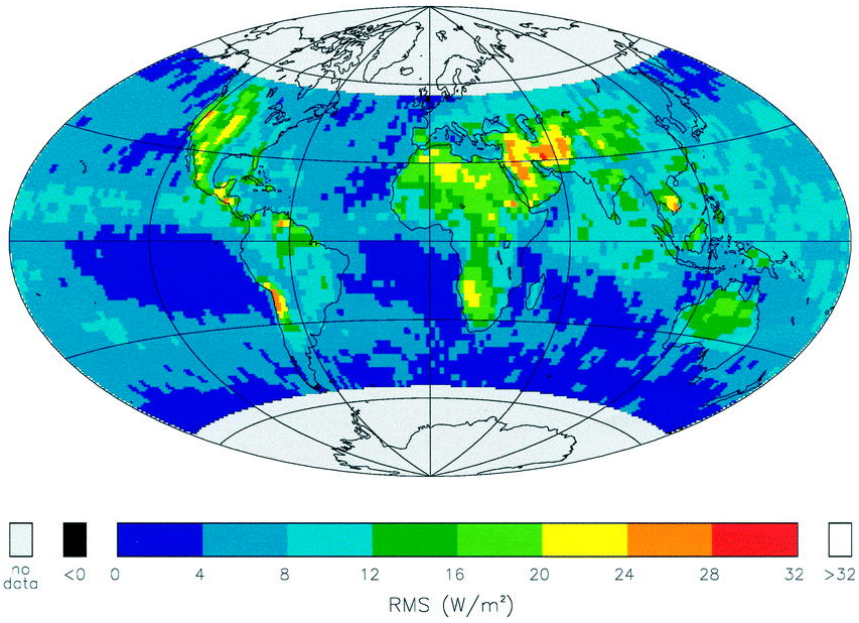


Figure 3.11: Variation in Earth IR flux between day and night, in W/m^2, during northern summer [48].

Having assumed a constant Earth IR radiance of 239 W/m^2 at the Earth's surface, this must be translated into the flux at the spacecraft. Unlike the assumption of parallel Sun rays, satellites in LEO are too close to the Earth to assume parallel rays. Instead, the view factor between the spherical Earth and the spacecraft is used. More on this is explained in Section 3.3.6. The heat flux from the Earth to the satellite is then computed with (still excluding the spacecraft's surface properties):

$$\dot{q}_{Earthshine} = \frac{\dot{q}_{E,tot} \cdot A_E \cdot F_{E \to sat}}{A_{sat}} = \dot{q}_{E,tot} \cdot F_{sat \to E} \tag{3.6}$$

where $\dot{q}_{Earthshine}$ is the heat flux on the satellite in [W/m^2], $\dot{q}_{E,tot}$ is the 239 W/m^2 value for the Earth IR at the Earth's surface, A are the Earth and satellite surface areas in [m^2] $F_{E \to sat}$ is the view factor from Earth to the satellite, and $F_{sat \to E}$ is the view factor from the satellite to Earth. Furthermore, the area reciprocity relation Equation 2.6 for view factors is used. Finally, this is converted to the actual heat in [W] with:

$$\dot{Q}_{Earthshine} = \dot{q}_{Earthshine} \cdot \varepsilon_{sat} \cdot A_{sat} = \dot{q}_{E,tot} \cdot F_{sat \to E} \cdot \varepsilon_{sat} \cdot A_{sat} \tag{3.7}$$

where ε_{sat} is the infrared emissivity/absorptivity of the satellite's outer surface.

3.3.5. Albedo Flux via Earth

The albedo coefficient of Earth depends on the Earth's surface characteristics, such as land, sea, or ice caps. On average, the albedo of Earth is 0.294 [43]. This value can be used for a simplified analysis, where the Earth is assumed to have a uniform surface with an albedo of 0.294, and as a consequence, the orientation of the Earth is of no influence. Such a computation would use the following equation [47]:

$$\dot{q}_{albedo,tot} = max\left(a \cdot \dot{q}_{solar} \cdot cos(\xi),\, 0\right) = max\left(a \cdot \dot{q}_{solar} \cdot cos(\beta)cos(\theta),\, 0\right) \tag{3.8}$$

where a is the albedo coefficient, $\dot{q}_{albedo,tot}$ is the albedo flux in [W/m^2] at the Earth's surface in the direction of the given orbital position, max indicates that the maximum value between both entries is taken, β is the angle [rad] between the Earth-Sun line and the spacecraft's orbital plane, θ is the true anomaly [rad] of the satellite, and ξ is the solar zenith angle [rad]. When the $cos(\xi) < 0$, the Earth's surface beneath the satellite does not receive any sunlight, and hence the albedo is approximately zero (assuming a low-Earth-orbit). The cosines are a special case of spherical trigonometry where the two angles β and θ are in perpendicular planes [49]:

$$cos(\xi) = cos(\beta)cos(\theta) + sin(\beta)sin(\theta)cos(90°) = cos(\beta)cos(\theta) \tag{3.9}$$

where ξ is the solar zenith angle (between the Earth-Sun and Earth-spacecraft lines) in radians [47]. To achieve the albedo received at the spacecraft, a similar computation as in Section 3.3.4 may be used, since both cases of radiation are considered diffuse:

$$\dot{Q}_{albedo} = \dot{q}_{albedo,tot} \cdot F_{sat \to E} \cdot \alpha_{sat} \cdot A_{sat} \tag{3.10}$$

where α is the absorption of the satellite in the solar spectrum.

If more detail is desired for the albedo variation over the Earth's surface, the Earth's orientation must be included in the model, and a database for albedo values at different longitudes and/or latitudes must be used [46].

3.3.6. View Factor of a Flat Plate and a Sphere

From the perspective of the satellite, the Earth IR and albedo radiation originate from the Earth's surface, and since a Low-Earth Orbit is so close to the Earth, the outgoing beams cannot be assumed to be parallel. Instead, the sphere-to-infinitesimal-plate view factor case from ECSS [32] is used, see Figure 3.12. The associated equations are given in Equation 3.11–3.18, where p stands for 'plate' and s stands for 'sphere'. Note that the fourth power in the B_3 term is found in the original source by Clark & Anderson [50], and the ECSS standard [32] incorrectly wrote a third power instead. This formula was tested with both the third and the fourth power, and only the fourth power (from [50]) yielded correct result.

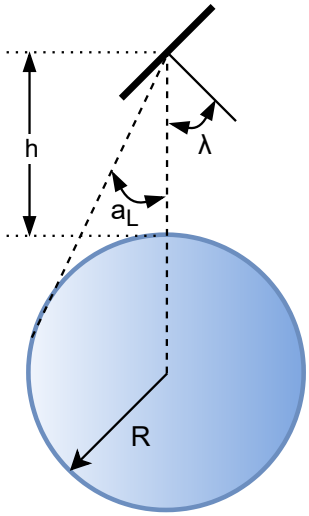


Figure 3.12: View factor case for an angled flat plate nearby a sphere (adapted from [32]).

$$F_{p\to s} = B_0 + B_1 cos(\lambda) + B_2 cos^2(\lambda) + B_3 cos^4(\lambda) + B_4 cos^6(\lambda) \tag{3.11}$$

$$B_0 = \frac{2}{7\pi}\left(\frac{577}{105} - 7cos(a_L) + \frac{4}{3}cos^3(a_L) - \frac{2}{5}cos^5(a_L) + \frac{4}{7}cos^7(a_L)\right) \tag{3.12}$$

$$B_1 = \frac{1}{2}sin^2(a_L) \tag{3.13}$$

$$B_2 = \frac{8}{7\pi}\left(cos(a_L) - 2cos^3(a_L) + 4cos^5(a_L) - 3cos^7(a_L)\right) \tag{3.14}$$

$$B_3 = \frac{4}{7\pi}\left(-cos(a_L) + \frac{40}{3}cos^3(a_L) - \frac{91}{3}cos^5(a_L) + 18cos^7(a_L)\right) \tag{3.15}$$

$$B_4 = \frac{8}{35\pi}\left(5cos(a_L) - 35cos^3(a_L) + 63cos^5(a_L) - 33cos^7(a_L)\right) \tag{3.16}$$

$$a_L = arcsin\left(\frac{1}{1 + h/R_E}\right) \tag{3.17}$$

Furthermore, the angle λ can be defined in terms of the angle η (Equation 3.18). η is the angle between the x-axis and $\vec{n}$ (Figure 3.8), and since τ and ϕ are in perpendicular planes, the same simplification of spherical geometry applies as in Equation 3.9.

$$\lambda = 180° - \eta \tag{3.18}$$

$$cos(\eta) = cos(\phi)sin(\tau) \tag{3.19}$$

Finally, Equation 3.11 can end up slightly negative in some instances, when the plate is pointing away from the sphere, but the formulas show some oscillations in those cases [50]. To prevent any future errors due to slightly negative view factors (which is by definition impossible), the `ViewFactor` function will return zero instead. A visualisation of Equation 3.11 is depicted in Figure 3.13.

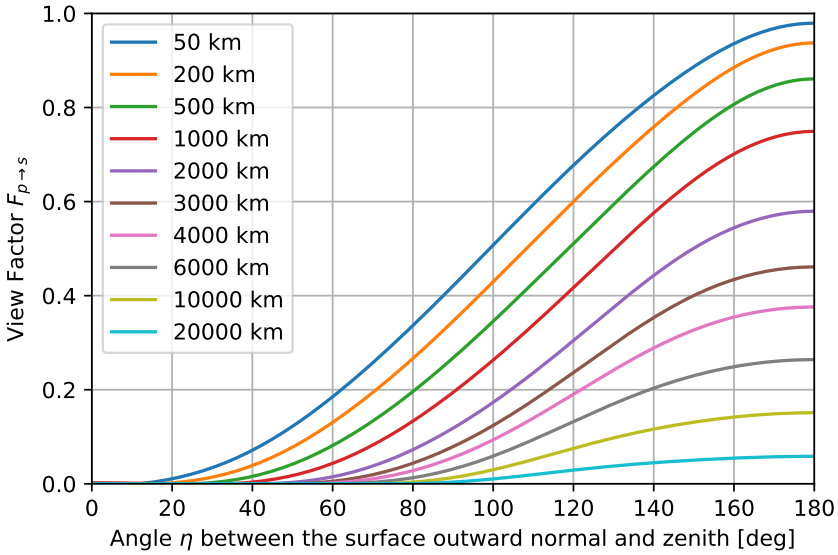


Figure 3.13: Visualisation of Equation 3.11, where each line is a different orbit altitude. The graph would be mirrored horizontally when λ would be used instead of η.

3.3.7. Eclipse Characteristics for a Circular Orbit Around Earth

For the estimation of the eclipse entry and exit points, the LEO assumption results in a cylindrical shadow. The difference between umbra and penumbra is thereby ignored.

For a given orbit altitude, there is a maximum (limit) beta angle for which there is still an eclipse; if β exceeds this, there is no eclipse for the given orbit. The associated figures are Figure 3.14 and Figure 3.15 (the xyz-coordinates are different from Figure 3.8). The limit beta angle can be calculated as follows:

$$\beta_{lim,ecl} = arccos\left(\frac{R_E}{R_E + h}\right) \tag{3.20}$$

where R_E is the Earth's mean radius (6,371 km [43]) and h is the orbit height. In case the absolute value of β is below this limit, there will be an eclipse.

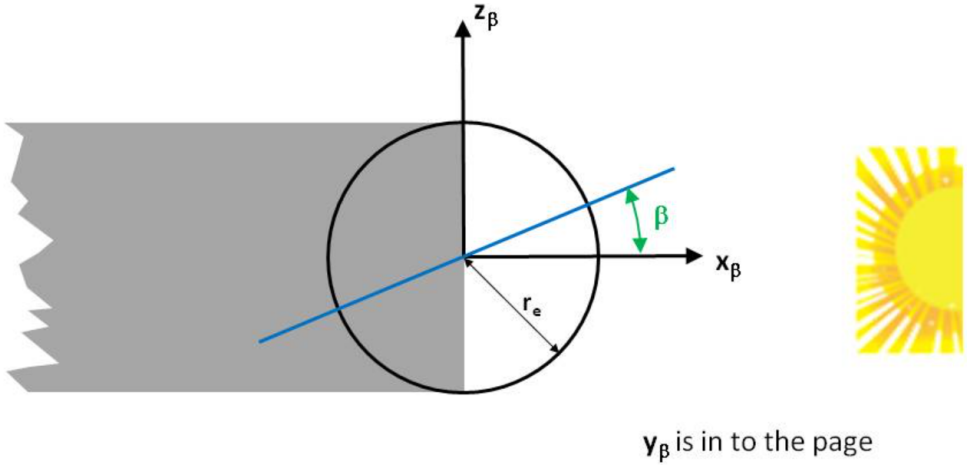


Figure 3.14: Side view of the beta angle and eclipse region [51].

Figure 3.15 is used to find an expression for the eclipse angle given an altitude and beta angle.

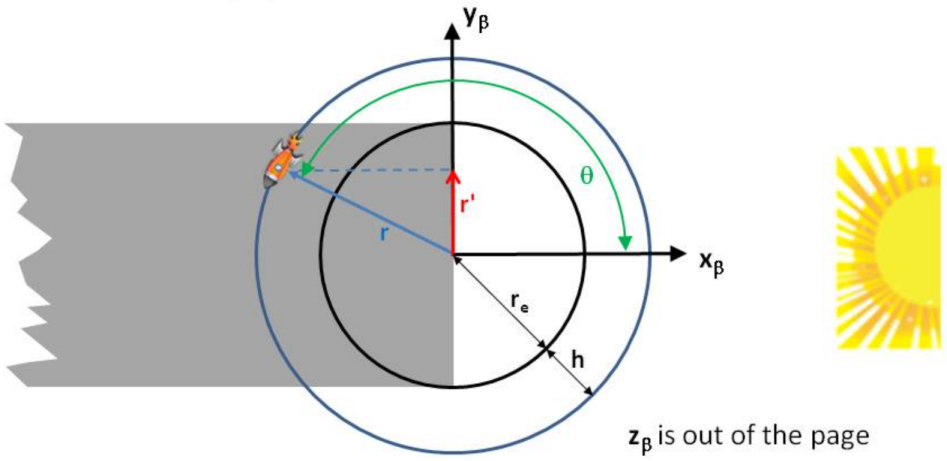


Figure 3.15: Top view of the beta angle and eclipse region [51].

The method from [51] uses the statement that for the following position vector, the satellite is in eclipse, provided that $\beta < \beta_{lim}$ and that $\pi/2 < \theta < 3\pi/2$:

$$|\vec{r}'| < R_E \tag{3.21}$$

The vector $\vec{r}'$ is defined as:

$$\vec{r}' = (r_E + h) \begin{bmatrix} 0 \\ sin(\theta) \\ cos(\theta)sin(\beta) \end{bmatrix} \tag{3.22}$$

The magnitude of $\vec{r}'$ is:

$$|\vec{r}'| = (R_E + h)\sqrt{sin^2(\theta) + cos^2(\theta)sin^2(\beta)} \tag{3.23}$$

Setting this equal to R_E to find the limit of Equation 3.21:

$$\begin{aligned} R_E &= (R_E + h)\sqrt{sin^2(\theta) + cos^2(\theta)sin^2(\beta)} \\ \left(\frac{R_E}{R_E + h}\right)^2 &= sin^2(\theta) + \left(1 - sin^2(\theta)\right) sin^2(\beta) \\ \left(\frac{R_E}{R_E + h}\right)^2 &= sin^2(\theta)\left(1 - sin^2(\beta)\right) + sin^2(\beta) \\ \left(\frac{R_E}{R_E + h}\right)^2 &= sin^2(\theta)cos^2(\beta) + sin^2(\beta) \\ sin(\theta_{ecl}) &= \sqrt{\frac{1}{cos^2(\beta)}\left[\left(\frac{R_E}{R_E + h}\right)^2 - sin^2(\beta)\right]} \end{aligned} \tag{3.24}$$

where θ_{ecl} [rad] is the true anomaly of the onset of the eclipse. Note that θ_{ecl} is by definition larger than $\pi/2$, but if the arcsine is to be taken of Equation 3.24, only values below $\pi/2$ will be returned. However, due to the sine's symmetry around $\pi/2$, $sin(\pi - \theta_{ecl}) = sin(\theta_{ecl})$, the value returned by the arcsine is going to be $\pi - \theta_{ecl}$, which can be defined as the eclipse half-angle $\psi_{ecl}/2$. The total eclipse angle is then ψ_{ecl}. Hence:

$$\theta_{ecl} = \pi - \frac{\psi_{ecl}}{2} = \pi - arcsin\left(\sqrt{\frac{1}{cos^2(\beta)}\left[\left(\frac{R_E}{R_E+h}\right)^2 - sin^2(\beta)\right]}\right) \tag{3.25}$$

From this, the eclipse fraction (Equation 3.26) and duration (Equation 3.27) can be found:

$$ratio_{ecl} = \frac{\psi_{ecl}}{2\pi} \tag{3.26}$$

$$T_{ecl} = ratio_{ecl} \cdot T = ratio_{ecl} \cdot 2\pi\sqrt{\frac{(R_E+h)^3}{\mu_E}} \tag{3.27}$$

where a circular orbit is assumed, T is the orbital period of the satellite, and μ_E is the gravitational parameter of Earth (=3.986·10^{14} [m^3/s^2] [43]).

3.3.8. Beta Angle from Orbital Parameters

In reality, the beta angle (solar declination) is not a design parameter for the satellite's mission, but it follows from another set of orbital parameters, which can then be converted to β.

Geocentric Orbits

For geocentric orbits, i.e. the orbit is fixed with respect to the vernal/march equinox, the orbital parameters used are often the Right Ascension of the Ascending Node (RAAN) Ω, the inclination i, and the ecliptic true solar longitude Γ. A celestial sphere with Ω and i is shown in Figure 3.16. Similarly to this, Γ can be defined for the Sun, but the Sun's inclination with respect to the Earth's equator (obliquity of the ecliptic) is approximately constant at $\varepsilon = 23.44°$ [43]. Γ is then the solar angle on the ecliptic (between 0 and 360°), and is zero when the Sun is at the vernal equinox, which is around 20 March (on average, day 79.25 starting at January 1st [52]).

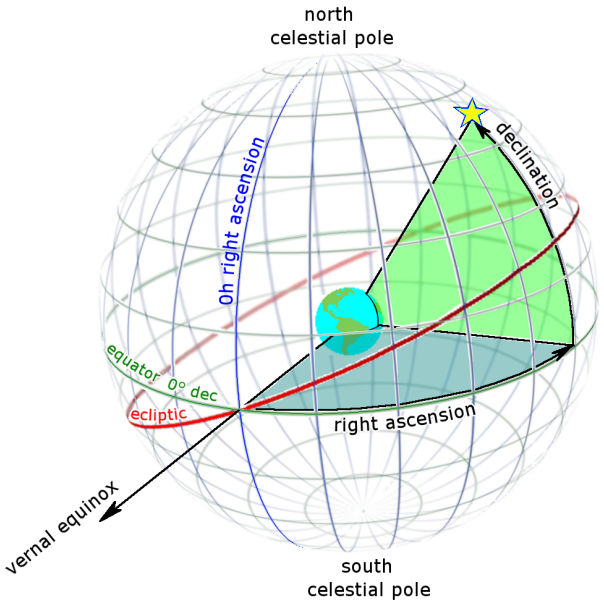


Figure 3.16: Illustration of the celestial sphere around Earth [53].

Hence, Γ can be computed with Equation 3.28, and β follows from Equation 3.29 [51]. The last equation originates from the dot product of the satellite's orbit vector and the solar vector.

$$\Gamma = 2\pi \frac{day - day_{march-equinox}}{T} \tag{3.28}$$

$$\beta = arcsin(cos(\Gamma)sin(\Omega)sin(i) - sin(\Gamma)cos(\varepsilon)cos(\Omega)sin(i) + sin(\Gamma)sin(\varepsilon)cos(i) \tag{3.29}$$

where day is the number of days after January 1st, $day_{march-equinox}$ is the day of the vernal equinox (=79.25), T is the orbital period of Earth around the Sun in days (365.25), β is the solar declination [rad], Γ is the ecliptic true solar longitude [rad], Ω is the RAAN of the satellite's orbit [rad], i is the inclination of the satellite's orbit [rad], and ε is the obliquity of the ecliptic [rad].

It is important to note that in this thesis, β is always assumed to be positive in the direction of the north pole. This is also illustrated in Figure 3.6, but it appears that it is sometimes confused in the paper by Ruiz [7] and the lecture by Rickman [47]. The sign convention of positive β northward is also followed by the ESATAN software, see Figure 3.17.

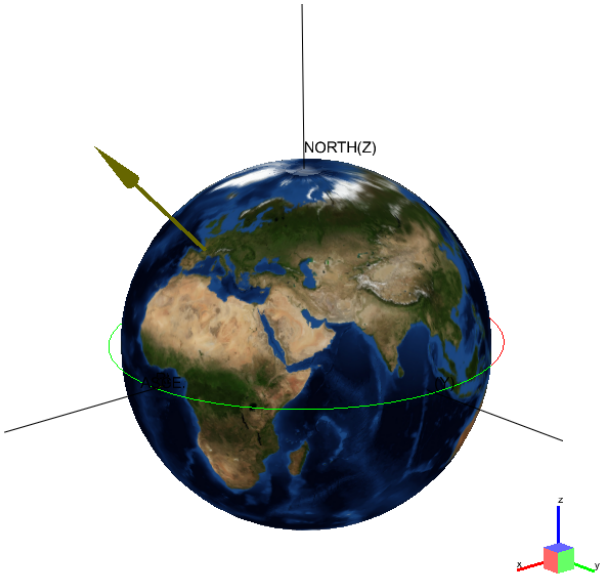


Figure 3.17: Example of the beta angle sign convention in ESATAN-TMS, where β is set to +45°. The solar vector indeed points northward.

3.3.9. Angular Rates on a Flat Plate

An additional feature is included, where the plate can be given angular rates about the x-, y-, or z-axis. This is extremely useful for the thermal analysis of tumbling spacecraft. Rotational rates were also present in the MATLAB tool by Ruiz [7], but the implementation was quite different (surface areas were multiplied by a correction factor, instead of directly performing matrix rotations on the attitude). In this Python tool, the `Rotation` object from SciPy is used to perform the rotations [54]. It transforms various types of coordinate systems into quaternions, computes the rotations, and transforms the angles back to the original coordinate system.

The spherical τ and ϕ angles are not supported by SciPy, so those have to be manually converted to another coordinate system. The simplest coordinate system that is compatible with SciPy is a Cartesian coordinate system, consisting of a unit vector in the 3-dimensional xyz space. The spherical angles are converted to Cartesian coordinates using the standard `from_euler` function in SciPy, where the spherical angles are represented as an Euler rotation: τ is first applied on the y-axis, and then ϕ is applied on the z-axis. The rotation is calculated automatically by SciPy.

However, converting the Cartesian coordinates back to τ and ϕ after SciPy has computed the desired rotation, would be less trivial using SciPy, since there are multiple possible combinations of Euler angles possible. Therefore, this step is performed manually. The simplest method to find τ and ϕ is to compute a standard case, and then use some logic to set the singularities. If this is not done in the correct order, attempting to take arccosines and arctangents might not be possible.

Using Figure 3.8, the polar angle τ is always defined as:

$$\tau = arccos\,(z) \tag{3.30}$$

where z is the z-component of the unit vector. The azimuth angle ϕ cannot always be accurately computed using simply an arctangent (although `NumPy.arctan2` could also fix the issue); therefore, the following logic was implemented:

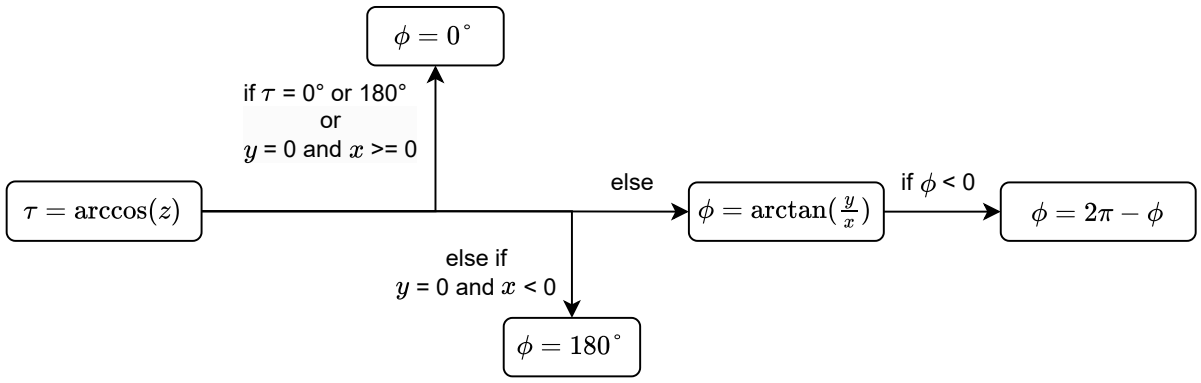


Figure 3.18: Logic diagram for determining the value of the azimuth angle ϕ from Cartesian x, y, and z components.

3.3.10. Total Heat Flux on Flat Plate in Orbit

The final Python function `heat_received` combines the aforementioned calculations and returns the final solar, albedo, and planetary flux on the plate in orbit. These outputs are in [W/m^2] and are not yet multiplied by the spacecraft's surface area and absorptivity/emissivity; this is done later in the code.

For an entire spacecraft, multiple plates (orientations) can be entered in the OrbitalModel object, and all their heat fluxes will simply all be computed using a for-loop over the plates.

3.4. Computation of Heat Flows

Originally, the tool by Ruiz [7] only considers conductive heat flow for the internals of the satellite. For the new software, internal radiation shall also be considered, of which the effects are analysed in more detail in Chapter 5. Furthermore, the matrices and computational methods used in this thesis are different from those used by Ruiz.

Differently to the external heat computation, the nodal model is mainly defined as an object (Python class) called `NodalModel`. This object is the entire model with all its nodes, powers, conductances, etc. Figure 3.4 gives an overview of the class.

Note that the object does not need to be initialized all at once with all vectors and matrices defined. `NodalModel` can be initialized with just a title name, essentially starting as a zero-node model. The nodes can then be added later with the `add_node` function.

3.4.1. Transient Calculation Assumptions

Besides the assumptions from Section 3.3.1 on the environmental radiation analysis, some more assumptions apply for the transient calculation. The following assumptions are applied (justifications are argued in-text, and a sensitivity analysis can be found in Chapter 5):

- The satellite has a constant mass and constant material properties throughout time. Hereby, e.g. degradation of paint, degradation of thermal conductance, loss of propellant, etc. are neglected. A manual workaround can be to split up the analysis into multiple subsequent runs.
- Rays of thermal radiation are only calculated once when they leave their original surfaces; multiple reflections are not included. For emissivities of unity, the two cases are identical. For emissivities lower than one, errors start to build up as the reflection of (1-ε) is ignored. This is further explored in Chapter 5.
- SmallSat components are assumed to either have no geometry, or to be a flat plate at parallel or orthogonal angles with respect to the rest of the spacecraft.
- There is no convection in/around the entire spacecraft.

3.4.2. Heat Balance Equation

The backbone of the transient solver is the heat balance equation. For the entire spacecraft as a whole system, this is essentially Equation 2.1, which is repeated here:

$$\left(m \cdot c_{cap} \cdot \frac{dT}{dt}\right)_{spacecraft} = \dot{Q}_{solar} + \dot{Q}_{albedo} + \dot{Q}_{Earthshine} + \dot{Q}_{internal} - \dot{Q}_{out} \tag{3.31}$$

Here, all heat $\dot{Q}$ has the unit [W], the mass m is in [kg], the specific heat capacity c is in [J/(kg·K)], and the temperature change dT/dt is in [K/s] or [°C/s]. The mass and specific heat capacity are often combined into the absolute heat capacity C in [J/K] (capital C instead of lowercase).

The only unknown in this equation is the temperature change dT/dt, so the equation can be rewritten to be explicit for dT/dt. Moreover, the terms on the right-hand side can be expanded into the individual components which are used in the computational algorithm. Furthermore, when considering a single node i, there are also heat fluxes to and from other nodes due to a temperature difference. In reality, telecommunications also emit power away from the spacecraft; it would require a sophisticated analysis if this were to be implemented into the thermal model, so it is ignored here.

$$\left(\frac{dT}{dt}\right)_i = \frac{1}{C_{cap,i}} \left\{ \begin{array}{c} A_i \left[\alpha_i \left(\dot{q}_{i,solar} + \dot{q}_{i_{albedo}}\right) + \varepsilon_i \dot{q}_{i_{Earthshine}}\right] + \dot{Q}_{i_{internal}} + \\ + \sum \left[C_{con,ij} \left(T_j - T_i\right)\right] + \sum \left[R_{ij}\sigma \left(T_j^4 - T_i^4\right)\right] - \texttt{outer} \cdot A_i \varepsilon_i \sigma \left(T_i^4 - T_{space}^4\right) \end{array} \right\} \tag{3.32}$$

where the lowercase $\dot{q}$ represents the heat flux in [W/m^2], which is the output of the orbital radiative analysis, uppercase $\dot{Q}$ is the absolute power input in [W], $C_{cap,i}$ is the heat capacity of node i in [J/K], $C_{con,ij}$ is the conductance between nodes i and j in [W/K], R_{ij} is the radiative coupling between nodes i and j in [m^2], and `outer` is a Boolean (True/False) indicating whether the node radiates to space or not. In essence, it is a ”zero” or ”one” view factor of the plate with respect to space. For more complicated geometries, such as deployable solar panels, this method is not as reliable. T_{space} is set to 3 K [55].

The calculation of the conductive couplings $C_{con,ij}$ and radiative couplings R_{ij} is performed using Equation 2.15, Equation 2.16, and Equation 2.7. To find the Gebhart factor G_{ij} in Equation 2.7, it is approximated that $G_{ij} \approx F_{ij}$ by neglecting multiple reflections. Then, the view factors are determined with Equation 2.10 and Equation 2.11. Although these equations only apply for perfectly parallel and perpendicular plates, they are exact in those cases.

3.4.3. Numerical Implementation

In reality, the spacecraft has three-dimensional elements and continuous features. The geometry can be arbitrarily shaped, and the material/electrical/thermal properties may change throughout time and can be unpredictable. This complex system can be simplified into a finite number of nodes, with a finite number of fixed properties. This process of building a numerical model from a physical model is illustrated in Figure 3.19.

The heat equation, Equation 3.32, is based on a finite number of nodes interacting with each other. As the number of nodes increases, the method of numerical implementation will become more important, since the computational time might rise drastically if an inadequate framework is used. Vectors and matrices are often used, since matrices can easily be expanded when nodes are added, while keeping the computation scheme identical. Moreover, Python libraries such as NumPy are optimised for the use of vectors and matrices, improving the model's scalability.

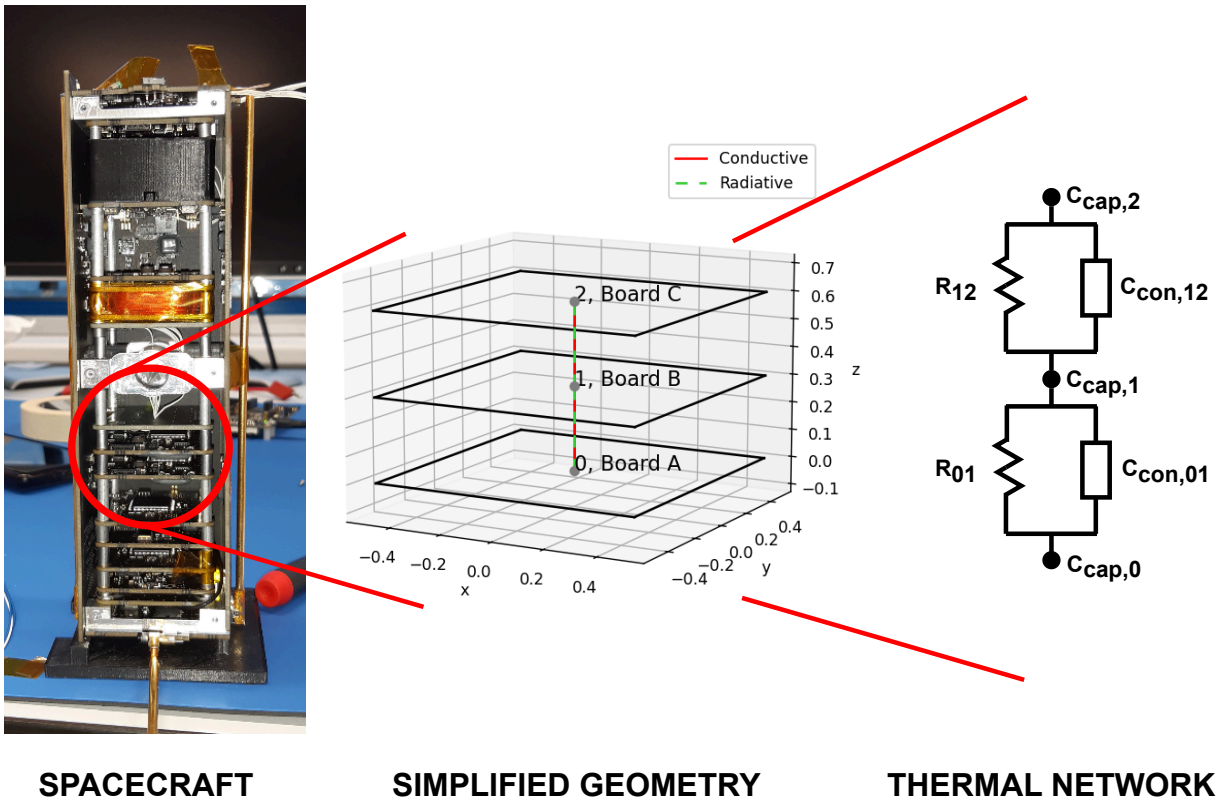


Figure 3.19: Visualisation of the discretisation of a real spacecraft (Delfi-PQ), to a simplified model, to a thermal network. Adapted from [10].

Parameter Definitions

For a single-node system, Equation 3.32 can be used directly, although the terms with $C_{con,ij}$ and R_{ij} would disappear. For a multi-node system, this becomes a system of equations, which is more easily solved using matrices and arrays. The parameters in Equation 3.32 are categorised as follows:

- **Constants**: $C_{cap,i}$, A_i, α_i, ε_i, $C_{con,ij}$, R_{ij}, σ, T_{space}, `outer`. The variables $C_{con,ij}$ and R_{ij} come from nxn size matrices (n is number of nodes), which are also symmetric ($C_{con,ij} = C_{con,ji}$ and $R_{ij} = R_{ji}$):

$$C_{con} = \begin{bmatrix} C_{con,1,1} & C_{con,1,2} & \cdots & C_{con,1,n-1} & C_{con,1,n} \\ C_{con,2,1} & C_{con,2,2} & \cdots & C_{con,2,n-1} & C_{con,2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ C_{con,n-1,1} & C_{con,n-1,2} & \cdots & C_{con,n-1,n-1} & C_{con,n-1,n} \\ C_{con,n,1} & C_{con,n,2} & \cdots & C_{con,n,n-1} & C_{con,n,n} \end{bmatrix} \tag{3.33}$$

$$= \begin{bmatrix} 0 & C_{con,1,2} & \cdots & C_{con,1,n-1} & C_{con,1,n} \\ C_{con,2,1} & 0 & \cdots & C_{con,2,n-1} & C_{con,2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ C_{con,n-1,1} & C_{con,n-1,2} & \cdots & 0 & C_{con,n-1,n} \\ C_{con,n,1} & C_{con,n,2} & \cdots & C_{con,n,n-1} & 0 \end{bmatrix} \tag{3.34}$$

$$R = \begin{bmatrix} R_{1,1} & R_{1,2} & \cdots & R_{1,n-1} & R_{1,n} \\ R_{2,1} & R_{2,2} & \cdots & R_{2,n-1} & R_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ R_{n-1,1} & R_{n-1,2} & \cdots & R_{n-1,n-1} & R_{n-1,n} \\ R_{n,1} & R_{n,2} & \cdots & R_{n,n-1} & R_{n,n} \end{bmatrix} = \begin{bmatrix} 0 & R_{1,2} & \cdots & R_{1,n-1} & R_{1,n} \\ R_{2,1} & 0 & \cdots & R_{2,n-1} & R_{2,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ R_{n-1,1} & R_{n-1,2} & \cdots & 0 & R_{n-1,n} \\ R_{n,1} & R_{n,2} & \cdots & R_{n,n-1} & 0 \end{bmatrix} \tag{3.35}$$

- **Time-varying parameters**: all instances of $\dot{q}$, $\dot{Q}$, T_i, T_j. To ensure that the temperature gradients $(T_j - T_i)$ and $(T_j^4 - T_i^4)$ are compatible with $C_{con,ij}$ and R_{ij}, the following is done: the vector T (length n) is converted into a matrix of size nxn and subtracted with its transpose. The result is the gradient resulting in a heat flow **towards** each node. So a positive entry means heat flow to the node, and a negative entry means heat flow away from the node. The same process is done for $(T_j^4 - T_i^4)$. The result is a matrix which is symmetric in magnitude, but the signs of the entries below the diagonal are opposite to the signs of the entries above the diagonal.

$$\nabla T = \begin{bmatrix} T_1 & T_2 & \cdots & T_{n-1} & T_n \\ T_1 & T_2 & \cdots & T_{n-1} & T_n \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ T_1 & T_2 & \cdots & T_{n-1} & T_n \\ T_1 & T_2 & \cdots & T_{n-1} & T_n \end{bmatrix} - \begin{bmatrix} T_1 & T_1 & \cdots & T_1 & T_1 \\ T_2 & T_2 & \cdots & T_2 & T_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ T_{n-1} & T_{n-1} & \cdots & T_{n-1} & T_{n-1} \\ T_n & T_n & \cdots & T_n & T_n \end{bmatrix} \tag{3.36}$$

$$= \begin{bmatrix} 0 & T_2 - T_1 & \cdots & T_{n-1} - T_1 & T_n - T_1 \\ T_1 - T_2 & 0 & \cdots & T_{n-1} - T_2 & T_n - T_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ T_1 - T_{n-1} & T_2 - T_{n-1} & \cdots & 0 & T_n - T_{n-1} \\ T_1 - T_n & T_2 - T_n & \cdots & T_{n-1} - T_n & 0 \end{bmatrix}$$

$$\nabla T^4 = \begin{bmatrix} T_1^4 & T_2^4 & \cdots & T_{n-1}^4 & T_n^4 \\ T_1^4 & T_2^4 & \cdots & T_{n-1}^4 & T_n^4 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ T_1^4 & T_2^4 & \cdots & T_{n-1}^4 & T_n^4 \\ T_1^4 & T_2^4 & \cdots & T_{n-1}^4 & T_n^4 \end{bmatrix} - \begin{bmatrix} T_1^4 & T_1^4 & \cdots & T_1^4 & T_1^4 \\ T_2^4 & T_2^4 & \cdots & T_2^4 & T_2^4 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ T_{n-1}^4 & T_{n-1}^4 & \cdots & T_{n-1}^4 & T_{n-1}^4 \\ T_n^4 & T_n^4 & \cdots & T_n^4 & T_n^4 \end{bmatrix}$$
$$= \begin{bmatrix} 0 & T_2^4 - T_1^4 & \cdots & T_{n-1}^4 - T_1^4 & T_n^4 - T_1^4 \\ T_1^4 - T_2^4 & 0 & \cdots & T_{n-1}^4 - T_2^4 & T_n^4 - T_2^4 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ T_1^4 - T_{n-1}^4 & T_2^4 - T_{n-1}^4 & \cdots & 0 & T_n^4 - T_{n-1}^4 \\ T_1^4 - T_n^4 & T_2^4 - T_n^4 & \cdots & T_{n-1}^4 - T_n^4 & 0 \end{bmatrix} \tag{3.37}$$

Geometry Definition

The conductive heat flows can be calculated without the definition of any geometry. In some cases, geometry and material properties could be used to compute the conductive couplings, but it is not strictly needed. On the other hand, geometry is essential for determining the internal radiation couplings R_{ij}, for Equation 2.10 and Equation 2.11.

The geometry strictly needed to perform the internal radiative analysis are the dimensions of the flat plate (width and height) and the origin coordinates of the plate, see Figure 3.20. With this information, the relative position between two plates can be extracted by the software to convert those dimensions into the ones needed for Equation 2.10 and Equation 2.11. The Python software automatically generates a 3D plot of the nodal model, if the nodes are assigned a geometry, see Figure 3.21.

The radiative couplings are calculated semi-automatically. The user must define which nodes are connected to which, after which the view factors are calculated automatically based on the pre-assigned geometry.

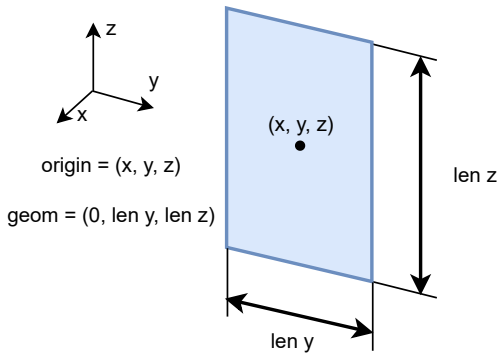


Figure 3.20: Example definition of a flat plate in the y-z plane, where the outward normal of the plane must be defined as zero thickness.

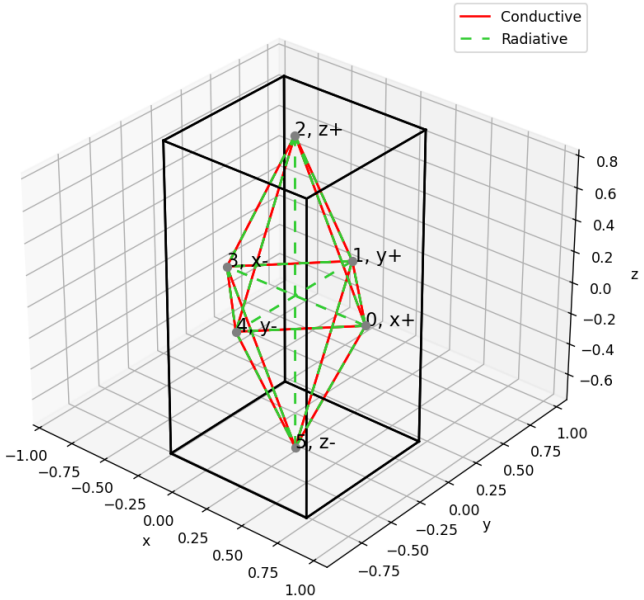


Figure 3.21: 3D plot showing each node's origin (grey dots), the outlines of all nodes/plates (solid black lines), the conductive (solid red lines) and radiative (dashed green lines) radiative connections, and the node names such as "x+" etc.

Integration Scheme
The heat equations for all nodes (Equation 3.32) must be integrated to find the transient response. The SciPy library `scipy.integrate` contains a number of integration schemes [56]:

- Explicit Runge-Kutta 5(4) (RK45): Fifth-order explicit Runge-Kutta method using fourth-order error estimations to determine the required time step resulting in the desired error tolerance.
- Explicit Runge-Kutta 8 (DOP853): Eighth-order explicit Runge-Kutta method using seventh-order error estimations to determine the required time step resulting in the desired error tolerance.
- Implicit Runge-Kutta 5 (Radau IIA): Fifth-order implicit Runge-Kutta method, belonging to the Radau IIA family, using third-order error estimations to determine the time step resulting in the desired error tolerance.

Runge-Kutta methods are commonly used, and their order can be increased simply by adding more terms. The explicit methods follow the Runge-Kutta following form [57, pp.907-935]:

$$\begin{aligned} y_{n+1} &= y_n + h\sum_{i=1}^{s} b_i k_i, \text{ where} \\ k_i &= f(t_n + c_i h,\, y_n + h\sum_{j=1}^{i-1} a_{ij} k_j),\, i = 1, 2, \cdots, s \end{aligned} \tag{3.38}$$

while the implicit methods have a slightly different form for k_i, following:

$$k_i = f(t_n + c_i h,\, y_n + h\sum_{j=1}^{s} a_{ij} k_j) \tag{3.39}$$

where it can be noticed that the sum now goes up to $j = s$ instead of only $j = i - 1$. The coefficients a_{ij} form the Runge-Kutta matrix, b_i are the weights, and c_i are the nodes. Depending on the order (and type, explicit vs implicit) of the method, this matrix will vary.

A great advantage of using either of these methods from the SciPy library is that they all include variable time-stepping. In essence, the regular Runge-Kutta algorithms are used, but low-order error estimations are used to vary the time step accordingly. This is especially valuable for thermal models, since their stiffness depends on the physical properties of the model. Specifically, the time constant τ of the system determines how small the time step must be for the integration. For a purely conductive model, the time constant can be calculated with [58]:

$$\tau = \frac{C_{cap}}{C_{con}} \tag{3.40}$$

where τ is the time constant in [s], C_{cap} is the thermal capacity in [J/K], and C_{con} is the thermal conductivity in [W/K]. A slow system has a large τ because it has a large mass and low conductivity; a fast system has a small τ because it has a small mass and high conductivity. For models that include both conduction and radiation, the exact time constant is more difficult to compute, as the radiative transfer adds a fourth order (T^4) to the equations. However, the physical meaning of the time constant remains the same. Consequently, the adaptive time-stepping from the SciPy library allows to solve both slow and fast thermal systems, without the user requiring to manually tweak the time step.

The previously indicated methods were tested for convergence with a sample thermal model. This sample model is described in more detail in Chapter 5, and it is a simplified version of a satellite in orbit. The implicit method (Radau) was chosen as a benchmark; all other methods will be compared to the benchmark as if it is the absolute truth.

It should be noted that the variable time-stepping in SciPy can sometimes result in inaccurate results, since it may skip a number of time steps and interpolate the results in between. If the user notices oscillatory behaviour in the output, they can set the user parameter `limit_step` to True, preventing SciPy from making large jumps in time. Evidently, this also increases the computational time.

Lastly, the Python implementation of the heat balance equation only has external heat data (solar flux etc.) at the fixed time steps. When SciPy varies the time step dynamically, either the closest value of the input data can be used, or the input data can be interpolated. The user of the software can indicate this with the Boolean `interp_inputs`.

The results of the integrator convergence analysis are shown in Figure 3.22 and Figure 3.23. Firstly, ”non_rot” refers to a non-rotating spacecraft. If a rotating spacecraft were used for this benchmark, numerical effects such as aliasing would have corrupted the consistency of the RMS errors. Secondly, it is clear that the implicit methods (all versions of Radau) perform better than the explicit methods (RK45 and DOP853), as the explicit methods reach a plateau beyond a time step of 10^0 seconds. Thirdly, it appears that the interpolation of the external heat inputs does not have a large effect on convergence, while the limitation of SciPy's variable time stepping does have an effect.

The Python code has the default integrator set to Radau, with the `interp` and `limstep` commands set to False. If the simulation is especially slow, the switch could be made to DOP853 or RK45, possibly with the `limstep` command set to True. DOP853 shows stronger, higher frequency, oscillations than RK45, if both `interp` and `limstep` are False.

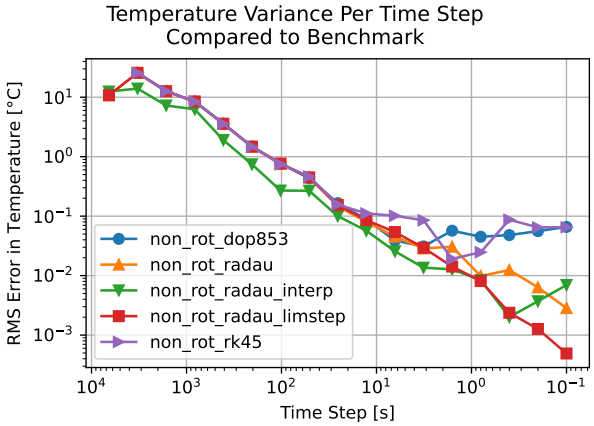


Figure 3.22: Convergence (RMS error) of numerous integrators from the SciPy library, compared to the benchmark Radau method. ”non_rot” indicates that the spacecraft is not rotating.

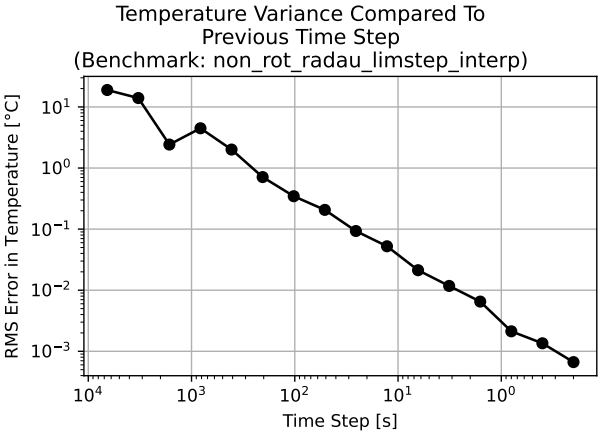


Figure 3.23: Convergence (RMS error) of the benchmark Radau method, where the variable time-stepping is constrained to be maximally the fixed time step, and where the external heat inputs are interpolated for the variable time steps. ”non_rot” indicates that the spacecraft is not rotating.

3.5. Example Case - FUNcube-1 Satellite

An example case to illustrate the workflow of the software is shown in this section, using the model of FUNcube-1 which will also be used for the validation and sensitivity analysis. All Python files are also found on GitLab [8], and a backup repository is also found on GitHub [9]; it is recommended to refer to those files when getting started with the software. Furthermore, the user manual, see Appendix C, is also a useful reference, providing more detail on the required syntax etc.

When creating a thermal model, it is recommended to initiate a separate Python file, and use the import statements as shown below. Moreover, it is recommended to use the `show_available_materials` function to print a summary of the material database in the Python console. The material names can then be easily copied into the material assignment of thermal nodes.

```
import numpy as np
from ThermalBudget import Node, NodalModel, OrbitalModel, show_available_materials
from CommonNodalModels import make_5node_pcb, assign_q_ext_to_pcb

show_available_materials()
```

3.5.1. Example Orbit & Environment Radiation

Most of the time, the analysis starts with the orbital heat flux analysis, which was explained in Section 3.3. The Python class responsible for this is `OrbitalModel`. The class' functions were also visualised in Figure 3.2. One of the ways to define an orbit and the surfaces receiving heat is as shown below. `outer_nodes` are the directions of the outer panels of the spacecraft, using the coordinates from Figure 3.7. The beta angle can also be indirectly defined by providing the inclination and RAAN. If the angular rates are not provided, they are assumed to be zero; if they are provided, they should be formatted as shown below, where the rates are $[\omega_x, \omega_y, \omega_z]$ in deg/s. After defining the model, it can be computed, and the time array is assigned as shown, to be used later when defining the NodalModel.

```
outer_nodes = ['x+', 'y+', 'z+', 'x-', 'y-', 'z-']
FUNcubeOrbit = OrbitalModel(h=596e3, surfaces=outer_nodes, beta=30, day=226, n_orbits=2, dt=5,
    angular_rates=[0., 0., 0.])
FUNcubeOrbit.compute()
t = FUNcubeOrbit.t
```

The heat fluxes are now stored inside the FUNcubeOrbit (OrbitalModel) object. Next, the outer surfaces are defined, after which the heat fluxes of FUNcubeOrbit are assigned to those outer surfaces. In this case, the outer surface is a PCB. Use can be made of the `make_5node_pcb` function to create a 5-node PCB, after which the `assign_q_ext_to_pcb` function assigns the heat fluxes to the PCBs. If the outer node is not a PCB but a "standard" node, the syntax is slightly different, see Appendix C.

"xplus" is the NodalModel of the PCB, while "xplusA", "xplusB" etc. are the nodes inside of the xplus NodalModel. It can be nice to have both the NodalModel objects and the Node objects separately available, therefore, they are both returned by the `make_5node_pcb` function. As seen below, the PCB is defined with a mass and geometrical properties (`origin` is the xyz location of the centre of the PCB; `geom` is the geometry of the PCB in xyz directions (thickness is set to zero there, and is defined separately)). Furthermore, `t=t` assigns the FUNcubeOrbit.t time array to the PCB. The coating is taken from the material database, and the `outer` argument states that these nodes are on the outside of the spacecraft, and thus receive environmental heat, and also lose heat radiatively to space.

```
xplus, xplusA, xplusB, xplusC, xplusD, xplusE = make_5node_pcb(title='x+', mass=0.02, origin
    =(0.05, 0, 0), geom=(0, 0.1, 0.1), thickness=0.0016, t=t, coating='
    solar_cell_mix_black_paint', outer=True)
# (...) same for the other 5 outer panels

assign_q_ext_to_pcb([xplus, yplus, zplus, xmin, ymin, zmin], FUNcubeOrbit)
```

3.5.2. Example Thermal Nodal Model & Transient Solution

Defining Individual Nodes and NodalModels

Next, the other nodes of the nodal model can be defined using the Python class `Node`, as in Figure 3.3. The inner PCBs are defined similarly as the outer PCBs shown before. If any extra nodes are desired to be added, such as a battery to one of the PCBs, that is done as shown below. First, the battery node is created, after which it is added to the PCB, and connected to the centre PCB node ('PCB2A') as a contact conduction, using the material database and a contact surface area.

```
PCB2, PCB2A, PCB2B, PCB2C, PCB2D, PCB2E = make_5node_pcb(title='PCB2', mass=mass_PCB, origin
    =(0., 0., -0.02), geom=(0.1, 0.1, 0.), thickness=thickness_PCB, t=t, power=0.160)
# Also give this PCB a battery
```

```
PCB2bat = Node(name='PCB2bat', material='copper', coating='black_paint', mass=0.05, origin=(0., 0., -0.015))
PCB2.add_node(PCB2bat)
PCB2.connect('PCB2A', PCB2bat, contact_obj='graphite', A=0.05**2)
# (...) similar for the other inner PCBs, but without the battery.
```

Non-PCB nodes are easier to create, and a separate function such as `make_5node_pcb` is not needed. The example below shows how to define a structural rod, simply as one node, and also how to define aluminium spacers between the PCBs; they could all be defined as just Nodes, but they can be added to a NodalModel as a bookkeeping tool. NodalModel `spcr12` contains the four spacers between PCB1 and PCB2, NodalModel `spcr23` between PCB2 and PCB3, etc.

```
rodB = Node(name='rodB', material='al7075', coating='al_unpolished', mass=0.08, origin=(0.05, 0.05, 0.))

spcr12 = NodalModel(title='spcr12')
spcr12B = Node(name='spcr12B', material='al7075', coating='al_unpolished', mass=0.02, origin=(0.04, 0.04, -0.03))
spcr12C = Node(name='spcr12C', material='al7075', coating='al_unpolished', mass=0.02, origin=(-0.04, 0.04, -0.03))
spcr12D = Node(name='spcr12D', material='al7075', coating='al_unpolished', mass=0.02, origin=(-0.04, -0.04, -0.03))
spcr12E = Node(name='spcr12E', material='al7075', coating='al_unpolished', mass=0.02, origin=(0.04, -0.04, -0.03))
spcr12.add_node([spcr12B, spcr12C, spcr12D, spcr12E])
# (...) similar for the spacers between the other PCBs.
```

Defining Main NodalModel, Connect Nodes, and Solve

Now that all the nodes have been defined, the main NodalModel must be created, and all nodes should be added to this model. If an entire NodalModel is added instead of just one Node, all Nodes of that model will be added. Next, some nodes need to be thermally connected. A radiative connection is simply defined with `rad=True`, assuming that the geometry is defined properly. Appendix C provides more details on how to define the geometry for radiative connections. A conductive connection can be either a contact or through connection (or a manually defined value in W/K). Contact connections can use the material database, and a contact area must be provided. Through connections need a distance between nodes, and a perpendicular surface area. For a through connection, both nodes must have the same material, and that material's properties are automatically used.

The simulation is run with the `solve` command, which stores the results within the FUNcubeModel object. The results can be plotted in a standardised format using `show_plots`, but custom plots can also be made by manually extracting data such as FUNcubeModel.T for the temperatures, etc. If the model contains many nodes, it is recommended to use the `whichnodes` argument. With this argument, the desired nodes are entered in a list, and only those nodes and their connections are shown in the plots. The arguments `showrad`, `shownodes`, `showtemps`, and `showflows` are by default True, but they can be used to turn off certain plots if they are not needed.

The `animate_attitude` command shows a 3D animation of the spacecraft's attitude, which can be a good sanity check when giving the spacecraft nonzero rotational rates.

```
FUNcubeModel = NodalModel(t=t, title='FUNcubeModel')
FUNcubeModel.add_node([PCB1, PCB2, PCB3, PCB4, PCB5, xplus, yplus, zplus, xmin, ymin, zmin, spcr12, spcr23, spcr34, spcr45, ring_zm, ring_z, rodB, rodC, rodD, rodE])
FUNcubeModel.connect(PCB1A, [zminA, PCB2A], rad=True)  # radiative
FUNcubeModel.connect(PCB1B, spcr12B, contact_obj='al_al', A=1e-5)  # contact
FUNcubeModel.connect(ring_zm, [rodB, rodC, rodD, rodE], L_through=0.05, A=0.05**2)  # through
# (...) more connections

FUNcubeModel.solve()
FUNcubeModel.show_plots(showrad=True, shownodes=True, showtemps=True, showflows=True, whichnodes=['z-A', 'z-B', 'z-C', 'z-D', 'z-E'])
FUNcubeOrbit.animate_attitude()
```

3.5.3. Example Plots

After calling `show_plots`, 5 different plots are created. They are explained in the following sub-sections, according to the nicknames used in the `show_plots` function.

”shownodes” Plots

The first plot is a 3D visualisation of the nodes in the NodalModel. This plot refers to the aforementioned `shownodes` argument. If the model contains many nodes, the plot would look like Figure 3.24. This is unreadable, unless the plot is zoomed in. To fix this, the `whichnodes` argument in the `show_plots` function can be used to select the desired nodes. The result is Figure 3.25; a more readable plot.

These plots show nodes that have geometrical properties (radiative nodes that appear as a flat plate), but they also show nodes that just are a point in space. The latter type of node does not actually have any geometry, but the point in the 3D plot still gives a useful overview of which nodes are approximately where in the model.

The structure of a 5-node PCB (explained in Section 4.2.2) can also be seen in Figure 3.25. The solid black outline represents the geometry of the centre node ”A”, while the other nodes B-E represent the corners of the PCB that are usually connected to aluminium spacers that are in between the PCB stack.

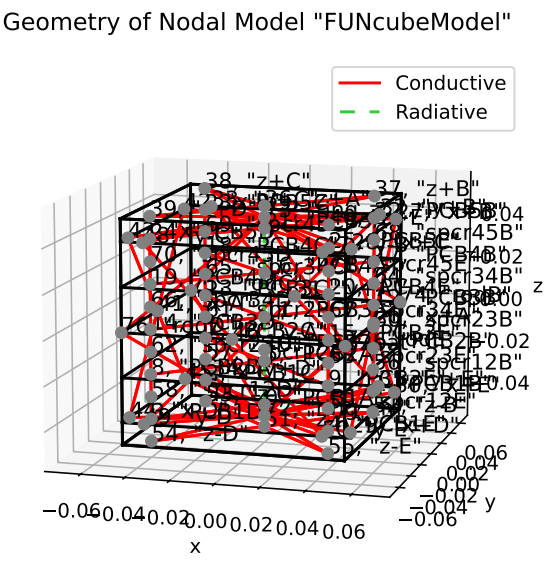


Figure 3.24: Unreadable plot if `whichnodes` is not used. 3D plot showing the NodalModel’s nodes and connections.

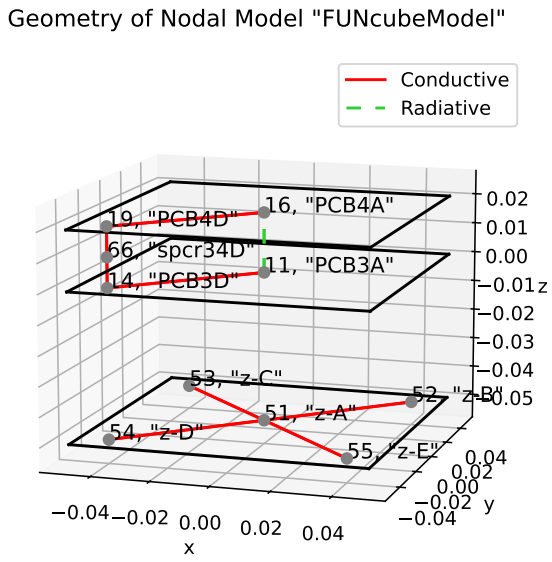


Figure 3.25: Readable plot using `whichnodes` to select fewer nodes. 3D plot showing the NodalModel’s nodes and connections.

”showrad” Plots

Next, the incoming heat on the outer nodes is shown in a plot as Figure 3.26. This plot refers to the aforementioned `showrad` argument. The incoming planet (Earth) IR heat, albedo heat, solar heat, and internal powers are shown. It should be noted that these are not the heat fluxes as output by the OrbitalModel, but the actual powers in Watt.

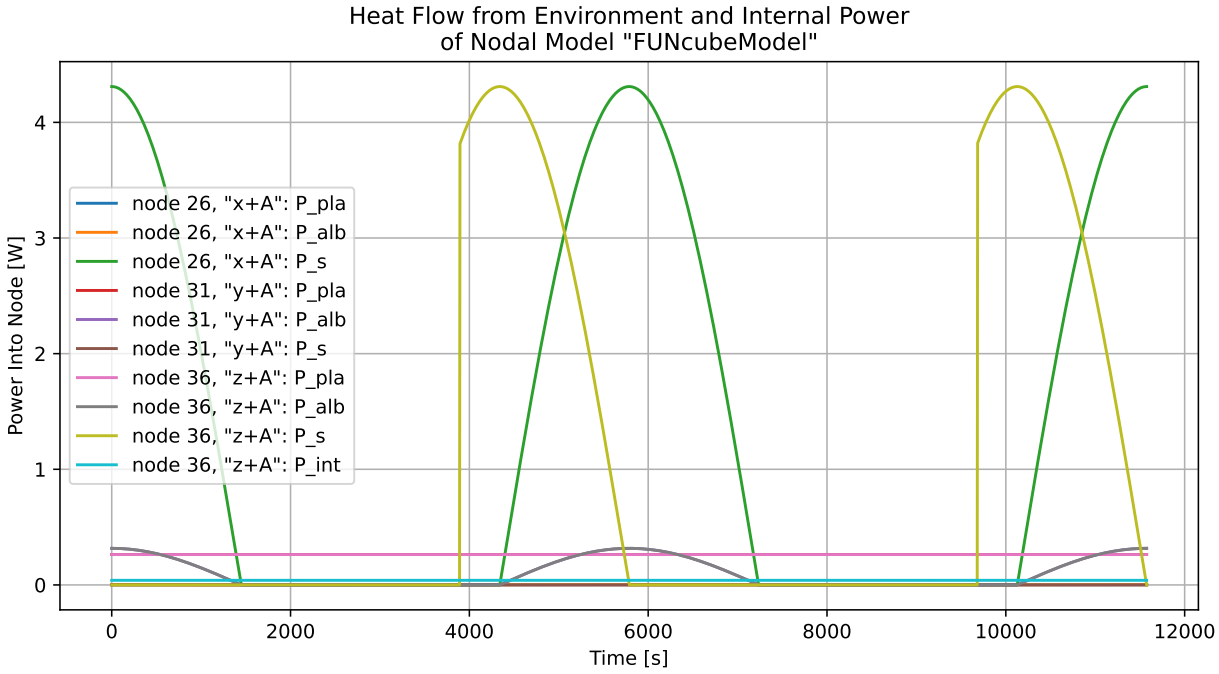


Figure 3.26: Incoming heat on a number of outer nodes of FUNcube-1.

”showtemps” Plots

The next plot are the selected nodes’ temperatures, as shown in Figure 3.24. This plot refers to the aforementioned `showtemps` argument. This plot may still be useful if all nodes are shown, since the general temperature profile can be seen even with many lines in the plot. However, it would be difficult to determine which line belongs to which node, since the colours in the Python plot repeat after a number of lines have been plotted.

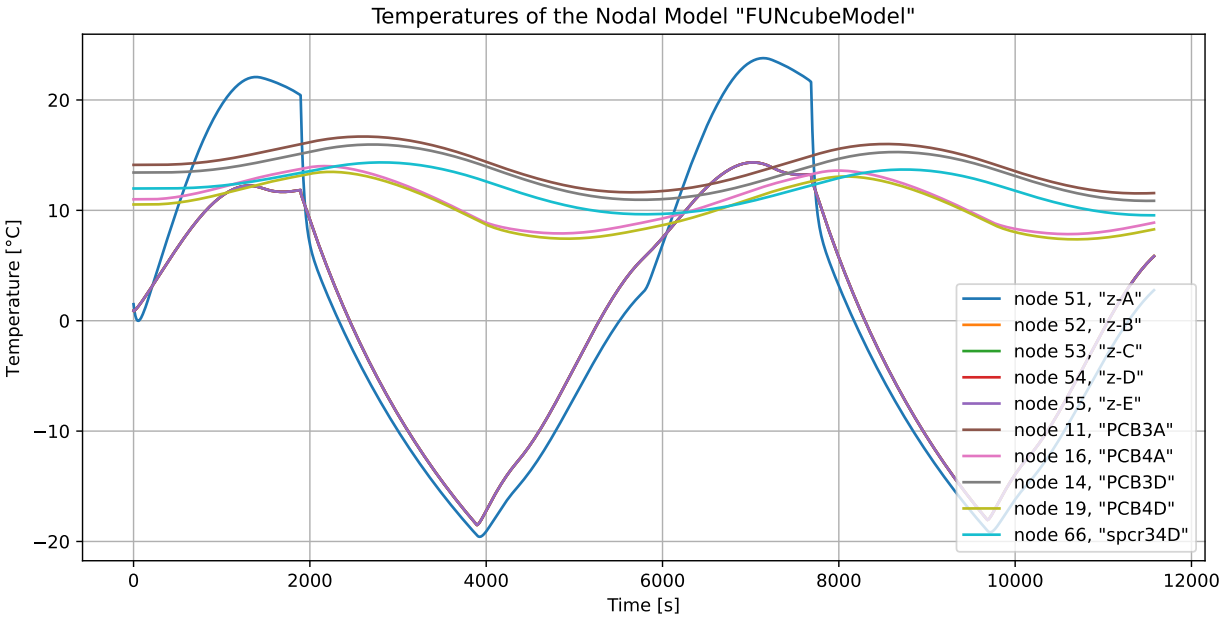


Figure 3.27: Transient temperatures of a number of selected nodes of FUNcube-1.

”showflows” Plots
Next, two more plots show the heat flows between the selected nodes. Figure 3.28 displays the heat flows of all conductive connections present, while Figure 3.29 displays the radiative heat flows. These plots refer to the aforementioned `showflows` argument.

Such figures may be useful during testing campaigns; for some types of testing setup, the heat flow through a certain part of the setup is known. For such tests, that heat flow can be compared to the values shown in these heat flow plots

The arrows in the legend indicate that the graph is positive when the heat flows from node A $\rightarrow$ B. E.g., the first element in Figure 3.28 shows nodes 11 $\rightarrow$ 14 (which are the nodes PCB3A and PCB3D). If that line is above zero, heat flows from node 11 to node 14; if it is negative it flows from node 14 to node 11.

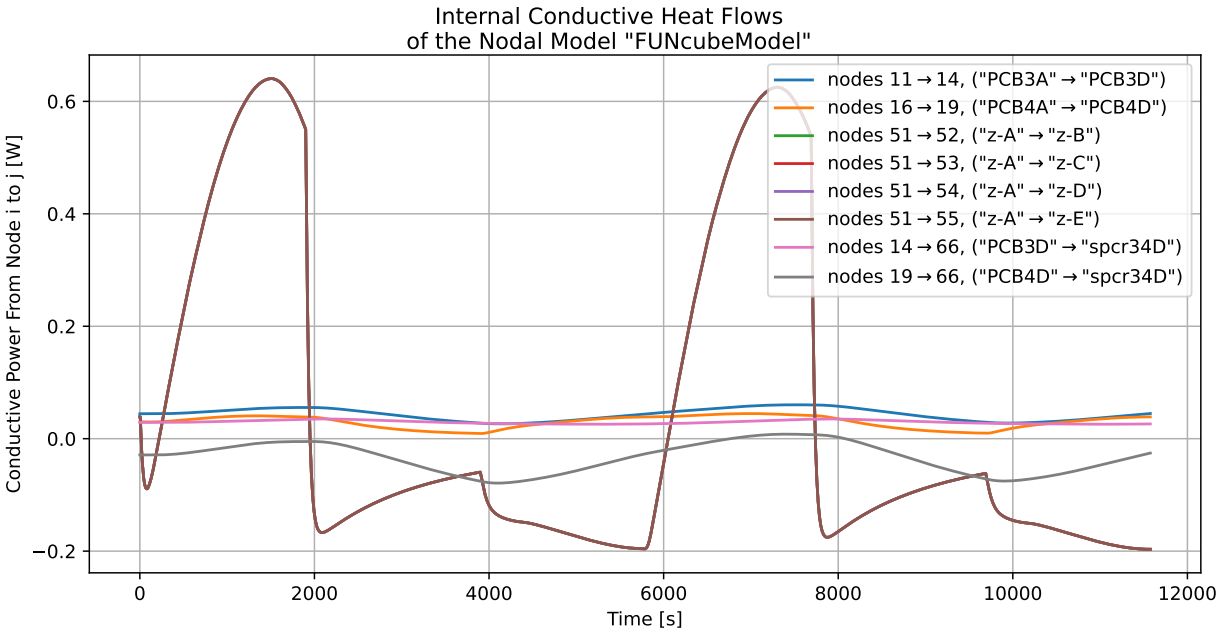


Figure 3.28: Conductive heat flows between a number of selected nodes of FUNcube-1.

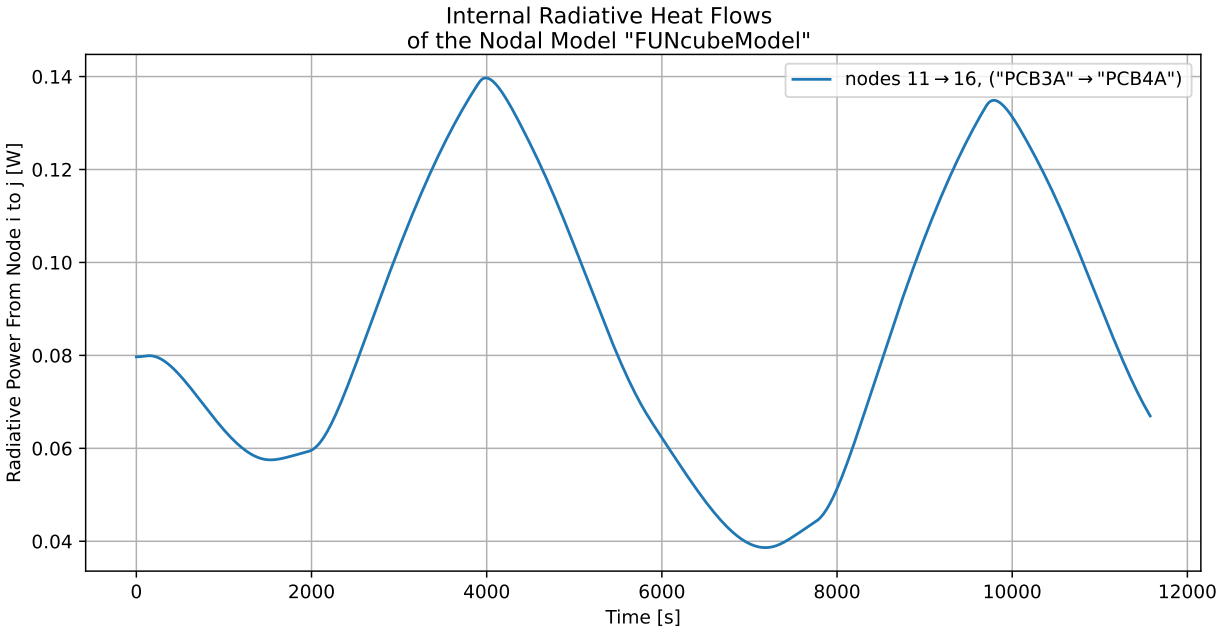


Figure 3.29: Radiative heat flows between a number of selected nodes of FUNcube-1.

Animate Attitude

Finally, the `OrbitalModel.animate_attitude` function can be used if the spacecraft has nonzero angular rates. A screenshot is shown in Figure 3.30. This plot is most likely just use to visually confirm whether the angular rates appear to be the same as input by the user.

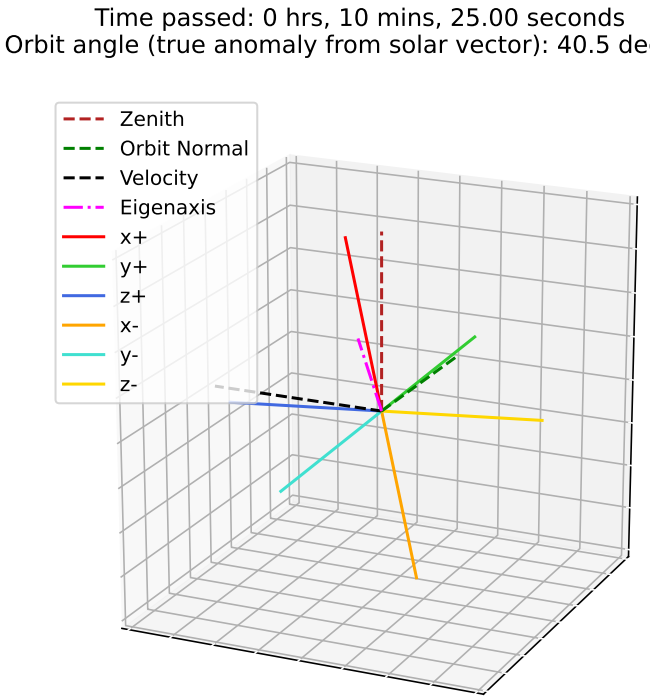


Figure 3.30: Screenshot of the 3D animation shown with `OrbitalModel.animate_attitude`. The eigenaxis refers to the axis around which the spacecraft is effectively rotating.

4

Verification & Validation

In this chapter, the outputs from the software are compared with results from ESATAN-TMS, and with flight data from the FUNcube-1 satellite. This type of verification and validation analysis is crucial for the reliability of the software. The verification ensures that the software performs the computations as intended, and that the results line up reasonably well with other, validated software (such as ESATAN-TMS). The validation uses real flight data to test the software's accuracy for real missions, and not just other simulations.

For this software, the verification is done with component-level and tests (view factors, eclipse, and beta angle) and system-level tests (external heat fluxes, non-orbiting thermal model, and an orbiting spacecraft). The validation is done with flight data from the FUNcube-1 spacecraft; data is available from 2016 and from 2024. The temperatures in both years are substantially different, which allows to test the software two different scenarios.

4.1. Verification

The software is verified on two different levels: component-level and system-level. First, component-level tests are performed to ensure that computations work in an isolated environment. For example, the view factors for different geometries are plotted and compared to reference data. Second, system-level tests verify that the computation as a whole works as expected; these computations are compared with ESATAN.

4.1.1. Component-Level Tests

Each test that does not contain numerous interrelated computations is considered a component-level test. In essence, the computation is isolated and can thus be tested relatively easily for its accuracy. In this sub-section, the view factors, eclipse, and beta angle computations are tested and compared to literature.

View Factors

Verification data is used from ECSS-E-HB-31-01 Part 1A [32], where many different formulas are given for different types of radiating geometries. Three different view factors are calculated in the software:

- **From a flat plate to a sphere**: the flat plate is a face of the spacecraft in orbit, and the sphere represents the Earth. This view factor is used for the Earth IR heat and albedo heat computations. The computation uses the equations from Section 3.3.6. Figure 4.1 and Figure 4.2 show the results from Python and ECSS, respectively. It should be noted that Figure 4.1 is almost identical to Figure 3.13, but the x-axis is defined differently.
- **Between two parallel plates**: one of the two formulas implemented to estimate the internal radiation in the spacecraft, using Equation 2.11. Figure 4.3 and Figure 4.4 show the results from Python and ECSS, respectively.
- **Between two perpendicular plates**: one of the two formulas implemented to estimate the internal radiation in the spacecraft, using Equation 2.10. Figure 4.5 and Figure 4.6 show the results from Python and ECSS, respectively.

The figures below are visually identical between the Python plots and the ECSS plots. The plots show identical shapes and identical values on the x- and y-axis, so the computations are considered verified.

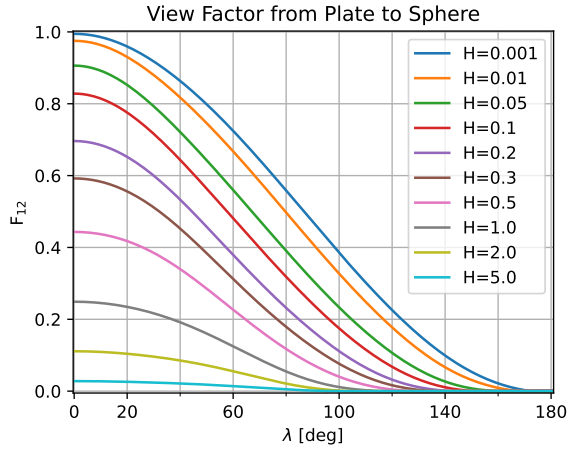


Figure 4.1: View factors from a flat plate to a sphere, generated by the Python software. $H = h/R_{Earth}$, as shown in Figure 3.12.

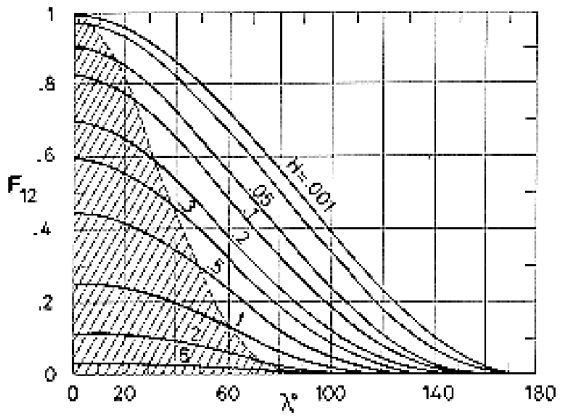


Figure 4.2: View factors from a flat plate to a sphere, taken from ECSS [32, p.19]. The grey area is the region for which a simplified formula can be used (not used in the Python software).

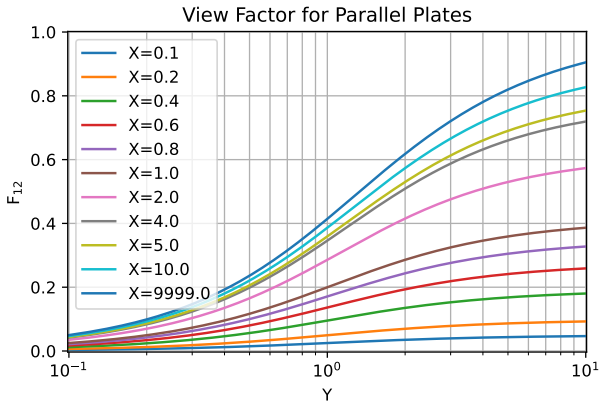


Figure 4.3: View factors for parallel plates, generated by the Python software. $X = a/c$ and $Y = b/c$, as shown in Figure 2.16.

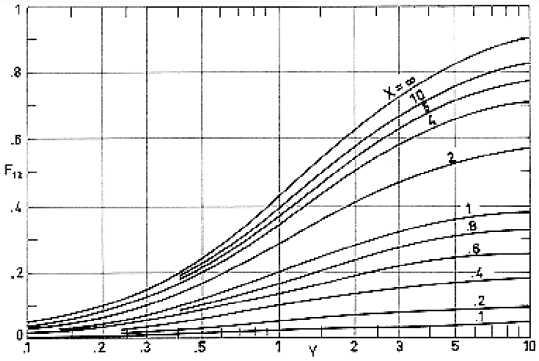


Figure 4.4: View factors for parallel plates, taken from ECSS [32, p.36].

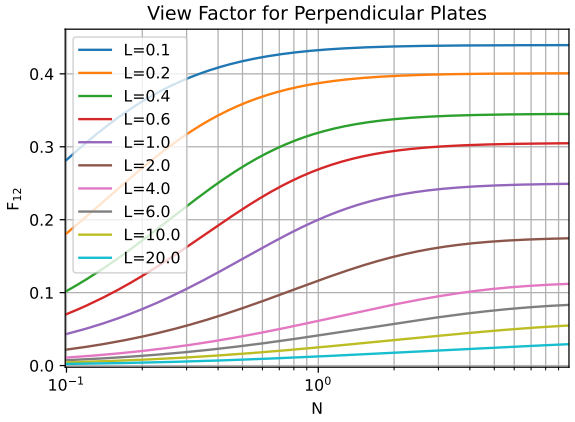


Figure 4.5: View factors for perpendicular plates, generated by the Python software. $N = a/b$ and $L = c/b$, as shown in Figure 2.15.

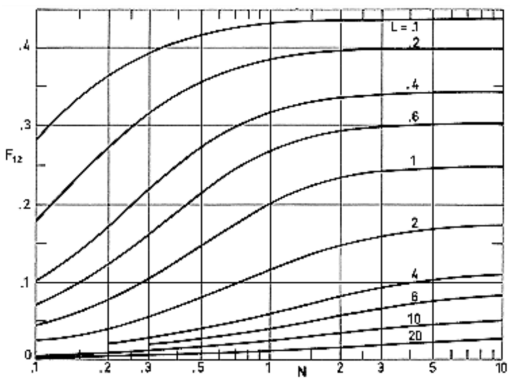


Figure 4.6: View factors for perpendicular plates, taken from ECSS [32, p.40].

Eclipse Characteristics

The eclipse fraction of a circular Low-Earth Orbit was computed in Section 3.3.7. Depending on the altitude, there is a certain cut-off beta angle, after which the orbit is completely illuminated. In this case of an orbit of 408 km altitude, Figure 4.7 and Figure 4.8 show that the cut-off angle is $\beta \approx 70°$. Furthermore, the maximum eclipse fraction, when $\beta = 0°$, is 0.39 in both plots. The function is considered verified.

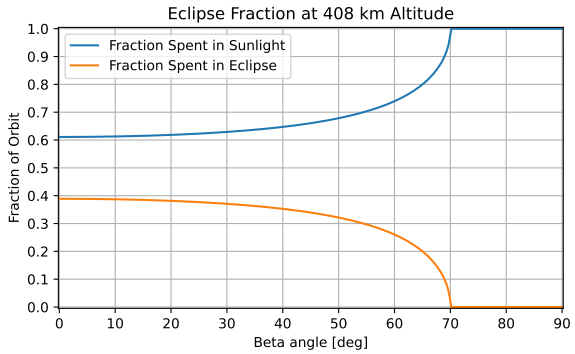


Figure 4.7: Fraction of the orbit that is in eclipse, for all positive beta angles (the plot mirrors around $\beta = 0$ for negative β), at 408 km altitude, computed by the Python software.

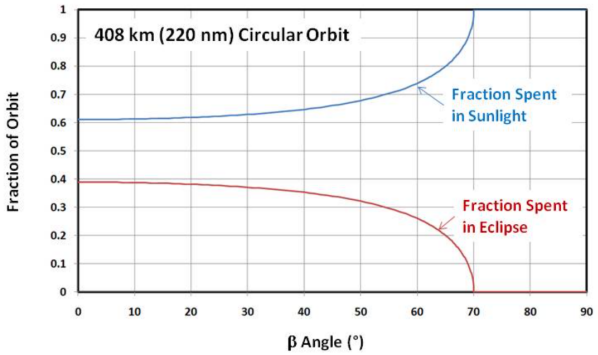


Figure 4.8: Fraction of the orbit that is in eclipse, for all positive beta angles (the plot mirrors around $\beta = 0$ for negative β), at 408 km altitude, taken from Rickman's online lecture [51].

Beta Computation from Orbital Parameters

Section 3.3.8 discussed a way of computing the beta angle if the Right Ascension of the Ascending Node (RAAN) and inclination are known. Besides performing a check on the equation written in the software, some visual feedback in a number of figures would also improve the level of verification. Since no reference data was found on this (besides Equation 3.29 itself), logical reasoning will be performed to ensure that the overall behaviour of the equation is correct. Values cannot be verified exactly.

Figure 4.10 shows the beta angles for different combinations of inclination and RAAN. This figure corresponds to March in Figure 4.9. Evidently, if RAAN = 0°, $\beta = 0°$ regardless of the inclination. When the RAAN is increased, see also Figure 4.9, β becomes positive, and attains its maximum value for a polar orbit (incl. = 90°). This once again makes sense when Figure 4.9 is consulted.

A similar plot is made for an orbit in the month December, shown in Figure 4.11. For an inclination of 0°, the beta angle is -23.44° regardless of the RAAN. The equatorial orbit (black colour) in Figure 4.9 is that orbit, and indeed shows that the Sun points "below" the orbit and is therefore negative, and its value is equal to the obliquity of the ecliptic. Once again observing Figure 4.11, increasing the RAAN results in smaller beta angles, regardless of inclination. This makes sense when imagining the December orbit in Figure 4.9 to have a change in RAAN.

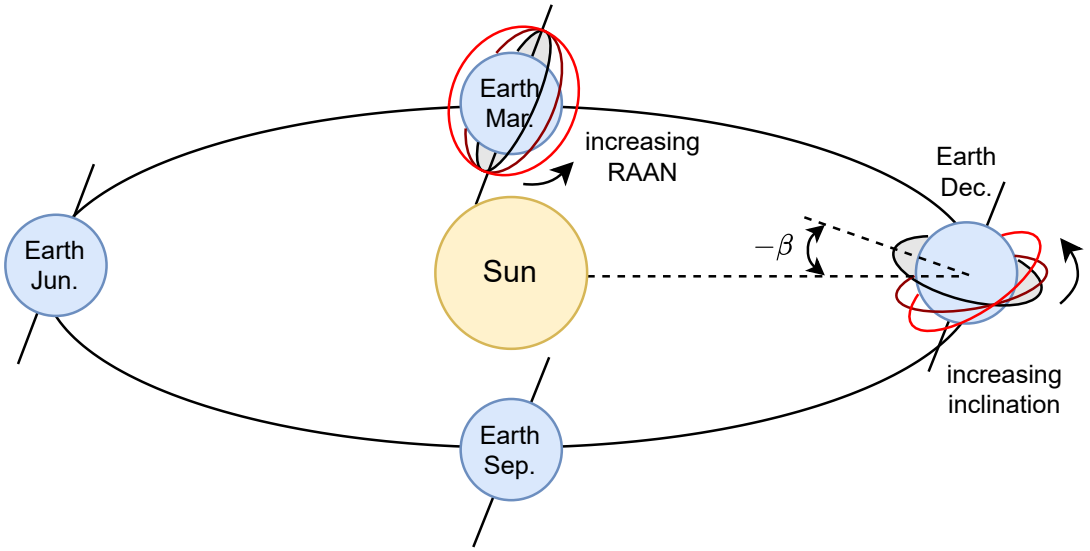


Figure 4.9: Schematic diagram showcasing the effect of orbital parameters on the beta angle. This figure aids the understanding of Figure 4.10 and Figure 4.11. β is indicated as negative, since it is positive when it looks ”onto” the orbit, see also Figure 3.6.

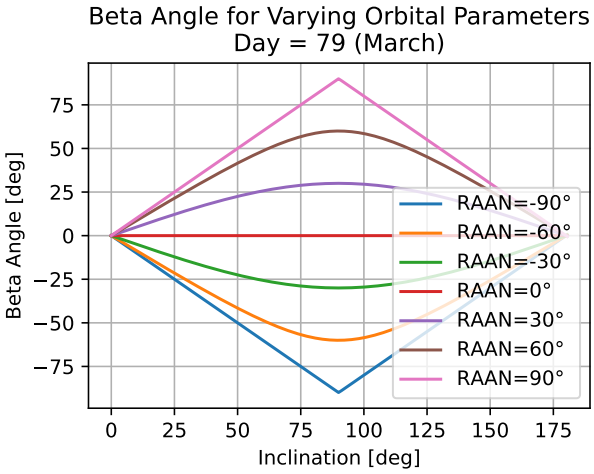


Figure 4.10: Beta angle for numerous combinations of the RAAN and inclination. On day 79, the vernal equinox points directly towards the Sun.

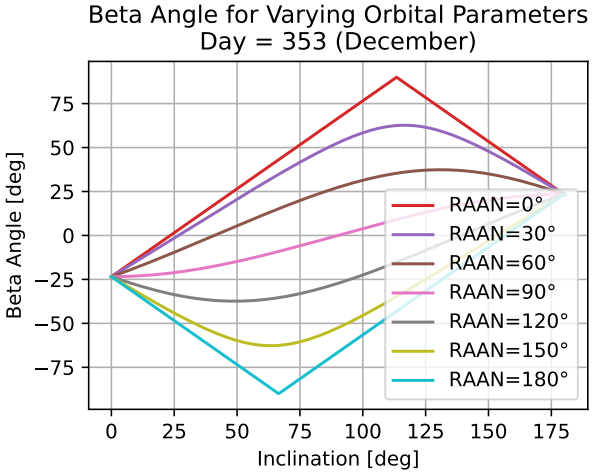


Figure 4.11: Beta angle for numerous combinations of the RAAN and inclination, for the month December.

4.1.2. System-Level Tests

This sub-section displays the results of numerous system-level tests. Since the software essentially contains two modules, the orbital module and the thermal module, both are tested separately, and then as a whole.

External Heat Fluxes

The first system-level test to be performed is the external heat flux analysis. It combines the functions that were tested in the component-level tests, and computes meaningful results. For now, a single plate is put in an orbit at 408 km altitude, which is the altitude used by L. Rickman [47, 51], so the results at this altitude can also be verified with Rickman’s examples. Different orientations of the plate were tested and compared with results from ESATAN-TMS. The main case will be a plate pointing towards the velocity vector ($\tau, \phi = 0$), with $\beta = 0$. However, both of these parameters are varied to test a few different scenarios:

- A ”default” case where the face is pointing in the flight direction and the altitude is 408 km. This is Figure 4.12 and Figure 4.13.
- A case with a different face orientation (nadir), at 408 km. There are six of these cases possible, since there are six flight directions (x+, y+, z+, x-, y-, z-, see Figure 3.7), which were all tested, but for conciseness are not shown here. This is Figure 4.14 and Figure 4.15.
- Two cases with different beta angles (45° and 80°), still based on the default case with the plate facing the flight direction, at 408 km. This is Figure 4.16, Figure 4.17, Figure 4.18, and Figure 4.19.
- Two cases with different altitudes (300 km and 1000 km), still based on the default case with the plate facing the flight direction, with $\beta = 0°$. This is Figure 4.20, Figure 4.20, Figure 4.22, and Figure 4.23.

The plots on the left side of the page show the heat fluxes on the plate by the Earth IR, albedo, and solar radiation. The solid lines are the Python simulation, while the scattered icons are from ESATAN. The plots appear to match well, although the albedo shows slightly larger errors than the other heat fluxes. To really show the errors better, error plots are shown on the right-hand side of the page. Finally, a table with all the error values are shown in Table 4.1.

The plots on the right side of the page show the difference between the ESATAN and Python simulations. For example, when Figure 4.13 shows a positive error, it means that the Python simulation estimated a too high heat flux, where ESATAN is considered the truth. If it is a negative error, the heat flux was underestimated by Python. The Root-Mean-Square Error (RMSE) errors were converted to a percentage by dividing them by the mean heat flux of the corresponding variable; the percentages were not calculated using the time-varying heat fluxes, since those are occasionally zero, making it impossible to calculate a percentage.

The right-hand side plots also show an ”x” for outliers, but those are cropped out of the frame to improve the plot’s readability. Such an outlier occurs when the satellite is entering or leaving the eclipse. If the eclipse time is slightly off, a large ”error” is detected, but the error is not valid since it is a time-shift issue, not a heat flux issue. Hence, the outliers are marked and not considered in the RMSE computations.

- The **Earth IR** error is essentially a constant value, dependent on the orientation of the plate. This is likely due to the fact that the Earth IR is assumed to be constant in both the Python code and in ESATAN, but the actual value used may be slightly different. The error is negligible; the largest error is seen in the nadir-facing case, Figure 4.14 and Figure 4.15, yet the error is still only (2.19 W/m^2)/(209 W/m^2)$\approx$1%.

- The **albedo** error shows large percentage errors, although the actual error in W/m^2 is comparable to the solar flux error. As also seen in Table 4.1, the error increases with increasing altitude. With increasing beta angle, the actual error decreases, but the percentage increases.
  It appears that, in most cases, the albedo is underestimated by the Python code. Figure 4.19 shows a clear trend where the error magnitude is the largest near the dark side of Earth (not necessarily in eclipse, since the case of Figure 4.18 and Figure 4.19 does not have an eclipse). An educated guess is that this error may occur due to a perfect ”cut-off” situation as the spacecraft passes a true anomaly of 90° (see Section 3.3.5). However, in reality, there is some albedo that reaches beyond this point, simply due to sunlight being reflected further. This is not taken into account in the Python simulation. The albedo error is more significant than the Earth IR error, and the physical mechanism that causes the albedo error is more complicated, as well.

- Finally, the **solar flux** error is the same order of magnitude as the albedo error. The two are related, since albedo is simply a reflection of sunlight. Figure 4.13, Figure 4.17, Figure 4.19, Figure 4.21, and Figure 4.23 all show an underestimation of the solar flux in the second half of the orbit, and the albedo shows the same trend. If the error were from the solar constant used, it would be directly proportional, a constant percentage, to the value of the solar heat flux. However, especially from Figure 4.19, the error appears to have a sinusoidal shape. It is unsure what the source is of this error, but due to the sinusoidal nature, it is expected that it may be related to the attitude of the plate.

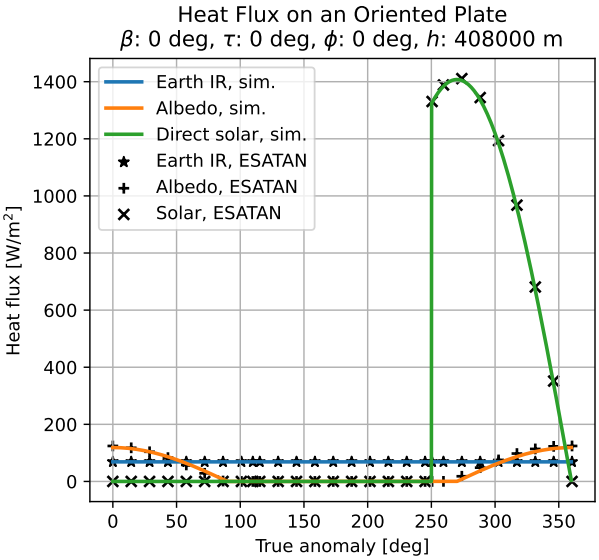


Figure 4.12: Python simulation compared with ESATAN results of the Earth infrared, albedo, and direct solar heat flux for a single plate oriented towards its velocity vector, for $\beta = 0°$.

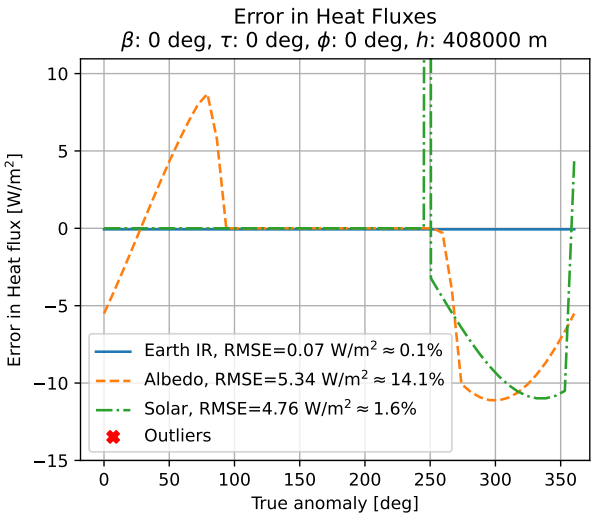


Figure 4.13: Errors between Python and ESATAN simulations, for a single plate oriented towards its velocity vector, for $\beta = 0°$. An outlier is present near the eclipse point, and cropped out of view.

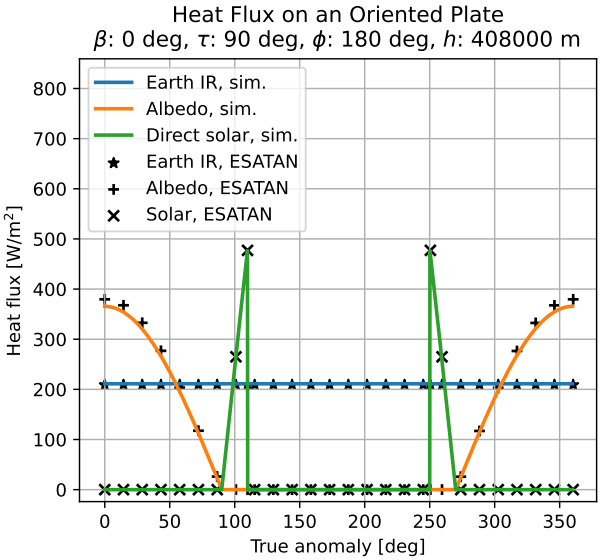


Figure 4.14: Python simulation compared with ESATAN results of the Earth infrared, albedo, and direct solar heat flux for a single plate oriented nadir Earth, for $\beta = 0°$.

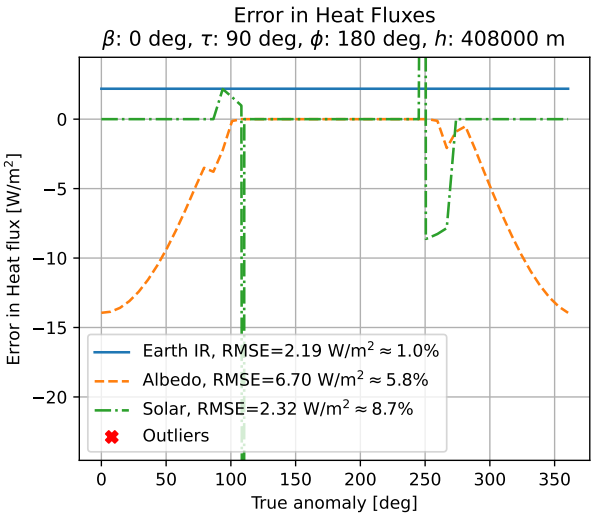


Figure 4.15: Errors between Python and ESATAN simulations, for a single plate oriented nadir Earth, for $\beta = 0°$. Two outliers are present near the eclipse points, and cropped out of view.

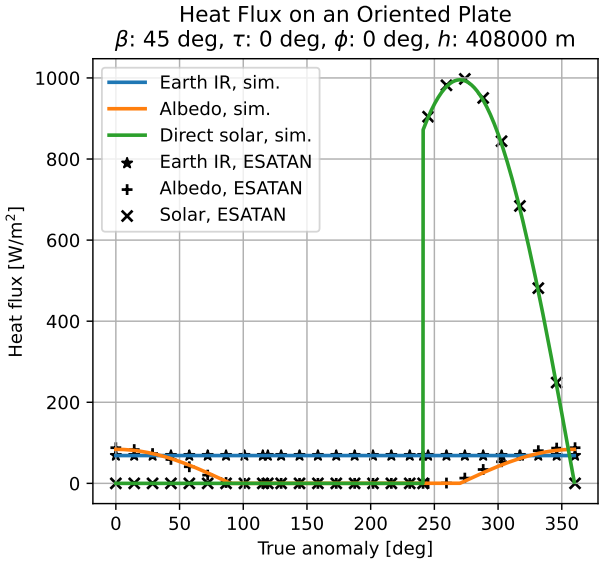


Figure 4.16: Python simulation versus ESATAN results of the Earth infrared, albedo, and direct solar heat flux for a single plate oriented towards its velocity vector, for $\beta = 45°$.

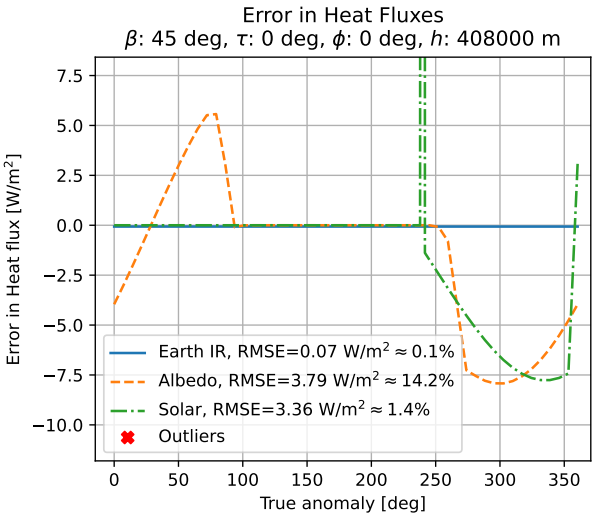


Figure 4.17: Errors between Python and ESATAN simulations, for a single plate oriented towards its velocity vector, for $\beta = 45°$. An outlier is present near the eclipse point, and cropped out of view.

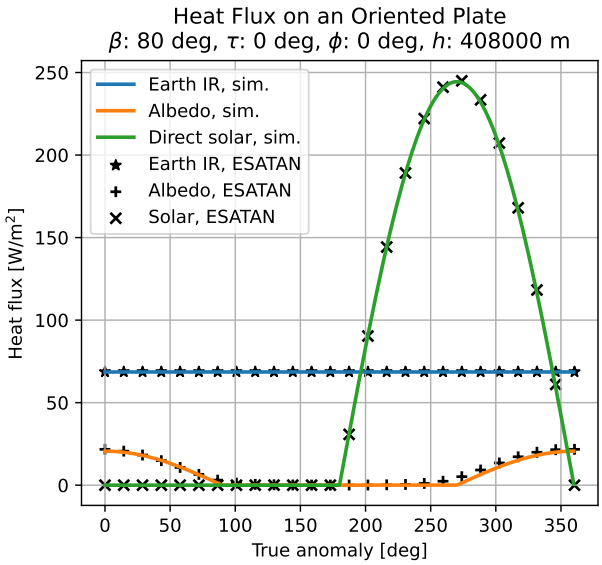


Figure 4.18: Python simulation versus ESATAN results of the Earth infrared, albedo, and direct solar heat flux for a single plate oriented towards its velocity vector, for $\beta = 80°$.

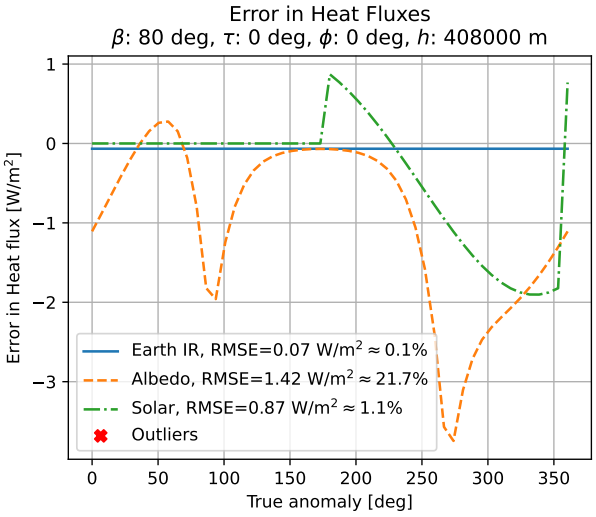


Figure 4.19: Errors between Python and ESATAN simulations, for a single plate oriented towards its velocity vector, for $\beta = 80°$. No outliers are present due to the absence of an eclipse.

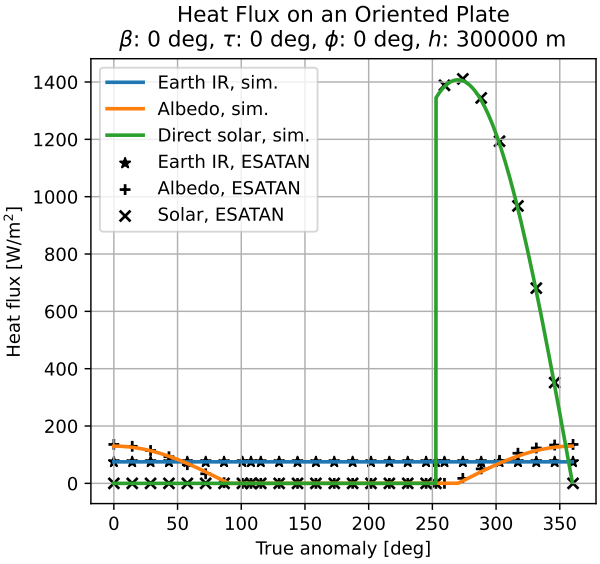


Figure 4.20: Python simulation versus ESATAN results of the Earth infrared, albedo, and direct solar heat flux for a single plate oriented towards its velocity vector, for $\beta = 0°$ at an altitude of 300 km.

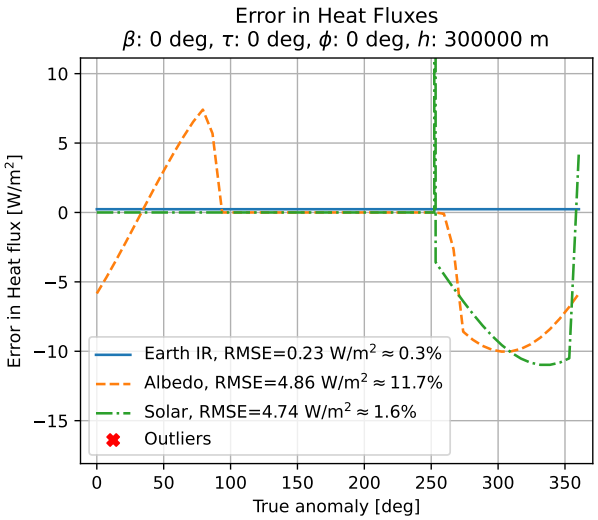


Figure 4.21: Errors between Python and ESATAN simulations, for a single plate oriented towards its velocity vector, for $\beta = 0°$ at an altitude of 300 km. An outlier is present near the eclipse point, and cropped out of view.

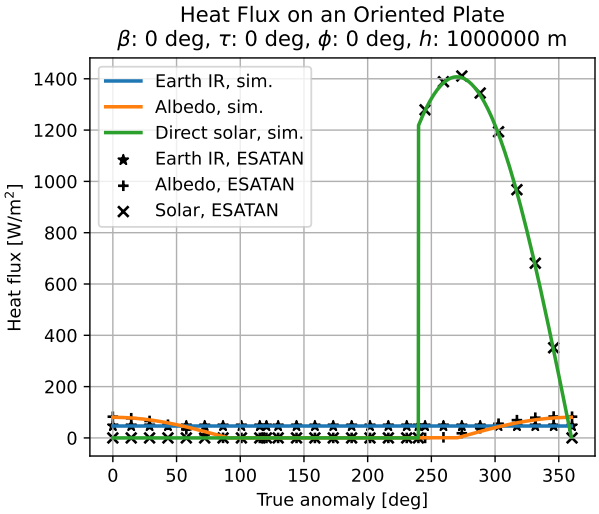


Figure 4.22: Python simulation versus ESATAN results of the Earth infrared, albedo, and direct solar heat flux for a single plate oriented towards its velocity vector, for $\beta = 0°$ at an altitude of 1000 km.

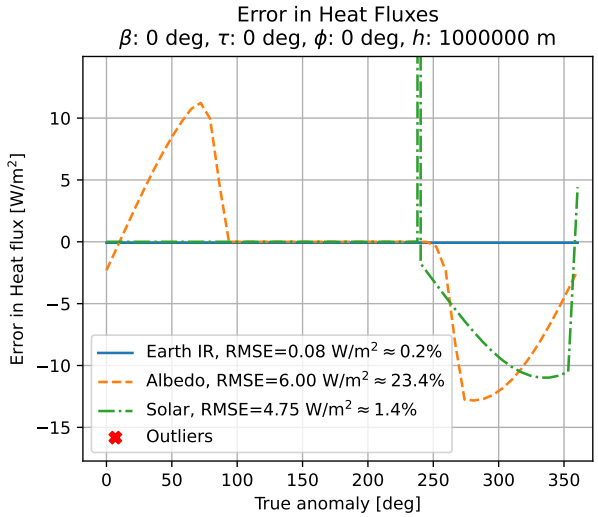


Figure 4.23: Errors between Python and ESATAN simulations, for a single plate oriented towards its velocity vector, for $\beta = 0°$ at an altitude of 1000 km. An outlier is present near the eclipse point, and cropped out of view.

Table 4.1: RMSE between the environmental fluxes computed by the Python software and computed by ESATAN-TMS. The Earth IR errors are negligibly low. The solar errors are moderately high, but low in percentage. The albedo errors are high.

Case (altitude, beta, attitude)	Earth IR		Albedo		Solar	
	W/m^2	%	W/m^2	%	W/m^2	%
408 km, $\beta = 0°$, toward velocity	0.07	0.1	5.34	14.1	4.76	1.6
408 km, $\beta = 0°$, nadir	2.19	1.0	6.70	5.8	2.32	8.7
408 km, $\beta = 45°$, toward velocity	0.07	0.1	3.79	14.2	3.36	1.4
408 km, $\beta = 80°$, toward velocity	0.07	0.1	1.42	21.7	0.87	1.1
300 km, $\beta = 0°$, toward velocity	0.23	0.3	4.86	11.7	4.74	1.6
1000 km, $\beta = 0°$, toward velocity	0.08	0.2	6.00	23.4	4.75	1.4

Temperature Response of Non-Orbiting Nodes

To test the thermal computations in an isolated way, a number of nodes are connected to each other via conductive couplings, and the power inputs are manually defined and unrelated to an orbit. A graphical representation of the example model used in this verification case is shown in Figure 4.24. The parameters were chosen such that the model is not symmetrical, and such that numerous sanity checks could be performed. A higher conductivity should result in the temperatures being close together; a large mass should result in a more constant temperature; different initial temperatures should show how the most extreme temperatures are pulled towards the average temperature.

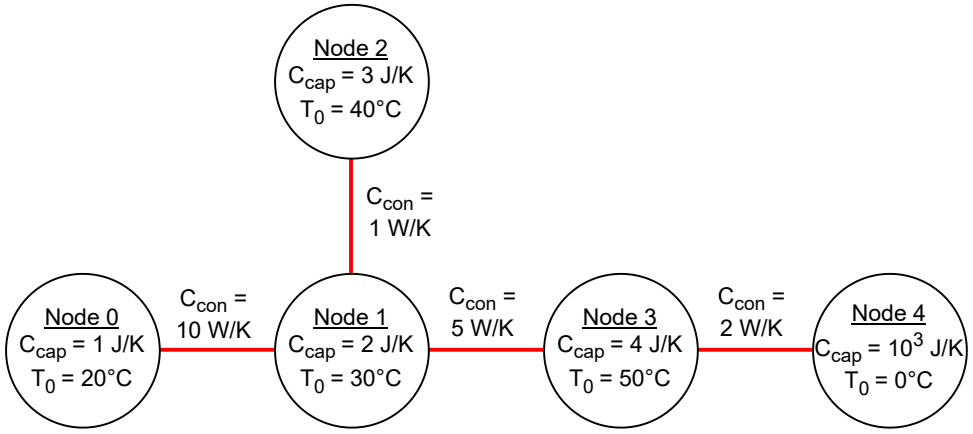


Figure 4.24: Five thermal nodes with their properties and connections, used for a transient analysis verification case.

The resulting temperatures are shown in Figure 4.25, and the temperature errors of the Python simulation compared to ESATAN are shown in Figure 4.26. As expected, the node with the largest heat capacity, node 4, has a nearly constant temperature compared to the other nodes. Moreover, nodes 0 and 1 have very similar temperatures, due to their high conductance value. The errors shown in Figure 4.26 are extremely small, and it was found that these errors depend on the solver chosen. The optimal solver is Radau (more information is available in Section 3.4.3), as shown in the figure. The error magnitude is proportional to the node's thermal mass. The lightest node, node 0, has the largest errors, while the heaviest node, node 4, has the smallest errors. Regardless, all errors are of negligible magnitude.

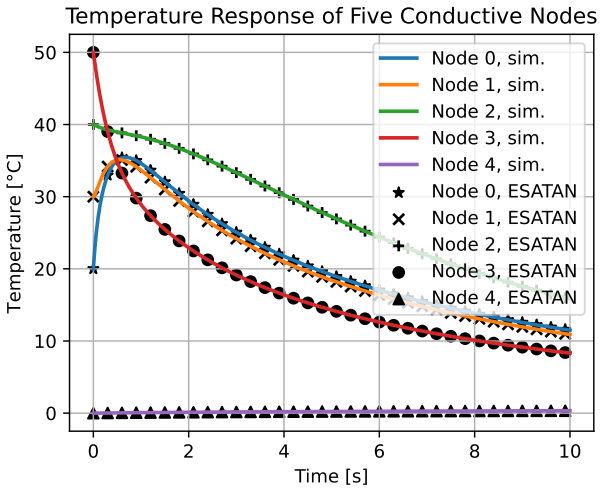


Figure 4.25: Temperatures of five nodes that are have conductive couplings, computed by a Python simulation and ESATAN.

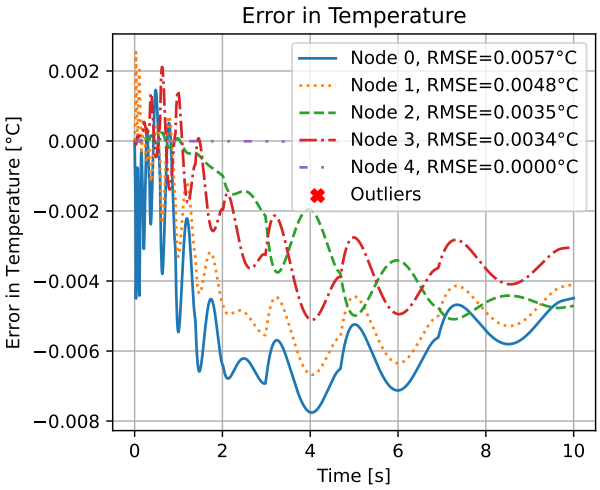


Figure 4.26: Errors in temperature of five conductively connected nodes. There are no outliers present.

Temperatures of Orbiting Box

After having verified the orbital module and the thermal module separately, they are combined into one model of a simple orbiting box. The box has dimensions of 1x1x1 metre, each node (face) having a thermal capacitance of 1000 J/K. The plates are connected radiatively, and conductively with 1 W/K to their neighbouring nodes. These dimensions are much larger than small satellites, but units of 1 m are easy to work with, and as long as all relative dimensions are the same, the size does not influence the thermal behaviour, as long as the conductances are held constant. In reality, size does evidently influence the behaviour, since the conductances usually decrease with increasing size.

Simultaneously with assessing a combined orbital & thermal model, the effect of internal radiation is assessed. The view factors were already verified in Section 4.1.1, but now, the radiation is tested within the context of a full simulation. Therefore, one simulation (in both Python and ESATAN) is conducted with internal radiation, and one without.

The results for the test with radiation are shown in Figure 4.27 and Figure 4.28, while the tests without radiation are shown in Figure 4.29 and Figure 4.30. Only two nodes are shown, to prevent the plots from becoming unreadable, but the RMSE values of the other nodes are still summarised in Table 4.2. The model without internal radiation shows larger errors, which is likely due to the more extreme temperatures attained. The internal radiation pulls the temperatures of the nodes closer together, hence resulting in slightly more mild temperatures.

Furthermore, it can be seen in the figures hat the Python simulation, on average, estimates the heat transfer between nodes to be slightly too large, compared to ESATAN. This shows in the orange, dashed lines of the x- node in Figure 4.28 and Figure 4.30; the Python temperatures are too low compared to ESATAN during the hotter periods, and the Python temperatures are too high during the colder periods. In other words, Python estimates a stronger thermal connection between the nodes. It is not due to radiation, because the same effect is seen for both cases (with and without radiation). This would suggest that the conduction is the source of error, but it was already shown in Section 4.1.2 that purely conductive nodes only exhibit extremely low temperature errors. It is therefore assumed that the errors are a result of the errors in the environmental heat flux computations, as shown in Section 4.1.2.

To conclude, the temperature errors in the combined orbital & thermal model are most likely due to the environmental heat flux computation. There is no data to suggest that the assumptions on internal radiation

for the Python code result in large errors in temperature, even for this extreme case where temperatures of $\pm 100°C$ were reached.

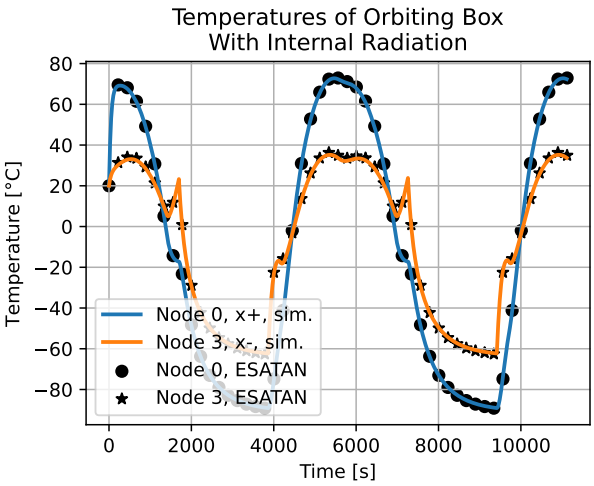


Figure 4.27: Temperatures of nodes x+ and x-, with internal radiation, computed by a Python simulation and ESATAN.

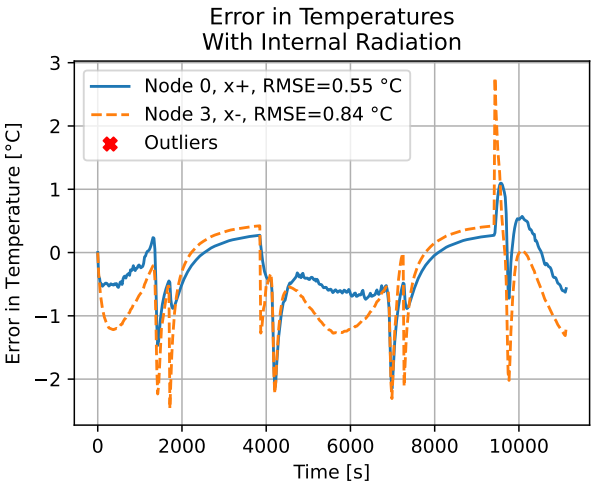


Figure 4.28: Errors in temperatures of nodes x+ and x-, with internal radiation. There are no outliers present.

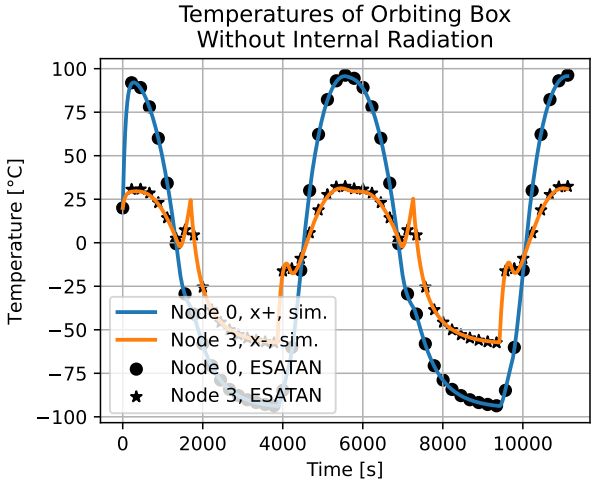


Figure 4.29: Temperatures of nodes x+ and x-, without internal radiation, computed by a Python simulation and ESATAN.

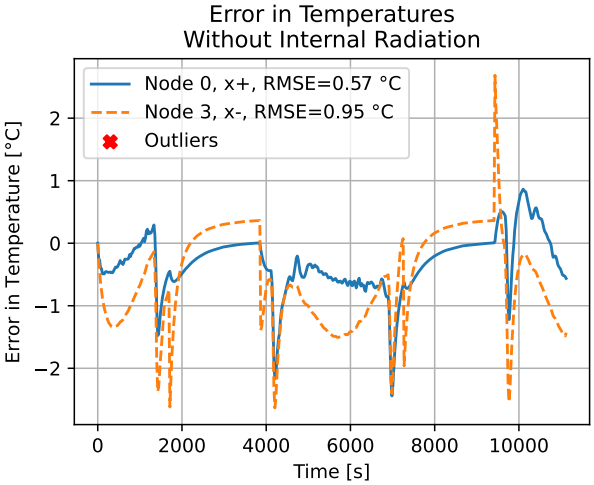


Figure 4.30: Errors in temperatures of nodes x+ and x-, without internal radiation. There are no outliers present.

Table 4.2: Root Mean Square Error of all nodes of the box, for both the case with and without internal radiation.

Node	RMSE with rad.	RMSE without rad.
0, x+	0.55°C	0.57°C
1, y+	0.59°C	0.89°C
2, z+	0.69°C	0.80°C
3, x-	0.84°C	0.95°C
4, y-	0.50°C	0.83°C
5, z-	0.70°C	0.80°C

4.2. Validation

After having verified the software with literature and ESATAN-TMS on component and system level, a realistic satellite model is created, and compared to flight data. The validation is performed using flight data from the FUNcube-1 satellite. This satellite was chosen over the Delfi-PQ spacecraft, since FUNcube-1 has whole-orbit flight data available, instead of the less frequent data points available from Delfi-PQ. Unfortunately, the hardware of the FUNcube-1 is not readily available, so the thermal model is based on all the information mentioned in the FUNcube handbook [59] and information from S. Speretta. Numerous images were used to acquire information on the satellite's architecture, as shown in Figure 4.31, Figure 4.32, and Figure 4.33. Finally, data on parameters such as internal power was acquired via oral communication with professor S. Speretta. The Python file defining the nodal model of FUNcube-1 can be found on GitLab [8] and GitHub [9].

Two different sources with whole-orbit flight data of FUNcube-1 were found:

- The first source is the AMSAT-UK online data warehouse [60]. This website shows very recent data (update frequency of a few hours) of a single orbit. The data from this source therefore dates 13 August 2024; the website underwent maintenance afterwards. Unfortunately, the data could not be directly downloaded, so an online plot digitiser was used to trace the data points[1].
- The second source is again whole-orbit flight data, but provided via e-mail communication with S. Speretta[2]. This data is from 4 February 2016; since it is much earlier data than the almost real-time 2024 data, the spacecraft's orbital properties are also different. This is discussed more in Section 4.2.4 and Section 4.2.5.

For the purpose of this analysis, and due to a lack of data of the hardware, a pragmatic approach was used for constructing the thermal model. Hence, parts such as the antennas and specific components on the PCBs are not modelled, since there is no such specific temperature data available anyway. However, the batteries are included due to their large mass and strong sensitivity to temperatures.

[1] PlotDigitizer: `https://plotdigitizer.com/`

[2] Acquired via e-mail communication with S. Speretta in August 2024.

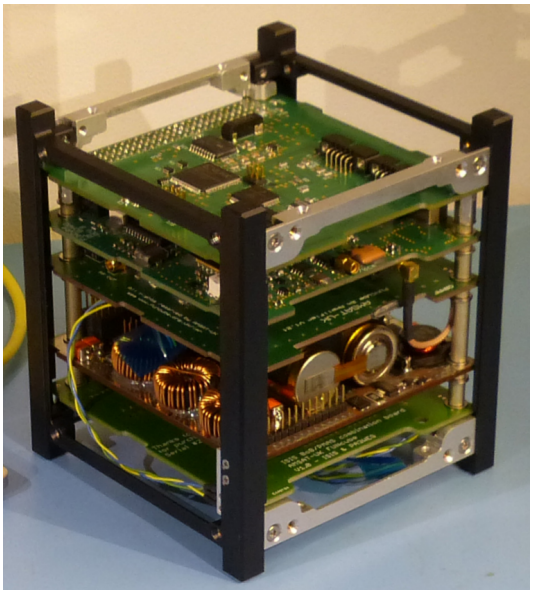


Figure 4.31: Engineering model of FUNcube-1 without the outer panels attached [61].

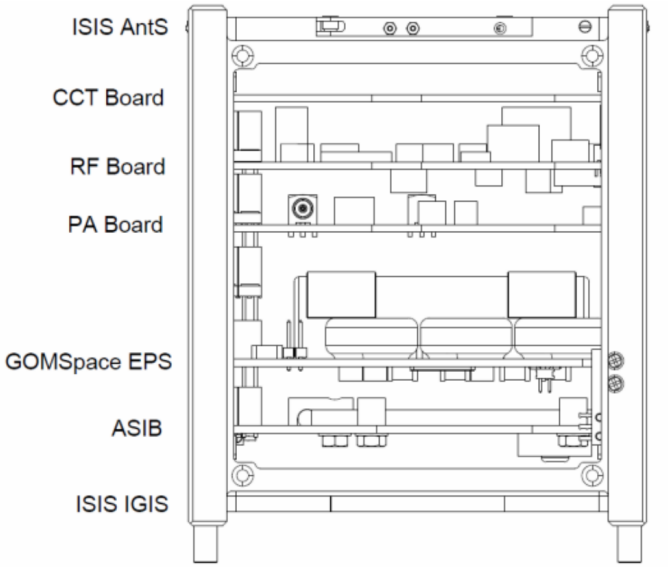


Figure 4.32: FUNcube-1 schematic diagram of the internal layout and the board names [59].

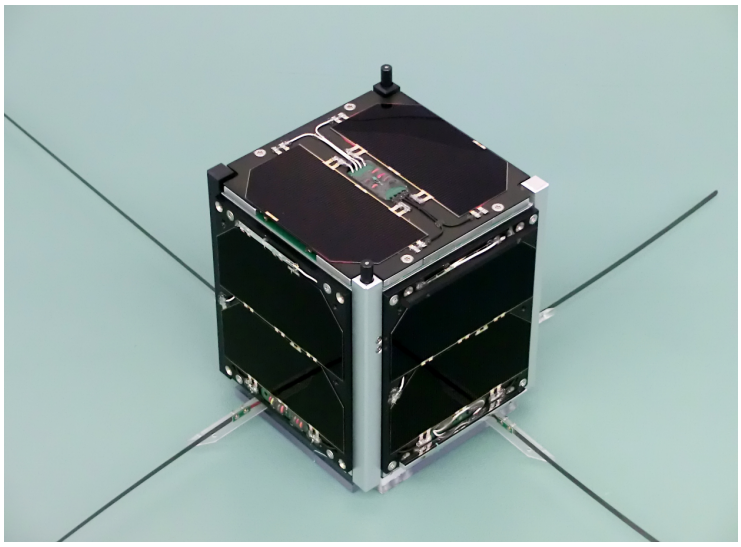


Figure 4.33: FUNcube-1 with the outer panels attached and the antennas deployed [61].

4.2.1. Modelling Approach

As mentioned in the introduction of this section, some elements such as the antennas and specific PCB structures were neglected in this model. The flight data of only a select number of components is known, so it would have no additional benefit to add even more detail to the model. Figure 4.34 shows which parts of the satellite are included in the thermal model. Additionally, Figure 4.35 shows a schematic diagram of the modelling method. Finally, a summary of all components and their properties is shown in Table 4.3.

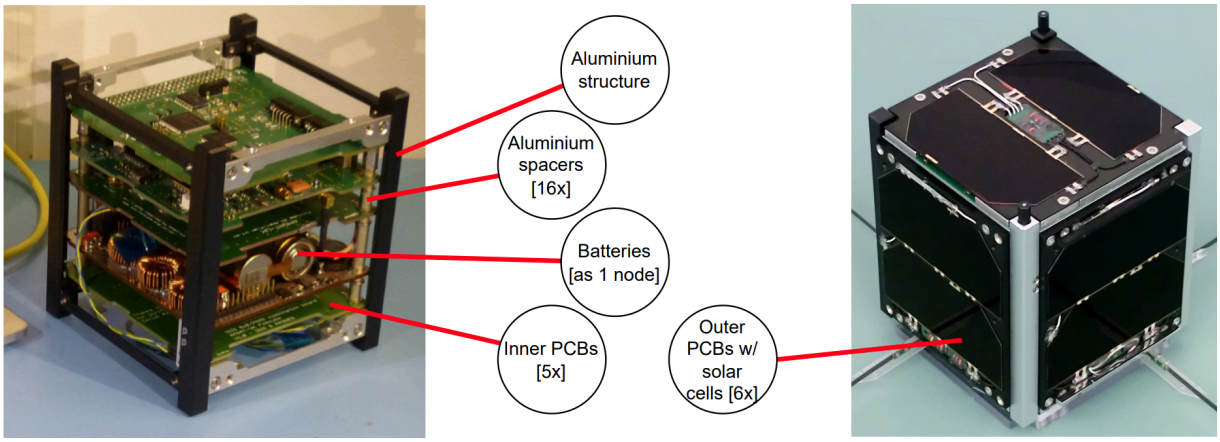


Figure 4.34: FUNcube-1 including text that indicates which parts are included in the thermal model (adapted from [61]).

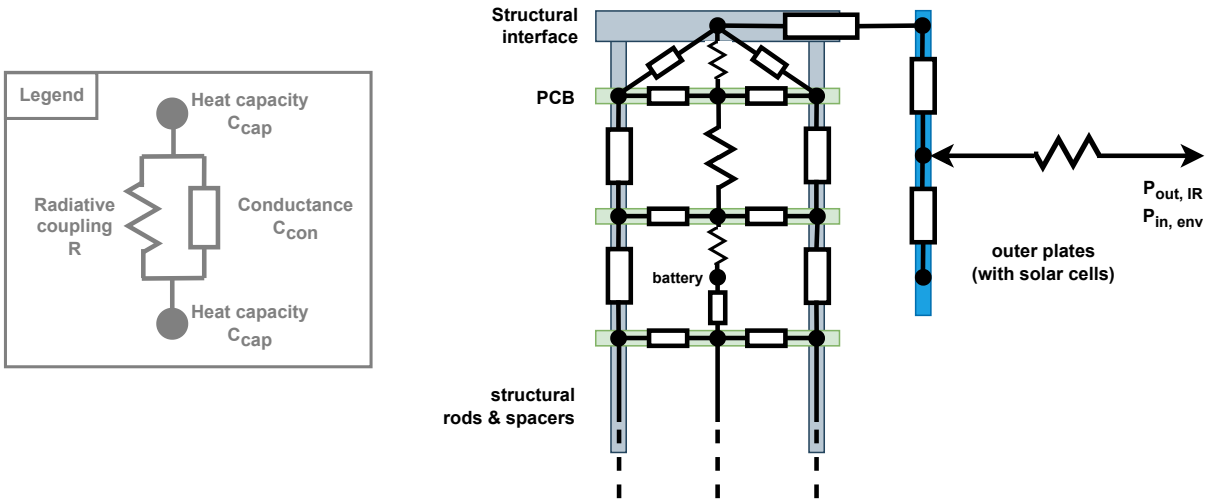


Figure 4.35: Schematic diagram of the method in which FUNcube-1 was modelled. The main building blocks (modules) are the outer plates, the inner PCBs (one includes a battery), the structure, and the rods and spacers.

Looking at Table 4.3, each component contains a number of key parameters. The mass values are estimations based on the available pictures and material properties, and they are also correlated with the flight data. The names under ”material” and ”coating” are the item names from the material database [8], see also Appendix B. Each component is given at least a mass and a material. The material assignment allows the software to convert the mass in kg to a heat capacity in J/K. A coating assignment, or manual assignment of the optical properties, is only needed if the node is either on the outside of the spacecraft (the outer PCBs), or has radiative connections with other nodes (the inner PCBs). The total power of 1.2 W ([3]) is distributed equally over the internal nodes. The battery pack is given the properties of copper for now, but due to the complex chemistry of a battery, it is recommended to either perform a more in-depth literature search, or thermal tests with an actual battery.

The total mass of the model is 1070 grams, which is slightly higher than the approximate 980 grams mentioned by S. Speretta[4]. However, the mass of 1070 grams was found while correlating the results with

[3] See footnote 2

[4] See footnote 2.

the flight data; in a thermal model, it is the heat capacity that influences the system, not the mass itself. Since some assumptions were made, such as lumping the steel rod mass into the aluminium spacers, slight mass discrepancies are likely to occur. The total heat capacity of the spacecraft is 1029.25 J/K.

The thermal connection between all components (except the through-conduction of the PCBs) is assumed to be similar to an aluminium-aluminium contact conduction, defined as ”al_al” in the material database. This assumption relies on the fact that the PCBs have copper rings around the corner holes, resulting in a copper-aluminium contact conduction between the PCBs and the spacers. Since contact conductances are very hard to predict analytically, the ”al_al” connection is used, since it is based on research (see Appendix B; this appendix also includes all sources used), while data for copper-aluminium connections specifically was not found. It is recommended that such thermal contacts are determined experimentally.

Table 4.3: All components in the FUNcube-1 thermal model. The masses are estimated from Figure 4.34, and the heat capacities are computed using the material database [8]. Each PCB is split into nodes A-E, as further described in Section 4.2.2. The coatings and their corresponding properties are stored in the material database [8], and are also shown in Appendix B. Non-radiative nodes do not need to have a coating assigned.

Name		**Mass [g]**	**Heat Capacity [J/K]**	**Material**	**Coating**
Outer PCBs [6x]		6x 20	6x 22	”PCB”	”solar_cell_mix _black_paint” ($\alpha = 0.75$, $\varepsilon = 0.88$)
Inner PCBs [5x]		5x 20	5x 22	”PCB”	”PCB_green” ($\alpha = 0.88$, $\varepsilon = 0.70$)
Battery pack		50	19.25	”copper”	-
Spacers [16x]		16x 20	16x 19.2	”al7075”	-
Structure	Upper & lower rings [2x]	2x 80	2x 76.8	”al7075”	-
	Corner rods [4x]	4x 80	4x 76.8	”al7075”	-
Total		**1070**	**1029.25**		

Table 4.4: FUNcube-1 PCB power distribution. The numbers are from S. Speretta (see footnote 2), except the CCT board power that was found in the FUNcube-1 handbook [59].

Board name (& name in Python)	Power [mW]
AntS (z+)	40
CCT (5)	15
RF (4)	190
PA (3)	303
EPS (2)	160
ASIB (1)	462
Total	**1170**

4.2.2. Defining the PCBs

As explained in the user's manual in Appendix C, there are two standardised ways to define a PCB: a 5-node PCB or a 9-node PCB. These two types can be easily defined using a custom Python function, but all parameters can be adjusted to the user's liking. For example, extra nodes can be added to the PCB, such as a battery.

A 5-node PCB was chosen for all FUNcube PCBs, since it is an easier model to compute (fewer nodes), while still representing the four rods and spacers on the corners of all PCBs (see Figure 4.31). The PCB model is shown in Figure 4.36. Details on how the masses, distances, areas, conductances etc. were calculated can be found in Section C.3.2.

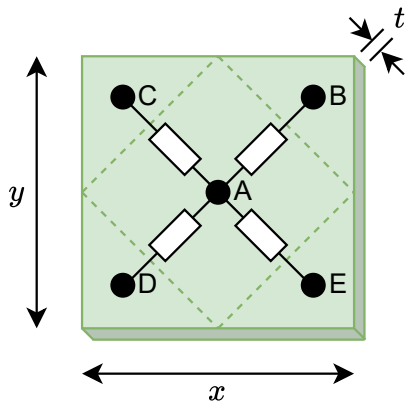


Figure 4.36: Schematic diagram of the template for a 5-node PCB, with a main central node (node A) and a node near each corner (nodes B-E).

4.2.3. Defining the Outer Surfaces

The outer surfaces are also defined as PCBs, as explained in the previous sub-section. However, since all outer surfaces have solar cells mounted on them (see Figure 4.33), their optical properties require dedicated calculations.

Figure 4.33 shows that the majority of FUNcube-1's outer surface is covered by the solar cells, but a reasonable fraction is the exposed panel covered in black paint. The optical properties from the black

paint are taken from the material database [8]: $\varepsilon_{blackpaint} = 0.85$ and $\alpha_{blackpaint} = 0.95$.

The solar cells convert part of the solar energy into electrical energy. This lowers the ”effective” absorptivity of the surface. A method from T. van Boxtel [38] was adopted, where the solar cell efficiency was used to compute α_{eff}:

$$\alpha_{eff} = \alpha_{solarcell} - \eta_{mppt} \cdot \eta_{conversion} = 0.91 - 0.9 \cdot 0.28 = 0.66 \tag{4.1}$$

where η_{mppt} is the efficiency of the Maximum Power Point Tracker (MPPT), and $\eta_{conversion}$ is the efficiency of converting the solar energy into electrical energy. The emissivity is independent of the solar cell efficiency, and is $\varepsilon_{solarcell} = 0.89$ The values were also taken from Boxtel, but may vary depending on the type of solar cell used and the environmental conditions. Finally, the absorptivity that is given to the outer surfaces is a combination between the black paint and the solar cells. The solar cells cover approximately 2/3 of the surface, while the black paint covers 1/3. Hence, $\alpha_{outerpanel} \approx 0.75$ and $\varepsilon_{outerpanel} \approx 0.88$.

4.2.4. Orbital Model

The orbital model is challenging to replicate, mainly due to uncertainties in attitude and/or rotational rates. According to the FUNcube handbook [59], the attitude is passively stabilised by means of two hysteresis rods that dampen the angular rates in two axes, and by means of a permanent magnet that aligns itself with the Earth’s magnetic field. The effect of the permanent magnet on the spacecraft’s attitude is illustrated in Figure 4.37. On the contrary, the attitude in the Python software is as shown in Figure 4.38. These figures do not show any possible spin of the spacecraft.

Since the Python software does not yet allow for such complicated attitude behaviour, simplifications must be made. For the 2024 data, it is assumed that the spacecraft’s attitude is fully stabilised, as in the coordinate system in Figure 3.7. For the 2016 data, the spacecraft is assumed to be stabilised with its z+ face (see Figure 3.7) toward its velocity vector, but spinning with 2°/s about its z-axis. This value was determined by comparing the simulation output with the flight data. This will become clear in the next sub-section.

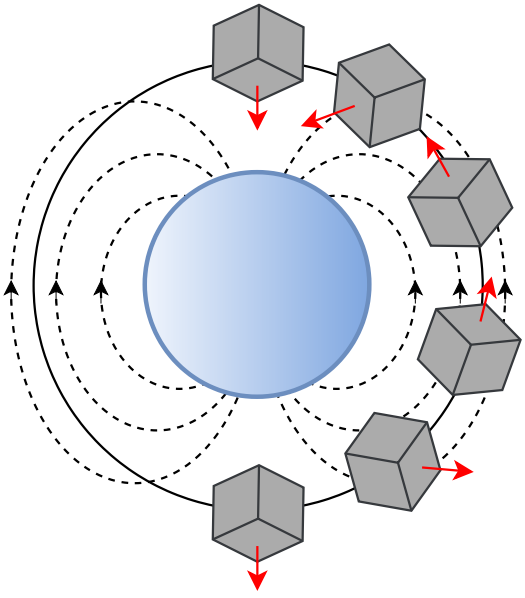


Figure 4.37: Spacecraft attitude throughout an orbit when one axis is aligned with the Earth’s magnetic field. This represents FUNcube-1’s actual orbit.

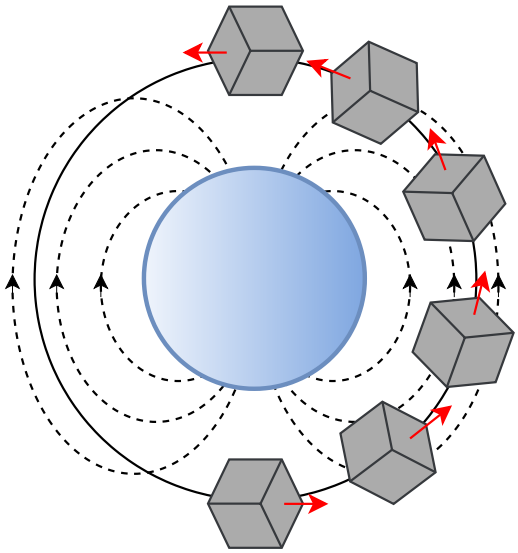


Figure 4.38: Spacecraft attitude throughout an orbit when the attitude follows the velocity vector, and is unaffected by the Earth’s magnetic field. This represents FUNcube-1’s orbit as simulated with Python.

Since the flight data was taken from the frequently updated AMSAT-UK data warehouse [60], the orbital data should also be as recent as possible. The N2YO.com website [62] shows real-time orbital parameters of the satellite. An example screenshot of the real-time orbital position is shown in Figure 4.39. The beta angle can also be found from this figure, by comparing the spacecraft's position with the Sun icon as it passes over the equator. The difference in longitude is approximately the beta angle.

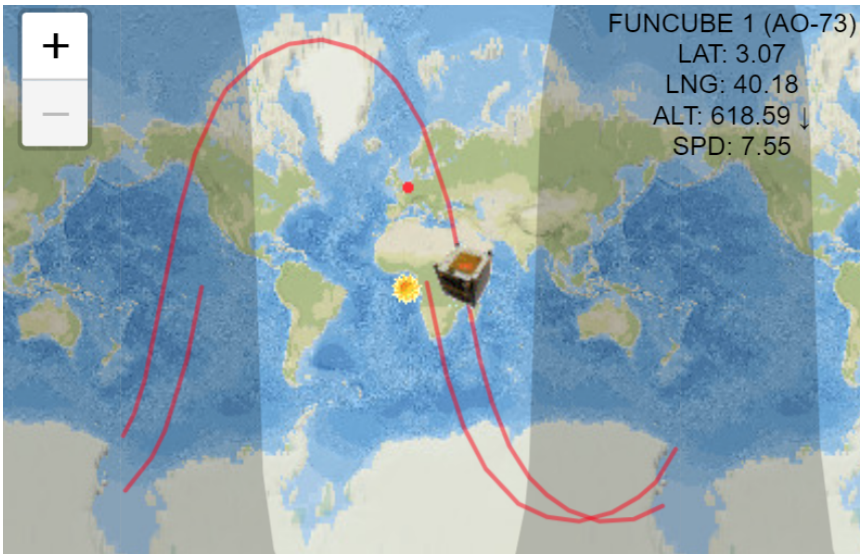


Figure 4.39: Orbit path and real-time orbital position of FUNcube-1, shown on N2YO.com [62].

A summary of all orbital data and attitude data is shown in Table 4.5. As explained before, the angular rates are not perfectly be zero, because the spacecraft rotates along with the Earth's magnetic field. This will result in some discrepancies in the final validation results.

Table 4.5: All orbital/attitude data of FUNcube-1 entered into the Python software.

Name	Value & Unit	Source
	2024 data	
Altitude	596 km	Actual orbit is 629x563 km on 13 August 2024 [62]
Beta angle	30°	Estimated from Figure 4.39
Day	225	13 August is day 225
Angular rates	0°/s	Attitude is passively stabilised [59]
	2016 data	
Altitude	644 km	Computed using the orbital period seen in the 2016 flight data (consists of 14 orbits)
Beta angle	30°	Approximately constant in Sun-Synchronous orbit; similar as in 2024
Day	35	4 February is day 35
Angular rates	2°/s in z+	Correlating with the flight data suggests 2°/s along the z+ axis

4.2.5. Validation Results

In this sub-section, the results from the Python simulation are compared with the flight data from 2024 and 2016. The data set from 2024 is presented first, since it appears that the spacecraft is not tumbling quickly, while the data from 2016 suggests that the spacecraft was spinning at a much faster rate at that time.

2024 Flight Data

The simulation is compared with the 2024 data in Figure 4.40, and the errors are displayed in Figure 4.41. Only the temperatures of the x+, x-, y+, and y- panels were available, aside from a number of chassis temperatures that would be difficult to validate with the simplified Python model. The coordinate system for an orbiting spacecraft used in this Python code (Figure 3.7) is not necessarily equal to that of FUNcube-1. By comparing the data in Figure 4.40, it is likely that the coldest temperatures (green data) belong to the y+ face of the Python coordinate system. Simultaneously, the hottest temperatures appear to belong to the y- face; it makes sense that the two opposing faces have opposing temperature levels as well. The x+ and x- faces are not completely enshadowed or illuminated; hence, they have more moderate temperatures.

As seen in Figure 4.40 and Figure 4.41, there is a difference in the shape of the temperature curves. The largest difference appears to be right before the eclipse, near the fortieth minute, and right after the eclipse, near the ninetieth minute. This may be due to (at least) two effects:

1. One axis (z+) is oriented differently throughout the orbit: it follows the magnetic field in the flight data, as in Figure 4.37, but the simulation assumes that it follows the velocity vector, as in Figure 4.38. This discrepancy is the strongest near the poles, as seen in the two aforementioned figures, while the attitude near the equator is roughly equal for both cases.
2. The other axes (x and y) are oriented differently. In the Python simulation, x+ faces perfectly zenith, z+ towards the velocity vector, and y+ completes the right-handed system, as in Figure 3.7. However, the actual FUNcube-1 may be angled differently, e.g. angled 45°around the z-axis. In that case, the temperature profile would look different than in the simulation, since all faces would receive different amounts of environmental radiation compared to the simulated attitude. More software features would have to be implemented to allow for more control over the attitude of the spacecraft; it is recommended to read Chapter 7.

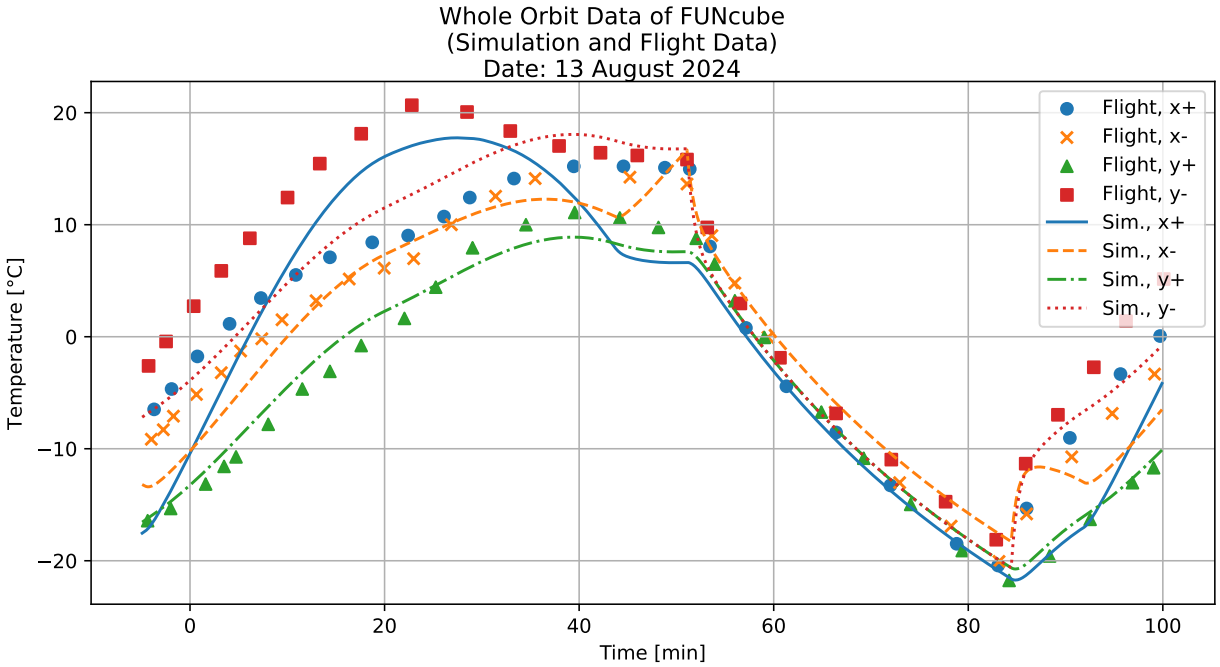


Figure 4.40: Flight data temperatures of four outer panels of FUNcube-1, from the live broadcast AMSAT-UK data warehouse [60], 2024. The eclipse region is between 51-84 minutes.

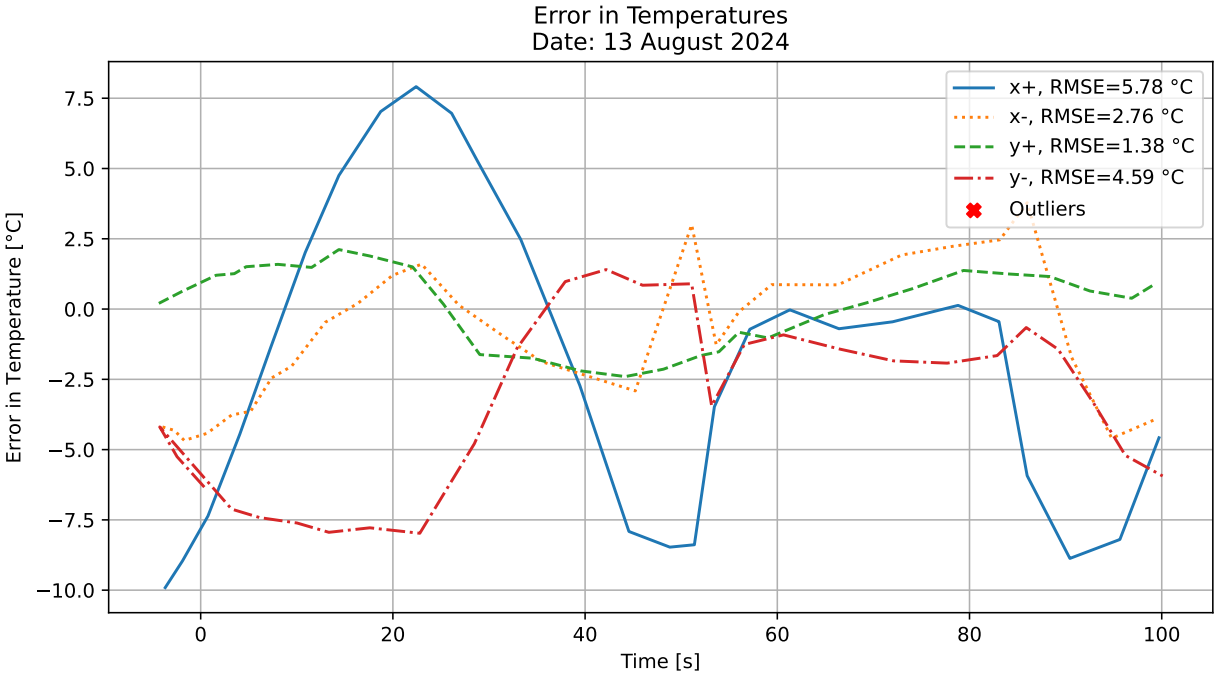


Figure 4.41: Errors in temperatures of four outer panels of FUNcube-1, from the live broadcast AMSAT-UK data warehouse [60], 2024. There are no outliers present.

2016 Flight Data

The data from 2016 is compared with a Python simulation in Figure 4.42 and Figure 4.43. The data looks vastly different from the 2024 data. The eclipse period, from the 65th minute onward, looks similar as the 2024 data. However, the rest of the data shows clear oscillatory behaviour. The only logical explanation for this is the fact that the spacecraft is spinning.

The spin rate of 2°/s about the z+ axis was determined looking at the frequency of oscillation, and the amplitude. Combining these two arguments results in a solid suggestion that the rotational rate was indeed $\approx$ 2°/s.

- The oscillation frequency in the flight data appears to be 21 rotations in 63 minutes, which is 1/3 rotation per minute. This translates to 2°/s. However, the sampling rate is in the same order of magnitude, so it is possible that the actual frequency is higher, but not captured in the sampled data.
- The second criterion to observe is the amplitude of the oscillation. For a slowly tumbling spacecraft, the illuminated face becomes very hot, and the enshadowed face becomes very cold: the temperature extremes are large. For a more quickly rotating spacecraft, the faces do not have as much time to heat up or cool down, so the temperature swing is much smaller. Assigning a rotational rate of 2°/s to the Python simulation resulted in a temperature amplitude of approximately 5-8°C outside of the eclipse. Any faster oscillation makes the temperature swing smaller, so it is unlikely that those were really the rotational rates of FUNcube-1 in orbit.

Looking at Figure 4.42 and Figure 4.43, there are not as many severe temperature drops or increases as there were in the 2024 data; the overall heating trend in the out-of-eclipse period follows a relatively predictable path. However, the simulation especially shows discrepancies right before entering the eclipse (minute 50) and after leaving the eclipse (minute 0). This could be due to the same reasoning as in argument 1 of the 2024 data: the real spacecraft follows the Earth's magnetic field, while the simulation follows the velocity vector.

The temperatures of the simulation are also overall colder than the flight temperatures. It was observed that the simulation always shows a decrease in temperature when the spacecraft is tumbling, compared to

when it is not tumbling and only rotating along with the velocity vector. This is discussed in more detail in Chapter 5, using Figure 5.11. Depending on the exact rotational axis of the spacecraft, the ”effective area” receiving sunlight might differ. This could explain a different average temperature.

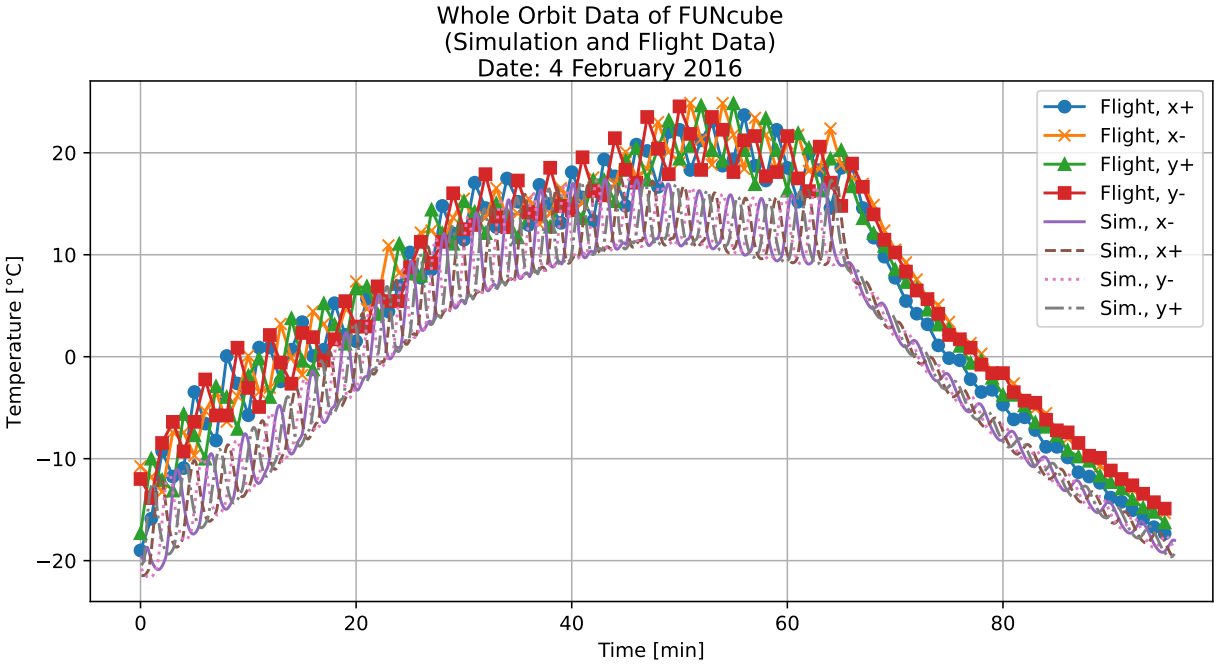


Figure 4.42: Flight data temperatures of four outer panels of FUNcube-1, from 2016 data provided by S. Speretta. The spacecraft is rotating at approximately 2°/s. The eclipse region is between 65-96 minutes.

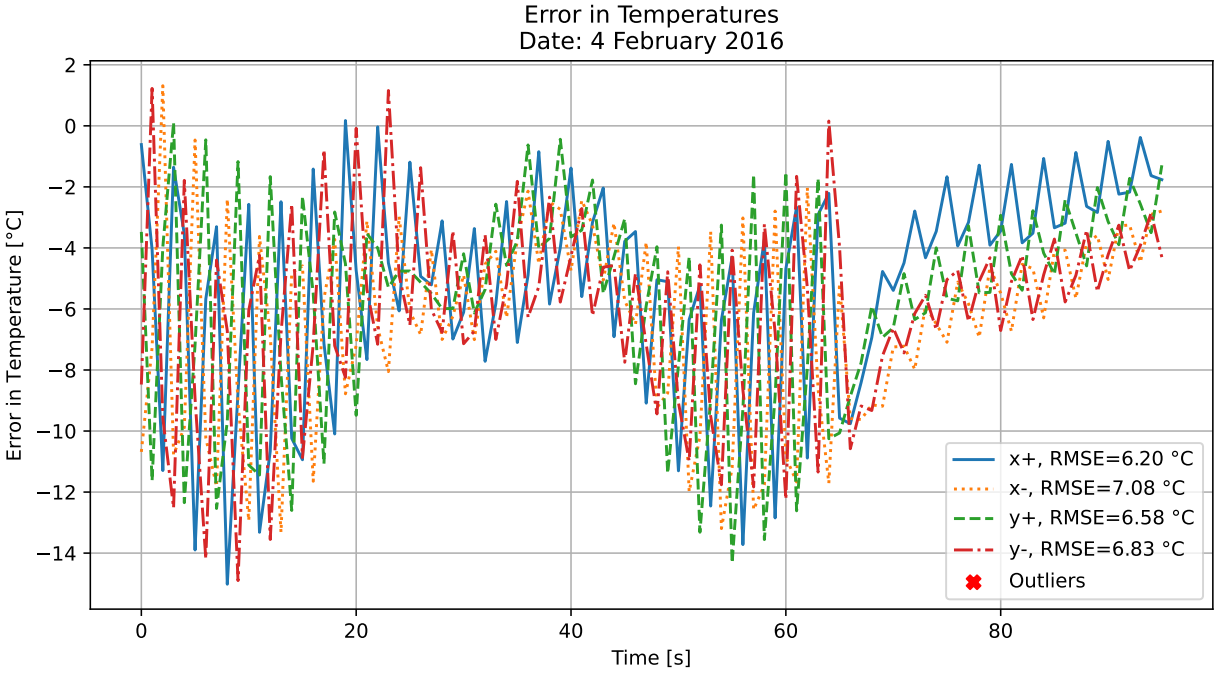


Figure 4.43: Errors in temperatures of four outer panels of FUNcube-1, from 2016 data provided by S. Speretta. The spacecraft is rotating at approximately 2°/s. There are no outliers present.

Summary and Conclusion
The root-mean-square errors (RMSE) were already displayed in Figure 4.41 and Figure 4.43, but are also tabulated below in Table 4.6. The case from 2024 shows lower errors than the 2016 case. In both cases, it is expected that a large part of the uncertainty is due to the attitude and rotational rates. When the software is extended with more attitude controllability, this hypothesis can be investigated more thoroughly.

Table 4.6: RMSE values for the validation cases of FUNcube-1 in 2024 and 2016.

Spacecraft Face	2024 RMSE [°C]	2016 RMSE [°C]
x+	5.78	6.20
x-	2.76	7.08
y+	1.38	6.58
y-	4.59	6.83

These validated temperatures are only the outer panels of the spacecraft. Flight data on, e.g., the battery temperatures was not available. While the outer panels experience extreme temperature changes, the inside of the spacecraft remains more stable, since the thermal resistance between the inside and outside act as dampers. To know the temperatures of the vital internal components, it is advised to perform a sensitivity analysis specifically on those components; this is thoroughly explained in Chapter 5. It will be clear that the internals of the spacecraft are much less sensitive to the environmental heat fluxes than the outer panels.

Finally, having access to actual hardware, having more orbital/attitude information, and possibly performing thermal tests would also increase the accuracy of the simulation. Most of all, it would narrow down the possible causes for wrongly estimated temperatures. To conclude, the simulation errors are most likely due to modelling errors; not due to the inaccuracy of the simulation, as was already shown in the ESATAN comparison at the start of this chapter.

5

Sensitivity Analysis

For all types of theoretical analysis, the sensitivity of the outputs to certain input parameters is of high importance. Not only does it highlight which outputs are most likely to be (in)accurate; it also shows which inputs could be considered more influential than others. The sensitivity analysis included in this software has two purposes: making general order-of-magnitude predictions on the accuracy of the software; and allowing the user to perform a sensitivity analysis specifically on their own model.

This chapter explains the sensitivity analysis tool that is part of the thermal analysis software. The first part of this chapter will elaborate on the basic principles behind this sensitivity analysis, and shows plots of some arbitrary simulations (these plots are just used to show how to read them). The second part will show the results of performing a sensitivity analysis on FUNcube-1, and it will show the general trends of all the sensitivity plots. Finally, the third part will use the sensitivity tool to assess the assumptions made in the calculations, and it will show realistically what order of magnitude all sensitivities will be.

5.1. Types of Sensitivity Analysis

Sensitivity analyses can be visualised in many different ways, since there is always a very large data set involved. The software splits up the sensitivity analysis in the following categories (all ”sensitivities” refer to the effect on spacecraft temperatures; more details follow in the rest of this chapter):

1. **All-encompassing plots**, showing the sensitivity of all parameters in essentially a single plot. These figures are useful to understand the orders of magnitude associated with certain input parameters. Such a plot can show you, for example, whether the altitude or the internal power has a larger effect on the spacecraft’s temperature profile.
2. **Variable-specific plots**, showing the sensitivity of a single parameter in high detail. Such a plot cannot give a quick overview of all the different parameters, but it can give you more insight in the behaviour of a single parameter.
3. **Internal radiation plots**, providing similar information as the variable-specific plot Figure 5.7. Since there is only an ’on’ and ’off’ option, all can be plotted in a single figure.
4. **Time step & angular rates plots**: showing similar information as Figure 5.3, but the peculiarity is that the time step and angular rates are interrelated; if they are not compatible, numerical effects such as aliasing may occur.

All of these analysis types rely on the same principle, where a baseline case is entered, and subsequently, a single input parameter is changed, and the case is run again. The baseline case is the unchanged nodal model (and orbital model) that the user entered. The baseline should be the most likely scenario to occur; it should be as close to reality as possible. Then, by varying all input parameters, it can be seen how stable that baseline is when the inputs are changed.

Furthermore, not all parameters can be given absolute values for the sensitivity analysis. For example, the conductance varies all throughout the model, so a more logical way to perform the sensitivity analysis is to apply a multiplication factor to the existing values. Therefore, this is done for all parameters except the beta angle, day in the year, and time step (percentages do not make much sense for those parameters).

5.1.1. All-Encompassing Plots

The all-encompassing plots provide a complete sensitivity analysis of the whole simulation, varying (almost) all input parameters by two user-specified percentages (for example, ±X-Y%), compared to a pre-defined baseline. Two example graphs are shown in Figure 5.1 and Figure 5.2. Figure 5.1 shows the sensitivity of all input parameters on the average spacecraft temperature, while Figure 5.2 shows the effect on the difference between maximum and minimum temperature throughout the orbit. The two figures (average temperature and temperature swing) are always shown together, since they show the same data set, but with a different output parameter. The all-encompassing plot should be interpreted as follows:

- The abbreviations used are: ABS (absorptivity); ALB (albedo heat flux); ALT (altitude); CAP (heat capacity); CON (conductance); EMI (emissivity); IR (Earth infrared heat flux); POW (internal power); SOL (solar heat flux). Some input parameters are missing, which is explained in Section 5.1.2.
- The baseline case (as explained in the introduction of Section 5.1) is taken as the **relative** zero-value (dashed black line) for both of the plots shown below. The **actual** baseline value is shown in the figure's title (e.g., the average temperature of the baseline case is 3°C, but the graph itself shows the **relative deviation** from this 3°C when the inputs are changed).
- The blue and orange lines indicate the **relative change** that occurs to the average temperature (or temperature swing, for Figure 5.2), when the input parameter is changed by ±X-Y%. In this case, blue represents a 10% change and orange a 20% change. The triangular-shaped arrows indicate the direction of a positive input change (+10 or +20% in this figure). For example, when the emissivity (EMS) is changed by +10 or +20%, the average temperature (or temperature swing in Figure 5.2) decreases, because the arrow points down.

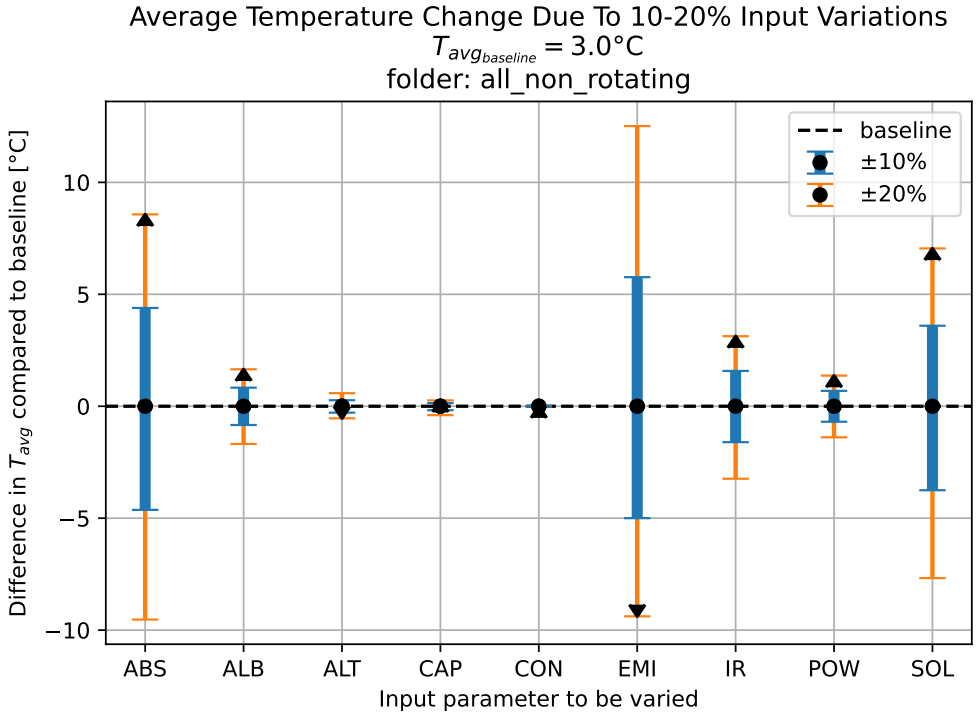


Figure 5.1: Sensitivity of a non-rotating spacecraft's average temperature on various input parameters.

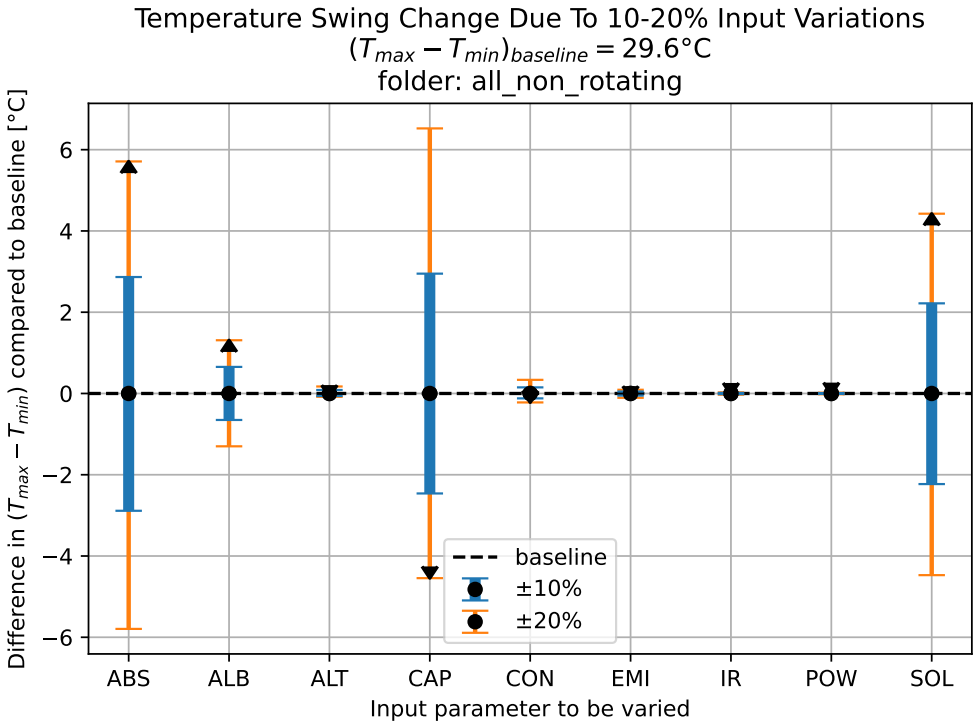


Figure 5.2: Sensitivity of a non-rotating spacecraft's temperature envelope on various input parameters.

An advantage of this analysis is that all input parameters can be easily compared in a single figure. For this example case, the emissivity and absorptivity have a strong impact on the average temperature, see Figure 5.1. However, the emissivity does not have a significant impact on the temperature swing, see Figure 5.2. A disadvantage of this plot is that it only shows the baseline case, and four percentage changes (-Y%, -X%, +X%, +Y%). More information can be extracted by running more cases with more values for each input parameter. That is done with the variable-specific plots.

5.1.2. Variable-Specific Plots

The all-encompassing plots show only five values per input variable (-Y%, -X%, baseline, +X%, +Y%), and furthermore, the transient behaviour throughout the orbit is not seen in the plots such as Figure 5.1 and Figure 5.2. Therefore, different plots are created as well: the variable-specific plots. The variable-specific plots in this section show more information of a specific input variable. This may reveal details such as the type of trend when the variable is changed (exponential, linear, logarithmic). Furthermore, it can improve the engineer's understanding of what happens when the parameter is changed, as opposed to the few numbers presented in the all-encompassing plots. There are two types of variable-specific plots (the following examples are for the solar absorptivity α):

- **Temperature variance (y-axis) versus parameter value (x-axis)** (see Figure 5.3): for this plot, a number of cases are ran with different values (multiplication factors) of the desired input parameter. This can be a list of arbitrarily many numbers. Unfortunately, the absolute values of the parameter that is changed cannot be plotted, since they may vary throughout the model. One should look in the code where the model is defined, to find the original (baseline) values.
  The plot shows the minimum, mean, and maximum temperatures within the entire model throughout the orbit for each case. Hence, time-varying information is still compressed into single values, similar to the all-encompassing plots. However, due to the arbitrary point density, any trends can be seen more clearly. For the absorptivity plot, see Figure 5.3, all temperatures increase logarithmically as the absorptivity is increased. Moreover, the difference between the maximum and minimum achieved temperature also increases as the absorptivity is increased. This makes sense, since a higher absorptivity results in more solar heat absorption, and hence a stronger difference between illuminated and eclipse conditions. For a beta angle where the orbit is always illuminated, there is no eclipse, and hence the aforementioned trends might differ.

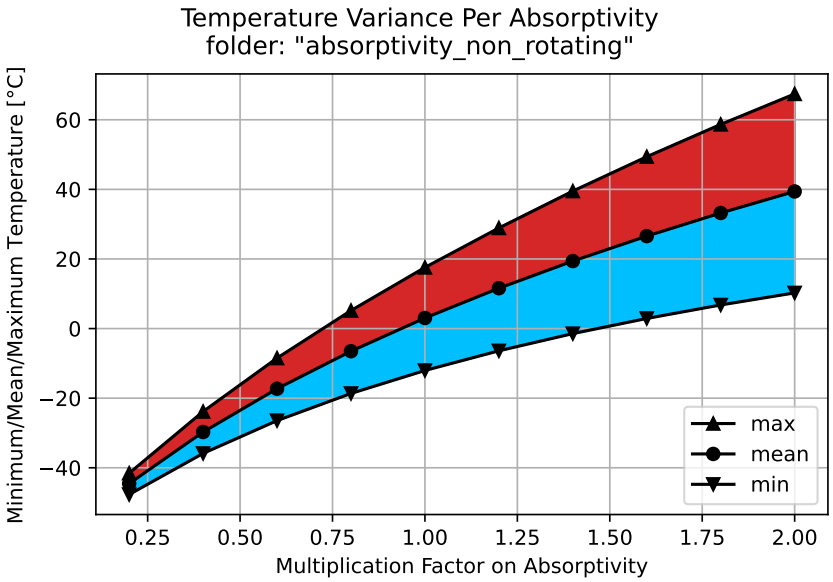


Figure 5.3: Variable-specific plot example for the absorptivity. It shows the minimum, mean, and maximum temperature of the spacecraft versus the multiplication factor on the absorptivity. A too high multiplication factor could result in $\alpha > 1$, which is impossible.

- **Temperatures or heat fluxes (y-axis) versus time (x-axis)** (see Figure 5.4, Figure 5.5, Figure 5.6, Figure 5.7): these types of plots are the most detailed of all the sensitivity plots; they show the transient temperatures or heat fluxes for all values of the input parameter (in this case the absorptivity α). Figure 5.4, Figure 5.5, and Figure 5.6 show the solar heat, albedo heat, and Earth IR heat, respectively. Each line is the temperature for a certain value of α, of a certain node. This node can be chosen by the user, and is "x+" (or "x-" for the albedo and Earth IR plots) in the cases below. This is also indicated in the title of the figures. Depending on the chosen node, the figures will look different. The x+ node is facing zenith (see Figure 3.7), hence, it does not receive any albedo or Earth IR heat. Therefore, Figure 5.4 and Figure 5.5 show the nadir-facing x- face instead.

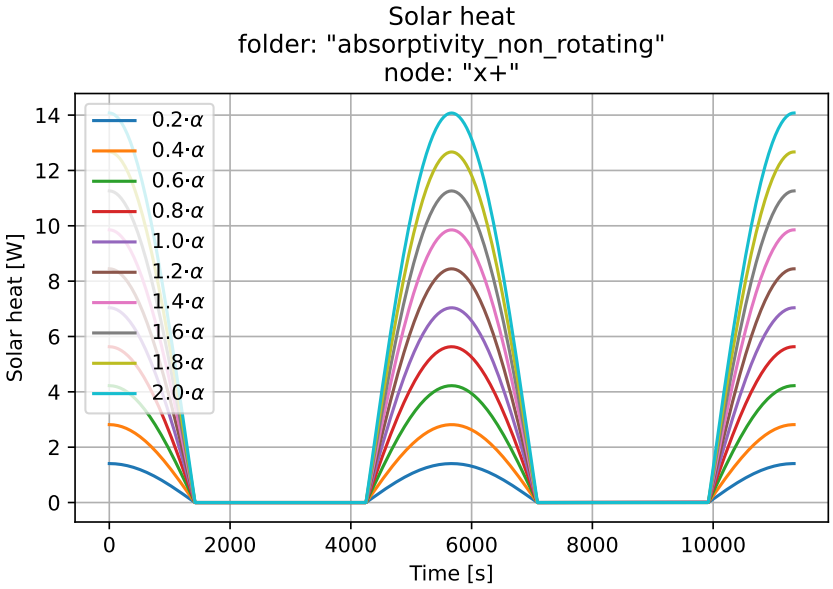


Figure 5.4: Variable-specific plot example for the absorptivity. It shows the solar heat on a selected node (here: 'x+') of the spacecraft versus time, for different multiplication factors on the absorptivity.

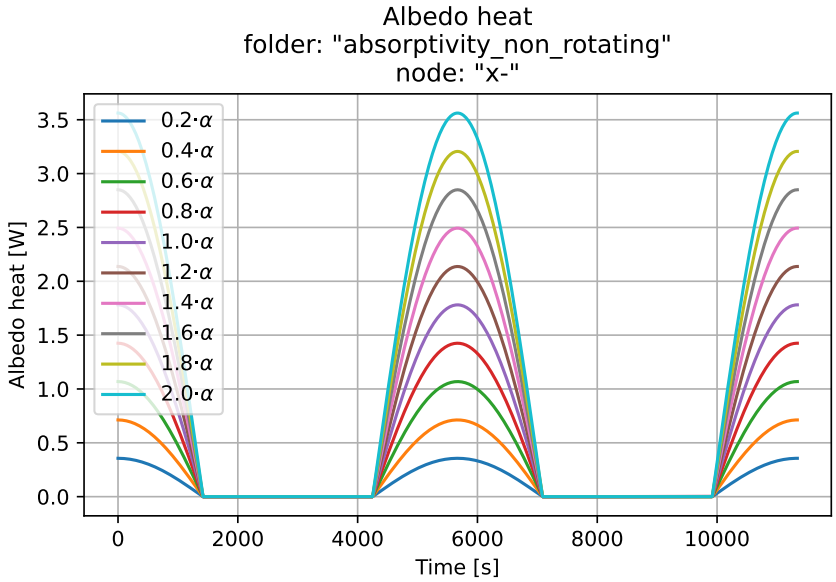


Figure 5.5: Variable-specific plot example for the absorptivity. It shows the albedo heat on a selected node (here: 'x-') of the spacecraft versus time, for different multiplication factors on the absorptivity. X- faces nadir, as opposed to the x+ face (zenith) that does not receive any albedo.

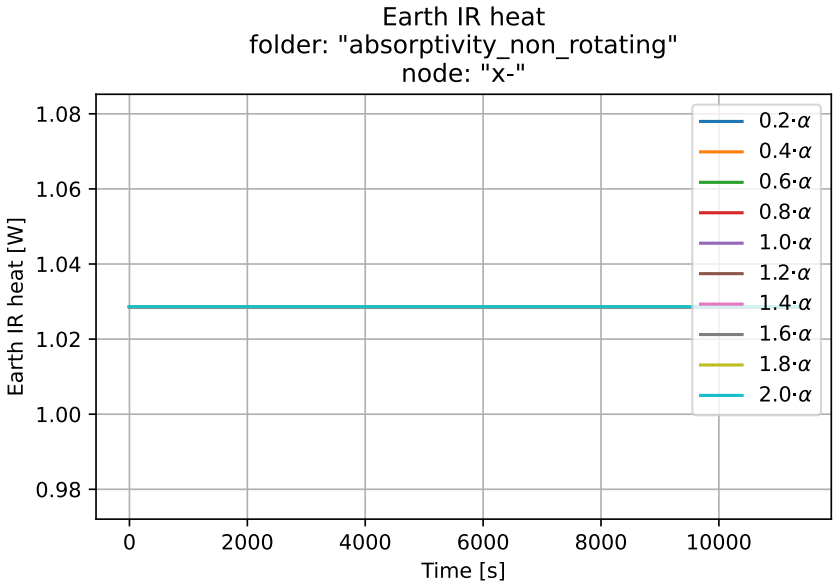


Figure 5.6: Variable-specific plot example for the absorptivity. It shows the Earth infrared heat on a selected node (here: 'x-') of the spacecraft versus time, for different multiplication factors on the absorptivity. The Earth IR heat is independent of α, since α determines the absorbed heat only in the visible spectrum.

Figure 5.7 shows the temperature of the selected node for all multiplication values of α. It is essentially a more complete version of Figure 5.3, showing the time-aspect as well. It can be seen that the temperature indeed increases as α increases, and the max-min difference also increases. However, be aware that the values in Figure 5.7 are not exactly equal to those in Figure 5.3, since the latter contains minimum, mean, and maximum values of the entire nodal modal, not just one node.

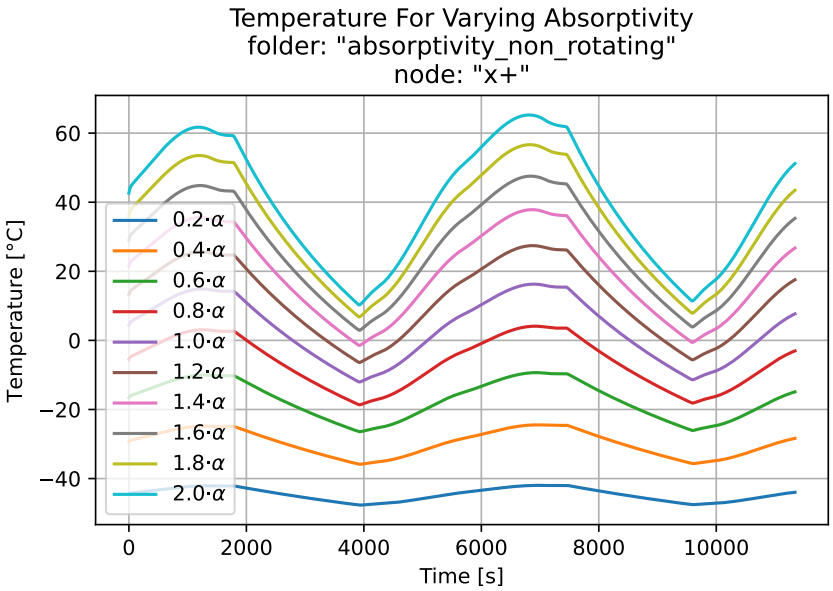


Figure 5.7: Variable-specific plot example for the absorptivity. It shows the temperature of a selected node (here: 'x+') of the spacecraft versus time, for different multiplication factors on the absorptivity.

5.1.3. Internal Radiation Plots

Internal radiation plots simply show the different behaviour between models with, and without internal radiation. It is only possible to execute such an analysis if the original model has all the radiative connections defined. For the sensitivity analysis, all of these are then set to zero.

Figure 5.8 shows the effect of internal radiation when the spacecraft is not rotating (constant attitude with respect to the velocity vector), and Figure 5.9 shows when it is rotating about all axes. Extra plots are available to see the exact difference values (see Section 5.2.2). Evidently, internal radiation plays a larger role for attitude-stabilised spacecraft, since the thermal gradients from one side (strongly illuminated) of the spacecraft to the other (less illuminated) are much stronger than when the spacecraft is rotating. Furthermore, the thermal gradients along the spacecraft are also larger when the components are less well thermally connected; hence, the smaller the conductive couplings are (more resistance), the larger the effect of internal radiation would be.

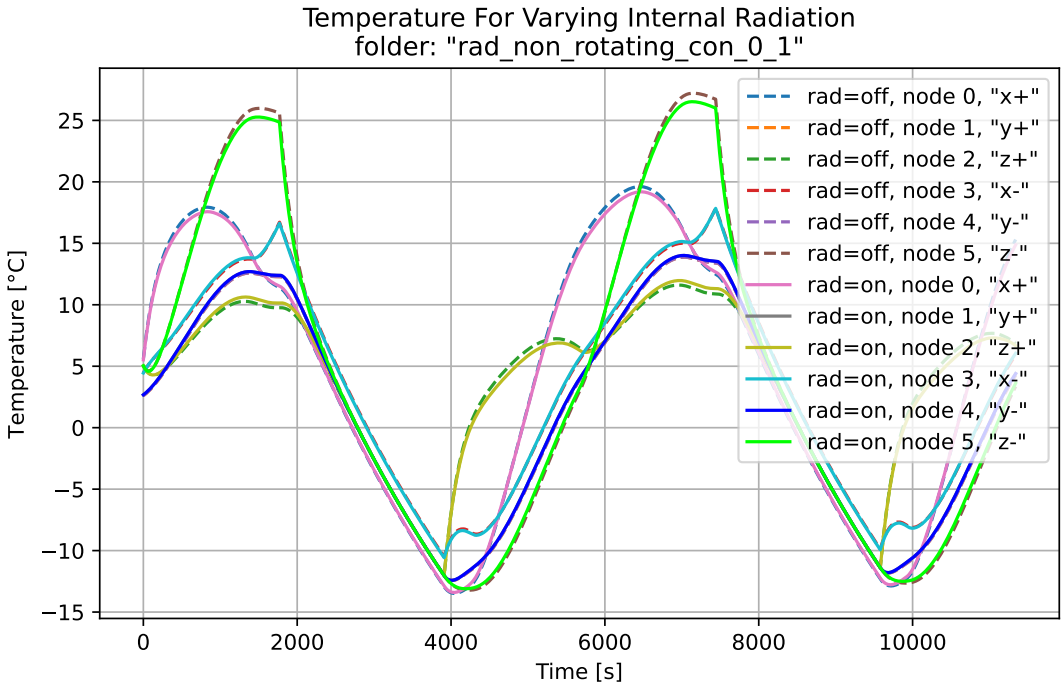


Figure 5.8: Internal radiation plot for a non-rotating spacecraft.

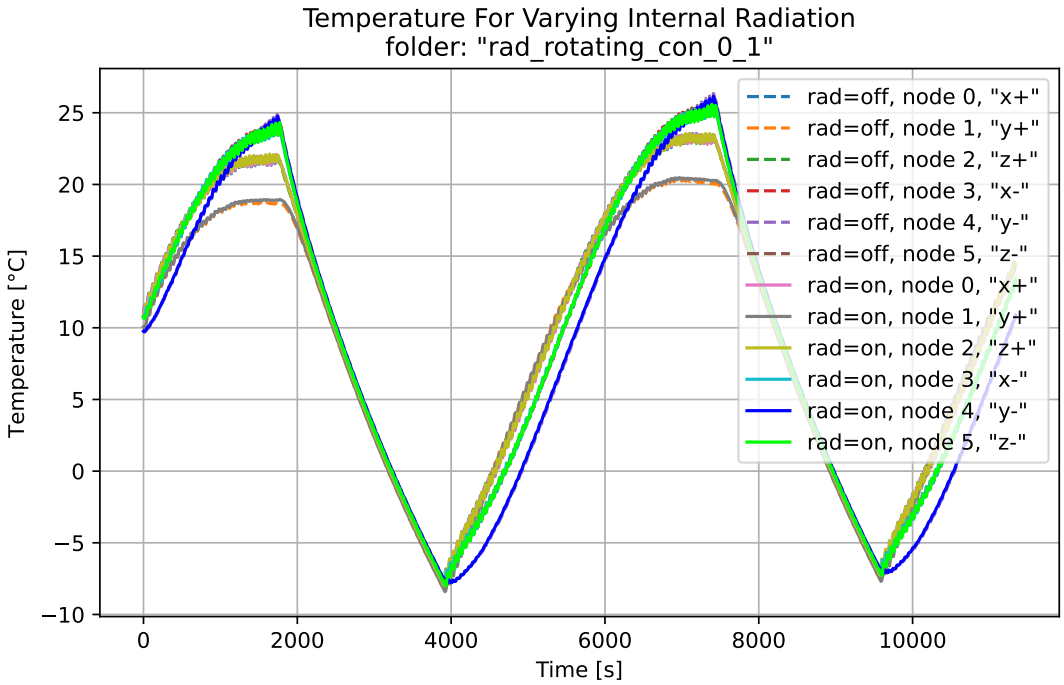


Figure 5.9: Internal radiation plot for a rotating spacecraft.

5.1.4. Time Step & Angular Rate Plots

When the spacecraft is given a nonzero angular velocity, the choice of time step becomes very important. The time step should be small enough such that the rotating motion of the spacecraft is sufficiently resolved. If the time step is too large, the simulation will become unpredictable and unreliable.

An example of this numerical behaviour is shown in Figure 5.10. It shows an example of a temperature versus time step plot generated by the `run_dt_rot` and `plot_dt_rot` functions. The time step should ideally not affect the simulation outputs (i.e., the temperatures), but it does. The temperatures drop as the time step reaches integer multiples of 9 seconds. This is because the angular velocity in Figure 5.10 is 10 deg/s, and therefore, a time step of 9 seconds results in a 90 degree rotation of the spacecraft, per time step. As a result, the simulation does not perceive the spacecraft rotating gradually, but it perceives an instant 90 degree rotation. Similarly, for time steps of 18, 27, and 36 seconds, the simulation perceives instant rotations of 180, 270, and 360 degrees per time step. The point where the time step size results in the spacecraft rotating 90 degrees per time step, is hereby referred to as the ”critical time step”:

$$dt_{cr} = \frac{90}{\omega} \tag{5.1}$$

These equations give an idea of the time step needed to resolve the rotating motion of the spacecraft. The time step chosen for the simulation should not be near this value, but significantly lower, as the time step sizes nearby the critical time step are also unreliable.

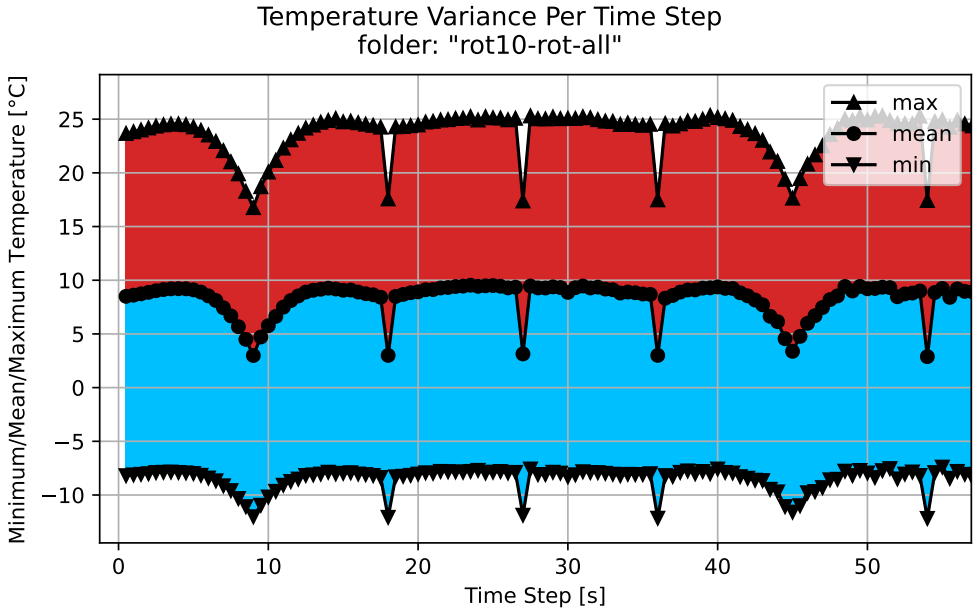


Figure 5.10: Minimum/mean/maximum temperature versus time step, where all axes of the spacecraft have an angular velocity of 10 deg/s (mentioned above the plot).

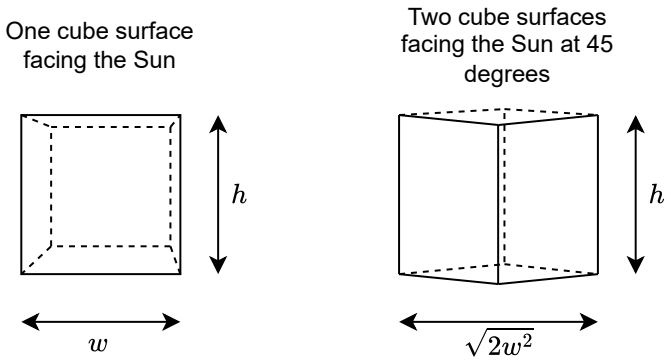


Figure 5.11: Schematic diagram showing how the attitude and/or angular rates of a spacecraft can impact the amount of solar power received. This cube has the dimensions wxwxh.

5.2. FUNcube-1 Example

This section applies the aforementioned types of sensitivity analysis to the FUNcube-1 model that has been created in Chapter 4. It gives an ”order-of-magnitude” idea of the sensitivities that can be expected when performing a thermal analysis on a small satellite. Details on how to run these cases are shown in the user manual, Section C.5.

The multiplication factors applied on the input variables in this analysis are quite large, to show the type of relationship between the parameter value and temperature. If only small deviations would be shown, all trends are likely to appear linear. For example, the Earth IR heat flux will never be off by a factor 2, but it is included in the results to show the type of trend present. In the last section, Section 5.3, the actual uncertainties and assumptions are linked to the sensitivities.

5.2.1. All-Encompassing Plots

The result of changing all input parameters by ±10-20% is shown in Figure 5.12 and Figure 5.13.

- For the **average spacecraft temperature**, the absorptivity, emissivity, and solar heat are the most important. Changing the absorptivity has a comparable effect to changing the solar heat: increasing either increases the temperature. On the contrary, increasing the emissivity results in a temperature decrease (as seen by the downwards pointing arrow on the line at EMI). Hence, having an accurate determination of the solar heat flux, as well as the absorptivity and emissivity of the spacecraft’s outer surfaces, is crucial to improving the simulation’s accuracy. For the satellite’s average temperature, the least important are the altitude, heat capacity, and thermal conductivity.
- For the temperature swing (**maximum minus minimum temperature**), the absorptivity, heat capacity, thermal conductivity, and solar heat are critical. The absorptivity and solar heat affect the temperature swing by changing the difference between eclipse and non-eclipse conditions. The heat capacity determines the ”speed” of the system, so a higher heat capacity results in a more stable temperature. Finally, the conductivity impacts the temperature swing because it determines how isolated the outer panels are. If the panels are thermally insulated from the rest of the spacecraft, the panels will heat up and cool down tremendously, compared to when they are connected well to the rest of the spacecraft.

Overall, the optical properties of the spacecraft and the solar heat flux are the most important parameters when it comes to simulation accuracy in terms of minimum, mean, and maximum temperatures experienced by the spacecraft. The orbital altitude appears to be the least influential.

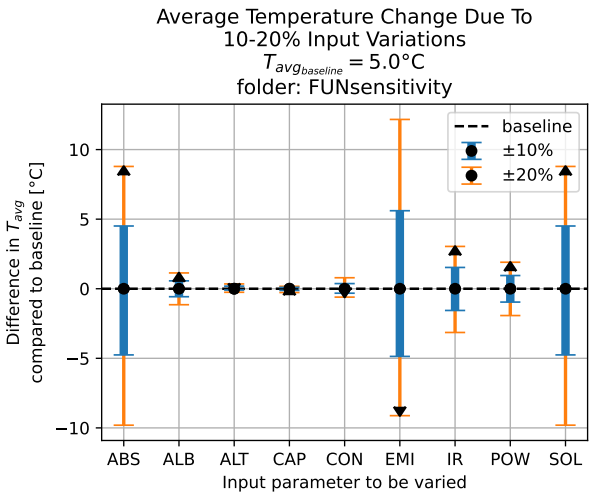


Figure 5.12: Sensitivity of FUNcube-1 (non-rotating) average temperature on various input parameters.

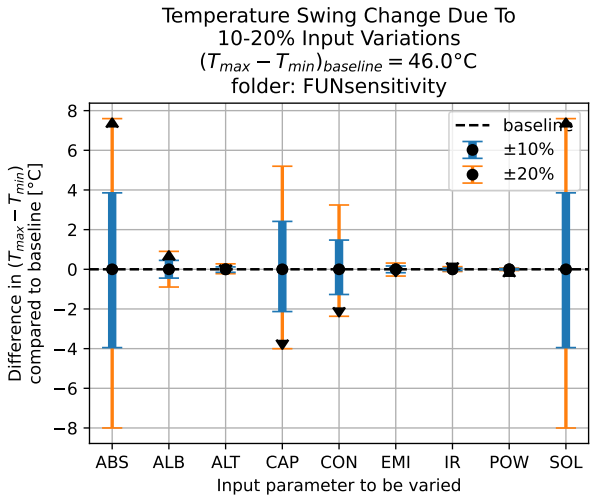


Figure 5.13: Sensitivity of FUNcube-1 (non-rotating) temperature envelope on various input parameters.

5.2.2. Internal Radiation

The internal radiation plots for FUNcube-1 are shown in Figure 5.14 and Figure 5.15. The left figure are the temperatures of two different nodes, the middle node of the outer z- face (”z-A”), and a battery placed on one of the internal PCBs (”PCB2bat”). The dotted lines show the temperatures when radiation is turned off, while the solid lines show the temperatures when internal radiation is turned on. The right figure shows the difference between the ”on” and ”off” radiation setting. The internal radiation allows the heat to spread more evenly throughout the satellite, compared to having no internal radiation. Moreover, inner PCBs that generate a lot of heat can radiate this heat away, which is the case for the battery node on PCB2, as seen inFigure 5.14. It does not seem to affect the outer node (z-A) much, only by 1.5°C at the peaks, but the battery node is approximately 5°C colder with internal radiation.

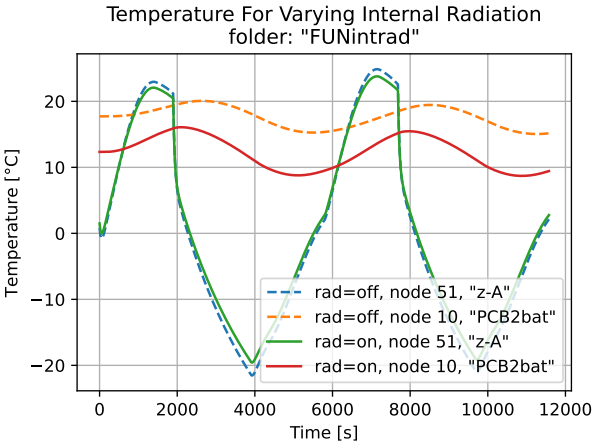


Figure 5.14: Temperatures of two nodes of FUNcube-1, for internal radiation turned off and on.

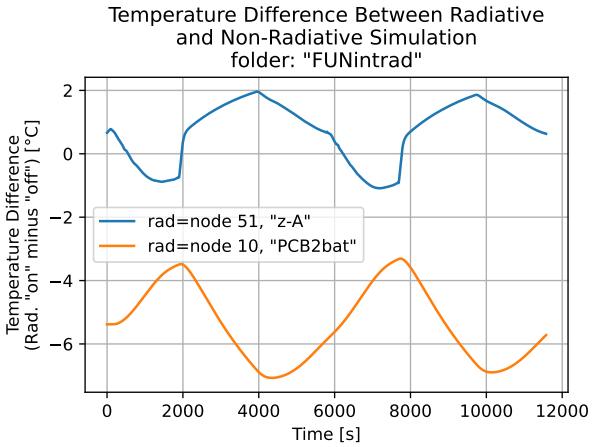


Figure 5.15: Temperature differences of two nodes of FUNcube-1, where the temperature of no internal radiation is subtracted from the internal radiation turned on case.

5.2.3. Beta Angle

The beta angle has a very nonlinear effect on the spacecraft’s temperatures, as seen in Figure 5.16 and Figure 5.17. Around an angle of approximately 70 degrees, the behaviour appears to suddenly change. This is because, as β increases, the eclipse duration shortens. At a certain point, the orbit has no eclipse anymore, and is in continuous sunlight. For FUNcube-1’s orbit altitude of approximately 600 km, this value lies close to 70 degrees. This was explained elaborately in Chapter 4.

Below this ”cut-off’ angle, the eclipse time decreases as β is increased. Hence, the satellite receives more solar heat, and the temperatures increase. After the cut-off point has been passed, the temperatures drop again. This could be due to the spacecraft’s attitude (the effective area receiving sunlight changes, see Figure 5.11), because the temperature drop after the cut-off is not really seen if the spacecraft were rotating (Figure 5.18).

The temperature profiles seen in Figure 5.17 also change a lot after the cut-off point. With $\beta > 70°$, there is no eclipse present, and hence the temperature profile is rather flat. At $\beta = 90°$, the temperature is constant, because with a non-rotating spacecraft, only one face (perpendicular to the orbit) receives sunlight.

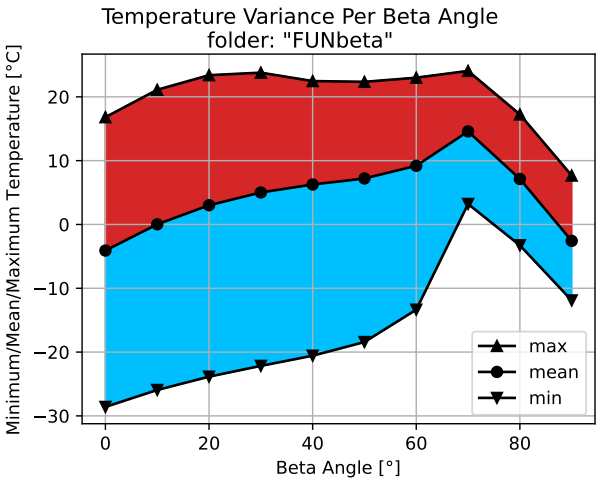


Figure 5.16: FUNcube-1's minimum, mean, and maximum temperature for different values of β.

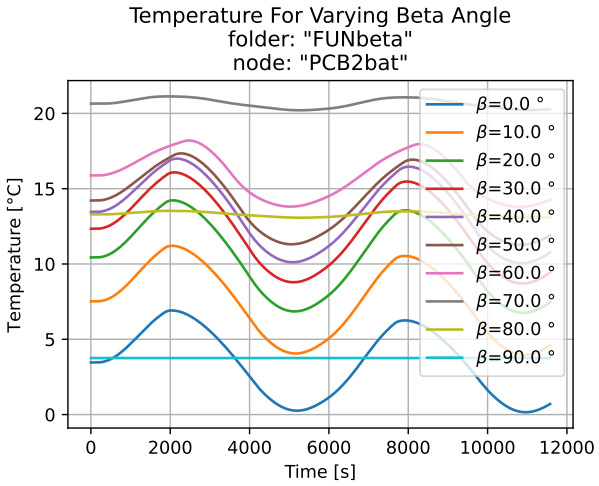


Figure 5.17: FUNcube-1's transient temperatures for different values of β. The node shown is the battery on one of the internal PCBs.

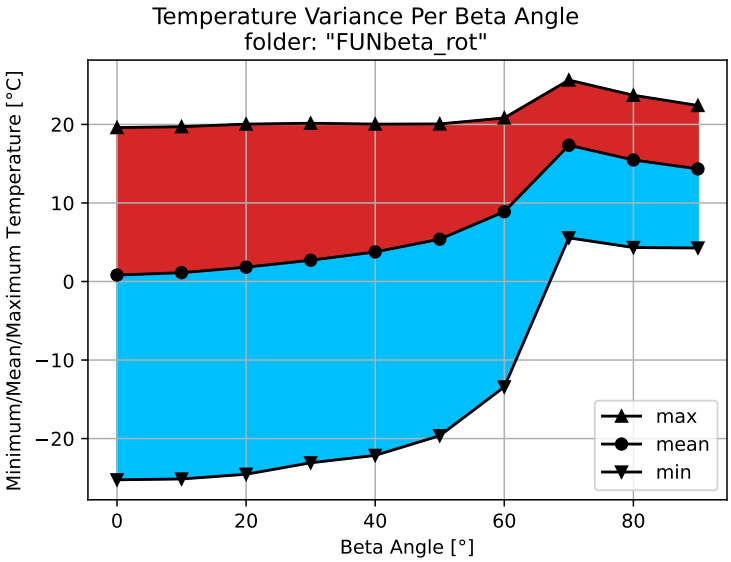


Figure 5.18: FUNcube-1's minimum, mean, and maximum temperature for different values of β, if the satellite were rotating at 1°/s around all axes.

5.2.4. Day of the Year

The day of the year impacts the amount of solar radiation received, and it may impact the beta angle as well. The latter is only impacted if the user defines the orbit with the RAAN and inclination angles; the day in the year will affect the computed beta angle, following Equation 3.29. If the beta angle is defined directly, and not via the RAAN and inclination, that beta angle will simply stay as the user defined it, regardless of the day. That is the case in Figure 5.19 and Figure 5.20. Since β does not change, only the solar radiation changes, according to the inverse-square law (Equation 3.2). During the months near January, the Earth is closer to the Sun than during the months near July, hence the higher temperatures near January.

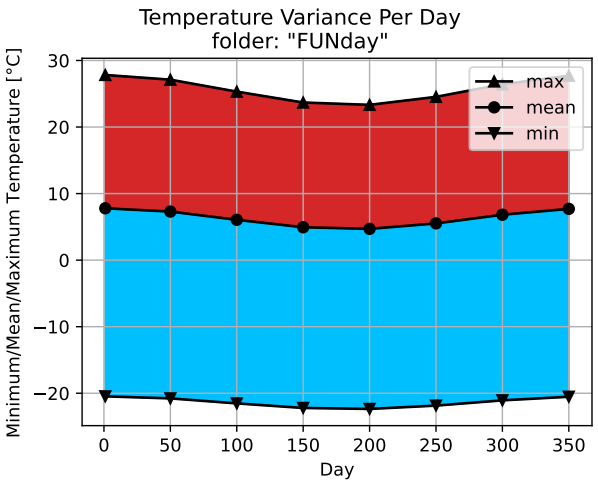


Figure 5.19: FUNcube-1's minimum, mean, and maximum temperature for different days in the year. Day 1 is January 1st.

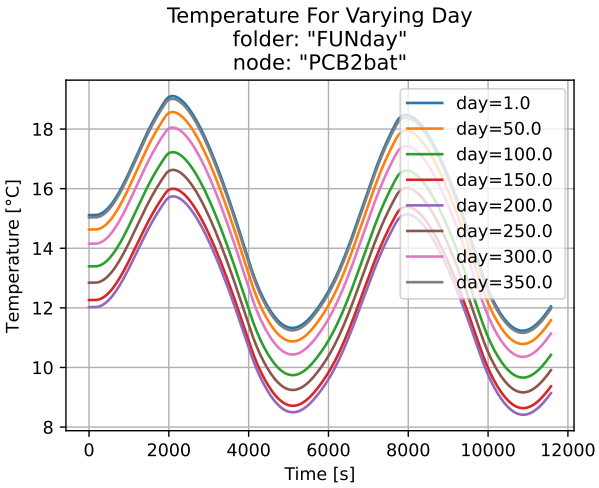


Figure 5.20: FUNcube-1's transient temperatures for different days in the year. The node shown is the battery on one of the internal PCBs.

5.2.5. Altitude

As seen in Figure 5.21 and Figure 5.22, the altitude does not have a large impact on the simulation outputs. For higher altitudes, the temperatures decrease slightly, and the eclipse fraction decreases as well. The latter would suggest a temperature increase, but simultaneously, the Earth IR flux and albedo flux decrease with increasing altitude.

It should be known that the simulation will become less accurate for higher altitudes, due to the Low-Earth Orbit assumption.

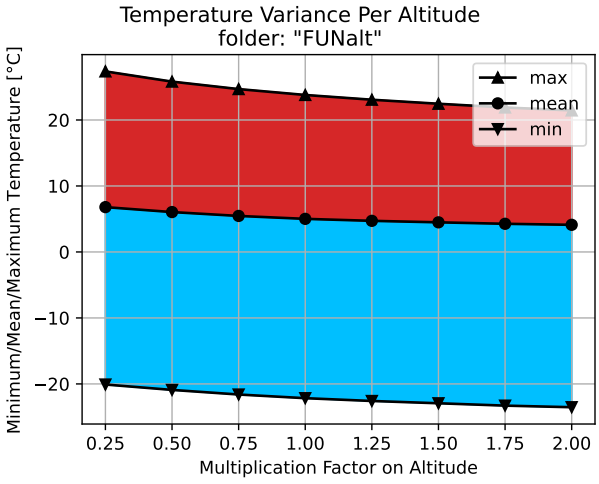


Figure 5.21: FUNcube-1's minimum, mean, and maximum temperature for varying altitude.

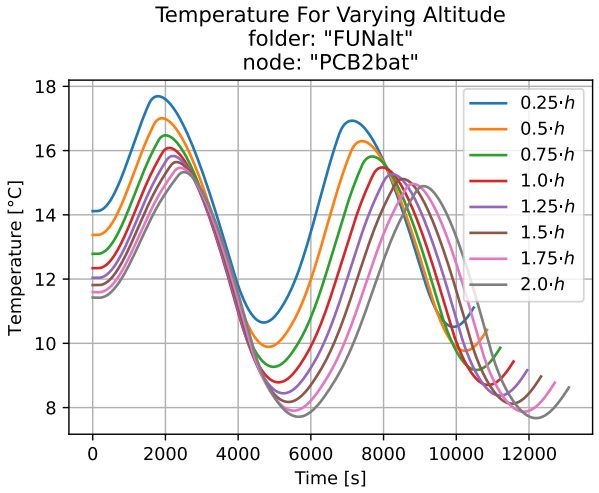


Figure 5.22: FUNcube-1's transient temperatures for varying altitude. The node shown is the battery on one of the internal PCBs.

5.2.6. Earth Infrared Heat Flux

The Earth infrared heat flux varies depending on the Earth's radiating temperature (see Section 3.3.4). The actual power received also depends on the orbit and the spacecraft's attitude. In the sensitivity analysis, the heat flux emitted by the Earth is the varied parameter. Hence, this is not an uncertainty in the spacecraft design, but an uncertainty in the Earth's assumed radiating temperature. More on this

assumption is presented in Section 5.3.

As seen in Figure 5.23 and Figure 5.24, there is a nearly linear trend between the spacecraft's temperatures and the Earth IR flux. More IR heat from the Earth results in a higher temperature. The shape of the temperature profile, which is seen in Figure 5.24, remains nearly identical, since the Earth IR power is constant throughout the orbit.

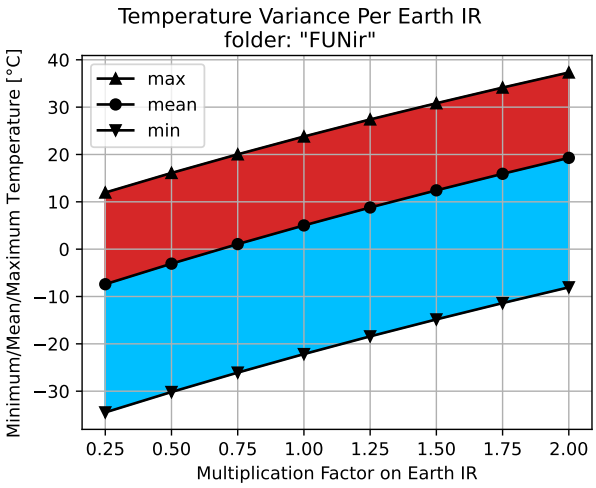


Figure 5.23: FUNcube-1's minimum, mean, and maximum temperature for a varying Earth IR power.

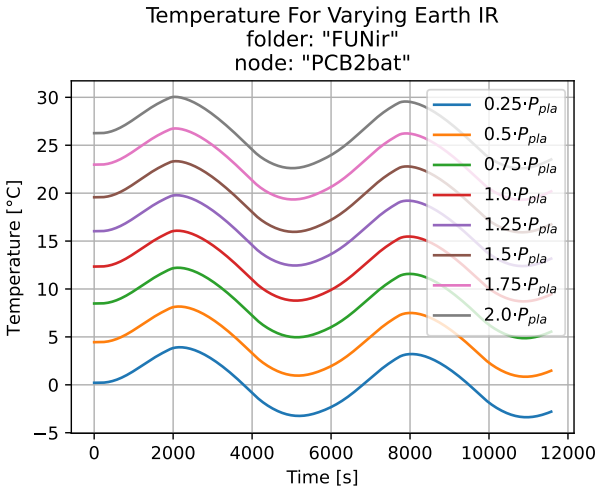


Figure 5.24: FUNcube-1's transient temperatures for varying Earth IR power. The node shown is the battery on one of the internal PCBs.

5.2.7. Albedo Heat Flux

As seen in Figure 5.25, the spacecraft's temperature varies nearly linearly with albedo power. Furthermore, it can be seen in both Figure 5.25 and Figure 5.26 that the temperature swing become slightly more strong as well; more albedo results in a stronger temperature increase during the non-eclipse period of the orbit.

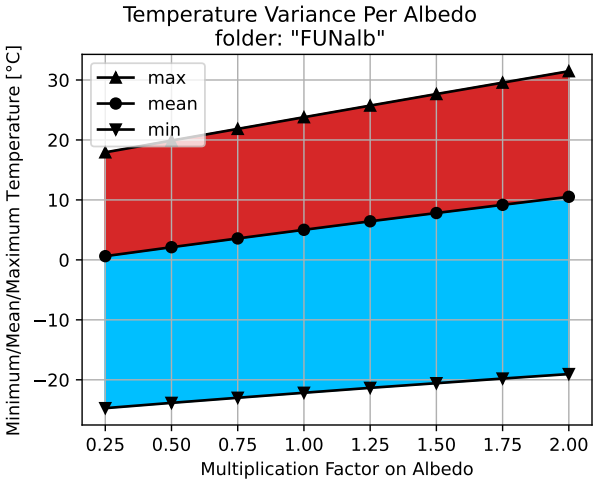


Figure 5.25: FUNcube-1's minimum, mean, and maximum temperature for varying albedo. A too high multiplication factor could result in a > 1, which is impossible.

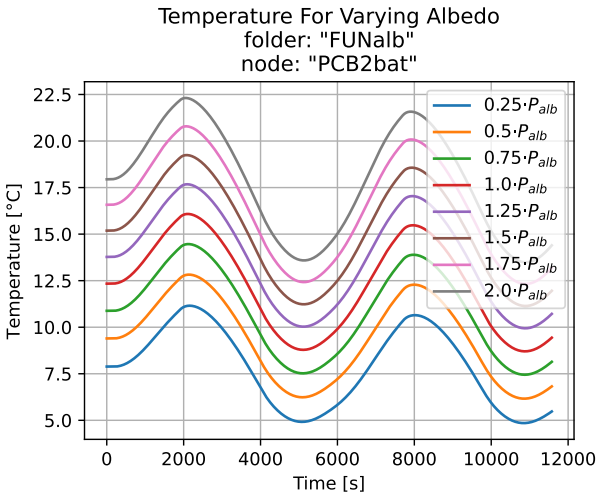


Figure 5.26: FUNcube-1's transient temperatures for varying albedo. The node shown is the battery on one of the internal PCBs.

5.2.8. Solar Heat Flux

The temperature variations due to a varying solar flux, as seen in Figure 5.27 and Figure 5.28, are very strong. The temperatures rise with solar flux, and the temperature swing increases as well. As the temperature rises, the spacecraft also loses more heat to space, and hence cools down faster during the eclipse. Moreover, the solar flux also indirectly impacts the albedo flux via a linear relationship.

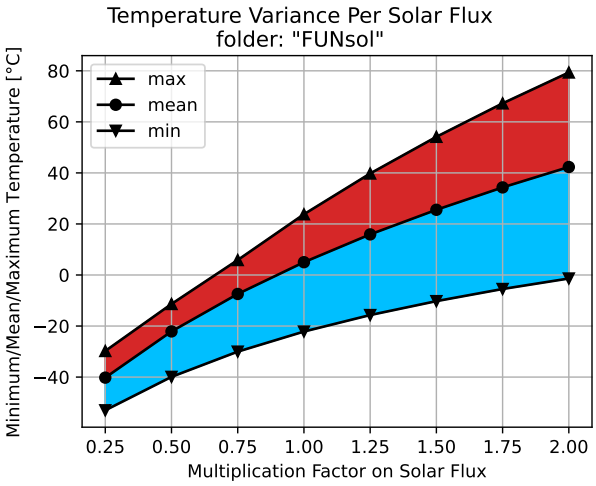


Figure 5.27: FUNcube-1's minimum, mean, and maximum temperature for varying solar heat.

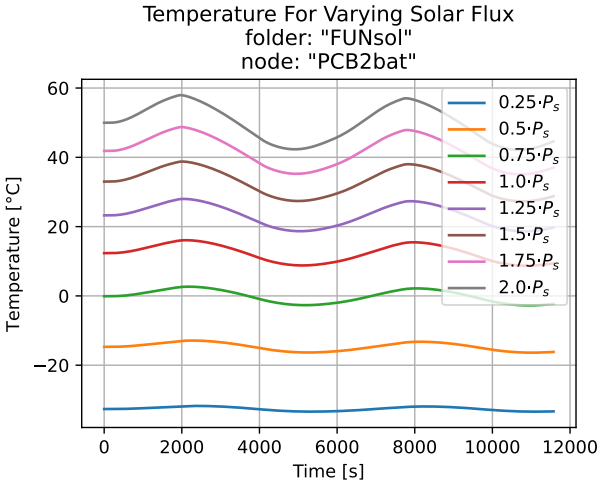


Figure 5.28: FUNcube-1's transient temperatures for varying solar heat. The node shown is the battery on one of the internal PCBs.

5.2.9. Internal Power

Similar to the Earth IR power, varying the internal power results in an almost linear increase temperature, see Figure 5.29 and Figure 5.30. Furthermore, the shape of the temperature profile also does not change significantly, since the internal power (in this case) is constant throughout the entire orbit.

However, there is a change in slope of the maximum temperature (Figure 5.29) around a multiplication factor of 1.5. This was found to be due to a different node becoming the hottest. For low internal powers, the outer nodes receiving sunlight reached the highest temperatures. However, as the internal power increases, the internal nodes (PCBs) become hotter, and surpass the outer nodes in terms of temperature.

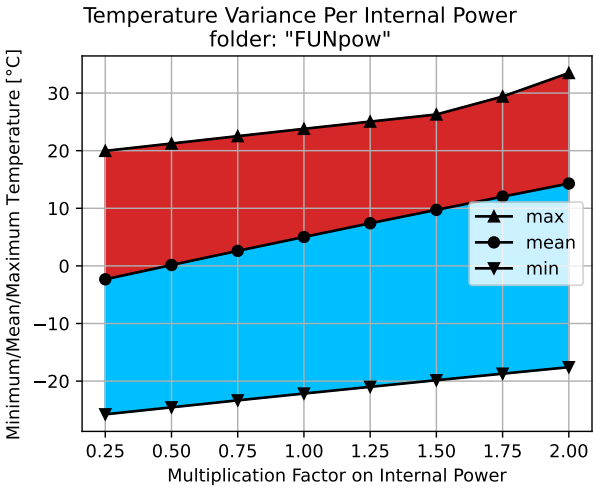


Figure 5.29: FUNcube-1's minimum, mean, and maximum temperature for varying internal power.

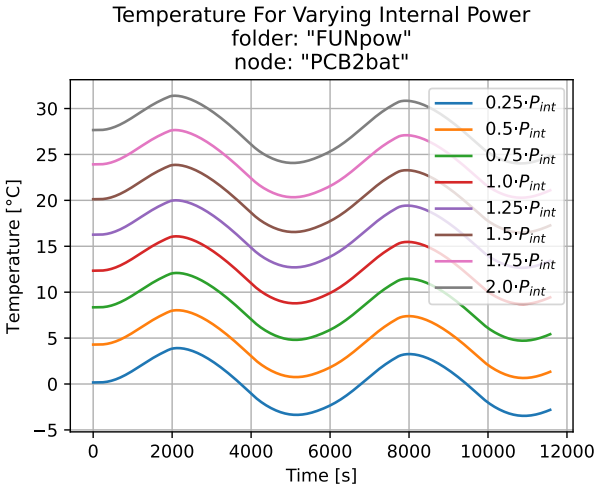


Figure 5.30: FUNcube-1's transient temperatures for varying internal power. The node shown is the battery on one of the internal PCBs.

5.2.10. Absorptivity

Figure 5.31 and Figure 5.32 show the impact of the solar absorptivity on the spacecraft's temperatures. It can be seen that they are identical to the Figure 5.27 and Figure 5.28, which showed the temperatures for varying solar power. The impact of absorptivity is identical to that of the solar power, since they are always multiplied with each other in the heat balance equation, Equation 3.32. However, the absorptivity is more likely to vary compared to the solar power, and therefore, its importance during the design is higher than estimating the solar heat flux exactly correctly.

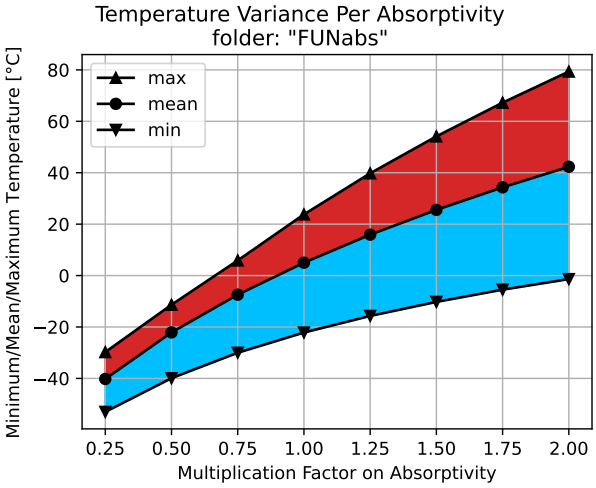


Figure 5.31: FUNcube-1's minimum, mean, and maximum temperature for varying absorptivity. A too high multiplication factor could result in $\alpha > 1$, which is impossible.

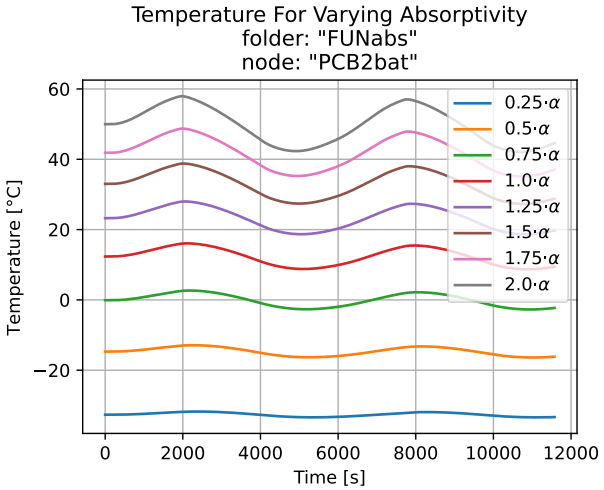


Figure 5.32: FUNcube-1's transient temperatures for varying absorptivity. The node shown is the battery on one of the internal PCBs.

5.2.11. Emissivity

Figure 5.33 and Figure 5.34 show that the temperatures decrease significantly as the emissivity is increased. This effect is strongest for low emissivities. As demonstrated by Figure 5.34, the transient temperature profile remains almost identical, except that the temperature shifts up or down. Hence, the temperature swing (maximum minus minimum temperature) does not change with the emissivity.

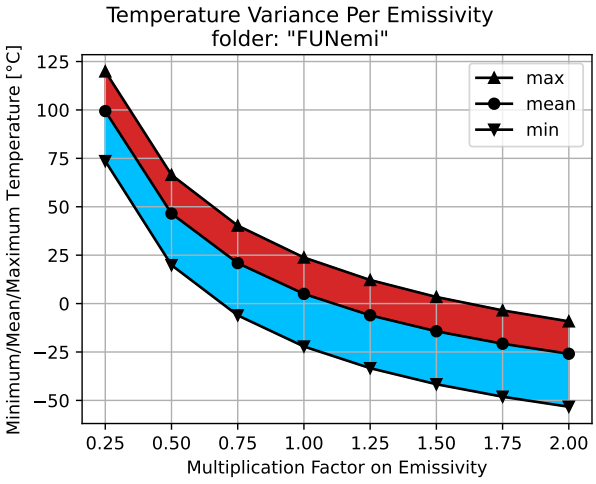


Figure 5.33: FUNcube-1's minimum, mean, and maximum temperature for varying emissivity. A too high multiplication factor could result in $\varepsilon > 1$, which is impossible.

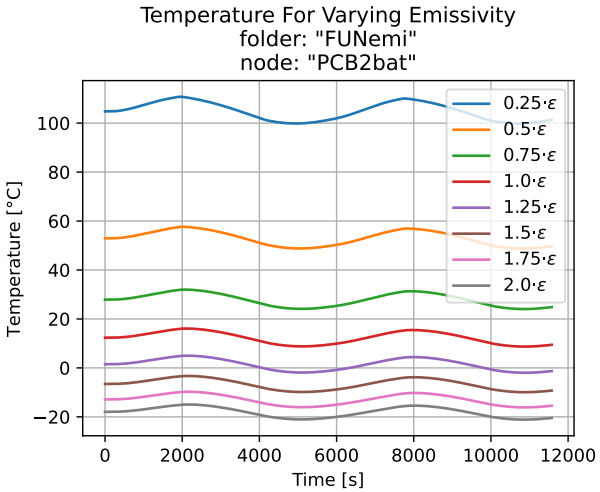


Figure 5.34: FUNcube-1's transient temperatures for varying emissivity. The node shown is the battery on one of the internal PCBs.

5.2.12. Heat Capacity

Evidently, the heat capacity of the satellite (its thermal mass) mostly impacts the temperature swing, and not the average temperature. This is demonstrated by Figure 5.35 and Figure 5.36. As the spacecraft becomes more lightweight, the temperature reacts much more quickly to environmental changes; the spacecraft heats up more quickly in sunlight, and cools down more quickly in eclipse.

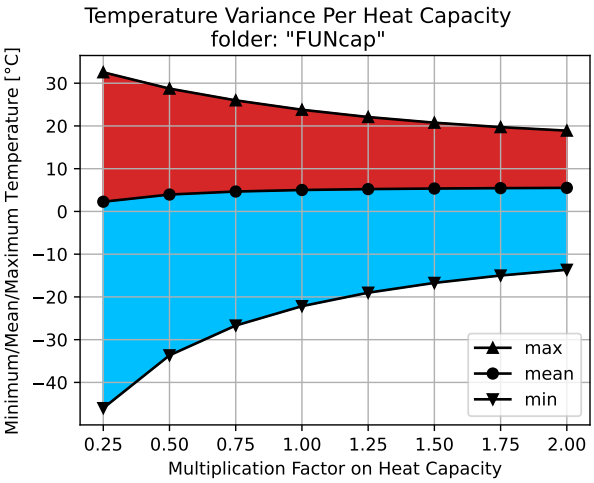


Figure 5.35: FUNcube-1's minimum, mean, and maximum temperature for varying heat capacity.

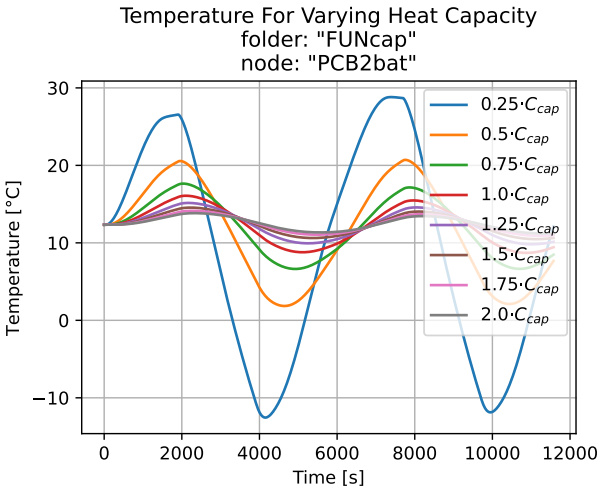


Figure 5.36: FUNcube-1's transient temperatures for varying heat capacity. The node shown is the battery on one of the internal PCBs.

5.2.13. Thermal Conductance

The thermal conductance between all nodes has a less "predictable" effect on the spacecraft's temperatures, as it is not as simple as adding more power to the system. Investigating Figure 5.37, the average temperature is not much affected by the conductance. Moreover, the temperature swing becomes narrower with increasing conductance, since the temperature gradients will be flattened due to more internal heat flow.

Since the conductance impacts different nodes differently, two nodes are shown here. Figure 5.38 shows the battery temperatures. The temperatures decrease with increasing conductance, likely because the internal power dissipates more easily through the PCBs. On the contrary, Figure 5.39 shows the temperatures for the zenith-facing outer panel, x+. In contrast to the battery, the mean temperature does not increase or decrease, but the swing decreases as the conductance increases. This is because the outer panel receives direct solar radiation, and also radiates heat to space. Increasing the conductance connects the outer panel better with the rest of the spacecraft, making its temperature more stable.

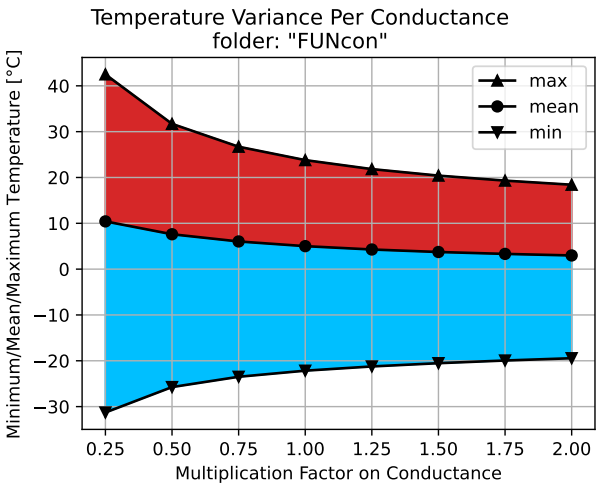


Figure 5.37: FUNcube-1's minimum, mean, and maximum temperature for varying thermal conductance.

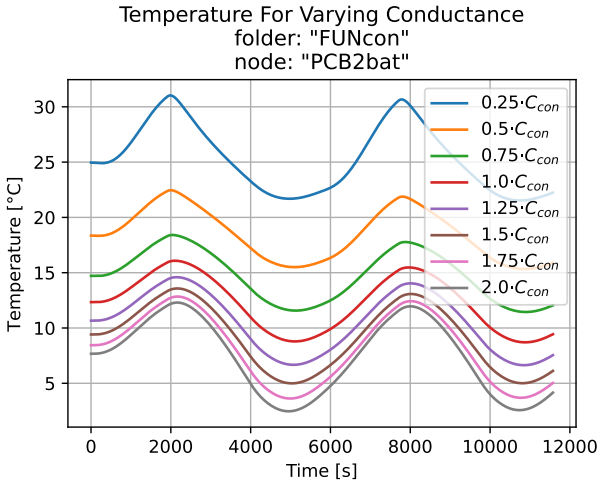


Figure 5.38: FUNcube-1's transient temperatures for varying thermal conductance. The node shown is the battery on one of the internal PCBs.

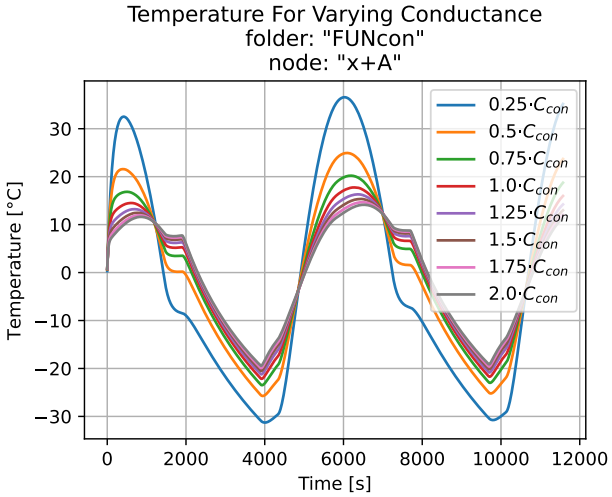


Figure 5.39: FUNcube-1's minimum, mean, and maximum temperature for varying thermal conductance. The node shown is the central node of the outer panel x+ (facing zenith).

5.3. Assessment of Assumptions & Sensitivity Results

This final section of the sensitivity analysis will reflect on assumptions made to perform the thermal analysis with the software, and assess to what extent the assumptions are valid. Moreover, all other input parameters are also tested for their estimated input deviations and the resulting output deviations. All sensitivity plots for FUNcube-1 were already shown in Section 5.2, but now, the results will be summarised into a single table.

5.3.1. Revisiting Assumptions

The assumptions made at the beginning of this project (Section 3.3.1 and Section 3.4.1) are repeated below. The expected effect of this assumption is also mentioned.

1. The satellite is in LEO; hence, the Sun-Earth vector is equal to the Sun-satellite vector. As explained in Section 3.3.3, a 500 km LEO orbit would result in a 0.0026 degree angle between the Sun-Earth and Sun-satellite vector. This angle would result in a slightly different angle of incidence on the spacecraft's faces. For example, if a face would be parallel to the Sun, it would not receive any

radiation. If it were rotated by 0.0026°, it would receive $sin(0.0026°) = 4.5 \cdot 10^{-5} \rightarrow 0.0045\%$ of the maximum possible heat; assuming 1361 W/m^2 of solar radiation [43, 45], the difference would be 0.061 W/m^2. This is far lower than any other deviations caused by differences in attitude or beta angle.

2. The Sun rays are parallel at Earth, resulting in a cylindrical eclipse volume. Hereby, the penumbra region is ignored. To give an order-of-magnitude idea of the size of the penumbra region, Figure 5.40 and Figure 5.41 are used to derive some approximations for the time that a spacecraft experiences in penumbra. The angles θ_1 and θ_2 are computed with:

$$\theta_1 = arctan\left(\frac{R_{Sun} - R_{Earth}}{1\ AU}\right) = 4.608 \cdot 10^{-3}\ rad = 0.264\ deg \tag{5.2}$$

$$\theta_2 = arctan\left(\frac{R_{Sun} + R_{Earth}}{1\ AU}\right) = 4.693 \cdot 10^{-3}\ rad = 0.269\ deg \tag{5.3}$$

Hence, one penumbra region is approximately 0.533 degrees.

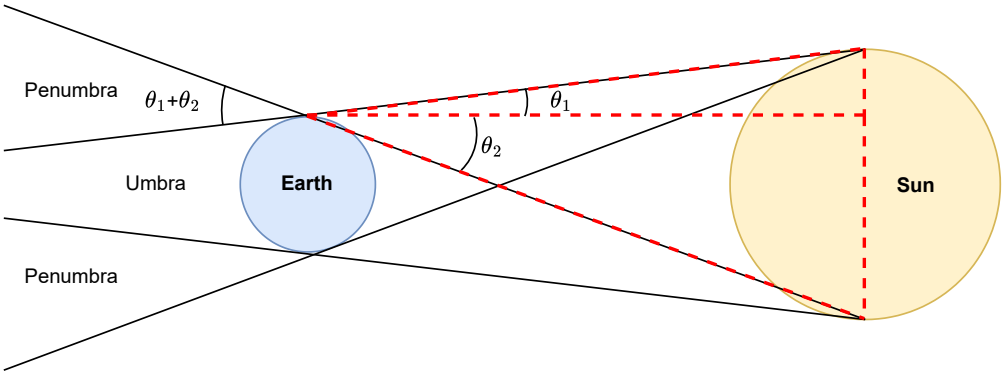


Figure 5.40: Schematic diagram showing the penumbra regions.

Using Figure 5.41, assuming an altitude of 500 km, distance d is:

$$d = \sqrt{(R_{Earth} + h)^2 - R_{Earth}^2} = 6871\ km \tag{5.4}$$

Hence, for one penumbra region, using a small-angle approximation from point X to distance d, the distance that the spacecraft travels through the penumbra region is in the order of:

$$s_{penumbra} \approx d \cdot (\theta_1 + \theta_2) \approx 64\ km \tag{5.5}$$

Finally, the orbital velocity of a circular orbit at 500 km is approximately 7.6 km/s, resulting in a penumbra duration of roughly 8 seconds. The penumbra happens twice in one orbit, so the satellite is in penumbra for roughly 16 seconds per orbit. This is totally negligible considering the orbit period of 5668 seconds, and looking at the FUNcube-1 simulation (Figure 4.40), extending or shortening the eclipse by 16 seconds would result in an approximate temperature difference of $\pm$0.14°C.

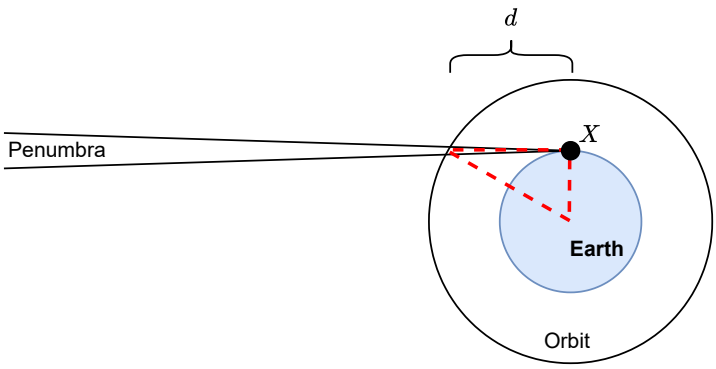


Figure 5.41: Schematic diagram showing an approximation for estimating the penumbra duration.

3. The Earth IR flux is uniform over the Earth's surface and constant throughout time. The flux at the Earth's surface was assumed to be 239 W/m^2, following from [46], see Section 3.3.4. The envelope of the Earth IR flux values throughout day- and nighttime and throughout the seasons is realistically between 200-260 W/m^2, which is close to a 15% deviation from the mean. With "realistically" it is meant that some outliers, such as the 90° latitude value is ignored, since it shows a very low value, but the surface area of this latitude is very small compared to lower latitudes. This 15% deviation is entered into the sensitivity analysis module of the software, and the results are shown in Table 5.1.
4. The albedo is uniform over Earth's surface and constant throughout time. This assumption has a similar character as assumption 3. The albedo is assumed to be 0.297, using [46], see Section 3.3.5. The variations throughout the year and for different latitudes are illustrated in Figure 5.42.

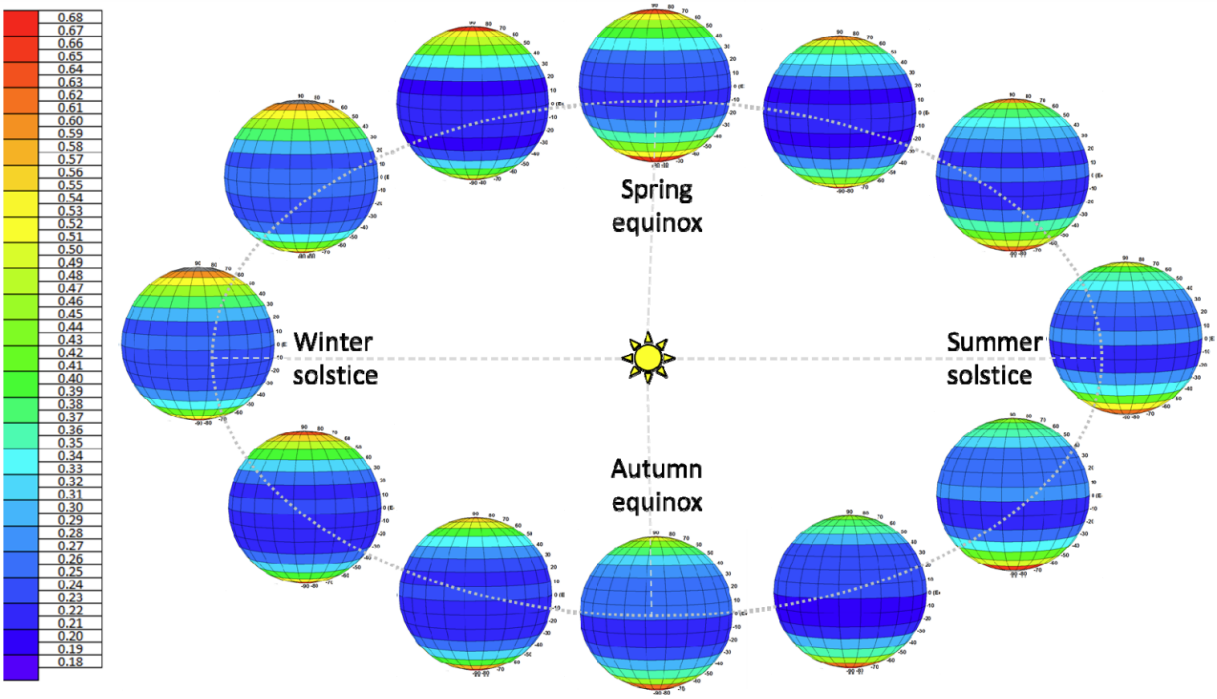


Figure 5.42: Earth albedo variation throughout the year, for different latitudes [46].

The latitudes 45-90 degrees have an average albedo of approximately 0.40, while the albedo near the equator (0-45 degrees) is closer to 0.20. This is a 30% deviation from the assumed value of 0.3. This value will be entered into the sensitivity analysis, and the result is entered in Table 5.1. The worst-case scenario would be an equatorial orbit, where the albedo can be assumed to always be low. In that case, one may consider permanently changing the albedo value in the software. However, in

most cases of a Sun-synchronous orbit, the effects of the low albedo near the equator and the high albedo near the poles, will average each other out through the orbit.

5. Free molecular (aerothermal) heating is ignored. As explained in Section 2.3.1, the expected heat flux received at a very low orbit (350 km) is roughly 1.9 W/m^2. As a simplification, this value will be added onto the Earth IR flux to compute a reasonably similar temperature increase, since the Earth IR flux is also constant throughout the entire orbit (unlike albedo an Sunlight). The result is shown in Table 5.1.
6. Charged-particle heating is ignored. As explained in Section 2.3.1, the magnitude of this type of heating depends on the spacecraft's temperature. The colder, the more heating occurs. A heating of 2 K could occur at temperatures near 100 K [25]. However, SmallSats will likely never experience such temperatures (except perhaps for specialised scientific experiments), and according to [25], the charged-particle heating is completely negligible for conditions near room temperature.
7. The satellite's mass and material properties are constant throughout time. For this sensitivity analysis, the loss of mass via propellant, and the change of optical properties due to Ultra-Violet (UV) exposure and atomic oxygen (AO), are considered. The solar absorptivity of white paints is the most prone to degradation (the well-known "yellowing" of white paints), where the absorptivity increases with increasing UV radiation and AO [63]. The degradation of the emissivity can be disregarded [63].

   These parameters may be time-varying, and the simulation assumes them to be constant. However, if some data is available on the expected changes over time, one could always conduct multiple thermal simulations with those different parameters, to attain a low-fidelity indication of the thermal changes throughout the years.

   The propellant loss can be in the order of 50 grams [64] (which would be a 5% decrease from the 1 kg of FUNcube-1), while the absorptivity may increase 14-39% in one year [63, 65] (a mean increase of 27% is assumed). The mass loss is simulated as a change in heat capacity, and the absorptivity change is simulated directly. The results are shown in Table 5.1.
8. Multiple reflections are not included. The internal radiation analysis is complicated to simulate within this Python software, so good comparisons can only be made by comparing the software with ESATAN. Chapter 4 shows a number of such comparisons between the software and ESATAN. This will not be further discussed in this section, although the effect of all internal radiation on or off is shown in Section 5.3 to get an idea of the order of magnitude of internal radiation.
9. Internal radiation is only computed for perfectly orthogonal or parallel plates. This assumption ties into assumption 7. It is recommended to visit Chapter 4 to assess this effect.
10. There is no convection in/around the entire spacecraft. Convection outside the spacecraft could occur if the altitude is low enough. This is already considered in assumption 5. Convection inside the spacecraft would occur if elements such as heat pipes are used. The simulation of such elements is very complicated, and H.S.B. Brouwer has written a MSc thesis about this topic [66]. This investigation is entered as a recommendation for future work in Chapter 7.

5.3.2. Sensitivity Results

As explained in the previous sub-section, assumptions 3, 4, 5, 7, and partly 10 are assessed by means of simulation of FUNcube-1. The results are presented in Section 5.3. The other assumptions were shown to be completely negligible, and assumptions 8 and 9 are dealt with in Chapter 4.

Furthermore, the other input parameters for FUNcube-1, as discussed in Section 5.2, are also included in Table 5.1, using realistic uncertainties for those inputs. It is crucial to note that these uncertainties can be reduced by performing more tests on the spacecraft. More optical tests can increase the accuracy on the absorptivity, more thermal tests can increase the accuracy on the conductances, etc.

Moreover, it should be noted that all the values below are related to performing a single simulation before launch. This would not take time-varying parameters into account. However, it is possible to split up the simulation into multiple simulations, for example, one right after launch, and then a new simulation for each year after that. This could mainly be useful for predicting the effects of absorptivity degradation, and

similar time-varying effects.

The estimated uncertainty in all other input parameters besides the assumptions was determined using the following reasoning:

- **Internal radiation**: In this case, the only simulation options available are an ”on” and ”off” setting. For more details, Chapter 4 can be visited, where ESATAN-TMS was used to compare the results with.
- **Beta angle**: Since most SmallSat orbits are Sun-synchronous (Figure 3.1, notice the 98° inclination for most orbits, indicating a Sun-synchronous orbit (SSO)), the beta angle remains almost constant throughout the year. 10:00 to 11:00 orbits are very common amongst Sun-Synchronised orbits, since they are optimal for Earth observation satellites, due to the presence of clear shadows on the Earth at that particular LTAN [28, pp.221-223]. A 10:00 orbit has a beta angle of 30 degrees, which is similar to FUNcube-1’s orbit. It is difficult to know the accuracy of this, since it mostly depends on orbit injection accuracy, which depends on the launcher, and it may then depend on orbit maintenance capabilities. For now, an uncertainty of $\beta \pm 10$ degrees is assumed; the results are shown in Table 5.1.
- **Day of the year**: The day of the year is influenced by the launch vehicle, and especially for smaller missions, the launcher availability is not always predictable. Therefore, the uncertainty is considered to be the entire year.
- **Solar flux**: The solar flux varies with the day of the year, but that effect is already included in the day parameter. Now, the solar ”constant” is assessed (it is not really a constant) based on the solar activity [67]. The variability purely due to Sun spots etc. is maximally 0.34%, for short time scales of days to weeks [67]. As seen in Table 5.1, the day has a far larger impact than the solar activity.
- **Internal power**: This value can be tested extensively on-ground, and if the different power modes (eclipse versus non-eclipse) are entered properly into the simulation, the uncertainty is expected to be fairly low; approximately 10% is assumed.
- **Absorptivity**: As seen in assumption 7, the absorptivity can vary greatly for white paints, as it degrades over time. However, assuming solely the knowledge of the absorptivity value at launch, it is dependent on the measurement technique. Regardless, looking at the material database of the software, the absorptivity varies much more than the emissivity (e.g., white paint and black paint have similar emissivities, but vastly different absorptivities). Hence, a rather large uncertainty of 10% is assumed. This excludes the degradation effect, which is shown as its own separate parameter.
- **Emissivity**: For different paints, the emissivity remains fairly constant. Hence, it is assumed to be known with a higher confidence than the absorptivity; 5% uncertainty is assumed.
- **Heat capacity**: The mass of the spacecraft can be known with essentially arbitrary accuracy, due to commonly available good weight scales. However, this does not immediately relate to the thermal capacity, since that depends on the materials used, as well. Thermal tests can be performed to estimate the time constant of the system, and hence correlate the thermal capacity of the simulation with the tests. However, this is less accurate than the scale, so an uncertainty of 10% is assumed.
- **Conductance**: The thermal conductance is much more difficult to determine than the capacity. An integrated satellite has so many connections that they are basically impossible to derive from one thermal test. Many smaller tests, or reliable databases, have to be used to gather better information. However, due to the unreliable nature of this parameter, a larger uncertainty of 20% is assumed.
- **Rotational rates**: It was shown in Section 5.1.4 of this chapter that there can be large differences in thermal behaviour depending on the attitude and/or tumbling rates. After running numerous simulations with different angular rates, it was found that the speed of the rotation does not have a significant impact on the average temperatures, but the most important factor is whether it is spinning or not, and in which direction. Hence, the rotational rates are simply compared as ”rotating” versus ”non-rotating”. The worst-case scenario was taken (positive spin about the x-axis, and negative spin about the y- and z-axes), 5°/s each (is not a high nor low freely tumbling rate; for the temperatures, the main importance is not the rates, but more whether it is spinning or not at all).

Table 5.1: Effect of variable uncertainty (& assumptions) on simulation output.

Variable	**Variable Uncertainty**	**Temperature Uncertainty** [°C]		
		T_{min} -22.2°C	T_{mean} 5.0°C	T_{max} 23.8°C
	Assumptions			
3. Earth IR Flux	15%	±2.3	±2.3	±2.2
4. Albedo Flux	30%	±1.0	±1.7	±2.3
5. & 10. Aerothermal/ Convection	Earth IR +1.9 W/m^2 (=0.8%)	+0.1	+0.1	+0.1
7.A. Propellant Loss	C_{cap}: -5%	-0.7	0.0	+0.4
7.B. Absorptivity Degradation (1 year)	+27%	+7.0	+11.7	+17.1
	Input Variables			
Internal Radiation	on/off	±0.3	±2.4	±1.1
Beta Angle (rotating satellite)	±10°	±1.6	±1.4	±0.6
Day of the Year	any day	±2.0	±3.1	±4.5
Earth IR Flux (assumption 3.)	15%	±2.3	±2.3	±2.2
Albedo Flux (assumption 4.)	30%	±1.0	±1.7	±2.3
Solar Flux (Solar Activity)	0.34 %	±0.1	±0.2	±0.2
Internal Power	10%	±0.5	±1.0	±0.5
Absorptivity	10%	±2.8	±4.5	±6.6
Emissivity	5%	±2.8	±2.7	±2.8
Heat Capacity	10%	±1.6	±0.1	±0.8
Conductance	20%	±1.0	±0.8	±2.2
Rotational Rates	non-rotating/ rotating	±1.2	±2.7	±5.3

6

Conclusion

The research question of this thesis was:

Research Question

How can thermal analysis tools be optimised for SmallSats, while maximising user-friendliness and maintaining the reliability of results?

Three research sub-questions were formulated to assist in answering the main research question:

Research Sub-Question 1

What is the impact of implementing modularity in a SmallSat thermal modelling software on the user experience and reliability of results?

Research Sub-Question 2

What level of accuracy can be achieved when validating a SmallSat thermal model with flight data?

Research Sub-Question 3

Which elements within a SmallSat thermal model demonstrate the greatest influence on simulation outputs?

First, the research sub-questions will be answered, after which a general conclusion is made by answering the main research question.

6.1. Answering the Research Questions

6.1.1. Research Sub-Question 1

Firstly, allowing a thermal model (`NodalModel` in Python context) to be connected to another thermal model with a single connect statement is the gateway to modularity. This allows to create hierarchical models of arbitrary depth, and easily re-use pre-made models, such as PCBs.

Secondly, a material database showed to be an effective way of introducing modularity into the software. It is a significant time-saver to store numerous materials, coatings, and contact connections in one database, compared to defining them on-the-fly in Python. However, optical properties vary between different brands of paint, and contact conductances vary depending on the applied pressure. The uncertainty in those properties should always be assessed.

6.1.2. Research Sub-Question 2

FUNcube-1 flight data from 2016 and data from 2024 was used for the validation; the Root-Mean-Square Errors for a number of outer panels of the satellite are summarised in Table 6.1. The internal temperatures of the spacecraft are more stable than these outer panels, and therefore will likely show lower errors. In 2016, FUNcube-1 had was spinning at approximately 2°/s, while in 2024, the spin rate was close to zero. In both cases, the z-axis of the spacecraft was aligned with the Earth's magnetic field.

If the spacecraft's attitude is known in advance of the launch, the temperature errors are likely to be much smaller, and the uncertainties easier to predict. If the attitude is not known, multiple different thermal simulations could be performed, each investigating a different attitude/tumbling scenario.

Table 6.1: RMSE values for the validation cases of four outer panels of FUNcube-1 in 2024 and 2016.

Spacecraft Face	2024 RMSE [°C]	2016 RMSE [°C]
x+	5.78	6.20
x-	2.76	7.08
y+	1.38	6.58
y-	4.59	6.83

6.1.3. Research Sub-Question 3

By performing a sensitivity analysis on FUNcube-1, it was found that the solar absorptivity, and absorptivity degradation in particular, result in the largest temperature uncertainties ($\pm$6.6°C uncertainty regardless of degradation, and +17.1°C for degradation):

- The absorptivity in general appears to be an uncertain parameter, since it varies much more depending on the type of paint etc. compared to the emissivity, which tends to vary less for different kinds of paint. Therefore, the combination of the absorptivity being uncertain and also having a strong impact on the spacecraft's temperatures, makes it perhaps the most important input parameter for thermal analysis.
- The absorptivity degradation mainly occurs in white paints; hence, if white paint is used, data on degradation due to UV and atomic oxygen exposure should ideally be provided by the manufacturer. If the level of degradation is known, the effect on the spacecraft's temperatures can be predicted by performing different simulations with different levels of degradation.

6.1.4. Main Research Question

Firstly, an issue of large, commercial thermal analysis tools is their steep learning curve and black-box appearance in terms of calculations. The learning curve can be flattened by:

- Including modularity into the software. A material database showed to increase the speed of making a thermal model, while also providing a comprehensive overview of all values used in a single file. Re-usable thermal models allow to easily connect existing models to new models.
- Making assumptions and not including each possible feature in the software. In this way, unnecessary complexity is avoided. If more complex features do need to be included, it should be done in such a way that it does not unnecessarily cross the user's path if they do not want to use the feature.
- Having an object-oriented software with most functions embedded in the objects. With this approach, the user does not need to perform any computations by hand. However, this might introduce a black-box feeling, so thorough documentation is essential. Moreover, the software should always maintain an ”overrule” function, where the user can overrule any automatically computed values with their own values.

Secondly, the software should maintain a sufficiently high reliability of results. This does not just mean to perfectly estimate the temperatures when a lot of information about the satellite is known, but also to know

the uncertainties when not all information about the satellite is known. In reality, especially with SmallSats, there is always information missing. A sensitivity analysis is a solution for cases with missing information: if the uncertainties are known, the model does not need to be perfect.

6.2. Lessons Learnt

This final section focuses on a number of important lessons that developing this software has brought to light. They are not conclusions of the results of this thesis, but conclusions of the process. The following lessons are considered of great importance:

- The user experience for any software is critical. One of the aims of developing this Python software was to increase the speed of performing a thermal analysis. However, if the documentation is confusing or lacking, if the syntax is convoluted, and if important features are missing, the user may decide to not use the software at all.
- Even if the software is ”perfect”, the results would only be reliable if the model is properly defined and verified. The software computes what the user defines, so if the user defines a model of poor quality, the results will be poor. In systems engineering terms, this is referred to as ”garbage in, garbage out”.
- Verification and validation are key to gaining confidence in the software and showing this to the user. It has happened during this thesis project that errors of certain computations were only found weeks after initially developing those computations. Some small sanity checks are important, but not sufficient.
- The computational time of a single simulation may be small (1-10 seconds), but in a sensitivity analysis, this time will add up to much larger numbers. For this reason, even if the computational efficiency seems to be an unimportant parameter, it may still be worth it to improve the computational scheme.

7

Recommendations

The thermal analysis software developed in this thesis offers reliable results for heat analyses in orbit, and for nodal thermal models of arbitrary complexity. Furthermore, the software has been written in Python, and has a particular focus on modularity; this leaves room for future students or engineers to expand or adapt the software to their liking.

This chapter provides an overview of some recommendations for the future continuation of this research project. The recommendations will be presented in no particular order; their relative importance and simplicity of implementation are shown in a matrix in Table 7.1. This table was also used during the thesis project as a rough guideline on which features to work on first. For the same purpose, the table is presented here. As expected, the recommendations mentioned here all fall in the lower right corner of the matrix, since the really important points have been tackled in this thesis.

1. Include a tool for test correlation. The unknown parameters (such as conductances) would be varied by, e.g., Monte-Carlo analysis or an artificial intelligence (AI) model, converging to a best fit. Such a tool would prevent the user from having to manually tweak each parameter and see what happens to the results. It is expected that this would be especially beneficial for complex models with many interrelated unknown parameters.
2. Provide more detailed control over the spacecraft's attitude throughout the orbit. Currently, the attitude is either fixed along the velocity vector, or it is assigned constant angular rates. It would be beneficial to allow for time-varying angular rates or very specific angular positions. Additionally, a Sun-pointing mode might be relevant if the spacecraft has some type of Sun-tracker.
3. Besides the RAAN and inclination orbit parameters, it could be useful to be able to define the orbit using the Local Time of the Ascending Node (LTAN). The software would have to convert this parameter to a beta angle.
4. Find more detailed flight data to validate the model with a higher certainty. With FUNcube, the validation shows that the general, approximate behaviour of the spacecraft can be estimated by the tool. However, due to many unknowns regarding the FUNcube hardware, as well as its exact attitude in orbit, it is not possible to validate data such as internal temperatures. Data from ground tests could also be helpful.
5. Adapt the orbital model to allow more general Kepler orbits, instead of only circular orbits. This would change the formulas for converting between true anomaly and time, the eclipse computation (due to eccentricity, the orbit may not be symmetrical anymore as seen from the Sun's perspective), and the Earth IR and albedo flux computation, since the angles of the cubesat with respect to the Earth's horizon change as well.
6. Optimise the code's integration scheme. Currently, some standard solutions from SciPy are used, but using other methods may reduce computational cost.
7. Add a functionality to give a node two surfaces with different optical properties. Currently, a node is assumed to have just one set of optical properties. However, in reality, different paints or coatings might be used. In the current version of the model, this could be realised by making two distinct nodes with different optical properties, with a very high conductance between them, connecting them

to simulate one node with two surfaces. The problem is that such a very high conductance can severely slow down the computation, since the time constant is reduced (see Equation 3.40). Some functionality that bypasses this, and just assumes that the two nodes have an identical temperature, might be a solution.

8. Replace the ECSS view factor-based approach for internal radiation by a Monte-Carlo Ray Tracing (MCRT) model. This would not only work for internal radiation, but might also allow to compute shadowing effects for the outside of the satellite (for example, in the case of deployed solar panels).
9. Similar to the plot showing the nodes, make a plot that shows the orbit. It can be good to get a visual confirmation of the orbit that was defined, especially if the future tool includes generic Kepler orbits.
10. Add a Graphical User Interface (GUI) to improve the user experience, and to minimise the effect of syntax errors by the user. It will improve the user experience, but also requires a high effort from the developer's side. This effort could be used on other, perhaps more important, aspects of the code.
11. Make it possible to define other shapes than only point masses or flat plates. This could be useful for slightly larger satellites with complicated geometry, but it will require a Monte-Carlo Ray Tracing implementation for the internal radiation computation.
12. Allow conductive couplings through a material to be computed 100% automatically by using the distance between nodes. In the current model, this distance is defined manually, but it can become time-consuming if the model has many nodes.
13. Implement Fault Detection, Isolation, and Recovery (FDIR) into the software, such that real-time flight data can be monitored more closely. It might not be an immediate implementation in the thermal analysis software, but it could be an improvement to the thermal subsystem as a whole.
14. Improve the user experience when entering an internal power that varies throughout time. Currently, the array must entered manually (except if the power is simply constant), but it would be easier if the array is automatically generated from a few assigned values, along with the times at which those powers should be assigned.
15. Implement a function to include, as a heat loss mechanism, Radio-Frequency (RF) power that is emitted by the communications subsystem. The emitted RF power is energy that is being sent away from the spacecraft, and thus acts as a heat loss mechanism. There is currently no standard function implemented to allow the user to add this to the model, although a similar effect could be reached by decreasing the internal power by the amount that is sent away by the communications system.
16. Consider adding a heat pipe simulation module, allowing to make (simplified) representations of heat pipes, such as the ones developed by H.S.B. Brouwer [66]. At the current technology readiness level, heat pipes are far away from being common use in SmallSats. However, if the interest changes, it might be worthwile to add such a functionality.
17. Expand the material database with more common materials, coatings, and contact connections.
18. Investigate the discrepancy between the Python and ESATAN computations for the albedo. Currently, the error varies between 5.8-23.4%, while the Earth IR and solar flux errors are much lower.

Table 7.1: Importance and simplicity matrix for recommendations for future work. This matrix is subjective in nature and should be tailored to the developer's personal skills and needs.

	solved in minutes	simple	doable	tricky	challenging
critical				4	
important				18	1
relevant		14, 17	2, 7	5	10
optional		9, 15	3, 12		8, 11
minor			6	13	16

References

[1] National Institute of Standards and Technology. “Stefan-boltzmann law.” (2022), [Online]. Available: https://physics.nist.gov/cgi-bin/cuu/Value?sigma (visited on 04/10/2024).

[2] International Astronomical Union, *Resolution b2: On the re-definition of the astronomical unit of length*, 2012.

[3] M. Langer and J. Bouwmeester, “Reliability of cubesats-statistical data, developers’ beliefs and the way forward,” *Small Satellite Conference*, 2016.

[4] T. Rühl, “Effective thermal testing and potential design solutions for pocketqube subsystems,” 2019.

[5] N. S. Bennett, A. Hawchar, and A. Cowley, “Thermal control of cubesat electronics using thermo-electrics,” *Applied Sciences*, vol. 13, no. 11, p. 6480, 2023.

[6] R. Avila de Luis, *Standardized thermal control solutions for pocketqubes*, 2019.

[7] C. Ruiz, *Research internship (pre) satellite thermal model for delfipq*, 2020. [Online]. Available: https://github.com/DelfiSpace/ThermalModel (visited on 02/12/2024).

[8] F. S. Meijering. “Gitlab: Delfispace thermal simulations.” (2024), [Online]. Available: https://gitlab.tudelft.nl/delfispace.

[9] F. S. Meijering. “Personal github page: Frankmeijering.” (2024), [Online]. Available: https://github.com/FrankMeijering.

[10] E. Kulu. “Nanosats database.” (Jun. 2022), [Online]. Available: https://www.nanosats.eu/ (visited on 02/12/2024).

[11] S. Speretta *et al.*, “Cubesats to pocketqubes: Opportunities and challenges,” in *Proceedings of the 67th International Astronautical Congress (IAC)*, IAF, 2016.

[12] R. P. Perumal, H. Voos, F. D. Vedova, and H. Moser, “Small satellite reliability: A decade in review,” 2021.

[13] M. Swartwout, “Secondary spacecraft in 2016: Why some succeed (and too many do not),” in *2016 IEEE Aerospace Conference*, IEEE, 2016, pp. 1–13.

[14] T. Villela, C. A. Costa, A. M. Brandão, F. T. Bueno, R. Leonardi, *et al.*, “Towards the thousandth cubesat: A statistical overview,” *International Journal of Aerospace Engineering*, vol. 2019, 2019.

[15] J. Guo, L. Monas, and E. Gill, “Statistical analysis and modelling of small satellite reliability,” *Acta Astronautica*, vol. 98, pp. 97–110, 2014.

[16] B. Yost *et al.*, “State-of-the-art small spacecraft technology,” NASA, Tech. Rep., 2023, pp. 202–222.

[17] Z. Zhang, “Orbital temperature model construction for delfi-pq,” 2022.

[18] J. Bouwmeester, A. Menicucci, and E. K. Gill, “Improving cubesat reliability: Subsystem redundancy or improved testing?” *Reliability Engineering & System Safety*, vol. 220, p. 108 288, 2022.

[19] S. A. Jacklin, “Small-satellite mission failure rates,” Tech. Rep., 2019.

[20] DelfiSpace. “Delfispace on twitter.” (Jan. 2024), [Online]. Available: https://twitter.com/DelfiSpace/status/1745588651545837723 (visited on 02/12/2024).

[21] S. Speretta. “Delfi-pq - grafana.” (2024), [Online]. Available: https://dashboard.satnogs.org/d/0-wbcw1nk/delfi-pq?orgId=1 (visited on 08/03/2024).

[22] E. Hoffman. “Fault-detection, fault-isolation and recovery (fdir) techniques.” (1994), [Online]. Available: https://llis.nasa.gov/lesson/839 (visited on 02/28/2024).

[23] L. Kirschenbaum, “A generic spacecraft fdir system,” in *2021 IEEE Aerospace Conference (50100)*, IEEE, 2021, pp. 1–7.

[24] A. Bowman. “What are smallsats and cubesats?” (Aug. 2023), [Online]. Available: https://www.nasa.gov/what-are-smallsats-and-cubesats/#:~:text=CubeSats%20are%20a%20class%20of,%2C%206%2C%20and%20even%2012U. (visited on 02/12/2024).

[25] D. Gilmore, *Spacecraft Thermal Control Handbook, Volume I: Fundamental Technologies*. American Institute of Aeronautics and Astronautics, Inc., Dec. 2002, vol. I, ISBN: 978-1-884989-11-7. DOI: 10.2514/4.989117.

[26] N. V. Pavlyukevich and Y. V. Polezhaev. “Accommodation coefficient.” (2011), [Online]. Available: https://www.thermopedia.com/content/286/ (visited on 03/18/2024).

[27] J. Bouwmeester *et al.*, “Utility and constraints of pocketqubes,” *CEAS Space Journal*, vol. 12, pp. 573–586, 2020.

[28] J. R. Wertz, D. F. Everett, and J. J. Puschell, “Space mission engineering: The new smad,” *(No Title)*, 2011.

[29] J. Van Es, T. van den Berg, A. van Vliet, T. Ganzeboom, H. Brouwer, and S. Elvik, “Mini mechanically pumped loop modelling and design for standardized cubesat thermal control,” *50th International Conference on Environmental Systems*, vol. 294, 2020.

[30] L. Anderson, C. Swenson, A. Mastropietro, and J. Sauder, “The active thermal architecture: Active thermal control for small-satellites,” in *2021 SmallSat Technology Partnerships Technology Exposition*, 2021.

[31] ECSS, “Thermal analysis handbook,” *Noordwijk, The Netherlands, ESA Requirements and Standards Division*, 2016.

[32] ECSS, *Space engineering thermal design handbook-part 1: View factors*, 2011.

[33] P. Reiss, *New methodologies for the thermal modelling of cubesats*, 2012.

[34] N. van der Pas, *Spacecraft thermal design w2.2 - heat exchanges 1*, Lecture Slides, 2023.

[35] H. G. Isik, A. B. Uygur, C. Omur, and E. Solakoglu, “The thermal analysis of a satellite by an in-house computer code based on thermal network method and monte carlo ray tracing technique,” 2011, pp. 978–983.

[36] S. Amberger, S. Pirker, and C. Kloss, *Thermal radiation modeling using ray tracing in liggghts*, 2013.

[37] N. van der Pas, *Spacecraft thermal design w3.2 - thermal analysis 1*, Lecture Slides, 2023.

[38] T. Van Boxtel, “Thermal modelling and design of the delffi satellites,” 2015.

[39] R. Stevens, “Digital twin for spacecraft concepts,” in *2023 IEEE Aerospace Conference*, IEEE, 2023, pp. 1–7.

[40] W. Yang, Y. Zheng, and S. Li, “Application status and prospect of digital twin for on-orbit spacecraft,” *IEEE Access*, vol. 9, pp. 106 489–106 500, 2021.

[41] B. Mattos. “Cubesat thermal power toolbox.” (2023), [Online]. Available: https://github.com/mattost14/CubeSat-Thermal-Power-App (visited on 02/28/2024).

[42] K. F. Wakker, *Fundamentals of Astrodynamics*. Institutional Repository TU Delft, 2015.

[43] D. R. Williams. “Nasa earth fact sheet.” (Jan. 2024), [Online]. Available: https://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html (visited on 04/09/2024).

[44] F. Espenak. “Earth at perihelion and aphelion: 2001 to 2100.” (Jan. 2021), [Online]. Available: http://astropixels.com/ephemeris/perap2001.html (visited on 04/09/2024).

[45] C. A. Gueymard, “A reevaluation of the solar constant based on a 42-year total solar irradiance time series and a reconciliation of spaceborne observations,” *Solar Energy*, vol. 168, pp. 2–9, 2018.

[46] R. Peyrou-Lauga, “Using real earth albedo and earth ir flux for spacecraft thermal analysis,” 47th International Conference on Environmental Systems, 2017.

[47] S. L. Rickman. “Introduction to on-orbit thermal environments part ii.” (Sep. 2011), [Online]. Available: https://nescacademy.nasa.gov/video/db97047cadcb4c2994d59022b7c2166a1d (visited on 04/09/2024).

[48] G. L. Smith and D. A. Rutan, “The diurnal cycle of outgoing longwave radiation from earth radiation budget experiment measurements,” *Journal of the atmospheric sciences*, vol. 60, no. 13, pp. 1529–1542, 2003.

[49] Wikipedia. “Spherical law of cosines.” (2024), [Online]. Available: https://en.wikipedia.org/wiki/Spherical_law_of_cosines (visited on 04/09/2024).

[50] E. Anderson and L. Clark, “Geometric shape factors for planetary-thermal and planetary-reflected radiation incident upon spinning and nonspinning spacecraft,” Tech. Rep., 1965.

[51] S. L. Rickman. “Introduction to on-orbit thermal environments part iii.” (Sep. 2011), [Online]. Available: https://nescacademy.nasa.gov/video/ec22d75a55464d3798fb25cb1d4eab3a1d (visited on 04/15/2024).

[52] F. Espenak. “Solstices and equinoxes: 2001 to 2100.” (Feb. 2018), [Online]. Available: http://www.astropixels.com/ephemeris/soleq2001.html (visited on 04/24/2024).

[53] Wikipedia. “Right ascension.” (2023), [Online]. Available: https://en.wikipedia.org/wiki/Right_ascension (visited on 04/09/2024).

[54] SciPy. “Scipy.spacial.transform.rotation.” (2024), [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.transform.Rotation.html (visited on 04/15/2024).

[55] T. M. Essinger-Hileman. “Cosmic background explorer.” (Nov. 1989), [Online]. Available: https://lambda.gsfc.nasa.gov/product/cobe/ (visited on 04/26/2024).

[56] SciPy. “Scipy.integrate.solve_ivp.” (2024), [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html (visited on 04/20/2024).

[57] W. H. Press, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[58] Wikipedia. “Thermal time constant.” (2024), [Online]. Available: https://en.wikipedia.org/wiki/Time_constant#Thermal_time_constant (visited on 04/09/2024).

[59] R. Limebear, “The amsat-uk funcube handbook,” AMSAT-UK, Tech. Rep., 2013.

[60] AMSAT-UK. “Funcube 1: Amsat-uk data warehouse.” (2024), [Online]. Available: http://warehouse.funcube.org.uk/ui/fc1-fm (visited on 08/13/2024).

[61] AMSAT-UK. “Funcube-1 (ao-73) satellite.” (2012), [Online]. Available: https://amsat-uk.org/funcube/funcube-cubesat/ (visited on 09/10/2024).

[62] N2YO.com. “Funcube 1 (ao-73).” (2024), [Online]. Available: https://www.n2yo.com/satellite/?s=39444 (visited on 08/13/2024).

[63] T. Liu, Q. Sun, J. Meng, Z. Pan, and Y. Tang, “Degradation modeling of satellite thermal control coatings in a low earth orbit environment,” *Solar Energy*, vol. 139, pp. 467–474, 2016.

[64] S. Alnaqbi, D. Darfilal, and S. S. M. Swei, “Propulsion technologies for cubesats,” *Aerospace*, vol. 11, no. 7, p. 502, 2024.

[65] A. Anvari, F. Farhani, and K. Niaki, “Comparative study on space qualified paints used for thermal control of a small satellite,” *Iranian Journal of Chemical Engineering*, 2009.

[66] H. S. B. Brouwer, “Performance characterization of water heat pipes and their application in cubesats,” 2016.

[67] G. Kopp and J. L. Lean, “A new, lower value of total solar irradiance: Evidence and climate significance,” *Geophysical Research Letters*, vol. 38, no. 1, 2011.

[68] Airbus Netherlands B.V. “Hiper flexlinks.” (2023), [Online]. Available: https://www.airbusdefenceandspacenetherlands.nl/products/hiper-flexlinks/ (visited on 02/19/2024).

[69] M. T. Barako *et al.*, “Evaluating variable-emissivity surfaces for radiative thermal control,” *Journal of Thermophysics and Heat Transfer*, vol. 36, no. 4, pp. 1003–1014, 2022.

[70] DNP Group. “The basics of heat pipes – their history, principle, and varieties explained.” (2022), [Online]. Available: https://www.global.dnp/biz/column/detail/10162360_4117.html (visited on 02/21/2024).

[71] B. Matonak and H. Peabody, “An Introductory Guide to Variable Conductance Heat Pipe Simulation,” Tech. Rep., 2006.

[72] J.-P. Collette *et al.*, “Phase change material device for spacecraft thermal control,” in *62nd International Astronautical Congress 2011*, 2011.

[73] Rubitherm GmbH. “Phase change material.” (2024), [Online]. Available: https://www.rubitherm.eu/en/ (visited on 02/22/2024).

[74] Minco. “Flexible heaters.” (2024), [Online]. Available: https://www.minco.com/products/flexible-heaters/ (visited on 02/22/2024).

[75] Clayborn Lab. “Heat tape overview.” (2023), [Online]. Available: https://claybornlab.com/heat_tape_overview.html (visited on 02/22/2024).

[76] PCF Electronics. “Thermoelectric cooler peltier element 12706 tec.” (2024), [Online]. Available: https://www.pcfelectronics.nl/nl/thermoelectric-cooler-peltier-element-12706-tec.html (visited on 02/22/2024).

[77] J. Olson, “Cubesat-sized space microcryocooler,” *Small Satellite Conference*, 2019.

[78] MatWeb. “Online materials information resource.” (2024), [Online]. Available: https://www.matweb.com/ (visited on 07/04/2024).

[79] Wikipedia. “Fr-4.” (2024), [Online]. Available: https://en.wikipedia.org/wiki/FR-4 (visited on 07/04/2024).

[80] Z. M. Peterson. “Fr4 thermal properties to consider during design.” (2020), [Online]. Available: https://www.nwengineeringllc.com/article/fr4-thermal-properties-to-consider-during-design.php (visited on 07/04/2024).

[81] AzurSpace, *30% triple junction gaas solar cell assembly*. [Online]. Available: https://www.azurspace.com/images/products/0003401-01-01_DB_3G30A.pdf (visited on 07/04/2024).

[82] J. H. Henninger, “Solar absorptance and thermal emittance of some common spacecraft thermal-control coatings,” NASA, Tech. Rep., 1984.

[83] E. D. F. da Silva and C. Ezio, “Experimental determination of the effective thermal properties of a multi-layer insulation blanket,” in *22nd International Congress of Mechanical Engineering (COBEM 2013) Ribeirão Preto, SP, Brazil*, 2013.

[84] VERTEX AEROSPACE. “Multi-layer insulation (mli).” (2021), [Online]. Available: https://www.thermalengineer.com/library/effective_emittance.htm (visited on 07/04/2024).

[85] Transmetra, *Table of emisivity of various surfaces*. [Online]. Available: https://www.transmetra.ch/images/transmetra_pdf/publikationen_literatur/pyrometrie-thermografie/emissivity_table.pdf (visited on 07/04/2024).

[86] D. Johnson. “Table of absorptivity and emissivity of common materials and coatings.” (2017), [Online]. Available: http://www.solarmirror.com/fom/fom-serve/cache/43.html (visited on 07/04/2024).

[87] V. Gorev, A. Kozlov, V. Y. Prokopyev, Y. M. Prokopyev, A. Stuf, and A. Sidorchuk, “The effect of the pcb solder mask type of the hull outer surface of the cubesat 3u on its thermal regime,” in *IOP Conference Series: Materials Science and Engineering*, IOP Publishing, vol. 734, 2020, p. 012 027.

[88] F. Lucia, “Development of a tool for thermal analysis of small spacecrafts.,” Ph.D. dissertation, Politecnico di Torino, 2023.

[89] A. S. Sabau, “Review of thermal contact resistance of flexible graphite materials for thermal interfaces in high heat flux applications,” US DEPARTMENT OF ENERGY, Tech. Rep., 2022.

[90] R. van Gils, R. Olieslagers, M. Mirsadeghi, and J. Oosterhuis, *Thermal contact conductance in vacuum*, Mar. 2022.

A

Thermal Control Technologies

This appendix shows some of the most common thermal control technologies, for SmallSats specifically. The following information may be relevant for readers without much prior knowledge on spacecraft thermal control, and it can provide a better understanding of the rest of this thesis.

A.1. Thermal Control Technologies

Thermal control technologies are generally divided into two categories: passive and active control. The former employs thermal management which does not require any input power, while the latter does require input power. The most common technologies are shown in Figure A.1; this listing is based on [16]. A number of technologies is listed below; methods which transport heat within the spacecraft are marked as ”internal transport”, and methods which exchange heat with the space environment are marked as ”external transport”.

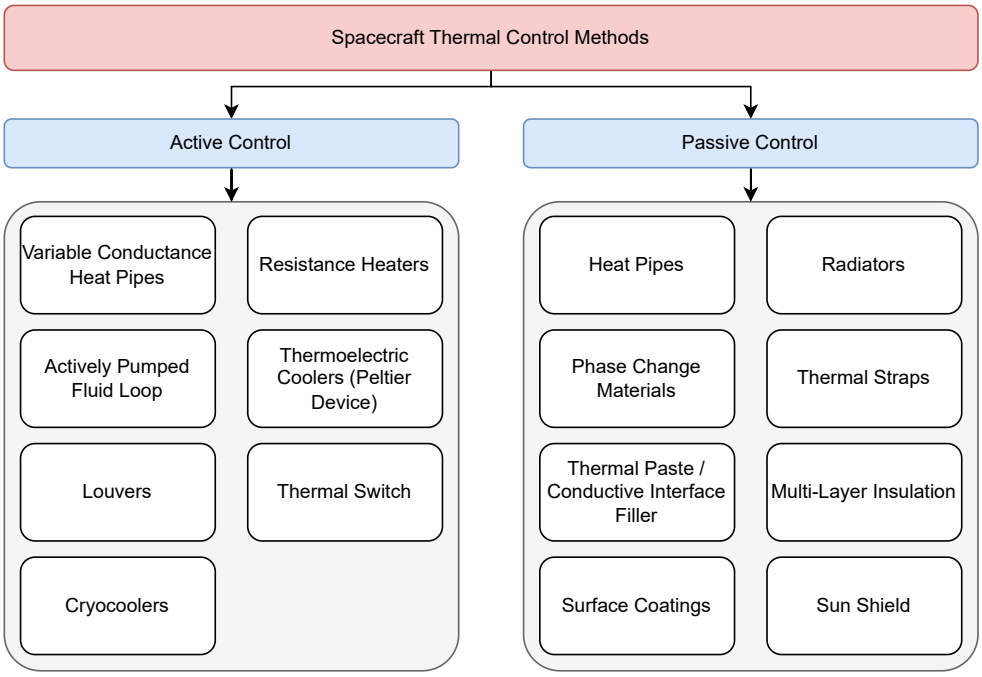


Figure A.1: Most common active and passive thermal control methods (based on [16]).

A.1.1. Passive Thermal Control Technologies

Passive thermal control is achieved by altering the satellite’s design in such a way that the desired thermal behaviour is reached, without requiring any power input [16]. Some of the technologies mentioned below also have active counterparts (e.g., heat pipes can be both active or passive, which is explained in the section on heat pipes).

Thermal Straps (Internal Transport)
Thermal straps consist of a highly conductive material which improves the conductance between two desired locations. Such straps can be both rigid or flexible, and employ materials such as copper, aluminium, or Pyrolytic Graphite Sheet (PGS) [16]. They are a passive heat transport method, since the heat flow originates from a temperature differential between the two outer edges. A common use case is to connect hot internal components to the radiator via such a thermal strap, to allow the heat to flow from the hot component to the radiator, and then to space. An example is the HiPeR Flexlink from Airbus Netherlands, as shown in Figure A.2.

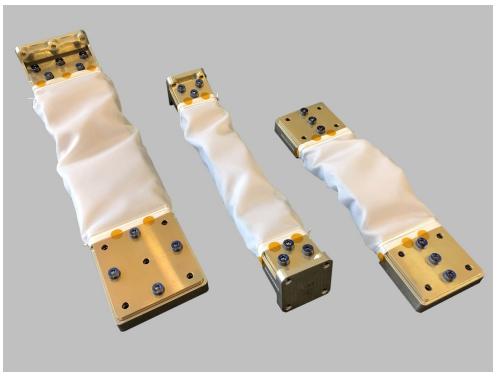


Figure A.2: HiPeR Flexlinks from Airbus Netherlands B.V. [68]. The inner conductive material is PGS, and the outer sleeve prevents any loose PG particles from escaping.

Radiators and Coatings (External Transport)
Radiators are used to expel heat from the spacecraft, which is why they are highly emissive in the infrared (IR) wavelengths of the spacecraft and not very absorptive in the visible wavelengths of the Sun. They are mainly used when the spacecraft has such a high internal power generation, that the heat must be released to space as efficiently as possible [16]. For low internal powers, such as most PocketQubes, the issue may not be as prevalent. Instead, coatings can be applied to the outer (or inner) surface to either retain more heat or reject more heat.

An issue with SmallSats is that there is not always enough space on the outside of the spacecraft to fit a radiator or apply coatings, since the solar cells can be body-mounted. An example of such a satellite with body-mounted solar cells is shown in Figure A.3, and a satellite with deployable solar panels is shown in Figure A.4. The first has almost no area available which can be designed for thermal control (i.e., the optical properties of the solar cells cannot be changed); the second has significantly more area exposed on the spacecraft bus, which can be altered for thermal control as desired.

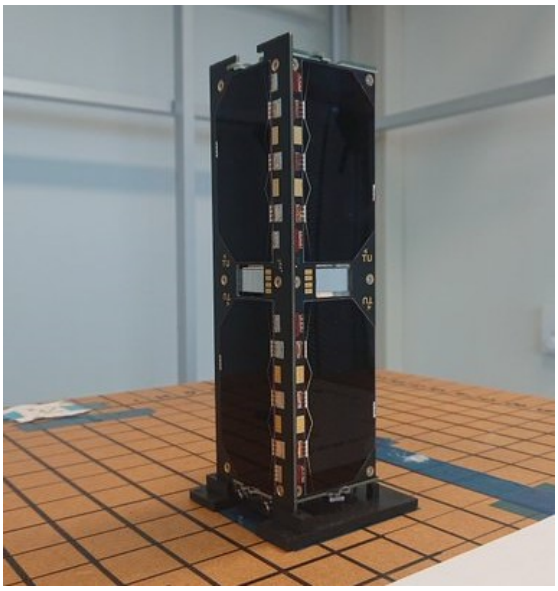


Figure A.3: Delfi-PQ PocketQube by TU Delft [10].

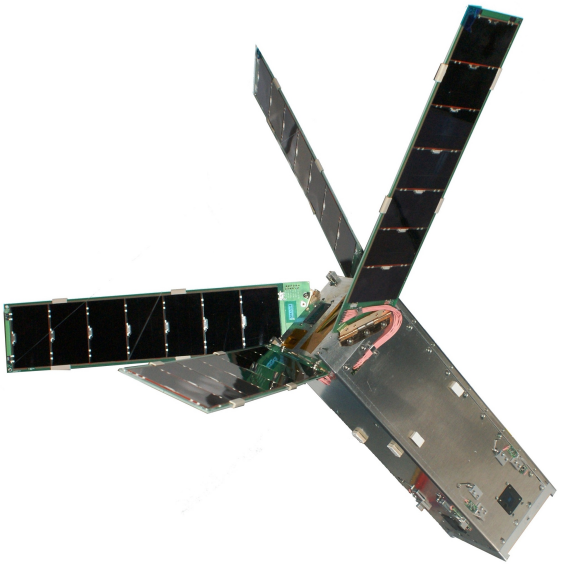


Figure A.4: Delfi-n3Xt CubeSat by TU Delft [10].

A way to deal with the limited area on the spacecraft available for radiator surfaces, is to use deployable radiators [16]. This is only really necessary when the spacecraft has a high internal power. Advantages of deployable radiators are the higher possible internal power of the spacecraft (although the solar arrays should also scale accordingly), and in some cases the radiator could be stowed/deployed on command when less/more heat is to be rejected. Disadvantages of deployable radiators are the added mass and complexity to the system [16].

Since many SmallSats are freely tumbling to some extent, such as the Delfi satellites, the optical properties of the spacecraft's outer surface cannot be tailored to a specific irradiation condition, but they must be designed based on average conditions. Since the requirements for Sun-facing surfaces and non-Sun-facing surfaces differ, this results in suboptimal thermal control. Additionally, the constant change from eclipse to non-eclipse conditions also forces the thermal design to be based on the time-average result. There have been initiatives to solve these problems, in the form of variable emissivity materials (VEMs). There are two common types of VEMs: materials that change optical properties when an electric current is applied, and materials that change optical properties when the temperature changes. The former is a category of active control, requiring a power input. The latter is passive control, in the form of negative thermal feedback: the emissivity increases when the spacecraft is hot, and it decreases when the spacecraft is cold [69].

Heat Pipes (Internal Transport)

Heat pipes are hollow pipes with an internal liquid, that transports heat by means of a liquid-gas phase change. Figure A.5 shows how the evaporator side receives heat, resulting in the internal fluid to evaporate, and move in the gaseous phase to the other side of the heat pipe. That side is the condenser, where heat is rejected by the gas condensing. The liquid moves along the wick, see Figure A.6, by means of capillary force. The heat pipe can transfer heat in both directions, depending on the temperatures of both ends.

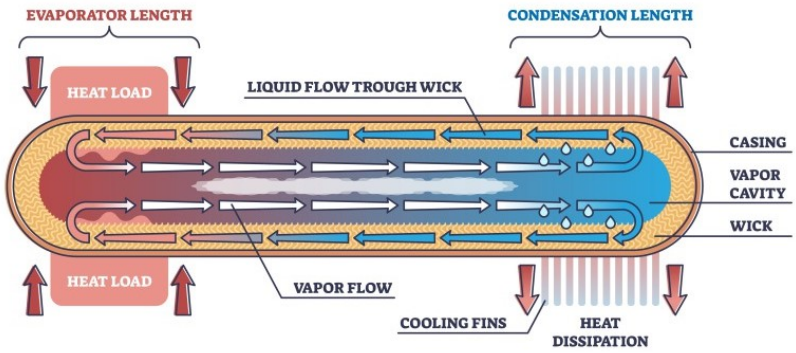


Figure A.5: Schematic diagram of the operating principle of a constant conduction heat pipe [70].

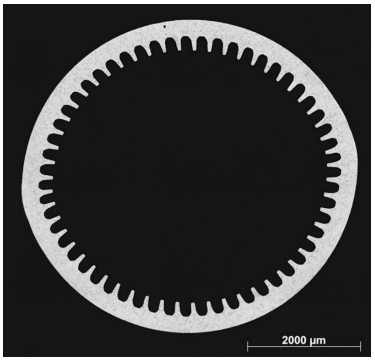


Figure A.6: Example of an internal wick structure of a heat pipe [66].

An alternative to the constant conductance heat pipe (CCHP) shown before is the variable conductance heat pipe (VCHP). Such a heat pipe includes a reservoir with a Non-Condensable Gas (NCG) which is used to suppress the condenser. This works by heating the NCG, thereby expanding it to occupy the condenser, hence blocking part of the condenser. An active control loop can be used in this way to tweak the conductance of the heat pipe [71].

Phase Change Materials

Phase Change Materials (PCMs) do not transport heat from one location to another, but they are energy storage devices. The PCM alternates between the solid and liquid phase, where heat is absorbed when melting and released when solidifying. The phase change allows heat absorption and rejection at (nearly) constant temperature, which is shown in Figure A.7. The result is a material that can store large amounts of heat for a small mass, while staying in a tight temperature envelope [72].

The main issue with PCMs is their complexity. Although the concept appears simple, there are many difficulties associated with designing a PCM module, such as: limited cycling stability; sensitivity to spacecraft temperature (i.e., PCMs are ineffective if the spacecraft has a different temperature than expected, which can occur in SmallSats); complex to model analytically due to multi-phase behaviour [72]. The severity of these issues depends on the type of PCM, but generally, paraffin (wax) is preferred. Such a paraffin is shown in Figure A.8.

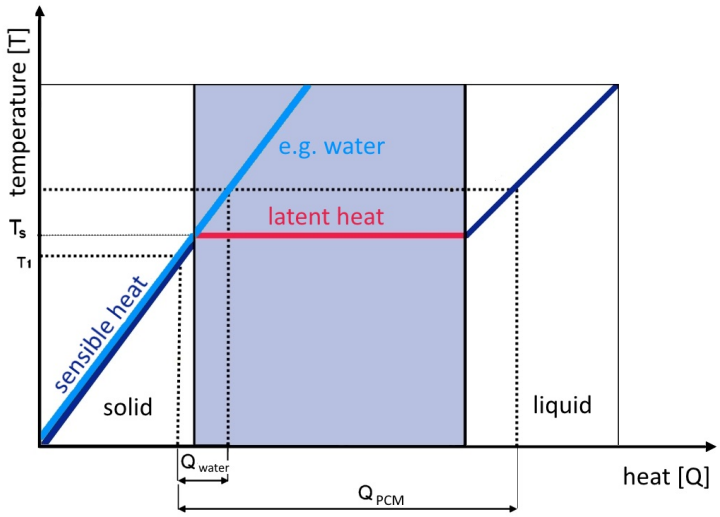


Figure A.7: Heating properties of phase change materials [73].

Figure A.8: Paraffin by the company Rubitherm GmbH [73].

A.1.2. Active Thermal Control Technologies

Active thermal control is achieved by using electrical power to change the spacecraft's thermal behaviour. This is less common in SmallSats due to their limited power budget, often related to having body-mounted solar cells [30, 16]. However, it can be good to investigate the potential of active control, since it could allow for more complex payloads that operate in tight temperature envelopes.

Resistance Heaters

Resistance heaters are the most simple type of active thermal control device, using power to heat up a wire. They are often used with batteries to prevent the battery temperature from dropping too much [16]. Because of the low (thermal) mass of SmallSats, a low power often suffices to keep the temperatures as desired [16].

There are many Commercial-Off-The-Shelf (COTS) heaters available. Some example companies are Minco [74], and Clayborn Lab [75]. The former makes standardised flexible heaters, see Figure A.9, while the latter makes heating tape which can be cut and soldered by the user as desired, see Figure A.10.

Figure A.9: Flexible resistance heater from Minco [74].

Figure A.10: Flexible, customisable resistance heater by Clayborn Lab [75].

Thermo-Electric Coolers (Peltier Devices)

Thermo-Electric Coolers (TEC) are solid-state heat pumps that absorb heat on one end and reject heat on the other when an electric current is applied [16, 5]. A TEC is shown in Figure A.11. They can be beneficial over fluid loops, as they have no moving parts [5] and are significantly smaller.

A complication with TECs is that they generate heat themselves due to the fact that they use input power. When the heat load to be removed is low, this does not pose any issues, but when the heat load is high, the TEC uses so much power that it has a net negative effect on the spacecraft temperature [5].

Figure A.11: Thermo-Electric Cooler [76].

Mechanically Pumped Fluid Loops
Mechanically Pumped Fluid Loops (MPFL) use convective heat transfer to achieve large heat fluxes [16, 29]. MPFLs are uncommon in PocketQubes since there is no need for tens of Watts of heat flow, and insufficient volume to fit a MPFL system inside. For larger CubeSats, however, MPFLs may be beneficial [29]. An example of such a miniaturised MPFL is shown in Figure A.12.

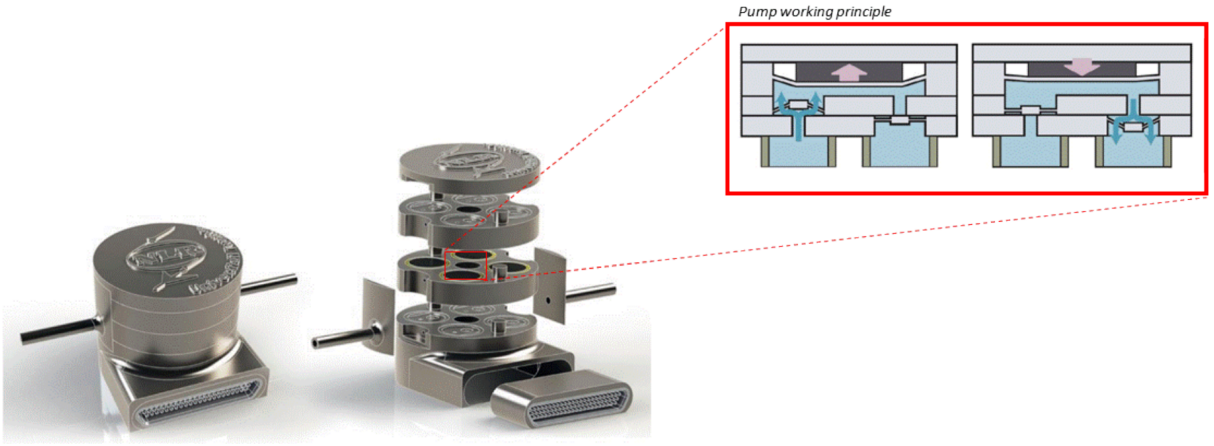


Figure A.12: Miniaturised MPFL by NLR, occupying a volume less than 1U [29].

Cryocoolers
A cryocooler is a refrigeration device that uses a cryogenic fluid, such as helium, to cool to temperatures near 100 K and below [16, 77]. They are only used if they are strictly needed to cool high-precision instruments, and if the mass, volume, and power budgets allow. A micro-cryocooler by Lockheed Martin is shown in Figure A.13.

Figure A.13: Micro-cryocooler by Lockheed Martin [16].

B

Material Database

This appendix shows the materials as they appear in the material database CSV file, including the sources used for all values. This file can also be found on GitLab [8] and GitHub [9]. The file on GitLab contains all the references used to find the values, and an additional column is present with arbitrary notes about that particular material, value, or source. An example of what the actual CSV file appears like is shown in Figure B.1.

Table B.1: ”Materials” section of the material database from GitLab [8].

name	density [kg/m3]	ref	Specific heat capacity [J/(kg.K)]	ref	Thermal conductivity ”k” [W/(m.K)]	ref
al2024	2780	[78]	875	[78]	121	[78]
al5052	2680	[78]	880	[78]	138	[78]
al6061	2700	[78]	896	[78]	167	[78]
al7075	2810	[78]	960	[78]	130	[78]
stainless_steel	8000	[78]	500	[78]	18.5	[78]
copper	8930	[78]	385	[78]	400	[78]
PCB	1850	[79]	1100	[80]	30	[4]
solar_cells	4200	[81]	493	[38]	56.7	[38]

Table B.2: ”Optical properties” section of the material database from GitLab [8].

name	Solar absorptivity [-]	ref	IR emissivity [-]	ref
black_paint	0.95	[25, 82]	0.85	[25, 82]
white_paint	0.25	[25, 82]	0.90	[25, 82]
MLI	0.005	[83]	0.05	[84]
al_unpolished	0.15	[25]	0.09	[85]
al_polished	0.15	[25]	0.05	[25, 85, 86]
al_anodised	0.14	[86]	0.70	[85, 86]

steel_unpolished	0.47	[25]	0.14	[25]
steel_polished	0.42	[25]	0.11	[25]
copper	0.30	[25]	0.03	[25, 85]
PCB_white	0.21	[87]	0.96	[87]
PCB_black	0.94	[87]	0.97	[87]
PCB_red	0.51	[87]	0.88	[87]
PCB_green	0.88	[88]	0.70	[88]
PCB_blue	0.89	[88]	0.90	[88]
solar_cell	0.66	[38]	0.89	[38]
solar_cell _mix_black _paint	0.75	[25, 38, 82]	0.88	[38]

Table B.3: ”Contact connections” section of the material database from GitLab [8].

name	**Heat transfer coefficient [W/(m2.K)]**	**ref**
graphite	50000	[89]
al_al	2000	[90]
steel_steel	1000	[90]
al_steel	1000	[90]
PCB_PCB_ perpen-dicular	188	[79]

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
23	name (no spaces)	density [kg/m^3]	ref	Specific heat capacity [J/(kg.K)]	ref	Solar absorptivity [-]	ref	IR emissivity [-	ref	Thermal conductivity "k" [W/(m.K)	ref	Heat transfer coefficient [W/(m^2.K)]	ref	notes
24														
25	MATERIALS-BEGIN													
26	al2024	2780	1	875	1					121	1			
27	al5052	2680	1	880	1					138	1			
28	al6061	2700	1	896	1					167	1			
29	al7075	2810	1	960	1					130	1			
30	stainless_steel	8000	1	500	1					18.5	1			
31	copper	8930	1	385	1					400	1			
32	PCB	1850	7	1100	8					30	18			PCBs are not o
33	solar_cells	4200	11	493	10					56.7	10			Mass of solar c
34	MATERIALS-END													
35														
36														
37														
38														
39	OPTICAL-BEGIN													
40	black_paint					0.95	2, 5	0.85	2, 5					
41	white_paint					0.25	2, 5	0.9	2, 5					
42	MLI					0.005	15	0.05	14					less effective o
43	al_unpolished					0.15	2	0.09	3					
44	al_polished					0.15	2	0.05	2, 3, 6					
45	al_anodised					0.14	6	0.7	3, 6					0.7 is average b
46	steel_unpolished					0.47	2	0.14	2					
47	steel_polished					0.42	2	0.11	2					
48	copper					0.3	2	0.03	2, 3					
49	PCB_white					0.21	16	0.96	16					
50	PCB_black					0.94	16	0.97	16					
51	PCB_red					0.51	16	0.88	16					
52	PCB_green					0.88	17	0.7	17					sources 16 and
53	PCB_blue					0.89	17	0.9	17					
54	solar_cell					0.66	10	0.89	10					absorptivity is
55	solar_cell_mix_black_paint					0.75	2, 5, 10	0.88	10					a surface havir
56	OPTICAL-END													
57														
58	CONTACT-BEGIN													
59	graphite											50000	13	highly depende
60	al_al											2000	12	highly depende
61	steel_steel											1000	12	highly depende
62	al_steel											1000	12	highly depende
63	PCB_PCB_perpendicular											188	7	"equivalent" he
64	CONTACT-END													

Figure B.1: Example view of the material database CSV file. The statements ”MATERIALS-BEGIN” etc. are used by the Python code to recognise where the materials, optical properties, and contact connections are to be found.

C

Software User Manual

This appendix contains a user manual which specifically aims to elaborate on the user-end of the thermal analysis tool. Computational methods etc. will not be repeated here, since they have been thoroughly discussed in the main body of this thesis. The source code of the software can be found on the GitLab page hosted by TU Delft [8], and an alternative repository is also present on the author's personal GitHub page [9]. It is recommended to refer to the official GitLab page.

A simple example using the FUNcube-1 satellite was already given in Section 3.5. Some information in this appendix might be a repetition of that example case, but the information in this appendix is more thorough and more focused on demonstrating all the different ways to use the software.

Section C.1 gives a succinct overview of the software, expressing the general idea and work flow. Next, Section C.2 demonstrates all the ways of defining an orbital model. On the other hand, Section C.3 demonstrates how to define a nodal model. In Section C.4, methods on how to solve and plot the model will be shown. The sensitivity analysis tool is explained in Section C.5. Finally, a lot of information is present in the docstrings within the Python code; these are presented in Section C.6.

C.1. General Overview and Workflow

This section will give a brief overview from the user's perspective of the software. More details are given in the upcoming sections, but especially if the user uses this software for the first time, it is recommended to read this section to acquire a general understanding of the workflow.

C.1.1. Simplified File Overview

A detailed file overview is given in the last section of this user manual, in Section C.6. However, for simplicity, this sub-section will describe which files are the most important to the user, and why. The following files are present in GitLab [8], and the following order is from most to least "relevant" for the average user:

- **ThermalBudget.py**: contains the definition of the Node, NodalModel, and OrbitalModel classes. It is important to understand how to use them, and the documentation present in those class definitions helps a lot with input syntax and troubleshooting.
- **CommonNodalModels.py**: contains some functions to easily define PCBs with multiple nodes. It is recommended to expand this file by adding more and more NodalModels that are frequently reused.
- **SensitivityAnalysis.py**: contains the necessary functions to perform a sensitivity analysis such as presented in Chapter 5.
- **Materials.csv**: is the only CSV file, and this file contains the material database. All values can be viewed or adapted here, and new materials can be added as well.
- **FUNcube.py**: this file is not directly used, but it can serve as an example that the user can copy and adapt to fit their own model.

- **Constants.py**: Contains universal constants required for various computations. They do not cross the user's path, but are indirectly used in the other files. They may be adapted by the user, but it is recommended to only do so if strictly needed. Each number has a source (url) from which it was taken.
- **VerificationValidation.py**: contains numerous thermal and orbital models, for the purpose of performing verification and validation. Results were shown in Chapter 4.
- **EnvironmentRadiation.py**: computes the orbit and environmental heat fluxes. However, the file is not directly accessed by the user, but indirectly via the OrbitalModel object defined in ThermalBudget.py.
- **Materials.py**: defines the Material, Coating, and Contact objects. However, these are likely never used by the user, but only used indirectly via the Node and NodalModel objects in ThermalBudget.py.
- **TestFile.py**: may be useful to test some things without messing with important other files.

Moreover, some additional folders are present in the GitLab repository:

- **ESATAN**: folder use to store thermal simulations made with ESATAN-TMS. Only used for verification purposes.
- **FUNcubeData**: folder with FUNcube-1 flight data. Only used for validation purposes.
- **SensitivityAnalysis**: folder which is used as a default to store results from any sensitivity analyses performed. The stored files are Python Pickle files, and contain NodalModel objects that have been solved, i.e., that contain simulation results.

When the user defines their own models, it is recommended to create a completely new file. This requires the following import statements (not all of them might be needed). Running the function `show_available_materials` is optional, but it provides a summary of all materials available in the database, without having to access the CSV file. It shows the material names as in Figure C.1 (the image is cropped and the list extends further down).

```
from ThermalBudget import Node, NodalModel, OrbitalModel, show_available_materials
from CommonNodalModels import make_5node_pcb, make_9node_pcb, assign_q_ext_to_pcb
from SensitivityAnalysis import run_all, run_variable, run_dt_rot, run_intrad, run_integrators,
    plot_all, plot_variable, plot_dt_rot, plot_intrad, plot_integrators

show_available_materials()  # optional
```

```
AVAILABLE MATERIALS:
al2024
al5052
al6061
al7075
stainless_steel
copper
PCB
solar_cells

AVAILABLE COATINGS:
black_paint
white_paint
MLI
```

Figure C.1: Example print statement in the console after calling `show_available_materials()`. These material names can be copy-pasted into Node assignments.

C.1.2. Modelling Workflow

Detailed functional diagrams have been shown in Section 3.2, so they will not be repeated here. However, a more user-focused overview is presented in Figure C.2, which shows the possible ways in which the software can be used. It is a highly simplified diagram, but it highlights the fact that the orbital and nodal model are in principle independent of each other. However, the orbital model

and nodal model can be linked by entering existing Node objects into the orbital model, and by entering the OrbitalModel's heat fluxes and time array (explained in the next sub-section) into the NodalModel.

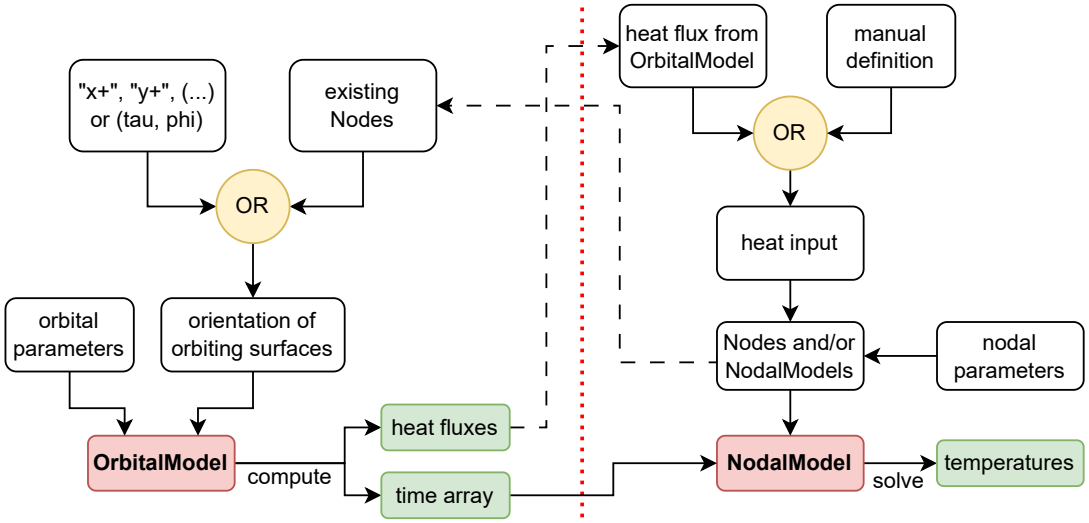


Figure C.2: Simplified code overview from the user's perspective. The left half of the diagram is related to the orbital model, while the right half is related to the nodal model. The dashed lines are possible interconnections between the two.

When both dashed lines in Figure C.2 are used, a chicken-and-egg problem appears to arise. The heat fluxes and time array from the orbital model are required for the nodal model, but the nodes from the nodal model are used as orbiting surfaces for the orbital model. To avoid such a situation, there are three slightly different modelling methods. There are likely more methods, but the ones mentioned here have been tested extensively. Actual code examples are given below as well.

1. **Example 1: manual surface definition**: First, the surfaces are defined using "x+", "y+", ... as identifiers, or sets of (τ, ϕ) angles as in Figure 3.8 (repeated below for convenience, Figure C.4). More details are given in Section C.2. The OrbitalModel is run, and the heat fluxes and time array are stored in the OrbitalModel object. Next, the Nodes and NodalModel can be created, using the heat fluxes and time array from the OrbitalModel as inputs.

```
surf_list = ['x+']

MyOrbitalModel = OrbitalModel(surfaces=surf_list, (...))
MyOrbitalModel.compute()
t = MyOrbitalModel.t

xplusA = Node(name='x+A', (...), q_ext=MyOrbitalModel.get_heat('x+'))  # example node that
    receives external heat
nodeB = Node(name='nodeB', (...))  # example node that does not receive external heat

MyNodalModel = NodalModel(t=t, title='MyNodalModel')
MyNodalModel.add_node([xplusA, nodeB])
MyNodalModel.connect(xplusA, nodeB, (...))
# (...)
```

2. **Example 2: using Nodes as surface definition**: First, the outer Nodes (i.e., the ones that receive environmental radiation) are defined, without any heat fluxes assigned. Next, the OrbitalModel is defined, using the Nodes as inputs. The Nodes must include either of "x+", "y+", "z+", "x-", "y-", "z-" in their names, corresponding to Figure 3.7 (repeated below for convenience, Figure C.3). The OrbitalModel is run, and the heat fluxes are automatically stored in the Node objects (and also in the OrbitalModel object). Next, the other Nodes and main NodalModel are created using the time array from the OrbitalModel as input.

```
xplusA = Node(name='x+A', (...))  # example node that receives external heat

MyOrbitalModel = OrbitalModel(surfaces=[xplusA], (...))
MyOrbitalModel.compute()
t = MyOrbitalModel.t

nodeB = Node(name='nodeB', (...))  # example node that does not receive external heat

MyNodalModel = NodalModel(t=t, title='MyNodalModel')
MyNodalModel.add_node([xplusA, nodeB])
MyNodalModel.connect(xplusA, nodeB, (...))
# (...)
```

3. **Example 3: importing a completely separate NodalModel**: Now, it is not possible to define the NodalModel using the OrbitalModel's heat fluxes and time array, since the model already exists. Instead, the existing model must be modified.
   First, the NodalModel is imported. Next, the surfaces are defined using "x+", "y+", ... as identifiers, or sets of (τ, ϕ) angles. The OrbitalModel is run, and the heat fluxes and time array are stored in the OrbitalModel object. Since the NodalModel already exists, the time array must be assigned using the `set_time` function, and the heat fluxes using `modify_node`.

```
import model from MyModelFile  # model does not have a time array

surf_list = ['x+']

MyOrbitalModel = OrbitalModel(surfaces=surf_list, (...))
MyOrbitalModel.compute()
t = MyOrbitalModel.t

MyNodalModel.set_time(t=t)
MyNodalModel.modify_node('x+A', (...), q_ext_new=MyOrbitalModel.get_heat('x+'))
# (...)
```

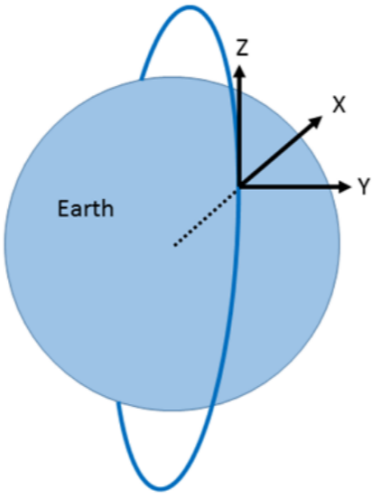


Figure C.3: Reference system for the satellite position, where x points zenith and z points in the flight direction [7].

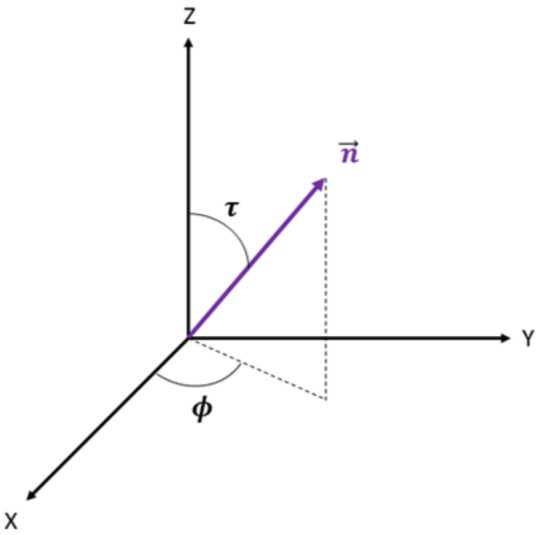


Figure C.4: Reference system for the orientation of the plate [7].

C.1.3. General Notes & Recommendations

This sub-section contains numerous general things to keep in mind when using the software. These notations are not part of one specific step in the modelling process, but can be considered throughout the entire process.

Orbital Model and Nodal Model Independence
In principle, the nodal model can be used independently from the orbital model. This was also illustrated in Figure C.2. That is, when making a model for a thermal vacuum test on ground, for example, the nodal model can be created without also having to create an orbital model. The conditions of the thermal test can be used as inputs for the nodal model, and the thermal simulation can be run. However, when simulating a spacecraft in orbit, the orbital model can be created and used for the thermal simulation.

Time Arrays and Time-Varying Parameters
An important note on time arrays is given in this sub-section. They were already mentioned before, but to be more specific, a ”time array” refers to a NumPy array with values that represent the time (x-axis) of the orbital and thermal simulations. Additionally, ”time-varying parameters” are Node properties that may vary throughout time: these are the internal power and environmental heat fluxes. Node objects can have time-varying parameters, but not a time array. A time array can only be assigned to a NodalModel. This is because the NodalModel class contains all the functionalities to run thermal simulations, but the Node class just defined a Node, and cannot run thermal simulations.

It becomes clear that time arrays and time-varying parameters should be compatible with each other. For example, if the time array of a NodalModel is [0, 1, 2, 3, 4], the time-varying parameters of its Nodes should have the same length, such as P_int = [0, 0, 1, 2, 3]. If the arrays do not have the same length, they are incompatible with each other; such incompatibility may occur if one Node with a certain time-varying parameter is added to a NodalModel with a differently sized time array. The software will throw an error, and erases the parameter and adds an empty array instead. This should always be avoided.

To avoid such conflicts, it is a good idea to always use the time array generated by the OrbitalModel. When an orbital model is defined, its property `OrbitalModel.t` contains a time array, and the computed environmental heat fluxes are compatible with this array. When creating the accompanying NodalModel and its Nodes, the orbital model’s time array can be used. This ensures that no conflicts occur between time arrays and/or time-varying parameters.

It is also possible to define a NodalModel without a time array assigned. This could be useful when making a ”bare-bone” model as an example model that can be copy-pasted into other models. For example, a PCB can be modelled without any time array, power inputs etc., as a sort of template to be copied into other models. This is essentially represented by example 3 from the previous sub-section.

Default Arguments
Many function/class inputs contain default parameters (e.g., P_int=None); such parameters with default values are not required to be given by the user. For example, if the user desires P_int to be zero, one can leave out the P_int argument, instead of manually defining P_int=0.

Many parameters have ”None” as a default parameter. This is done in case there are multiple ways to define said parameter, if the parameter is not strictly needed (e.g. initial temperature: if not given, it is estimated with a steady-state computation), or in case there are complex interactions within the function/class itself. For example, the beta angle can be defined directly, or indirectly via the inclination and RAAN. The software notices which parameters the user has entered, by confirming whether the parameter is None or not. If it is None, the parameter was not entered by the user, and therefore, some other parameters must have been assigned in order to fully define the property.

Node Parameters versus NodalModel Parameters
Nodes and NodalModels both contain similar parameters, besides a few exceptions. However, to avoid confusion, the difference between the parameters in Nodes and NodalModels is explained here.

As an example, a property of a Node can be its solar absorptivity α (`alpha`). This is stored in its attribute `Node.alpha`, as a scalar. When it is added to a NodalModel, the value is stored in an array with alpha values: `NodalModel.alpha`. This difference is illustrated below:

```
nodeA = Node((...), alpha=0.8)
nodeB = Node((...), alpha=0.5)
MyNodalModel = NodalModel((...))
MyNodalModel.add_node([nodeA, nodeB])

print(nodeA.alpha)
>>> 0.8
print(MyNodalModel.alpha)
>>> np.array([0.8, 0.5])
```

When a property of a Node is a 1-dimensional array, it becomes a 2-dimensional array when added to a NodalModel. Each column is a node, and each row is a time step.

```
t = np.array([1, 2, 3, 4, 5])
powerA = np.array([0, 0, 1, 2, 3])
powerB = np.array([2, 1, 0, 0, 0])

nodeA = Node((...), P_int=powerA)
nodeB = Node((...), P_int=powerB)
MyNodalModel = NodalModel((...))
MyNodalModel.add_node([nodeA, nodeB])

print(nodeA.P_int)
>>> np.array([0, 0, 1, 2, 3])
print(MyNodalModel.P_int)
>>> np.array([[0, 2], [0, 1], [1, 0], [2, 0], [3, 0]])
```

Note that, whenever a Node is added to a NodalModel, the entire Node object is stored in `NodalModel.nodes`. Thus, it is possible to retrieve the Node's properties in two ways: using `NodalModel.nodes[index].property` or alternatively `NodalModel.property[index]`. For the time-varying properties, the NodalModel arrays are 2-dimensional, so one should use: `NodalModel.nodes[index].property` or alternatively `NodalModel.property[:, index]`.

Celsius or Kelvin

The default unit for temperature in any NodalModel is degrees Celsius. If the user desires a NodalModel all based on temperatures in Kelvin, the following argument must be added to the NodalModel (this can only be set when initialising the NodalModel; it cannot be changed afterwards):

```
# Changing the unit of temperature to Kelvin (celsius=False)
MyNodalModel = NodalModel((...), celsius=False)
```

Error Handling

It was chosen to not use standard Python errors, since they stop the entire program, while in some cases, that is an unnecessarily harsh measure. Instead, there is an internal error handling feature present in the NodalModel and Node objects. If an error is thrown, there is always a detailed description about the error, and how to resolve it.

There are two types of error messages:

1. Errors: are thrown when some task could not be performed, and it invalidates the simulation. An example is shown in Figure C.5; a Node was not given any form of heat capacity or material, and the simulation cannot run without it.

```
----------------ERROR----------------
The heat capacity of node node1 could not be assigned
since no material nor heat capacity was given.
Either assign a heat capacity C_cap, or a material and a mass/volume.
-------------------------------------
```

Figure C.5: Example error message in the Python console.

2. Warnings: are thrown when some task could not be performed, but it is not necessarily a crucial issue. The warning message will describe how critical the issue is. For example, Figure C.6 shows a warning thrown when no optical properties are given to a certain Node. This may not be critical, since non-radiative Nodes do not use optical properties in any way.

```
--------------WARNING--------------
The absorptivity and emissivity of node node1 were not given,
hence they were set to 1.
This warning can be ignored if the node does not emit/receive any radiation.
-----------------------------------
```

Figure C.6: Example warning message in the Python console.

Since some other things are printed in the Python console, such as the progress in OrbitalModel and NodalModel computations, those error messages might get buried and the user misses them. To avoid this, the `NodalModel.show_plots` function recognises if any errors have been printed, and prints an extra statement at the very bottom of the console:

```
------------------------!ATTENTION!--------------------------
There are errors/warnings to view in the console.
Always view the entire console to not miss any errors/warnings.
-------------------------------------------------------------
```

Figure C.7: Attention message at the very bottom of the Python console that appears if any other errors or warnings have occurred.

C.2. Defining an Orbital Model

Defining an orbital model is not strictly necessary when a thermal model is made. The NodalModel class can be solved with just manually defined inputs, without an orbital model. This might be the case for thermal tests on ground. However, for predicting in-orbit satellite temperatures, an orbital model is a requirement.

All the inputs needed to define an OrbitalModel object are also mentioned in Section C.6.5, in great detail. However, some more details on more potentially confusing parameters and functions are given below.

C.2.1. Beta Angle Definition

The solar declination angle β must be provided directly or indirectly. Providing it directly is possible by entering the beta angle in degrees into the orbital model. Providing it indirectly requires both the inclination angle in degrees and the RAAN in degrees. In that case, β will be computed automatically, but it should be noted that the day in the year will be important as well. If β is provided directly, the day does not play a role, but if β is computed from the RAAN and inclination, the day takes part of the β calculation. The formula used for this is Equation 3.29. Below are two examples given on how to define the beta angle. The other input parameters can be ignored here. Note that the second example does not result in $\beta = 30$; it is just an arbitrary example.

```
surf_list = ['x+', 'y+', 'z+', 'x-', 'y-', 'z-']

# Example 1: defining beta directly
MyOrbitalModel = OrbitalModel(h=500e3, surfaces=surf_list, beta=30, day=100)

# Example 2: defining beta indirectly via inclination and RAAN
MyOrbitalModel = OrbitalModel(h=500e3, surfaces=surf_list, RAAN=45, incl=98, day=100)
```

If the second method is used, the beta angle can be retrieved and verified for correctness with the following example print statement (β must be input in degrees, but is automatically converted to radians):

```
print(f'Beta angle: {MyOrbitalModel.beta*180/np.pi:.2f} deg')
>>> Beta angle: 24.35 deg
```

C.2.2. Angular Rates Definition

The angular rates are to be given as an array or tuple in the form of [x, y, z] or (x, y, z), in degrees per second (it will be converted to rad/s). For example, (0, 0, 5) means that the spacecraft spins at 5 deg/s around its z-axis (as defined in Figure C.3). If a value such as (1, 2, 3) is given, where multiple axes have nonzero angular rates, the rotations are computed using an XYZ-rotation (each time step, the X rotation is computed first; then Y; and then Z).

Currently, the angular rates can only take constant values. More complex attitude assignments are not yet possible within a single simulation, but it is possible to run multiple simulations with different angular rates. Moreover, since the attitude is based on rotations in degrees per second, no actual dynamics are computed. Realistically, a spacecraft rotates differently with different moments of inertia, but for now, constant angular rates are assumed. This was done to avoid unnecessary complexity, since values such as the inertia tensor might not be known at the time of performing thermal analysis.

C.2.3. Surface Definition

As explained in Section C.1.2, the `surfaces` argument refers to an oriented plate that orbits around Earth, on which the environmental heat inputs are calculated. The orientation of the plate is essentially a set of spherical angles, polar angle τ and azimuth angle ϕ, as also shown in Figure C.4. These angles can be entered directly by the user, or indirectly. The following methods are available (some information from Section C.1.2 is repeated here). It can be noted that `get_heat` returns a tuple with (q_pla, q_alb, q_s), and `q_ext_new` takes this tuple, and converts into separate Node properties Node.q_pla, Node.q_alb, and Node.q_s.

1. Enter τ and ϕ directly as a tuple. The resulting heat fluxes must be manually stored in the corresponding Nodes. In this case, the `get_heat` function takes an integer value, showing which index in the list of τ & ϕ tuples to take.

```
surf_list = [(90, 0), (90, 90), (...)]  # tau and phi angles in degrees

MyOrbitalModel = OrbitalModel(surfaces=surf_list, (...))
MyOrbitalModel.compute()
t = MyOrbitalModel.t

xplusA = Node(name='x+A', (...), q_ext_new=MyOrbitalModel.get_heat(0))
```

2. Provide strings as inputs, which may be either of ”x+”, ”y+”, ”z+”, ”x-”, ”y-”, ”z-”, according to Figure C.3. All of those have their unique combinations of τ and ϕ, which are computed automatically by the software. The resulting heat fluxes must be manually stored in the corresponding Nodes. In this case, the `get_heat` function takes a string, directly referring to the name of the surface direction.

```
surf_list = ['x+', 'y+', 'z+', 'x-', 'y-', 'z-']

MyOrbitalModel = OrbitalModel(surfaces=surf_list, (...))
MyOrbitalModel.compute()
t = MyOrbitalModel.t

xplusA = Node(name='x+A', (...), q_ext_new=MyOrbitalModel.get_heat('x+'))
```

3. Feed the orbital model actual Node objects. For example, the nodal model contains six outer panels, called ”panel_x+”, ”panel_y+”, etc. When those nodes are entered into the orbital model, the ”x+” etc. handles in the node names are recognised automatically and converted into τ and ϕ angles. This does mean that those nodes must contain either of ”x+”, ”y+”, etc. The heat fluxes are automatically stored in the Nodes.

```
xplus = Node('x+A', (...))
(...)
surf_list = [xplus, (...)]

MyOrbitalModel = OrbitalModel(surfaces=surf_list, (...))
MyOrbitalModel.compute()
t = MyOrbitalModel.t
```

C.2.4. Modifying an Orbital Model

When an existing OrbitalModel is desired to be changed, the `.modify` function can be used. Using this function may be necessary when an orbital model is imported and the original definition cannot be changed, or to run the sensitivity analysis. For the latter, one orbital model is entered, and its input parameters are changed. That is all done with the modify function.

The function takes the same arguments as when an OrbitalModel is defined, and the same rules apply as described in the previous sub-sections, such as β, RAAN, and inclination. The parameters that are not entered in `.modify` remain the same as they were before. For example, if the altitude and day are changed, the following syntax would be sufficient (multiple parameters can be changed within one modify statement):

```
MyOrbitalModel.modify(h_new=400e3, day_new=100)
```

C.3. Defining a Nodal Model

The nodal model is the backbone of the thermal simulation. The ”main” NodalModel is a collection of Node objects and possibly other NodalModel objects. Since a NodalModel can be added to another NodalModel, a hierarchical model can be created, with an arbitraty ”depth” of NodalModels.

A Node can take many input parameters as properties, which is elaborately shown in Section C.6, but not all are necessary. It depends on the function of the node, and this will be explained in this section. Furthermore, it will be explained how Nodes are to be added to NodalModels, and how thermal connections between the Nodes can be made.

C.3.1. Defining a Node

The Node class is relatively simple and contains only two functions: the initialise function `__init__` and the `modify` function. Both functions take identical inputs; the only difference is that `__init__` has some mandatory arguments, while `modify` can take arbitrary arguments depending on which ones the user desires to change.

Node Names (and Numbers)

An important Node parameter is its name. The name is used throughout the entire software as an ”identifier handle”. It is therefore crucial that each Node in the NodalModel has a unique name. If the name is not unique, the software will throw an error and encourage to change the name. If no name is provided by the user, it will be an automatically generated number as soon as the Node is added to a NodalModel. This number is also a separate property of Node objects; it is only used in the context of a NodalModel, since a standalone Node cannot really have a unique identifier number without knowing the numbers of the other Nodes.

Additionally, it was already mentioned that outer nodes must contain, within their name, one of ”x+”, ”y+”, ”z+”, ”x-”, ”y-”, ”z-”. That shows to the OrbitalModel in what direction the surface points. It is possible that the name is longer than just those directions, so e.g. ”Node_x+_A” is also correct.

The names of all nodes in a NodalModel are stored in NodalModel.name. The actual name of the NodalModel itself is stored as a ”title”. The following example shows this more clearly:

```
print(nodeA.name)  # name of a Node
>>> 'nodeA'

print(MyNodalModel.name)  # names of all Nodes in a NodalModel
>>> ['nodeA', 'nodeB', 'nodeC', (...)]

print(MyNodalModel.title)  # title of a NodalModel
>>> 'MyNodalModel'
```

Heat Capacity (or Material)
The only strictly mandatory parameter to define a Node is its heat capacity C_cap. Without the heat capacity, it would be impossible to solve a thermal model that includes this node.

There are three methods to define the heat capacity. The second and third method use the material database, see Appendix B. The material properties from the database CSV file are imported automatically, and converted to a Material object. This object does not cross the user's path, but is used by the software itself. If too many parameters are given, the direct C_cap definition overrules any other definitions. For more overruling details, the source code should be consulted [8].

1. Define C_cap directly, in J/K. This is the most direct method, and might be useful when performing thermal tests on ground, since the heat capacity could be measured.

```
nodeA = Node(name='nodeA', C_cap=100, (...))
```

2. Define both the material and mass (in kg); the software immediately converts this to a heat capacity in J/K. The material is a string that corresponds to any of the available materials in the material database, see Appendix B. This is why the `show_available_materials()` function is so useful, since it prints those strings in the Python console, as in Figure C.1, and those strings can simply be copy-pasted into the definition of the Node.

```
nodeA = Node(name='nodeA', material='al7075', mass=0.01, (...))
```

3. Define both the material and volume (in m^3). The idea is the same as method 2, but now, the volume is entered by the user.

```
nodeA = Node(name='nodeA', material='al7075', volume=0.001, (...))
```

Optical Properties
Similarly to the heat capacity, the optical properties can be either entered directly by the user, or taken from the material database. If the properties are both directly defined and a "coating" (i.e., values from the database) is also given, the direct definitions of α and/or ε overrule the database values. The two methods are shown below with example codes:

1. Define α and ε directly. If either is not provided, a default value of 1 is set.

```
nodeA = Node(name='nodeA', alpha=0.8, epsilon=0.9, (...))
```

2. Define a coating; the α and ε values from that entry in the database are entered automatically into the Node.

```
nodeA = Node(name='nodeA', coating='black_paint', (...))
```

The optical properties are not always needed, but only if the Node is radiatively connected with other Nodes, or if the Node is an outer Node (i.e. receives environmental heat and emits heat to space). However, it is recommended to apply the properties anyway, to avoid accidentally using incorrect values.

Time-Varying Parameters (Heat Inputs)
It was already highlighted in Section C.1.3 that time-varying parameters may cause issues if the array length is not the same as the time array from the NodalModel to which the Node is added. It was also recommended that the time array from the OrbitalModel is used as much as possible, to avoid such incompatibilities.

Firstly, following methods for defining heat inputs are recommended for the internal power P_int:

1. Giving a single, constant value. The value can be entered as a float or integer value, and is automatically converted to the correct array size once it is added to the NodalModel. It is the simplest way to define an internal power.

```
nodeA = Node(name='nodeA', P_int=0.5, (...))
```

2. Giving a NumPy array with varying values (using the OrbitalModel time array). Using the OrbitalModel is not strictly required, but it ensures time array compatibility. The most important thing is that the time array of the entire NodalModel and Nodes is consistent. The following code is just an example of how such an array could be constructed.

```
t = MyOrbitalModel.t

p_int_varying = np.zeros(t.shape)
p_int_varying[10:20] = 0.5

nodeA = Node(name='nodeA', P_int=p_int_varying, (...))
```

Secondly, the following methods for defining heat inputs are recommended for the environmental heat fluxes (this was already touched upon in Section C.1.2). Note that the Node should have `outer=True`. Additionally, the user may notice that the heat fluxes computed by the OrbitalModel are in [W/m^2]. The actual power in Watts is only computed once the `NodalModel.solve` function is called. Therefore, the outer Nodes must have a surface area (or geometry) and optical properties (or coating) assigned.

1. Define the Node before the OrbitalModel, and enter it directly into the OrbitalModel. The heat does not need to be assigned manually.

```
xplusA = Node(name='x+A', (...), outer=True)

MyOrbitalModel = OrbitalModel(surfaces=[xplusA], (...))
MyOrbitalModel.compute()
t = MyOrbitalModel.t

MyNodalModel = NodalModel(t=t, title='MyNodalModel')
MyNodalModel.add_node(xplusA)
# (...)
```

2. Manually assign the heat to the Node, either when defining the Node, or afterwards using `Node.modify`.

```
surf_list = (...)  # either using tau&phi or x+, y+ etc.

MyOrbitalModel = OrbitalModel(surfaces=surf_list, (...))
MyOrbitalModel.compute()
t = MyOrbitalModel.t

#  Option 1: Assigning heat directly when the Node is defined
xplusA = Node(name='x+A', (...), outer=True, q_ext=MyOrbitalModel.get_heat('x+'))

# Option 2: Assigning the heat after having defined the Node
xplusA = Node(name='x+A', (...), outer=True)
xplusA.modify(q_ext_new=MyOrbitalModel.get_heat('x+'))

MyNodalModel = NodalModel(t=t, title='MyNodalModel')
MyNodalModel.add_node(xplusA)
# (...)
```

Finally, a special case is shown here. Section C.3.2 will show how to make a PCB NodalModel, and assigning internal power and heat fluxes to them is more complicated. It is possible to follow the usual method of assigning the heat to each individual Node. However, if a thermal model has, e.g., 10 PCBs, that is 50 Nodes, assuming each PCB has 5 Nodes. It would be convoluted to manually assign heat to all those Nodes. Hence, a special function was developed for this. The following methods can be used for defining the internal power and environmental heat fluxes:

1. When defining internal power P_int: the `make_5node_pcb` function takes a power argument and applies the power just at the central node (node "A"). The user could change the desired node in the source code of `make_5node_pcb`.

```
# Option 1: Apply internal power when defining the PCB
xplus, xplusA, xplusB, xplusC, xplusD, xplusE = make_5node_pcb(title='x+', power=0.5,
    (...))  # power applied to centre node xplusA

# Option 2: Apply internal power after defining the PCB, using modify
xplus, xplusA, xplusB, xplusC, xplusD, xplusE = make_5node_pcb(title='x+', (...))
xplus.modify_node(xplusA, P_int_new=0.5)  # internal power applied to centre node xplusA
```

2. When defining environmental heat fluxes: the time array from the OrbitalModel must be used when defining the PCB (t=t). Then, the dedicated function `assign_q_ext_to_pcb` can be used.

```
t = MyOrbitalModel.t

xplus, xplusA, xplusB, xplusC, xplusD, xplusE = make_5node_pcb(title='x+', t=t, (...))
assign_q_ext_to_pcb([xplus], MyOrbitalModel)
```

Geometrical Properties

Nodes do not need any geometrical properties assigned if they are not radiatively connected to other nodes. However, even if the node does not radiate, providing it with a location (”origin”) can be useful when plotting the model in 3D (shown in Section C.4), even though it would not be used in calculations. The geometry and origin must be defined if the node is radiatively connected to others. The node must then also have optical properties assigned.

The location of a Node is defined with the `origin` argument, which are (x, y, z) coordinates (in metres) of the centre of the Node. If the Node has no geometry, an origin may still be provided such that the node appears in the 3D plot. The width and height of a Node is defined using (x, y, z) dimensions (in metres), where one of the 3 dimensions must be zero; in other words, a plate is always assumed to have no thickness. For example, a 10x10 cm plate in the x-y plane, with its origin at 1 cm in the z-direction: `geom=(0.1, 0.1, 0)` and `origin=(0, 0, 0.01)`. This is also illustrated in Figure C.8 (from Section 3.4.3).

When a geometry is specified, the surface area of the Node is automatically calculated by the software. However, if the user desires to manually overrule the area, a value in [m^2] can be entered in the `area` input argument when defining or modifying a Node.

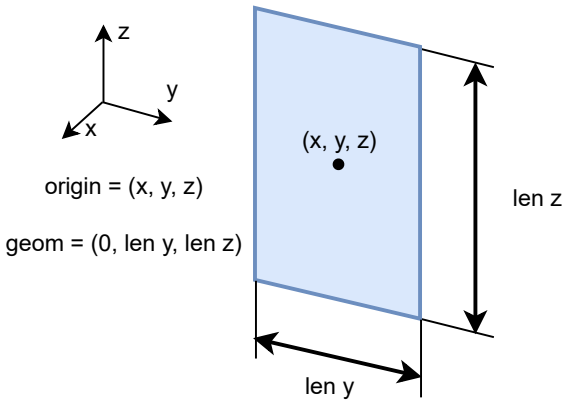


Figure C.8: Example definition of a flat plate in the y-z plane, where the outward normal of the plane must be defined as zero thickness.

Outer Nodes

Nodes that face outer space and therefore receive environmental radiation, as well as emit infrared radiation toward space, are called ”outer nodes”. This distinction between outer nodes and non-outer nodes (inner nodes) is made such that the transient solver knows which nodes lose heat to space and/or receive environmental heat. When defining a Node, `outer` is simply a boolean:

```
nodeA = Node(name='nodeA', outer=True, (...))
```

The default setting is `outer=False`, so the parameter is only required as an input when defining outer nodes. However, if the user forgets to define `outer=True`, the software still automatically assigns it in case the Node is given environmental heat fluxes. This is always correct, since a Node that receives external heat will also always lose heat to space.

Initial Temperature

Another optional parameter is the initial temperature `T0` of a Node. If all Nodes are given a `T0`, they are used as initial values for the simulation. Otherwise, the software will compute an initial steady-state instead, after which the transient simulation begins. This steady-state computation is based on the average power throughout the simulation. Additionally, it is possible to enter a `T0` in the `solve` function of a NodalModel, which sets the initial temperature of all Nodes to that value.

```
# Example 1: all nodes have a T0
nodeA = Node(name='nodeA', (...), T0=5)
nodeB = Node(name='nodeB', (...), T0=10)

MyNodalModel.add_node([nodeA, nodeB])
MyNodalModel.connect(nodeA, nodeB, C_con=1)
MyNodalModel.solve()
(...)

# Example 2: not all nodes have a T0, so a steady-state computation will be the initial guess
nodeA = Node(name='nodeA', (...))
nodeB = Node(name='nodeB', (...), T0=10)

MyNodalModel.add_node([nodeA, nodeB])
MyNodalModel.connect(nodeA, nodeB, C_con=1)
MyNodalModel.solve()  # will automatically compute an initial steady-state
(...)

# Example 3: no nodes have a T0, but the solve() function defines T0 of all nodes
nodeA = Node(name='nodeA', (...))
nodeB = Node(name='nodeB', (...))

MyNodalModel.add_node([nodeA, nodeB])
MyNodalModel.connect(nodeA, nodeB, C_con=1)
MyNodalModel.solve(T0=20)  # all nodes will start at this T0
(...)
```

C.3.2. Defining a PCB

As also previously shown in Section 3.5 and Section 4.2, it may be beneficial to define a PCB with multiple Nodes, instead of just having a single node represent the whole PCB. This could be done manually, but since it is expected that PCBs will be very common in spacecraft thermal models, two templates were made: one with five nodes, and one with nine nodes. They are shown in Figure C.9.

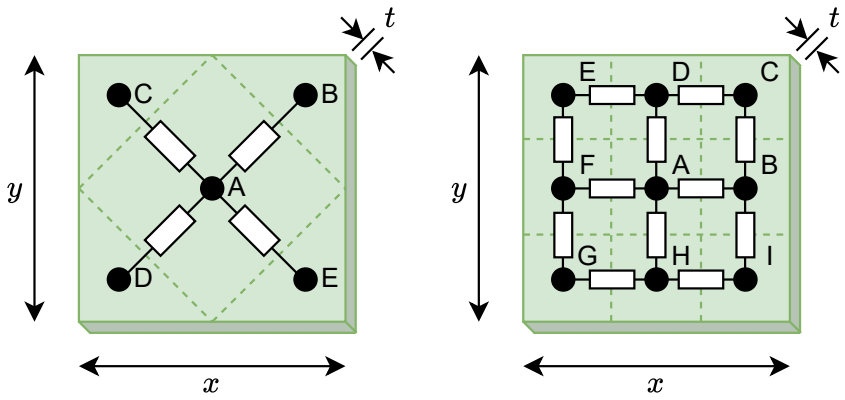


Figure C.9: PCB templates available in the software. The left PCB nodal model contains five nodes, and the right PCB contains nine nodes. The green dashed lines show the physical boundaries of all nodes, and the white rectangles are conductive connections between the nodes.

Both types of PCB have a dedicated function in the file `CommonNodalModels.py` that can be used as shown below. Not all input arguments are required to be given, since some of them have default parameters set (Section C.6). The first return argument (here: `PCB1`) is the NodalModel object of the PCB, while the other returns are the Node objects.

```
t = MyOrbitalModel.t

# 5 nodes
PCB1, PCB1A, PCB1B, PCB1C, PCB1D, PCB1E = make_5node_pcb(title='PCB1', mass=0.02, origin=(0., 0., -0.04), geom=(0.1, 0.1, 0.), thickness=0.0016, t=t, power=0.5, material='PCB', coating='PCB_green', outer=False)
# as an example, additional nodes can be added as follows:
PCB1battery = Node(name='PCB1battery', material='copper', coating='black_paint', mass=0.05, origin=(0., 0., -0.05))
PCB1.add_node(PCB1battery)


# 9 nodes
PCB1, PCB1A, PCB1B, PCB1C, PCB1D, PCB1E, PCB1F, PCB1G, PCB1H, PCB1I = make_9node_pcb(title='PCB1', mass=0.02, origin=(0., 0., -0.04), geom=(0.1, 0.1, 0.), thickness=0.0016, t=t, power=0.5, material='PCB', coating='PCB_green', outer=False)

# add to NodalModel
MyNodalModel.add_node(PCB1)  # adding the NodalModel adds the entire PCB (all Nodes)
MyNodalModel.connect('PCB1A', 'PCB2A', rad=True)  # example connection
```

If the PCB template is desired to be changed, or if more user control is desired, the end of `CommonNodalModels.py` contains the same PCB NodalModels, but written out in a ”manual definition” style; these templates can be copy-pasted into the user’s own model. It will take up more code lines, but all values can be manually tweaked easily.

The automatic PCB definition (as in the example code above) uses numerous geometric calculations to define the Nodes and their connections. These computations are done using the following equations (all based on Figure C.9). Also note that the PCB does not need to be a square; it may also be rectangular.

Additionally, a special case occurs with the surface area of the Nodes in the PCB. The default implementation in the `make_5node_pcb` and `make_9node_pcb` functions is: if the PCB has `outer=True` (it receives environmental heat and radiates heat to space), the area is divided over all Nodes with the same ratios as the mass; if the PCB has `outer=False` (it is inside the spacecraft), the area is ”lumped” in the centre node A, even though the mass is spread evenly over the Nodes. This is done because internal radiation is estimated using simplified view factor expressions from ECSS (see Section 2.4.1 and Section C.3.5). Lumping the surface area in the centre node A allows any radiation between PCBs to be computed properly using those view factor expressions. It is recommended to read Section C.3.5 to better understand the radiative connections.

5-Node PCB

As indicated by Figure C.9, the PCB is divided into five Nodes with all their own surface area, marked by the dashed green boundary lines. Based on that geometry, the mass of the PCB is divided amongst the five nodes with the following ratios:

$$m_A = \frac{1}{2} m_{tot} \tag{C.1}$$

$$m_B = m_C = m_D = m_E = \frac{1}{8} m_{tot} = \frac{1}{4} m_A \tag{C.2}$$

The distance between the nodes, used for calculating the conductance, is:

$$L_{AB} = L_{AC} = L_{AD} = L_{AE} = \frac{3}{4} \sqrt{\left(\frac{x}{2}\right)^2 + \left(\frac{y}{2}\right)^2} \tag{C.3}$$

To compute the conductance between nodes, the cross-sectional area is also needed (i.e. the area through which the heat flows). At the dashed line boundary, it is:

$$A_{con,AB} = A_{con,AC} = A_{con,AD} = A_{con,AE} = t \cdot \sqrt{\left(\frac{x}{2}\right)^2 + \left(\frac{y}{2}\right)^2} \tag{C.4}$$

With these properties, together with the PCB's thermal properties, the conductance in [W/K] is computed automatically, using Equation 2.13.

9-Node PCB
As indicated by Figure C.9, the PCB is divided into five Nodes with all their own surface area, marked by the dashed green boundary lines. Based on that geometry, the mass of the PCB is divided amongst the five nodes with the following ratios:

$$m_A = m_B = (...) = m_H = m_I = \frac{1}{9} m_{tot} \tag{C.5}$$

The distance between the horizontal nodes, used for calculating the conductance, is:

$$L_{ED} = L_{FA} = L_{AB} = (...) = \frac{x}{3} \tag{C.6}$$

The distance between the vertical nodes, used for calculating the conductance, is:

$$L_{FE} = L_{AD} = L_{HA} = (...) = \frac{y}{3} \tag{C.7}$$

To compute the conductance between nodes, the cross-sectional area is also needed (i.e. the area through which the heat flows). Horizontally, it is:

$$A_{con,ED} = A_{con,FA} = A_{con,AB} = (...) = t \cdot \frac{y}{3} \tag{C.8}$$

Vertically, it is:

$$A_{con,FE} = A_{con,AD} = A_{con,HA} = (...) = t \cdot \frac{x}{3} \tag{C.9}$$

With these properties, together with the PCB's thermal properties, the conductance in [W/K] is computed automatically, using Equation 2.13.

C.3.3. Defining a NodalModel Object & Adding Nodes

All the individual nodes do not yet form one nodal model that can be computed. Hence the "main" NodalModel must be generated.

Initialising a NodalModel
Initialising a NodalModel is very simple, as shown below. Generally, the time array from the OrbitalModel is used. Moreover, the parameter `celsius` is True by default, so does not need to be entered when using degrees Celsius. When using Kelvin, `celsius` should be set to False.

```
t = MyOrbitalModel.t

MyNodalModel = NodalModel(t=t, title='MyNodalModel')
```

It is also possible to define a NodalModel without a time array. As explained before, that may be useful when creating template models that are to be imported many times into different models. However, once the user desires to assign time-varying parameters (i.e., heat inputs), the NodalModel should be assigned a time array. Only once it has this time array, time-varying parameters can be assigned, and the model can be solved:

```
MyNodalModel = NodalModel(title='MyNodalModel')
```

```
MyNodalModel.add_node((...))
# (...)

t = MyOrbitalModel.t

MyNodalModel.set_time(t=t, erase=True)
```

This `set_time` function should be used with care. If the NodalModel and all its nodes have no time array and no time-varying parameters, the function will work smoothly (the `erase` parameter is not needed). However, when the NodalModel and/or its Nodes and/or its sub-NodalModels have a time array or time-varying parameters assigned, it should be ensured that the added time array is compatible with the existing ones. If `erase=False` (that is the default setting), `set_time` is aborted. If `erase=True`, the old data of the NodalModel and its Nodes will be deleted, and new, empty arrays will be initialised according to the newly set time array.

Adding Nodes or NodalModels

Next, Nodes can be added to the main NodalModel, or other NodalModels can be added. They can be added as follows:

```
MyNodalModel = NodalModel(title='MyNodalModel', (...))

nodeA = Node(name='nodeA', (...))
nodeB = Node(name='nodeB', (...))
MySubNodalModel = NodalModel(title='MySubNodalModel', (...))

MySubNodalModel.add_node(nodeB)  # possible to add a single object
MyNodalModel.add_node([nodeA, MySubNodalModel])  # also possible to add multiple at once
```

When a NodalModel (”submodel”) is added to the main NodalModel, all the Nodes of the submodel are essentially added individually; the submodel itself is also stored in the main NodalModel, but for the sake of computations, it is similar to adding all the Nodes of the submodel individually.

Similar issues to the `set_time` function (explained just before) may occur when adding a NodalModel to the main NodalModel. The following situations may occur:

- Both NodalModels have the same time array: the new model is added to the main model without issues.
- Both NodalModels have a different time array: an error is thrown and the addition of the models is cancelled.
- The main NodalModel has no time array, but the newly added NodalModel does: the main NodalModel inherits the time array from the newly added model, and the new model is added. However, since this time array assignment occurs automatically, possibly without the user’s knowledge, a ”light” warning is thrown to remind the user of what happened.
- The main NodalModel has a time array, but the newly added NodalModel does not: the newly added NodalModel inherits the time array from the main model, and the new model is added. However, since this time array assignment occurs automatically, possibly without the user’s knowledge, a ”light” warning is thrown to remind the user of what happened.
- Both NodalModels do not have a time array: the new model is added to the main model without issues, although neither has a time array, so the model cannot be solved yet.

C.3.4. Modifying Nodes

Node objects have a function `modify`, and NodalModel objects have a function `modify_node`. The principle is similar to `OrbitalModel.modify`, see also Section C.2.4.

The inputs of the `modify` and `modify_node` functions are similar to the inputs of initialising a Node or NodalModel. Only the parameter that is entered into the function will change, while the other parameters

that have not been entered stay the same as before.

One point of caution is modifying Nodes that are inside a NodalModel. Independent Nodes can just be modified themselves, but if a Node is inside a NodalModel, the Node should always be modified using the `NodalModel.modify_node` function, and NOT `Node.modify`. This is because a Node's properties are entered into the NodalModel's whole infrastructure, all its matrices, etc. When the Node is changed, the NodalModel must adapt as well. When using `NodalModel.modify_node`, not only is the Node itself being changed (`Node.modify` is actually automatically called within `NodalModel.modify_node`), but the NodalModel adapts accordingly as well. Some examples are given below:

```
# Example 1: Modifying an independent Node
nodeA = Node(name='nodeA', (...))
nodeA.modify(alpha_new=0.8, epsilon_new=0.85)  # can change multiple parameters at once

# Example 2: Modifying a Node that is part of a NodalModel
MyNodalModel = NodalModel((...))
nodeA = Node(name='nodeA', (...))
MyNodalModel.add_node(nodeA)

# Option 2A: Using the Node object as identifier
MyNodalModel.modify_node(node_mod=nodeA, alpha_new=0.8, epsilon_new=0.85)
# Option 2B: Using the Node name (string) as identifier
MyNodalModel.modify_node(node_mod='nodeA', alpha_new=0.8, epsilon_new=0.85)
```

C.3.5. Connecting Nodes

When a NodalModel has a number of Nodes assigned, those Nodes still need thermal connections. The `NodalModel.connect` function allows this.

Firstly, it should be known that both Node objects themselves, or their names as strings can be used to make connections. When a string is used, the connect function automatically finds the Node that corresponds to this name. This is why Nodes in a NodalModel must never have the same name. The two methods are shown below:

```
MyNodalModel = NodalModel((...))
nodeA = Node(name='nodeAstring', (...))
nodeB = Node(name='nodeBstring', (...))

# Option 1: use the Node objects
MyNodalModel.connect(nodeA, nodeB, (...))

# Option 2: use strings to refer to the Node names
MyNodalModel.connect('nodeAstring', 'nodeBstring', (...))
```

Furthermore, the choice can be made to make one connection at a time, or to make multiple connections within one statement. The options below can be ”mixed” with each other; for example, options 2 and 3 can both be used to define a conductive and/or radiative connection between nodeA-nodeB and nodeA-nodeC, all in a single statement.

```
MyNodalModel = NodalModel((...))
nodeA = Node(name='nodeAstring', (...))
nodeB = Node(name='nodeBstring', (...))
nodeC = Node(name='nodeCstring', (...))

# Option 1: define connections one at a time
MyNodalModel.connect(nodeA, nodeB, C_con=1)
MyNodalModel.connect(nodeA, nodeC, C_con=1)

# Option 2A: define multiple connections at once, with the same value
MyNodalModel.connect(nodeA, [nodeB, nodeC], C_con=1)
# Option 2B: define multiple connections at once, with different values
MyNodalModel.connect(nodeA, [nodeB, nodeC], C_con=[1, 1.5])

# Option 3: define both a conductive and radiative connection at once
MyNodalModel.connect(nodeA, nodeB, C_con=1, rad=True)
```

```

# Option 2B+3: define multiple connections at once, both conductively and radiatively
# If nodeA-nodeC does not want C_con assigned, it can either be zero or None
MyNodalModel.connect(nodeA, [nodeB, nodeC], C_con[1, None], rad=[False, True])

# (...)
```

One can always quickly check whether the correct thermal connections have been made by printing the `connections` property of the NodalModel. The example below has nodeA-nodeB having both a conductive and radiative connection, and nodeA-nodeC having a conductive connection. The connections can also visually be checked using the 3D plot, as shown in Section C.4.

```
print(MyNodalModel.connections)
>>> [('nodeA', 'nodeB', 'conductive'), ('nodeA', 'nodeB', 'radiative'), ('nodeA', 'nodeC', '
    conductive'), (...)]
```

More details on how to specifically define conductive and radiative connections are given in the two sub-sections below.

Conductive Connections

As discussed in Section 2.4.2, there are two different types of conductive connections: through-connections and contact connections. The former is conduction that occurs through a solid piece of material, while the latter occurs when two materials touch each other. Since the two types of connections are fundamentally different, there are different parameters to describe their level of conductance. These properties are stored in the material database, and can be accessed by assigning the desired material or contact connection to the Node or connection. Alternatively, it is also possible to define all the aforementioned parameters manually. This is explained below:

- Through-connections are measured using the thermal conductivity k in [W/(m·K)], the cross-sectional area A in [m^2], and the distance between nodes L in [m]. Equation 2.15 is repeated below:

$$C_{con,through} = \frac{k \cdot A}{L} \tag{C.10}$$

  The difference between the automatic definition (option 1) and the manual definition (option 2) is that the former takes `k_through` from the database, while the latter has it manually defined. Option 1 only works when the material of the two nodes is identical, since it cannot be a through-connection when two different materials are involved.

```
MyNodalModel = NodalModel((...))

# Option 1: auto-definition with material assigned from the database
nodeA = Node(name='nodeA', material='al7075', mass=0.01, (...))
nodeB = Node(name='nodeB', material='al7075', mass=0.01, (...))

MyNodalModel.add_node([nodeA, nodeB])
MyNodalModel.connect(nodeA, nodeB, L_through=0.01, A=0.001)

# Option 2: manually defining k_through
nodeA = Node(name='nodeA', (...))
nodeB = Node(name='nodeB', (...))

MyNodalModel.add_node([nodeA, nodeB])
MyNodalModel.connect(nodeA, nodeB, k_through=50, L_through=0.01, A=0.001)
```

- Contact connections are measured using the heat transfer coefficient h in [W/(m^{2}K)] and the contact area A in [m^2]. Equation 2.16 is repeated below:

$$C_{con,contact} = h \cdot A \tag{C.11}$$

  As seen in Appendix B, the database also contains a number of heat transfer coefficients h for different types of contact connections. These values should ideally be validated using test data, since they are highly dependent on contact pressure, surface finish, etc.

```
MyNodalModel = NodalModel((...))

# Option 1: auto-definition with contact conduction assigned from the database
nodeA = Node(name='nodeA', (...))
nodeB = Node(name='nodeB', (...))

MyNodalModel.add_node([nodeA, nodeB])
MyNodalModel.connect(nodeA, nodeB, contact_obj='al_al', A=0.001)

# Option 2: manually defining h_contact
nodeA = Node(name='nodeA', (...))
nodeB = Node(name='nodeB', (...))

MyNodalModel.add_node([nodeA, nodeB])
MyNodalModel.connect(nodeA, nodeB, h_contact=2000, A=0.001)
```

Radiative Connections

When a node is radiatively connected to others, the geometry must be assigned, since it is used to compute the view factors with other plates (see Section 2.4.1). Those figures are repeated in Figure C.10 and Figure C.11 for convenience. The equations used to compute the view factors are only valid if the plates in question are exactly perpendicular or exactly parallel. Furthermore, the width b for the perpendicular plates (Figure C.10) should be equal for both plates, and the lengths a and b for the parallel plates (Figure C.11) must be identical for both plates.

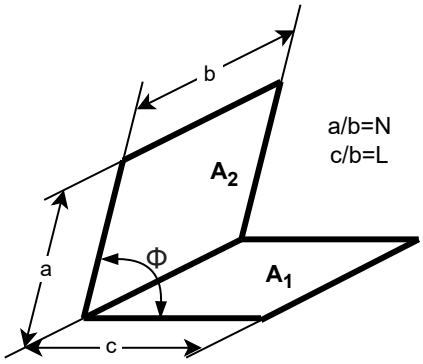


Figure C.10: View factor geometry for two angled plates (adapted from [32, p.37]). $\Phi = 90°$C is the case for perpendicular plates.

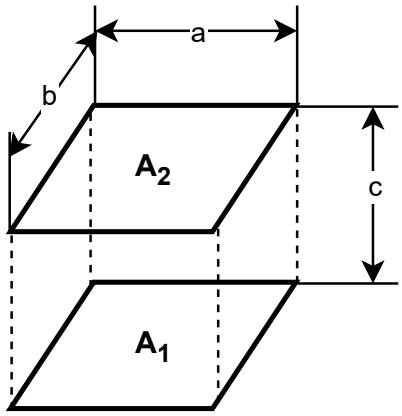


Figure C.11: View factor geometry for two identical parallel plates (adapted from [32, p.35]).

A correct and incorrect example of connecting nodes radiatively are shown below. The accompanying 3D geometry plots are shown in Figure C.12 and Figure C.13. Comparing the correct and incorrect cases:

- For the perpendicular plates: the correct case has the edges of plates A and C perfectly lined up. The touching edge must be the same length (dimension b in Figure C.10) and placement, but the other edges (a and c in Figure C.10) may differ.
  The incorrect case has correct dimensions, but plate C is not placed correctly; this offset is not allowed.
- For the parallel plates: the correct case has plates A and B of exactly the same dimensions (a and b in Figure C.11) and they are perfectly placed under/over each other.
  The incorrect case has a correct placement, but incorrect dimensions. Plate B is too large.

In case of an incorrect geometry definition, the software will throw an error, take the values of the first node, and continue the computations. If the nodes' geometries are almost the same but not exactly, the results will likely not be too far off.

```
# -------------------------- CORRECT ----------------------------
MyNodalModel = NodalModel(t=MyOrbitalModel.t, title='MyNodalModel')
```

```
3
4  # Node in the x-y plane
5  nodeA = Node(name='nodeA', C_cap=1, coating='black_paint', origin=(0, 0, 0.01), geom=(0.1, 0.1,
       0))
6
7  # Parallel Node
8  nodeB = Node(name='nodeB', C_cap=1, coating='black_paint', origin=(0, 0, -0.05), geom=(0.1,
       0.1, 0))
9
10 # Perpendicular Node
11 nodeC = Node(name='nodeC', C_cap=1, coating='black_paint', origin=(-0.05, 0, 0.035), geom=(0,
       0.1, 0.05))
12
13 MyNodalModel.add_node([nodeA, nodeB, nodeC])
14 MyNodalModel.connect(nodeA, [nodeB, nodeC], rad=True)  # radiatively connect both B and C to A
15 # (...)
16
17 # ------------------------- INCORRECT ---------------------------
18 MyNodalModel = NodalModel(t=MyOrbitalModel.t, title='MyNodalModel')
19
20 # Node in the x-y plane
21 nodeA = Node(name='nodeA', C_cap=1, coating='black_paint', origin=(0, 0, 0.01), geom=(0.1, 0.1,
       0))
22
23 # Node in the x-y plane that is parallel to can connect radiatively with node A
24 nodeB = Node(name='nodeB', C_cap=1, coating='black_paint', origin=(0, 0, -0.05), geom=(0.15,
       0.15, 0))
25
26 # Node in the  plane that is perpendicular to and can connect radiatively with nodeA
27 nodeC = Node(name='nodeC', C_cap=1, coating='black_paint', origin=(-0.05, 0.02, 0.035), geom
       =(0, 0.1, 0.05))
28
29 MyNodalModel.add_node([nodeA, nodeB, nodeC])
30 MyNodalModel.connect(nodeA, [nodeB, nodeC], rad=True)  # radiatively connect both B and C to A
31 # (...)
```

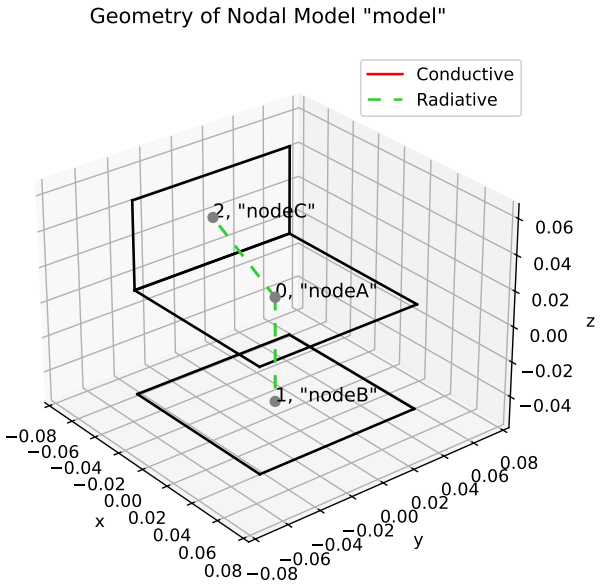


Figure C.12: Correct example of radiatively connecting a plate (A) to a parallel plate (B) and a perpendicular plate (C).

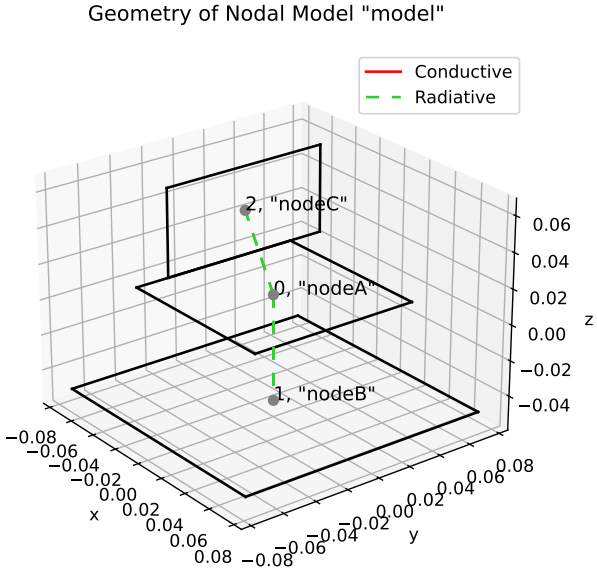


Figure C.13: Incorrect example of attempting to radiatively connect a plate (A) to a parallel plate (B) and a perpendicular plate (C). Plate B does not have the same dimensions as plate A, and plate C is misaligned with plate A (the common axis should be equal).

C.4. Running the Model & Generating Results

Now that the model and the appropriate thermal connections have been defined, it can be run, and a number of results will be generated. It is again emphasised that an orbital model is not strictly required to run a thermal simulation, so for thermal tests on ground, for example, solely a nodal model with the appropriate heat inputs would be sufficient. However, in the examples below, the simulation is run using both the orbital and nodal model.

C.4.1. Running the Model

Using the information from the previous sections, it is assumed that the orbital model and nodal model have been constructed. To run the orbital model, only the `compute` statement is required. It was explained in Section C.2.3 and Section C.3.1 (”Time-Varying Parameters (Heat Inputs)” how the heat fluxes computed by the orbital model can be assigned to nodes in the nodal model.

```
MyOrbitalModel.compute()
```

Now that the environmental heat fluxes have been computed and assigned to the outer Nodes in the NodalModel, the nodal model (thermal) simulation can be run. This is simply done using the `solve` function. It can take a number of optional parameters, which are explained in Section C.6.5, but those are generally not required.

```
MyNodalModel.solve()
```

C.4.2. Automatically Plotting Results

The computed temperatures are now stored in in the `MyNodalModel.T` matrix. The `show_plots` function is an automatic way of showing numerous plots. These plots were also already shown in Section 3.5 and explained in great detail, so they are not repeated here. However, some more details on Python syntax are given.

`show_plots` takes a number of optional arguments: the first four arguments determine which plots are shown and which are not, while the last argument (`whichnodes`) allows the user to select which nodes to be plotted, instead of all of them. It can either be a string referring to the Node name (such as ’z-A’), or the actual Node object.

`OrbitalModel.animate_attitude` can be used if the spacecraft is rotating. If any other figures are shown, those first need to be closed, after which the animation appears. It is possible to change the speed of the animation with the optional `speed` and `realtime` parameters. The former is a multiplication factor of the default speed (e.g., `speed=2` is twice as fast), and the latter is a boolean where the animation is shown at ”real-time speed”, if the user’s computer is fast enough.

```
# Most basic use of show_plots. All plots and all nodes are shown.
MyNodalModel.show_plots()

# Using show_plots with optional arguments. In this example, only the temperatures are shown,
    and only of nodes z-A to z-E. All boolean arguments are by default True.
MyNodalModel.show_plots(showrad=False, shownodes=False, showtemps=True, showflows=False,
    whichnodes=['z-A', 'z-B', 'z-C', 'z-D', 'z-E'])

# Three options available for animating the spacecraft:
MyOrbitalModel.animate_attitude()  # Most basic use of animate_attitude
MyOrbitalModel.animate_attitude(speed=0.5)  # halves the animation speed
MyOrbitalModel.animate_attitude(realtime=True)  # animates in real-time (if the computer is
    fast)
```

An additional visualisation shown by `show_plots` when `shownodes=True`, is a hierarchical model tree printed in the Python console. An example of this was shown in Section 3.2.3 as well. Essentially, it prints out the main NodalModel, all its Nodes, all its sub-NodalModels and their Nodes, etc. There is no ”depth” limit of how many sub-sub-sub-...-NodalModels there are within the main NodalModel. The example from Section 3.2.3 is used here, and the model tree would be printed into the console as in Figure C.14.

```
Spacecraft = NodalModel(title='Spacecraft')
Structural_ring_A = Node(name='Structural_ring_A', C_cap=1)
Structural_ring_B = Node(name='Structural_ring_B', C_cap=1)
Spacecraft.add_node([Structural_ring_A, Structural_ring_B])

Stack = NodalModel(title='Stack')
Spacer_A = Node(name='Spacer_A', C_cap=1)
Spacer_B = Node(name='Spacer_B', C_cap=1)
Stack.add_node([Spacer_A, Spacer_B])

PCB_A = NodalModel(title='PCB_A')
PCB_A_centre = Node(name='PCB_A_centre', C_cap=1)
PCB_A_corner1 = Node(name='PCB_A_corner1', C_cap=1)
PCB_A_corner2 = Node(name='PCB_A_corner2', C_cap=1)
PCB_A_corner3 = Node(name='PCB_A_corner3', C_cap=1)
PCB_A_corner4 = Node(name='PCB_A_corner4', C_cap=1)
PCB_A_battery = Node(name='PCB_A_battery', C_cap=1)
PCB_A.add_node([PCB_A_centre, PCB_A_corner1, PCB_A_corner2, PCB_A_corner3, PCB_A_corner4,
    PCB_A_battery])

PCB_B = NodalModel(title='PCB_B')
PCB_B_centre = Node(name='PCB_B_centre', C_cap=1)
PCB_B.add_node(PCB_B_centre)

Stack.add_node([PCB_A, PCB_B])
Spacecraft.add_node(Stack)

Spacecraft.show_plots(showrad=False, shownodes=True, showtemps=False, showflows=False)
```

```
MODEL TREE:
Spacecraft
└---Structural_ring_A
└---Structural_ring_B
└---Stack
    └---Spacer_A
    └---Spacer_B
    └---PCB_A
        └---PCB_A_centre
        └---PCB_A_corner1
        └---PCB_A_corner2
        └---PCB_A_corner3
        └---PCB_A_corner4
        └---PCB_A_battery
    └---PCB_B
        └---PCB_B_centre
```

Figure C.14: Hierarchical model tree shown in the Python console after calling `Spacecraft.show_tree`.

C.4.3. Manually Plotting Results

If the figures above do not suffice, it is possible to manually extract the results and manually plot them. However, it should be noted that functionalities like the `whichnodes` argument are not present when manually making plots, so nodes have to be selected using integer indices. These are the automatically assigned numbers of all nodes once they are added to a NodalModel. This was briefly explained in Section C.3.1.

Some examples are shown in this sub-section. The most relevant parameters are:

- **Node temperatures**: the main output of the thermal simulation is the spacecraft's temperature. These can be extracted from the NodalModel in the following way. The `whichnodes` argument present in the `show_plots` function is imitated as shown below.

```
import matplotlib.pyplot as plt

whichnodes = ['x+A', 'y+A', 'z+A', 'PCB1A']

fig = plt.figure()
ax = fig.add_subplot()
for i, node in enumerate(MyNodalModel.nodes):
    if node.name in whichnodes:
        ax.plot(MyNodalModel.t, MyNodalModel.T[:, i], label=f'{node.name}')
ax.set_title('Temperatures of Nodes in NodalModel')
if MyNodalModel.celsius:
    ax.set_ylabel(r'Temperature [$\degree$C]')
else:
    ax.set_ylabel(r'Temperature [K]')
ax.set_xlabel('Time [s]')
ax.legend()
ax.grid()
plt.show()
```

- **Environmental heat fluxes**: these are stored in the orbital model, in `OrbitalModel.q_pla`, `OrbitalModel.q_alb`, and `OrbitalModel.q_s`, but they can also be retrieved using `OrbitalModel.get_heat`. Meanwhile, the heat fluxes are also stored in the NodalModel itself, in `NodalModel.q_pla`, `NodalModel.q_alb`, and `NodalModel.q_s`. The example below shows how to extract this data. The if-statements in option 2 are used to filter out all the Nodes that do not have any heat fluxes. A similar approach such as the `whichnodes` variable shown in the temperature example could also be used.

```
import matplotlib.pyplot as plt

# Option 1: use OrbitalModel properties
fig = plt.figure()
ax = fig.add_subplot()
for i, surface in enumerate(MyOrbitalModel.surfaces):
    # Option 1A: use get_heat function
    q_pla, q_alb, q_s = MyOrbitalModel.get_heat(surface)
    ax.plot(MyOrbitalModel.t, q_pla, label=f'{surface}, q_pla')
    ax.plot(MyOrbitalModel.t, q_alb, label=f'{surface}, q_alb')
    ax.plot(MyOrbitalModel.t, q_s, label=f'{surface}, q_s')

    # Option 1B: manually extract heat flows
    ax.plot(MyOrbitalModel.t, MyOrbitalModel.q_pla[:, i], label=f'{surface}, q_pla')
    ax.plot(MyOrbitalModel.t, MyOrbitalModel.q_alb[:, i], label=f'{surface}, q_alb')
    ax.plot(MyOrbitalModel.t, MyOrbitalModel.q_s[:, i], label=f'{surface}, q_s')
ax.set_title('Environmental Heat Flux')
ax.set_xlabel('Time [s]')
ax.set_ylabel(r'Heat Flux [W/m$^2$]')
ax.legend()
ax.grid()
plt.show()

# Option 2: use NodalModel properties
fig = plt.figure()
ax = fig.add_subplot()
for i, node in enumerate(MyNodalModel.nodes):
    # Option 2A: plotting the heat fluxes
    if np.any(np.abs(MyNodalModel.q_pla[:, i]) > 1e-10):
        ax.plot(MyNodalModel.t, MyNodalModel.q_pla[:, i], label=f'{node.name}, q_pla')
        ax.plot(MyNodalModel.t, MyNodalModel.q_alb[:, i], label=f'{node.name}, q_alb')
        ax.plot(MyNodalModel.t, MyNodalModel.q_s[:, i], label=f'{node.name}, q_s')

    # Option 2B: plotting the heat in Watts
    if np.any(np.abs(MyNodalModel.q_pla[:, i]) > 1e-10):
        alpha_area = MyNodalModel.alpha[i]*MyNodalModel.area[i]
        ax.plot(MyNodalModel.t, MyNodalModel.q_pla[:, i]*alpha_area, label=f'{node.name}, q_pla')
        ax.plot(MyNodalModel.t, MyNodalModel.q_alb[:, i]*alpha_area, label=f'{node.name}, q_alb')
        ax.plot(MyNodalModel.t, MyNodalModel.q_s[:, i]*alpha_area, label=f'{node.name}, q_s')
ax.set_title('Environmental Heat Flux')
```

```
ax.set_ylabel(r'Heat Flux [W/m$^2$]')  # Option 2A
ax.set_ylabel('Heat [W]')  # Option 2B
ax.set_xlabel('Time [s]')
ax.legend()
ax.grid()
plt.show()
```

- **Internal power**: the internal power is in some ways similar to the heat fluxes, except that it is not part of the OrbitalModel. The internal power is only defined in the Nodes and NodalModel. The method of extracting the data is therefore identical to extracting temperatures. Alternatively to the if-statement below, it would also be possible to filter out nodes with zero power, using the method as in the example above for the environmental heat fluxes.

```
import matplotlib.pyplot as plt

whichnodes = ['x+A', 'y+A', 'z+A', 'PCB1A']

fig = plt.figure()
ax = fig.add_subplot()
for i, node in enumerate(MyNodalModel.nodes):
    if node.name in whichnodes:
        ax.plot(MyNodalModel.t, MyNodalModel.P_int[:, i], label=f'{node.name}')
ax.set_title('Internal Power of Nodes in NodalModel')
ax.set_ylabel('Internal Power [W]')
ax.set_xlabel('Time [s]')
ax.legend()
ax.grid()
plt.show()
```

- **Heat flows**: plotting the heat flows requires a slightly different strategy, because a heat flow is between two nodes. Therefore, simply looping over all nodes and plotting their heat flows will result in many duplicate entries and a difficult to understand figure. Ideally, one should loop over the actual connections. A NodalModel has a `NodalModel.connection` argument that contains all connections in the form of tuples with ('node1', 'node2', 'radiative') or ('node1', 'node2', 'conductive'). The following example is a simplified version of how the plot is made in `show_plots`.

```
import matplotlib.pyplot as plt

whichnodes = ['z-A', 'z-B', 'PCB1A', 'PCB1B']

fig = plt.figure()
ax = fig.add_subplot()
ax.set_title('Heat Flow Between Nodes')
for i, conn in enumerate(MyNodalModel.connections):
    node1 = np.argwhere(MyNodalModel.name == conn[0])[0, 0]  # index number of first node
    node2 = np.argwhere(MyNodalModel.name == conn[1])[0, 0]  # index number of second node

    if MyNodalModel.name[node1] in whichnodes and MyNodalModel.name[node2] in whichnodes:
        ax.plot(MyNodalModel.t, MyNodalModel.heatflow[:, i], label=fr'{MyNodalModel.name[
    node1]} $\rightarrow$ {MyNodalModel.name[node2]}, {conn[2]}')
ax.legend()
ax.grid()
ax.set_xlabel('Time [s]')
ax.set_ylabel('Conductive Power From Node i to j [W]')
plt.show()
```

C.5. Performing a Sensitivity Analysis on the Model

Chapter 5 has elaborately explained the purpose of the sensitivity analysis tool in the Python software, and it has shown the capabilities of the tool. To briefly summarise the types of sensitivity analysis plots (integrator performance plots are not included here; visit Section C.6 for details):

1. **All-encompassing plots** show the temperature sensitivity of all parameters in one plot for the average spacecraft temperature, and one plot for the minimum and maximum spacecraft temperatures.
2. **Variable-specific plots** show the temperature sensitivity of a single parameter in high detail.

3. **Internal radiation plots** provide similar information as the variable-specific plot, but just showing the difference between having internal radiation set ”on” or ”off”.
4. **Time step & angular rates plots** show similar information as the variable-specific plot, but showing the effect of numerical issues when the time step is not small enough.

C.5.1. Running a Sensitivity Analysis

To start the sensitivity analysis on a model, an OrbitalModel and NodalModel object must be available, representing the desired spacecraft thermal model. In most cases, these models are likely defined in a separate Python file. Either those files are imported into `SensitivityAnalysis.py`, or the functions from `SensitivityAnalysis.py` could be imported into the file defining the models.

```
# Option 1: importing model into SensitivityAnalysis.py
from MyModel import MyNodalModel, MyOrbitalModel

# Option 2: importing sensitivity functions into MyModel.py
from SensitivityAnalysis import run_all, run_variable, run_dt_rot, run_intrad, plot_all,
    plot_variable, plot_dt_rot, plot_intrad
```

Then, the simulations need to be run for the desired parameters. For most parameters, the values are multiplication factors (so `values=1.0` would mean that the parameter is multiplied by 1, so it represents the standard baseline case). More details on the function parameters are available in Section C.6.

The `run_variable`, `plot_variable`, `run_dt_rot`, and `plot_dt_rot` functions take an argument called `name`. This argument signals to the software which variable is/was the one being analysed. It is needed for the `run` functions so that the software knows which parameters to change, and the `plot` functions require the name so that the plot titles etc. are correct. The abbreviations used are: **ABS** (absorptivity); **ALB** (albedo heat flux); **ALT** (altitude); **BETA** (beta angle); **CAP** (heat capacity); CON (conductance); **DAY** (day of the year); **DT** (time step); **EMI** (emissivity); **IR** (Earth infrared heat flux); **POW** (internal power); **RAD** (internal radiation on or off); **ROT** (rotating or non-rotating); **SOL** (solar heat flux).

Furthermore, the `overwrite` argument determines, if the desired filename already exists, whether that file should be overwritten or not. The default setting is False, since `overwrite=True` could lead to accidentally deleting important data and overwriting it with other data.

Finally, the `subfolders` argument defines the file location. There is another input parameter `folder`, which is by default set to ”SensitivityAnalysis”. This can be changed, but that is probably not needed. However, the `subfolders` argument further specifies the folders inside the main folder. If the results from the sensitivity analysis are desired to be stored in, e.g., ”PythonFilePath.../SensitivityAnalysis/MySatellite/BetaAngle”, then `folder='SensitivityAnalysis'` and `subfolders=('MySatellite', 'BetaAngle')`.

```
# All-encompassing plots
run_all(dev1=0.1, dev2=0.2, nodal_model=MyNodalModel, orbital_model=MyOrbitalModel, overwrite=
    True, subfolders=('MySatellite', 'AllEncompassing'))

# Variable-specific plots (only show one example here)
run_variable(name='ABS', values=[0.5, 0.75, 1.0, 1.25, 1.5], nodal_model=MyNodalModel,
    orbital_model=MyOrbitalModel, overwrite=False, subfolders=('MySatellite', 'Abs'))

# Internal radiation plots
run_intrad(nodal_model=MyNodalModel, orbital_model=MyOrbitalModel, overwrite=True, subfolders=(
    'MySatellite', 'Intrad'))

# Time step plots
run_dt_rot(name='DT', nodal_model=MyNodalModel, orbital_model=MyOrbitalModel, overwrite=True,
    subfolders=('MySatellite', 'Dt'))
```

The `run` functions store the solved NodalModels as Python Pickle[1] files. A Pickle file contains an entire Python object; in this case, it contains the entire NodalModel, including all the computed temperatures etc. An example folder with the results of one sensitivity analysis are shown in Figure C.15.

[1] Python Pickle documentation: `https://docs.python.org/3/library/pickle.html`

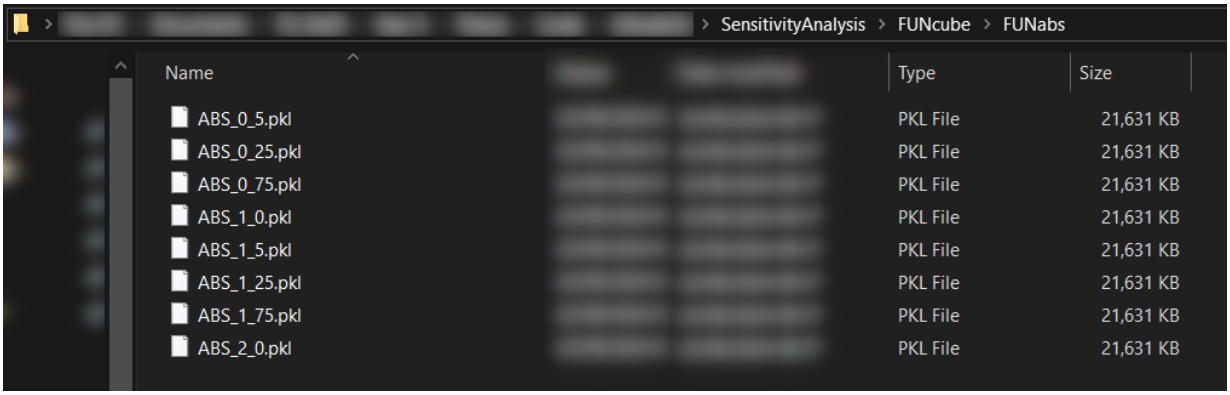


Figure C.15: Example of how Python Pickle files are stored for a sensitivity analysis case.

C.5.2. Loading & Plotting a Sensitivity Analysis

To show the plots, the following commands are used (the `folder` input was omitted since the default "SensitivityAnalysis" folder was used). The use of the `whichnode` parameter is similar to the one used in `NodalModel.show_plots`, as described in Section C.4. Both the Node object or the name (string) of the Node may be entered. The difference is that some plots can only show one Node at a time, hence the argument `whichnode` in singular form.

```
plot_all(subfolders=('MySatellite', 'AllEncompassing'))
plot_intrad(subfolders=('MySatellite', 'Intrad'), whichnodes=['z-A', 'PCB2bat'])
plot_variable('ABS', subfolders=('MySatellite', 'Abs'), whichnode='PCB2bat')
plot_dt_rot(name='DT', subfolders=('MySatellite', 'Dt'))
plt.show()  # matplotlib.pyplot.show() is always needed at the end of the file
```

The resulting plots were shown in Chapter 5 and explained in great detail, so they will not be repeated here. More information can also be found in Section C.6.

C.6. Python Files, Classes, and Functions

This section is a compilation of all the Python files, functions, and classes. ChatGPT[2] was used to convert the Python docstrings to the format shown in this section. All this information can also be found within the Python files themselves: the docstrings can be viewed in the source code, but in many Integrated Development Environments (IDEs), if the mouse is hovered over the function, the documentation is automatically displayed. For example, in PyCharm, the documentation is shown as in Figure C.16 when hovering the mouse over the function.

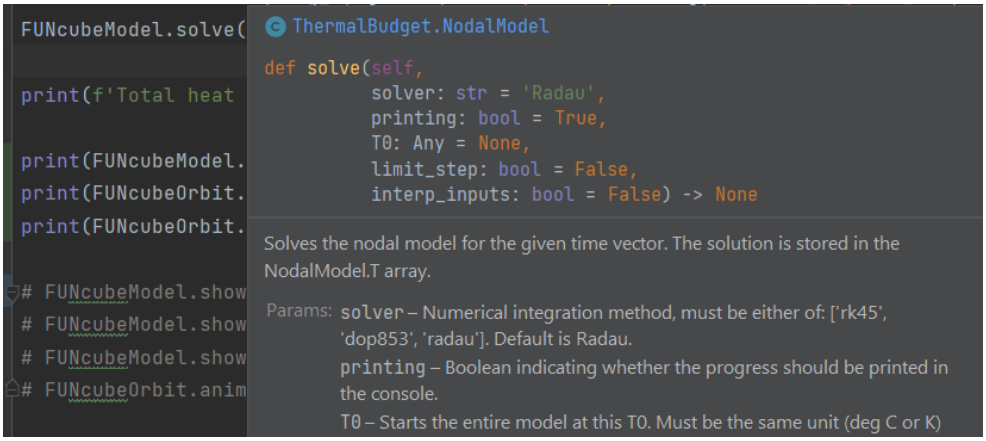


Figure C.16: Documentation shown when hovering the mouse over a function in PyCharm.

[2]ChatGPT: `https://chatgpt.com/`

Of main interest to the general user are `ThermalBudget.py`, `SensitivityAnalysis.py`, and `Materials.csv`. Furthermore, `FUNcube.py` can serve as an example on how to make an orbital model and nodal model, similarly to `CommonNodalModels.py`.

Many of the other functions shown here do not cross the general user's path, but could improve one's understanding of how the code works. Those functions that the user generally does not see, are coloured with a slightly lighter tint of grey.

C.6.1. Constants.py

"Constants.py is a collection of physical constants used for all the computations. Some values are not actual constants, such as the Solar 'constant'. Please refer to the websites mentioned below. This file does not cross the general user's path, since the constants are embedded into the computations in EnvironmentRadiation.py and ThermalBudget.py."

This file does not contain any functions or classes that are relevant to the general user, except if the constants are desired to be changed or updated.

C.6.2. Materials.py

"Materials.py reads the Materials.csv database and converts it to Python objects and dictionaries. This file does not cross the general user's path, since this file is embedded into the ThermalBudget.py classes."

(Hidden) Function: get_file(file)

Returns the path of a file in the same directory as this Python file; works on most operating systems.

Parameters:
- **file**: Name (str) of the file to be retrieved.

Returns:
- Path of the file.

(Hidden) Function: get_folder_file(folder, file, subfolders=())

Returns the path of a file in a folder within the same directory as this Python file; works on most operating systems.

Subfolders can be an arbitrary number of sub-folders, such as:
get_folder_file('folderA', 'file.txt', ('folderB', 'folderC'))
–> C:/...your-directory.../folderA/folderB/folderC/file

Parameters:
- **folder**: Name (str) of the folder which the file is in.
- **file**: Name (str) of the file to be retrieved.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.

Returns:
- Path of the file.

Class Material

"Material class with all its properties that are relevant for thermal analysis."

(Hidden) Function: __init__(name, density, c_cap, k_through)

Initialise a new Material object. Ensure that the correct units are used!

Parameters:
- **name**: Name of the material.
- **density**: Density [kg/m^3].
- **c_cap**: Specific heat capacity [J/(kg.K)].
- **k_through**: Thermal conductivity [W/(m.K)].

Class Coating
"Coating to be applied on another material, assigning the optical properties."

(Hidden) Function: __init__(name, alpha, epsilon)

Initialise a new Coating object.

Parameters:
- **name**: Name of the coating.
- **alpha**: Absorptivity in the solar spectrum..
- **epsilon**: Emissivity in the IR spectrum..

Class Contact
"Approximation of contact conductance for two materials pressed together. It highly depends on the applied pressure."

(Hidden) Function: __init__(name, h_contact)

Initialise a new Contact object. Ensure that the correct units are used!

Parameters:
- **name**: Name of the contact connection.
- **h_contact**: Heat transfer coefficient [W/(m^2.K)].

C.6.3. CommonNodalModels.py

"CommonNodalModels.py is the beginning of a collection of NodalModels. The modularity of the code allows for models to be created and copy-pasted into a new model. Hence, a file such as this one can contain commonly used nodal models (such as a PCB), which can then be used in other models. Currently, there are functions to make a 5- or 9-node PCB model, but alternatively, the code at the bottom of this file can be simply copy-pasted and adapted as desired."

Function: assign_q_ext_to_pcb(nodal_models, orbit_obj)

Since a PCB has multiple nodes, it can be tedious to assign environmental heat fluxes to all of them. This function does this automatically.

Parameters:
- **nodal_models**: either one PCB (NodalModel) or a list of PCBs (list of NodalModels). The nodes that include either of 'x+', 'y+', ... are the ones given the heat flux. E.g., the middle PCB node is 'PCBx+A', then 'PCBx+B', etc.
- **orbit_obj**: OrbitalModel object that has been computed already (hence contains heat fluxes).

Function: make_5node_pcb(title, mass, origin, geom=(0.1, 0.1, 0.), thickness=0.0016, t=None, power=None, material='PCB', coating='PCB_green', outer=False)

Make a 5-node PCB (1 central node + 4 corner nodes).
Ordering of the nodes is first the centre node (A), then counterclockwise starting on the top right (B, C, D, E).

For outer nodes, the received environment heat is spread out over the nodes, proportional to their area.
For inner nodes, applied power is concentrated in the centre node A. For internal radiation, node A is assumed to be the only radiating face, but representing the entire PCB area. This is ensured automatically.

Parameters:

- **title**: Title/name that the NodalModel has.
- **mass**: Mass in kg of the PCB.
- **origin**: Tuple (x, y, z) of origin coordinates of the node.
- **geom**: Tuple (x_width, y_width, z_width) of the rectangular shape of the node, in metres.
- **thickness**: Thickness [m] of the PCB. Default is 0.0016 m (1.6 mm).
- **t**: Time array [s] of the entire simulation. It is recommended to use OrbitalModel.t as the time array.
- **power**: Power (heat) dissipation in W of the PCB. Can be a single value or an array of values throughout time, as long as it is compatible with the OrbitalModel time array.
- **material**: Material object or string containing the bulk properties (C_cap) of the node.
- **coating**: Coating object or string containing the absorptivity (alpha) and emissivity (epsilon) of the node.
- **outer**: Whether the node radiates freely to space or not. Default is False.

Returns:

- NodalModel of a 5-node PCB, and all individual Node objects as well.

Function: make_9node_pcb(title, mass, origin, geom=(0.1, 0.1, 0.), thickness=0.0016, t=None, power=None, material='PCB', coating='PCB_green', outer=False)

Make a 9-node PCB (3x3 grid).
Ordering of the nodes is first the centre node, then counterclockwise starting at the centre right node.

For outer nodes, the received environment heat is spread out over the nodes, proportional to their area.
For inner nodes, applied power is concentrated in the centre node A. For internal radiation, node A is assumed to be the only radiating face, but representing the entire PCB area. This is ensured automatically.

Parameters:
- **title**: Title/name that the NodalModel has.
- **mass**: Mass in kg of the PCB.
- **origin**: Tuple (x, y, z) of origin coordinates of the node.
- **geom**: Tuple (x_width, y_width, z_width) of the rectangular shape of the node, in metres.
- **thickness**: Thickness [m] of the PCB. Default is 0.0016 m (1.6 mm).
- **t**: Time array [s] of the entire simulation. It is recommended to use OrbitalModel.t as the time array.
- **power**: Power (heat) dissipation in W of the PCB. Can be a single value or an array of values throughout time, as long as it is compatible with the OrbitalModel time array.
- **material**: Material object or string containing the bulk properties (C_cap) of the node.
- **coating**: Coating object or string containing the absorptivity (alpha) and emissivity (epsilon) of the node.
- **outer**: Whether the node radiates freely to space or not. Default is False.

Returns:
- NodalModel of a 9-node PCB, and all individual Node objects as well.

C.6.4. EnvironmentRadiation.py

"EnvironmentRadiation.py contains functions to ultimately compute all external heat fluxes (solar, albedo, and Earth IR) onto an orbiting plate or multiple plates. The heat_received function is the one that computes the final outputs. This file or any of the functions in this file do not cross the general user's path, since these functions are embedded into the ThermalBudget.py classes."

(Hidden) Function: earth_orbit(day)

Compute Earth-Sun distance, based on the day from January 1. Perigee is on January 3.

Parameters:
- **day**: Number of days from January 1 (January 1 itself is day 1).

Returns:
- Sun-Earth distance [m] at the given day.

(Hidden) Function: solar_flux(day)

Compute solar flux on Earth based on the Earth-Sun distance on the given day. Solar minima/maxima are currently not included.

Parameters:
- **day**: Number of days from January 1 (January 1 itself is day 1).

Returns:
- Solar flux in W/m^2 at the given day.

(Hidden) Function: albedo_flux(day, beta, theta)

Compute albedo flux at the Earth's surface, in the direction of a given orbital position. For the actual power in Watt, this value must be multiplied by (Re/(Re+h))^2, the area, spacecraft absorptivity, and the view factor.

Parameters:
- **day**: Number of days from January 1 (January 1 itself is day 1).
- **beta**: Angle [rad] between the Sun vector and orbital plane.
- **theta**: True anomaly [rad] of the spacecraft's orbit around the Earth, where theta = 0 at the orthogonal projection of the solar vector.

Returns:
- Albedo flux in W/m^2 at the Earth's surface in the direction of the given orbital position.

(Hidden) Function: planet_flux()

Average Earth infrared flux at the Earth's surface. For the actual power in Watt, this value must be multiplied by the Earth's surface area and emissivity (assume 1), spacecraft absorptivity, and the view factor.

Returns:
- Earth total infrared flux in W/m^2 at the Earth's surface.

(Hidden) Function: view_factor(h, eta)

Returns view factor of a flat surface and a sphere (ECSS-E-HB-31-01 Part 1A 4.2.2 Planar to spherical). Valid for 0 < eta < 180 degrees. Note that lambda = 180 - eta.

Parameters:
- **h**: Altitude [m] of the orbit above Earth.
- **eta**: Angle [rad] between the outward normal of the surface and the zenith direction with respect to Earth.

Returns:
- View factor between the Earth and a flat plate; is zero when the plate faces away from Earth.

(Hidden) Function: eclipse_time(beta, h)

Calculates eclipse parameters based on a simplified cylindrical shadow behind Earth. Penumbra effect is neglected.

Parameters:
- **beta**: Beta angle [rad] of Sun with respect to the spacecraft's orbital plane.
- **h**: Orbital altitude [m].

Returns:
- Time of eclipse [s].
- Orbital ratio of eclipse.
- True anomaly of eclipse onset [rad].

(Hidden) Function: heat_received(day, beta, theta, h, tau, phi)

Compute the solar, albedo, and Earth IR flux received by an oriented flat face of a satellite. This does not yet include the surface area nor absorptivity.

Parameters:
- **day**: Number of days from January 1 (January 1 itself is day 1).
- **beta**: Beta angle [rad] of Sun with respect to the spacecraft's orbital plane.
- **theta**: Array of true anomalies [rad] of the spacecraft's orbit around the Earth, where theta = 0 at the orthogonal projection of the solar vector.
- **h**: Orbital altitude [m].
- **tau**: Polar angle [rad] in spherical coordinates (z-axis parallel to velocity direction).
- **phi**: Azimuth angle [rad] in spherical coordinates.

Returns:
- Earth IR heat flux q_pla [W/m^2].
- Albedo heat flux q_alb [W/m^2].
- Solar heat flux q_s [W/m^2].

(Hidden) Function: t_to_theta(h, t)

Converts orbital time [s] to true anomaly [rad].

Parameters:
- **h**: Orbital altitude [m].
- **t**: Time [s] after passing the Sun-Earth vertical plane.

Returns:
- True anomaly [rad].

(Hidden) Function: theta_to_t(h, theta)

Converts true anomaly [rad] to orbital time [s].

Parameters:
- **h**: Orbital altitude [m].
- **theta**: True anomaly [rad].

Returns:
- Time [s] after passing the Sun-Earth vertical plane.

(Hidden) Function: beta_angle(day, RAAN, i)

Computes the solar declination angle beta for the given day, for a given orbit.

Parameters:
- **day**: Number of days from January 1 (January 1 itself is day 1).
- **RAAN**: Right ascension of the ascending node [rad] of the satellite's orbit.
- **i**: Inclination [rad] of the satellite's orbit with respect to the Earth's equator.

Returns:
- Solar declination (beta) angle [rad].

(Hidden) Function: tau_phi(direction)

Converts a direction ('x+', 'y-', etc) into spherical coordinates.

Parameters:
- **direction**: String indicating the direction of a face (one of 'x+', 'y+', 'z+', 'x-', 'y-', 'z-').

Returns:
- Polar angle tau [rad] in spherical coordinates (z-axis parallel to velocity direction).
- Azimuth angle phi [rad] in spherical coordinates.

(Hidden) Function: spherical_to_cartesian(tauphi)

Takes spherical coordinates (tau angle from the z-axis and phi angle from the x-axis) and converts to a cartesian unit vector.

Parameters:
- **tauphi**: Tuple of tau (polar angle [rad]) and phi (azimuth angle [rad]) in spherical coordinates, (z-axis parallel to velocity direction).

Returns:
- Cartesian unit vector.

(Hidden) Function: cartesian_to_spherical(cartesian)

Takes a cartesian unit vector and converts to spherical coordinates (tau angle from the z-axis and phi angle from the x-axis).

Parameters:
- **cartesian**: Cartesian coordinates of a point.

Returns:
- Tuple of tau (polar angle [rad]) and phi (azimuth angle [rad]) in spherical coordinates, (z-axis parallel to velocity direction).

C.6.5. ThermalBudget.py

"ThermalBudget.py is the backbone of this thermal simulation tool. The three major classes (Node, NodalModel, and OrbitalModel) are defined here. It is recommended to import those classes in a separate file, and build your models there. Also import the show_available_materials() function and run it at the start of your code (not obligatory, but useful); it shows a summary of available materials from the Materials.csv file."

(Hidden) Function: dT_dt(t, T_prev, const, n, C_cap, C_con, q_pla, q_alb, q_s, P_int, outer, alpha, epsilon, area, celsius, rad, t_array, cnt, printing=True, interp_inputs=False)

Calculates the derivative of the temperature of all nodes with respect to time, to be integrated afterward.

Parameters:

- **t**: Time instance [s], is unused but needed for scipy.integrate.solve_ivp.
- **T_prev**: Temperatures of the previous time step i-1 in [K] or [deg C].
- **const**: Dictionary of universal constants or Earth properties.
- **n**: Total number of nodes.
- **C_cap**: Heat capacities of the nodes [J/K].
- **C_con**: Conductance between nodes [W/K].
- **q_pla**: Earth IR heat flux [W/m^2]. These powers must stay separated since they have different spectra.
- **q_alb**: Albedo heat flux [W/m^2].
- **q_s**: Solar heat flux [W/m^2].
- **P_int**: Internal power [W] on each node (can be internal, external, or both).
- **outer**: Boolean array, whether the node radiates freely to space or not.
- **alpha**: Absorptivities in the solar spectrum of all nodes.
- **epsilon**: Emissivities in the IR spectrum of all nodes.
- **area**: Surface area [m^2] of all nodes.
- **celsius**: Boolean, whether the units are in Celsius (True) or Kelvin (False).
- **rad**: Radiative exchange factors between nodes. Same matrix shape as C_con, also symmetric.
- **t_array**: Entire time array (numpy array) of the simulation, used to track progress within scipy.solve_ivp.
- **cnt**: List with a zero value (as: cnt = [0]), used for printing progress in the console.
- **printing**: Boolean indicating whether the progress should be printed in the console.
- **interp_inputs**: Boolean indicating whether the environmental inputs (q_pla, q_alb, q_s, P_int) should be interpolated during scipy's variable time stepping. Improves accuracy, but also increases computational time. Default is False.

Returns:

- Derivative of temperature with respect to time in [K/s].

(Hidden) Function: view_factor_perp(a2, b, c1)

Computes the view factor between two perpendicular plates with a common edge (b).

Parameters:

- **a2**: Width [m] of the receiving plate.
- **b**: Length [m] of the common edge of both plates (must be equal).
- **c1**: Width [m] of the emitting plate.

Returns:

- View factor [-] from plate 1 to 2.

(Hidden) Function: view_factor_par(a_width, b_depth, c_dist)

Computes the view factor between two parallel plates with the same size, exactly placed above/under each other. Parameters a_width and b_depth are interchangeable due to the symmetry in the formula.

Parameters:
- **a_width**: Width [m] of both plates (must be equal).
- **b_depth**: Depth [m] of both plates (must be equal).
- **c_dist**: Distance [m] between both plates (F12 = F21).

Returns:
- View factor [-] between the two plates (F12 = F21).

(Hidden) Function: plot_plate(origin, size, ax, n, name)

Plot the outer lines of a flat, rectangular plate, as well as the origin.

Parameters:
- **origin**: (x, y, z) position of the centre of the plate.
- **size**: (x, y, z) dimensions of the plate. The size in the normal direction of the surface must be set to zero, and the other two dimensions (width and height) must be nonzero.
- **ax**: Matplotlib axis object (mpl_toolkits.mplot3d.axes3d.Axes3D) which can be defined with: matplotlib.pyplot.figure().add_subplot(projection='3d').
- **n**: Node number of the drawn plate.
- **name**: Name (string) of the drawn plate.

(Hidden) Function: plot_point(origin, ax, n, name)

Plot the origin of a node.

Parameters:
- **origin**: (x, y, z) position of the centre of the plate.
- **ax**: Matplotlib axis object (mpl_toolkits.mplot3d.axes3d.Axes3D) which can be defined with: matplotlib.pyplot.figure().add_subplot(projection='3d').
- **n**: Node number of the drawn plate.
- **name**: Name (string) of the drawn plate.

(Hidden) Function: plot_connections(point1, point2, axis, radiative=False)

Plot a line representing a conductive or radiative link between two nodes.

Parameters:
- **point1**: (x, y, z) position of the first node.
- **point2**: (x, y, z) position of the second node.
- **axis**: Matplotlib axis object (mpl_toolkits.mplot3d.axes3d.Axes3D) which can be defined with: matplotlib.pyplot.figure().add_subplot(projection='3d').
- **radiative**: Boolean indicating whether it is a radiative (True) or conductive (False) connection.

(Hidden) Function: plot3dline(point1, point2, axis, colour='k', lbl='', style='solid', ordr=None)

Converts two points into a line and plots it on the given axis.

Parameters:
- **colour**: Colour of the desired plot.
- **point1**: Starting coordinate.
- **point2**: Ending coordinate.
- **axis**: Figure axis to be plotted on.
- **lbl**: Label of the axis.
- **style**: Linestyle.
- **ordr**: Zorder of the line (how far it is above or below others).

(Hidden) Function: plot3dframe(frame, axis, colour1='k', colour2='k', colour3='k', lbl1='', lbl2='', lbl3='', style='solid', ordr=None)

Plots a reference frame with three axes in a 3D plot.

Parameters:
- **frame**: 3x3 Numpy array containing the three unit vectors in the desired directions.
- **axis**: Matplotlib axis to be plotted on.
- **colour1**: Colour of the X-axis to be plotted.
- **colour2**: Colour of the Y-axis to be plotted.
- **colour3**: Colour of the Z-axis to be plotted.
- **lbl1**: Label of the X-axis to be shown in the legend.
- **lbl2**: Label of the Y-axis to be shown in the legend.
- **lbl3**: Label of the Z-axis to be shown in the legend.
- **style**: Linestyle.
- **ordr**: Zorder of the line (how far it is above or below others).

(Hidden) Function: ensure_list(x, keep_entity=False)

If the input is a list, it will return itself; if the input is not a list, it will return the input as a list.

Parameters:
- **x**: Arbitrary input.
- **keep_entity**: Boolean indicating whether tuples/numpy arrays are transformed to a list, or put into a list.

Returns:
- Input x inside a Python list.

(Hidden) Function: nonzero(x)

Checks whether input x is a nonzero number.

Parameters:
- **x**: Arbitrary input.

Returns:
- Boolean (True if x is a nonzero number; False if x is 0, 0.0, None, np.nan, 'string', ...).

(Hidden) Function: show_tree(model, depth=0)

Prints a model tree in the command line, showing the order of (sub)NodalModels and Nodes. Only the model argument should be entered by the user (do not enter a value for depth).

Parameters:
- **model**: Mother NodalModel object.
- **depth**: At what sublevel the NodalModel is. Used to automatically track the recursive depth of this function.

Function: show_available_materials()

Print the available materials, coatings, and contact connections from Materials.csv in the console. These names can be copy-pasted into the material, coating, and connect arguments of Nodes and NodalModels.

Class Node

"The Node class is an object that defines one node. It contains the thermal and physical properties of a node."

Figure 3.3 can be viewed alongside the following descriptions of the Node class functions.

Function: __init__(name='', material=None, coating=None, C_cap=None, mass=None, volume=None, T0=None, q_ext=None, P_int=None, outer=None, alpha=None, epsilon=None, area=None, origin=(np.nan, np.nan, np.nan), geom=(np.nan, np.nan, np.nan))

Initiates a Node object. Beware of over-defining parameters; for example, applying a coating whilst also manually assigning alpha and epsilon results in two possible values for both alpha and epsilon. In such cases, the code will always prioritise the manually defined value.

Parameters:
- **name**: Unique name used to identify the node within a NodalModel. If not provided, it will be assigned a number when added to a NodalModel.
- **material**: Material object or string containing the bulk properties (to compute C_cap) of the node.
- **coating**: Coating object or string containing the absorptivity (alpha) and emissivity (epsilon) of the node.
- **C_cap**: Heat capacity [J/K].
- **mass**: Mass of the node [kg], only needed if a material is applied and no C_cap is given.
- **volume**: Volume of the node [m^3], only needed if a material is applied and no C_cap or mass is given.
- **T0**: Initial temperature [deg C] or [K].
- **q_ext**: Tuple (q_pla, q_alb, q_s) of external heat fluxes [W/m^2] throughout time. Will be converted to three separate Node attributes called Node.q_pla, Node.q_alb, Node.q_s.
- **P_int**: Internal power [W] throughout time.
- **outer**: Whether the node radiates freely to space or not. Default is False.
- **alpha**: Absorptivity in the solar spectrum. Default is 1.
- **epsilon**: Emissivity in the IR spectrum. Default is 1.
- **area**: Surface area [m^2] of the added node. Default is 1.
- **origin**: Tuple (x, y, z) of origin coordinates of the node. Default is (0, 0, 0).
- **geom**: Tuple (x_width, y_width, z_width) of the rectangular shape of the node. Default is (0, 0, 0).

Function: modify(name_new=None, material_new=None, coating_new=None, C_cap_new=None, mass_new=None, volume_new=None, T0_new=None, q_ext_new=None, P_int_new=None, outer_new=None, alpha_new=None, epsilon_new=None, area_new=None, origin_new=None, geom_new=None)

Adapt values of the Node object. Beware of over-defining parameters; for example, applying a coating whilst also manually assigning alpha and epsilon results in two possible values for both alpha and epsilon. In such cases, the code will always prioritise the manually defined value.

Parameters:

- **name_new**: Unique name used to identify the node within a NodalModel.
- **material_new**: Material object or string containing the bulk properties (C_cap) of the node.
- **coating_new**: Coating object or string containing the absorptivity (alpha) and emissivity (epsilon) of the node.
- **C_cap_new**: Heat capacity [J/K].
- **mass_new**: Mass of the node [kg], only needed if a material is applied and no C_cap is given.
- **volume_new**: Volume of the node [m^3], only needed if a material is applied and no C_cap or mass is given.
- **T0_new**: Initial temperature [deg C] or [K]. Unit must be compatible with the unit of the NodalModel.
- **q_ext_new**: Tuple (q_pla, q_alb, q_s) of external heat fluxes [W/m^2] throughout time. Will be converted to three separate Node attributes called Node.q_pla, Node.q_alb, Node.q_s.
- **P_int_new**: Internal power [W] throughout time.
- **outer_new**: Whether the node radiates freely to space or not. Default is False.
- **alpha_new**: Absorptivity in the solar spectrum. Default is 1.
- **epsilon_new**: Emissivity in the IR spectrum. Default is 1.
- **area_new**: Surface area [m^2] of the added node. Default is 1.
- **origin_new**: Tuple (x, y, z) of origin coordinates of the node. Default is (0, 0, 0).
- **geom_new**: Tuple (x_width, y_width, z_width) of the rectangular shape of the node. Default is (0, 0, 0).

Class NodalModel

"The NodalModel class is an object that defines one nodal model. It contains functions to define and expand the model, and to solve the model."

Figure 3.4 can be viewed alongside the following descriptions of the NodalModel class functions.

Function: __init__(t=None, celsius=True, title='')

Initialises the NodalModel object. The temperatures can be defined in either degrees Celsius or Kelvin; the output unit matches the input.

Parameters:

- **t**: Time vector [s].
- **celsius**: Boolean, whether the temperatures are in Celsius (True) or Kelvin (False). Default is Celsius.
- **title**: Title/name that the NodalModel has.

Function: add_node(node_or_model)

Adds a node (Node object) or collection of nodes (NodalModel object) to the current NodalModel object. Can also add multiple objects at once by feeding a list of objects into the function.

Parameters:

- **node_or_model**: Node or NodalModel object. May also be a list of Nodes and/or NodalModels.

Function: set_time(t, erase=False, print_err=True)

If the NodalModel did not yet have time-varying parameters assigned, this function adds a time array and adds zero-valued entries for all other time-varying parameters. Also sets the same time for sub-NodalModels.
If any previous time data is to be removed and replaced by the new data, provide the input erase=True.

Parameters:

- **t**: Time vector [s].
- **erase**: Boolean to indicate whether previously existing time data is allowed to be erased and replaced by the new data.
- **print_err**: To be ignored by the user. Boolean to avoid printing the same error multiple times.

Function: connect(node1, nodes2, contact_obj=None, rad=None, C_con=None, k_through=None, L_through=None, h_contact=None, A=None)

Defines connections between the first node to be entered (node1) and the specified other nodes (nodes2). This function can be used to define a single connection between two nodes or define multiple connections. Multiple connections are defined using a single node1, and a list of nodes2 and corresponding parameters.

The function adds the connections as: [node1<->nodes2[0], node1<->nodes2[1], node1<->nodes2[2], ...].

If a list of multiple connections is used, type None for the connections that are not desired to be made; for example, for two connections (one conductive, one radiative), C_con would look like C_con=[value, None].

If it is a through-connection and both nodes have the same Material assigned, only L_through and A need to be provided.

Parameters:

- **node1**: Node object, string, or integer indicating the first connected node.
- **nodes2**: (List of) Node object(s), string(s), or integer(s) indicating the second connected node(s).
- **contact_obj**: (List of) Contact object(s), using a pre-defined contact interface.
- **rad**: (List of) Boolean(s) indicating whether the connections include radiative heat transfer.
- **C_con**: (List of) conductance value(s) [W/K] for the corresponding connections.
- **k_through**: (List of) thermal conductivity(ies) [W/(m.K)] between the nodes, for a through-material connection.
- **L_through**: (List of) distance(s) between nodes [m], for a through-material connection.
- **h_contact**: (List of) heat transfer coefficient(s) [W/(m^2.K)], for a contact connection.
- **A**: (List of) orthogonal surface area(s) [m^2] between the nodes.

Function: modify_node(node_mod, name_new=None, material_new=None, coating_new=None, C_cap_new=None, mass_new=None, volume_new=None, T0_new=None, q_ext_new=None, P_int_new=None, outer_new=None, alpha_new=None, epsilon_new=None, area_new=None, origin_new=None, geom_new=None)

Adapt values of an existing Node in the NodalModel. The Node object itself will be changed, as well as all its values within the NodalModel matrices/arrays.

Parameters:

- **node_mod**: Node object, string, or integer indicating the node to be modified.
- **name_new**: Unique name used to identify the node within a NodalModel.
- **material_new**: Material object or string containing the bulk properties (C_cap) of the node.
- **coating_new**: Coating object or string containing the absorptivity (alpha) and emissivity (epsilon) of the node.
- **C_cap_new**: Heat capacity [J/K].
- **mass_new**: Mass of the node [kg], only needed if a material is applied and no C_cap is given.
- **volume_new**: Volume of the node [m^3], only needed if a material is applied and no C_cap or mass is given.
- **T0_new**: Initial temperature [deg C] or [K]. Unit must be compatible with the unit of the NodalModel.
- **q_ext_new**: Tuple (q_pla, q_alb, q_s) of external heat fluxes [W/m^2] throughout time. Will be converted to three separate Node attributes called Node.q_pla, Node.q_alb, Node.q_s.
- **P_int_new**: Internal power [W] throughout time.
- **outer_new**: Whether the node radiates freely to space or not. Default is False.
- **alpha_new**: Absorptivity in the solar spectrum. Default is 1.
- **epsilon_new**: Emissivity in the IR spectrum. Default is 1.
- **area_new**: Surface area [m^2] of the added node. Default is 1.
- **origin_new**: Tuple (x, y, z) of origin coordinates of the node. Default is (0, 0, 0).
- **geom_new**: Tuple (x_width, y_width, z_width) of the rectangular shape of the node. Default is (0, 0, 0).

Function: solve(solver='Radau', printing=True, T0=None, limit_step=False, interp_inputs=False)

Solves the nodal model for the given time vector. The solution is stored in the NodalModel.T array.

Parameters:

- **solver**: Numerical integration method, must be either of: ['rk45', 'dop853', 'radau']. Default is Radau.
- **printing**: Boolean indicating whether the progress should be printed in the console.
- **T0**: Starts the entire model at this T0. Must be the same unit (deg C or K) as the NodalModel.
- **limit_step**: Boolean used for scipy's variable time stepping, indicating whether the solver can use its optimisation algorithms to skip time steps and reduce computational time (limit_step=False), or whether a higher accuracy is desired and no time steps are allowed to be skipped (limit_step=True). Default is False.
- **interp_inputs**: Boolean indicating whether the environmental inputs (q_pla, q_alb, q_s, P_int) should be interpolated during scipy's variable time stepping. Improves accuracy but also increases computational time. Default is False.

Function: show_plots(showrad=True, shownodes=True, showtemps=True, showflows=True, whichnodes=None)

Standard method to plot the temperatures, external radiation, heat flows, a 3D plot of the nodes, and a hierarchical tree of the NodalModel.

Parameters:

- **showrad**: Boolean whether the external heat fluxes are shown.
- **shownodes**: Boolean whether the 3D plot with nodes and hierarchical tree (printed in console) are shown.
- **showtemps**: Boolean whether the transient temperatures are shown.
- **showflows**: Boolean whether the conductive and radiative heat flows are shown.
- **whichnodes**: Node object, string, or integer indicating which nodes are to be shown in all plots. This also means that only connections between the selected nodes are shown.

Class OrbitalModel

"The OrbitalModel class is an object that defines an orbit, its discretisation, and any (angled) plates that receive radiation. It can then calculate the planet IR, albedo, and solar fluxes."

Figure 3.2 can be viewed alongside the following descriptions of the OrbitalModel class functions.

Function: __init__(h, surfaces, beta=None, RAAN=None, incl=None, day=1, n_orbits=1., dtheta=None, dt=None, angular_rates=None)

Initialises an OrbitalModel object. Beta angle can be specified directly, or via the RAAN and inclination. If surfaces is/contains a Node object, its property Node.name must include the direction, so for example 'x+' would work, but also 'x+01' would work.

Parameters:

- **h**: Orbital altitude [m].
- **surfaces**: Either a (list of) Node object(s), or string with direction ('x+' or 'y-' etc.), or a tuple/list with the tau and phi spherical angles. If it is a Node object, its property Node.name must include the direction, so for example 'x+' would work, but also 'x+01' would work.
- **beta**: Solar declination [deg]. MUST BE DEGREES, will be converted to radians.
- **RAAN**: Right ascension of the ascending node [deg] of the satellite's orbit. MUST BE DEGREES.
- **incl**: Inclination [deg] of the satellite's orbit with respect to the Earth's equator. MUST BE DEGREES.
- **day**: Number of days from January 1 (January 1 itself is day 1).
- **n_orbits**: Number of orbits to be calculated. Default is 1.
- **dtheta**: Step size [deg] in true anomaly (theta). MUST BE DEGREES, will be converted to radians. Default is 0.01 rad.
- **dt**: Step size [s] in time. Can be entered instead of dtheta. If both are given, dtheta overrides dt.
- **angular_rates**: Angular velocities [deg/s] around the x, y, and z axes. MUST BE DEGREES / S.

Function: modify(h_new=None, surfaces_new=None, beta_new=None, RAAN_new=None, incl_new=None, day_new=None, n_orbits_new=None, dtheta_new=None, dt_new=None, angular_rates_new=None)

Adapt values of the OrbitalModel object. Beware that using this function (particularly surfaces_new, dtheta_new, or dt_new) will reset previously computed outputs, since the shape of the heat flux array must change to accommodate the changes (hence, the previous results are not discarded).

Parameters:

- **h_new**: Orbital altitude [m].
- **surfaces_new**: Either a (list of) Node object(s), string with direction ('x+' or 'y-' etc.), or a tuple/list with the tau and phi spherical angles. If it is a Node object, its property Node.name must include the direction, so for example 'x+' would work, but also 'x+01' would work.
- **beta_new**: Solar declination [deg]. MUST BE DEGREES, will be converted to radians.
- **RAAN_new**: Right ascension of the ascending node [deg] of the satellite's orbit. MUST BE DEGREES.
- **incl_new**: Inclination [deg] of the satellite's orbit with respect to the Earth's equator. MUST BE DEGREES.
- **day_new**: Number of days from January 1 (January 1 itself is day 1).
- **n_orbits_new**: Number of orbits to be calculated. Default is 1.
- **dtheta_new**: Step size [deg] in true anomaly (theta). MUST BE DEGREES, will be converted to radians.
- **dt_new**: Step size [s] in time. Can be entered instead of dtheta. If both are given, dtheta overrides dt.
- **angular_rates_new**: Angular velocities [deg/s] around the x, y, and z axes. MUST BE DEGREES / S.

Function: animate_attitude(speed=1, realtime=False)

Shows an animation (previous plotting windows must be closed) of a line or collection of lines that are rotating throughout time.

Parameters:

- **speed**: Default is 1, showing all time frames. If it is increased, time frames are skipped to speed up the animation.
- **realtime**: If True, the duration between each time step is set to the dt of the simulation.

Function: compute(printing=True)

Run the orbital model (func heat_received) and store the results in self.q_pla, self.q_alb, and self.q_s. If the surfaces are of type Node, the heat fluxes will directly be assigned to the nodes. However, if those nodes are already in a NodalModel, the heat fluxes must also be assigned to the NodalModel with:
NodalModel.modify_node(node, q_ext_new=OrbitalModel.get_heat(node)), or alternatively:
NodalModel.modify_node(node, q_ext_new=tuple(node.q_pla, node.q_alb, node.q_s))

Parameters:

- **printing**: Boolean indicating whether the progress should be printed in the console.

Function: get_heat(index)

Return the planet, albedo, and solar flux as a tuple, for the given surface.

Parameters:
- **index**: Surface given as an integer (order in the list self.surfaces), string, or Node.

Returns:
- Tuple (q_pla, q_alb, q_s) in [W/m^2].

C.6.6. SensitivityAnalysis.py

"SensitivityAnalysis.py contains numerous functions to perform a sensitivity analysis on your custom models. The 'run_...' functions compute the cases and store them in Pickle (.pkl) files. The plot_...' functions read those files and plot them accordingly."

(Hidden) Function: get_file(file)

Returns the path of a file in the same directory as this Python file; works on most operating systems.

Parameters:
- **file**: Name (str) of the file to be retrieved.

Returns:
- Path of the file.

(Hidden) Function: get_folder_file(folder, file, subfolders=())

Returns the path of a file in a folder within the same directory as this Python file; works on most operating systems.

Subfolders can be an arbitrary number of sub-folders, such as:
get_folder_file('folderA', 'file.txt', ('folderB', 'folderC'))
–> C:/...your-directory.../folderA/folderB/folderC/file

Parameters:
- **folder**: Name (str) of the folder which the file is in.
- **file**: Name (str) of the file to be retrieved.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.

Returns:
- Path of the file.

(Hidden) Function: copy_orbital_nodalmodel(orbital_old, nodal_old)

Makes a deepcopy of an OrbitalModel and NodalModel object, while keeping the links between them. This is needed for the sensitivity analysis, since the original object should remain unchanged.

Parameters:
- **orbital_old**: OrbitalModel object to be copied.
- **nodal_old**: NodalModel object to be copied.

Returns:
- The copied OrbitalModel object.
- The copied NodalModel object.

(Hidden) Function: calc_rmse(y_short, y_long, multip)

Calculate the Root Mean Square Error (RMSE) between two differently shaped, but overlapping, arrays. The short array (large time step) only has fewer data points than the long array (small time step). If multip is 2, y_long is twice as long as y_short, i.e. the time step of y_long is twice as small as that of y_short. The x-values of y_short must be present in y_long; for example: x_short = [0, 2, 4] and x_long =[0, 1, 2, 3, 4] for multip = 2.

Parameters:
- **y_short**: Y-values of the short array (large time step).
- **y_long**: Y-values of the long array (small time step).
- **multip**: Integer value showing by which multiple the time step differs. See example above.

Returns:
- Root Mean Square Error (RMSE) between y_short and y_long.

(Hidden) Function: read_folder(folder='SensitivityAnalysis', subfolders=())

Reads all pickle files available in the given folder.

Parameters:
- **folder**: Name (str) of the folder where the pickle files are to be read from.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.

Returns:
- List of names of the cases/files.
- List of the solved NodalModels for all cases.

Function: run_case(variable, value, nodal_model, orbital_model, printing=True, solver='Radau', limit_step=False, interp_inputs=False)

Runs an orbital and transient analysis for the given input parameters.

- The parameter 'value' is the relative fraction for all parameters (except DAY, BETA, DT, and RAD), since the use of absolute numbers would potentially alter the entire nature of the NodalModel.
- If the altitude or time step is changed, the time array changes as well. Hence, the internal power cannot follow the exact same transient as before (except if it is constant), so an average internal power will be assumed along the entire timeline. If the power is constant, this is no issue and the same value is used.
- If any input parameter is by default zero, any multiplications (+-10% etc.) still result in zero; this could be a reason why the sensitivity analysis has no impact on certain parameters, e.g. if the internal power is zero.

Parameters:

- **variable**: Name (string) of the variable which is being changed.
- **value**: Multiplication factor if the variable is either of: [ALT, CAP, IR, ALB, SOL, POW, CON, EMI, ABS, ROT], and the actual absolute value if the variable is either of: [DAY, BETA, DT, RAD]. For RAD, the value should be 'on' or 'off' (string).
- **nodal_model**: NodalModel to be subjected to the sensitivity analysis.
- **orbital_model**: OrbitalModel to be subjected to the sensitivity analysis.
- **printing**: Boolean indicating whether progress should be printed in the command line.
- **solver**: Numerical integration method, must be either of: ['rk45', 'dop853', 'radau']. Default is Radau.
- **limit_step**: Boolean used for scipy's variable time stepping, indicating whether the solver can use its optimisation algorithms to skip time steps and reduce computational time (limit_step=False), or whether a higher accuracy is desired and no time steps are allowed to be skipped (limit_step=True). Default is False.
- **interp_inputs**: Boolean indicating whether the environmental inputs (q_pla, q_alb, q_s, P_int) should be interpolated during scipy's variable time stepping. Improves accuracy, but also increases computational time. Default is False.

Returns:

- NodalModel object with the solved case.

Function: run_all(dev1, dev2, nodal_model, orbital_model, overwrite=False, folder='SensitivityAnalysis', subfolders=(), printing=True, solver='Radau', limit_step=False, interp_inputs=False)

Runs all sensitivity analysis cases and writes them to pickle files.
If overwrite is True, existing pickle files with the same name are overwritten.

Subfolders can be an arbitrary number of sub-folders, such as:
run_all(..., folder='folderA', subfolders=tuple('folderB', 'folderC'))
–> C:/...your-directory.../folderA/folderB/folderC/file
The file name is automatically generated.

Parameters:

- **dev1**: # +− small deviation (fraction) for ALT, CAP, IR, ALB, SOL, POW, CON, EMI, and ABS.
- **dev2**: # +− large deviation (fraction) for ALT, CAP, IR, ALB, SOL, POW, CON, EMI, and ABS.
- **nodal_model**: NodalModel to be subjected to the sensitivity analysis.
- **orbital_model**: OrbitalModel to be subjected to the sensitivity analysis.
- **overwrite**: Boolean to determine whether existing pickle files with the same name are to be overwritten.
- **folder**: Name (str) of the folder in which the files are to be exported.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.
- **printing**: Boolean indicating whether progress should be printed in the command line.
- **solver**: Numerical integration method, must be either of: ['rk45', 'dop853', 'radau']. Default is Radau.
- **limit_step**: Boolean used for scipy's variable time stepping, indicating whether the solver can use its optimisation algorithms to skip time steps and reduce computational time (limit_step=False), or whether a higher accuracy is desired and no time steps are allowed to be skipped (limit_step=True). Default is False.
- **interp_inputs**: Boolean indicating whether the environmental inputs (q_pla, q_alb, q_s, P_int) should be interpolated during scipy's variable time stepping. Improves accuracy, but also increases computational time. Default is False.

Function: run_variable(name, values, nodal_model, orbital_model, overwrite=False, folder='SensitivityAnalysis', subfolders=None, printing=True, solver='Radau', limit_step=False, interp_inputs=False)

Runs arbitrary analysis cases for a given variable, using the other default values, and writes them to pickle files. The difference with the run_all function is that run_variable is not limited to deviation percentages. Any array with values can be used. Furthermore, it will result in different plots (plot_variable).
The values for IR, ALB, and SOL parameters must be multiplication factors (actual value is computed automatically).
If overwrite is True, existing pickle files with the same name are overwritten.

The parameter 'value' is a list of the relative fractions with which the variable should be multiplied (except for DAY, BETA, DT, for which the values are the actual absolute values).

Subfolders can be an arbitrary number of sub-folders, such as:
run_variable(..., folder='folderA', subfolders=tuple('folderB', 'folderC'))
–> C:/...your-directory.../folderA/folderB/folderC/file
The file name is automatically generated.

Parameters:

- **name**: Name (str) of the variable. Must be one of: ['DAY', 'ALT', 'BETA', 'CAP', 'IR', 'ALB', 'SOL', 'POW', 'CON', 'EMI', 'ABS', 'ROT', 'DT', 'RAD']. Internal radiation (RAD) is recommended to be used with its own function (plot_intrad).
- **values**: List/tuple/array of values. Is a multiplication factor if the variable is either of: [ALT, CAP, IR, ALB, SOL, POW, CON, EMI, ABS, ROT], and the actual absolute value if the variable is either of: [DAY, BETA, DT]. RAD would be either 'on' or 'off'.
- **nodal_model**: NodalModel to be subjected to the sensitivity analysis.
- **orbital_model**: OrbitalModel to be subjected to the sensitivity analysis.
- **overwrite**: Boolean to determine whether existing pickle files with the same name are to be overwritten.
- **folder**: Name (str) of the folder in which the files are to be exported.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.
- **printing**: Boolean indicating whether progress should be printed in the command line.
- **solver**: Numerical integration method, must be either of: ['rk45', 'dop853', 'radau']. Default is Radau.
- **limit_step**: Boolean used for scipy's variable time stepping, indicating whether the solver can use its optimisation algorithms to skip time steps and reduce computational time (limit_step=False), or whether a higher accuracy is desired and no time steps are allowed to be skipped (limit_step=True). Default is False.
- **interp_inputs**: Boolean indicating whether the environmental inputs (q_pla, q_alb, q_s, P_int) should be interpolated during scipy's variable time stepping. Improves accuracy, but also increases computational time. Default is False.

Function: run_dt_rot(name, nodal_model, orbital_model, max_val=None, point_density=1., overwrite=False, folder='SensitivityAnalysis', subfolders=None, printing=True, solver='Radau', limit_step=False, interp_inputs=False)

Generates data points for a temperature versus angular velocity graph, then executes run_variable. Those data points (time steps or angular rates) are determined automatically, based on the expected time step needed to resolve a 90-degree rotation.
For the angular rates analysis (name='ROT'), set those angular rates to (1., 1., 1.) so that the multiplication factor matches with the actual degrees per second.

It is recommended to make extra sub-subfolders, such as: subfolders=('rot', 'dt10') or ('dt', 'rot-z').
If overwrite is True, existing pickle files with the same name are overwritten.

Subfolders can be an arbitrary number of sub-folders, such as:
run_dt_rot(..., folder='folderA', subfolders=tuple('folderB', 'folderC'))
–> C:/...your-directory.../folderA/folderB/folderC/file
The file name is automatically generated.

Parameters:

- **name**: Name (str) of the variable. Must be either 'ROT' or 'DT'.
- **nodal_model**: NodalModel to be subjected to the sensitivity analysis.
- **orbital_model**: OrbitalModel to be subjected to the sensitivity analysis.
- **max_val**: Maximum value of the angular rate [deg/s] or time step [s] to be computed. Defaults are 20 deg/s and 100 s.
- **point_density**: Multiplication factor for the number of points generated. Default of 1.0 means that the likely most optimal number of points is used. Setting the value to 2.0 computes twice as many points, or setting it to 0.5 halves the number of points, for example.
- **overwrite**: Boolean to determine whether existing pickle files with the same name are to be overwritten.
- **folder**: Name (str) of the folder in which the files are to be exported.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.
- **printing**: Boolean indicating whether progress should be printed in the command line.
- **solver**: Numerical integration method, must be either of: ['rk45', 'dop853', 'radau']. Default is Radau.
- **limit_step**: Boolean used for scipy's variable time stepping, indicating whether the solver can use its optimisation algorithms to skip time steps and reduce computational time (limit_step=False), or whether higher accuracy is desired (limit_step=True). Default is False.
- **interp_inputs**: Boolean indicating whether the environmental inputs (q_pla, q_alb, q_s, P_int) should be interpolated during scipy's variable time stepping. Improves accuracy but increases computational time. Default is False.

Function: run_integrators(nodal_model, orbital_model, values=None, overwrite=False, folder='SensitivityAnalysis', subfolders=('dt', 'integrators'), printing=True)

Run a number of cases with different integrators. Mainly used for thesis report.
If overwrite is True, existing pickle files with the same name are overwritten.

Subfolders can be an arbitrary number of sub-folders, such as:
run_dt_rot(..., folder='folderA', subfolders=tuple('folderB', 'folderC'))
–> C:/...your-directory.../folderA/folderB/folderC/file
The file name is automatically generated.

Parameters:

- **nodal_model**: NodalModel to be subjected to the sensitivity analysis.
- **orbital_model**: OrbitalModel to be subjected to the sensitivity analysis.
- **values**: Time step values [s] at which to evaluate the models.
- **overwrite**: Boolean to determine whether existing pickle files with the same name are to be overwritten.
- **folder**: Name (str) of the folder in which the files are to be exported.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.
- **printing**: Boolean indicating whether progress should be printed in the command line.

Function: run_intrad(nodal_model, orbital_model, overwrite=False, folder='SensitivityAnalysis', subfolders=('rad',), printing=True, solver='Radau', limit_step=False, interp_inputs=False)

Runs the analysis cases for different internal radiation scenarios, using the other default values, and writes them to pickle files.
If overwrite is True, existing pickle files with the same name are overwritten.

Subfolders can be an arbitrary number of sub-folders, such as:
run_intrad(..., folder='folderA', subfolders=tuple('folderB', 'folderC'))
–> C:/...your-directory.../folderA/folderB/folderC/file
The file name is automatically generated.

Parameters:

- **nodal_model**: NodalModel to be subjected to the sensitivity analysis.
- **orbital_model**: OrbitalModel to be subjected to the sensitivity analysis.
- **overwrite**: Boolean to determine whether existing pickle files with the same name are to be overwritten.
- **folder**: Name (str) of the folder in which the files are to be exported.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.
- **printing**: Boolean indicating whether progress should be printed in the command line.
- **solver**: Numerical integration method, must be either of: ['rk45', 'dop853', 'radau']. Default is Radau.
- **limit_step**: Boolean used for scipy's variable time stepping, indicating whether the solver can use its optimisation algorithms to skip time steps and reduce computational time (limit_step=False), or whether higher accuracy is desired (limit_step=True). Default is False.
- **interp_inputs**: Boolean indicating whether the environmental inputs (q_pla, q_alb, q_s, P_int) should be interpolated during scipy's variable time stepping. Improves accuracy but increases computational time. Default is False.

Function: plot_all(folder='SensitivityAnalysis', subfolders=())

Plot all sensitivity analysis cases from the given (sub)folder.

Parameters:
- **folder**: Name (str) of the folder where the pickle files are to be read from.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.

Function: plot_variable(name, folder='SensitivityAnalysis', subfolders=(), whichnode=None)

Plot temperature data for varying values of the given variable name, from the given (sub)folder. The default node to be plotted for the transient (temperature-time and heat flux-time) graphs is the first node of the model (node 0). It is recommended that a node is chosen which is reasonably representative of the spacecraft temperature. For all other plots, the temperatures of all nodes are used to extract min/mean/max values.

Parameters:
- **name**: Name (str) of the variable. Must be one of: ['DAY', 'ALT', 'BETA', 'CAP', 'IR', 'ALB', 'SOL', 'POW', 'CON', 'EMI', 'ABS', 'ROT', 'DT', 'RAD']. Internal radiation (RAD) is recommended to be used with its own function (plot_intrad).
- **folder**: Name (str) of the folder where the pickle files are to be read from.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.
- **whichnode**: Node object, string, or integer indicating the node to be plotted for the transient graphs.

Function: plot_dt_rot(name, folder='SensitivityAnalysis', subfolders=())

Plot temperature data for varying values of the given variable name, from the given (sub)folder.

Parameters:
- **name**: Name (str) of the variable. Must be either 'ROT' or 'DT'.
- **folder**: Name (str) of the folder where the pickle files are to be read from.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.

Function: plot_integrators(benchmark_filename, benchmark_folder, plot_extra=False, folder='SensitivityAnalysis', subfolders=('dt', 'integrators'))

Plot convergence data for different time steps. Mainly used for thesis report.

Parameters:
- **benchmark_filename**: Name of the file which is considered as the benchmark computation.
- **benchmark_folder**: Name of the folder in which the benchmark is located (is assumed to be in the same directory as the subfolders parameter).
- **plot_extra**: Boolean indicating whether extra figures should be plotted, showing the transient temperatures for each time step. Each time step is shown in a separate figure (many figures will appear).
- **folder**: Name (str) of the folder where the pickle files are to be read from.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.

Function: plot_intrad(whichnodes=None, folder='SensitivityAnalysis', subfolders=())

Plot temperature data for varying internal radiation conditions from the given (sub)folder.

Parameters:
- **whichnodes**: String or integer (not Node) indicating which nodes are to be shown in all plots. This also means that only connections between the selected nodes are shown.
- **folder**: Name (str) of the folder where the pickle files are to be read from.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.

C.6.7. VerificationValidation.py

"VerificationValidation.py computes numerous cases and compares them to ESATAN data and FUNcube flight data. Mainly used for the thesis report; not intended to be used by the general user."

This file does not contain any functions or classes that are relevant to the general user. It contains specific verification and validation functions related to Chapter 4 of this thesis.

(Hidden) Function: get_file(file)

Returns the path of a file in the same directory as this Python file; works on most operating systems.

Parameters:
- **file**: Name (str) of the file to be retrieved.

Returns:
- Path of the file.

(Hidden) Function: get_folder_file(folder, file, subfolders=())

Returns the path of a file in a folder within the same directory as this Python file; works on most operating systems.

Subfolders can be an arbitrary number of sub-folders, such as:
get_folder_file('folderA', 'file.txt', ('folderB', 'folderC'))
–> C:/...your-directory.../folderA/folderB/folderC/file

Parameters:
- **folder**: Name (str) of the folder which the file is in.
- **file**: Name (str) of the file to be retrieved.
- **subfolders**: Tuple/list of names (str) of any sub-folders which the file is in.

Returns:
- Path of the file.

C.6.8. FUNcube.py

"FUNcube.py contains a reconstruction of the thermal model of the FUNcube-1 satellite. This file can also be considered as an example on how to create orbital models and nodal models. However, there are generally more ways in which the code can be used, so it is recommended to refer to the user's manual for more details."

This file does not contain any functions or classes.