POLITECNICO DI TORINO

Master's Degree program

in Aerospace Engineering

Master's thesis

Development and validation of an "in-the-loop" simulator for small satellites

Supervisors

Prof. Sabrina Corpino

Eng. Fabrizio Stesina

Candidate

Sorba Roberto

JULY 2018

Abstract

This thesis represents the synthesis of almost two years of collaboration within the CubeSat team at the Polytechnic University of Turin, and its scope is to illustrate the development of a user-friendly, multi-functional simulator for small satellites.

This simulator, called by the proprietary name of StarSim v2.0, will have to perform at least three different kind of in-the-loop simulation: Algorithm-in-the-loop, Software-in-the-loop and Hardware-in-the-loop, and each of these has to be consistent with the previous ones.

In the first chapter, a general introduction to System Engineering is provided, completed with the necessary knowledge to Model-Based Design and Validation techniques, upon which all the rest of this work is based.

Chapter 2 shows in details the completed product, completed with user experience description, basic source code explanation and graphic diagrams to explain the general architecture of the Software for the three different operative modes.

In Chapter 3 the standard library in detailed: this library of algorithmical models have been developed in parallel to StarSim and provides all the basic models to perform a standard simulation, such as the orbit propagator, the sun vector computer, the solar panel, the battery models and so on.

Chapter 4 is devoted to the case study, a comprehensive dynamic modeling and simulation of the electric power system of an orbiting CubeSat, from the Algorithms to the actual hardware, which fulfills the aim of validating all the previous work.

Contents

Chapter 1: System Engineering and Simulation Technologies.........9

1.1) System engineering for Aerospace professional.........9
1.1.1) Model-Based System Engineering 12
1.1.2) UML & SysML.........13
1.1.3) Benefits of MBSE.........15

1.2) Theory of simulation.........16
1.2.1) Types of simulations.........18
1.2.2) Model Based Testing and Validation.........20
1.2.3) Model Based Development and Verification.........22

1.3) Simulator features.........25

Chapter 2: Using the simulator.........27

2.1) Usage of the simulator.........27
2.2) Main window 28
2.3) AIL simulation.........29
2.3.1) Setting conditions.........30
2.3.2) Simulation setup.........32
2.3.3) Plotting.........34

2.4) SIL simulation.........35
2.4.1) SIL communication setup.........36
2.4.2) Initial values.........37

2.5) Source code analysis.........38
2.5.1) Pre-processing.........47
2.5.1.1) GUI spawner.........47
2.5.1.2) Model Parser.........48

2.5.1.3) Model lister……50
2.5.1.4) Parameter parser……51

2.6) Process.py……53
2.7) Post-processing ……55

Chapter 3: The standard library……56

3.1) Mission models……56
3.1.1) Orbit propagator……56
3.1.2) Sun motion……60
3.1.4) Magnetic field (dipole)……62
3.1.5) Thermal fluxes……64

3.2) Power sources and actuators……65
3.2.1) Costant resistive load……66
3.2.1) Solar panel (full)……66
3.2.1) MPPT……69
3.2.1) Battery……70

Chapter 4: Case study……72

4.1) Modeling the Electric Power System……72
4.1.1) Dissipated power……73
4.1.2) Solar Panel……74
4.1.3) Battery……74
4.1.4) Full scheme……75

4.2) AIL EPS setup and scenarios……76
4.2.1) Single panel, fully irradiated……76
4.2.2) Five panels, standard orbit……79

4.3) AIL results analysis 80
4.4) HIL setup..84
4.4.1) Hardware integration..87
4.4.2) HIL process...89
4.4.3) HIL results and comparison..90

Conclusion and further development..92

Bibliography..93

Appendix A: Models code...95

A.1) Orbit propagator..95
A.2) Sun motion...97
A.3) Sun vector...100
A.4) Magnetic field (dipole)..101
A.5) Constant resistive load..102
A.6) Solar panel (full)...103
A.7) battery..106
A.8) thermal fluxes..107

Appendix B: Simulator code...110

A.1) Model parser..110
A.2) Parameter parser...112
A.3) Simulation setup..115
A.4) self-generated *Process* file for case study...121

List of Figures

Picture 1-1 : Project phases 11
Picture 1-2: structure of modeling languages 14
Picture 1-3: block structure of s SysML diagram 15
Picture 1-4: standard V-model for system engineering..23
Picture 2-1: simulator schematics...28
Picture 2-2: StarSim main window...29
Picture 2-3: StarSim project window 30
Picture 2-4: setting condition ona certain model 31
Picture 2-5: sample orbit simulator setup 32
Picture 2-6: Set parameters window 33
Picture 2-7: select desired output 34
Picture 2-8: plotting window 35
Picture 2-9: select software processes for SIL simulation 36
Picture 2-10: open new pipe...36
Picture 2-11: pipes setup 37
Picture 2-12: setting initial arguments 38
Picture 2-13: StarSim complete flow diagram 46
Picture 2-14: GUI action diagram..48
Picture 2-15: model parser action diagram 49
Picture 2-16: model lister action diagram 50
Picture 2-17: parameter parser action diagram 53
Picture 2-18: main simulation process 54
Picture 3-1: sample orbit propagator results 58
Picture 3-2: numerically diverging orbit 59
Picture 3-3: numerically stable orbit for a time step lesser than 0.01 59
Picture 3-4: standard solar panel charachteristc curve 68
Picture 3-5: solar panel curve implemented in MATLAB for testing 69
Picture 3-6: MPPT perturbe & observe algorithm 70
Picture 4-1: CubeSat's operating modes 73
Picture 4-3: solar cell features 74

Picture 4-2: solar cell size......74
Picture 4-4: battery features......75
Picture 4-5: battery size......75
Picture 4-6: full EPS simplified model......75
Picture 4-7: schematic model for single-panel simulation......77
Picture 4-8: EPS simulation implemented in StarSim project window......78
Picture 4-9: parameters setup for EPS simulation......78
Picture 4-10: three panels simulation setup......79
Picture 4-11: parameters setup for full-panels simulation......80
Picture 4-12: solar panel voltage results......80
Picture 4-13: solar panel current results......81
Picture 4-14: current drawn from battery......81
Picture 4-15: battery capacity results......82
Picture 4-16: load voltage results......82
Picture 4-17: battery actual voltage......83
Picture 4-18: current drawn from battery......84
Picture 4-19: addding hardware interface......85
Picture 4-21: isotech-ips-3202 generator......86
Picture 4-20: channel setup......85
Picture 4-22: RS232 standard......86
Picture 4-23: laboratory setup......88
Picture 4-24: HIL setup schematics......89
Picture 4-25: actual voltage results......91

Acronyms

ADCS: Attitude Determination and Control System
AIL: Algorithm in the loop
COMSYS: Communication System
CIL: Computer in the loop
EPS: Electric Power System
HIL: Hardware in the loop
INCOSE: International Council of System Engineers
MBDV: Model Based Design and Validation
MBSE: Model Based System Engineering
MPPT: Maximum Power Point Tracking
OBC: On-Board Computer
OBHW: On-Board Hardware
ONSW: On-Board Software
PCDU: Power Control and Distribution Unit
SIL: Software in the loop
SP: Solar Panel
SySML: System Modeling Language
UML : Unified Modeling Language
WMM: World Magnetic Model

Chapter 1

1 System Engineering and Simulation Technologies

1.1) System Engineering for Aerospace professionals

A *system* is a more or less articulated set of elements, created to the purpose of obtaining a specific goal.

Complexity of a system arises from its structural relationships and the dynamics of its component entities, and it's not fully determined by its size alone.

A system consists in components or block which, considered in their mutual interaction, run towards a shared target.

Main feature of a system is the fact that block and components, through this interaction, are meaningful only if considered as integral part of the system and may miss the target or loose significance if taken singularly.

The definition of system engineering, as given by the *International Council of System Engineering* (INCOSE) is:

"Systems Engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem"

System engineering is concerned with:

1) Definition and documentation of system requirements in the early feasibility study.
2) Development phase in which the project begins taking its own shape
3) Testing of the completed system

System engineering comes into account along the entire project development process, starting from the first creative idea and going thorough design, assembly, verification and validation, encompassing both technical and economical aspects.
It is important to define also the *systems-of-systems* (*SoS*), aggregates of heterogeneous systems coordinated and interacting among themselves called subsystems.
The overall system thus will be a set of subsystems, developed in order to obtain one or more goals.

The definition of system engineering encompasses the mutual dependencies and dynamics among the different components of a system; within system engineering some conventional terms play a major role due to their current widespread use, such as System, element, subsystem, assembly, subassembly, component, part.
These terms intensify the overall complexity of the system in a more and more detailed way, going from the general to the specific.
System engineering as a topic doesn't come yet with a single modeling language and has to free itself completely from the specific technical subjects that constitute the system: its main concept, as well as its most challenging task, consists in addressing all the blocks and components towards the shared goal, aggregating them in a coherent and logical way.
This target is reached by taking advantage of a common pool of tools and operations both is the stage of requirements analysis and in the stage of project development, such as design, development, verification, testing, validation, integration, documentation, risk analysis and possible future evolution.
In the field of Aerospace Engineering, a major source of information and standardization has been coming insofar from ESA's ECSS E-10 family, a comprehensive set of documents encompassing every possible aspect of System Engineering to be used at system level in ESA projects.
They include:

- ECSS E-10 Standards
- ECSS E-10 Handbooks
- ECSS E-10 Technical Memoranda
- ISO Norms

- ESSB Handbooks

These standards shall be used (possibly after tailoring) to *complement* a project's own specific requirements documents, which traditionally include:

- Mission or System Requirements Document (MRD/SRD)
- Tasks description
- Documents for Interfaces (with Launcher Authority, Payload, etc.)
- Specific documents (Environment definition, Regulations, etc.)

This impressive amount of documents provides a general description and guidelines on system engineering tasks in the field of Aerospace Engineering, as well as partitioning them per project phases: it defines what should be available from system viewpoint at the end of each phase.

The outline of project development phases is described by the following table:

Project phases				
Phase A	Phase B	Phase C	Phase D	Phase E
Mission analysis **· Development of system concept and configuration alternatives** **· Analysis of these concepts and configurations, "system-trade-offs"** **· Development of standardized documentation for the selected variant**	System design refinement and design verification · Development and verification of system and equipment specifications · Functional algorithm design and performance verification · Design support regarding interfaces and budgets	Subcontracting of component manufacturing · Detailed design of components and system layout · EGSE development and test · On-board software development and verification · Development and validation of test procedures · Unit and subsystem tests	Software verification · System integration and tests · Validation regarding operational and functional performance · Development and verification of flight procedures	Ground segment validation · Operator training · Launch · In-orbit commissioning · Payload calibration · performance evaluation · Prime contractor provides trouble shooting support for spacecraft

Picture 1-1 : Project phases

1.1.1) Model based system engineering

It can be noticed from the previous paragraphs how the traditional approach to System Engineering is strongly document-based, and this began to cause an increasing number of concerns to involved professionals.

In facts, its main challenge consists in creating unequivocal and formally correct specification documents, in which every relevant information to all the stakeholders participating in the projects are to be presented in a manner promoting synchronization and compatibility among all the different disciplines.

Until recently there wasn't a single tool commonly used to support these writing processes, and problems began to arise as System Engineers had to interface with Software Engineers.

Interaction between system- and software engineering plays indeed an important role in making sure that system requirements will be translated correctly in the correspondent software application.

The *Model Based System Engineering* concept was born in this context, in order to avoid unnecessary duplication of information and parallel development between System- and Software teams.

In fact, System- and Software Engineers work at different abstraction levels and with different points of view: while System Engineering is concerned with the definition of *what* has to be created, the purpose of Software Engineering is to define *how* that goal is achieved.

Both activities can work through models, which are anyway of a consistently different kind: in System Engineering the model is an *abstraction* of the real system, while the software developer thinks of the model as a good *decomposition* of the real system; possible discrepancies have to be taken care of in the verification phase.

Conventional solutions for System Engineering are mostly based on drawing tools and databases, often correlated with some Excel sheet or similar frontend software.

These solutions however do not specify a standard process or a set of convections, so it's possible that every team is using its own different modeling language or its own tools, besides implementing different overall philosophies.

Model-based solutions have already been common practice among professional software businesses, and are enjoying an increasing popularity also among System engineers.

These solutions offer new concepts for the analysis and development of critical systems in the Space engineering industry: they build heavily on block diagrams and state machines, are formally well-defined and provide a strong and unambiguous tool to describe univocally the behavior of the intended object.

The two major frameworks for MBSE that are presented in the following sections, UML and SysML, fulfill the main goal of a more efficient and effective system engineering by moving from a document-centric to model-centric approach making use of the capabilities that modern computer can offer.

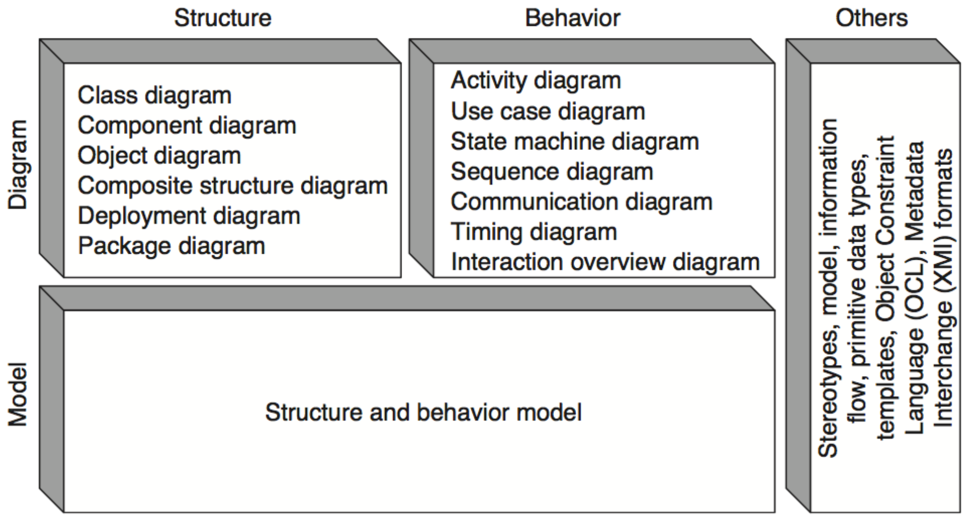
1.1.2) UML & SysML

Unified modeling language has been the first standard graphic language developed by the *Object Management Group* with the purpose of univocally define the prospective behavior of a software.

Given its generalist nature, it makes use of graphical notations in order to create abstract models starting from complete systems or partial subsystems: this allows for easy specification, visualization, development and documentation of particular software.

UML consists of two distinct levels, **Model** and **Diagram**, and features a modular structure, so that it is possible to work with just one part of the language without losing accuracy in the system modeling process.

The model level is concerned with the complete description of the system, while the diagram level consists of various portions, each analyzing only one specific aspect:



Picture 1-2: structure of modeling languages

In April 2006, its successor SysML (*Sysem Modeling Language*) has been accepted as standard, even though actually this standardization process has prolonged by almost one year until OMG published version 1.0 in September 2007.

SYSML is a graphical modeling language, extending the capabilities of UML to the main purpose of allowing the description, analysis and verification of more complex systems under multiple points of view such as hardware and software as well as database management, human resources management and other business-related considerations.

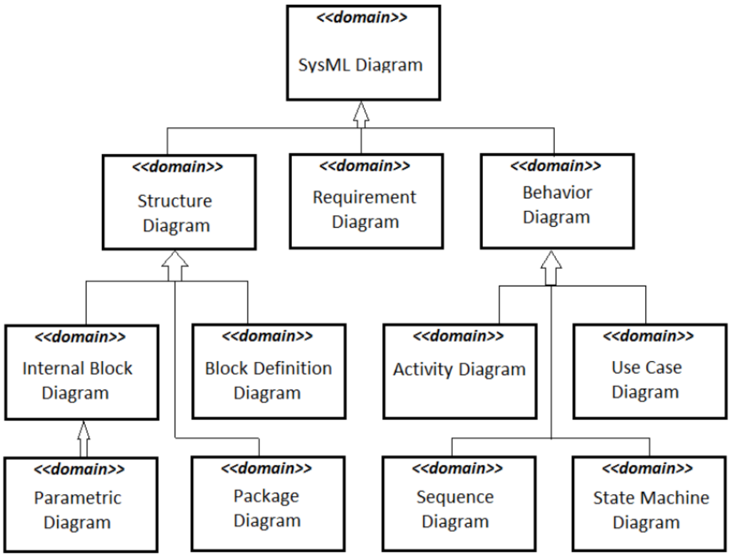
In the framework of SysML it is possible to re-use many diagrams already present in UML; some other are being renamed and extended, and only two are totally brand new: the requirement diagram and the parametric diagram.

SysML also consists of two levels: the Model level and the diagram level;

With respect to UML, the framework has been sub-divided into three sections: Structural, Behavioral and Generic.

SysML is based on four main pillars:

1. **Structure**: System hierarchies, Interconnections (block diagram, internal block diagram)
2. **Behavior**: Function-Based Behaviors, State-Based Behaviors (use case, interaction, activity, state diagrams)
3. **Requirements**: Requirements Hierarchies, Traceability
4. **Proprieties**: Parametric Models, Time Variable Attributes



Picture 1-3: block structure of s SysML diagram

1.1.3) Benefits of MBSE

MBSE offers several advantages over its traditional document-centered competitor:

- Improved communications, as it is quicker and easier to share information in the form of graphical diagrams, independently of the tool and methodology of the specific field.

- Assists in managing complex system development: complex systems can be represented efficiently in SysML in a compact and standardized way, whereas this would require potentially hundreds of pages in verbose, document-based form.

- Separation of concerns: each specialist can model its own system independently and the diagrams can be simply integrated at the end, reducing the need for cross-subject text revisions.

- Hierarchical modeling: subsystems can inherit all the properties and method of their parent class and extend them with their own particularities; this translates into one of the core strengths of Object Oriented Programming, which has been widely taken advantage of during the writing of StarSim v.02.

- Supports incremental development & evolutionary acquisition: standard diagrams are easier to extend in time and the result will be much cleaner with respect to the same output in verbose form.

- Improved design quality, because it forces the developers to complete all the nine required diagrams, thus thinking extensively about any possibility that they could have missed or anything that could be improved.

- Reduced errors and ambiguity, since the diagrams are standard.

- More complete representation, since every possible aspect of the system has been taken into account and standardized, so that nothing can be forgotten or omitted.

- Early and on-going verification & validation to reduce risk: this is probably the most significant strength of MBSD with respect to the scope of this work, as it allows the simulation to start before the system is actually assembled or even before it is completely defined.

- Enhanced knowledge capture, as information provided in graphic form as usually easier to retain and the process of drawing the required number of diagram forces the developers to grab a good overall understanding of the system in object.

1.2) Theory of simulation

The concept of simulation theory represents the first step towards the realization of the goal to perform a system simulation for a small satellite. Authors such as Law, Kelton and Zeigler provide in their books *Simulation Modeling and Analysis* [Law & Kelton '00] and *Theory of Modeling and Simulation* [Zeigler '76] a detailed insight into the theory of simulation and a guide with practical approaches and techniques for creating simulation models. The following paragraphs are intended to give a rough overview of the state-of-art of this subject as needed for understanding the rest of this work.

The term *simulation* generally refers to imitating the behavior of a system or process to the purpose of analyzing such systems that are too complex for analytical or formal treatment.

Especially when dynamic systems are involved, simulation becomes a very helpful tool on the way to system analysis.

Specifically, simulation means performing experiments on a model in order to gain insight over the real-world object; in the context of simulation the *system* is said to be *implemented* through the realization of one or more algorithmic models, that involve an abstraction of the original system to be analyzed focusing on its structure, function ad behavior.

The first step therefore consist in finding a suitable existing model or creating a new one, which takes as input a certain number of parameters to feed to the constitutive equations.

After that, running a simulation with concrete values is what we previously mentioned as the *simulation experiment*; the result will be subsequently interpreted and transferred back to the real system.

By varying the input parameters of the simulation with regard to the actual situation or alternatively a desired target one and observing how the outputs change accordingly, it is possible to formulate conclusions and behavioral laws about the real system.

Needless to say, nowadays simulations are almost exclusively computer based, even if theoretically it is also possible to run this process by hand, e.g. for educational purposes.

To carry out a scientific study of the system involved, *assumption* about the real behavior are required to be built into the model, and these assumptions usually take the form of mathematical or logical relationships.

If the model relationships are simple enough, it may be the case that a mathematical or theoretical model is able to describe them completely with an analytical solution; most systems in the real world however are way too complex for analytical solutions and require numerical methods.

In this case great importance must be posed onto correct modeling, because faulty or inadequate model relationships can quickly lead to out-of-control errors.

As seen before, a system is a set of objects, not necessarily of the same type, that interact with each other in a logical context.
The models are written in such a way to provide answers to the particular questions that are being investigated; the *status* of a system is defined as the set of models parameters that are necessary to determine the answers to the specific research goal at any given time

1.2.1) Type of simulations

Systems can be generically divided into two groups: *discrete* and *continuous*.
In a discrete system the parameters change only at specific time intervals, whereas continuous systems have parameters that evolve continuously during time; actually, only a few systems are simply fully discrete or fully continuous.
In a continuous simulation the state parameters of the models change their value smoothly as the time goes by, and relationships for such models are generally in the form of simple differential equations for which an analytical solution is usually not possible.
We have therefore to recur to numerical methods and make sure that a consistent initial value is provided in order to solve these equations.
Conversely, in a simulation with discrete time advancement the parameters are coordinated to change their value at specific time steps, which must be defined within the models themselves.
At these points can also other kind of event be called into action, which can change again the system status or bring about some specific action; this way each step changes the current status of the simulation or at least influences its future behavior.
Many models need to combine continuous and discrete relationships, with help of a so-called *combined simulation*; such a case is increasingly common and three different types of interaction can be defined between continuous and discrete parameters:

1) A discrete time step can cause a discrete change of an otherwise continuous parameter

2) A discrete time step can cause a continuous state parameter to change at a specific time point.

3) When continuous parameters reaches its physical limit can trigger a discrete event e.g. being reset for a future time point.
   This is frequently the case in *StarSim* for parameters than need to be contained within determined boundaries (e.g. the charge level of a battery cannot drop below zero and cannot exceed the maximum capacity value); when such parameters hit their boundary, a exception-handling routine is called to take appropriate action and make sure the simulation does not loose physical meaning.

The dynamic nature of a discrete simulation requires the step-by-step advancement of time during its course; to this purpose a model is required, that increments the simulation time within itself and makes it available to the other models.

Complex models have been developed to find the optimal time-advancement step, but since StarSim need to synchronize many different models each with its own numerical solver hard-coded, a simpler fixed-step advancing mechanism has been preferred.

Generally speaking there is no relation between the simulated time within a model and the (actual) time that the computer requires to run the simulation; for a basic simulation is therefore unimportant how fast the simulated time is advancing.

Only when the model need to interact with an eternal object with real-time requirements, such as an on-board processor or even only a piece of real-time software, then simulated time must be correspond to real time.

Such software-in-the-loop or hardware-in-the-loop simulations require a strict synchronization mechanism among all the object taking part in the simulation and represent a major technical challenge.

1.2.2) Model based testing and validation

Since the beginning of the golden era of MBSE in the early ‘00s, the traditional space system industry has seen a wide assortment of elaborate and thus costly models being developed so far; in fact any discipline such as attitude control, thermal or structural design, requires its own specialized models to explore functional issues in the design and experimentation of the actual space conditions.

The final verification of the whole design system will eventually take place on a full-scale mockup satellite model.

Renowed satellite manufacturers have been using these technological frameworks for several years, already from the early stage mission development and design in order to ensure the greatest possible success probability.

It is indeed at the phase of testing and validation that MBSE unleashes its full potential: the particular use of MBSE to this specific purpose is known as the **Model Based Development and Verification** technique, and consistently with the already seen general advantages offers a systematic and standardized development and verification framework in the frame of a comprehensive, multi-system satellite simulator.

This allows the user to see the satellite components modeled and verified on a functional level (with no need for the physical object), so that complex and cost-intensive development of individual subsystems and key-technologies are drastically reduced.

Especially for low-budget & small satellites this technological environment brings about a significant breakthrough while simultaneously increasing the efficiency of the overall system design.

The test and simulation principles used in this environment can be divided in two main groups: firstly, the purely functional simulation, in which each component is represented by one or more scripts in the target programming languages.

Secondly, the more detailed simulation, taking into account the real behavior of the project-specific hardware and software i.e. including them in the simulation cycle.

For the present work, these two groups share the fact that they are only supposed to implement and simulate individual systems, while a complete satellite simulation remains beyond scope as, besides probably exceeding computational resources, will come with an unacceptable degree of uncertainty.

In simpler simulations like the one here described it is indeed possible to integrate satellite hardware and software into the simulation cycle, but only if the on-board processor is used with the real on-board software the results will be consistent and the test setup will provide advance verification capabilities.

This testing potential is not fully exhausted in small satellite projects since the simulation setup is limited to individual subsystems; the system-wide point of view of a spacecraft, as far as simulation and testing are concerned, has anyway been proven to increase the reliability of the system as a whole.

This is usually the case of small satellite projects, mainly due to financial reasons and time constrains that limit the verification of requisites that had previously been detailed by looking at the spacecraft as an overall system.

The scope of this work consisted in the development and verification of an innovative spacecraft system simulator which has been called *StarSim v.02* to enhance both the legacy and the differences with respect to the previous version (v.01), from which it differs both in programming languages, capabilities and design pattern.

The use of freely available open source software eliminates the needs for costly licenses; actually since of the major aims of model based verification and validation is cost reduction, especially for small sized projects, the application exclusively of open source software is a prominent strength of this work.

The concrete results in the subject of system simulation for the *CubeSat* project can be divided into two areas: firstly, *StarSim* supports and enables the verification of potentially the entire satellite under real-time condition.

This is relevant mainly for the development of on-board control algorithms, attitude determination and control algorithms and testing the transition between operating modes.

The second great advantage comes from the software and hardware verification tool itself: starting with a purely algorithmically simulation, it is possible to integrate progressively all the software (first) and the hardware (later) relative to one or more desired subsystems, verifying correct behavior at each step.

1.2.3) Model Based development and verification

In space flight engineering there are many systems to be tested fully before the take-off of the rocket vector; after take-off is a correction usually not possible anymore and so every possible fault must be unquestionably identified in advanced and taken appropriate care of.

Traditionally all the system models of the satellite were eventually implemented in hardware to run tests about their functional, electrical or thermal behavior; these models where realized taking care to correspond in the greatest possible measure to the condition in which they would be operational in outer space.

This, before the advent of MBDV, has been for a long time the only way to test the spacecraft components; this implies that to run parallel simulation on multiple systems different models had to be built.

Assembly, verification and validation play a great role in the budget allocation for a spacecraft development project, and moreover the overall feasibility of the project is limited by the available testing technologies.

In the latest space exploration projects, this method has become financially almost unbearable, besides having just as likely hit its technological limits.

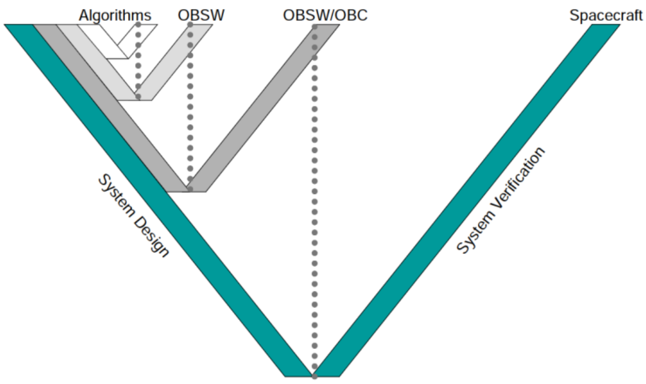
In more contemporary times it has become common practice for spacecraft producers to integrate model based computer-run simulations in their verification processes; this has approach has been preferred for the great cost-reduction that comes with, while being made possible by the continuously increasing processor computational capabilities.

The definitive functionality of on board system is still tested on ready-to-fly models, but this new simulation approach can almost completely replace physical models in all the previous phases of the design process.

In the earliest project phases, each subsystem is separately developed and simulated; as the project moves forward, functional integration of all the different systems brings two major concerns: the physical and communicative interaction between such subsystems.

Both aspects cannot be tested effectively in the independent simulation that were run in the previous phases of the process, but now is possible to research these issues with the help of simple, cost-effective simulated models.
This provides manufacturers with a second, even greater, advantage in terms of time and money: they don't have to wait until the real hardware is ready to test the overall functionality of the system; software models can be written starting from the datasheet of the producers and according to their own interest.

Model based validation and verification provides a substantial advantage when it comes to test the functionality of a system: its simulator infrastructure allows the creation of models independently from the project phase, and these models will be progressively integrated with on-board software and hardware when they will become available.
Furthermore, elements of previous simulation can be re-used and adapted to present needs, greatly reducing both project costs and risks.
The whole design and verification process can be reassumed in the well-known V-shape, here shown in the version of J. Eickhoff from his milestone work *Simulating Spacecraft Systems*



Picture 1-4: standard V-model for system engineering

The whole V-model, high lightened in blue in the picture above, actually breaks down in a series of interlinked V-steps: firstly the integrated control algorithms must be developed and tested, then they are implemented in actual On-Board Software, again to be developed, optimized and tested.

Finally, specific hardware (OBHW / OBC) has to be design and verified in order to accommodate the software.
As it can be seen in the picture, completing one of these verification steps marks the beginning of the verification of the next one.

Practically speaking, the process in divided into four phases:

1. The first step concerns the control algorithms: they are not yet implemented in the target programming language, neither on targeted hardware, but simply run from within the simulator to check their accuracy. This type of test is called **Algorithm in the Loop** (AIL).

2. Secondly, the algorithms are coded in software in the target language. The now available control software is completed with its environment and software communication lines to be fully operational. This type of tests is called **Software in the Loop** (SIL).

3. The third step is to load the control software onto a representative target computer. The final software on the target computer now has simulated system physics. This principle is called **Controller in the Loop** (CIL).

4. The fourth and final step of system testing now aims to make the control software on the target hardware now control the real system, and no longer the test stand's system simulation. This deployment phase is called **Hardware in the Loop** (HIL)

A strong tendency to shift entirely toward MBDV "*in-the-loop-simulations*" is settling in among professionals due to the following advantages:

1. It allows dynamic, detailed modeling with the purpose of investigating the overall functionality of the component.

2. Simulations can be run even in the early project phases, before on-board software or hardware is available; sometimes even before the system is completely defined.
3. The on-board software can be tested independently from its hardware
4. Functional procedure encoded in the on-board software can be fully tested and understood in a software-in-the-loop simulation; timing synchronization between hardware and software will take place in the next step, hardware-in-the-loop.
5. The simulator can be used as training tool by prospective systems engineers, and to test new potential software even outside the framework of an on-going space exploration project.
6. Development of numerical models is quicker and cheaper than the assembly of hardware testing mockups
7. Software models can be much more easily adapted to changes and undergo corrections
8. Simulators based only on software models can be quickly installed to speed up the process, eliminated issues regarding hardware availability and compatibility.
9. It eliminates completely the needs for shipping and logistics.

1.3) Simulator features

Keeping in mind the aforementioned advantages, the following list of requirements has been written for StarSim v.02:

1. **Multi-language support:** the simulator will be able to work with models written in Python in the AIL simulation section; this is a very popular, object-oriented programming language very easy to learn, to keep the creation of new models as simple as possible. When SIL or HIL simulations are involved, it will work with C models, the actual on-board target programming language.

2. **Real-time capability:** it will be able to perform simulations in a real time environment by appropriately synchronizing internal and external time; this is particularly important for SIL and HIL simulations.

3. **No previous experience needed:** user input will be limited to the selection of the intended models to be simulated and the input of related parameters; no previous programming experience is required.

4. **Automatic code generation:** the actual code performing the simulation will be self-generated according to the chosen model list and the chosen parameters; this allows for maximum flexibility and efficiency with minimal user input.

5. **Optimized, cross-compiling:** in the context of SIL simulation, it will be able to compile the target simulating program according to a specifying processor unit, optimizing it for speed and accuracy.

6. **Same functionality among all test phases:** passing to one simulation phase to the next will be easy and straightforward, as the simulator will behave exactly in the same manner and no ambiguity will be allowed.

7. **Simple, self-explanatory GUI:** the Graphic User Interface will be elegantly designed but at the same time sober and minimalistic, concentrating focus on the main functionalities and intuitively guiding the user step by step.

Chapter 2

2 Using the simulator

2.1) Usage of the simulator

At the present date, StarSim v.02 consists of about 114 files, for a total of over 2,600 lines of code.

The use of code snippets has been kept to a bare minimum, in order not to create too much confusion in the reader; anyway, the full source code of StarSim v.02 is available in the *StarSim Project* directory of the CubeSat Team Dropbox folder.

No previous experience is required to run a simulation, except of course a sufficient understanding of MBSE; again, to enhance simplicity of use, graphic user interface design has been kept sober and minimalistic.

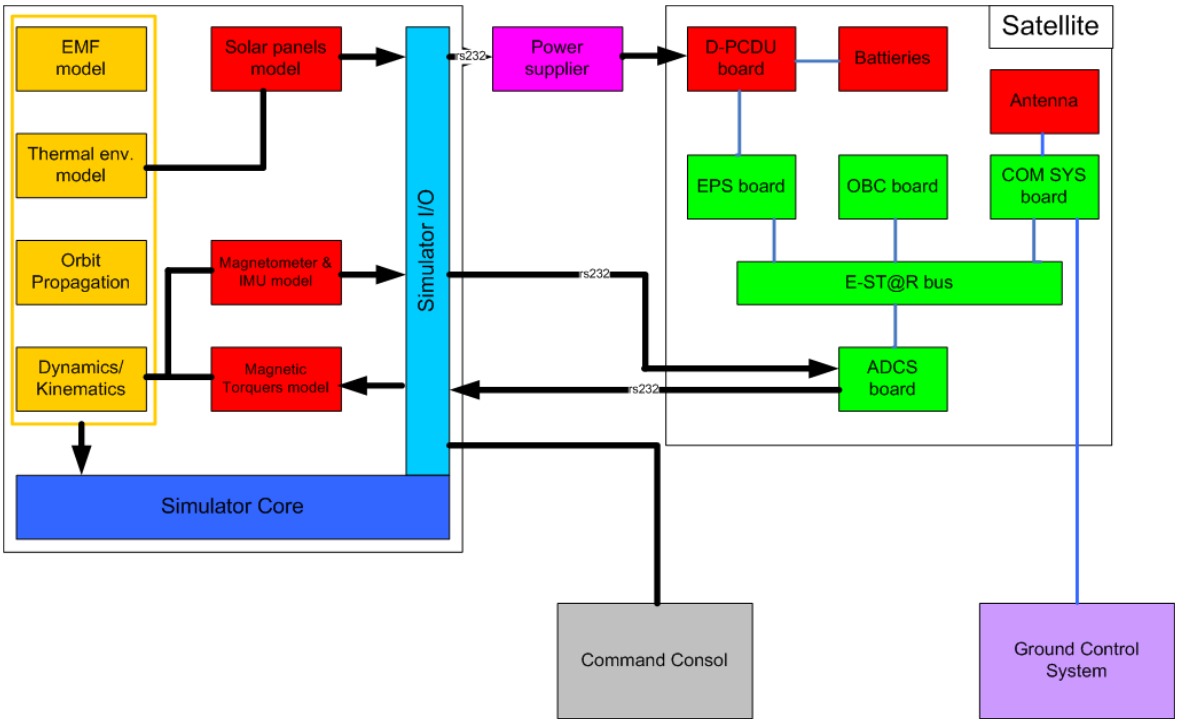
The working low of the simulator will be shown first; then the main algorithm behind it will be explained with help of graphic flow diagrams and possibly some code snippet; the division of the software into a certain number of programs will be then introduced.

Picture 2-1 belows shows a simple schematics of the whole simulator infrastructure.

The *Command Consol* block is usually hosted on a standard PC and hosts the Graphic User Interface as well as other software components to process user's inputs.

The *simulator block* (the one enclosed with a black line on the left) is also hosted on the same PC as the *Command Consol* in this case, but generally speaking it can be made to run on a separate workstation if high speed or real time precision is required.

The yellow blocks represent "well-known" models, whose output is known to be correct and are used to provide complementary information to the models to be tested, i.e. the red blocks, whose output is not known and needs to be validated.



Picture 2-1: simulator schematics

These “red blocks” can be simply algorithmical models in the same programming language as the rest of the simulator (AIL simulation) or can be software models in the target programming language (SIL simulation).

Wishing to perform a HIL simulation, the satellite block (on the right) can be added and integrated via standard RS232 interface.

Ground Control System is the block responsible for reading the hardware results of the HIL simulation, receiving them on in form of radio-transmitted packages as it would do with a real orbiting satellite.

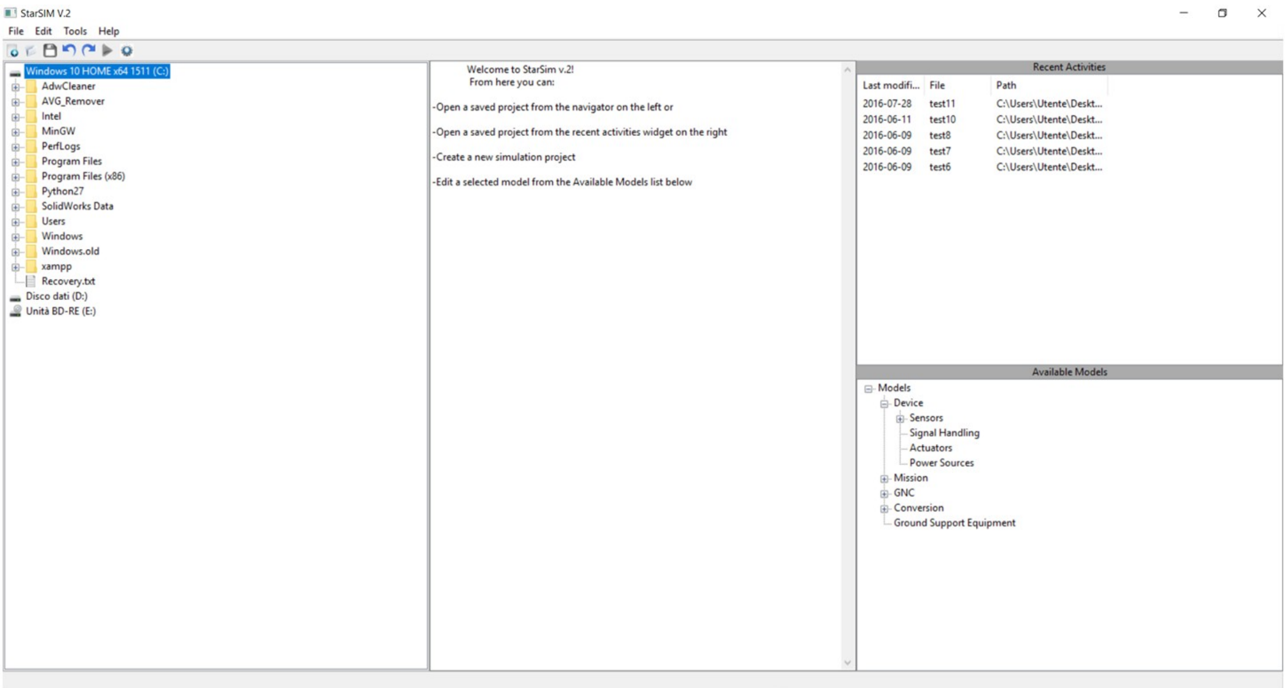
2.2) Main window

The main window of StarSim v.02 is shown in picture below: it consists of three main vertical sections, namely the *directory navigator* on the left side of the page, the *text* *container* in the middle and the auxiliary section on the right composed of the *Recent Activities* tab and the *Available Models* tree.

To load one a previous simulation setup from the *Recent Activities* tab, the user can simply double click on it: related simulation results will be shown in the main text container and all other simulation parameters will be loaded.

To load another saved simulation not present in the *Recent Activities* tab, navigate to it in the left-side directory navigator and double click on the related .sts file, or click *File > Open* project and select it in the pop-up window.

To obtain information about a single AIL model, double click on it on the *Available Models* tab: it will show name, category, and full python code in the main text container.



Picture 2-2: StarSim main window

2.3) AIL simulation

To start a new AIL simulation, click *File > New AIL simulation project* or the equivalent icon on the icon bar.

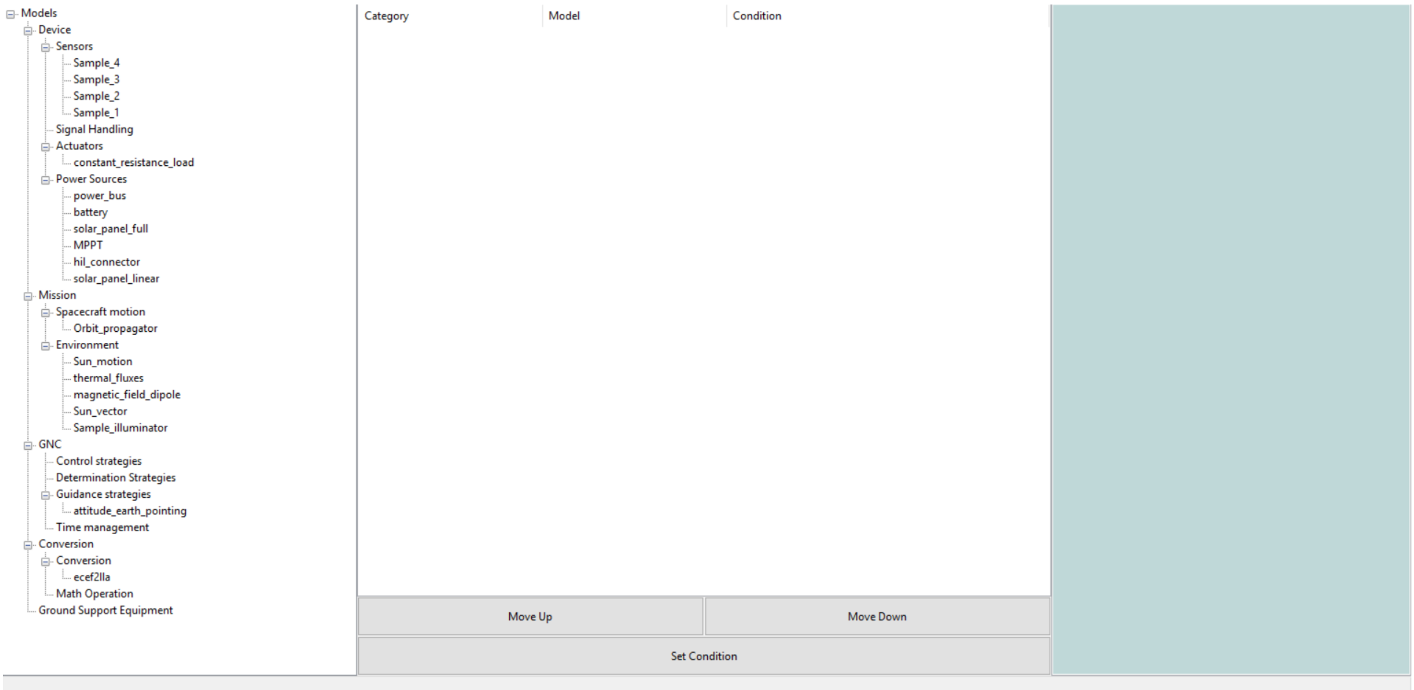
This will show the *Project window*, again divided in three vertical sections:

The *simulation model tree* on the left, already divided by categories, the main *model list container* in the middle and a right section temporarily unused; it can be used, if the user wishes so, to display an image associated to the current model (e.g. a workflow diagram).

The whole simulation setup process is organized in a very linear and straightforward manner: the user starts by selecting the algorithmic models he wants to include by double clicking on them on the left section.

The selected model will appear in the central section, the model list, under the column "*model*".

To adjust the execution order, *Move Up* and *Move Down* buttons are provided; to remove a model from the list just double click on it.

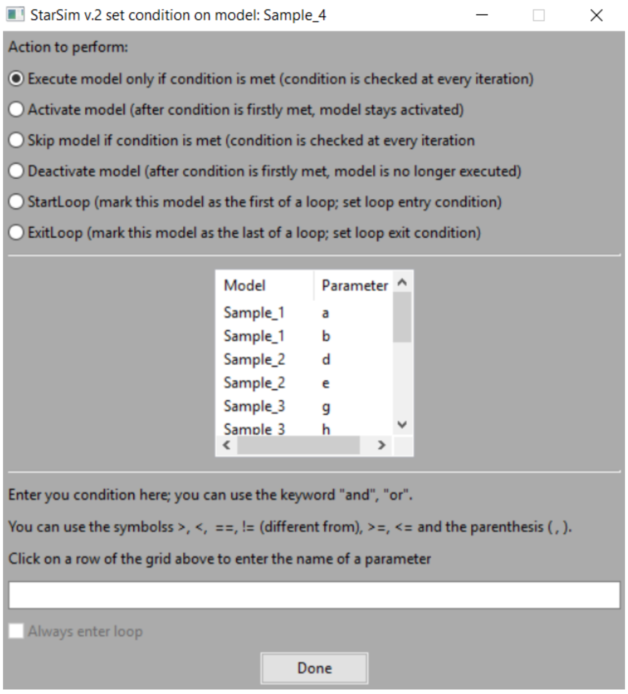


Picture 2-3: StarSim project window

2.3.1) Setting conditions

In order to enhance the flexibility of the simulation, the user has the possibility to set some condition on particular models to determine in which case they should be executed (normally a model is executed at every iteration): after selecting the model from the main model list, click the

Set Condition button to open the related window.



Picture 2-4: setting condition ona certain model

The list of possible conditions to impose ("*Action to perform*") is pretty much self-explanatory: it allows for condition-based loops, permanent activation or deactivation of a model after a specific condition has been met or conditional execution of a model (where the condition is checked at every iteration).

In the bottom text field the user can enter his condition using standard symbols *<, >, ==, =!* (different from), *>=, <=;* to enter the name of a parameter to perform the condition on, he is required not to write it directly but select it from the table above, which has been automatically assembled by parsing all the free parameters from all the already selected models.

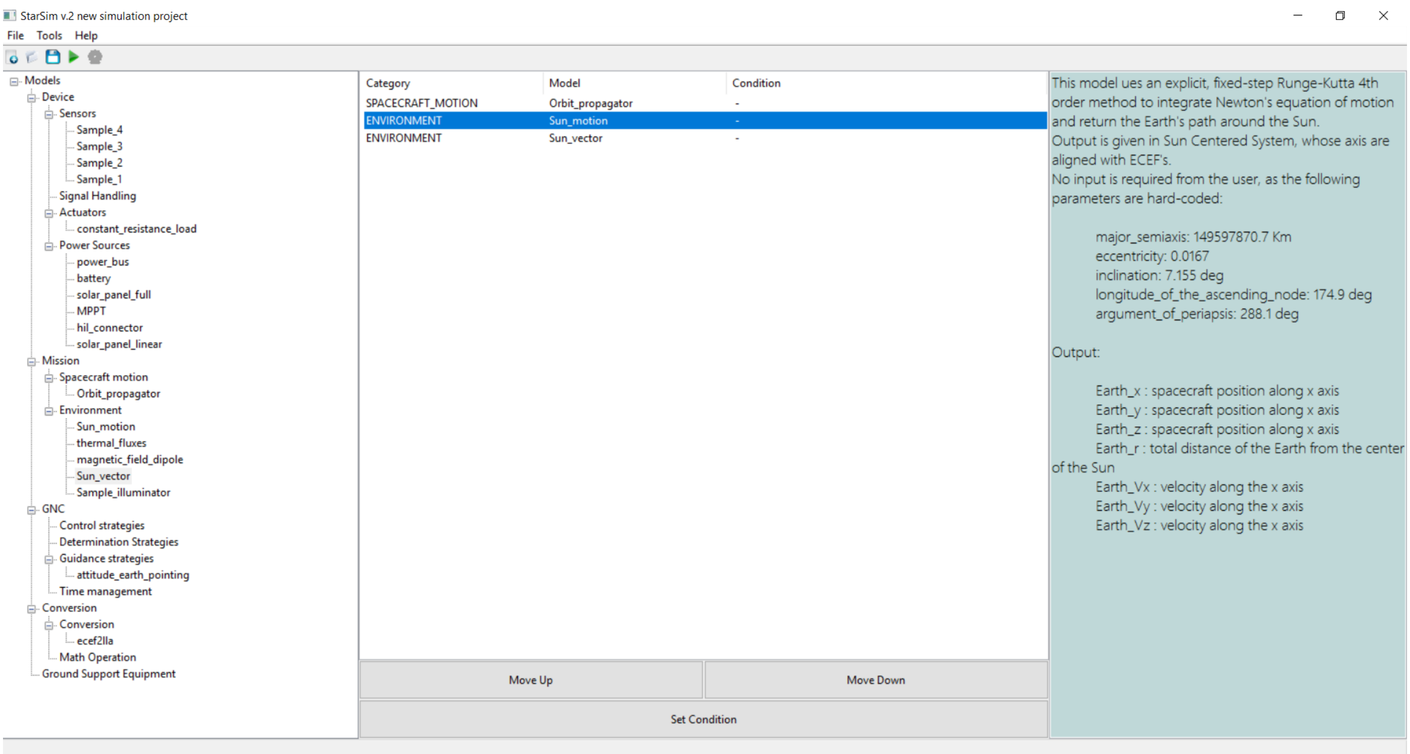
That is because when the simulation file will be created, all free parameters and variable will be initialized in a single common class, called "*var*".

This allows for simplicity and flexibility in source code, as every AIL model can simply call class *var* to access any possible variable of the simulation, even if pertinent to other model (otherwise we will need a complex setup of "*import*" statements, making for a heavier and less readable code).
Disadvantage to this strategy is that, according to Python rules, to call a variable for class *var* from outside the class, it needs to have "*var.*" prefixed to it. (e.g. variable *time* becomes *var.time*).

So by clicking on an entry named *time* in the variables list, it will actually insert *var.time* in the text field (if the user is just a bit more expert, he can directly write *var.time* himself).

2.3.2) Simulation setup

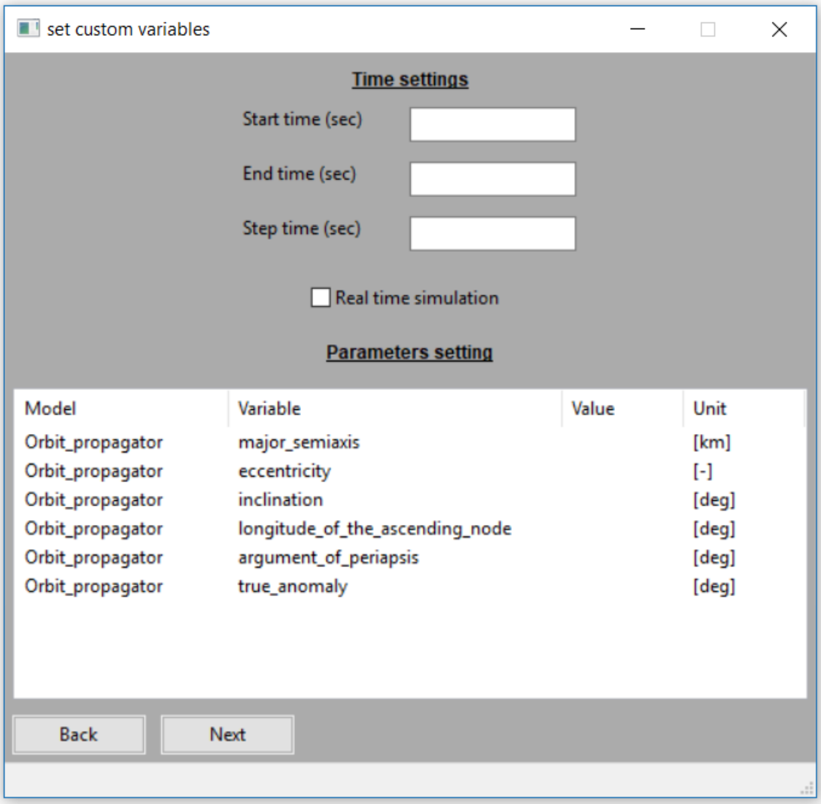
As the user proceeds in his choice of models, the auxiliary column on the right side of the screen provides information about the model input, output and general behavior:



Picture 2-5: sample orbit simulator setup

In the picture above, a project window is show where the user has selected models relative to the spacecraft's motion; the *Sun vector* model (i.e the currently selected model) is detailed on the right column.

After the model flow has been determined and the necessary conditions imposed, the user can start the simulation setup process: click on the *Run* icon (the green arrow) in the icon bar to call the *Set Variables* window: it will ask the user to provide the start time of the simulation (usually *0*), the final time and the time step.



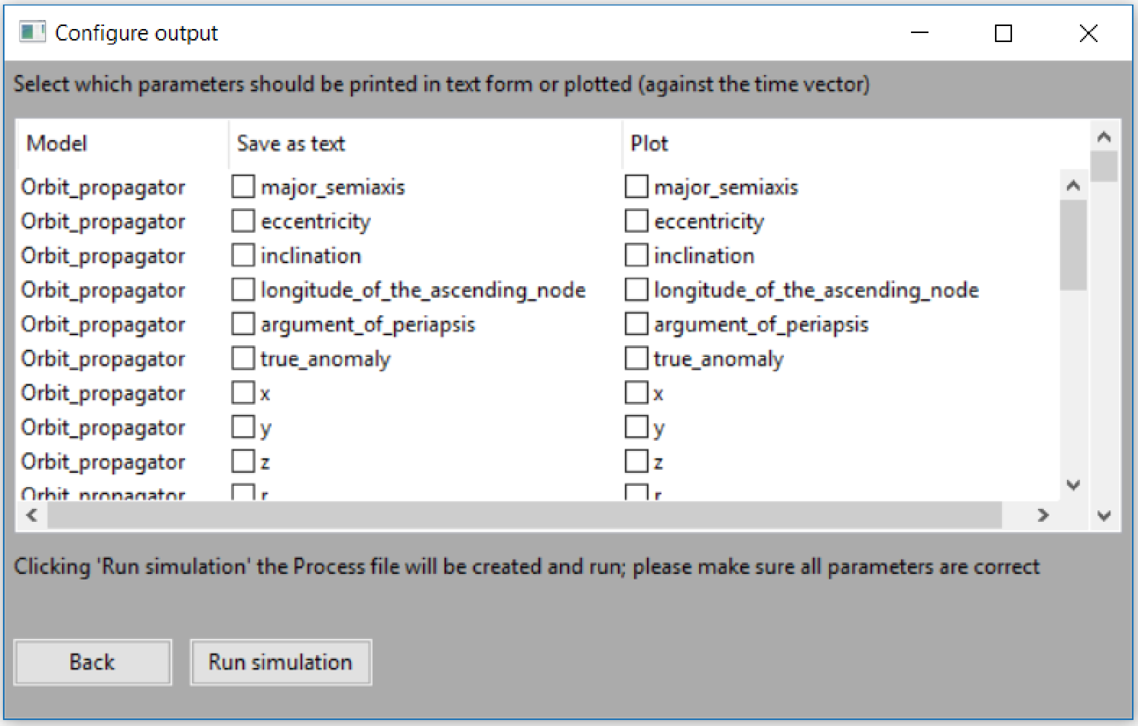
Picture 2-6: Set parameters window

Below it, all the free parameter from all selected model are presented (automatically parsed) and the user is asked to provide a value for them.

By click next, the *Configure output* window is called: in this window, the user decides which variables he wants to see as output in the main text container of the main window at the end of the simulation, and which parameter he wants to plot.

Each of the selected variables will be shown as column vector correlated to the time vector (obviously, one value of the variable for each step of the time vector).

By clicking next, the simulation file is automatically created and run; the main window is called back on top of everything and results of the selected variables are displayed in its main text container.



Picture 2-7: select desired output

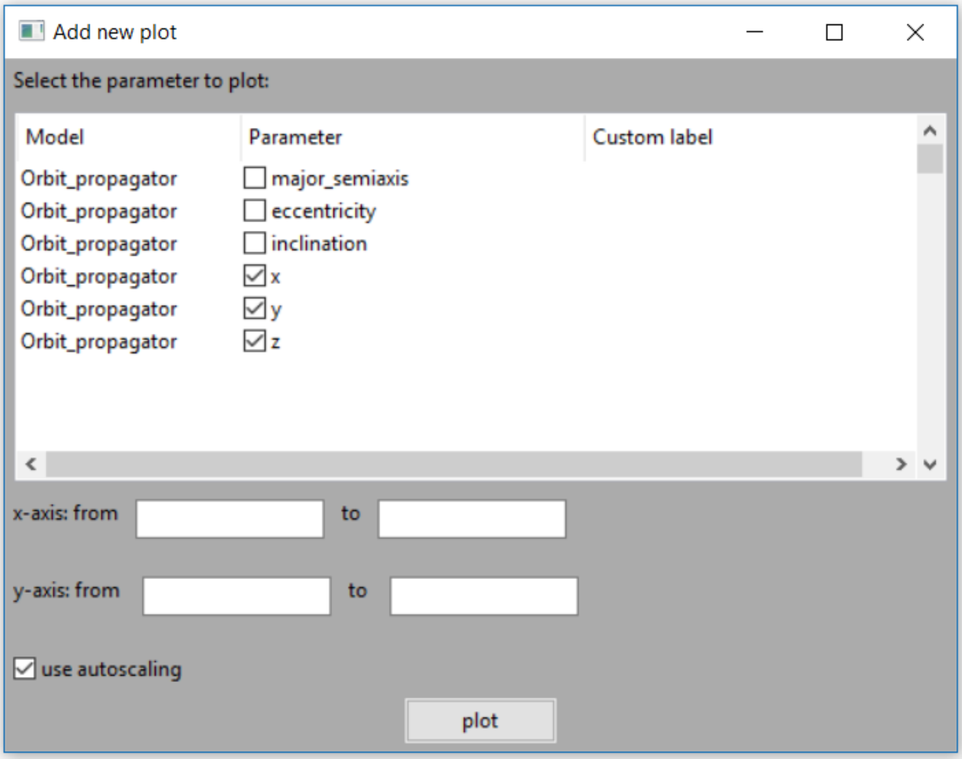
2.3.3) Plotting

Plotting happens automatically if the user selected the desired parameter to be plotted in the *configure output* window; anyway, he can always decide to add some more information to the plot or plotting new results, so a feature is provided to add a new plot one the simulation has ended by clicking *Tools* > *Set Plot*.

The *Plot window* is again very self-explanatory and allows the user to select different start / end values, color and line style for every parameter he wishes to plot; furthermore, it allows the user to select custom scales for both axes, choose the autoscale option (absolute maximum and minimum values of plotted variables are used as extreme axis values) or logarithmic scale.

The plot is opened in a separate window that allows for custom view sizing, subplot configuration and image saving.

To save these results, select *File* > *Save Project*; they will be saved as .sts file and anyway added to the *Recent Activities* tab.



Picture 2-8: plotting window

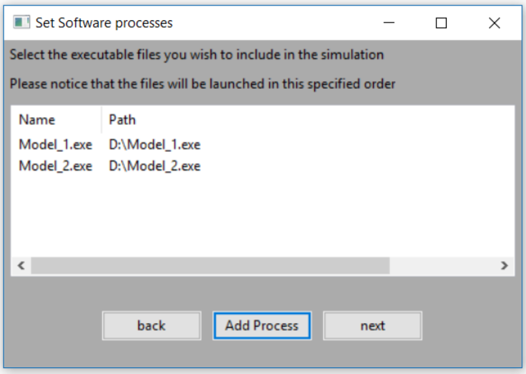
2.4) SIL Simulation

The SIL (Software-in-the-loop) simulation is the direct evolution of the AIL simulation: in this case, the models to iterate are separate executable files, completely autonomous from StartSim itself, written in the real programming language that is supposed to be used in the mission (usually C).

Practically, the simulation aims at reproducing the behavior of the real source code that will be actually used, complete with its environment and communication lines to other section of the software.

To start a SIL-simulation, click *File > New SIL simulation project*; this will immediately call the *Set Software processes* window, in which the user has to select the external files that will constitute the simulation loop.

All entries must be ready-to-run, compiled, executable files; in this instance, we have selected only two sample files named *Model_1.exe* and *Model_2.exe.*



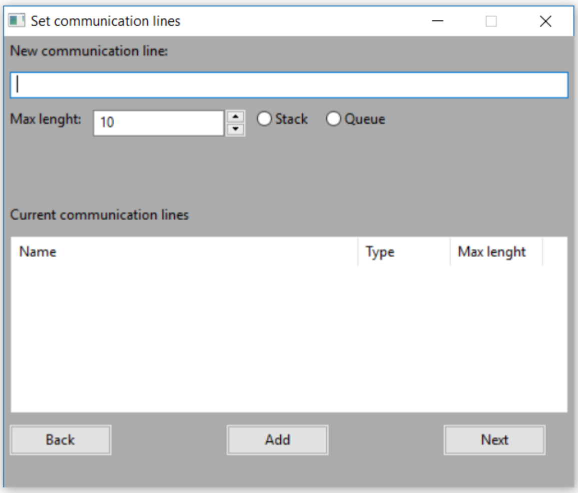
Picture 2-9: select software processes for SIL simulation

Generally these files must be selected in the order in which they are supposed to run, even though while defining the communication lines later on it is possible to allow exceptions.

2.4.1) SIL Communication setup

Once the models have been selected, the *Set communication lines* window pops-up by clicking on next.

This window allows the user to create as many communication lines (i.e. pipes) as he wants; option are provided to ensure a maximal length and decide between Stack and Queue model for each pipe (that means, FIFO or LIFO behavior).



Picture 2-10: open new pipe

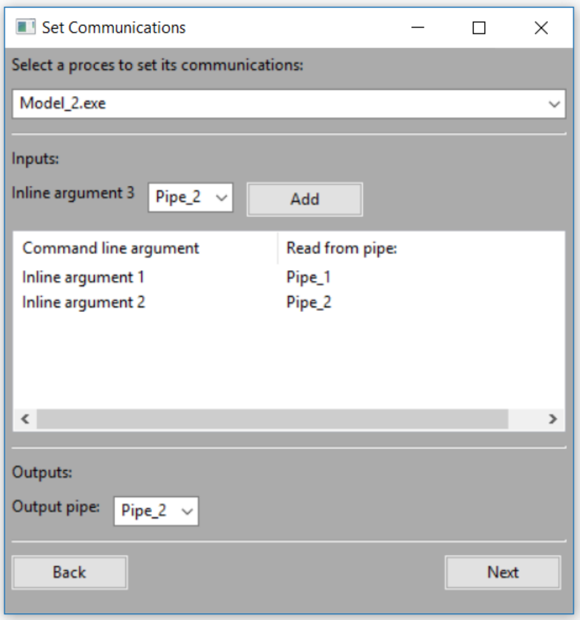
Once the communication lines have been created, the users has to decide how to connect them to the various models to setup his intended simulation architecture: this work is carried out in the *Set communication* window, which allows deciding how pipes are connected to each models.

In this software, the C-written executable files that constitute the core of the simulation work by command-line argument: each file, representing a particular function to be performed in the mission, accepts one or more parameters in form of command line arguments and provides one (and only one) output.

The pipes practically work as lists, memorizing the outputs of a particular file and feeding them as command-line argument to another one, in the same order; so for instance if a function requires three parameters, all of which are output of previously executed functions, the users has to create three communication lines, connect their origin to the output of those function and connect their end to the input of the current file in the right order, so that each expected command line argument can be provided accordingly.

The *Set communication* windows allows the user to select the pipe on which the current model will write its output (bottom section), and to decide which pipes must be taken as input: in this example we have instructed the model "*Model_2.exe*", that expects two command-line arguments, to take its first one from *Pipe_1* and its second one from *Pipe_2*, as well as to write its output on *Pipe_2.*

This means that *Model_2.exe* is a recursive function, that takes its own output as input.

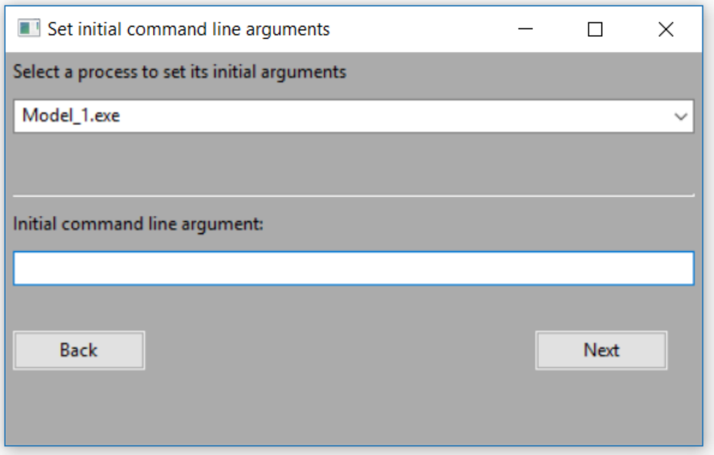


Picture 2-11: pipes setup

2.4.2) Initial values

Of course, this whole process cannot work properly at the first iteration, since the pipes are still empty and can't provide inputs.

This is why a final window pops-up, the *Set Initial command line argument* window, in which the user is asked to write, relatively to the first iteration, the full string of command-line arguments for each model as if it were called as stand-alone file on the command prompt.
This is the equivalent of setting the initial values and boundary condition of the simulation.



Picture 2-12: setting initial arguments

Differently from the AIL simulation, in order to test the efficiency of the software pipes, the SIL simulation is intrinsically programmed to run real-time; this means that on operating systems supporting this functionality, such as UNIX with custom real-time kernel, it will provide exact real time result.
On different operating systems, it will anyway stop the program for all the time that has to be simulated (pseudo real-time), but in this case time values are not to be taken with great accuracy due to OS interrupts and operation that the software cannot handle.

2.5) Source code analysis

The "structural core" of the simulator is the file called *config.py*, which works as a module containing all the different objects needed both functional (lists, dictionaries…) and structural (windows).
Here is shown a small snippet of the config file:

```
plot_dict = {}
out_dict = {}
path_to_open = ""
plot_selected_value = ""
frame = None
pw = None
svw = None
```

The first four lines refer to internal variables of the simulator, namely two dictionaries holding data about plotting and outputting and two strings, all initialized empty.

The last three lines refers as windows (*frame* is the main window, *pw* is the *project window* and *svw* is the *Set Variable window*) and are all initialized to *None* (non-existent).

When an object is created or modified, the change in status is performed only on the *config* file; since this package is imported by all the other StarSim files, the modification will be immediately shared among all components, allowing for great flexibility for the programmer and saving a lot of hassle that we would have otherwise coordinating all the modules.

For instance, when the *Set Variable* windows need to be called, the process goes like this:

```
if config.svw is None:
    # Creates and spawns Set Variable Window
    config.svw = svw_gui.SetVariablesWindow(config.pw, "set custom variables")
else:
    config.svw.Show()
    config.svw.Raise()
```

In the third line you can see the calling function assigning an instance of the class *SetVariableWindow,* which is contained in the module *svw_gui* (extended, it sounds like "Graphic user interface of the *set variable* window") to the *svw* object in the imported package *config*.

This class requires two arguments to create an instance and those are provided between brackets: the name of the parent window (*pw* stands for *Project windows*, and it also is an object belonging to *config*) and the name of the newly-created child window.

All the modules that make up StarSim are divided in just two folders: GUI and Sources.

The GUI file, which is solely responsible for the graphic user interface, needs to import its related source code in order to be actually working: the two files are distinguished also by a name convention, i.e. the main part if the shortened name of the window followed either by *_gui* or *_src*: *svw_gui* and *svw_src.*

The source file needs to import the GUI file of the **next** window to be called in order to be able to effectively generate it, otherwise trying to create an instance of that class will result in an error.

This snippet, taken from the very beginning of the *pw_src* file (i.e. the file that manages the internal working of the AIL *Project Window*) gives an idea of all the import statements that are necessary in order to successfully connect the various modules:

```
import wx
import config
from GUI import SetVariableWindow as svw_gui
from GUI import SetConditionWindow as scw_gui
```

Here are imported the *wx* library that allows the creation of GUIs, the *config* file to ensure synchronization with respect to all other modules, and the GUI file of the two windows that can be spawned from within the *Project window*, namely the *Set Variable* window and the *Set Condition* window.

Of course, any window has all the necessary functions (in object-oriented programming called *methods*) to read the input from the user, parse it, check it for completeness and accuracy, and save it in an appropriate data structure in the config file.

Once completed this jobs, it carries on by calling the next window with some lines very similar to those shown in the second snippet and this process iterates for how many windows are necessary.

When all the windows have been called and all user input has been read, it is the moment to perform the actual simulation; the process can be sum up in three steps:

1) A module named *SimSetup* calls its (only) method *create_process()*: this creates a new .py file and write on it the final simulation code; if there are no conditions imposed on the models, that code consists simply in a list of import statements (one for each model), a line initializing a new method named *Process()* and a list of the names of the models to be called, enclosed in a for-loop.

2) The *SimSetup* module closes the file he was writing, which now becomes available for further use; it consist of a class called *Variables*, which contains all the variables from all the selected models, and the actual *Process()* method, that works on an instance of the Variables class named *var*.
The idea behind this class is that of enabling easy access to any variable from any part of the code, independently of the particular model it is originated from; that means, given a variable called *random_var* we can simply access it via *var.random_var* and be sure that this works, otherwise we would have to retrieve the model it's firstly declared in, by parsing again through a list or dictionary, and access the variable from its particular parent model, e.g. *Sample_4.random_var*, making for a much more complex code.
The *Process.py* is the file that contains the actual (auto-generated) simulation code; it's pretty short and can be wholly shown here, in the event of a simple simulation with four sample models and no condition imposed:

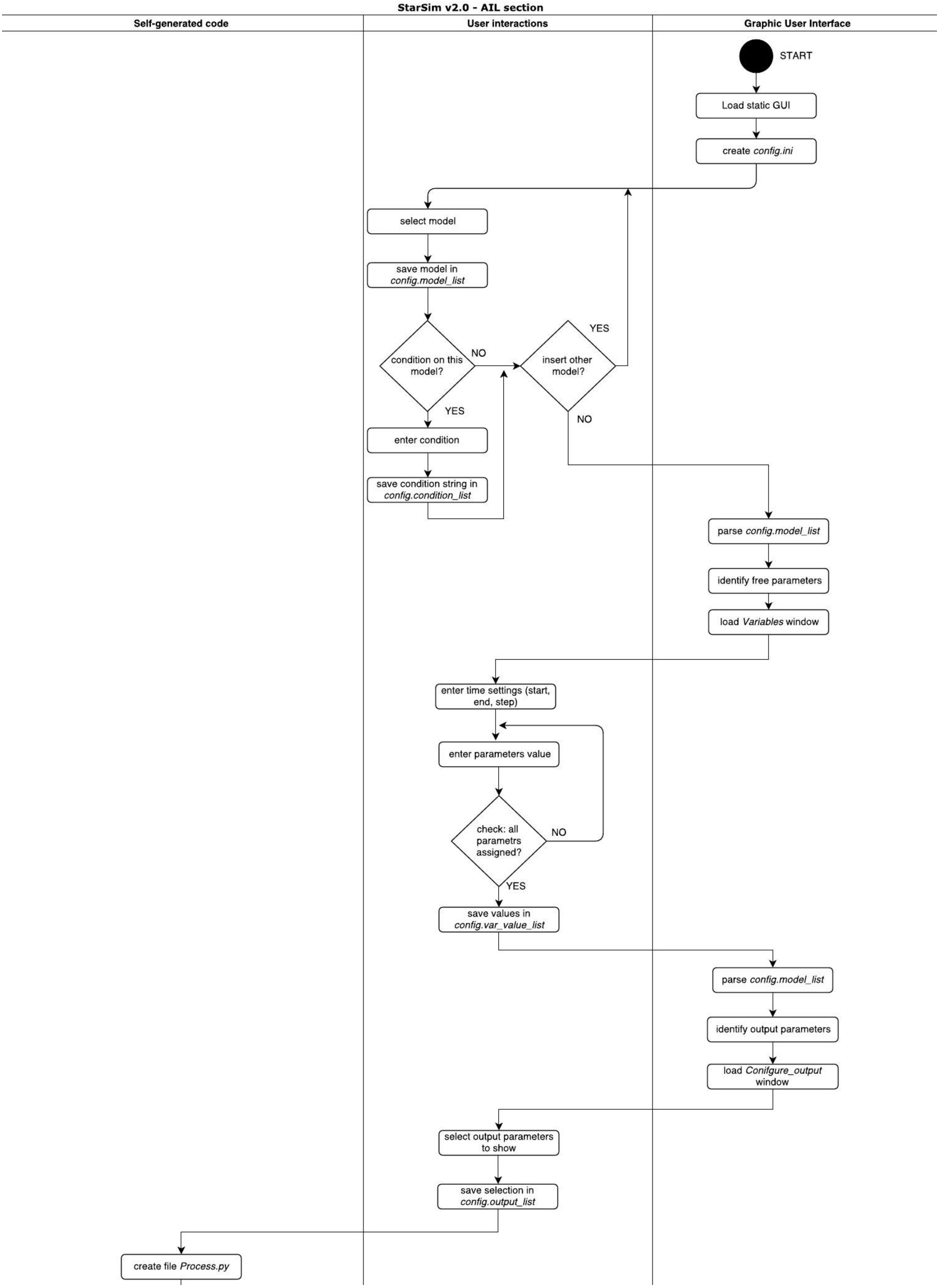
```
from Models import Sample_1_file
from Models import Sample_2_file
from Models import Sample_3_file
from Models import Sample_4_file
import config

class Variables():
  def __init__(self):
    self.t = 1
    self.a = 1
    self.b = 2
```

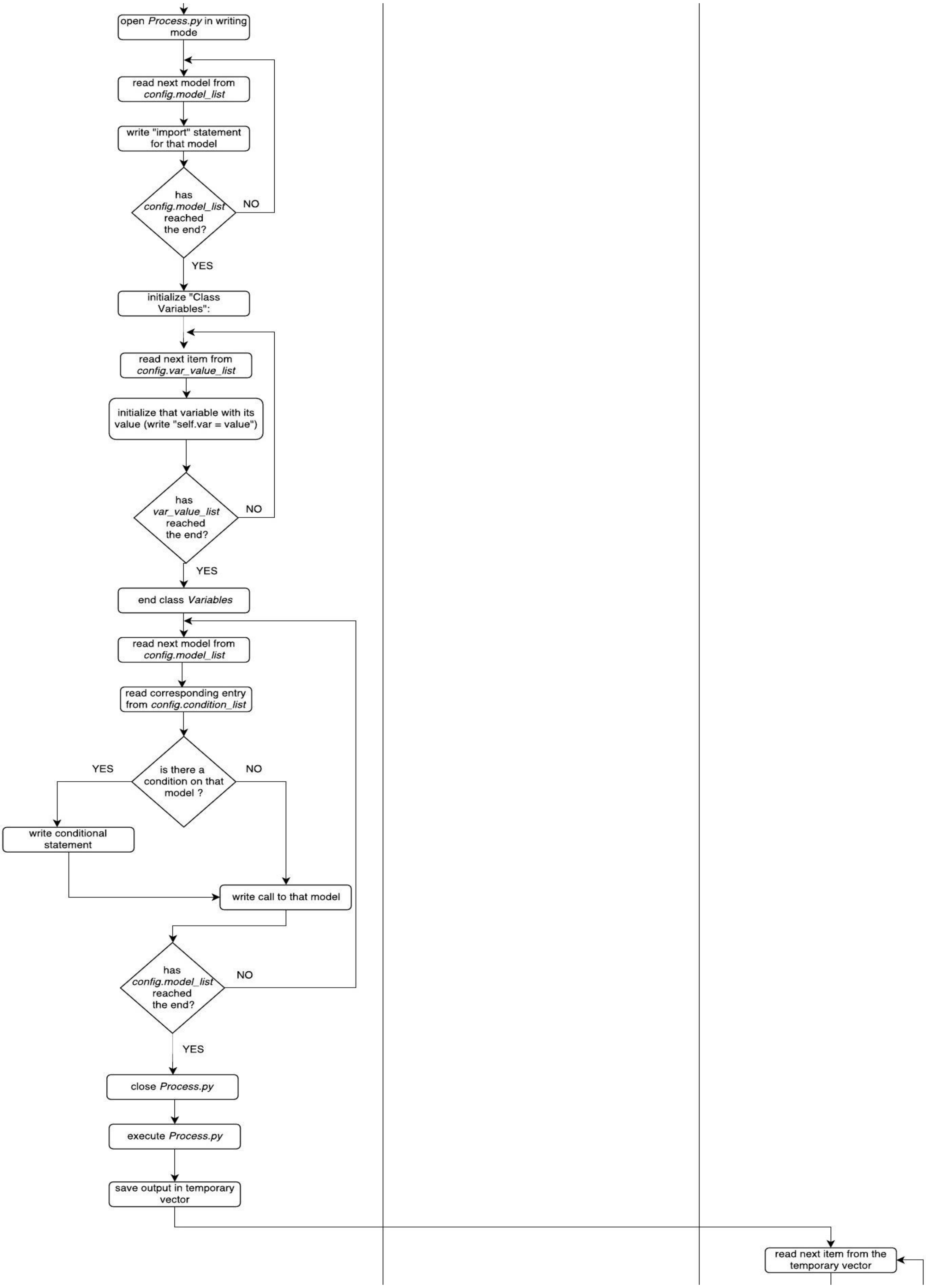
```
    self.d = 3
    self.e = 4
    self.g = 5
    self.h = 6
    self.l = 7
    self.m = 8
    self.n = 0
    self.c = 0
    self.f = 0
    self.i = 0

def Process():
  var = Variables()
  output_list = []
  for n in range(0, 13):
    output_list.append([])
  n = 1
  while var.t <= 12:
    Sample_1_file.Sample_1(var)
    Sample_2_file.Sample_2(var)
    Sample_3_file.Sample_3(var)
    Sample_4_file.Sample_4(var)
    output_list[0].append(var.t)
    for i in range(0, len(config.var_save_list)):
      output_list[n].append((vars(var))[config.var_save_list[i]])
    var.t +=1
    n +=1

  del var
  return output_list
```

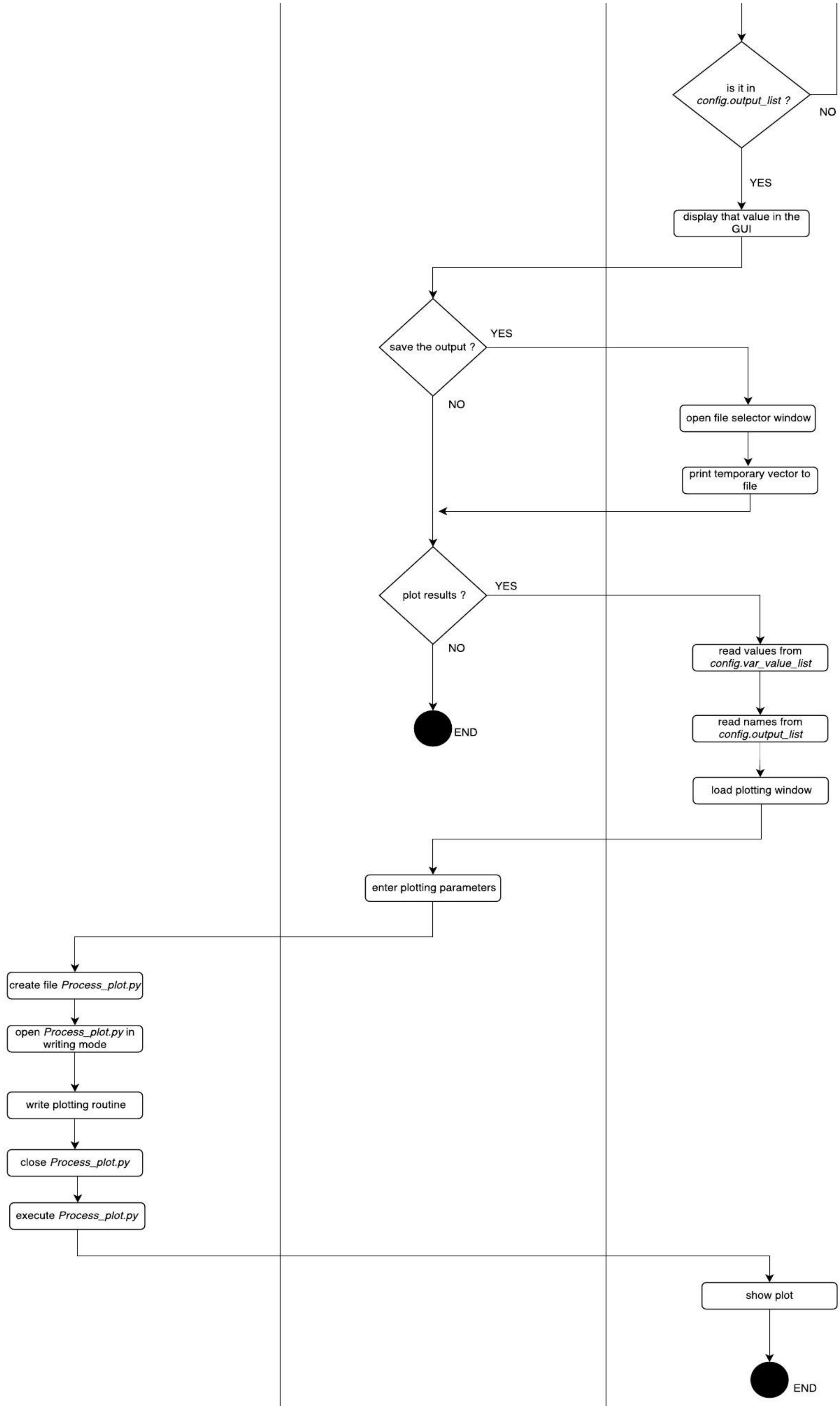
The *Process()* methods create an instance of the *Variables* class, creates a list of 12 empty elements (12 is the final time selected by the user in this case), and run 12 iterations of the four sample models, saving their output in pre-defined data structures.

3) A module called *execute.py* calls its two methods in a row: firstly *execute_process*, to actually run the above file, and then *display_output* to write the results in the text section of the main window.

The same whole process runs also for the SIL simulation, with minimal differences.
The whole action diagram for a StarSim AIL simulation is detailed in the following pages (given its length it has been split up in more figures):


StarSim v2.0 - AIL section
Self-generated code
User interactions
Graphic User Interface
START
Load static GUI
create config.ini
select model
save model in config.model_list
condition on this model?
NO
YES
insert other model?
YES
NO
enter condition
save condition string in config.condition_list
parse config.model_list
identify free parameters
load Variables window
enter time settings (start, end, step)
enter parameters value
check: all parametrs assigned?
NO
YES
save values in config.var_value_list
parse config.model_list
identify output parameters
load Conifgure_output window
select output parameters to show
save selection in config.output_list
create file Process.py


open Process.py in writing mode
read next model from config.model_list
write "import" statement for that model
has config.model_list reached the end?
NO
YES
initialize "Class Variables":
read next item from config.var_value_list
initialize that variable with its value (write "self.var = value")
has var_value_list reached the end?
NO
YES
end class Variables
read next model from config.model_list
read corresponding entry from config.condition_list
is there a condition on that model ?
YES
NO
write conditional statement
write call to that model
has config.model_list reached the end?
NO
YES
close Process.py
execute Process.py
save output in temporary vector
read next item from the temporary vector



Picture 2-13: StarSim complete flow diagram

2.5.1) Pre-Processing

The pre-processor fulfils the purpose of providing all information to the user in a human-friendly manner, allowing for him to choose models and parameters of the simulation, translating back his choices into machine code and feed the result to the (self-generated) actual process file.
In order to achieve this scope it consists of the following functional units:

- *GUI spawner*: spawns the graphic user interface and keeps listening for user inputs.

- *Model parser:* presents all the models available in the folder in form of a hierarchical tree to the user, so that he can choose.

- *Model lister:* after the user has made his choice of models, translates this in form of a Python list.

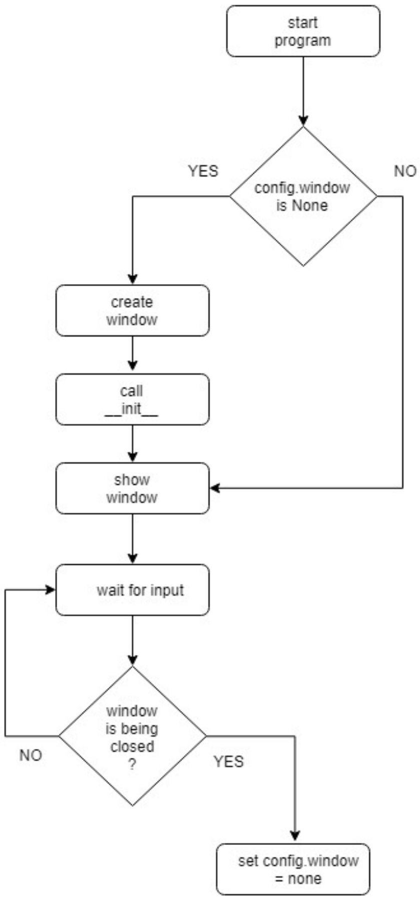
- *Parameter parser*: for each model in the list, its custom parameter are extracted and presented back to the user in the *Variables* window in order to receive an actual value.

Note that these are not programs by themselves; these are just functional units for the scope of clarification, each consisting of more programs according to Python's best practices.

2.5.1.1) GUI spawner

A certain number of programs are responsible for the correct management of the Graphic User Interface and its interactions; this is a very complex part of the design of StarSim and it's not fully reported in this work as it is not strictly related to aerospace engineering; it's listed anyway in appendix B.

The generic flow chart that sums up the behavior of the whole GUI is the following:



Picture 2-14: GUI action diagram

Note that the main window is an object (like anything else in python) defined in the *config* file, and thus available to all other files for synchronization purposes.

The __*init*__ call invokes python's initialization method and actually creates the window with its defined parameters

2.5.1.2) Model parser

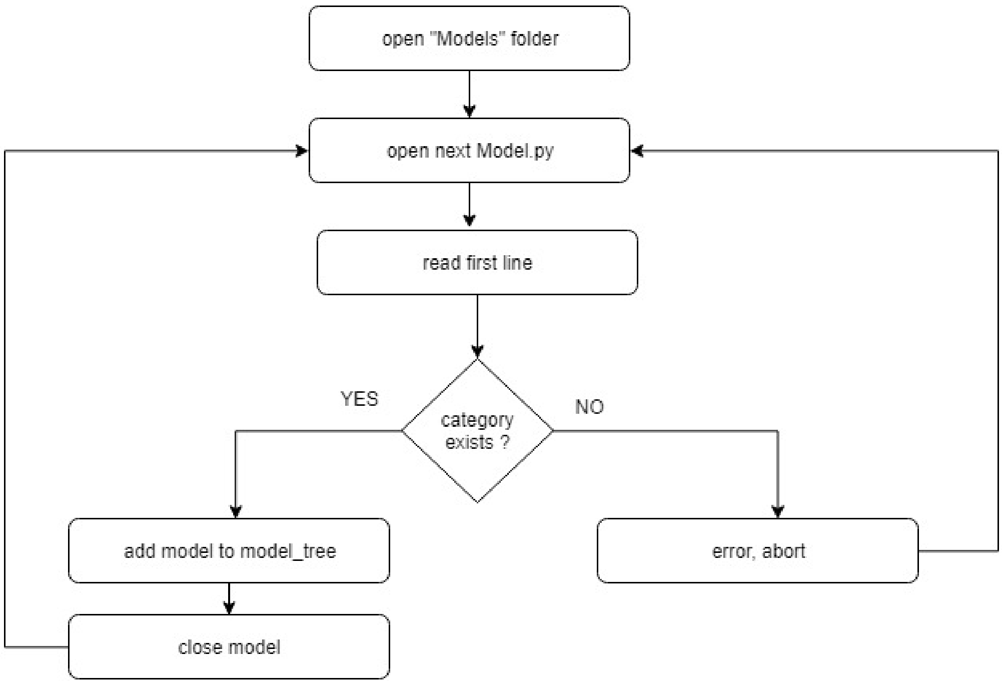
The model parser scans each model present in the *Models* folder and reads its first line; according to StarSim's models protocol, this first line indicates the category of the model, which can be one of the following:

- Sensors
- Signal Handling
- Actuators

- Power Sources
- Spacecraft motion
- Environment
- Control strategies
- Determination strategies
- Guidance strategies
- Time management
- Conversions
- Math operation
- Ground support equipment

The models contained within each category and their inner functioning are detailed in chapter 3, *The standard library.*

Once that its category has been determined, the model can be added to the model tree; here is the flow chart diagram for the model parser:



Picture 2-15: model parser action diagram

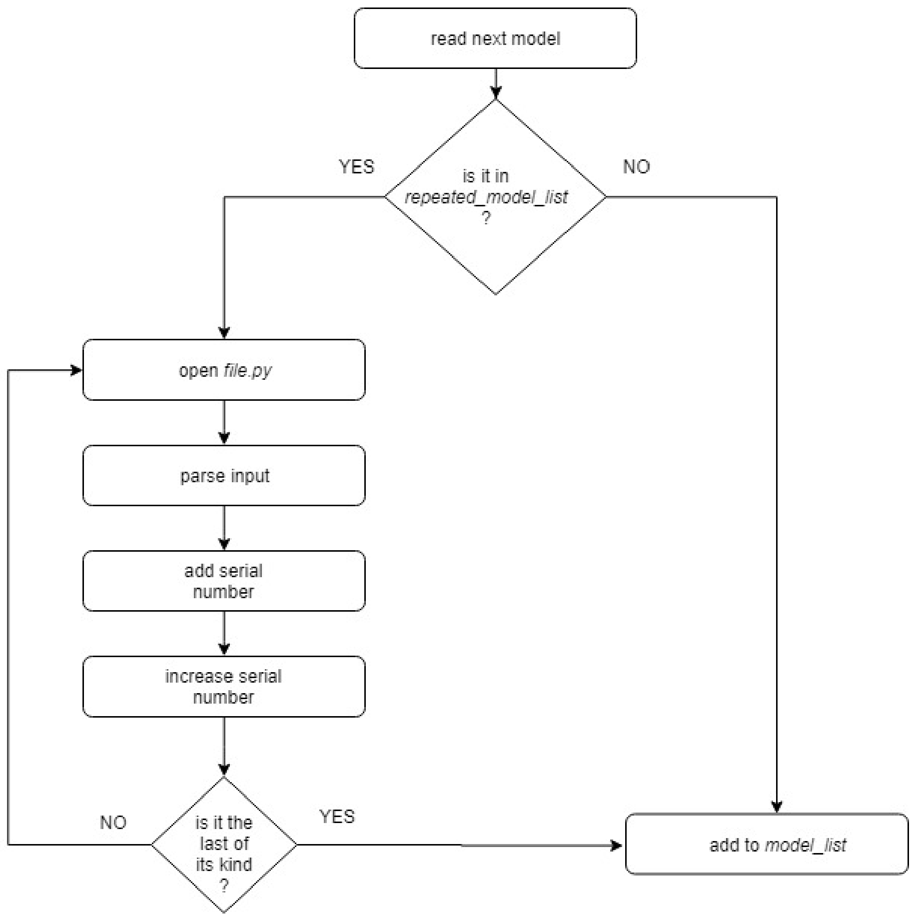
2.5.1.3) Model lister

Once the GUI has been created and the available model listed up in the model tree, the users selects the models he wants to use in its simulation; the model lister program comes here into play and makes sure that every selected model is added to the model_list array that will be fed to the actual simulation process.

In particular, if a model is selected more than once (e.g. a solar panel), it must automatically add a serial number to it to avoid confusion; to achieve this scope, a *repeated_model_dict* is created, a dictionary to which a model is immediately added at the moment of selection if it's already in the selected model section of the GUI.

This dictionary will associate to each selected model the number of it occurrences, and it will be useful later on to the parameter parser, that needs to add a serial number to each repeated model to avoid confusions.

Here is the model lister flow diagram:



Picture 2-16: model lister action diagram

2.5.1.4) Parameter parser

Once that the user has chosen its models ant those have been listed into the *model_list* structure by the *model_lister*, it's time to choose the actual values of the parameters: According to StarSim's models protocol, each model begins with a commented section that, being fully transparent to the Python code, brings important information such as the model category, its input and output parameters.
In particular, this section is organized according to the following scheme (the '#' symbol and the triple quote string both represent a commented line or section in Python):

```
# model_category
'''
CONFIG
Input_paramater_1 [unit]
Input_paramater_2 [unit]
Input_paramater_3 [unit]
Output
Output_parameter_1 [unit]
Output_parameter_3 [unit]
END_CONFIG
'''
```

Here is an example from the battery model:

```
# power_sources
"""
CONFIG
nominal_voltage [V]
capacity [mAh]
depth_of_discharge [%]
max_discharge_rate [C]
output
battery_I_out
actual_voltage
END_CONFIG
"""
```

The action flow for the parameter parser is thus the following:

It starts by reading one by one every single line in the beginning of the model code; as soon as it find the “CONFIG” word, it sets up flag telling the software to save every successive line as an independent input parameter into the var_dict structure.

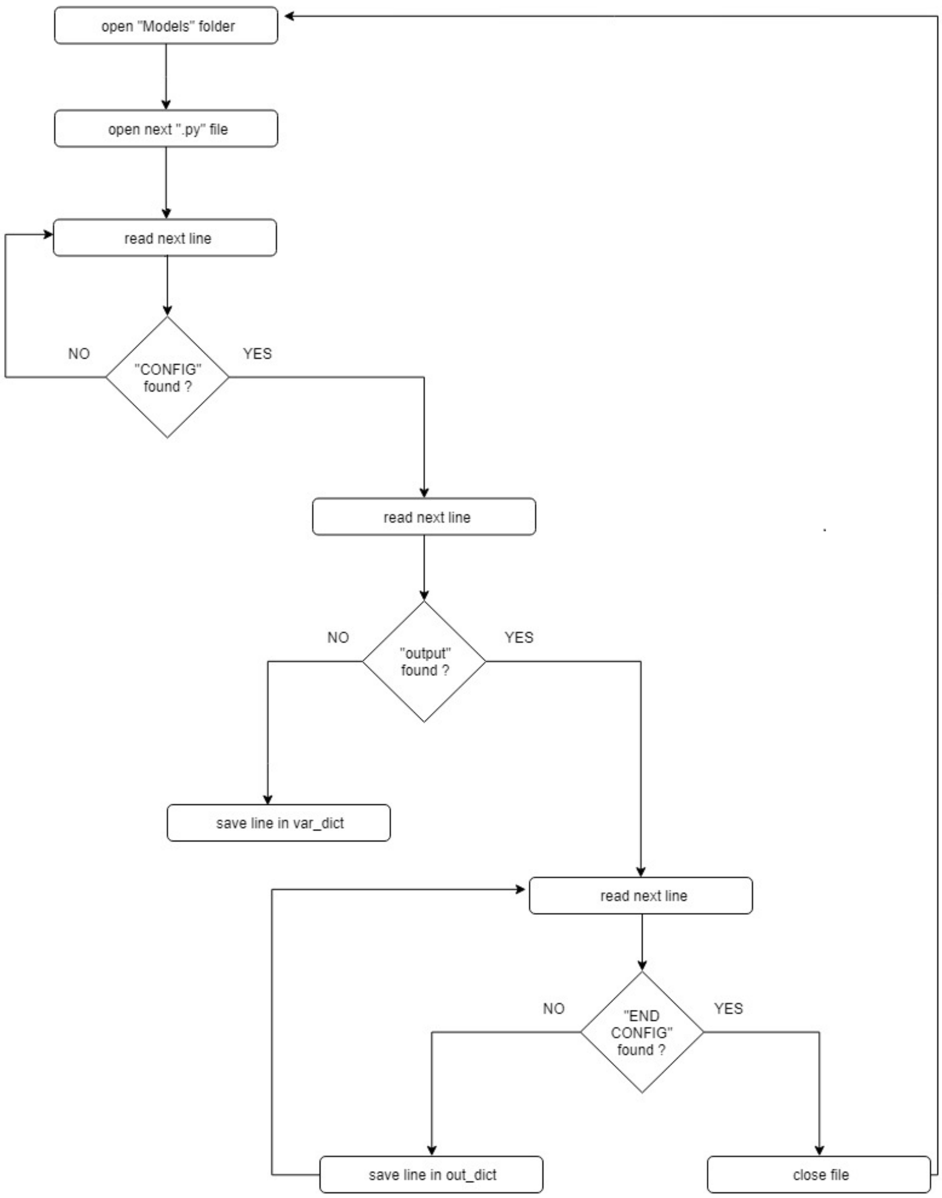
This will leave out the first line, which as already seen is useful for the model parser in order to find out the current category.

A secondary function is run on each read line to extract the measurement unit and save it separately.

When the “output” word is found, the flag changes to that the parser save any future line in the out_dict data structure, flagging them as output parameters.

This continues until the “END_CONFIG” signal is reached, telling the parser to close this model (no more parameters to read) and start next one.

The procedure is exemplified in the following diagram:

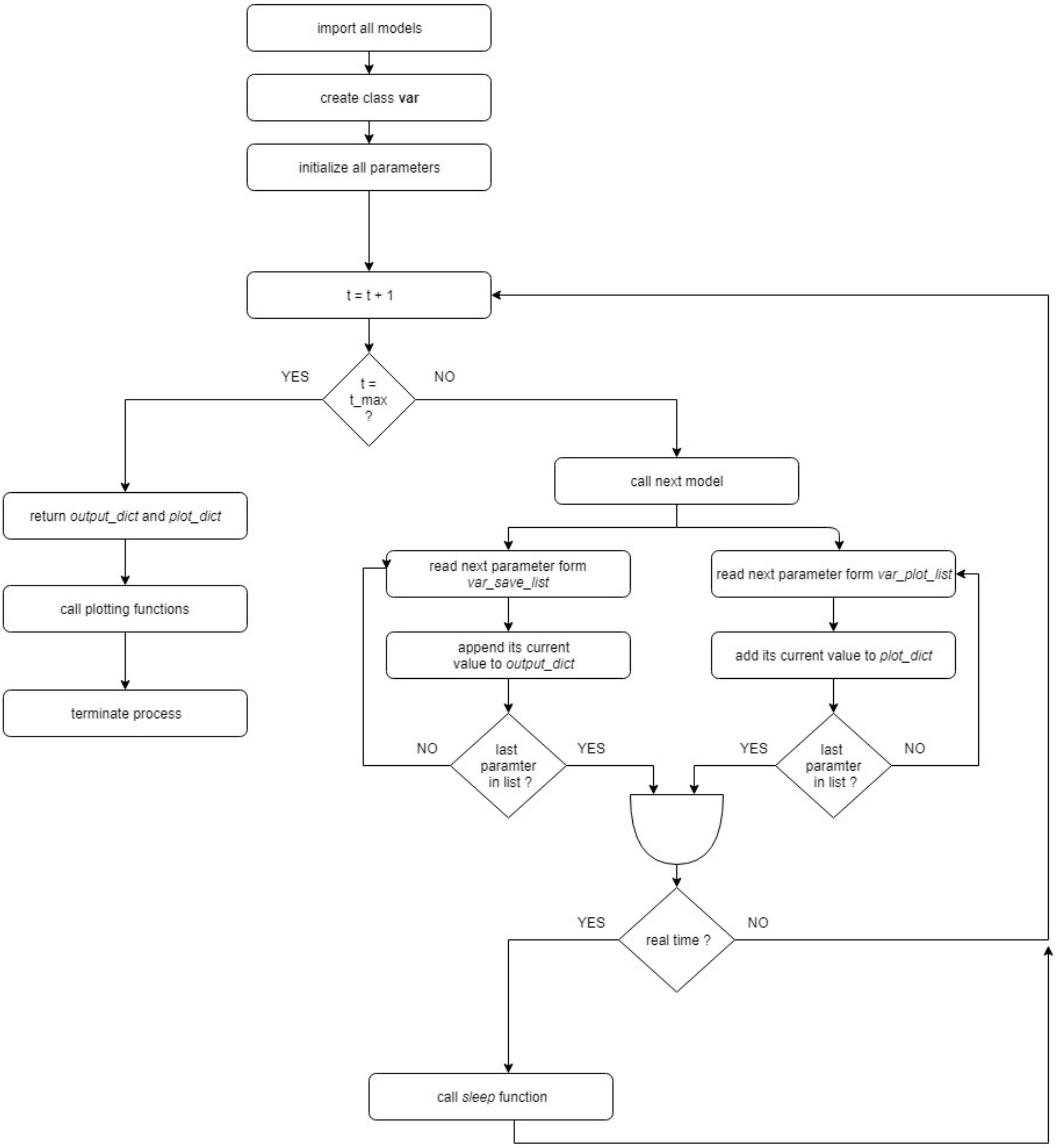


Picture 2-17: parameter parser action diagram

2.6) Process.py

The Process.py file is the real heart and soul of StarSim: it call one by one the selected models in a loop (until the user-defined time expires) and registers the value of the parameters that the user selected to be saved (or plotted)

Its action diagram can be sum up as follows:



Picture 2-18: main simulation process

The *Process.py* file work by the following steps:

- Initialize a new class var, containing all the parameters of the selected models
- Initialized the value of those parameters as selected by the user

- Start time loop; for every iteration, call all selected models in a row

- Each model is passed as argument the previously created class var, so that it modifies parameter in that class which will be later passed to the next model; this ensures that ach models works on the latest version of the parameter set and avoid the annoyance of passing them one by one in the models call.

- At the end of every iteration, parameters saved in the *save_list* or *plot_list* are read from the var class and their value is stored in the appropriate output list.

- If the simulation is real-time, the *sleep()* function is now called for about one second, otherwise next iteration starts immediately.

2.7) Post-processing

The post processor has the aim to present simulation results back in a user-friendly way by following these steps:

- Plot selected parameters automatically

- Call back main window

- Activate new menu item that were previously grayed out: Add plot, save, export

Chapter 3

3 The standard library

3.1) Mission models

These models are normally selected first when setting up a simulation and simulate the satellite motion, its orbit, the relative position of the sun vector and other environmental phenomena such as the magnetic field and the thermal fluxes balance.

3.1.1) Orbit propagator

This model constitutes the foundation of every simulation and has the scope to simulate the satellite's orbit due to Earth's gravitational field.

First step is then is to make sure that the motion of the satellite is described in an appropriate coordinate system, namely the Earth-Centered-Earth-Fixed reference system (ECEF).

This system does not rotate and its origin lies in the center of the Earth; its z-axis comes out from the North pole, the y-axis exits from the interception point of the prime meridian and the equator, and the x-axis complete the tern according to the right-hand rule.

The motion of the satellite around the Earth, assuming its mass to be totally negligible (total mass of a CubeSat is about 1.3 Kg), is accurately described by the following system of equations:

$$\begin{cases} \dot{v} = -\dfrac{GM_{earth}}{|r|^3} \cdot r \\ \dot{r} = v \end{cases}$$

Where G is the gravitational constant.

Note that this model does not include determination of the satellite's attitude, which has to be determined separately.

Many integration method are available to solve the above mentioned system, each with its own advantages and disadvantages; this model implements a so-called RK4 integrator (4^{th} order Runge-Kutta), which was found to be a good compromise between accuracy and computational load.

The RK4 need as input already known *pos* (3-components position vector) and *v* (velocity) values, as well as the function to be approximated:

$$g(pos) = -\frac{GM_{earth}}{|r|^3} \cdot r$$

Where *pos* is the representation of the position vector *r* in Python syntax:

$$pos = \ r[0]^2 + r[2]^2 + r[2]^2$$

Then it works by dividing a single time steps into four sub-steps at which the function is evaluated, and computing a weighted average of those value representing the total increment.

The algorithms used to compute these sub-steps make use of the user-defined time step *h* (usually 1 second):

- first sub-interval:

$$k_0 = h \cdot v$$
$$l_0 = h \cdot g(pos)$$

- Second sub-interval:

$$k_1 = h \cdot (v + \frac{1}{2} l_0)$$
$$l_1 = h \cdot g(pos + \frac{1}{2} k_0)$$

- Third sub-interval:

$$k_2 = h \cdot (v + \frac{1}{2} l_1)$$
$$l_2 = h \cdot g(pos + \frac{1}{2} k_1)$$

- Fourth sub-interval:

$$k_3 = h \cdot (v + \frac{1}{2} l_2)$$

$$l_3 = h \cdot g(pos + \frac{1}{2} k_2)$$

- Compute total increment based on the three sub intervals, giving greater weight to the central values:

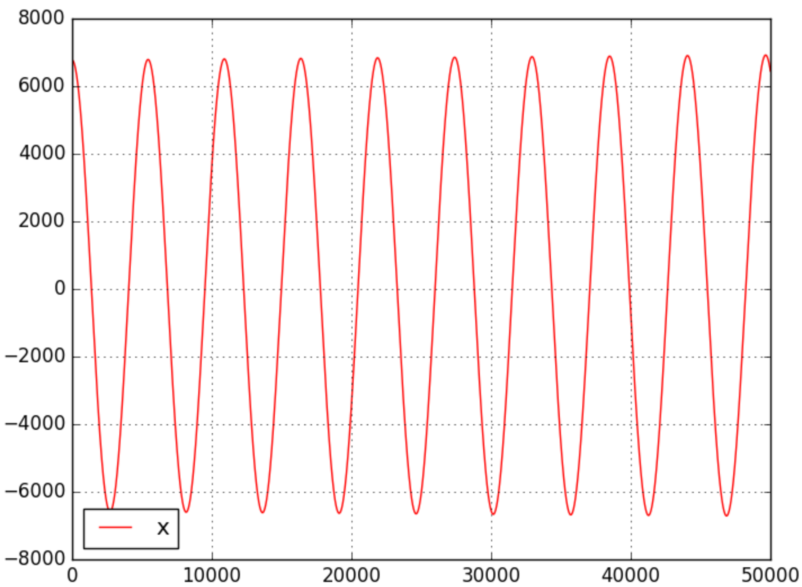
$$incr_{pos} = \frac{k_0 + 2k_1 + 2k_2 + k_3}{6}$$

$$incr_v = \frac{l_0 + 2l_1 + 2l_2 + l_3}{6}$$

- Update current values:

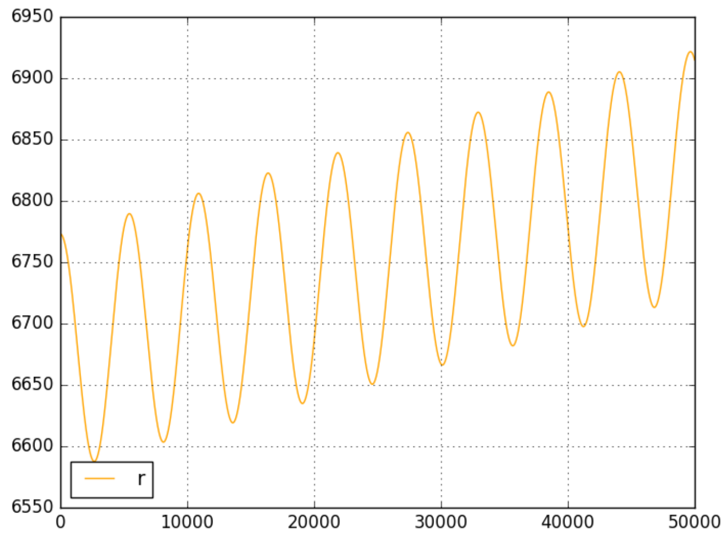
$$pos^{n+1} = pos^n + incr_{pos}$$

$$v^{n+1} = v^n + incr_v$$

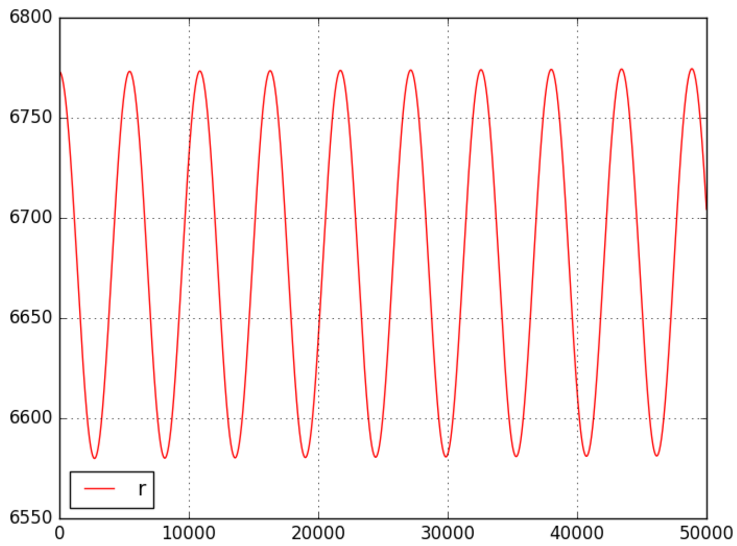


Picture 3-1: sample orbit propagator results

Note that this is an *explicit* method is used i.e. solution at the next step in time depends only from results at previous time steps and not from itself already; this is simpler to implement as it avoids the need of solving nonlinear systems at each step, but brings about the disadvantage that if the step is not chosen wisely (i.e. if the step it's to large) the method can be unstable, so the orbit appear to diverge.



Picture 3-2: numerically diverging orbit



Picture 3-3: numerically stable orbit for a time step lesser than 0.01

Inputs and outputs for the Orbit Propagator model can be sum up as follows:

Input:

major_semiaxis [km] : from the surface (not including Earth's radius)

eccentricity [-]

Inclination [deg]

longitude_of_the_ascending_node [deg]

argument_of_periapsis [deg]

true_anomaly [deg]

Output:

x : spacecraft position along x axis

y : spacecraft position along x axis

z : spacecraft position along x axis

r : total distance of the spacecraft from the center of the Earth

Vx : velocity along the x axis

Vy : velocity along the x axis

Vz : velocity along the x axis

3.1.2) Sun motion

This model uses an explicit, fixed-step Runge-Kutta 4th order method to integrate Newton's equation of motion and return the Earth's path around the Sun.
Output is given in Sun Centered System, whose axis are aligned with ECEF's.
No input is required from the user, as the following parameters are hard-coded:

major_semiaxis: 149597870.7 Km

eccentricity: 0.0167

inclination: 7.155 deg

longitude_of_the_ascending_node: 174.9 deg

argument_of_periapsis: 288.1 deg

Output:

Earth_x : spacecraft position along x axis

Earth_y : spacecraft position along x axis

Earth_z : spacecraft position along x axis

Earth_r : total distance of the Earth from the center of the Sun

Earth_Vx : velocity along the x axis

Earth_Vy : velocity along the x axis

Earth_Vz : velocity along the x axis

3.1.3) Sun vector:

This model calculates the components of the sun vector; no input is required if "Sun Motion" model is present, otherwise the user has to manually enter the three coordinates of the sun vector:

Input:

earth_x [km]: Earth x-coordinate in Sun-centered reference system

earth_y [km]: Earth y-coordinate in Sun-centered reference system

earth_z [km]: Earth z-coordinate in Sun-centered reference system

x [km]: satellite x-coordinate in ECEF system

y [km]: satellite y-coordinate in ECEF system

z [km]: satellite z-coordinate in ECEF system

Output:

sun_vector_x: sun vector x-component in Sun-centered system

sun_vector_y: sun vector y-component in Sun-centered system

sun_vector_z: sun vector z-component in Sun-centered system sun_vector_magnitude: module of the sun vector

sun_vector_direction_i: normalized x-component of the sun vector

sun_vector_direction_j: normalized y-component of the sun vector

sun_vector_direction_k: normalized z-component of the sun vector

It is as simple as performing a vector-wise sum between the Earth’s and the spacecraft position vectors.

3.1.4) Magnetic field dipole:

This model computes the three mutually normal components of Earth's magnetic field in ECEF system.
It takes as input a specific point in the (latitude, longitude, altitude) format, so an converter from ECEF to *latitude-longitude-altitude* model is required; this model is also provided in the standard library.
Earth's magnetic field is approximated as simple dipole field.

Input:

lat [deg]: latitude of current point

long [deg]: longitude of current point

alt [deg]: altitude of current point

Output:

Bx: x-component of magnetic vector in ECEF system

By: y-component of magnetic vector in ECEF system

Bz: z-component of magnetic vector in ECEF system

B_mod: module of the magnetic vector

It approximates the radial and tangential component of the magnetic field as:

$$B_r = \frac{2\mu m \cos{(\theta)}}{4\pi r^3}$$

$$B_\theta = \frac{\mu m \sin(\theta)}{4\pi r^3}$$

Where R includes Earth's radius, μ is the magnetic permeability, m is a constant of value $m = 7.94 \cdot 10^{22}$ and θ is the angled defined as $\frac{\pi}{2} - \lambda$, λ being the usual latitude value.

This approximation is definitely not too much accurate, but will do the job for a quick, first order estimation.

To obtain a much greater accuracy, a more detailed model such as the standard World Magnetic Models is required.

The World Magnetic Model, freely available over the Internet, describes in a particular detailed fashion Earth's magnetic field and its changes in time; it was originally developed jointly by the United States *National Geophysical Data Center* and the British *Geological Survey* and it is actualized every 5 years.

It works by approximating the magnetic field with help of 12 spherical harmonic expansion of the magnetic potential of the geomagnetic main field generated in the Earth's core; due to its highly intrinsecal complexity its computational cost is also very high, so it has been decided to avoid using this model is the present thesis; such a great order of accuracy is not needed anyway for a preliminary phase.

Anyway, WMM has already become the standard in many national and international services and it's the default choice in all devices equipped with a magnetic sensor (e.g. Smartphones); it is capable of delivering the correct measure of Earth's magnetic field for one kilometer under Earth's surface until 850 Km above.

Another magnetic mode that has been taken into consideration was the *International Geomagnetic Reference Field (IGRF)* developed by the *International Association of Geomagnetism and Aeronomy (IAGA).*

3.1.4) Thermal fluxes

Knowledge of each face temperature is required to perform a detailed simulation, as it deeply affects the current output generated by the solar panels.

This model solves the thermal balance equation separately for each face to determine face temperature.

Only radiative fluxes are considered (convection between adjacent faces is neglected).

Infrared radiation from Earth is assumed to be $239\ \frac{W}{m^2}$ and albedo is 29% of the solar constant.

The following constants are assumed:

absorption coefficient = 0.92 (α)

emissivity = 0.85 (ϵ)

density = 5000 $\frac{Kg}{m^3}$ (average density of face and solar panel combined)

Cp = 1500

Input:

Temperature [K]: initial temperature for all faces

face_area []: face area of all faces

Output:

T_x: temperature of face "x"

T_y: temperature of face "y"

T_z: temperature of face "z"

T_neg_x: temperature of face "-x"

T_neg_y: temperature of face "-y"

T_neg_z: temperature of face "-z"

The model works by the following steps:

- Calculate the incoming heat flux:

$$q_{sc} = 1367 \cdot \cos(\theta_{sun})$$
$$q_{ir} = 239 \cdot \cos(\theta_{earth})$$
$$q_{albedo} = 0.29 \cdot 1369 \cdot \cos(\theta_{earth})$$
$$q_{in} = \alpha \cdot (q_{sc} + q_{ir} + q_{albedo}) \cdot A$$

- Compute the flux generated by the satellite itself by dissipating power within its electronics; since we can assume that this flux equally distributed among all 6 faces, and being known the efficiency of the internal load we can write:

$$q_{gen} = \frac{(1-\eta) \cdot P_{load}}{6}$$

- Compute the flux irradiated by the satellite into space according to Boltzman's law:

$$q_{out} = \sigma \cdot \varepsilon(T_i^4 - T_{space}^4) \cdot A$$

- Compute total flux exchanged by the satellite:

$$q_{tot} = q_{in} - q_{out}$$

- Compute the final temperature of that face:

$$T_i = \frac{q_{tot}}{\rho C_p V} \cdot t_{step}$$

3.2) Power sources and actuators

These models have been studied specifically to model the electric power system of the satellite, finding a reasonable compromise solution between accuracy and computational cost.

Since this thesis is modeled around the needs of a students' CubeSat, only solar panels are considered as appropriate power sources.

3.2.1) Constant resistive load:

This model represents a simple, constant resistive load.
A *power_bus* model must be present in order to connect this model to solar panels or batteries and all resistive loads are considered to be set in parallel.

Input:

load_dissipated_power [mW]: power dissipated by the satellite

Output:

load_voltage
load_current
load_dissipated_power

This model assumes that at any given time the spacecraft dissipates a constant, known amount of power, given in mW. The actual value of power dissipated varies according to the current mission phase and their evolution is detailed in the section "modeling the eps"; it can be roughly estimated as 1590 mW when the satellite is not transmitting to the main ground station.
Output values are easily computed from Ohm's law.

3.2.2) Solar panel:

This model represent a realistic, non-linear solar panel.
The model comes with the following hard-coded constants:

Ki = 0.002: single cell short circuit current [A]
n = 1.2: diode ideality factor

Tr = 298.15: nominal temperature [K]

Eg0 = 1.1: energy band gap [eV]

Rs = 0.001: series resistance [Ohm]

Rsh = 1000: shunt resistance [Ohm]

Input:

sun_vector_direction_i [-]: not present if "Sun_vector" model is included

sun_vector_direction_j [-]: not present if "Sun_vector" model is included

sun_vector_direction_k [-]: not present if "Sun_vector" model is included

face_position [string]: face on which the panel is positioned, e.g "x", "z", "-x"

short_circuit_current [A]: as reported on the datasheet, under standard irradiation values

open_circuit_voltage [V]: as reported on the datasheet, under standard irradiation values

load_resistance [Ohm]: not present if at least one resistive model is included in the simulation.

Output:

solar_panel_I_out [A] : current being produced by the solar panel

solar_irradiation [W/m^2]

This model can work in three different modes:

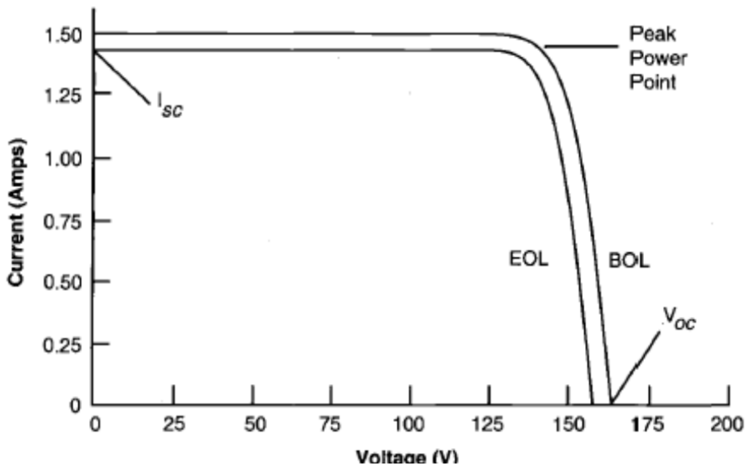
1. Eclipse: is the spacecraft is being eclipsed; its output voltage and current are both zero.

2. With MPPT (Maximum Power Point Tracker): is a MPPT is also present in the selected model list, this panel will output in maximum possible current value; for more information about how the MPPT works, refer to its section in this chapter.

3. Direct load connection: is no MPPT is present the panel is connected directly to the load (possibly with a battery in between); In this case, as the characteristic

equations are given in the form on I-V dependency, it performs an iteration cycle to reach convergence in the value of V (an thus I), given the load.

In mode 2) and 3), output voltage and current are required to conform to the general solar panel characteristic curve (below), which is found by following these steps:



Picture 3-4: standard solar panel charachteristc curve

1. Photovoltaic current:

$$I_{ph} = [I_{sc} + K_i(T - 298)]\frac{I_r}{1000}$$

2. Reverse saturation current:

$$I_{rs} = \frac{I_{sc}}{exp\left[\frac{q \cdot V_{oc}}{N_s KnT} - 1\right]}$$

3. Saturation current:

$$I_0 = I_{rs}\left[\frac{T}{T_r}\right]^3 exp\left[\frac{q \cdot E_{g0}}{nK}\left(\frac{1}{T} - \frac{1}{T_r}\right)\right]$$

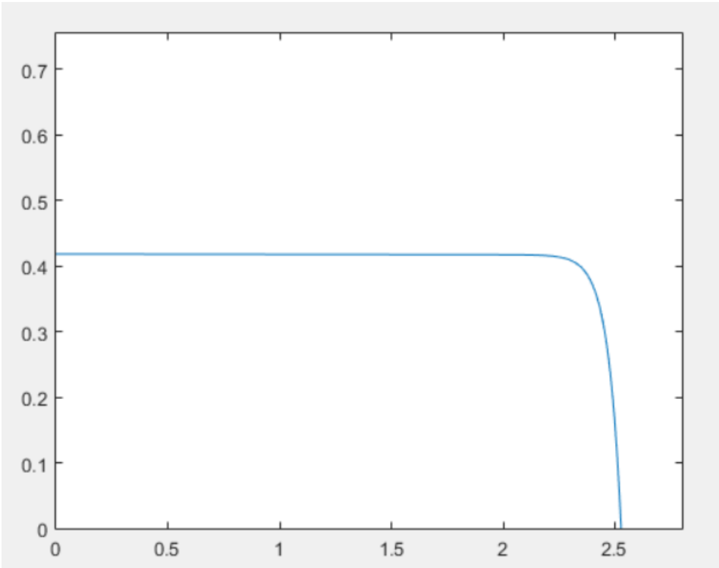
4. Shunt-resistance current:

$$I_{sh} = \frac{V\frac{N_S}{N_P} + I \cdot R_S}{R_{sh}}$$

5. Output current:

$$I_{out} = N_p I_{ph} - N_p I_0\left[exp\left(\frac{\frac{V}{N_s} + I\frac{R_S}{N_P}}{nV_t}\right) - 1\right] - I_{sh}$$

As already mentioned before, since both V and I_{out} are originally unknown, the algorithm iterates over steps 4) and 5) until convergence is reached.



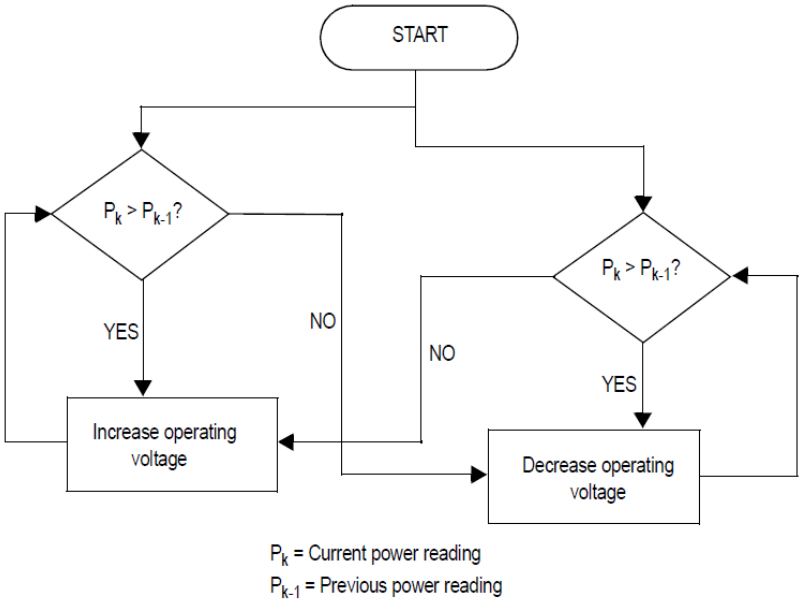
Picture 3-5: solar panel curve implemented in MATLAB for testing

3.2.3) MPPT

This acronym stand for *Maximum Power Point Tracking* and it's the models responsible for maximizing the output of a given solar panel.

The algorithm used is relatively simple and its key concept consist in introducing a perturbation in the panel operating voltage (physically this would be done by modifying a converter duty cycle).

Picture 3 below illustrates the functioning principle of a Perturbe and Observe algorithm for a MPPT:



Picture 3-6: MPPT perturbe & observe algorithm

After performing an increase in the panel operating voltage, the algorithm compares the current power reading with the previous one. If the power has increased, it keeps the same direction (increase voltage), otherwise it changes direction (decrease voltage). This process is repeated at each MPP tracking step until the MPP is reached.

After reaching the MPP, the algorithm naturally oscillates around the correct value.

3.2.4) Battery

This model represent a standard LiPo battery pack, and comes with the following implemented features:

- 8-th order polynomial to represent variation of nominal voltage as function of the Depth of Discharge i.e. nominal voltage at each time step is computed as

$$V_{actual} = V_{nominal}\big(-8.281DOD^7 + 23.5743\mathrm{DOD}^6 - 30\mathrm{DOD}^5 + 23.703\mathrm{DOD}^4 - 12.587\mathrm{DOD}^3 + 4.135\mathrm{DOD}^2 - 0.865\mathrm{DOD} + 1\big)$$

- Peukert's law for capacity-discharge rate dependency with hard-coded exponent 1.15

Input:

nominal_voltage [V]: as reported on the datasheet, at 1C discharge rate

capacity [mAh]: maximum capacity of all cells combined, at 1C discharge rate

load_resistance [Ohm]: not present if at least one other resistive model is present in the simulation

maximum discharge rate [C]: maximum rate at which the current can be drawn out of the battery; 1C is the rate of discharge that will completely discharge the battery in one hour.

Output:

battery_I_out [A]: the current being drawn out of the battery

Chapter 4

4 Case Study

4.1) Modeling the Electric Power System

A cubesat satellite can be expected to work in different operative modes according to the current mission phase, summed up in the table below.
The different operating modes can be detailed as follows:

- Dormant mode: the satellite will stay turned off during the launch, with no voltage present; this is the so called *dormant mode*.

- Activation mode: this is a transient mode, which activates as the satellite exits the launch Pod: it then turns on its OBC and starts its activation sequence.

- Detumbling mode: this mode starts as soon as the activation sequence is complete and lasts about 100 minutes (one orbit), and the ADCS is operated;

- Basic Mission / Full Mission mode: the nominal operating mode if a Cubesat, distinguished into basic or full according to which system is actually operating.

- Fail safe mode: in this mode, the OBC and EPC remain active and consume minimum power; its main aim is to avoid an incorrect activation of the satellite.

- Safe Mission mode: this mode is activated in the eventuality of payload loss; it’s still possible to communicate with the spacecraft, even though its attitude can no longer be controlled.

Operative mode	Active SubSystems
Dormant	none
Detumbling	OBC+ADCS+EPS
Antenna	OBC+ADCS+EPS
Commissioning	OBC+ADCS+EPS+COMSYS
Mission	
1. Full	OBC+ADCS+EPS+COMSYS (30sec)
1. Basic	OBC+ADCS+EPS+COMMS (120sec)
Safe Mission	OBC+EPS+COMMS
Save Energy	OBC*+EPS*+ADCS*

Picture 4-1: CubeSat's operating modes

4.1.1) Dissipated power

The total power consumption for the Full Mission mode can be computed by taking into account the power consumption for a single orbit.

There are three main kinds of orbit that can affect the total power requirement:

• Case I) the spacecraft does not pass over the main GCS. The e-st@r Cubesat is designed to transmit the telemetry every two minutes, and the signal lasts about 2 seconds. It remains this basic transmission mode for 103 sec per orbit (1.72 min/orbit).

Transmitting the signal takes up about 2500 mW, while idle power consumption in this mode is estimated in 900 mW.

This means:

$$P_I = \frac{1.72}{60} h \cdot 2500\, mW + \frac{103 - 1.72}{60} h \cdot 900\, mW = 1590\, mW$$

• case II) the spacecraft passes over the main GCS. The satellite stays in full transmission mode for 11.4 min, meaning that in this period it is transmitting continuously. Therefore it remains in basic transmission mode for 2 seconds every two minute for 103-11.4 min per orbit (1.52 min/orbit).

$$P_{II} = \frac{1.52}{60} h \cdot 2500\, mW + \frac{11.4}{60} h \cdot 2500\, mW + \frac{103 - 11.4 - 1.52}{60} h \cdot 900\, mW = 1890\, mW$$

• case III) detumbling: taking into account how the of ADCS subsystem has been designed, a longer phase has to be considered to achieve the stabilization of the satellite.

This phase lasts about 9000 seconds (150 min).

Case III (detumbling) will be ignored in the following of this work, and attention will be focused mainly on case I, the one that most commonly occurs.

In order to proceed with a detailed modeling of a Cubesat's EPS, precise information about the actual solar panels and battery in use is needed.

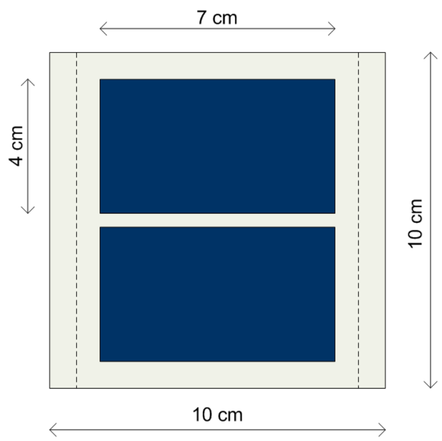
4.1.2) Solar panel

Our Cubesat relies mainly on GaAs (Gallium Arsenide) Triple junction solar cells

The primary energy source for e-st@r satellite are Triple Junction GaAs solar cells, positioned on five out of the six available faces (one face is reserved for the antenna and its deployment system). Each solar panel is constituted by 2 solar cells connected in series, the single solar cell dimensions featuring 4 cm x 7 cm in size (limited by face dimensions)

Solar cell type	GaAS Triple Junction
Efficiency	27.82 %
Open circuit voltage	2.60 V
Short circuit current	454.67 mA
Voltage @ Pm	2.33 V
Current @ Pm	427.56 mA

Picture 4-3: solar cell features



Picture 4-2: solar cell size

4.1.3) Battery

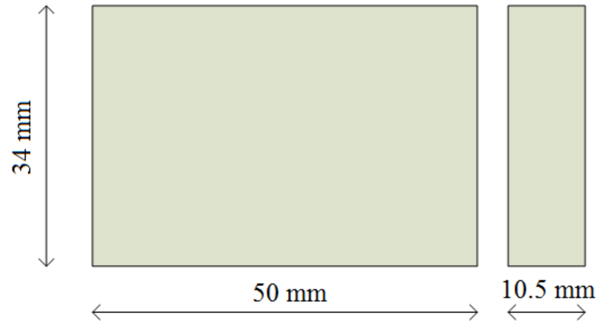
The Cubesat comes equipped with two batteries, each being constituted by two Li-Ion cells in series; this kind on batteries are one of the most common off-the-shelf components and have already been used by a certain number of small satellites in the past.

They feature a dimension fully compatible with a Cubesat's size restriction and their energy density is also acceptable.

The main characteristics of such batteries are listed in the table below:

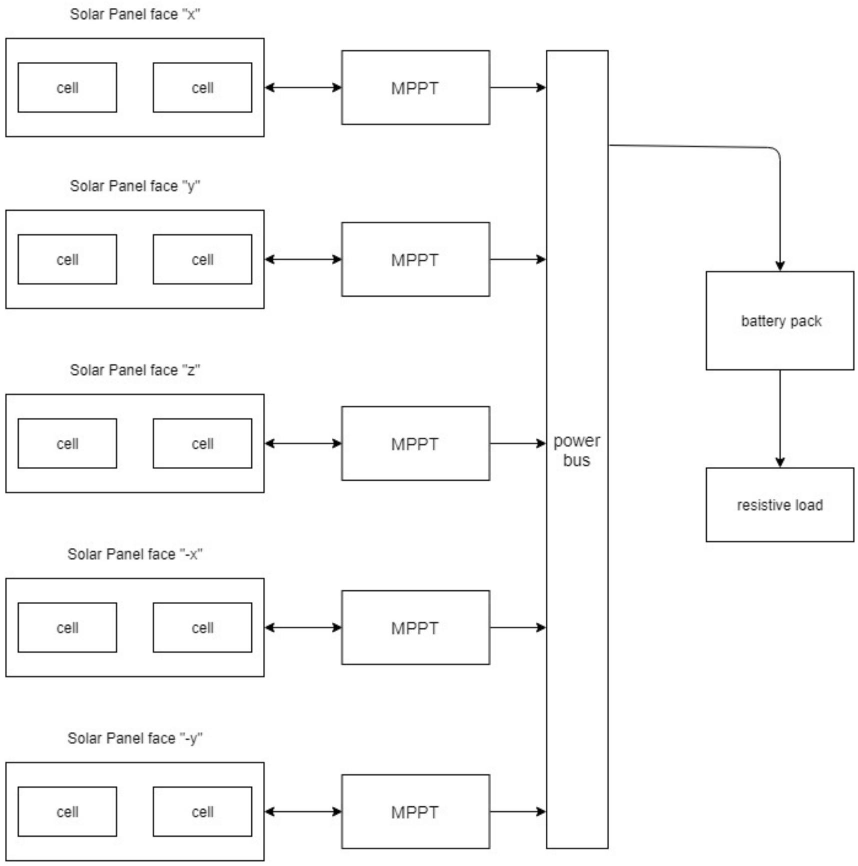
Cell type	Li-Ion
Nominal voltage	3.7 V
Capacity	1800 mAh
Charge rate	Standard (0.5 C)
Max. discharge rate	1C
Height	10.5 mm
Width	34.0 mm
Length	50.0 mm
Weight	41.2g

Picture 4-4: battery features



Picture 4-5: battery size

4.1.4) Full model



Picture 4-6: full EPS simplified model

The dynamic behavior of solar panels, batteries and MPPT has already been detailed in chapter 3.
A certain number of assumptions has been made in order to simply the model to be simulated:

- Filters and voltage regulator are negliged

- The satellite dissipates a constant power equal to 1590 mW as seen in section 4.1.1

- MPPTs are totally efficient and produce no oscillation in their outputs

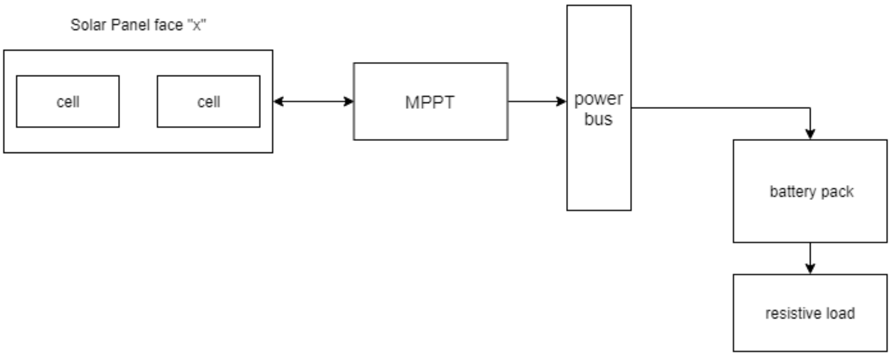
- The two batteries are modeled as a single battery pack with a cumulative voltage of 7.4 V

4.2) AIL EPS Setup and scenarios

A certain number of AIL simulations can be now carried on to validate the proposed schematics of the EPS.

4.2.1) Single panel, fully irradiated

In this scenario, we suppose to have only one solar panel in full irradiation (pointing directly towards the Sun) and all the other in full shadow; the satellite is flying a standard orbit and no telemetry is currently being transmitted apart from the usual 2 seconds signal.
Power consumption thus can be assumed as 1590 mW.



Picture 4-7: schematic model for single-panel simulation

This translates to the following model list:

Environmental models:

- Orbit propagator
- Sun motion
- Sun vector
- Thermal fluxes

Actuators:

- Constant resistive load

Power Sources:

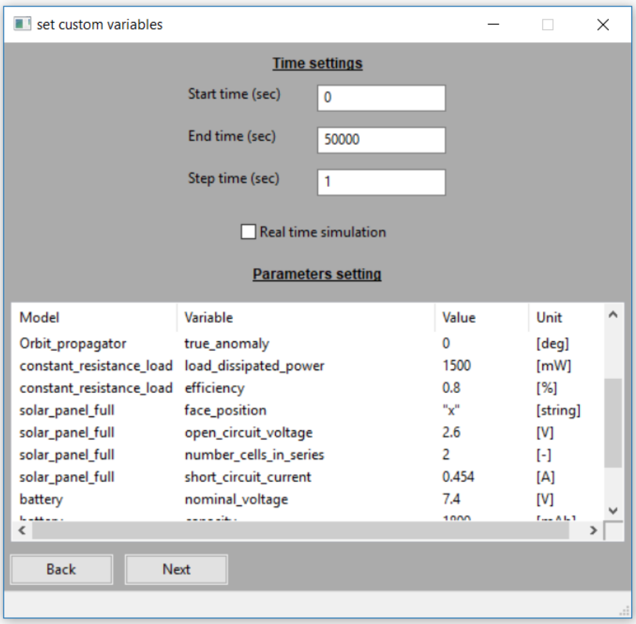
- Solar panel
- MPPT
- Power bus
- Battery

Category	Model	Condition
SPACECRAFT_MOTION	Orbit_propagator	-
ENVIRONMENT	Sun_motion	-
ENVIRONMENT	Sun_vector	-
ENVIRONMENT	thermal_fluxes	-
ACTUATORS	constant_resistance_load	-
POWER_SOURCES	solar_panel_full	-
POWER_SOURCES	MPPT	-
POWER_SOURCES	power_bus	-
POWER_SOURCES	battery	-

Picture 4-8: EPS simulation implemented in StarSim project window

The picture above shows a partial screenshot of StarSim's project window where this setup has been implemented.

The simulation is setup to run 50.000 seconds (about 8 orbits), with a time step of one second; no real-time is required.



4.2.2) Three panels, standard orbit

If no attitude determination algorithm is present (i.e. the satellite does not rotate),then no more than three panels need to be simulated as the remaining two are constantly shadowed.

Standard orbit for a CubeSat mission is about 400 km of height and 96° of inclination; again, estimated dissipated power is 1590 mW.

The model flow changes with respect to the situation above as long as now three solar panel models are present instead of one:

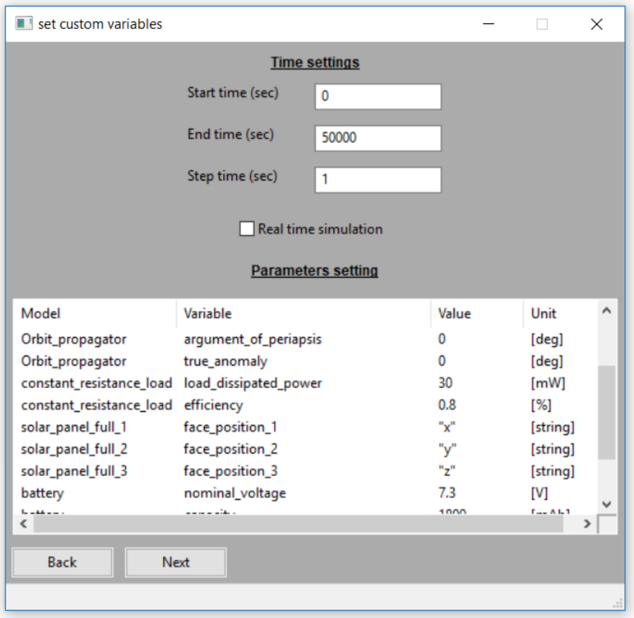
Category	Model	Condition
SPACECRAFT_MOTION	Orbit_propagator	-
ENVIRONMENT	Sun_motion	-
ENVIRONMENT	Sun_vector	-
ACTUATORS	constant_resistance_load	-
POWER_SOURCES	solar_panel_full	-
POWER_SOURCES	solar_panel_full	-
POWER_SOURCES	solar_panel_full	-
POWER_SOURCES	MPPT	-
POWER_SOURCES	power_bus	-
POWER_SOURCES	battery	-

Picture 4-10: three panels simulation setup

The picture above shows a partial screenshot of StarSim's project window where this setup has been implemented.

The simulation is setup to run 50.000 seconds (about 8 orbits), with a time step of one second; no real-time is required.

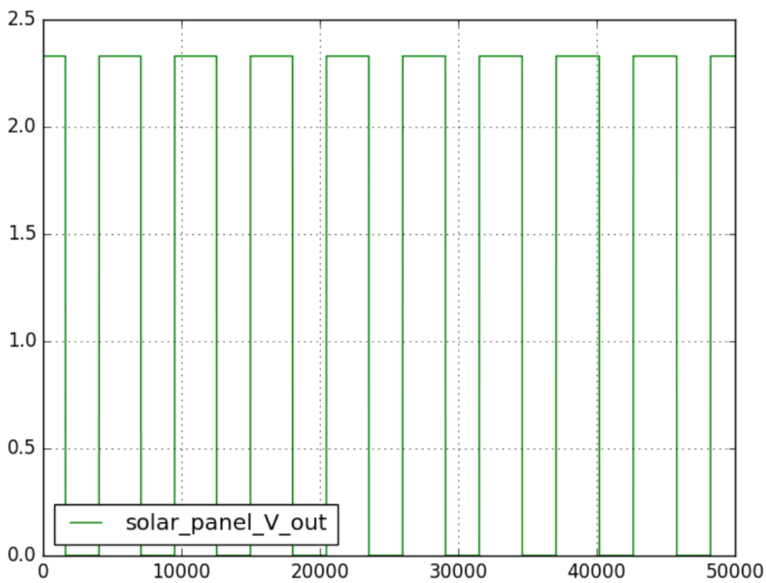
In the parameter definition window, StarSim has already taken care of differentiating them by adding a serial number:



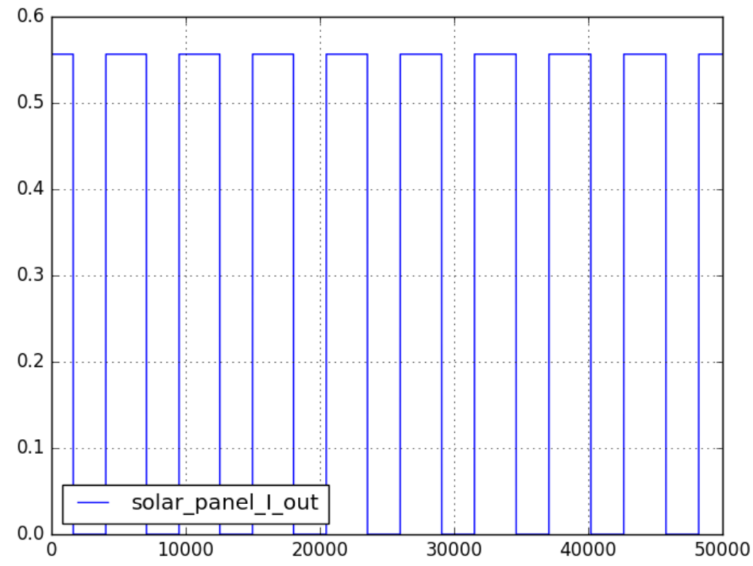
Picture 4-11: parameters setup for full-panels simulation

4.3) AIL results analysis

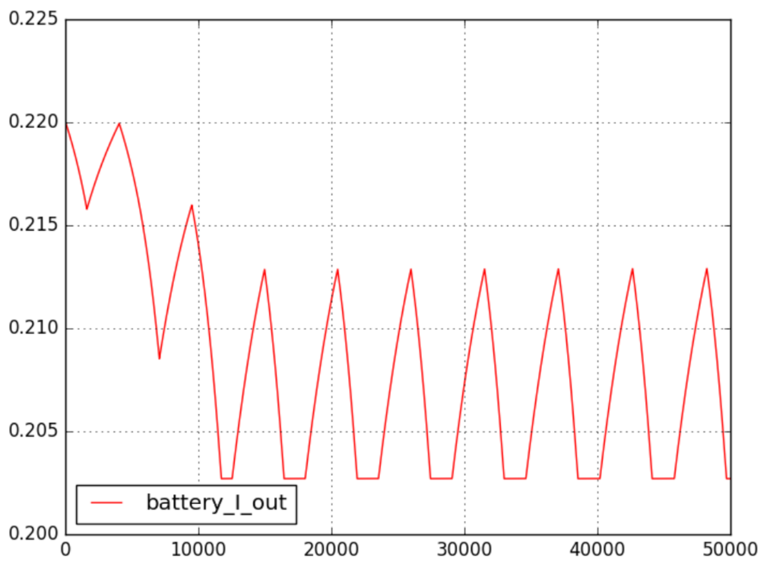
Results for the single-panel, standard-orbit simulation are shown here in graphic form: StarSim makes use of the common matlab-style *matplotlib* library to enable the user to produce easy and fast plots and to save them in a multiplicity of formats.



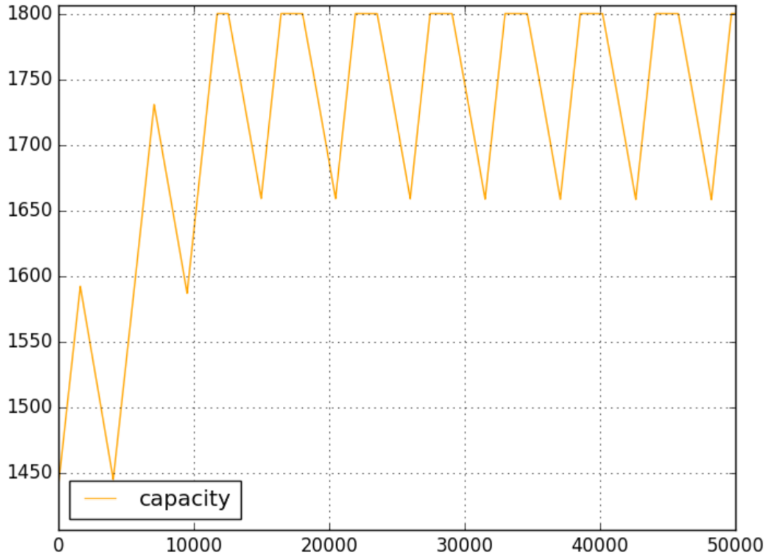
Picture 4-12: solar panel voltage results



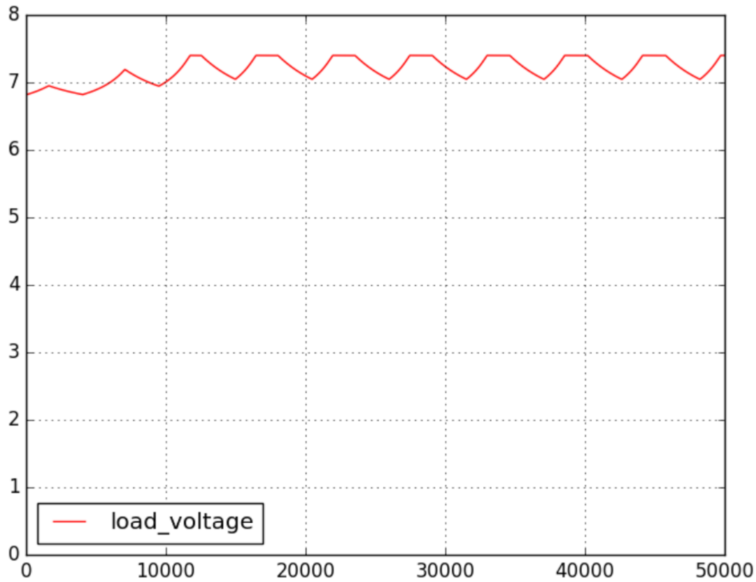
Picture 4-13: solar panel current results



Picture 4-14: current drawn from battery



Picture 4-15: battery capacity results



Picture 4-16: load voltage results

The voltage and current values provided by the solar panel model are fully satisfactory; they appear to be constant (apart of course for the eclipse periods) because the simulated time doesn't run long enough to see a significant difference as the Earth changes its angulation revolving around the Sun (keep in mind that in this scenario the solar panel is being constantly kept pointing towards the Sun).

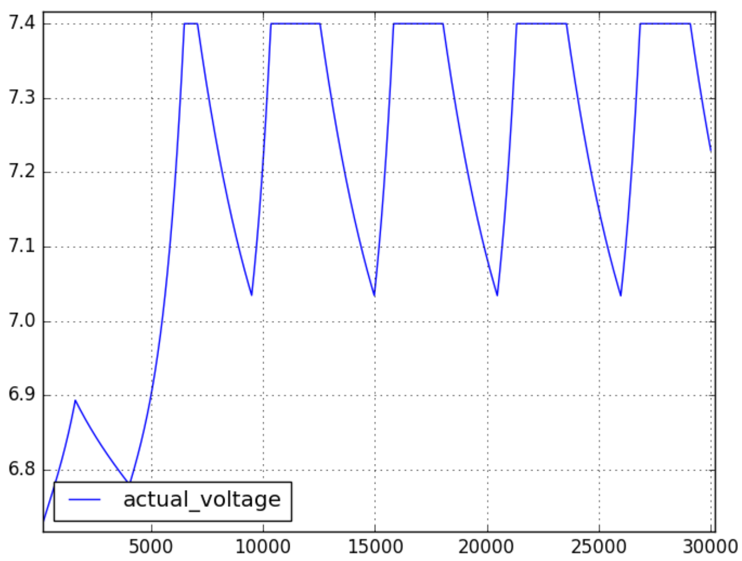
The battery starts the mission with a 20% depth of discharge and quickly comes back to its maximum nominal value of 1800 mAh (the charge then stops abruptly, like if a regulator circuit was present).

The maximum discharge reached by the battery in this scenario is roughly 10%.

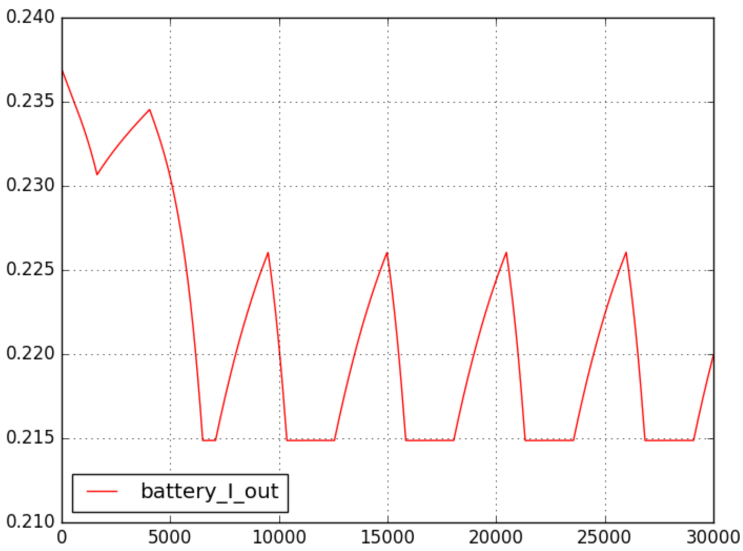
The actual voltage provided by the battery pack oscillates (modestly) around the nominal value of 7.4V due to the variation in the depth of discharge; in this case nominal voltage is the one defined when the battery is fully charged, but be aware that different manufacturers can define their nominal voltage as the one provided when the battery is 80% charged (it must be anyway stated in the datasheet).

Since the load model is programmed to dissipate a constant power of 1590 mW, the actual current flowing out of the battery also oscillates (describing sinusoid arcs) following the fluctuation in the battery voltage to keep the power constant.

For the second simulation scenario, plots of battery state of charge and current drawn can be immediately extracted as well:



Picture 4-17: battery actual voltage



Picture 4-18: current drawn from battery

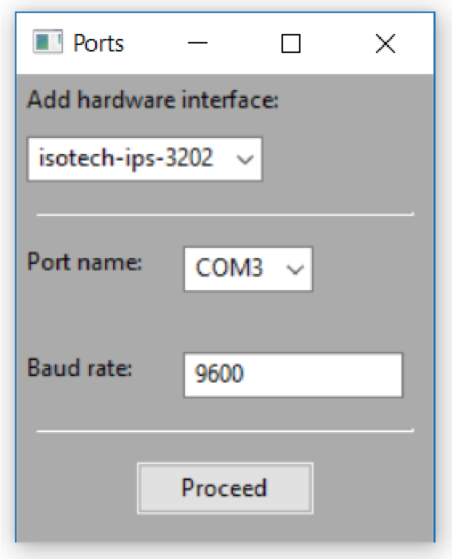
Note that the battery discharge very little in this scenario, no more than 10%.

4.4) HIL setup

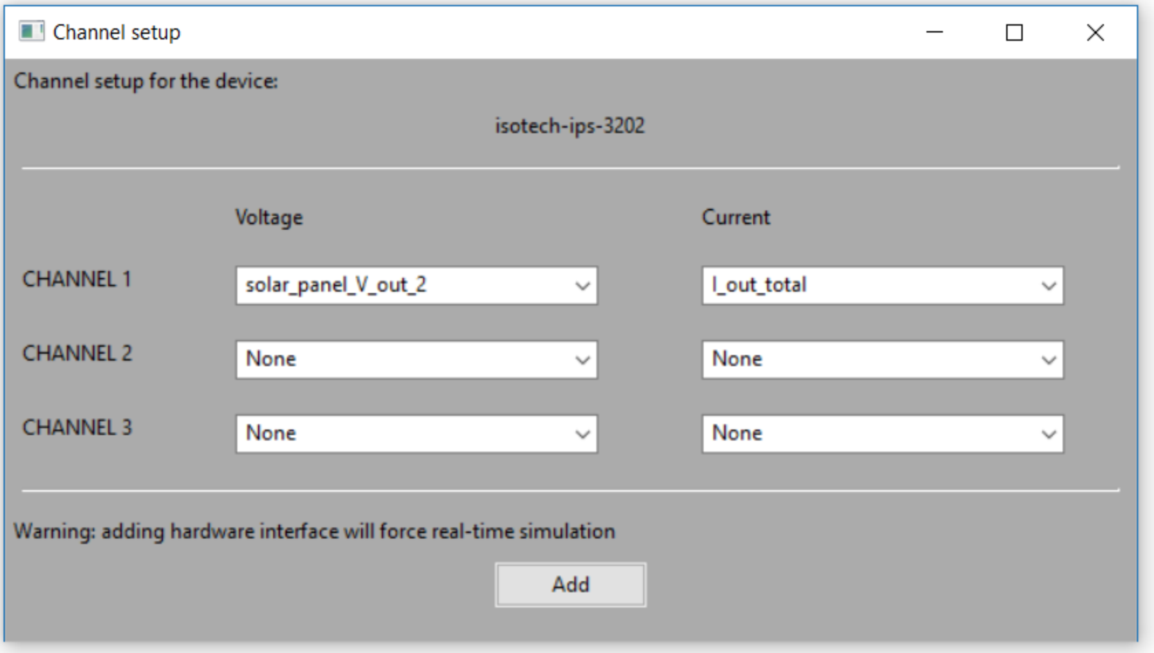
It is possible to perform a Hardware-in-the-loop simulation by integrating a signal generator device into the algorithm loop; the process follow the following steps:

- Open a serial port communication for the device
- Set its address and baud rate
- Define a one-to-one correspondence between channels of the device and parameters of the simulation
- Send the device in output mode and then start the simulation.

Once the AIL model flow has been defined, the user can click over *Tools > Add hardware interface* and set up the following windows:



Picture 4-19: addding hardware interface



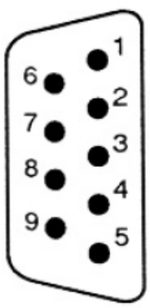
Picture 4-20: channel setup

Adding a hardware interface will force real-time mode on the simulator, but it is still possible to set a time step different from 1.

The device used in this simulation was an *isotech-ips-3202* generator, connected to the workstation through a RS-232 serial port.

It takes advantages of three different channels, and each one of them can accept an input command in the form of either voltage or current.

Picture 4-21: isotech-ips-3202 generator



1. No connection
2. Receive Data (RxD) (input)
3. Transmit Data (TxD) (output)
4. No connection
5. Signal Ground (GND)
6. No connection
7. No connection
8. No connection
9. No connection

Picture 4-22: RS232 standard

A personal computer or workstation fitted with a COM port is essential in order to operate the device via the RS232 interface; the port must then be made available to the software by the simple routine such as the following:

```python
def port_init(address, timeout_time, baudrate):
    ser = serial.Serial(address, timeout=timeout_time)
    ser.baudrate = baudrate
    port_name = ser
    return port_name
```

The actual values of address, timeout time and baud rate are defined in the process.py file; while the baud rate is almost always constant at 9600, the timeout time is set to be 2 seconds more than the total simulated time (to be sure the port is freed at the end of the simulation) and the address (e.g. COM1, COM2…) depends upon which physical port has been connected.

This function returns the in-the-loop port name (in this case, *ser*) that will be used by the software to command the device.

Commands for a single channel are send in the following form:

$$:\mathrm{CHAN1}:\mathrm{VOLT}\ 9.36;\ \mathrm{CURR}\ 0.340\backslash\mathrm{n}$$

Where *\n* is the usual syntax for the line feed (LF) that signals the termination of a command.

Wishing to command all the three channels, such strings can simply be concatenated:

$$:\mathrm{CHAN1}:\mathrm{VOLT}\ 9.36;\ \mathrm{CURR}\ 0.340\backslash\mathrm{n}:\mathrm{CHAN1}:\mathrm{VOLT}\ 8.54;\ \mathrm{CURR}\ 0.440\backslash\mathrm{n}$$
$$:\mathrm{CHAN1}:\mathrm{VOLT}\ 2.36;\ \mathrm{CURR}\ 0.05\backslash\mathrm{n}$$

A formatting function is then necessary, to read input parameters from the results provided by the AIL simulation and format it to be ready to be fed to the generator.

4.4.1) Hardware integration

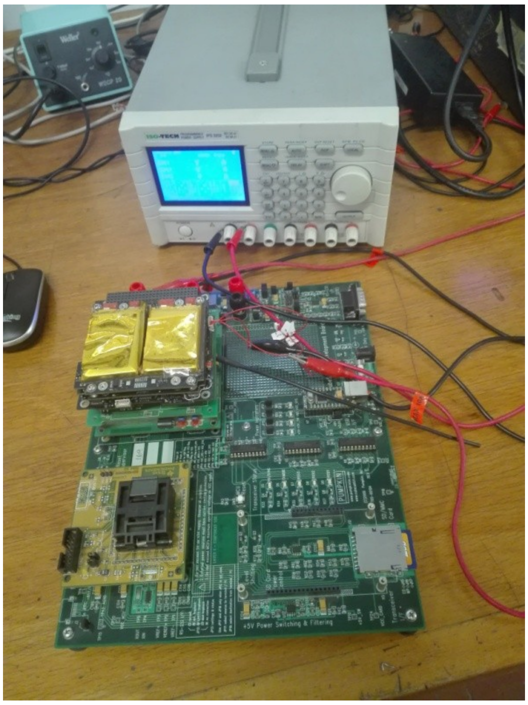
The complete setup for the hardware-in-the-loop simulation consisted of the following items:

- Personal computer where *StarSim* was running

- Function generator to output currents and voltages of the solar panels

- Satellite boards completed with batteries (item to be tested)

- Ground station to receive real-time diagnostic data

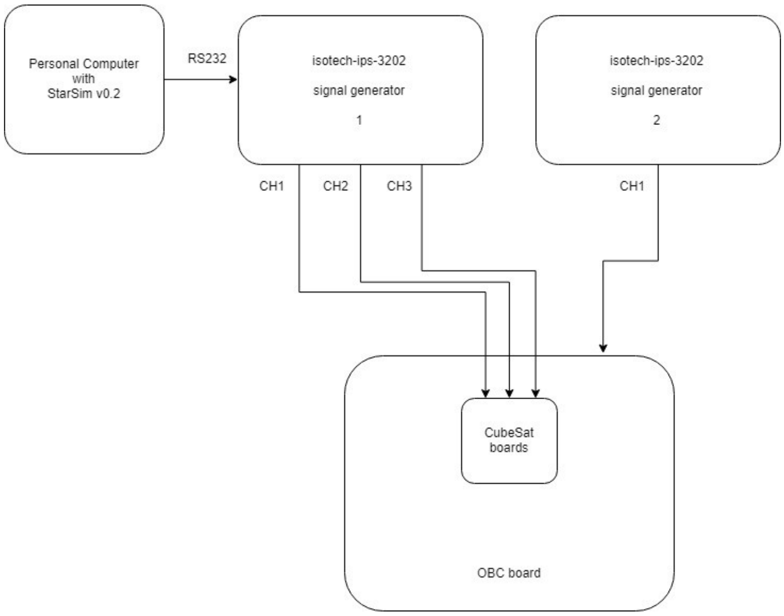


Picture 4-23: laboratory setup

In the picture represents the generator and its connection to the CubeSat's internal boards: from the bottom to the top they are the ADCS (Attitude & Determination Control System), the COMSYS and the EPS, with the two batteries enveloped in the bright reflective layer on top.

The larger board at the bottom is an expanded version of the OBC (On-Board Computer), which allows for easier testing and experimenting.

The setup schematic was the following:



Picture 4-24: HIL setup schematics

Scope of this HIL simulation is to validate the behavior of the solar panels and the charge/discharge cycle of the batteries; therefore, three channels are needed, one for each solar panel (the other two of them are constantly shadowed).

4.4.2) HIL process

The Hardware-in-the-loop core process does not differ significantly from the standard AIL simulation, except of course in the addition of hardware interfaces; the whole process can be summed up in the following steps:

Initialization phase:

- Initialize a new class *var*, containing all the parameters of the selected models (same as AIL simulation)

- Initialized the value of those parameters as selected by the user

- Open communication port selected by the user and setup its baud rate

Simulation phase:

- Start time loop; for every iteration, call all selected models in a row

- Each model is passed as argument the previously created class var, so that it Modifies parameter in that class which will be later passed to the next model;

- At the end of every iteration, parameters saved in the *save_list* or *plot_list* are read from the var class and their value is stored in the appropriate output list.

Command phase:

- At the end of every iteration, parameter selected to be sent to the device as read from the var class and formatted into the string pattern described in the previous chapter.

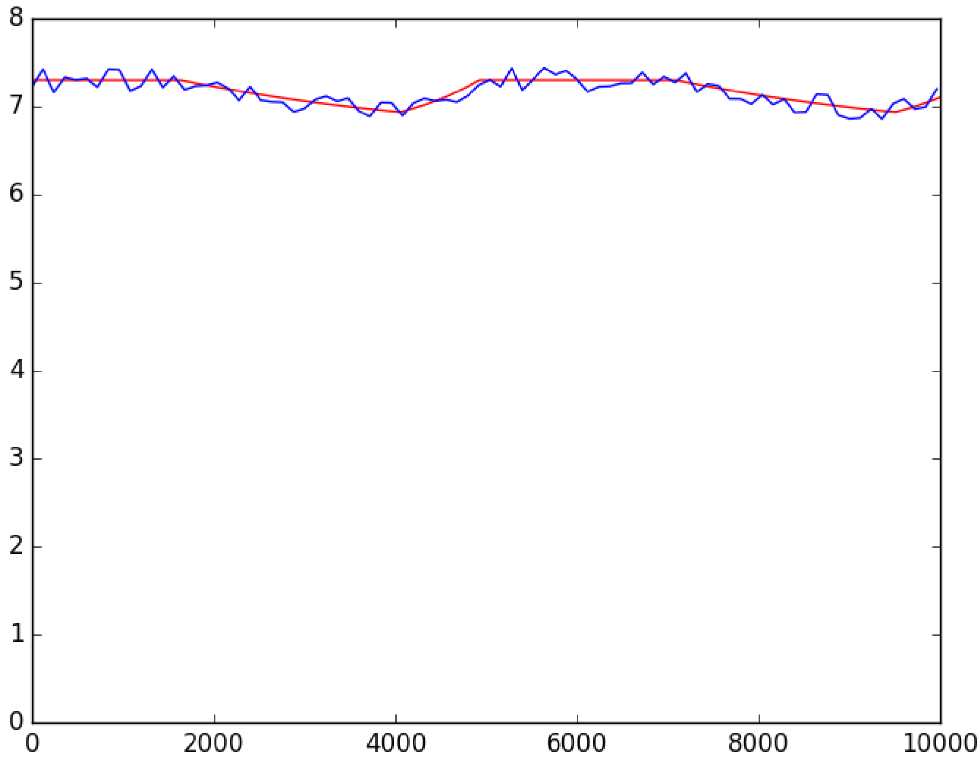
- This command string is then sent to the device input port; voltages and currents on the output channels of the device are instantly updated.

Sleep phase:

- Since a Hardware-in-the-loop simulation forces real-time mode, the *sleep()* function is now called for one second.

4.4.3) HIL results and comparison

A simulation was carried out for 10800 seconds (3 hours), enough to simulate of three complete Earth orbits; results are in good accord with the expected behavior computed by the algorithmical simulation.



Picture 4-25: actual voltage results

Oscillations naturally occur and are mainly due to:

- Old equipment, on which other tests had already been run and that probably was already damaged.
- Inability of the generator to output exactly voltage and current as commanded, due to the resistive nature of the load.
- Approximation in the model used to describe the battery state of charge.

Conclusion and further development

StarSim v.02 can well satisfy its intended requirements of simplicity, flexibility and performance; although minor bugs are surely still present (for instance in the plotting window), the software is ready to be taken towards further development.
This can include, in supposed chronological order:

1. Enhancing the present library by adding different models and algorithms used in the AIL simulation; this work will require the assistance of the experts in each particular field, and great emphasis shall be posed onto standardization of the AIL models source code, following the example of those already present in the folder *Models* in the StarSim directory.

2. Build the software on a UNIX core with real-time capabilities, to enhance significance of the SIL and HIL simulations.

3. Have the software running on a network server, to instantly share result among all team members, allow for multi-user simulation or run-time user intervention during the simulation (e.g. to simulate radio commands).

Biblioghrapy

I. Jens Eichhoff, *Simulating Spacecraft System*, Springer 2009

II. W. M. Saslow, *How batteries discharge: A simple model*, Department of Physics, A&M University, College Station, Texas

III. David Sanz Morales*, Maximum Power Point Tracking Algorithms for Photovoltaic Applications*, Faculty of Electronics, Communications and Automation, Aalto University

IV. Guido Colasurdo, *Astrodynamics*, SpacE Exploration and Development Systems, Politecnico di Torino 2006

V. Sabrina Corpino, Fabrizio Stesina*, Verification of a CubeSat via Hardware-in-the-Loop Simulation*, IEEE TRANSACTIONS ON AEROSPACE AND ELECTRONIC SYSTEMS VOL. 50, NO. 4 OCTOBER 2014,

VI. Sabrina Corpino, Fabrizio Stesina, *HARDWARE IN THE LOOP TEST CAMPAIGN FOR E-ST@R CUBESAT*, The 4S Symposium, Portoroz 2012

VII. Jeffrey Aristoff, Aubrey B. Poore, *Implicit Runge-Kutta Methods for Orbit Propagation*, American Institute of Aeronautics and Astronautics 2015

VIII. *ECSS-E-ST-10-04C*, ECSS Secretariat ESA-ESTEC, 2008.

IX. Stefan Maus, Susan Macmillan, Susan McLean, Brian Hamilton, Manoj Nair, Alan Thomson, and Craig Rollins (NGA). “*The US/UK World Magnetic Model for 2010-2015*".

X. Gilbert Fanchini, *Attitude Determination and Control System Design and Development for Nano Satellite Applications: Test and Verification Via Hardware In the Loop Simulation*, Politecnico di Torino 2010

XI. Fabio Nichele, *Electrical Power System design and development for nanosatellite applications*, Politecnico di Torino 2010

Appendix A

Models code

A.1) Orbit propagator

```
# spacecraft_motion
"""
CONFIG
major_semiaxis [km]
eccentricity [-]
inclination [deg]
longitude_of_the_ascending_node [deg]
argument_of_periapsis [deg]
true_anomaly [deg]
output
x
y
z
r
Vx
Vy
Vz
END_CONFIG

INFO
Orbit Propagator
END_INFO
"""

def Orbit_propagator(var, np):
    h = var.t_step
    mu = 398600
    earth_radius = 6373

    if var.IsFirstIteration == True:
        a = var.major_semiaxis + earth_radius
        e = var.eccentricity
        i = np.deg2rad(var.inclination)
        W = np.deg2rad(var.longitude_of_the_ascending_node)
        o = np.deg2rad(var.argument_of_periapsis)
        n = var.true_anomaly
```

```
        p = a*(1-e**2)
        r_perifocal = p / (1 + e*np.cos(n))
        Vp = np.sqrt(mu/p)*(-np.sin(n))
        Vq = np.sqrt(mu/p)*(e + np.cos(n))

        r11 = np.cos(W)*np.cos(o) -
np.sin(W)*np.sin(o)*np.cos(i)
        r12 = - np.cos(W)*np.sin(o) -
np.sin(W)*np.cos(o)*np.cos(i)
        r13 = np.sin(W)*np.sin(i)
        r21 = np.sin(W)*np.cos(o) +
np.cos(W)*np.sin(o)*np.cos(i)
        r22 = - np.sin(W)*np.sin(o) +
np.cos(W)*np.cos(o)*np.cos(i)
        r23 = -np.cos(W)*np.sin(i)
        r31 = np.sin(o)*np.sin(i)
        r32 = np.cos(o)*np.sin(i)
        r33 = np.cos(i)

        R = np.matrix(([r11, r12, r13], [r21, r22, r23],
[r31, r32, r33]), dtype=float)
        pos_perifocal = np.array([[r_perifocal*np.cos(n),
r_perifocal*np.sin(n), 0]])
        V_perifocal = np.array([[Vp, Vq, 0]])
        pos = R * pos_perifocal.T
        v = R * V_perifocal.T
    else:
        pos = np.array([var.x, var.y, var.z])
        v = np.array([var.Vx, var.Vy, var.Vz])


    def g(pos):
        r = np.sqrt(pos[0]**2 + pos[1]**2 + pos[2]**2)
        r3 = np.power(r, 3)
        value = (-(mu*pos)/r3)
        return value

    var.r = np.sqrt(pos[0] ** 2 + pos[1] ** 2 + pos[2] ** 2)

    k0 = h*v
    l0 = h*g(pos)

    k1 = h*(v+0.5*l0)
    l1 = h*g(pos+0.5*k0)
```

```
    k2 = h*(v+0.5*l1)
    l2 = h*g(pos+0.5*k1)

    k3 = h*(v+0.5*l2)
    l3 = h*g(pos+0.5*k2)

    afx = (k0+2*k1+2*k2+k3)/6
    afv = (l0+2*l1+2*l2+l3)/6

    pos = pos + afx
    v = v + afv

    if var.IsFirstIteration == True:
        var.x = pos[0,0]
        var.y = pos[1,0]
        var.z = pos[2,0]
        var.Vx = v[0,0]
        var.Vy = v[1,0]
        var.Vz = v[2,0]
    else:
        var.x = pos[0]
        var.y = pos[1]
        var.z = pos[2]
        var.Vx = v[0]
        var.Vy = v[1]
        var.Vz = v[2]
```

A.2) Sun Motion

```
# environment
"""
CONFIG
true_anomaly [deg]
output
earth_x
earth_y
earth_z
earth_r
earth_Vx
earth_Vy
earth_Vz
END_CONFIG
"""
```

```
def Sun_motion(var, np):
    h = var.t_step
    mu = 1.327*(10**11)
    sun_radius = 695508

    #  Earth orbit parameters

    major_semiaxis = 149597870.7
    eccentricity = 0.0167
    inclination = 7.155
    longitude_of_the_ascending_node = 174.9
    argument_of_periapsis = 288.1

    if var.IsFirstIteration == True:
        a = major_semiaxis + sun_radius
        e = eccentricity
        i = inclination
        W = longitude_of_the_ascending_node
        o = argument_of_periapsis
        n = var.true_anomaly
        p = a*(1-e**2)
        r_perifocal = p / (1 + e*np.cos(n))
        Vp = np.sqrt(mu/p)*(-np.sin(n))
        Vq = np.sqrt(mu/p)*(e + np.cos(n))

        r11 = np.cos(W)*np.cos(o) -
np.sin(W)*np.sin(o)*np.cos(i)
        r12 = - np.cos(W)*np.sin(o) -
np.sin(W)*np.cos(o)*np.cos(i)
        r13 = np.sin(W)*np.sin(i)
        r21 = np.sin(W)*np.cos(o) +
np.cos(W)*np.sin(o)*np.cos(i)
        r22 = - np.sin(W)*np.sin(o) +
np.cos(W)*np.cos(o)*np.cos(i)
        r23 = -np.cos(W)*np.sin(i)
        r31 = np.sin(o)*np.sin(i)
        r32 = np.cos(o)*np.sin(i)
        r33 = np.cos(i)

        R = np.matrix(([r11, r12, r13], [r21, r22, r23],
[r31, r32, r33]), dtype=float)
        pos_perifocal = np.array([[r_perifocal*np.cos(n),
r_perifocal*np.sin(n), 0]])
        V_perifocal = np.array([[Vp, Vq, 0]])
```

```
        pos = R * pos_perifocal.T
        v = R * V_perifocal.T
    else:
        pos = np.array([var.earth_x, var.earth_y,
var.earth_z])
        v = np.array([var.earth_Vx, var.earth_Vy,
var.earth_Vz])


    def g(pos):
        r = np.sqrt(pos[0]**2 + pos[1]**2 + pos[2]**2)
        r3 = np.power(r, 3)
        value = (-(mu*pos)/r3)
        return value

    var.earth_r = np.sqrt(pos[0] ** 2 + pos[1] ** 2 + pos[2]
** 2)

    k0 = h*v
    l0 = h*g(pos)

    k1 = h*(v+0.5*l0)
    l1 = h*g(pos+0.5*k0)

    k2 = h*(v+0.5*l1)
    l2 = h*g(pos+0.5*k1)

    k3 = h*(v+0.5*l2)
    l3 = h*g(pos+0.5*k2)

    afx = (k0+2*k1+2*k2+k3)/6
    afv = (l0+2*l1+2*l2+l3)/6

    pos = pos + afx
    v = v + afv

    if var.IsFirstIteration == True:
        var.earth_x = pos[0, 0]
        var.earth_y = pos[1, 0]
        var.earth_z = pos[2, 0]
        var.earth_Vx = v[0, 0]
        var.earth_Vy = v[1, 0]
        var.earth_Vz = v[2, 0]
        var.earth_r = np.sqrt(pos[0,0] ** 2 + pos[1,0] ** 2 +
pos[2,0] ** 2)
```

```
    else:
        var.earth_x = pos[0]
        var.earth_y = pos[1]
        var.earth_z = pos[2]
        var.earth_Vx = v[0]
        var.earth_Vy = v[1]
        var.earth_Vz = v[2]
        var.earth_r = np.sqrt(pos[0] ** 2 + pos[1] ** 2 +
pos[2] ** 2)
```

A.3) Sun Vector

```
# environment
"""
CONFIG
earth_x [km]
earth_y [km]
earth_z [km]
x [km]
y [km]
z [km]
output
sun_vector_x
sun_vector_y
sun_vector_z
distance
sun_vector_magnitude
sun_vector_direction_i
sun_vector_direction_j
sun_vector_direction_k
END_CONFIG
"""

def Sun_vector(var, np):
    var.sun_vector_x = var.earth_x + var.x
    var.sun_vector_y = var.earth_y + var.y
    var.sun_vector_z = var.earth_z + var.z
    #
    #  eclipse detector
    #

    eclipse = False
    earth_radius = 6371
```

```
    if np.sign(var.x) == np.sign(var.earth_x) and (var.z <
earth_radius) and (var.y < earth_radius):
        eclipse = True
    if eclipse == True:
        var.sun_vector_x = 0
        var.sun_vector_y = 0
        var.sun_vector_z = 0
        #
        var.sun_vector_direction_i = 0
        var.sun_vector_direction_j = 0
        var.sun_vector_direction_k = 0
        var.sun_vector_magnitude = 0
        #
    else:
        var.sun_vector_direction_i = -1
        var.sun_vector_direction_j = 0
        var.sun_vector_direction_k = 0
        '''
        var.sun_vector_magnitude =
np.sqrt((var.sun_vector_x**2)+
(var.sun_vector_y**2)+(var.sun_vector_z**2))
        var.sun_vector_direction_i = var.sun_vector_x /
var.sun_vector_magnitude
        var.sun_vector_direction_j = var.sun_vector_y /
var.sun_vector_magnitude
        var.sun_vector_direction_k = var.sun_vector_z /
var.sun_vector_magnitude
```

A.4) Magnetic field (dipole)

```
# environment
'''
CONFIG
lat [deg]
long [deg]
alt [deg]
output
Bx
By
Bz
B_mod
END_CONFIG
'''
```

```
def magnetic_field_dipole(var, np):
    R = 6371
    r = (var.alt/1000) + R
    m = 7.94e22
    mu0 = np.pi * 4e-7
    theta = np.pi/2 - np.deg2rad(var.lat)

    Br = (2 * mu0 * m * np.cos(theta)) / (4 * np.pi * r ** 3)
    Bt = (mu0 * m * np.sin(theta)) / (4 * np.pi * r ** 3)

    if var.long > -np.pi/2 and var.long < np.pi/2:
        Brx = Br * np.sin(theta)
        Brz = Br * np.sin(theta)
        Btx = Bt * np.sin(np.deg2rad(var.lat))
        Btz = -Bt * np.cos(np.deg2rad(var.lat))
    else:
        Brx = -Br * np.sin(theta)
        Brz = Br * np.sin(theta)
        Btx = -Bt * np.sin(np.deg2rad(var.lat))
        Btz = -Bt * np.cos(np.deg2rad(var.lat))

    var.Bx = Brx + Btx
    var.Bz = Brz + Btz
    var.By = 0
    var.B_mod = np.sqrt(var.Bx**2+var.Bz**2)
```

A.5) Constant resistive load

```
# actuators
"""
CONFIG
load_dissipated_power [mW]
efficiency [%]
output
load_voltage
load_current
END_CONFIG

INFO
dummy resistive load for eps testing purposes
END_INFO
"""
'''
def constant_resistance_load(var, np):
```

```
    from Sources import config
    if "power_bus" in config.model_list:
        var.load_current = var.I_out_total
    if "battery" in config.model_list:
        var.load_current = var.battery_I_out
    var.load_voltage = var.load_current * var.load_resistance
    var.load_dissipated_power = var.load_current *
var.load_voltage
'''
def constant_resistance_load(var, np):
    if var.IsFirstIteration == True:
        setattr(var, "target_power",
var.load_dissipated_power)
    var.load_current = var.battery_I_out
    var.load_voltage = var.actual_voltage
    var.load_dissipated_power = var.load_current *
var.load_voltage
```

A.6) Solar panel (full)

```
# power_sources
"""
CONFIG
sun_vector_direction_i [-]
sun_vector_direction_j [-]
sun_vector_direction_k [-]
face_position [string]
short_circuit_current [A]
open_circuit_voltage [V]
number_cells_in_series [-]
load_resistance [Ohm]
output
solar_panel_I_out
solar_panel_V_out
solar_irradiation
END_CONFIG

INFO
Rectangular solar panel; normal vectors are in own body
reference frame.
END_INFO
"""

def solar_panel_full(var, np):
    from Sources import config
```

```
    if var.IsFirstIteration is True:
        if not hasattr(var, 'face_normals'):
            normals_dict = {'x': np.array([1, 0, 0]), 'y':
np.array([0, 1, 0]), 'z': np.array([0, 0, 1]),
                            '-x': np.array([-1, 0, 0]), '-y':
np.array([0, -1, 0]), '-z': np.array([0, 0, -1])}
            setattr(var, "face_normals", normals_dict)

    normal_vector = var.face_normals[var.face_position]
    sun_vector = np.array([var.sun_vector_direction_i,
var.sun_vector_direction_j, var.sun_vector_direction_k])
    scalar_product = np.dot(normal_vector, sun_vector)
    theta = (np.arccos(scalar_product))
    var.solar_irradiation = float(1367) * (- np.cos(theta))
    I_sp = 0
    #
    # case 1) eclipse
    #
    if var.solar_irradiation < 0:
        var.solar_irradiation = 0
        I_sp = 0
        V_sp = 0

    #
    # case 2) no MPPT present, voltage is determined by load
    #
    elif "MPPT" not in config.model_list:
        I_sp = 0
        load = var.load_resistance
        attempted_V = 1
        converged = False
        already_attemped_V = []
        while not converged:
            resulting_I = find_I(attempted_V, var, np)
            already_attemped_V.append(attempted_V)
            expected_I = attempted_V / load
            error = abs(resulting_I - expected_I)
            if error < 0.01:
                converged = True
                V_sp = attempted_V
                I_sp = resulting_I
            else:
                attempted_V = resulting_I * load
                if attempted_V in already_attemped_V:
                    converged = True
```

```
                I_sp = 0
    #
    # case 3) MPPT present, maximum current is generated
    #
    else:
        I = []
        P = []
        V = []
        start = var.open_circuit_voltage/100*80
        for v in np.arange(start, var.open_circuit_voltage,
0.01):
            i = find_I(v, var, np)
            V.append(v)
            I.append(i)
            P.append(i*v)
        P_max_index = P.index(max(P))
        I_sp = I[P_max_index]
        V_sp = V[P_max_index]

    var.solar_panel_I_out = I_sp
    var.solar_panel_V_out = V_sp

def find_I(V, var, np):
    Isc = var.short_circuit_current
    Ki = 0.002  # single cell short circuit current
    Voc = var.open_circuit_voltage
    q = 1.6e-19  # electron charge
    Np = float(1)
    n = 1.2  # diode ideality factor
    k = 1.3805e-23  # Boltzman constant
    Tr = 298.15  # nominal temperature
    Eg0 = 1.1  # energy band gap [eV]
    Ns = var.number_cells_in_series
    Rs = 0.001  # series resistance
    Rsh = float(1000)  # shunt resistance

    Ir = var.solar_irradiation
    T = float(280)

    Iph = (Isc + (Ki * (T - 298))) * (Ir / 1000)
    Irs = Isc / (np.exp(q * Voc / (Ns * k * n * T)) - 1)
    I0 = Irs * ((T / Tr) ** 3) * np.exp((q * Eg0 / (n * k)) *
(1 / T - 1 / Tr))
    Vt = (k * T) / q
    Ish = (V * (Np / Ns) + Iph * Rs) / Rsh
```

```
    I_out = Np * Iph - Np * I0 * np.exp(((V / Ns) + (Rs /
Np)) / (n * Vt) - 1) - Ish
    if I_out < 0:
        I_out = 0
    return I_out
```

A.7) Battery

```
# power_sources
"""
CONFIG
nominal_voltage [V]
capacity [mAh]
depth_of_discharge [%]
max_discharge_rate [C]
output
battery_I_out
actual_voltage
END_CONFIG
"""

def battery(var, np):
    from Sources import config
    #
    if var.IsFirstIteration == True:
        setattr(var, "capacity_max", var.capacity)
        var.capacity = var.capacity * (100 -
var.depth_of_discharge) / 100
    #
    dod = (var.capacity_max - var.capacity) /
var.capacity_max
    actual_voltage = var.nominal_voltage*(-8.281 * dod ** 7 +
23.5743 * dod ** 6 - 30 * dod ** 5 + 23.7053 * dod ** 4
                           - 12.5877 * dod ** 3 + 4.1325 *
dod ** 2 - 0.8658 * dod + 1)
    if 'constant_resistance_load' in config.model_list:
        I_out_max = var.capacity_max/float(1000) *
var.max_discharge_rate
        var.battery_I_out = (var.target_power/float(1000)) /
actual_voltage
        if var.battery_I_out > I_out_max:
            var.battery_I_out = I_out_max
    #
    # discharge
```

```
    #
    var.capacity = var.capacity - var.t_step *
var.battery_I_out*(float(1000) / 3600)
    if var.capacity <= 0:
        var.capacity = 0
        var.battery_I_out = 0
        actual_voltage = 0
    #
    # charge
    #
    if "power_bus" in config.model_list:
        var.capacity = var.capacity + var.t_step *
var.I_out_total * (float(1000) / 3600)
        if var.capacity >= var.capacity_max:
            var.capacity = var.capacity_max

    var.depth_of_discharge = dod * 100
    var.actual_voltage = actual_voltage
```

A.9) Thermal fluxes

```
'''
CONFIG
Temperature [K]
face_area [m^2]
output
T_x
T_y
T_z
T_neg_x
T_neg_y
T_neg_z
q_x
END_CONFIG
'''

def thermal_fluxes(var, np):
     from Sources import config
     alpha = 0.92
     epsilon = 0.85
     area = var.face_area
     dn = 100
     Sc = 1367 / (1+0.33412*np.cos(2*np.pi*(dn-3)/365))
     IR = 239
     Albedo = 0.29*Sc
```

```
    sigma = 5.6704e-8
    T_space = 4
    T = var.Temperature
    rho = 5000
    V = area*0.1
    Cp = 1500
    #
    if var.IsFirstIteration == True:
         if not hasattr(var, "face_normals"):
              normals_dict = {'x': np.array([1, 0, 0]), 'y':
np.array([0, 1, 0]), 'z': np.array([0, 0, 1]),
                                  '-x': np.array([-1, 0,
0]), '-y': np.array([0, -1, 0]), '-z': np.array([0, 0, -1])}
              setattr(var, "face_normals", normals_dict)
              temp_dict = {'x': T, 'y': T, 'z': T, '-x': T,
'-y': T, '-z': T }
              setattr(var, 'temp_dict', temp_dict)

    dissipated_power = 0
    if "dummy_load" in config.repeated_models_dict:
         for i in range(0,
config.repeated_models_dict["dummy_load"]):
              string = "(1 -
var.efficiency_"+str(i)+")*var.load_dissipated_power_"+str(i)
              dissipated_power += eval(string)
    elif "dummy_load" in config.model_list:
         dissipated_power = (1-
var.efficiency/100)*var.load_dissipated_power
    #
    faces = ["x", "y", "z", "-x", "-y", "-z"]
    earth_versor = np.array([var.x, var.y, var.z]) / var.r
    if var.sun_vector_magnitude != 0:
         sun_vector = np.array([var.sun_vector_direction_i,

var.sun_vector_direction_j, var.sun_vector_direction_k])
         sun_versor = sun_vector / var.sun_vector_magnitude
    else:
         sun_versor = np.array([0, 0, 0])
    for face in faces:
         normal = var.face_normals[face]
         cos_sun_angle = np.dot(sun_versor, normal)
         cos_earth_angle = np.dot(earth_versor, normal)
         #
         Sc = Sc *cos_sun_angle
         IR = IR * cos_earth_angle
```

```
            Albedo = Albedo * cos_earth_angle
            q_in_ext = alpha*(Sc+IR+Albedo)*area
            q_in = q_in_ext + dissipated_power/6
            q_out = sigma*epsilon*(var.temp_dict[face]**4 -
T_space**4)*area
            q = q_in - q_out
            var.temp_dict[face] += q/(rho*Cp*V)*var.t_step
        #
        var.T_x = np.linalg.norm(var.temp_dict["x"])
        var.T_y = np.linalg.norm(var.temp_dict["y"])
        var.T_z = np.linalg.norm(var.temp_dict["z"])
        var.T_neg_x = np.linalg.norm(var.temp_dict["-x"])
        var.T_neg_y = np.linalg.norm(var.temp_dict["-y"])
        var.T_neg_z = np.linalg.norm(var.temp_dict["-z"])
```

Appendix B

Simulator code

B.1) Model parser

```
def parse_lines(model_list):
    flag = 0
    flag_out = 0
    vardict = {}
    out_dict = {}
    count = 0
    temp_varlist = []
    temp_outlist = []

    # Opens every selected models file and reads the CONFIG
section
    for i in range(0, len(model_list)):
        with open("Models" + '/' + model_list[i] + ".py",
"r") as ins:
            for line in ins:
                if line.find("CONFIG") != -1:
                    flag = 1
                if (flag == 1) and (line.find("CONFIG") == -
1) and (line.find("output") == -1):
                    # So here we are in the CONFIG input
section; here are listed the parameters that
                    # the users has to fill in order to have
the model running
                    var = str(line.split()[0]).strip()
                    if (var not in temp_varlist) and (var not
in temp_outlist):
                        vardict[count] = [model_list[i],
[var, str(line.split()[1])]]
                        count = count + 1
                        temp_varlist.append(var)
                if line.find("output") != -1:
                    flag = 0
                    flag_out = 1
                if (flag_out == 1) and
(line.find("END_CONFIG") == -1) and (line.find("output") == -
1):
                    out_dict[count] = [model_list[i],
str(line).strip()]
```

```
                    temp_outlist.append(str(line).strip())
                    count = count + 1
                if line.find("END_CONFIG") != -1:
                    flag = 0
                    flag_out = 0
        ins.close()
        config.out_dict = out_dict
        #
        for item in vardict.values():
            config.param_dict[item[1][0]] = item[0]
        for item in out_dict.values():
            if item not in config.param_dict.keys():
                config.param_dict[item[1]] = item[0]
                #
    return [vardict, out_dict]


def configure_repeated_models():
    #
    #  creating copy files for repeated models
    #
    for entry in config.repeated_models_dict.keys():
            with open("Models/" + str(entry) + ".py", "r") as
f_original:
                var_list = []
                flag = 0
                for line in f_original:
                    if line.find("CONFIG") != -1:
                        flag = 1
                    if line.find("END_CONFIG") != -1:
                        flag = 0
                    if flag == 1:
                        var_list.append(line.split()[0])
                var_list.remove("CONFIG")
                var_list.remove("output")
                var_list = remove_constants(var_list)  # e.g
the sun vector is constant for all repeated models,
                # it must not be given the incremental number
                f_original.close()
                #
                for i in range(1,
config.repeated_models_dict[entry]+1):
                    with open("Models/" + str(entry) + ".py",
"r") as f_original:
```

```
                        with open("Models/" + str(entry) +
"_" + str(i) + ".py", "w") as f_copied:
                            for line in f_original:
                                if line.find("def") != -1:
                                    line =
line.replace(entry, entry+"_"+str(i))
                                for item in var_list:
                                    if
line.find(item.split()[0]) != -1:
                                        new_item =
item+"_"+str(i)
                                        line =
line.replace(item, new_item)
                                f_copied.write(line)
                            f_copied.close()
                        f_original.close()


def remove_constants(var_list):
    constant_list = ["sun_vector_x", "sun_vector_y",
"sun_vector_z", "sun_vector_magnitude",
                       "sun_vector_direction_i",
"sun_vector_direction_j", "sun_vector_direction_k"]
    items_to_remove = []
    for item in var_list:
        if item in constant_list:
            items_to_remove.append(item)
    #
    for item in items_to_remove:
        var_list.remove(item)
    return var_list
```

B.2) Parameter parser

```
def parse_lines(model_list):
    flag = 0
    flag_out = 0
    vardict = {}
    out_dict = {}
    count = 0
    temp_varlist = []
    temp_outlist = []
```

```
    # Opens every selected models file and reads the CONFIG
section
    for i in range(0, len(model_list)):
        with open("Models" + '/' + model_list[i] + ".py",
"r") as ins:
            for line in ins:
                if line.find("CONFIG") != -1:
                    flag = 1
                if (flag == 1) and (line.find("CONFIG") == -
1) and (line.find("output") == -1):
                    # So here we are in the CONFIG input
section; here are listed the parameters that
                    # the users has to fill in order to have
the model running
                    var = str(line.split()[0]).strip()
                    if (var not in temp_varlist) and (var not
in temp_outlist):
                        vardict[count] = [model_list[i],
[var, str(line.split()[1])]]
                        count = count + 1
                        temp_varlist.append(var)
                if line.find("output") != -1:
                    flag = 0
                    flag_out = 1
                if (flag_out == 1) and
(line.find("END_CONFIG") == -1) and (line.find("output") == -
1):
                    out_dict[count] = [model_list[i],
str(line).strip()]
                    temp_outlist.append(str(line).strip())
                    count = count + 1
                if line.find("END_CONFIG") != -1:
                    flag = 0
                    flag_out = 0
        ins.close()
        config.out_dict = out_dict
        #
        for item in vardict.values():
            config.param_dict[item[1][0]] = item[0]
        for item in out_dict.values():
            if item not in config.param_dict.keys():
                config.param_dict[item[1]] = item[0]
                #
    return [vardict, out_dict]
```

```
def configure_repeated_models():
    #
    #  creating copy files for repeated models
    #
    for entry in config.repeated_models_dict.keys():
            with open("Models/" + str(entry) + ".py", "r") as
f_original:
                var_list = []
                flag = 0
                for line in f_original:
                    if line.find("CONFIG") != -1:
                        flag = 1
                    if line.find("END_CONFIG") != -1:
                        flag = 0
                    if flag == 1:
                        var_list.append(line.split()[0])
                var_list.remove("CONFIG")
                var_list.remove("output")
                var_list = remove_constants(var_list)  # e.g
the sun vector is constant for all repeated models,
                # it must not be given the incremental number
                f_original.close()
                #
                for i in range(1,
config.repeated_models_dict[entry]+1):
                    with open("Models/" + str(entry) + ".py",
"r") as f_original:
                        with open("Models/" + str(entry) +
"_" + str(i) + ".py", "w") as f_copied:
                            for line in f_original:
                                if line.find("def") != -1:
                                    line =
line.replace(entry, entry+"_"+str(i))
                                for item in var_list:
                                    if
line.find(item.split()[0]) != -1:
                                        new_item =
item+"_"+str(i)
                                        line =
line.replace(item, new_item)
                                f_copied.write(line)
                            f_copied.close()
                        f_original.close()
```

```
def remove_constants(var_list):
    constant_list = ["sun_vector_x", "sun_vector_y",
"sun_vector_z", "sun_vector_magnitude",
                       "sun_vector_direction_i",
"sun_vector_direction_j", "sun_vector_direction_k"]
    items_to_remove = []
    for item in var_list:
        if item in constant_list:
            items_to_remove.append(item)
    #
    for item in items_to_remove:
        var_list.remove(item)
    return var_list
```

B.3) Simulation Setup

```
import config
import execute
from HIL import hil_connector


def create_process():
        #
        # 'Prints "import" statements in process.py
        temp_list = []
        activate_condition_list = []
        deactivate_condition_list = []
        act_cont = -1 # used to count the conditions of the
"ACTIVATE" type
        deact_cont = -1
        with open("Sources\Process.py", "w") as process:
            for i in range(0, len(config.model_list)):
                if config.model_list[i] not in temp_list:
                    process.write("from Models import
"+str(config.model_list[i])+"\n")
                    temp_list.append(config.model_list[i])
        # 'Print the variables class in Process.py
            process.write("import matplotlib.pylab as
plt\nplt.switch_backend('WXagg')\nimport time\nnp =
__import__('numpy', globals(), locals())\n")
            process.write("spr = __import__('subprocess')\n")
            if config.hil == 1:
```

```
                process.write("from HIL import
hil_connector\n")   # <----------------------------------------
-----------------------HIL
            process.write("import config\n\nclass
Variables():\n\tdef __init__(self):\n\t\tself.t =
"+str(config.svw.time_settings[0])+"\n")
            process.write("\t\tself.t_step =
"+str(config.svw.time_settings[2])+"\n\t\tself.IsFirstIterati
on = True\n")
            for i in range(0, len(config.var_value_list)):

process.write("\t\tself."+str(config.var_value_list[i][1])+"
= "+config.var_value_list[i][2]+"\n")
            for i in range(0, len(config.out_dict.values())):

process.write("\t\tself."+str(config.out_dict.values()[i][1])
+" = 0\n")
            n_max = int(((config.svw.time_settings[1])-
(config.svw.time_settings[0]))/(config.svw.time_settings[2]))
            process.write("\ndef Process():\n\tvar =
Variables()\n\toutput_dict = {}\n\ttime_vector = []\n")
            #
            # The "activate" and "deactivate" condition are
initialized and the list is created
            #

            for i in range(0, len(config.condition_list)):
                if config.condition_list[i].split(' ', 1)[0]
== "ACTIV":
                    act_cont = act_cont + 1

activate_condition_list.append(str(config.condition_list[i].s
plit(' ', 1)[1]))

process.write("\tactivate_condition_flag_" + str(act_cont) +
" = 0\n")
            for i in range(0, len(config.condition_list)):
                if config.condition_list[i].split(' ', 1)[0]
== "DEACTIV":
                    deact_cont = deact_cont + 1

deactivate_condition_list.append(str(config.condition_list[i]
.split(' ', 1)[1]))
```

```
process.write("\tdeactivate_condition_flag_" +
str(deact_cont) + " = 0\n")

            #
            process.write("\tfor param in
vars(var):\n\t\toutput_dict[param] = []\n")
            if config.hil == 1:
                port_address = str(config.port_address)
                baudrate = str(config.baud_rate)
                process.write("\tport_name =
hil_connector.port_init('" + port_address + "', 100, " +
baudrate + ")\n")
            process.write("\tn = 1\n\twhile var.t <=
"+str(config.svw.time_settings[1])+":\n\t\t\n")  # start_time
= time.clock()\n")
            #
            # Checking the "activate" and "deactivate"
conditions
            #
            for i in range(0, len(activate_condition_list)):
                process.write("\t\tif " +
activate_condition_list[i] +
":\n\t\t\tactivate_condition_flag_" + str(i) + " = 1\n")
            for i in range(0,
len(deactivate_condition_list)):
                process.write("\t\tif " +
deactivate_condition_list[i] +
":\n\t\t\tdeactivate_condition_flag_" + str(i) + " = 1\n")
        #
        #
            act_cont = -1
            deact_cont = -1
            nest_cont = 0
            extra_tab_counter = 0
            extra_tab = ""
            exit_condition = ""
            for i in range(0, len(config.model_list)):
                #
                #
                if  config.condition_list[i].split(' ', 1)[0]
== "EXEC":
                    process.write(extra_tab + "\t\tif " +
config.condition_list[i].split(' ', 1)[1] + ":\n\t")
```

```
                    process.write((extra_tab + "\t\t" +
config.model_list[i] + "." + config.model_list[i]) + "(var,
np)\n")
                elif config.condition_list[i].split(' ',
1)[0] == "SKIP":
                    process.write(extra_tab + "\t\tif not(" +
config.condition_list[i].split(' ', 1)[1] + "):\n\t")
                    process.write((extra_tab +"\t\t" +
config.model_list[i] + "." + config.model_list[i]) + "(var,
np)\n")
                elif config.condition_list[i].split(' ',
1)[0] == "ACTIV":
                    act_cont = act_cont + 1
                    process.write(extra_tab + "\t\tif
activate_condition_flag_" + str(act_cont) +" == 1:\n\t")
                    process.write((extra_tab + "\t\t" +
config.model_list[i] + "." + config.model_list[i]) + "(var,
np)\n")
                elif config.condition_list[i].split(' ',
1)[0] == "DEACTIV":
                    deact_cont = deact_cont + 1
                    process.write(extra_tab + "\t\tif not
(deactivate_condition_flag_" + str(deact_cont) +" ==
1):\n\t")
                    process.write((extra_tab + "\t\t" +
config.model_list[i] + "." + config.model_list[i]) + "(var,
np)\n")
                    #
                elif config.condition_list[i].split(' ',
1)[0] == "LOOP_START" or config.condition_list[i].split(' ',
1)[0] == "LOOP_START_always":
                    for j in range(i,
len(config.condition_list)):
                        if config.condition_list[j].split('
', 1)[0] == "LOOP_START" or config.condition_list[j].split('
', 1)[0] == "LOOP_START_always":
                            nest_cont = nest_cont + 1
                            print "added one: nest_cont = ",
nest_cont
                        if config.condition_list[j].split('
', 1)[0] == "LOOP_EXIT":
                            nest_cont = nest_cont - 1
                            print "removed one: nest_cont =
", nest_cont
                            if nest_cont == 0:
```

```
                                exit_condition =
config.condition_list[j].split(' ', 1)[1]
                    if config.condition_list[i].split(' ',
1)[0] == "LOOP_START_always":
                        process.write(extra_tab + "\t\tif
True:\n")
                    else:
                        process.write(extra_tab + "\t\tif " +
config.condition_list[i].split(' ', 1)[1] + ":\n")
                    process.write(extra_tab + "\t\t\twhile "
+ exit_condition +":\n")
                    extra_tab_counter = extra_tab_counter + 2
                    extra_tab =
extra_tab_update(extra_tab_counter)
                    process.write(
                        (extra_tab + "\t\t" +
config.model_list[i] + "." + config.model_list[i]) + "(var,
np)\n")

                #
                elif config.condition_list[i].split(' ',
1)[0] == "LOOP_EXIT":
                    process.write(
                        (extra_tab + "\t\t" +
config.model_list[i] + "." + config.model_list[i]) + "(var,
np)\n")
                    extra_tab_counter = extra_tab_counter - 2
                    extra_tab =
extra_tab_update(extra_tab_counter)
                #
                else:
                    process.write(
                        (extra_tab + "\t\t" +
config.model_list[i] + "." + config.model_list[i]) + "(var,
np)\n")

            #   NEW STUFF

process.write("\t\ttime_vector.append(var.t)\n\t\tfor i in
range(0, len(config.var_save_list)):\n")
            process.write("\t\t\tparam =
config.var_save_list[i]\n")

process.write("\t\t\toutput_dict[param].append((vars(var))[pa
ram])\n")
```

```
            process.write("\t\tfor k in range(0,
len(config.var_plot_list)):\n")
            process.write("\t\t\tparam =
config.var_plot_list[k]\n")
            process.write("\t\t\tif param not in
config.var_save_list:\n")

process.write("\t\t\t\toutput_dict[param].append((vars(var))[
param])\n")

            process.write("\t\tvar.t
+="+str(config.svw.time_settings[2])+"\n\t\tn +=1")

process.write("\n\t\tconfig.apb.AIL_bar.SetValue(100*n/" +
str(n_max) + ")" )
            if config.svw.realtime.IsChecked():
                #  process.write("\n\t\telapsed_time =
(time.clock() - start_time)")

process.write("\n\t\ttime.sleep(var.t_step)\n\t\t")   #print
time.clock()")
            process.write("\n\t\tvar.IsFirstIteration =
False")
            if config.hil == 1:
                process.write("\n\t\tcommand_string =
hil_connector.read_values(var, config)")  # <----------- HIL

process.write("\n\t\thil_connector.command_input(port_name,
command_string)")  #  <----------- HIL
            process.write("\n\tconfig.apb.Close()")
            process.write("\n\tconfig.output_dict =
output_dict\n\tconfig.time_vector = time_vector")
            if config.hil == 1:

process.write("\n\thil_connector.port_close(port_name)")
            if len(config.var_plot_list) > 0:
                process.write("\n\n\t# PLOT SECTION\n")
                process.write("\n\tif
len(config.var_plot_list) > 0:")
                colors = ['red', 'blue', 'green', 'orange']
                for i in range(0, len(config.var_plot_list)):
                    process.write("\n\t\tplt.figure()")
                    label = str(config.var_plot_list[i])
                    color = colors[i % len(colors)]
                    #  output_plot_list[" + str(i) + "], "
```

```
process.write("\n\t\tplt.plot(time_vector, output_dict['" +
str(config.var_plot_list[i]) +"'],"  +

"label='" + label + "', color= '" + color + "')")

process.write("\n\t\tplt.grid(True)\n\t\tplt.legend(loc='lowe
r left')\n\t\tplt.axis((0, 10000, 0, 8))\n\t\tplt.show()\n")

            process.close()
            execute.exec_process()



def extra_tab_update(extra_tab_counter):
    extra_tab = ""
    for t in range(0, extra_tab_counter):
        extra_tab = extra_tab + "\t"
    return extra_tab
```

B.4) self-generated *Process* file for case study

```
from Models import Orbit_propagator
from Models import Sun_motion
from Models import Sun_vector
from Models import constant_resistance_load
from Models import solar_panel_full_1
from Models import solar_panel_full_2
from Models import solar_panel_full_3
from Models import MPPT
from Models import power_bus
from Models import battery
import matplotlib.pylab as plt
plt.switch_backend('WXagg')
import time
np = __import__('numpy', globals(), locals())
spr = __import__('subprocess')
from HIL import hil_connector
import config

class Variables():
     def __init__(self):
          self.t = 0.0
          self.t_step = 1.0
```

```
self.IsFirstIteration = True
self.major_semiaxis = 400
self.eccentricity = 0
self.inclination = 96
self.longitude_of_the_ascending_node = 0
self.argument_of_periapsis = 0
self.true_anomaly = 0
self.load_dissipated_power = 1590
self.efficiency = 0.8
self.face_position_1 = "x"
self.face_position_2 = "y"
self.face_position_3 = "z"
self.nominal_voltage = 7.3
self.capacity = 1800
self.depth_of_discharge = 0
self.max_discharge_rate = 1
self.x = 0
self.y = 0
self.z = 0
self.r = 0
self.Vx = 0
self.Vy = 0
self.Vz = 0
self.earth_x = 0
self.earth_y = 0
self.earth_z = 0
self.earth_r = 0
self.earth_Vx = 0
self.earth_Vy = 0
self.earth_Vz = 0
self.sun_vector_x = 0
self.sun_vector_y = 0
self.sun_vector_z = 0
self.distance = 0
self.sun_vector_magnitude = 0
self.sun_vector_direction_i = 0
self.sun_vector_direction_j = 0
self.sun_vector_direction_k = 0
self.load_voltage = 0
self.load_current = 0
self.solar_panel_I_out_1 = 0
self.solar_panel_V_out_1 = 0
self.solar_irradiation_1 = 0
self.solar_panel_I_out_2 = 0
self.solar_panel_V_out_2 = 0
```

```
        self.solar_irradiation_2 = 0
        self.solar_panel_I_out_3 = 0
        self.solar_panel_V_out_3 = 0
        self.solar_irradiation_3 = 0
        self.I_out_total = 0
        self.battery_I_out = 0
        self.actual_voltage = 0

def Process():
    var = Variables()
    output_dict = {}
    time_vector = []
    for param in vars(var):
        output_dict[param] = []
    port_name = hil_connector.port_init('COM3', 100, 9600)
    n = 1
    while var.t <= 10800.0:

        Orbit_propagator.Orbit_propagator(var, np)
        Sun_motion.Sun_motion(var, np)
        Sun_vector.Sun_vector(var, np)

    constant_resistance_load.constant_resistance_load(var,
np)
        solar_panel_full_1.solar_panel_full_1(var, np)
        solar_panel_full_2.solar_panel_full_2(var, np)
        solar_panel_full_3.solar_panel_full_3(var, np)
        MPPT.MPPT(var, np)
        power_bus.power_bus(var, np)
        battery.battery(var, np)
        time_vector.append(var.t)
        for i in range(0, len(config.var_save_list)):
            param = config.var_save_list[i]
            output_dict[param].append((vars(var))[param])
        for k in range(0, len(config.var_plot_list)):
            param = config.var_plot_list[k]
            if param not in config.var_save_list:

    output_dict[param].append((vars(var))[param])
        var.t +=1.0
        n +=1
        config.apb.AIL_bar.SetValue(100*n/10800)
        time.sleep(var.t_step)

        var.IsFirstIteration = False
```

```
            command_string = hil_connector.read_values(var,
config)
            hil_connector.command_input(port_name,
command_string)
      config.apb.Close()
      config.output_dict = output_dict
      config.time_vector = time_vector
      hil_connector.port_close(port_name)

      # PLOT SECTION

      if len(config.var_plot_list) > 0:
            plt.figure()
            plt.plot(time_vector,
output_dict['I_out_total'],label='I_out_total', color= 'red')
            plt.grid(True)
            plt.legend(loc='lower left')
            plt.axis((0, 10000, 0, 8))
            plt.show()

            plt.figure()
            plt.plot(time_vector,
output_dict['capacity'],label='capacity', color= 'blue')
            plt.grid(True)
            plt.legend(loc='lower left')
            plt.axis((0, 10000, 0, 8))
            plt.show()

            plt.figure()
            plt.plot(time_vector,
output_dict['battery_I_out'],label='battery_I_out', color=
'green')
            plt.grid(True)
            plt.legend(loc='lower left')
            plt.axis((0, 10000, 0, 8))
            plt.show()

            plt.figure()
            plt.plot(time_vector,
output_dict['actual_voltage'],label='actual_voltage', color=
'orange')
            plt.grid(True)
            plt.legend(loc='lower left')
            plt.axis((0, 10000, 0, 8))
            plt.show()
```