17th International Conference on Space Operations, Dubai, United Arab Emirates, 6 - 10 March 2023.

SpaceOps-2023, ID # 655

DEVELOPMENT AND USAGE OF THE GODOT ASTRODYNAMICS SOFTWARE AT TELESPAZIO GERMANY

Francesco Castellini[(1)], Stephan Kranz[(2)], Steffen Weber[(3)], Bernard Godard[(4)], Ruaraidh Mackenzie[(5)], Paul Steele[(6)], Jan Siminski[(7)]

[(1)] *Telespazio GmbH, located at ESA/ESOC, Robert-Bosch-Str. 5, Darmstadt, 64293, Germany, francesco.castellini@telespazio.de*
[(2)] *Telespazio GmbH, Europaplatz 5, Darmstadt, 64293, Germany, stephan.kranz@telespazio.de*
[(3)] *Telespazio GmbH, Europaplatz 5, Darmstadt, 64293, Germany, steffen.weber@telespazio.de*
[(4)] *Telespazio GmbH, located at ESA/ESOC, Robert-Bosch-Str. 5, Darmstadt, 64293, Germany, bernard.godard@telespazio.de*
[(5)] *ESA/ESOC, Robert-Bosch-Str. 5, Darmstadt, 64293, Germany, ruaraidh.mackenzie@esa.int*
[(6)] *ESA/ESOC, Robert-Bosch-Str. 5, Darmstadt, 64293, Germany, paul.steele@esa.int*
[(7)] *ESA/ESOC, Robert-Bosch-Str. 5, Darmstadt, 64293, Germany, jan.siminski@esa.int*

Keywords: *Orbit Determination, Deep Space Missions, Flight Dynamics Operations, GODOT.*

1. Introduction

For organisational and historical reasons, astrodynamics tasks within the European Space Agency (ESA) and European industry overall have been performed in the past decades with a variety of software libraries and tools. In particular, orbit determination and trajectory optimization for Flight Dynamics operations at the European Space Operations Centre (ESOC) have been performed in the past 20 years with software originally developed in the 90s, the NAvigation Package for Earth Orbiting Satellites (NAPEOS, [1]) for Earth orbiters and the Advanced Modular Facility for Interplanetary Navigation (AMFIN, [2]) for deep space missions. In parallel, other players such as ESOC's mission analysis, navigation and space debris offices have relied on separate in-house tools, while the European industry has developed different solutions.

In order to modernise the software base, improving its usability, extensibility and maintainability, and to homogenize the astrodynamics tasks across different sections, ESA/ESOC started a large development effort in 2016 to design and implement a common successor for all these software. Such effort has been conceived from the start with an ambitiously wide scope, aiming at providing a modular, easily extensible, common library to form the core of all orbit related tasks, primarily within the ESA/ESOC Flight Dynamics (FD) division but also for a larger community, including other ESA offices and the overall European industry and academia. The result of this effort is GODOT, the Generic Orbit Determination and Optimisation Toolkit ([3], [4]), which is available under ESA Community License on space-codev ([5]).

GODOT is written in C++ with both C++ and Python user interfaces, and is available under an ESA Community License. GODOT is not a complete end-to-end application, but rather a set of libraries for performing generic orbit related computations for estimation, optimisation and orbital analysis, practically for any space mission. It was designed with operational orbit determination and trajectory optimisation as well as non-operational mission analysis as main applications.

However, it can be easily adapted for other astrodynamics scenarios and can even be used for attitude propagation and other attitude related tasks. As a consequence of its design as a toolkit, a substantial amount of work is required on top of GODOT to derive user libraries and applications.

Within this frame, Telespazio-Germany (TPZG) has been a key player from the beginning, participating with its staff in the GODOT core development as well as in the design and implementation of the operational layer for Orbit Determination (OD) of deep space missions. TPZG is also leading external projects for the exploitation of GODOT for the ESA Space Debris office and for the conversion of the ESOC SIMULUS generic simulation infrastructure to replace with GODOT the existing position and environment model (PEM).

This paper aims to summarise the contributions by Telespazio Germany to the success of GODOT and its usage, by describing some of the development and utilisation activities performed by the company's staff for both ESA-internal applications and external projects.

2. GODOT for deep space orbit determination applications

Working together with ESA staff and contractors in the ESOC FD division, Telespazio Germany's staff has largely contributed to the development of the libraries which make up GODOT, as well as to the design, implementation, testing and operational validation of the derived deep space OD system.
The OD system for deep space missions shares different software components with other teams both within and outside of ESOC's FD. Its high-level software design is summarised in Figure 1, where all the main software components – both in C++ and Python – and data repositories are represented, together with their interdependencies.

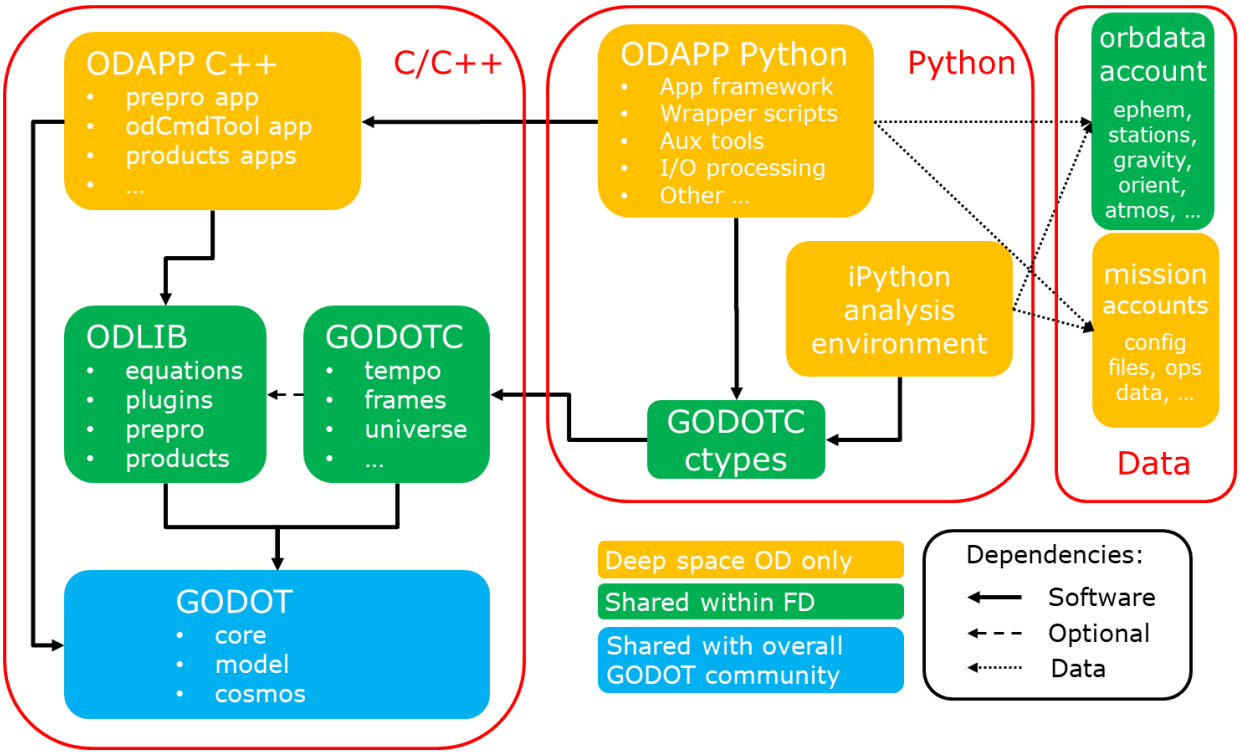

Figure 1. Deep space OD software high level design, covering both C++ and Python parts

The overall deep space OD software has been designed based on a few key drivers:

- **Sharing approach**: software and data should be shared where possible, i.e. avoiding overcomplicating the development/maintenance for this sole purpose, with the other FD subsystems (e.g. manoeuvre optimization, commands generation) for deep space missions and with counterparts for Earth orbiting missions. Moreover, GODOT is shared with the overall GODOT community, i.e. control over the content and design of its libraries is not fully under FD control.
- **Operational constraints**: operational software should be written in compiled language, scripting languages are only allowed for wrappers, input/output processing, analysis tools, etc. GODOT's main Python interface *godotpy*, written in Pybind and giving access to most classes and their interfaces, cannot therefore be used for operational nor analysis purposes.
- **Modularity and interfaces**: the software should be easily extensible and maintainable, hence based on object oriented programming paradigm and modern design methods. The most basic functionalities should nevertheless also be available with the legacy functional programming approach, in particular as interface for other FD subsystems and for enabling an interactive analysis environment in support of real-time operations.
- **Generality, flexibility, and usability**: The software should be generic, i.e. it should be possible to use the same software for all missions by simply adapting the configuration in the most simple cases, or by extending its functionalities and/or developing ad-hoc scripting tools if needed. Particular emphasis should be put on operational flexibility and usability: it should be possible to tackle very different scenarios with the same, modular applications; also, any orbit determination expert should be able to easily understand and perform simple changes to the configuration, even with limited experience on the system.
- **Accuracy and performances**: the software should allow to achieve the "same" results as with AMFIN legacy software ([2]), within the accuracy required for OD operations. Also, the software should have good performances, to the point of allowing OD operations within "reasonable" computational times, although some performance penalty is accepted in front of a much higher functional flexibility with respect to the legacy AMFIN software.

Due to these drivers, the design has naturally evolved to include the features in Figure 1. The main operational software is written in C++ and is split in three main layers: 1) the GODOT libraries; 2) ODLIB, another pure library layer shared with the OD system for Earth orbiters and expanding GODOT functionalities for OD tasks; and 3) ODAPP, developed on top of GODOT and ODLIB and made mostly of user applications for all operational tasks in support of deep space OD. To simplify user utilisation and automate recurring tasks, a wide suite of Python functions and scripts is also implemented in ODAPP, covering applications wrapping, inputs/outputs (I/O) processing, visualisation tools, etc. For completeness, a top layer similar to ODAPP in terms of functionalities was also developed, tailored for support to ESA Earth orbiting missions (e.g. higher focus on automation, less on models accuracy). TPZG was not involved in its development, and information can be found in a separate paper ([6]).

Since the full GODOT Python interface cannot be used for operations and a simpler programming interface based only on fundamental types must be available as interface to other FD subsystems, a separate C-style interface for GODOT is included in the software design. This is called GODOTC, is shared with other FD subsystems, and contains basic functions for time and frame

conversions, orbit and attitude data access, etc. To provide a connection to the Python world, a thin layer interface using Python ctypes for a 1-to-1 representation of the C interface is also included in GODOTC, and constitutes the core of an interactive Python analysis environment for deep space OD scenarios.

To complete the OD system, operational data accounts are also represented in Figure 1. For FD operations, a large set of data common to all Earth orbiting and deep space missions is maintained in a single data account, ***orbdata***. This contains both static data, such as planetary ephemeris or gravity fields, and dynamically updated data such as Earth Orientation Parameters (EOP) or solar flux files. Finally, each mission requires dedicated data accounts to store the setup, e.g. configuration files, environment variables, etc., as well as folders for input data, processing logs and outputs.

Based on the presented software design, the operational workflow and processes for generic OD support for any deep space mission are summarised in Figure 2, together with all the main input and output data flowing within the internal processes and from/to external parties. Note that the diagram is simplified, i.e. only the main processes and data flows are shown, in order to provide a clearer understanding of the most typical scenarios. The processes in the figure are representative of the three major steps in OD support (pre-processing, OD and orbital products) plus a series of auxiliary processes for automation, post-processing and visualisation of the results.

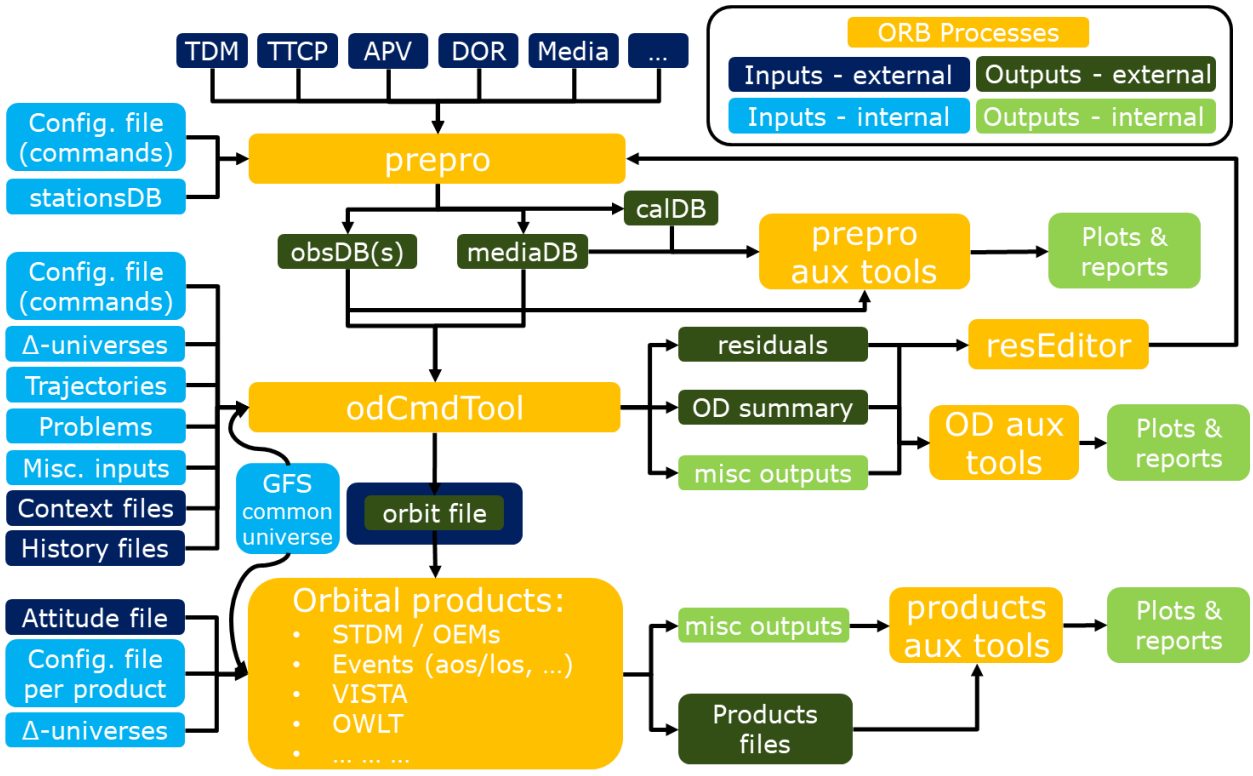

Figure 2. Generic operational workflow, processes and inputs/outputs in the OD system.

Pre-processing is performed with a single application named *prepro*, which is based on a modular user interface which executes an arbitrary list of commands, which can be sequentially combined

via configuration file to specialise the pre-processing for each mission. For instance, different tracking data types can be processed with different *prepro* commands and combined into one or more databases. For example, radiometric data or angular data can be imported in various formats specific to different types of ground station modems or antenna controllers in use across ESA stations, and both can also be processed from the standard Tracking Data Format (TDM) used for inter-agency exchanges. Similarly, station calibrations or media calibrations (e.g. meteo or GNSS corrections) can be processed and combined. The outputs of such process are one or more observations, calibrations, and media corrections databases (*obsDB*, *calDB* and *mediaDB*), which all have a common JSON format but a content which depends on each specific data type. Note that exactly the same process is followed for non-radiometric observations, such as antenna angles, ground optical observations, GNSS state or even on-board sensors data such as optical imaging or altimeters.

The next step is the OD process, which is performed with the *odCmdTool* application. Similar to *prepro*, also *odCmdTool* is based on a set of commands which perform specific tasks and hence allow assembling the OD process (or multiple processes) in the way that is most suited for each mission. The *odCmdTool* commands range from very simple tasks such as associating a specific object to the observations for modelling to gradually more complex ones such as trajectory propagation, observables modelling, OD iterations, mapping of given expressions to one or more epochs, etc. A typical operational procedure for most deep space missions would therefore combine these building blocks to run a pass-through for data validation, a full orbit determination, mapping to specific epochs (e.g. for swing-by targeting), generation and merging of orbit file including prediction in the future, and a final passthrough for OD solution verification. These steps could be carried out in a single or in multiple runs depending on operational decisions, configuring *odCmdTool* accordingly.

The final step is the generation of orbital products from an orbit file, which depending on the cases can be the orbit as generated by the OD or an external orbit from the MAN subsystem or other sources (e.g. for third party missions support). Due to the large number of possible products, a suite of different applications is available – each dedicated to the generation of a specific type of output. On top of these main three processes described, a wide range of tools – mostly written in Python exploiting the GODOTC interface and other common Python libraries – is implemented in ODAPP. Providing a detailed description of all these processes is out of the scope of this paper, but the most relevant ones are shown in Figure 2. Particularly worth mentioning is the residuals editing process using *resEditor*, a Python tool connected to I/O processing scripts, which allow the user to visualise the residuals of any type of observation, together with the associated statistics and observation properties. Via the editor, observations deemed to be outliers can be flagged or different weights can be applied when needed, generating a *prepro* configuration file, to be used to automatically update via *prepro* the *obsDB* files (feedback loop in Figure 2). Other auxiliary tools are available at all three steps to summarise the results for user inspection, for instance via plots of the tracking data time span or of raw troposphere corrections and fitted polynomials, summary tables for the OD solution, plotting of delta-v or range bias estimates, continuous accelerations plots, B-plane targeting plots, station or quasar visibility plots, events or time series summary tables, etc.

The input and output external interfaces, exchanged outside of the OD system, are represented in dark blue and dark green in Figure 2. The main inputs are all types of observation and correction

data from stations or other sensors (range, Doppler, Differential One-way Range or DOR, angles, meteo and GNSS corrections, optical measurements, etc.), "context" files generated by the command generation subsystem and representing the predicted spacecraft dynamics (attitude, articulation angles, etc.) and "history" files generated by ATT and representing reconstructions from telemetry or ground models (Solar Radiation Pressure SRP, thruster pulses). The main outputs are instead all orbital products to be delivered for various purposes, such as station predictions, one-way light-time, orbital events or time series files, and some additional outputs for use by other FD subsystems. Among the latter, particularly relevant are the *prepro* databases generated by *prepro*, and *residuals* and *OD summary* files generated by *odCmdTool*. These files, all in JSON format and with content defined by associated schema files which act both as validation and as documentation, are the main interface to an independent Test & Validation Orbit team (TVO), which is in charge of checking OD solutions and products before they can be sent externally.

TPZG's staff has provided major contributions at all three levels. At the bottom level, GODOT builds upon several third party libraries such as calceph for ephemeris access or eigen for linear algebra, and is composed of three main layers: *core*, a collection of astrodynamics utilities to build the higher-level software components; *model*, a collection of libraries which implement interfaces to allow a generic modelling scheme in the subsequent layers and applications; and *cosmos*, a library which provides high level functionalities for common orbit problems. Many of the *core*, *model* and *cosmos* libraries were mostly developed by TPZG staff, for instance the *frames* and *gravity* libraries which allow the generic definition of points, axes and gravitational trees. And all others saw major participation of TPZG staff, for instance: *core*'s *tempo* (time representation library), *autodif* (automatic differentiation library), and *orient* (bodies orientation library); *model*'s *prop* (propagator library); and *cosmos*'s *universe*, which provides a single user access point to a wide array of functionality, and *problem* which allows to define equations and parameter settings for an OD. TPZG staff has also designed and implemented the GODOTC interface and its 1-to-1 Python ctypes interface, exposing some selected functionalities of GODOT.

Both the *universe* and *problem* libraries can be conveniently expanded using a plugin system, which was exploited at the second level in ODLIB, which expands the GODOT functionalities specifically for orbit determination, for all those aspects that are common between Earth orbiting and deep space missions. ODLIB is constituted of a *util* library, which includes generic helper classes and functions such as for XML and JSON processing, and four main components: *plugins*, *prepro*, *equations*, and *products*. The *plugins* library is a container for all extensions to the GODOT basic constructs which are useful for OD applications, such as *universe* and *problem*. The *prepro* library includes all classes and methods which are required to read, modify, or write any type of *DataSets* which constitute the *obsDB, mediaDB* and *calDB* databases shown in Figure 2. Each *DataSet* can store one type of "observation", a term which is more widely used to also represent station calibrations, media corrections, etc. In order to better define the *DataSet* concept, the inheritance diagram of its base and derived classes is shown in Figure 3. Similarly, Figure 4 shows the inheritance diagram of the *EquationGenerator* classes in the ODLIB *equations* library, which expands the list of available equations defined within GODOT's orbit determination Problem, to cover the equation generators connected to ODLIB's *prepro*. Finally, the *products* library includes all classes and methods which are required to generate, read, or write data for the orbital products, of which the most common are shown in the output interfaces of Figure 2.

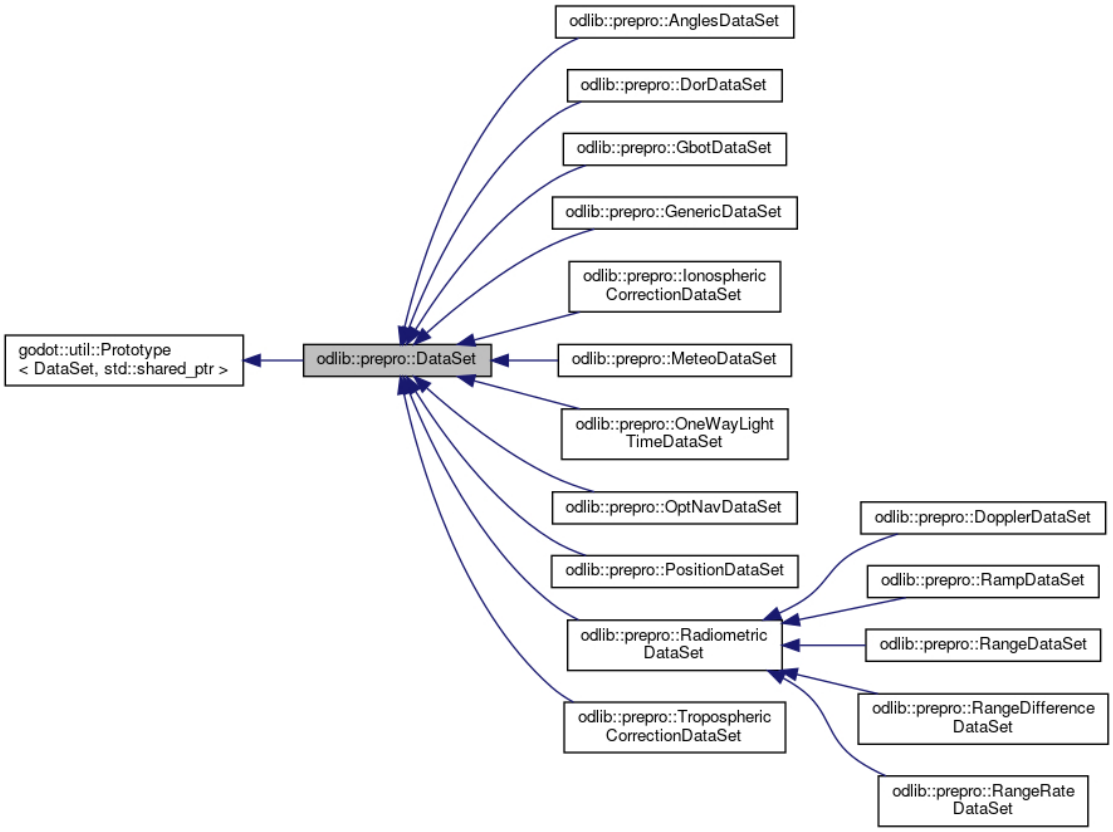

Figure 3. Inheritance diagram for ODLIB's *DataSet* classes.

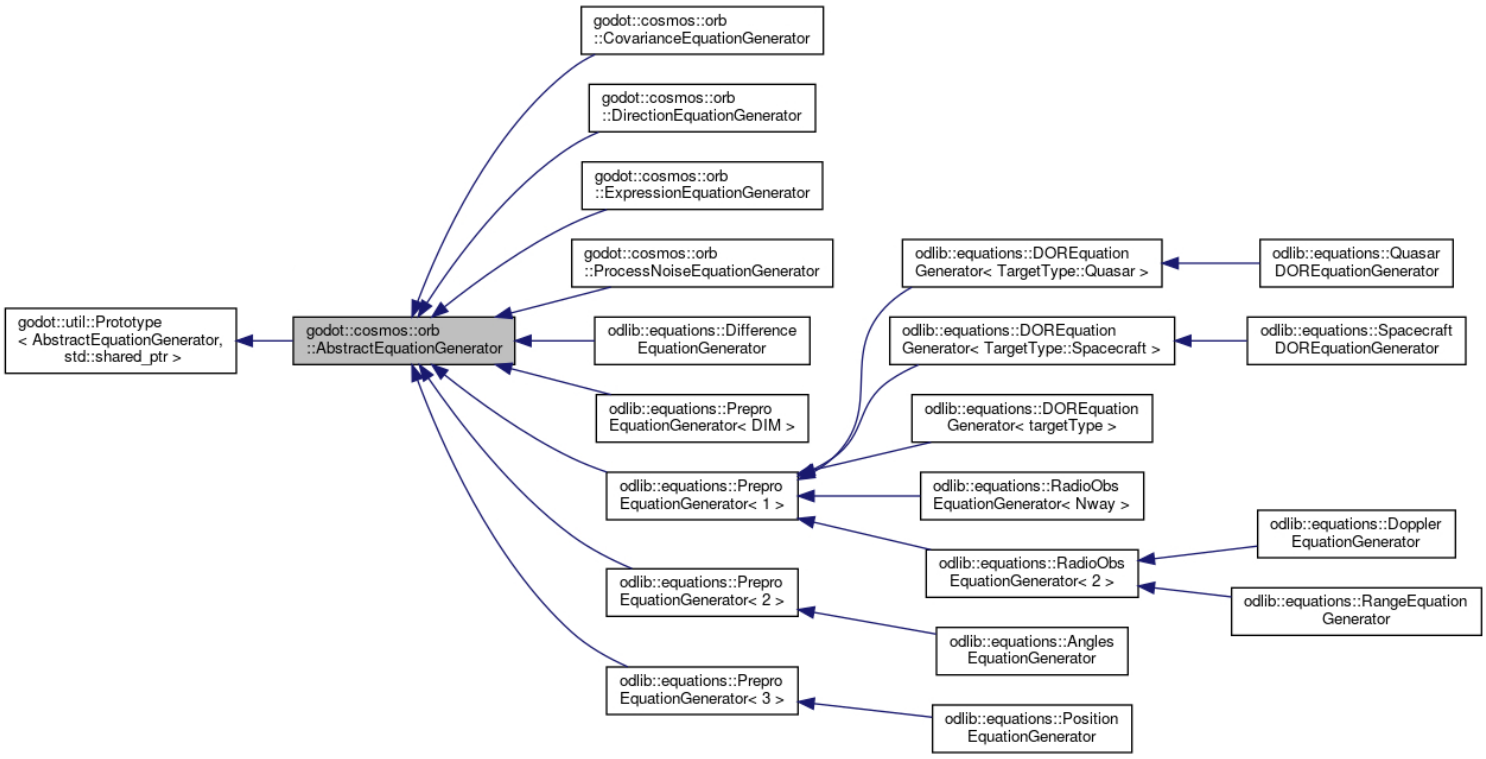

Figure 4. Inheritance diagram for ODLIB's *EquationGenerator* classes.

At the highest level, ODAPP is composed of a C++ and a Python section. The C++ part is constituted of main applications and related libraries, targeting specific operational scenarios for supporting any kind of deep space mission OD scenario. It relies heavily on the functionalities provided by GODOT and ODLIB, where most of the complexity is implemented, and is therefore a relatively simple layer. In addition to a small *util* library and few additional plugins (e.g. quasars database dedicated to deep space DDOR measurements), ODAPP/C++ contains the applications for the three main processes described above: *prepro*, *odCmdTool* and *products*. The Python part consists instead of several Python modules and a suite of Python scripts for a variety of purposes, relying on the GODOTC one-to-one Python interface and on other classical Python packages such as numpy, matplotlib, json, etc. The modules contain generic, mathematical and input/output processing utilities, as well as a general-purpose scripting framework developed for running all the applications in the OD system. This latter functionality has the purpose of defining and imposing a consistent approach for running any GODOT-based application for deep space OD, with an identical user interface and avoiding code duplication. On top of these modules and of the GODOTC interface, the ODAPP Python scripts are divided in three categories: *wrappers* of the C++ applications using the framework described above, *ioprocessing* scripts to generate inputs or post-process outputs, with the purpose of automating the generation of input files or better visualising output data, and stand-alone *tools* used to support operations (e.g. plot a trajectory and its characteristic parameters or to compare two different orbits).

Development of the deep space OD system is almost complete, at least for all aspects which allow supporting currently flying ESA deep space missions and the upcoming launches of JUICE, Euclid, and Hera. TPZG staff has been heavily involved in the design, implementation and testing of the ODLIB and ODAPP layers, on top of GODOT itself, and is now leading the operational validation of the overall system. Although GODOT, ODLIB and ODAPP make extensive use of unit testing and the accuracy of GODOT building blocks has been proven via cross-comparison tests against AMFIN and NAPEOS, additional validation activities were set up. Among others, the most important were the comparison of dynamics and observables modelling against independent TVO software for a JUICE escape trajectory and the parallel OD operations for Solar Orbiter's third Venus swing-by navigation campaign. A detailed description of these activities is beyond the scope of the current paper, but a more detailed overview can be found in ([7]). Following these activities, confidence in the new ESOC deep space OD system has reached a level sufficient for operational deployment. At the beginning of December 2022, the new GODOT-based system replaced the legacy AMFIN-based system for Solar Orbiter's prime OD operations, and it is expected to be used in 2023 for the launches of Juice in April and Euclid in July.

3. GODOT for Simulus generic simulation infrastructure

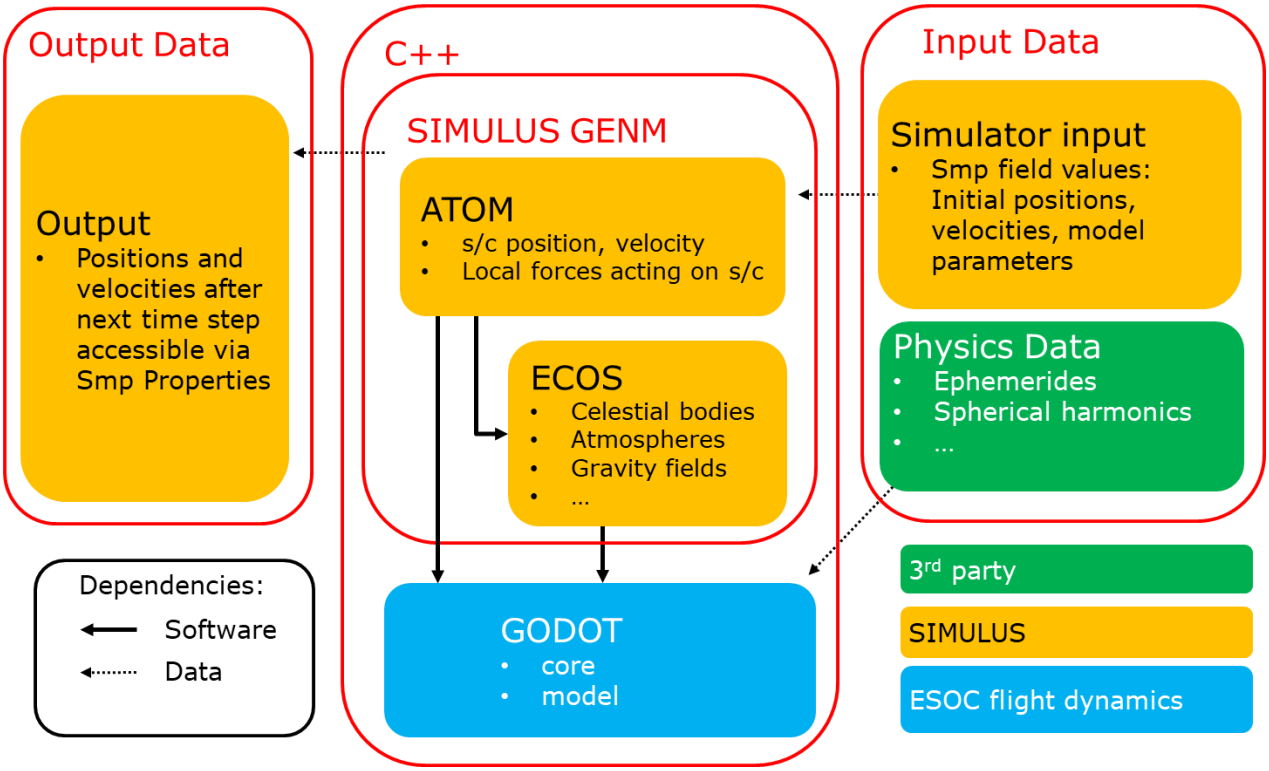

Figure 5. Integration of GODOT inside SIMULUS GENM

Within the ESOC operational simulator infrastructure SIMULUS, the **P**osition and **E**nvironment **M**odel (PEM), as part of the **Gen**eric **M**odels (GENM) is responsible for the propagation of the orbit of simulated spacecrafts taking into account its physical environment and the forces acting on it. It is based on a FORTRAN core with a C/C++ layer around it, allowing the integration into the rest of the SIMULUS code written in C++. While providing good performance and high execution speed, it has some limitations due to its old age and its architecture: For example, variables are stored in a common block memory which limits its use to the simulation of a single spacecraft, and the usage of FORTRAN makes the codebase harder to maintain and extend in comparison to more modern programming languages.

For these reasons it has been decided to replace the current PEM during the development of SIMULUS Next Generation ([10]) with a new environment modelling and orbit propagation system based on GODOT. Work has been ongoing at Telespazio Germany on this new system for the past year, and it is now nearing its completion.

The simplified summary of the overall architecture is shown in Figure 5. The two innermost GODOT layers *core* and *model* are used, while no usage is made of the *universe* layer. The new GENM component is split into the two sub-components *ATOM* and *ECOS*. The ATOM model is responsible for the propagation of a rigid body – a spacecraft or small body – through space, based on the gravitational forces acting on it as well as perturbation due to 3rd bodies, atmospheric drag, solar radiation pressure, and so on. Any number of ATOM models can be instantiated. The ECOS

simulation service provides the physical environment, i.e. the simulation of celestial bodies and their gravity fields, atmospheres, solar radiation, and so on.

Both ATOM and ECOS consist of several ECSS-SMP components that can be configured by the user via setting of field values. The components contain instances of corresponding GODOT classes that perform the calculations. They are set up and configured with the values from the components fields. The user also has to provide input files with physical parameters of the models. These files are passed on to the GODOT classes.

The orbit propagation is performed in small time steps during the running simulation, a typical update cycle is 250 ms. For each time step, the GODOT Propagator class is initialised at the current epoch time, the propagation performed and the position and velocity field values of the GENM ATOM models are updated. The updated values can then be accessed by other parts of the simulator of by the user via Properties.

This discrete time step simulation is not the typical use-case for GODOT, which initially has been developed for flight dynamics applications and thus is capable of performing high-precision calculations of long orbits in one step using sophisticated numerical integration methods, but has not been optimized to do many short propagations with a high frequency.

To improve execution speed, a simplified 4^{th} order Runge-Kutta integrator has been added to GODOT for the simulators use case, with a much faster execution than the default 7^{th} order one, but sufficient precision for the short time steps used in a typical simulation.

The correctness and accuracy of the implemented methods have been tested within GENM with a group of integration tests based on sample scenarios with orbits around the earth or in interplanetary flight, based on AMFIN and NAPEOS test cases. The end results of the orbit propagation are compared to the reference values and show excellent accuracy of the orbit propagation implemented in the simulator environment. To compare with an analytical reference, an integration test replicating a 5.15h LEO to GEO Hohmann Transfer based on ([8] page 147) has been added which shows an absolute error of 86 µm in the final orbit radius compared to the expected result of 42,160 km.

The execution speed is slower than the old FORTRAN implementation, however it still uses significantly less time than other parts of the simulator, and is far from endangering the real-time operation of the simulators.

Some of the PEM functionalities have no equivalent implementation inside PEM. This includes the calculation of the reflection of sunlight that reaches a solar panel due to the albedo of a celestial body, or the magnetic field of celestial bodies. In these cases the extensible architecture of GODOT has been exploited and local extension to GODOT have been written, making use of some of the GODOT functionality, such as the frames system, whereas the actual implementation of the physics models has been adopted from the original code, based on the Flatley-Moore Albedo Model, and the IGRF Magnetic Field Model. In the case of the magnetic field, a general *MagneticField* interface has been added to GODOT, with the implementation left to the user.

On the other side, the new GODOT-based implementation extends the PEM functionality: For example, it is now possible to simulate more than one spacecraft, or several small bodies simultaneously. Another new feature is the possibility to use high-accuracy 7th order Runge-Kutta integration for the simulation of longer time, e.g. for a time jump in the simulation. In the future more GODOT functionality can be easily added to the simulation environment by integrating them in ECSS-SMP Models. Possible candidates are different atmospheric models or the usage of relativistic calculations in the orbit propagation.

In addition, due to its modular design, the GODOT functionality can be easily extended locally to account for special needs not present in the software, as already done for the magnetic field model or the Flatley-Moore Albedo Model.

The next steps in the development will be the integration into a full mission simulator, the foreseen candidate is JUICESIM, the operational simulator for the Jupiter Icy Moons Explorer mission.

4. GODOT for Space debris applications

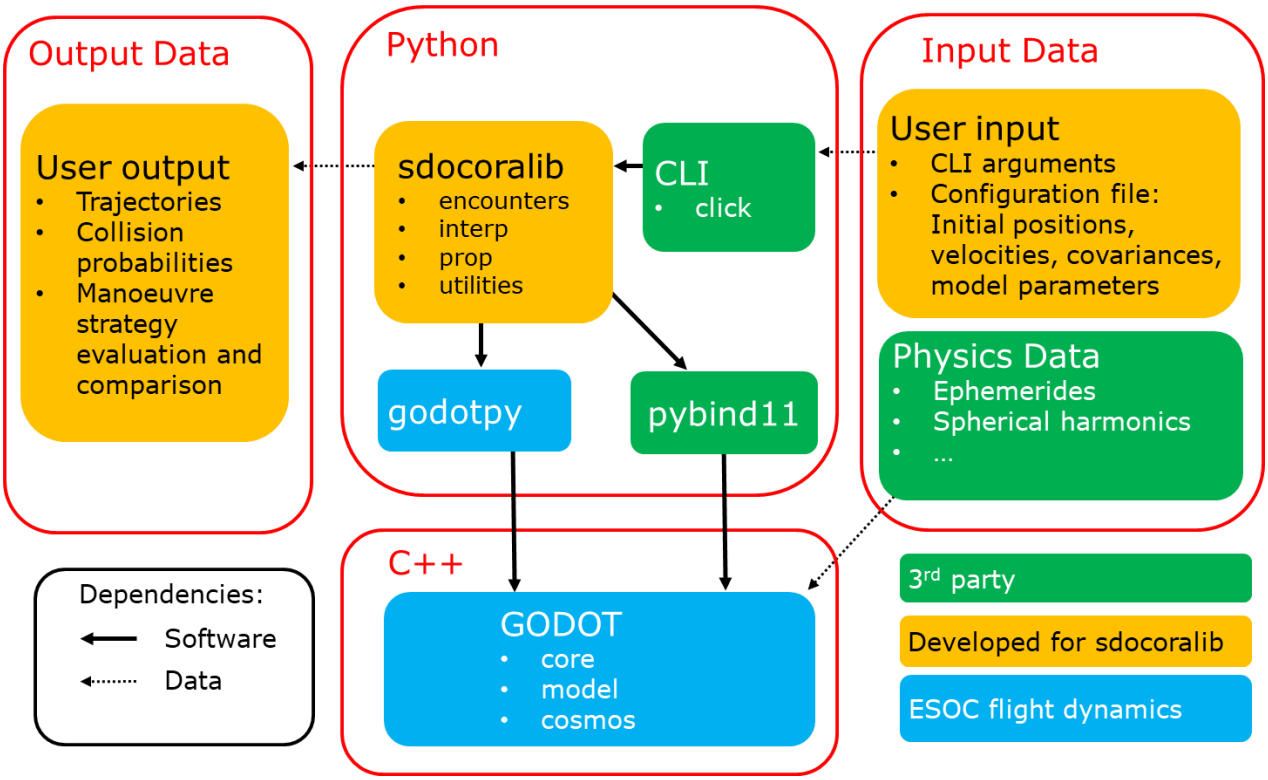

Figure 6. Space debris software high level design and data flow

Telespazio Germany is also leading a project to replace the current FORTRAN-based toolkit CORAM ([9]) used by ESOC's Space Debris Office by a new Python based library and command line interface (CLI), called *sdocoralib*.

The new library uses GODOT for conjunction detection between spacecrafts and space debris, collision probability calculations and mitigation strategy planning. A simplified high level design

is shown in Figure 6. It makes use of all the GODOT layers *core*, *model* and *cosmos*. Most of the GODOT functionality is accessed in Python via GODOT's Python interface *godotpy*. Some parts of GODOT are not exposed yet via its Python interface, notably the BlockDataInterpolator class and related classes. Although the *godotpy* interface is being expanded to cover external projects requests, these classes have been in the meantime made available to sdocorablib via local *pybind11* wrapper. The sdocoralib library itself is composed of four sub-packages: *encounters*, *interp*, *prop* and *utilities*. In addition to using sdocoralib as a library also a CLI has been made available, it has been implemented with *click*.

The user of the library has to provide a configuration file specifying the initial conditions, i.e. the initial positions and velocities of the spacecraft ("target") and space debris object ("chaser"), together with the covariance matrices, the starting time and duration of the simulation, pre-planned manoeuvres, together with uncertainties on them and which perturbations to consider. The values are passed on to GODOT's *universe* layer, which creates a trajectory configuration as well as linear equations problem for the final states and covariances out of it. The user has to provide as well input files with the parameters for the physical models, such as spherical harmonics coefficients, and so on.

Depending on what scenario the user wants to simulate, the *prop* package of the library is used for propagation of target and chaser, the *interp* package for interpolation between time steps, and the *encounters* package for the calculation of collision probabilities. The *utilities* package contains code for plotting and parsing input files.
For the orbit propagation and state interpolation the library calls GODOT classes from the *model* and *core* layers via their *godotpy* interface or *pybind11*. For propagation, the *Propagator* class is used and for the calculation of the final covariances the *Problem* and *Solver* classes. The library makes as well use of other common Python packages, such as numpy or matplotlib.

The output of the library consists of data that can be further used in the collision avoidance framework, as well as plots that can be inspected by the user. The outcome of a typical use case is illustrated in Figure 7. For a collision scenario the minimum required manoeuvre size is depicted in order to avoid this collision with a certain probability. The manoeuvre is in flight direction of the spacecraft, and the plot shows as a function of the time of the manoeuvre, the required impulse for a collision probability lower than 1 in a millions or 2 in a million. The user can then e.g. choose a manoeuvre time that offers a sufficiently low collision probability with the least fuel spending. In the depicted case an early manoeuvre would be optimal.

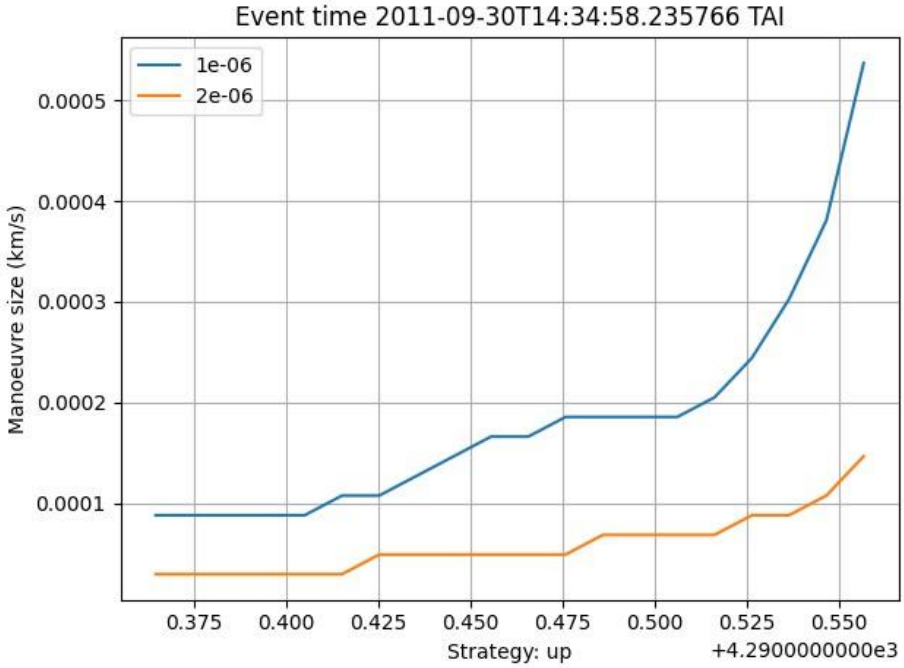

Figure 7. Sdocoralib usage example: Minimal required manoeuvre size for a collision probability of less than 1 in a million or 2 in a million as function of the manoeuvre time.

The performance and accuracy of the library has been tested by comparing its results in default scenarios taken from CORAM test cases. During development these tests were integrated in a CI/CD pipeline as regression test in order to ensure the correctness of the software when adding new features.

All of CORAM's key features, i.e. collision probability calculation, manoeuvre planning and optimization have been implemented in the new library. It is foreseen to replace it in the operational environment in the future. For future developments it would be advantageous to have the full GODOT functionality exposed via its Python interface, to avoid having to manually implement the Python wrapping with pybind11. Work is ongoing in this direction, in coordination with the GODOT core development team.

5. Conclusions

Through three main topics, this paper provides a high level overview of GODOT and of some of its possible applications to different domains, with a focus on the key role played by Telespazio Germany in its development and early use cases. In particular, in the field of deep space operational orbit determination, a GODOT-based system has been developed and is being phased into operations for ESA deep space missions, providing a significant boost in flexibility, usability and extensibility, while keeping and in some areas improving the accuracy and robustness of the legacy software. For operational simulators, the SIMULUS infrastructure is being updated to replace the existing Fortran propagation tools with modern C++ libraries connecting to GODOT, removing some important limitations of the legacy implementation while at the same time drastically improving maintainability, with negligible impact on the performances. Finally, for space debris

analyses and operations, a new Python package is being developed, aimed at replacing existing operational tools for collision avoidance purposes, also improving usability and extensibility with the added advantage of extreme flexibility granted by the use of the GODOT Python interface and other Python common packages. These widely diverse and successful applicative scenarios, prove how GODOT's new approach to shared orbit software infrastructure can represent a game changer for the European institutional, industrial and academic communities.

Acknowledgments

The authors would like to acknowledge all colleagues at ESA/ESOC in both the GODOT core development team and the deep space missions Orbit Determination team, with whom a significant part of the work presented in this paper was carried out.
Furthermore, the authors would like to acknowledge the ESA/ESOC Space Debris Office for funding und supporting the "CAP – State and Covariance Propagation and Interpolation" project and IMS Space Consultancy GmbH for the smooth and fruitful collaboration.
Finally, the authors would like to acknowledge ESA/ESOC OPG-GD for funding und supporting the on-going development project of the SIMULUS Next Generation (SIMULUS-NG) infrastructure.

References

[1] Garcia-Matamoros M. A., Kuijper D. and Righetti P.L., "NAPEOS: ESA/ESOC navigation package for earth observation satellites", Proceedings of the European Workshop on Flight Dynamics Facilities, Darmstadt, Germany, 2003.
[2] Budnik, F., Morley, T. and Mackenzie, R.A. "ESOC's System for Interplanetary Orbit Determination", 18th International Symposium on Space Flight Dynamics, Munich, Germany, October 2004.
[3] Mackenzie, R.A. "GODOT Flight Dynamics Infrastructure Software for operations and analysis", 1st European Workshop on Space Flight Dynamics Services, Systems and Operations, ESA-ESOC, Darmstadt, Germany, September 2021.
[4] https://godot.io.esa.int, https://godot.io.esa.int/docs, https://godot.io.esa.int/godotpy
[5] https://gitlab.space-codev.org
[6] Ramos Bosch P., Vasconcelos A., Kuchynka P., and Sanchez J., "The New ESOC Flight Dynamics Operational Software for Earth Orbiting satellites (GENEOS)", 28th International Symposium on Space Flight Dynamics, Beijing, China, August 2022.
[7] Castellini F., Godard B., Dei Tos D., Mackenzie R., and Budnik F., "ESOC's New Orbit Determination System for Deep Space Mission Operations", 28th International Symposium on Space Flight Dynamics, Beijing, China, August 2022.
[8] Wertz, J. R, Larson W. J., "Space Mission Analysis and Design, third edition", Microcosm Press.
[9] Cobo, J. et al., "CORAM: ESA's Collision Avoidance Risk Assessment and Avoidance Manoeuvres Computation Tool", 2014
[10] https://www.esa.int/Enabling_Support/Operations/Investing_in_space_on_Earth