IWSCFF 2022

Novel Architecture for Numerical Multi-satellite Simulations

João Vaz Carneiro* **and Hanspeter Schaub†**

This work aims to introduce a new architecture that allows for easy integration of multiple satellites in a single simulation. The architecture is implemented in Basilisk, an open-source, flight-proven physics and flight software engine, although the fundamental principles can be applied to any software application. The new design focuses on modularity, expandability and easy scriptability while maintaining the high-fidelity and speed features of Basilisk. Modularity is important to make highly specialized simulations, which includes parameters related to the environment (gravitational bodies, ground locations), but also specific to each spacecraft (mass properties, attitude control system, flight software modes). Expandability and scriptability are also key, as one of the goals is to ease the effort of creating simulations with a large number of satellites. Through an overhauled messaging system, the architecture also allows for easy addition of homogeneous or heterogeneous satellites with reduced overhead. Throughout this paper, code snippets and multithreading simulation performance are shown to discuss how the architecture achieves its underlying objectives of simplicity and performance.

keywords: Innovative technologies for distributed systems, Modelling and Parameterization of Relative Dynamics, Mega constellations

1. Introduction

As the interest for spacecraft constellations has been developing over the years, the focus has shifted from science-oriented projects sponsored by government agencies to communications-oriented applications sponsored by commercial partners. The Starlink[‡] project, which focuses on delivering internet services to remote and under-served locations, is financed by SpaceX and has already launched more than two thousand satellites, with tens of thousands in development. Amazon's Project Kuiper[§], while in an earlier development stage, aims to tackle a similar problem. Undoubtedly, large spacecraft constellations will become a core part of the spacecraft population in Earth's orbit. Their successful implementation requires sophisticated software that can accurately simulate each spacecraft in orbit.

Building a simulation with many satellites can become cumbersome if the software architecture is not built for multi-spacecraft prototyping. Many software packages are able to simulate a single spacecraft with high fidelity and speed, such as AGI's Systems Tool Kit (STK) [1], or NASA's General Mission Analysis Tool (GMAT) [2]. Creating a simulation with multiple satellites brings new challenges. Usually, the user must manually include and specify every single spacecraft, which for large constellations gets increasingly time-consuming and yields cluttered and hard-to-follow scripts. Moreover, if the architecture is not built with multi-spacecraft simulations in mind, then it likely simulates every spacecraft in series. This means that the additional time it takes to run the simulation will roughly increase linearly with the number of spacecraft, which becomes a problem if the goal is to simulate tens or hundreds of satellites at a time. The new architecture tackles the challenges that come with multi-satellite simulations, with the goal of making them simple to set up and maintain, while using an engineering-friendly Python scripting interface.

While the focus of this paper is to propose a general architecture framework that supports simulations of multiple spacecraft, the specific software implementation is also addressed. For this work, the Basilisk [3] astrodynamics software tool is used to implemented the architecture and create the example scenarios. Basilisk is a flight-proven modular mission simulation framework and it is used to set up high-fidelity simulations. Its modular nature [4] allows for the simple integration of complex simulation tasks, such as power generation and consumption, fully-coupled attitude control devices [5], or orbital perturbations. Modules are created using C/C++ for rapid execu-

*Graduate Research Assistant, Ann and H.J. Smead Department of Aerospace Engineering Sciences, University of Colorado, Boulder (CO), United States, joao.carneiro@colorado.edu

†Professor, Glenn L. Murphy Chair in Engineering, Ann and H.J. Smead Department of Aerospace Engineering Sciences, University of Colorado, Boulder (CO), United States, hanspeter.schaub@colorado.edu

‡`https://www.starlink.com/`

§`https://tinyurl.com/amazonkuiper`

tion, while the user interacts and connects modules using Python, for easy scriptability. The software implementation of this architecture, as well as all example scenarios, are available online and are free to use[¶].

The implementation of this redefined architecture has been made possible by the addition of the new messaging system [6] in Basilisk. While multi-satellite simulations had been created using the old messaging system with Basilisk 1, the overhauled messaging system makes the simulation design substantially easier. With its peer-to-peer message connections, Basilisk 2 allows the modules to be easily connected upon initialization, without having the user figure out how to connect and name modules from a single message pool.

2. Architecture Design

The goal of the proposed redesign is to create an architecture that effectively stimulates multiple satellites, while avoiding compromising speed and fidelity. To that end, this novel framework is based on four principles: modularity, scalability, parallelization and scriptability.

2.1 *Modularity*

In this context, modularity is the ability of a software framework to be divided in smaller pieces that can be joined to create the simulation. It means that each simulation feature or module is detached from the other, and multiple modules need to be added to run the simulation as intended. While this takes a toll on simulation time, as the modules are run separately, it saves on development time.

This is particularly important in the context of complex spacecraft simulation. Different missions may require different attitude control systems. Some may use reaction wheels for their precise pointing characteristics, while other may use control moment gyroscopes (CMGs) for their larger torque needs. While different, both these systems have the same objective of generating a requested torque for attitude control. By modularizing the software framework, the developer is able to quickly switch between each attitude control device, implemented as distinct modules, without having to overhaul the entire attitude control system. Basilisk has been built from the ground up as a modular system, and this work takes advantage of that structure.

[¶] `http://hanspeterschaub.info/basilisk/index.html`

However, for the simulation to run, the individual modules need to communicate and share information with each other. For the attitude control system example, this means that the requested torque from the attitude control module must be passed onto the attitude control device module (reaction wheels or CMGs). In Basilisk, the information is shared through a messaging system, which is discussed in-depth in section 4.

2.2 *Scalability/Expandability*

Scalability, or expandability, represents the ability to increase the number of satellites in a single simulation. This is critical for scenarios with a large number of satellites, and it is what sets this architecture apart from usual software designs.

For this work, scalability is achieved by standardizing the creation of the class that sets up the simulation environment, as well as the classes responsible for simulating each spacecraft's dynamics and flight software (FSW) routines. The addition of new spacecraft, which is done by creating more instances of dynamics and FSW classes, is implemented through a loop that creates and connects every module necessary for every spacecraft. This way, the user can add and customize as many spacecraft as needed with no major changes to the framework.

2.3 *Parallelization*

Parallelization allows the software to exploit the architecture of modern CPUs. Most processing units contain multiple cores, with some cores consisting of multiple threads. This means that different processes can be run at the same time using different threads.

The importance of parallelization for multi-spacecraft simulation is clear. Running each spacecraft's process in parallel lessens the otherwise linear increase in simulation time for an increasing number of satellites. Simulating each spacecraft in parallel decreases the simulation time associated with more satellites, although it is limited by the number of available threads.

This parallelization allows for multithreading, which is when multiple threads are used at the same time for different processes. Multithreading can have its drawbacks, such as when two threads read and modify the same data simultaneously, or when two threads are not coordinated in time properly. Making a program multithread-safe is non-trivial. For this work, the proposed architecture is built to allow for multithreading.

It should be noted that it is not possible to parallelize the dynamics of a single spacecraft, as it con-

tains strongly coupled nonlinear differential equations. However, it is possible to run the dynamics of each spacecraft in separate threads. Thus, with the presented multi-threaded approach, the simulation speed itself is not increased, but rather the numerical speed up is achieved by simulating a large number of spacecraft.

2.4 *Scriptability*

Scriptability relates to the ease of simulation setup and development. Performing a simulation with multiple spacecraft usually implies code repetition, inefficient routines and overall hard-to-follow scripts. Since this architecture is focused on easing the effort of implementing multi-satellite simulations, the effort of creating multiple satellites (whether homogeneous or heterogeneous) is greatly simplified. This allows the user to focus on adding the necessary modules to create the intended simulation. Another advantage is that a user can create different classes that, if implemented correctly, can be changed between each other in a plug-and-play fashion. For example, two different environment classes, one around Earth and another around Mars, can be created and swapped as easily as changing a single line of code. It also means that debugging is vastly simplified, as the scripts are organized per spacecraft and are easier to follow.

In addition, no recompiling is necessary to create and modify the simulation scripts. The user can add, connect and change all simulation modules in Python without having to go through the lengthy process of recompiling the C and C++ Basilisk files.

3. Architecture Framework

Guided by the goals and constraints expressed in the previous section, the multi-satellite architecture is now presented. The architecture's diagram is shown in Figure 1.

There are four classes: Master, Environment (Env), Dynamics (Dyn) and Flight Software (FSW). Only one Master and Environment classes exist, while there exist as many Dynamics and Flight Software classes as the number of satellites in the simulation. The connections between classes are done through modules taking advantage of the messaging system, which is used to share information between them.

Analyzing the diagram, the architecture's parallelization is evident. The Dynamics and Flight Software classes of each spacecraft are independent, so they can be run simultaneously, saving on computation time. This architecture is also easily expandable by simply attaching more Dyn and FSW classes according to the total number of satellites.

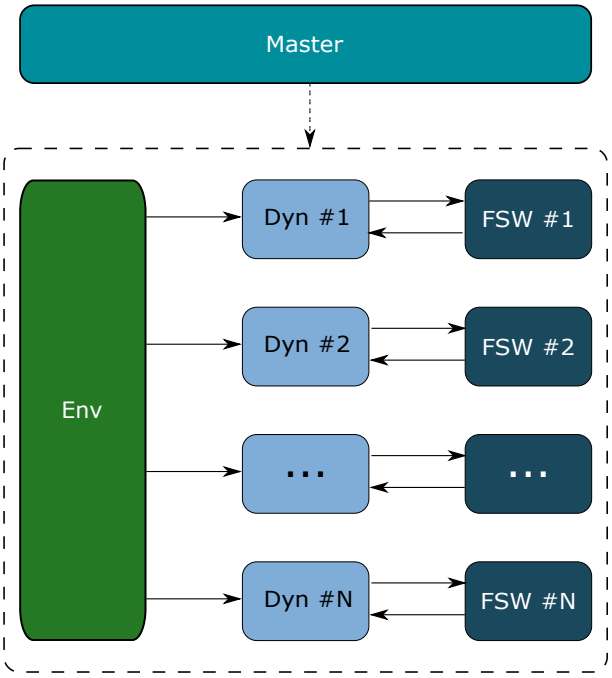

Fig. 1: Multi-satellite architecture diagram. Solid arrows represent information sharing through messages. The dashed arrow below the Master class corresponds to the functions that initialize and access all classes. The dashed box encompasses all the simulation classes, which contain the modules that are run during the simulation.

3.1 *Master Class*

The Master class contains the methods that create and manage the other three classes. This class is created in the scenario script, where these methods can be called for initialization and retrieval. To attach the Environment class to the simulation, see the code snippet in Listing 1.

In Listing 1, `BSK_EnvEarth` is a Python script that contains the Environment class, with its modules inside. Similarly, the Dynamics and Flight Software

Listing 1: Attach an Environment model to the simulation.

```
self.set_EnvModel(BSK_EnvEarth)
```

Listing 2: Attach Dynamic and Flight Software models to the simulation.

```
self.set_DynModel([BSK_MultiSatDyn]*numberSpacecraft)
self.set_FswModel([BSK_MultiSatFsw]*numberSpacecraft)
```

Listing 3: Attach heterogeneous Dynamic and Flight Software models to the simulation.

```
self.set_DynModel([BSK_MultiSatDyn1, BSK_MultiSatDyn2,
↪  BSK_MultiSatDyn3])
self.set_FswModel([BSK_MultiSatFsw1, BSK_MultiSatFsw2,
↪  BSK_MultiSatFsw3])
```

classes are attached through the function call shown in Listing 2.

Again, `BSK_MultiSatDyn` and `BSK_MultiSatFsw` are Python scripts that contain the Dynamics and Flight Software classes, respectively. We see that the argument of the methods shown is a list of Dyn or FSW classes. In this case, all spacecraft are the same, which is why a list of identical classes is added as an input. If a heterogeneous set of satellites is to be implemented, the list would contain different classes in the proper order, as shown in Listing 3. The user would have to create each class to suit the simulation requirements.

It is also in the Master class where the methods that access the the Env, Dyn and FSW classes are implemented. Within the scenario script, the functions in Listing 4 are called to retrieve and access all classes and its modules.

3.2 *Environment Class*

The environment class contains modules that are not spacecraft-specific, but that instead describe the simulation environment. The gravity field is modeled within this class, along with atmospheric perturbations such as the effect of drag through the use of an atmospheric density model. Ground stations for communications or imaging are also added to this class.

Listing 4: Retrieve simulation classes.

```
EnvModel = self.get_EnvModel()
DynModels = self.get_DynModel()
FswModels = self.get_FswModel()
```

Since this class is not spacecraft specific, it is shared among all spacecraft classes. This also means that the user can readily change between different environments, such as the Martian or Cislunar environments, without having to make changes to each spacecraft. However, one must be careful about setting each spacecraft's initial conditions: a reasonable orbit around the Moon might not work around Earth. To solve this, the spacecraft's initial conditions are set using orbital elements with a canonical semi-major axis value, i.e. the semi-major axis is set to be x times the main body's radius, with the constraint that the orbit's periapsis must be larger than the main body's radius.

3.3 *Dynamics Class*

The dynamics class contains the modules that recreate the spacecraft and its components, which means one must exist for each spacecraft. While in most cases all spacecraft are identical, there may be situations where the simulated spacecraft may have to be different. This might be the case for a mission where a single mother ship centralizes the information coming from several smaller spacecraft. The proposed architecture allows for both situations to be simulated, and it is up to the user to configure different dynamics classes if a heterogeneous constellation is required.

It is within the dynamics class that critical subsystems are implemented. This includes the power system, which contains solar panels for charging energy and batteries for storing it. Components that require energy, such as attitude control devices, transmitters or cameras are also integrated into the power system, so that the energy consumption and generation is properly accounted for. The attitude control system is also implemented in the dynamics class, and it includes reaction wheels, control moment gyroscopes and thrusters. It should be noted that only the dynamics of these attitude control devices, and the corresponding effect on the spacecraft, are simulated within this class. The control law is part of the flight software class. Finally, the modules related to the instrument system, which includes imagers, transmitters and data buffers, are also included in the dynamics class.

3.4 *Flight Software Class*

The FSW class contains the logic that would go into the on-board computer of the spacecraft. While the environment and dynamics classes simulate physical phenomena, the flight software class includes the modules that make decisions based on the spacecraft's position, velocity, attitude, etc. It contains the code that would be uploaded to the real spacecraft's computer, contains the instructions and logic to make the spacecraft meet its mission objectives.

It is within the FSW class that the attitude modes are set, which dictate where the spacecraft should point. These may include pointing the solar panels at the Sun for battery charging (Sun pointing), pointing the antenna at the Earth for downlinking data (Nadir pointing), or pointing a sensor at a target on the planet's surface for imaging (target pointing). This class also contains the logic for the attitude control system. Given a reference attitude and attitude rate, a required torque is computed and mapped onto the attitude control devices (reaction wheels, control moment gyroscopes, thrusters), which drives the spacecraft's attitude to the reference attitude.

Beyond attitude control, relative orbit control is also implemented, which calculates the necessary burns to enable specific formation flying maneuvers. Both the attitude and relative orbit control laws are derived in Schaub and Junkins [7].

4. Messaging System

Due to its modular nature, a messaging system is necessary for this architecture. Its purpose is to transfer information from one module to the other, so that all modules have the most up-to-date information at run time. For the simulation to work properly, the messaging system needs to be fast while still retaining accuracy in the information it delivers between modules. Another important aspect is its user-friendliness: the more intuitive the system, the faster the user can connect modules without making mistakes.

Basilisk's messaging system uses messages, which, at their core, consist of C/C++ structures. This architecture would not have been possible without substantial modifications to the messaging system in the release of Basilisk 2.0. An explanation of the new messaging system is given by Carnahan, Piggott and Schaub [6].

The old messaging system relied on a message pool. All messages were stored in a container and were available to all modules. A particular module would grab the correct message by searching the container for the message by its name. The name was auto-generated, which meant that the message connections were implicit: the user did not have to set a name for every message, and the modules were developed in such a way that they would search for the message with an expected predefined name. For example, the attitude control device module would expect an input message with the same name as the output message of the attitude control law module. A diagram showing the structure of the old messaging is shown in Figure 2. The advantages of the old messaging system included the speed of the connections, simplicity and readability of message identifiers to users and implicit message connections, where the user did not need to worry about connecting the correct messages between modules [6].

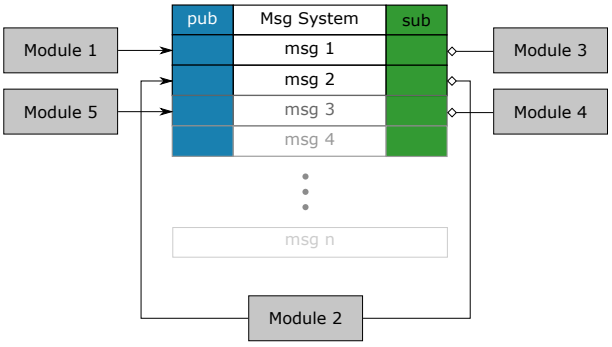

Fig. 2: Old messaging system diagram. Modules read and write messages in an universal message pool, which is categorized by message name.

However, the system had fundamental challenges, which were particularly evident in multi-spacecraft simulations. First, because multiple instances of the same module were created, the user had to manually change the name of each affected message for the connections to be properly set. Take a simulation of 3 spacecraft, each with an attitude feedback controller and a set of three reaction wheels: the user would have to manually assign a name for each output message of the three attitude feedback controller modules, so that the required torque is passed onto the correct set of reaction wheels. It is easy to imagine how cumbersome this would have become for a simulation of tens of satellites. Moreover, typos in message names would have made modules unable to access the proper data, requiring the user to go through a complicated troubleshooting process. The system also had no way to verify that the proper message type was being connected, which could have lead to

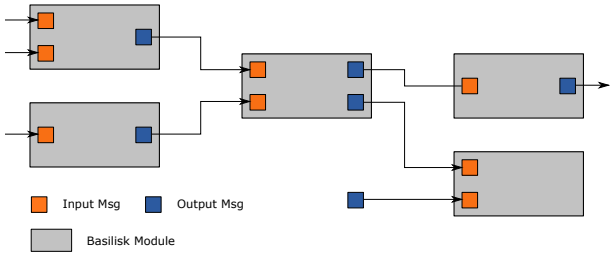

Fig. 3: New messaging system diagram. Messages are subscribed to a specific module in a peer-to-peer system.

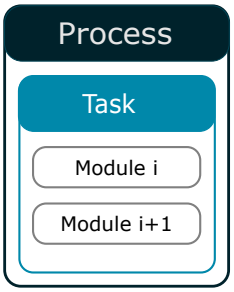

Fig. 4: Basilisk process architecture.

the simulation running with incorrect information if it was not properly configured. The false configuration issues become stronger as the number of satellites increases. Therefore, an overhaul to this system was implemented.

Instead of a message storage container, the new system uses message classes. Messages are now explicitly connected between modules by the user, which tackles most of the drawbacks of the old system. There is no need to name the messages anymore, which takes care of the naming problem the old system had. Moreover, connecting messages between modules allow for strong type checking, as the message identifier is now a class instead of a string. When the simulation is initialized, each module verifies that the input messages correspond to the expected type; if not, an error flag is thrown and the user can quickly troubleshoot the problem. A diagram for the new messaging system structure is shown in Figure 3.

Most importantly for this architecture, the new explicit connections allow for much easier expandability. There is no need anymore to name messages for instances of the same module. Moreover, it is possible to automatically connect messages between modules in a loop for every spacecraft instance, which makes creating simulations of hundreds of satellites as easy as simulations of a single one.

5. Process and Module Design

To understand the simulation flow, process and module design is important. The process architecture for Basilisk is explained in depth in [3]. A simplified diagram of this process architecture is shown in Figure 4.

Processes (or task groups in [3]) correspond to the top-level structures in Basilisk. They can contain one or more tasks, which contain individual modules. Modules within a task run at the same integration rate. This allows the user to group modules that require similar time step fidelity with each other. For example, attitude control devices should be updated more frequently due to their dynamics, while orbit propagation is usually less dynamic. To stop running the modules within a task, a task can be disabled.

For this work, each class contains its own process. The environment and dynamics processes (one per spacecraft) only contain one task, as all modules can be updated at the same rate and will never be disabled. However, each FSW process contains multiple tasks associated with each flight mode. This happens because when a flight mode is active, all others should be disabled. When running the scenario using multithreading, each process is assigned to a single thread. It is not possible to separate and assign different tasks within a process to different threads. Nonetheless, more than one process can run within each available thread.

For the simulation to run as intended, the order of initialization and execution are very important. Wrong initialization of the simulation classes can lead to modules trying to connect messages to other modules that do not exist, as they have not been created yet. Poor execution order leads to modules having mismatched and potentially outdated information, which can impact the guidance and control algorithms.

For this architecture, the initialization and execution orders are identical. This is because the flow of information dictates both how the modules are created, but also how they are updated at each time step. The modules that do not depend on other to run are created/updated first, and the ones with the most dependencies are last in line.

Following that hierarchy, the environment class is the first to be initialized and updated. It contains modules that compute the position and velocity of the gravitational bodies, the density of the atmosphere or even the position of ground stations. All

these modules do not depend on the spacecraft, and therefore have no external dependencies. The gravitational bodies' information is drawn from a catalogue, the density of the atmosphere follows a statistical model, and the position of ground locations can be calculated from initial conditions and the planet's rotation.

The dynamics class is updated next. Some modules used are self-contained and do not need information from the environment class. For example, the attitude is propagated using the spacecraft's previous attitude, as well as its inertia and the dynamics from attitude control devices such as reaction wheels. However, the environment modules do influence some of the dynamics modules. Orbit propagation is done in the dynamics class, and it depends on where the gravitational bodies are located and what their properties are. Therefore, this class directly depends on the environment class modules.

The final class to be updated is the Flight Software. The FSW class contains the modules that are most dependent on the other classes: ground locations for tracking and spacecraft attitude for the control law are examples of this. However, the FSW class can have some effect on dynamics modules. For example, given the current and reference attitudes, one FSW module uses a control law to request a desired torque. This torque is mapped onto the attitude control devices, which impacts their dynamics. However, these devices are simulated within the dynamics class. This can potentially create problems, as the dynamics have already been updated once the FSW modules are run. Ultimately, this is not a problem because the information passed from FSW to the dynamics class only needs to affect the simulation at the next time step, when the Environment-Dynamics-FSW loop is run once again.

With multiple spacecraft, this execution order becomes even more important. All dynamics classes are initialized and run before any FSW classes, which are updated afterwards. This is to ensure that all FSW classes have the most up-to-date information about the spacecraft's properties. While most FSW modules concern their own spacecraft, there are times where the information regarding other spacecraft is used. Suppose a constellation of satellites is set to do science on Earth. Depending on the current science objective, the satellite formation might take different shapes, such as a string of pearls or a double echelon. Assuming that there is a chief satellite, the other spacecraft must receive information about its position and velocity to maintain or change from one formation shape to the other. Therefore, it is critical that the FSW classes of each deputy satellite have the updated information regarding the dynamics of the chief, and potentially other in the constellation.

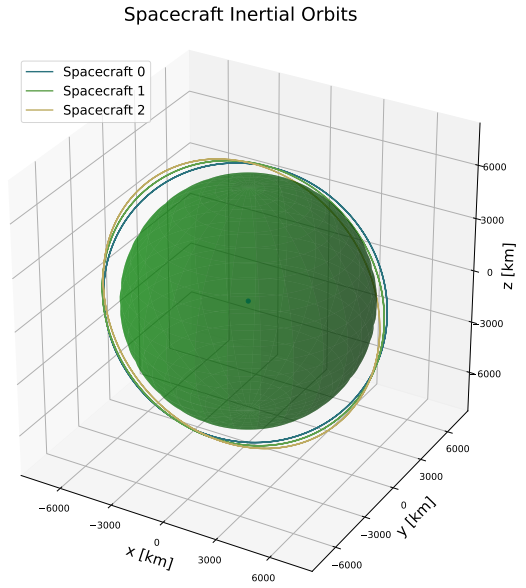

Fig. 5: Orbits for a three-spacecraft simulation around Earth.

Listing 5: Setup for a simulation around Earth.

```
self.set_EnvModel(BSK_EnvironmentEarth)
self.set_DynModel([BSK_MultiSatDynamics]*numberSpacecraft)
self.set_FswModel([BSK_MultiSatFsw]*numberSpacecraft)
```

6. Numerical Simulations

6.1 *Example Scenario*

As an illustrative example of the proposed architecture's capabilities, a three-spacecraft simulation around low-Earth orbit is created in Basilisk. The inertial orbits of all spacecraft are shown in Figure 5.

Here, an environment class with Earth as the main gravity body is added to the simulation. All spacecraft are homogeneous, which means they have the same dynamics and FSW classes. The code for the class setup is shown in Listing 5, where the number of spacecraft is set to `numberSpacecraft` = 3.

Let us now change the main gravity body to Mercury. Assuming an environment class with Mercury as the main body exists and is properly setup, chang-

Listing 6: Setup for a simulation around Mercury.

```
self.set_EnvModel(BSK_EnvironmentMercury)
self.set_DynModel([BSK_MultiSatDynamics]*numberSpacecraft
self.set_FswModel([BSK_MultiSatFsw]*numberSpacecraft)
```

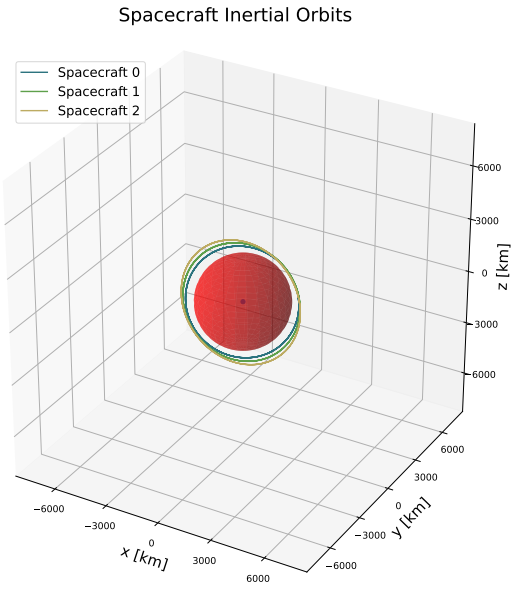

Fig. 6: Orbits for a three-spacecraft simulation around Mercury.

ing the environment requires little effort. In fact, the setup in the scenario script is almost identical to the one with the Earth environment, as shown in Listing 6

The inertial orbits for this scenario are shown in Figure 6. The scale of this plot is purposely the same as the one Figure 5 to show the different size of Mercury and, therefore, of the satellite's orbits. This happens because the spacecraft's initial conditions are set through orbital elements, with the major axis being proportional to the main body's equatorial radius. This allows the user to freely change the gravity body without worrying about the orbits intersecting the planet, keeping them in the low orbit regime.

For the final example, the environment class is set back to Earth as the main body. The number of spacecraft is increased to six, which is done through setting `numberSpacecraft` = 6. With no other change, a six-satellite simulation is created, together with dynamics and FSW modules for each spacecraft instance.

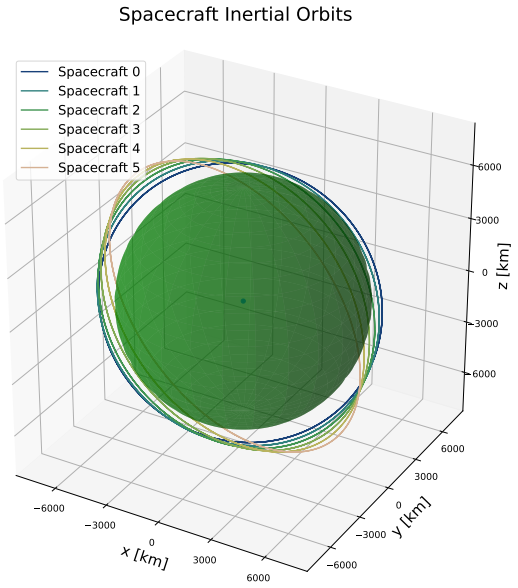

Fig. 7: Orbits for a six-spacecraft simulation around Earth.

Listing 7: Initial conditions loop.

```
for i in range(self.numberSpacecraft):
    self.oe.append(orbitalMotion.ClassicElements())
    self.oe[i].a = 1.1 * EnvModel.planetRadius +
    ↪  1E5*(i+1)  # meters
    self.oe[i].e = 0.01 + 0.001*i
    self.oe[i].i = 45.0 * macros.D2R
    self.oe[i].Omega = (48.2 + 5.0*i) * macros.D2R
    self.oe[i].omega = 347.8 * macros.D2R
    self.oe[i].f = 85.3 * macros.D2R
```

This includes attitude control system, power system, etc. The orbits for this scenario are shown in Figures 7. Since the initial conditions are set in a loop for all spacecraft, the software is able to adapt to any number of spacecraft that the user intends to simulate. This loop is shown in Listing 7.

6.2 *Multithreading Performance*

To show how the parallelization impacts performance, simulations using single and multiple threads are run with an increasing number of satellites. Performance is measured through the simulation time, which accounts for all initialization routines, as well as the simulation itself. The results are shown in Figure 8.

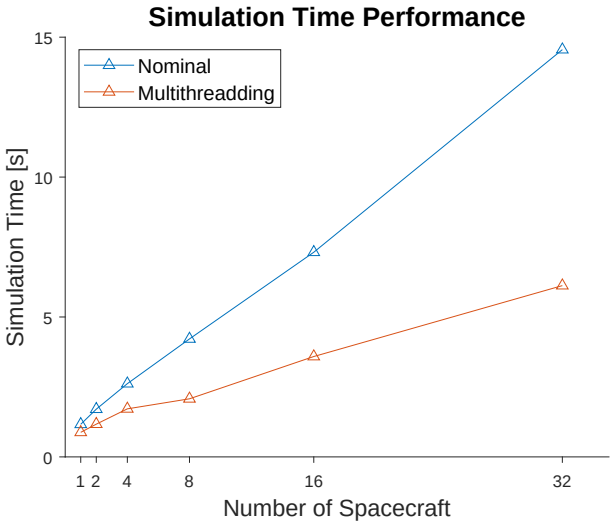

Fig. 8: multithreading performance in terms of simulation time. The time is averaged over 10 runs on a CPU with 16 threads.

As expected, the multithreading application consistently delivers faster simulation times when compared to the single-threaded scenario. More importantly, the slope of the multithreading curve is lower than for the single thread. This means that the additional simulation time incurred from adding more satellites is smaller, which is critical for simulations of tens or hundreds of satellites.

It should be noted that for this scenario a constellation of satellites is used, as opposed to a formation of satellites. The difference lies in the fact that no communications between spacecraft is present, nor is the shape of the constellation controlled. This means that there is no data sharing between different spacecraft. While data-sharing is possible to implement using multithreading, it is harder to do because all processes must be up to date across all threads before updating the modules that need that information. Failure to do so means that modules (mostly within the respective FSW classes) will run with incorrect information. Therefore, the current implementation is not multithread safe, although there are plans to implement that feature.

7. Conclusion

The increasing interest in launching formations of satellites to space requires software architectures that are capable of simulating multiple satellites. The intrinsic challenges of creating a multi-satellite simulation, such as the increased computation time and effort, mean that the software's architecture needs to be built around the ability to simulate multiple satellites at once.

In this work, the underlying principles of the architecture design (modularity, scalability, parallelization and scriptability) are presented to justify the design choices made. The architecture framework is presented, aiming to solve the drawbacks of a multi-satellite simulation. While the architecture is implemented around the Basilisk software tool, it is framed in a general enough way to be applied to any other software application.

The challenge of information sharing between different modules is tackled using the new messaging system, which is based on a peer-to-peer message connections. This system, combined with the proposed process and module design, is used to guarantee that each module has the most up-to-date data at each iteration.

The example scenarios show the ease of changing the simulation parameters after the architecture is put into place. This ease of scriptability is important when different simulation parameters are to be evaluated, as it speeds the process of changing the environment or the spacecraft itself. Although still in early development, the multithreading capabilities show promising speed increases, with the most notable changes when the number of simulated satellites is greatest.

8. Acknowledgments

Part of this research was supported under the NASA STTR Phase 1 grant No. 80NSSC21C0117.

References

[1] AGI. Systems tool kit (stk). `http://www.agi.com/products/stk`, April 2022.

[2] NASA. General mission analysis tool. `https://opensource.gsfc.nasa.gov/projects/GMAT/index.php`, April 2022.

[3] Patrick W. Kenneally, Scott Piggott, and Hanspeter Schaub. Basilisk: A flexible, scalable and modular astrodynamics simulation framework. *Journal of Aerospace Information Systems*, 17(9):496–507, Sept. 2020.

[4] Cody Allard, Manuel Diaz-Ramos, Patrick W. Kenneally, Hanspeter Schaub, and Scott Piggott. Modular software architecture for fully-coupled

spacecraft simulations. *Journal of Aerospace Information Systems*, 15(12):670–683, 2018.

[5] John Alcorn, Cody Allard, and Hanspeter Schaub. Fully coupled reaction wheel static and dynamic imbalance for spacecraft jitter modeling. *AIAA Journal of Guidance, Control, and Dynamics*, 41(6):1380–1388, 2018.

[6] Scott Carnahan, Scott Piggott, and Hanspeter Schaub. A new messaging system for basilisk. In *AAS Guidance and Control Conference*, Breckenridge, CO, Jan. 30 – Feb. 5 2020. AAS 20-134.

[7] Hanspeter Schaub and John L. Junkins. *Analytical Mechanics of Space Systems*. AIAA Education Series, Reston, VA, 4th edition, 2018.