JOURNAL OF AEROSPACE INFORMATION SYSTEMS
Vol. 17, No. 9, September 2020

Basilisk: A Flexible, Scalable and Modular Astrodynamics Simulation Framework

Patrick W. Kenneally,* Scott Piggott,† and Hanspeter Schaub‡
University of Colorado, Boulder, Boulder, Colorado 80309-0431

https://doi.org/10.2514/1.I010762

The Basilisk astrodynamics framework is a spacecraft simulation tool developed with an aim of strict modular separation and decoupling of modeling concerns in regard to coupled spacecraft dynamics, environment interactions, and flight software algorithms. Modules, tasks, and task groups are the three core components that enable Basilisk's modular architecture. These core components are described and their functionality demonstrated. The Basilisk message-passing system is a critical communications layer that facilitates the routing of input and output data between modules. Furthermore, this paper outlines Basilisk's data logging and Monte Carlo simulation functionality. The implementation of Basilisk's Python wrapped C++/C technology stack is described. Finally, a sample spacecraft attitude control simulation demonstrates the modularity and flexibility of the framework.

I. Introduction

SPACECRAFT simulation software tools are an indispensable part of modern spacecraft design processes. The continual increase in complexity of spacecraft mission and maneuver design, dynamical and kinematic design verification, and postlaunch telemetry analysis all heavily rely on software simulation tools. These simulation tools provide engineers with the ability to increase the quality of design and testing by reducing cost and duration of development. For example, proposed changes to a mission's configuration, parameter tuning, or in-flight anomalies may be explored via Monte Carlo simulation [1,2]. Additionally, hardware-in-the-loop (HWIL) testing allows for verification and validation of the spacecraft hardware and software systems in a controlled laboratory environment. Hardware-in-the-loop testing can expose technical faults and system integration problems, saving considerable project financial and personnel resources before launch to space. While there are both commercial and open source tools available that solve some of these challenges, there has not been an open astrodynamics software tool to address all.

Astrodynamics simulation tools can be broadly categorized into three groups: commercial off the shelf (COTS), government off the shelf (GOTS), and general open source. A number of tools had their origin in the GOTS category, and subsequently moved to the open source category. Popular tools to simulate either full missions, orbits, attitude motion, or flight algorithms (or that perform hardware and software in the loop) include the MATLAB/Simulink [3] combination to simulate algorithms and autocode to flight C code, Analytic Graphics, Inc. (AGI) Systems Tool Kit (STK) [4] to model orbital simulations and mission scenarios, a.i. FreeFlyer [5] to simulate spacecraft dynamics, NASA General Mission Analysis Tool (GMAT) [6] to perform orbital trajectory optimizations, NASA Trick [7] to simulate complex spacecraft physics, OreKit [8] to simulate spacecraft using open source Java libraries, Jet Propulsion Laboratory's (JPL's) Dynamics Algorithms for Real-Time Simulation (DARTS)/Dshell [9] software to simulate complex spacecraft behaviors and control solution using closed software, and NASA 42 [10] to simulate spacecraft with open software.

Each tool is developed with a specific subset of space asset simulation purposes in mind. For example, the OreKit, GMAT, and STK tools were initially developed with a focus on high-fidelity orbit dynamics, orbit estimation, orbit propagation, and trajectory design. As a result, these tools include a range of different propagators, complex multibody gravity models, drag, solar radiation pressure, and orbit determination tools. For example, the Orekit tool includes six optional methods to model atmospheric density ranging from simple exponential models to empirical predictive models such as the Marshall Solar Activity Future Estimation [11].

When assessing software packages in the context of their ability to simulate full spacecraft dynamics, it is important to identify how the dynamics are computed and the impact this has on the implementation's software architecture. For example, tools such as OreKit and STK have increased their ability to accommodate spacecraft attitude. STK can be paired with the SOLIS plug-in: a commercial plug-in that models spacecraft translational and attitude dynamics [12]. While the SOLIS plug-in enhances STK's spacecraft dynamics, it does not model disturbances that may alter the spacecraft's center of mass [13]. Similarly, OreKit models the spacecraft as a rigid body, and the dynamics are primarily focused on defining perturbations as uncoupled external forces and torques.

Two tools that do provide increased modularity, coupled dynamics, and the ability to customize the spacecraft dynamics are JPL's DARTS environment and NASA's "42" software package [10,14]. The DARTS tool uses spatial operator algebra for the development of multibody dynamics to generate a spacecraft system mass matrix in a form that is efficiently solved recursively [15]. The simulation package 42 allows for spacecraft composed of multiple rigid or flexible bodies using a tree topology to formulate the dynamics. Both of these formulations allow developers to add arbitrary models to the simulation without significant change to the code base.

It seems an unreasonable requirement to expect a tool, which simulates the high complexity of a spacecraft system, to accommodate all possible mission configurations and spacecraft subtleties as out-of-the-box modeling functionality. On this basis, it is reasoned that the facility to extend a tool with modular additions via means of scripting and custom code development is needed to allow engineers to adapt the tool to the particular specifications and requirements of their mission. All of the tools listed include some basic level of scriptability, whereas others enable significantly more customization. For example, AGI's STK offers their Connect and Object Model APIs, which facilitate the addition of custom simulation models (except for coupled spacecraft dynamics). In contrast, JPL's DARTS tool allows a user to compile and add a custom model to any part of the simulation framework. This may include a model of the flexible

Received 28 May 2019; revision received 14 February 2020; accepted for publication 18 February 2020; published online 20 May 2020.

*Graduate Research Assistant, Department of Aerospace Engineering Sciences, 431 UCB, Colorado Center for Astrodynamics Research. Student Member AIAA.

†ADCS Integrated Simulation Software Lead, Laboratory for Atmospheric and Space Physics.

‡Professor, Glenn L. Murphy Endowed Chair, Department of Aerospace Engineering Sciences, 431 UCB, Colorado Center for Astrodynamics Research. Fellow AIAA.

dynamics of a large solar panel boom or the addition of a simulated ground station.

While it is not surprising that none of the COTS tools use a version of an open source license, it is interesting to note that COTS tools are typically not cross-platform, in so far as they often support installation on just one or two of the primary operating systems: macOS, Windows, and Linux. This limits the reach of the tool to research organizations that are already using a particular operating system.

Finally, a large feature that is not available by default in most tools is HWIL and software-in-the-loop (SWIL) functionality. Of the tools listed, MATLAB/Simulink and the DARTS/Dshell tools support HWIL and SWIL functionality without significant modification. Hardware-in-the-loop and SWIL functionality allows engineers to use the same set of tools and flight algorithms through multiple phases of the mission and in multiple engineering teams across an organization.

Basilisk[§] is a novel astrodynamics framework that simulates complex spacecraft systems in the space environment. While many simulation tools possess overlapping features with Basilisk, no others possess the combined characteristics of Basilisk. Basilisk is a highly modular Python-user-friendly open-source simulation framework that provides accurate coupled vehicle position and attitude dynamics along with optional structural flexing, imbalanced momentum exchange device, and fuel slosh dynamics with at least a 365-time speedup (one mission year in one compute time day). Basilisk can be used to support engineering activities during the early mission design phases, detailed design and validation phases, and postlaunch telemetry analysis and spacecraft command sequence validation phases. Modeling fidelity in Basilisk is configurable by a simulation engineer's choice of model and algorithms, as well as by the prescription of the time step at which each model is integrated. The software has also been used in distributed hardware-in-the-loop simulations.

This paper describes the Basilisk framework and the underlying message-passing system that enables a very modular approach to open spacecraft simulation development. In Sec. II, the Basilisk framework's software stack is introduced. Section III gives a detailed account of the aforementioned novel Basilisk system architecture, which allows for the rapid development of a simulation for of a wide variety of complex spacecraft systems. The key architectural components discussed are the Basilisk message system that facilitates data passing between models and the Basilisk spacecraft dynamics implementation. Section IV outlines the simulation execution flow of control and how the fundamental components of modules, tasks, and task groups work together to provide the user with flexible control over simulation design, integration rates, and message passing. Sections V and VI provide an overview of the Basilisk multiprocessing Monte Carlo tools and the ability to log, process, and analyze large (multi-gigabyte) datasets. In the final section of this paper, an example Basilisk simulation configuration will be presented to illustrate how Basilisk's modular design allows the end-user engineer to build a detailed simulation scenario from simple Basilisk building blocks.

II. Software Stack and Build

The core Basilisk architectural components and physics simulation modules are written in C++11 to allow for object-oriented development and fast execution speed. However, Basilisk modules can also be developed using Python for easy and rapid prototyping and C to allow flight software modules to be easily ported directly to flight targets. There exists a range of space environment models, such as the various planetary global reference atmospheric models, which are written in FORTRAN. Simulation developers can implement a simple C-code wrapper to translate the execution control and data flow to and from a module, thereby transparently integrating their FORTRAN code base as a Basilisk module.

Whereas Basilisk modules are developed in a number of programming languages, Basilisk users interact with and script simulation scenarios using the Python (2.6/7 and 3.X) programming language. Python bindings are available for all modules and supporting simulation utilities, as indicated in Fig. 1. There are many approaches that can be used to produce Python bindings of the C/C++ library code. Bindings can be created either manually or autogenerated as a stage in the software build process. Handwritten bindings can be constructed using the Python C API, the Python `ctypes` module, or the C foreign function interface for Python. While these methods are viable solutions for exposing small portions of C/C++ code at the Python layer, it is judged that the handwritten approach requires too much initial development time and ongoing maintenance effort when exposing entire component interfaces. As a result, the Python bindings are autogenerated and the following broad criteria were used to conduct a trade to assess and select candidate tools. The tool should 1) be compatible with Windows, Linux, and MacOS operating systems; 2) be able to generate bindings for both C and C++ code; 3) minimize the lines of supporting code needed to produce and facilitate binding generation; and 4) target multiple languages (at a minimum, all current Python versions).

Technologies that facilitate generating bindings include Software Interface Generator (SWIG),[¶] Cython,[**] and Boost Python.[††] Additional technologies exist but have been omitted from this list because they fail to support the first criteria.

The Cython language is a superset of the Python language that supports calling C functions and declaring C data types. The primary assumption when adopting a Cython approach is that the application is primarily written in Python and the Cython language is used in places to either increase performance or wrap and interleave portions of C/C++ code. When using SWIG or Boost Python, the primary assumption is that the application or library being developed is primarily C/C++ code. From the outset, it was decided that a key requirement of Basilisk's was computational speed and that achieving such speed would require the majority of the framework's components be written in C/C++. As a result, a Python interface was needed primarily to provide ease of configuration and scripting of the simulation. Ultimately, Python was not to be part of the code that controlled the run loop and models; and for this reason, Cython was excluded.

Whereas SWIG is a code generator, the Boost Python library is an interface library. A code generator will map the C/C++ interfaces and automatically generate the Python wrappers. An interface library requires the developer to explicitly declare the mapping between C++ and Python by writing additional code. To reduce the burden of writing this additional mapping code, one may employ a separate Python bindings generator tool such as Py++[‡‡] in tandem with Boost Python. Historically, Boost Python has provided more complete coverage of mapping language features from the C++ Standard Library. However, contrary to Boost Python, the SWIG project possesses an active developer community that has remedied such shortcomings. Finally, given that SWIG provides the option to generate wrappers for other target languages, there is the possibility that the Basilisk framework can also be provided in other target languages such as Java.

Python bindings for each core component and module are autogenerated by SWIG at compile time. A minimal module is defined by a header file (.h), a source file (.c/cpp), and a SWIG interface file (.i). The SWIG interface file contains compiler directives, which at compile time are parsed and map the class interface (.h) to a class interface defined in the target language (Python). At compile time, three build products are produced for each module's compilation. These three build products are a module library (.so or .dll), a Python interface to the underlying library (.py), and a Python-to-source language translation file (.cxx). The Python interface mirrors the underlying C++ class' variables and functions. The Python bindings allow users to employ the module's functionality within the Python environment through the typical package import mechanism, as demonstrated in Listing 1.

[§]Data available online at https://hanspeterschaub.info/bskMain.html [retrieved 2020].

[¶]Data available online at http://www.swig.org [retrieved 2020].

[**]Data available online at https://cython.org [retrieved 2020].

[††]Data available online at https://www.boost.org/doc/libs/1_70_0/libs/python/doc/html/index.html [retrieved 2020].

[‡‡]Data available online at https://pyplusplus.readthedocs.io [retrieved 2020].

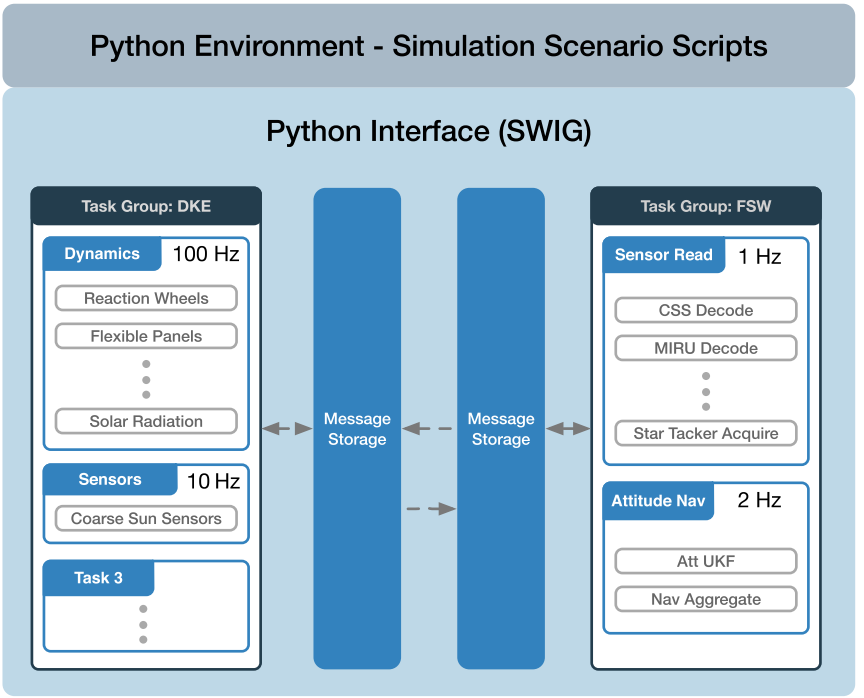


Fig. 1 An example layout of a complete Basilisk simulation where each element of the system has SWIG-generated Python interfaces available in the Python environment.

Listing 1: Selective Python imports of Basilisk modules:

```
from Basilisk.simulation import reactionWheelStateEffector, rwVoltageInterface, simple_nav, spacecraftPlus
```

The Basilisk architecture demonstrates a particular form of decoupling software components known as the dependency inversion principle [16]. In the context of generating target language binding for each module in the build process, conventional object-oriented design would have higher-level components of Basilisk directly link against the lower-level module libraries. Rather, this dependency is inverted by introducing a number of interfaces (described in the following section) to which modules and core Basilisk components adhere. As such, generating a separate SWIG-wrapped library object (.so or .dll) for each module results in neither compile time dependencies between one module and another nor a module and Basilisk core components. This low coupling relieves the user of needing any software compilation knowledge and provides the ability to rapidly develop and reconfigure a simulation scenario solely at the Python language level of the technology stack. This critical feature allows a spacecraft simulation to be modified later by selectively replacing modules without impacting other simulation modules or requiring them to be revalidated.

III. Modularity in Basilisk

The types of missions that Basilisk can be used to simulate span a spectrum from Earth-orbiting CubeSats to interplanetary probes and spacecraft constellations. The hallmark of the Basilisk framework is its emphasis of the low coupling and high cohesion software design principles [17]. Coupling in this context is measured as the number and complexity of contact points that one module of code must have with another module of code. Cohesion is defined as the complexity within a module of code and is often determined by the clarity of responsibility of that module. In pursuit of the low coupling and high cohesion design ideal, Basilisk implements only two core system components: the Basilisk message system and Basilisk simulation controller. Basilisk's low coupled design is achieved by three key design choices. The first is the complete decoupling of model and run loop dependence. The second design choice is to use a messaging system approach to manage module input and output data and intermodule data requirements. The message-passing paradigm supports a component-based development (CBD) approach [18]. The CBD approach shifts a simulation engineer's focus from low-level code development to scripting and composition of already compiled units of code. The third design choice is a novel method for managing the mathematically, fully coupled nature of a spacecraft rigid-body dynamics in a computationally efficient manner [19].

A. Components

Spacecraft onboard computers typically employ real-time operating systems that execute algorithms at both a fixed rate and within a fixed allocation of time. Similarly, a dynamic simulation employs either a fixed or variable time-step integration of the equations of motion (EOMs). Both of these time rate-driven processes motivate the conceptualization of the core Basilisk components introduced in this section. The result is a unique flexibility and configurability of a Basilisk simulation scenario.

A Basilisk simulation is built up from modules, tasks, and task groups. These fundamental abstractions are depicted in their conceptual relationship to each other in Fig. 2. A Basilisk module is standalone code that typically implements a specific model (e.g., an actuator, sensor, and dynamics model) or self-contained logic (e.g., translating a control torque to a reaction wheel (RW) command voltage). The Basilisk design encourages module developers to exemplify the high-cohesion design principle by providing the highly flexible connection of input and output data flows through messages. A module receives input data as message instances by subscribing to desired message types available from the messaging system. Similarly, a module publishes output data as message instances to the messaging system.

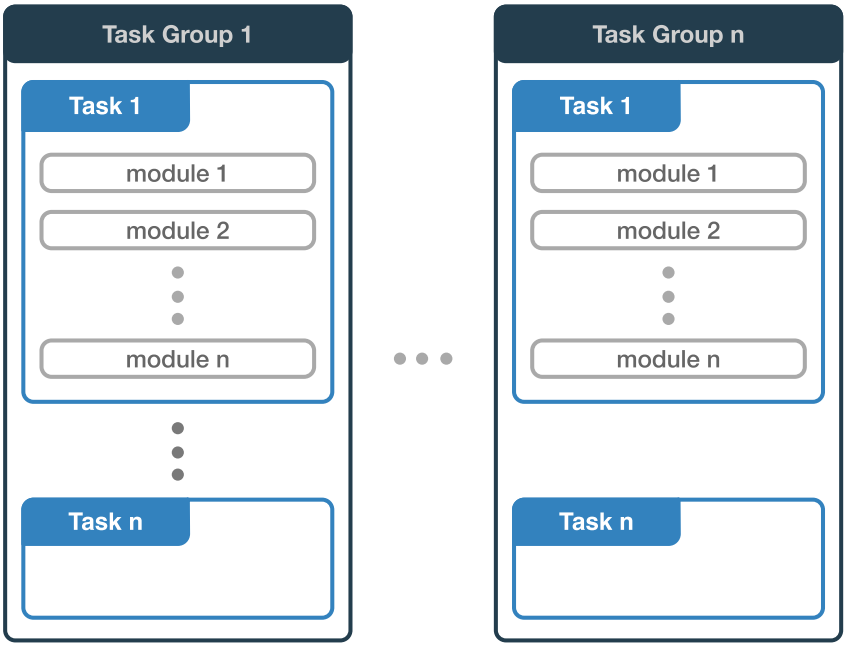


Fig. 2 Basilisk task group, tasks, and module layout.

Tasks are groupings of modules. Each task has an individually set integration rate. The task integration rate directs the update rate of all modules assigned to that task. As a result, a simulation may group modules with different integration rates according to desired time-step fidelity. Furthermore, the configured update/integration rate of each task can be adjusted during a simulation to capture increased resolution for a particular phase of the simulation. For example, a user may increase the integration rate for the task containing a set of spacecraft dynamics modules, such as flexing solar panels and thrusters, in order to capture the high-frequency flexing dynamics and thruster firings during Mars orbit insertion. Otherwise, the integration time step may be kept to a longer duration during the less dynamically active mission phases such as interplanetary cruise.

The execution of a task, and therefore the modules within that task, is controlled by either enabling or disabling the task. A task's enabled status can be toggled, any time during a simulation, from within a Python layer simulation script. Toggling a task's enabled status is particularly useful for enabling or disabling flight software (FSW) specific modules contained within a task related to the simulated spacecraft's FSW mode, e.g., safe mode or sun pointing.

Task groups are the highest-level grouping of Basilisk components. Task groups act as a container for tasks and provide a mechanism for resolving message dependencies between modules as discussed in greater detail in Sec. III.B. Task groups can be considered silos of tasks and the messages published and subscribed by modules within the task group. Figure 3 shows the relationships between the described Basilisk components as a unified modeling language (UML) diagram. The `SimModel` class is the application root and controls system initialization, simulation time stepping, and orchestration of the simulation. As previously discussed with reference to Fig. 2, the `SimModel` class has references to one or more instances of the `TaskGroup` class, which each have references to one or more instances of the `Task` class. Each instance of a `Task` maintains one or more references to an instance of a `Module` class, which inherits from the `SysModule` parent class. The `SysModule` parent class encapsulates core `Module` data and behavior: most importantly, the `moduleID`.

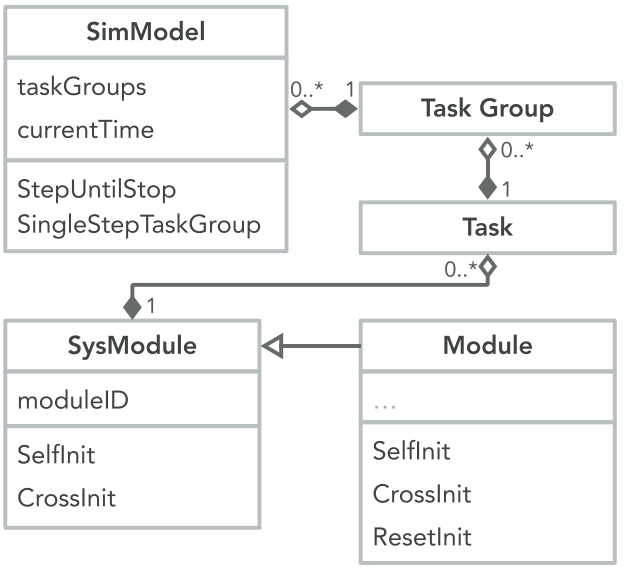


Fig. 3 Basilisk task group, tasks, and module class UML.

B. Message System

The Basilisk messaging system facilitates the input and output of data between simulation modules. The messaging system decouples the data flow between modules and task groups and removes explicit intermodule dependency, resulting in no run-time module dependencies.

A Basilisk module reads input message instances and writes output message instances to the Basilisk messaging system. The message system acts as a message broker for a Basilisk simulation. The messaging system employs a publisher–subscriber message-passing nomenclature, as shown in Fig. 4. A single module may read and write any number of message types. A module that writes output data, registers the "publication" of that message type by creating a new message-type entry within the message system. Similarly, a module that requires data output by another module subscribes to the message type published by the other module. The messaging system then maintains the message types read and written by all modules and the network of publishing and subscribing modules.

A message type is defined by a unique message name, a message identifier (ID), and a payload data structure (typically a C/C++ struct). The messaging system maintains metadata for each message type in a message header. The message-type header metadata include a list of allowed message publishers, subscribers, buffer memory locations, and read/write statistics.

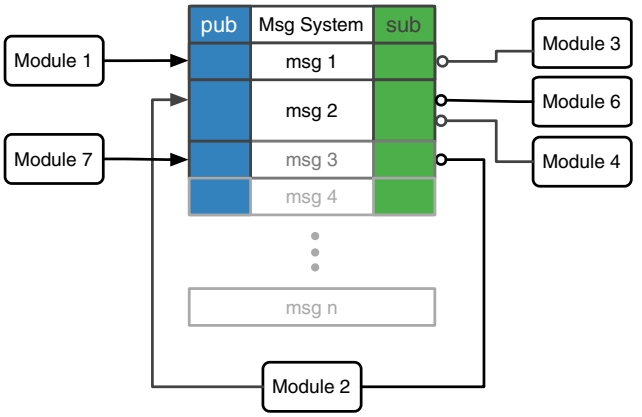


Fig. 4 Notional messaging system's publish and subscribe map for a message storage container of a single task group.

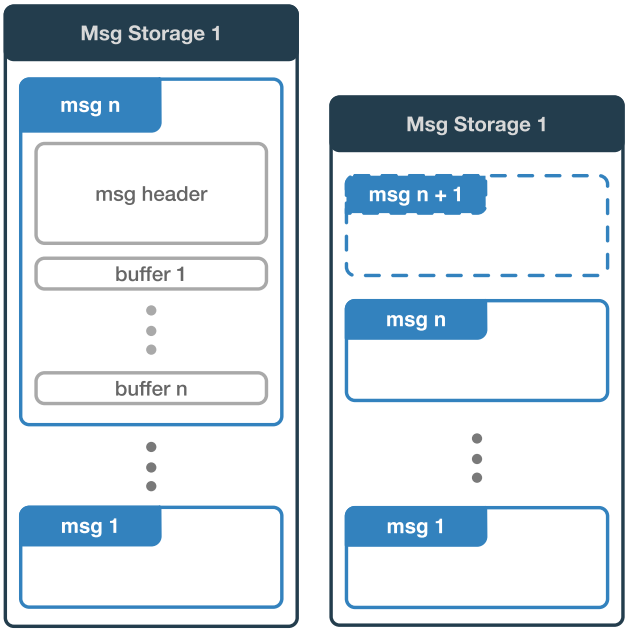
The messaging system implements the message storage as directly managed memory. As shown in Fig. 5a, a region of memory is allocated and managed as a the message storage container. The messaging system manages multiple storage containers: one for each task group. The size of the allocated memory for each storage container is determined by the combined size of the number of created message types, their associated headers, and the number of message instance buffers allocated for each message type. It is important to note that all message-type storage is at least double buffered in the messaging system. Ring buffer logic is used for message-type entries that are registered with more than two buffers. As shown in Fig. 5a, a module can declare to increase the number of buffers for a specific message type.

A message type is created in the message system when a module invokes the call, shown in Listing 2, on the SystemMessaging singleton. This function call takes a unique message name string, the maximum size in bytes of the message payload struct, the number of buffers into which an instance of the message type may be written, the type of the payload struct, and the identifier of the module creating the new message type. As demonstrated by Fig. 5b, the memory allocated in the task group's message storage container is increased, and existing message instances are moved within the allocated memory to accommodate the new message instance "`msg n + 1`." The messaging system returns to the module a message type with a unique message ID. It is this message ID with which a module will reference to write or read message instance data during runtime.

Listing 2: Register a new message with the messaging system:

```
SystemMessaging::CreateNewMessage(
    std::string messageName,
    uint64_t maxSize,
    uint64_t numMessageBuffers,
    std::string messageStruct,
    int64_t moduleID)
```

Simulation initialization and the associated resolving of message-type publish–subscribe pairs is discussed in greater detail in Sec. IV. However, the functionality of a task group interface is described as follows. A task group interface is a unidirectional message exchange from one task group to a second task group. Each task group has a single associated message storage container. This one-to-one design seeks to accommodate simulation configurations where one set of tasks is to remain isolated and separate from another set of tasks. An example of this is the desire to mimic a flightlike configuration where the dynamic and environment modules remain wholly separate from the flight software modules. This separation, while being useful to organize related modules within a simulation, becomes significantly useful when operating Basilisk as a distributed simulation across multiple compute resources. For example, in a SWIL configuration, the dynamics and environment modules execute on a desktop



a) Message system memory layout **b) Message system memory layout upon new message creation**

Fig. 5 Basilisk messaging (msg) system memory layout and organization.

computer, whereas the FSW executes on a separate flight target processor or processor emulator. A less stereotypical example is the case in which an analyst runs a computationally expensive module within a task group on more powerful remote computing resources (ray traced solar radiation pressure or high-degree gravity models) and the remaining modules in locally executing task group. As a result, to facilitate the exchange of messages between task groups, task group interfaces are available to make this connection. A task group interface is a unidirectional message exchange from one task group to a second task group. This allows for modules in a first task group to publish messages to a second task group and, as implied, allows modules in the second task group to subscribe to messages published in the first task group.

C. Dynamics Manager

The third and final piece of Basilisk's modular design is the implementation of the dynamics manager. The spacecraft dynamics are modeled as fully coupled multibody dynamics with the generalized EOMs being applicable to a wide range of spacecraft configurations. The implementation, as detailed in Ref. [19], uses a backsubstitution method to modularize the EOMs and leverages the resulting structure of the modularized equations to allow the arbitrary addition of both coupled and uncoupled forces and torques to a central spacecraft hub. The ability to add arbitrary dynamic effector modules to a spacecraft hub is a new architecture that sets Basilisk apart from similar tools. Take, for example, the highly capable and often used 42 spacecraft attitude and orbit dynamics simulation software. The 42 software includes many useful spacecraft dynamic models; however, an inspection of the code base will show that there exists, in multiple files, direct reference to data structures and semantic information regarding the assumed existence of dynamic models such as reaction wheels, magnetic torque bars, thrusters, etc. While it is sensible to assume the potential existence of these models in the simulation of a spacecraft, doing so explicitly introduces coupling across multiple pieces of code, which has the potential to increase code complexity and reduce maintainability and scalability. In contrast, the Basilisk dynamics engine assumes only the potential existence of dynamic effectors and state effectors. No explicit assumptions are made in code about the nature of these models, beyond their contribution to resolving the dynamical system of equations for the spacecraft. As a result, any conceivable dynamic interaction can be modeled as a Basilisk module and can be

applied to the spacecraft hub without code changes to any part of the dynamic engine [20].

A module that impacts the translational or rotational dynamics is called an effector. Effectors are classified as either a state effector or a dynamic effector. State effectors are those modules that have dynamic states to be integrated, and therefore contribute to the coupled dynamics of the spacecraft. Examples of state effectors are reaction wheels, flexible solar arrays, variable-speed control moment gyroscopes, and fuel slosh. In contrast, dynamic effectors are modules that implement dynamics phenomena that result in external forces or torques being applied to the spacecraft. Examples of dynamic effectors include gravity, thrusters, solar radiation pressure, and drag.

For a module to operate as either a state or dynamic effector, the implemented module class must inherit from the StateEffector or DynamicEffector parent classes. The developer of a dynamics module is responsible for implementing only the dynamics of the effector model. For a state effector, a developer must provide a custom implementation of the three functions shown in Listing 3. Listing 4 shows the single method for the noncoupled dynamic effector, for which a developer must provide a custom implementation.

Listing 3: StateEffector required methods:

```
// Provide contributions to the spacecraft's mass and inertia properties.
virtual void updateEffectorMassProps(double integTime);
// Provide coupled contributions to the back-substitution matrices.
virtual void computeStateContribution(double integTime);
// Compute the Module's own state derivatives.
virtual void computeDerivatives(double integTime,  Eigen::Vector3d rDDot_BN_N, Eigen::Vector3d omegaDot_BN_B,
↪  Eigen::Vector3d sigma_BN);
```

Listing 4: DynamicEffector required method:

```
// Compute the body or inertial frame force and/or torque due to the Effector.
virtual void computeForceTorque(double integTime);
```

The dynamics manager transparently organizes and aggregates the various dynamic contribution of each effector module in a simulation. It ensures all dynamic states are updated and propagated. The user may select from various numerical integration schemes to propagate the spacecraft dynamics. Moreover, the interface between the dynamics manager and the integrator has been generalized to allow other developers to implement their own desired numerical integration scheme.

IV. Execution Control

A Basilisk simulation steps through a number of distinct initialization, integration, and shutdown phases. The application flow of control for a Basilisk simulation is shown in Fig. 6. Basilisk modules, tasks, task groups, and their associated message storage and linkages are initialized by a two-stage process. Each Basilisk module inherits from the `SysModel` class. As shown in Listing 5, the `SysModel` abstract class defines an interface of four functions, which a module must implement. These functions are called on each module as part of the overall simulation flow of control process.

Listing 5: SysModel abstract class interface definition:

```
virtual void SelfInit();
virtual void CrossInit();
virtual void UpdateState(uint64_t currentSimNanos);
virtual void Reset(uint64_t currentSimNanos);
```

The two stages of simulation initialization are self-initialization and cross-initialization. During self-initialization, each module's `selfInit()` function is called, allowing a module to register the message types it intends to publish with the messaging system. Next, each module's `crossInit()` function is called, allowing a module to subscribe to message types that were made available as published messages in the previous self-initialization stage. As the last major step before beginning the run loop, `Reset()` is called for each module. The `Reset()` function provides each module an opportunity to set up to a "clean" known initial state.

The simulation flow of control is governed by three loop iterations. The outermost loop iterates through each of the instantiated task groups according to each task group's assigned priority level. Within each task group, each task is looped through. Subsequently, all modules within a task are iterated through according to their priority within the task. For each module in a task, the module's `updateState()` function is called. The logic contained in the `updateState()` function is custom to each module. However, a typical sequence of many `updateState()` implementations is to read subscribed input message types, perform a computation defined by the module, and then write published output message instances for use by other modules. Of particular importance is the special `SpacecraftDynamics` module, which implements the aforementioned dynamics manager. The `updateState()` of the `SpacecraftDynamics` module is responsible for triggering the dynamics integration process and, in doing so, determines the integration rate of the spacecraft dynamics.

Following the iteration through each of the task groups and task loops, the next call times for a task and task group are set. This is required because each task within a task group may have a different update rate and tasks may be enabled or disabled at various times during the simulation. As a result, the next call time for a task, task group, and therefore the modules can change from one loop to the next; and updating the next call time allows the simulation to skip forward to the next expected update time according to the combined task and task group update rates.

V. Data Logging

Data output by modules through messages or internal module variables (which have a declared `public` scope in their C++ class definition) may be logged. Data to be logged are determined before a simulation run where a user may specify complete message types, a single variable within a message type, or internal simulation variables to be logged and the logging rate desired. The highest logging frequency is driven by the highest frequency at which the task, containing the module producing the data, is executed. No interpolation is done for data logged at a frequency higher than the frequency at which data samples are produced. As shown in Fig. 6, the simulation data logger reads the requested messages and variables at the

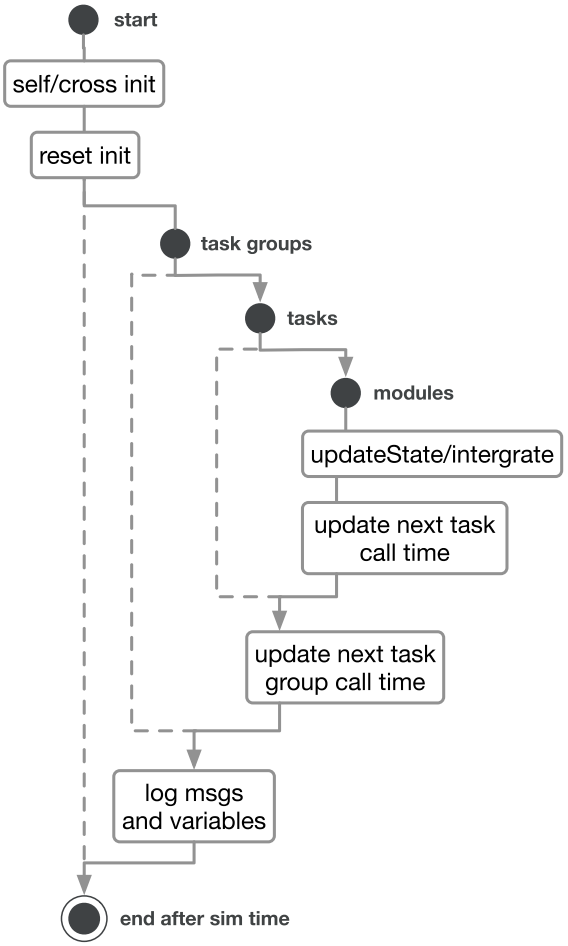


Fig. 6 Basilisk application flow of control for simulation (sim) execution.

end of each loop through all task groups. At the conclusion of the simulation, the user may retrieve the data with each message type and variable made available as a series of values associated with time stamps. This returned data format may be directly used in postprocessing scripts developed in Python using tools like NumPy and PANDAS.

VI. Monte Carlo Capability

A key benefit of Basilisk's Python interface is the ability to take any simulation script and, with minimal code changes, configure that script as a Monte Carlo simulation. A Monte Carlo simulation can be executed in a serial or multiprocessing fashion. As a multiprocessing execution, the simulation can be executed on multiple local CPU cores or a highly parallel remote execution environment. Additionally, the Monte Carlo functionality includes run-time-generated variable dispersions, logging and saving of simulation dispersed initial conditions, and logging of simulation data. The logged simulation data are made available in the portable Dataframes data structure from the PANDAS Python module.

Variable dispersions are built upon base Python implementations of scalar, vector, and tensor variable-type dispersion classes. Currently, Basilisk maintains uniform and normal dispersion for Cartesian variables, Euler angles, and modified Rodrigues parameter (MRP) attitude descriptions [21]. However, each of these individual base dispersions can be inherited by a user's custom dispersion implementation, allowing users to generate dispersions for variables with different physical bounds, variances, and specific statistical distributions.

The initial conditions, including the dispersed variables and random number seeds, are saved in a JavaScript Object Notation (JSON) file format for each Monte Carlo run. This allows a user to rerun and examine closer one or more particular runs of interest from a Monte Carlo simulation, with bit-for-bit repeatability.

Multiprocess capability is a key benefit of the Monte Carlo tools. The Monte Carlo controller uses the Python multiprocessing module to spawn and manage as many Python Basilisk simulation processes as the user or host machine allows. For example, a computer with a four-core CPU (each physical core with two virtual cores) will be used by the Monte Carlo controller as a machine with eight processors. The controller will launch eight simulations at once and continues to provide simulations to the worker pool of processes until all simulation work is complete. Each simulation execution is handled individually with data logging, initial conditions, and failures all logged for later analysis. Postprocessing of Monte Carlo data makes use of the convenient PANDAS statistical and data manipulation functions. While single simulation plotting is done with the more traditional Matplolib package, plotting of large multi-gigabyte datasets is achieved using the DataShaders plug-in to the Bokeh plotting library. This module employs a rasterized plotting approach to display, in a few seconds of execution, time plots containing extremely large datasets.

VII. Development Approach: Open Source

Basilisk's initial motivation was to support the design and development of the attitude determination and control system for an interplanetary spacecraft. The intention was to use Basilisk as an early mission phase A/B design and analysis tool, a flight algorithm verification and validation tool during latter phase C, and finally as the space environment and dynamics simulator for HWIL and SWIL testing during phase D. The referenced missions of phases A, B, C, and D correspond to the project lifecycle definitions found in the NASA Systems Engineering Handbook [22]. Basilisk has been used in all these mission phases. Basilisk's increasing utility has prompted the original development team at the Laboratory for Atmospheric and Space Physics (LASP) and the Autonomous Vehicle Systems (AVS) Laboratory to make the project available as an open source project. Basilisk uses an Internet Systems Consortium license, which is a permissive software license that simply requires attribution and relinquishes of the creator of liability [23]. It is anticipated that such a permissive license will help to encourage experimentation and contribution back to the main Basilisk project.

Basilisk has undergone an internal verification and validation effort within the LASP and the AVS Laboratory. Furthermore, the framework does not contain any export controlled components. Rather, all the included simulation and astrodynamics control algorithms are from open published literature. If a user needs to create modules that contain company proprietary tools or export controlled solutions, then the user would create these modules outside the regular Basilisk framework and import them separately in the Python simulation script. This allows Basilisk to model several common dynamical systems such as reaction wheels dynamics in a very general fashion, but no reaction wheel specific communication interfaces are included because these are vendor or mission specific.

Basilisk provides users with the ability to model and analyze spacecraft attitude control algorithms, vehicle rotational and translation dynamics, spacecraft trajectory modeling, optical navigation, validation of FSW algorithms (either SIL or HIL), and visualization and playback of spacecraft behavior. Priority features to be implemented include the addition of more accurate and configurable numerical integration schemes, type checking safety during message read and write operations, migration to a fully thread-safe implementation, and formalization of the existing two mechanisms for simulating multiple intercommunicating spacecraft.

VIII. Example Basilisk Attitude Control Simulation

Constructing a Basilisk simulation scenario requires the creation of task groups, assigning tasks to these task groups, and the instantiation of modules within each task. The following example demonstrates the key Basilisk function calls that configure a simple Earth-orbiting spacecraft whose attitude control system must align to a chosen inertial attitude. The simulation uses two task groups. The first task group

contains all dynamics, kinematics, and environment (DKE) modules. The second task group contains all flight software algorithm modules. The arrangement of task groups, tasks, and associated modules is presented in Fig. 7.

The modules within the DKE task group are separated into two tasks. In the first task, the modules included are the spacecraft hub, reaction wheels (fully coupled to the hub), and the Earth gravity field. In the second task, a module called SimpleNav is included. The SimpleNav module receives the spacecraft states through the output message type of the spacecraft module. The SimpleNav module perturbs the truth state of the spacecraft using a Gauss–Markov error model. For simple simulations, such as this example, the SimpleNav module is used in place of the more complex nominal spacecraft navigation system output. There is a single task in the FSW task group, and it contains all the modules required to implement an MRP inertial pointing controller using reaction wheels [21]. The FSW determines the attitude error by reading the navigation sensor output message type, whereas the reaction wheel motor torque module outputs a message type that drives the resulting reaction wheel assembly (RWA) dynamics.

Task groups and tasks are created and linked for the DKE and FSW task groups. This is done by creating a task group (also referred to as a process), and then adding a task to this task group as shown in Listing 6. The Dynamics task integration rate is set to 0.1 s, and the sensors task rate set to 0.5 s. In Basilisk, the base time scale is nanoseconds, and so the sec2nanos() conversion utility is used for convenience. Recall that task groups are message instance containers. Message types within a task group may only be published and subscribed to by modules within that task group. To facilitate message passing between task groups, a task group interface must be created for each of the desired message flow directions. In this simulation, it is desired that certain message types generated in the DKE task group are available to the modules in FSW, and vice versa. As shown in Fig. 7, two task group interfaces are created to facilitate the transparent exchange of message instances between the two task groups.

Listing 6: Simulation, task group, tasks, and task group message sharing interface instantiation:

```
# Instantiate simulation container
scSim = SimulationBaseClass.SimBaseClass()

# Create Task Groups (Processes)
dynProcess = scSim.CreateNewProcess("dynProcess")
fswProcess = scSim.CreateNewProcess("fswProcess")

# Create and add Tasks to each Task Group
dynProcess.addTask(scSim.CreateNewTask("dynTask", sec2nanos(0.01)))
dynProcess.addTask(scSim.CreateNewTask("sensorTask", sec2nanos(0.1)))
fswProcess.addTask(scSim.CreateNewTask("fswTask", sec2nanos(0.5)))

# Create interfaces and define message sharing directionality
intDynToFsw = sim_model.SysInterface()
intFswToDyn = sim_model.SysInterface()
intDynToFsw.addNewInterface("dynProcess", "fswProcess")
intFswToDyn.addNewInterface("fswProcess", "dynProcess")

# Add interfaces to Task Groups
dynProcess.addInterfaceRef(intDynToFsw)
fswProcess.addInterfaceRef(intFswToDyn)
```

With the core Basilisk structures instantiated, the next step is to populate the simulation with various Basilisk modules. As shown in Listing 7, the DKE modules are instantiated and assigned to their respective tasks. The `SpacecraftPlus()` module instantiates the rigid-body hub to which other `StateEffectors` and `DynamicEffectors` can be associated.

Listing 7: Instantiate DKE modules and assign to respective tasks:

```
# Create spacecraft hub effector
scObject = spacecraftPlus.SpacecraftPlus()
scSim.AddModelToTask("dynTask", scObject, None, 1)
# Create earth gravity body DynamicEffector
gravBodies = gravFactory.createBodies(['earth'])
scObject.gravField.gravBodies = spacecraftPlus.GravBodyVector(gravFactory.gravBodies.values())
# Create reaction wheel StateEffectors
RW1 = rwFactory.create('Honeywell_HR16', [1, 0, 0], maxMomentum=50., Omega=100.)
RW2 = rwFactory.create('Honeywell_HR16', [0, 1, 0], maxMomentum=50., Omega=200.)
RW3 = rwFactory.create('Honeywell_HR16', [0, 0, 1], maxMomentum=50., Omega=300.)
# Add reaction wheel StateEffectors to spacecraft object and Task
rwStateEffector = reactionWheelStateEffector.ReactionWheelStateEffector()
rwFactory.addToSpacecraft("ReactionWheels", rwStateEffector, scObject)
scSim.AddModelToTask("dynTask", rwStateEffector, None, 2)\
```

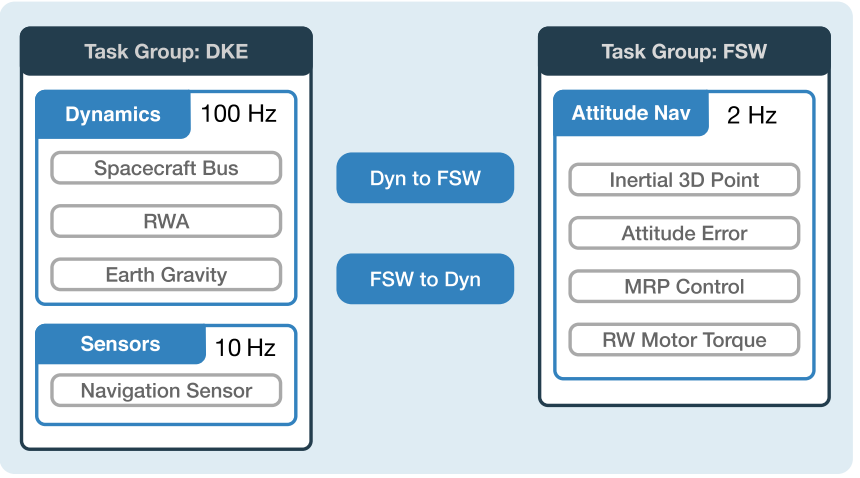


Fig. 7 Concept diagram of simple attitude feedback control Basilisk simulation configuration.

The FSW modules are created and populated in the FSW task group as shown in Listing 8. While modules can be developed in either Python, C++, or C, the FSW modules employed in this simulation are developed in C. Developing these modules in C allows analysts to run the same code and algorithm in simulation, SWIL, and eventually HWIL. The FSW modules included are 1) `Inertial3dPoint`, which computes the spacecraft reference attitude; 2) `AttitudeError`, which determines the spacecraft attitude error from the reference attitude; 3) `MRPControl`, which computes required control torques according to the MRP-based feedback control law; and 4) `RWMotorTorque`, which maps the attitude control torque onto a set of reaction wheel torque commands.

The novel utility of Basilisk's modularity is demonstrated by the arrangement of these FSW algorithm modules. Each of these FSW modules computes a specific kinematic or control-related quantity. As such, each module can be used as a building block to compose complex FSW behaviors. In this simulation scenario, four modules are used to create an inertial pointing control scheme. The movement of output data generated and the input data required by these modules are facilitated by published and subscribed message instances. Greater detail of the application and theory enabled by this building block approach is contained in Ref. [24].

To begin the simulation, three function calls are made. The first initializes the task groups, tasks, and modules by calling the `SelfInit()`, `CrossInit()`, and `ResetInit()` functions. Following this, the simulation stop time is set, and then the simulation is launched.

Listing 9: Launching a simulation:

```
scSim.InitializeSimulation()
scSim.ConfigureStopTime(simulationTime)
scSim.ExecuteSimulation()
```

External changes to the simulation configuration can be made via conditionally triggered events or, more simply, after a set duration of execution as demonstrated in Listing 10. This is useful to simulate specific spacecraft sequence instruction sets and FSW mode changes. Events can be set to trigger a custom user-provided function. This user-provided function allows an analyst to trigger and change any variable/state in the simulation that is available through the Python interface of each Basilisk module.

Listing 8: Instantiate FSW modules and assign to respective task:

```
# Setup the attitude determination Module.
inertial3DConfig = inertial3D.inertial3DConfig()
inertial3DWrap = scSim.setModelDataWrap(inertial3DConfig)
scSim.AddModelToTask("fswTask", inertial3DWrap, inertial3DConfig)

# Setup the attitude tracking error evaluation Module.
attErrorConfig = attTrackingError.attTrackingErrorConfig()
attErrorWrap = scSim.setModelDataWrap(attErrorConfig)
scSim.AddModelToTask("fswTask", attErrorWrap, attErrorConfig)

# Setup the MRP Feedback control Module.
mrpControlConfig = MRP_Feedback.MRP_FeedbackConfig()
mrpControlWrap = scSim.setModelDataWrap(mrpControlConfig)
scSim.AddModelToTask("fswTask", mrpControlWrap, mrpControlConfig)

# Setup the control torque to RW torque translation Module.
rwMotorTorqueConfig = rwMotorTorque.rwMotorTorqueConfig()
rwMotorTorqueWrap = scSim.setModelDataWrap(rwMotorTorqueConfig)
scSim.AddModelToTask("fswTask", rwMotorTorqueWrap, rwMotorTorqueConfig)
```

Listing 10: Spacecraft mode changes made after the simulation executes for a specified duration:

```
scSim.ConfigureStopTime(sec2nanos(20))
scSim.ExecuteSimulation()
# Command the FSW to go into safe mode and advance to ~ periapsis
scSim.modeRequest = 'safeMode'
scSim.ConfigureStopTime(sec2nanos(60))
scSim.ExecuteSimulation()
# Command the FSW to go into Nav only mode
scSim.ConfigureStopTime(sec2nanos(60 * 11 * 1 + 30))
scSim.modeRequest = 'navOnly'
scSim.ExecuteSimulation()
```

Plots are created from the simulation generated data using NumPy, Matplotlib, and PANDAS python packages. For the presented simulation, the evolution of the attitude is shown in Fig. 8. It is evident that the spacecraft controls to the reference attitude, with convergence achieved after 8 min. Figure 9 shows the computed control torques and the resulting actuated reaction wheel control torques. In this simulation, each reaction wheel's maximum available torque has been set as 0.2 N/m. As shown, reaction wheels 2 and 3 saturate their actuated torque early in the simulation. Finally, the resulting reaction wheel control speeds are shown in Fig. 10.

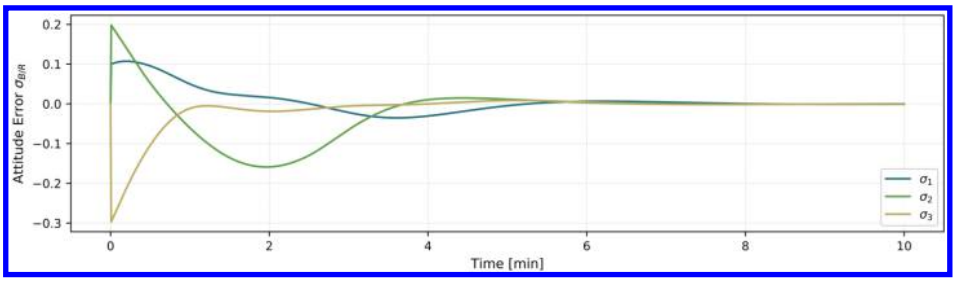


Fig. 8 Evolution of attitude error in each MRP component.

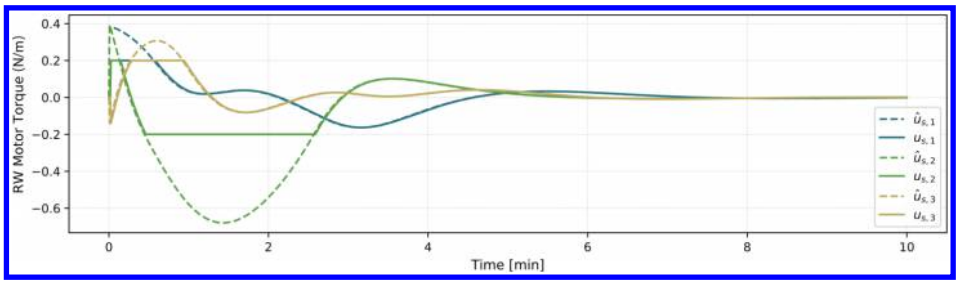


Fig. 9 Evolution of computed reaction wheel torques (dashed lines) and the actual reaction wheel torques.

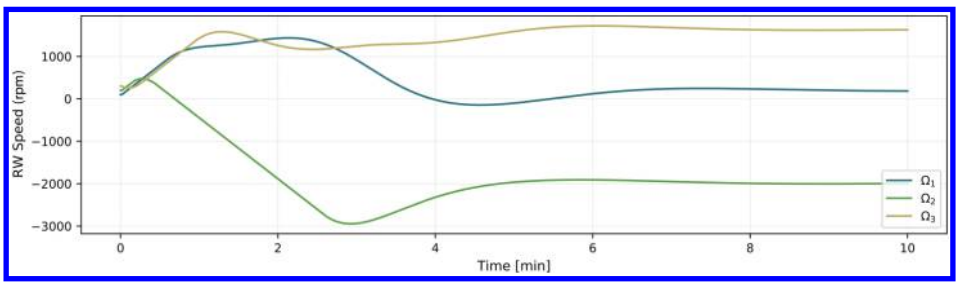


Fig. 10 Evolution of reaction wheel speeds.

IX. Example Basilisk Multibody Dynamics Orbital Simulation

The following simulation demonstrates Basilisk's ability to simulate a spacecraft trajectory under the influence of multiple gravity bodies. The simulation contains a single task group, containing a single task and seven modules. The Basilisk simulation models a single orbit of the Hubble Space Telescope's (HST's) trajectory and compares the simulated trajectory to both the HST trajectory SPICE [25] kernel (`hst_edited.bsp`[§§]) and to a GMAT-simulated trajectory of the same initial conditions and dynamic environment. The Basilisk SPICE module outputs a message for the ephemeris of each gravity body object at each time step. Each gravity body reads its position and velocity ephemeris message. As shown in Listing 11, a number of gravity bodies

[§§]Data available online at https://naif.jpl.nasa.gov/pub/naif/HST/kernels/spk/aareadme.txt [retrieved 2020].

are created and the Earth gravity body is assigned to be the central body. Assigning Earth as the central body ensures that the spacecraft hub translational states are computed relative to the Earth as the `zeroBase` location of the coordinate frame. The Earth gravity body is set to employ a spherical harmonics gravity model where GRACE Gravity Model 03 (GGM03) provides the first 100 harmonic coefficients [26]. Finally, the gravity model is assigned to the spacecraft's `gravityField` reference. The SPICE module is configured as shown in Listing 12. The same gravity body factory convenience functions are used to set up the gravity bodies are used to configure the SPICE module. The SPICE module is also configured with Earth as the zeroBase location of the coordinate frame in which the gravity bodies are defined. Finally, the SPICE module is added to the simulation task.

Listing 11: Configuring multibody gravity:

```
gravBodies = gravFactory.createBodies(['earth', 'mars barycenter', 'sun', 'moon', "jupiter barycenter"])
gravBodies['earth'].isCentralBody = True
# set and configure Earth spherical harmonics gravity model
gravBodies['earth'].useSphericalHarmParams = True
simIncludeGravBody.loadGravFromFile(bskPath +'/supportData/LocalGravData/GGM03S.txt'
                                    , gravBodies['earth'].spherHarm
                                    , 100
                                    )
# attach gravity model to spaceCraftPlus
scObject.gravField.gravBodies = spacecraftPlus.GravBodyVector(gravFactory.gravBodies.values())
```

Listing 12: Configuring NAIF SPICE module:

```
gravFactory.createSpiceInterface(bskPath +'/supportData/EphemerisData/', timeInitString)
gravFactory.spiceObject.zeroBase = 'Earth'
# add spice interface object to task list
scSim.AddModelToTask(simTaskName, gravFactory.spiceObject, None, -1)
```

The spacecraft's initial position and velocity are set by querying a position and velocity from the HST SPICE kernel. The spacecraft trajectory is simulated for 100 min at a 1 s time step. The integrator used for the Basilisk simulation is a fourth-order Runge–Kutta (RK4) method, whereas a variable-step Runge–Kutta45 (RK45) is the simplest integrator available within the GMAT. The Basilisk-simulated position is compared with the SPICE kernel position. The difference between the Basilisk-simulated inertial position and the HST SPICE position ephemeris is displayed in Fig. 11. The inertial position difference reaches a maximum of 856 m. The difference between the Basilisk-simulated inertial position and the GMAT-simulated inertial position is displayed in Fig. 12. It is evident that the GMAT's more accurate and configurable RK45 integrator produces less error in the spacecraft's propagation than Basilisk's default RK4. The GMAT propagation demonstrates close (within 100 m) errors, giving confidence that the Basilisk trajectory error is within an acceptable magnitude and attributable to the less capable fixed time-step RK4 integrator. Providing more advanced integration methods is a high-priority feature extension for future Basilisk development.

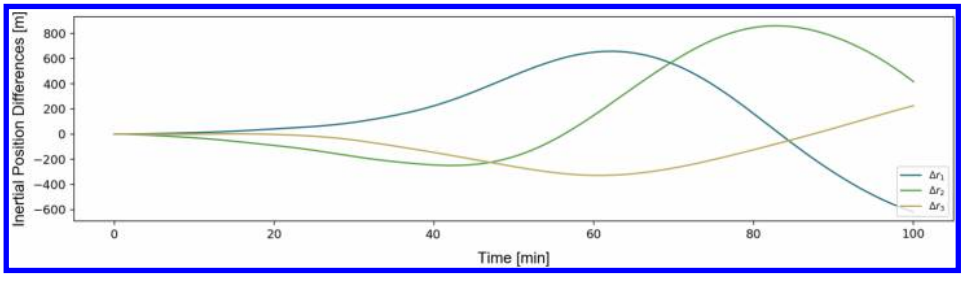


Fig. 11 Position error of Basilisk-simulated trajectory with respect to position as given by the HST SPICE kernel.

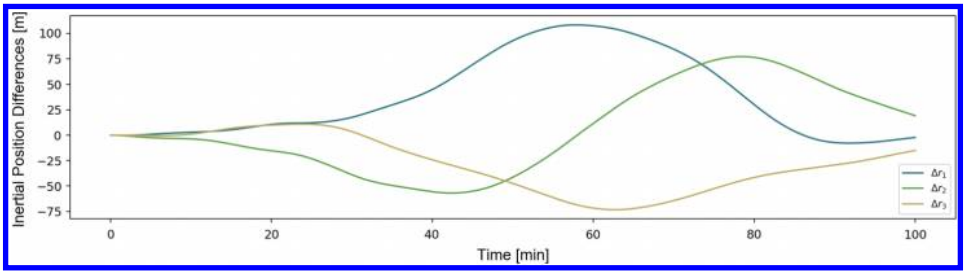


Fig. 12 Position error of Basilisk-simulated trajectory with respect to a GMAT-simulated HST trajectory.

X. Conclusions

The Basilisk astrodynamics framework provides a new open source alternative for fully coupled spacecraft dynamics mission simulation with integrated flight algorithm emulation. Among the suite of other available simulation tools, Basilisk provides an enabling mix of usability, extensibility, and computational speed. Basilisk is able to achieve this usability by providing a Python user interface for each component. The Python interface enables users to leverage the depth of the Python math and data analysis package ecosystems. Basilisk's modular architecture of modules, tasks, task groups, and the messaging system supports this usability by enabling users to configure simulation scenarios from the very simple early feasibility analysis to complex mission verification and validation.

References

[1] Schiff, C., and Dove, E., "Monte Carlo Simulations of the Formation Flying Dynamics for the Magnetospheric Multiscale (MMS) Mission," *Journal of Aerospace Engineering, Sciences and Applications*, Vol. 4, No. 4, 2012, pp. 66–78. https://doi.org/10.7446/jaesa.0404.06

[2] Dichmann, D. J., Alberding, C. M., and Yu, W. H., "Stationkeeping Monte Carlo Simulation for the James Webb Space Telescope," *24th International Symposium on Space Flight Dynamics*, Vol. 1, 2014, pp. 1–21.

[3] MATLAB/Simulink, "MathWorks," Software Package Version 9.5, Natick, MA, 2018, https://www.mathworks.com/products/matlab.html [retrieved 23 Oct. 2018].

[4] Systems Tool Kit, "Analytic Graphics," Software Package Version 11.4, Exton, PA, 2018, https://www.agi.com/products/engineering-tools [retrieved 20 Oct. 2018].

[5] FreeFlyer, "a.i. Solutions," Software Package Version 7.3, Lanham, MD, 2018, https://ai-solutions.com/freeflyer/ [retrieved 21 Oct. 2018].

[6] General Mission Analysis Tool, "NASA Goddard Space Flight Center," Software Package Version 17.4.0, Greenbelt, MD, 2018, https://software.nasa.gov/software/GSC-17177-1 [retrieved 27 Oct. 2018].

[7] "Trick Simulation Environment [online database]," NASA Johnson Space Center, Houston, TX, 2018, https://nasa.github.io/trick/ [retrieved 20 Oct. 2018].

[8] CS Systèmes d'Information, "OreKit Software Package Version 9.1: An Accurate and Efficient Core Layer for Space Flight Dynamics Applications," 2018, https://www.orekit.org [retrieved 15 Oct. 2018].

[9] DARTS Shell (Dshell), "Jet Propulsion Lab., DARTS Lab.," Software Package Version 2018, Pasadena, CA, 2018, https://dartslab.jpl.nasa.gov [retrieved 1 Oct. 2018].

[10] "42: A Comprehensive General-Purpose Simulation of Attitude and Trajectory Dynamics and Control of Multiple Spacecraft Composed of Multiple Rigid or Flexible Bodies"), Software Package, NASA Goddard Space Flight Center, Greenbelt, MD, Oct. 2018, https://software.nasa.gov/software/GSC-16720-1 [retrieved 1 Oct. 2018].

[11] Vaughan, W. W., Owens, J. K., Niehuss, K. O., and Shea, M. A., "The NASA Marshall Solar Activity Model for Use in Predicting Satellite Lifetime," *Advances in Space Research*, Vol. 23, No. 4, 1999, pp. 715–719. https://doi.org/10.1016/S0273-1177(99)00140-4

[12] Cuseo, J., "STK/SOLIS and STK/ODySSy Flight Software: Supporting the Entire Spacecraft Lifecycle," *Workshops on Spacecraft Flight Software*, Johns Hopkins Univ. Applied Physics Lab., Laurel, MD, 2011.

[13] "STK SOLIS: Commercial Plug-In to the Analytical Graphics, Inc (AGI) Systems ToolKit (STK)," Software Package, Advanced Solutions, Littleton, CO, Oct. 2018, http://www.go-asi.com/solutions/stk-solis/ [retrieved 1 Oct. 2018].

[14] Lim, C. S., and Jain, A., "Dshell++: A Component Based, Reusable Space System Simulation Framework," *Proceedings—2009 3rd IEEE International Conference on Space Mission Challenges for Information Technology, SMC-IT 2009*, IEEE, New York, 2009, pp. 229–236.

[15] Jain, A., and Rodriguez, G., "Recursive Flexible Multibody System Dynamics Using Spatial Operators," *Journal of Guidance, Control, and Dynamics*, Vol. 15, No. 6, 1992, pp. 1453–1466. https://doi.org/10.2514/3.11409

[16] Martin, R. J., *Agile Software Development, Principles, Patterns, and Practices (with Contributions from W. Newkirk and Robert S. Koss)*, 1st ed., Pearson, Upper Saddle River, NJ, Oct. 2002, pp. 94–98.

[17] Stevens, W. P., Myers, G. J., and Constantine, L. L., "Structured Design," *IBM Systems Journal*, Vol. 13, No. 2, 1974, pp. 115–139. https://doi.org/10.1147/sj.132.0115

[18] Sommerville, I., *Software Engineering*, 10th ed., Pearson, Upper Saddle River, NJ, 2015, pp. 464–490.

[19] Allard, C., Ramos, M. D., Schaub, H., Kenneally, P., and Piggott, S., "Modular Software Architecture for Fully Coupled Spacecraft Simulations," *Journal of Aerospace Information Systems*, Vol. 15, No. 12, 2018, pp. 670–683. https://doi.org/10.2514/1.I010653

[20] Allard, C. J., "Modular Software Architecture for Complex Multi-Body Fully-Coupled Spacecraft Dynamics," Ph.D. Thesis, Univ. of Colorado, Aerospace Engineering Sciences Dept., Boulder, CO, Aug. 2018.

[21] Schaub, H., and Junkins, J. L., *Analytical Mechanics of Space Systems*, 3rd ed., AIAA Education Series, AIAA, Reston, VA, 2014. https://doi.org/10.2514/4.102400

[22] Hirshorn, S. R., Voss, L. D., and Bromley, L. K., "NASA Systems Engineering Handbook," NASA TR SP-2016-6105, Feb. 2017.

[23] "ISC License [online database]," Open Source Initiative, Palo Alto, CA, 2018, https://opensource.org/faq [retrieved 15 Oct. 2018].

[24] Cols-Margenet, M., Schaub, H., and Piggott, S., "Modular Attitude Guidance: Generating Rotational Reference Motions for Distinct Mission Profiles," *Journal of Aerospace Information Systems*, Vol. 15, No. 6, 2018, pp. 335–352. https://doi.org/10.2514/1.I010554

[25] Acton, C. H., "Ancillary Data Services of NASA's Navigation and Ancillary Information Facility," *Planetary and Space Science*, Vol. 44, No. 1, 1996, pp. 65–70.

[26] Tapley, B., Ries, J. C., Bettadpur, S., Chambers, D., Cheng, M., Condi, F., and Poole, S., "The GGM03 Mean Earth Gravity Model from GRACE," *AGU Fall Meeting Abstracts*, Nov. 2007, Paper G42A-03.

J. P. How
Associate Editor