BASILISK: A FLEXIBLE, SCALABLE AND MODULAR ASTRODYNAMICS SIMULATION FRAMEWORK

Patrick W. Kenneally[*], *Scott Piggott*[†] *and Hanspeter Schaub* [‡]

University of Colorado, Boulder

ABSTRACT

The Basilisk astrodynamics framework is a spacecraft simulation tool developed with an aim of strict modular separation and decoupling of modeling concerns in regards to coupled spacecraft dynamics, environment interactions and flight software algorithms. Modules, Tasks, and Task Groups are the three core components which enable Basilisk's modular architecture. These core components are described and their functionality demonstrated. The Basilisk message passing system is a critical communications layer which facilitates the routing of input and output data between Modules. This paper outlines Basilisk's data logging and Monte Carlo simulation functionality. The implementation of the Basilisk, a Python wrapped C++/C technology stack is described. Finally, example simulation configurations and results are provided to demonstrate the modularity and flexibility of the framework.

Index Terms— simulation, modular, python, dynamics,

1. 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 post-launch 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. Additionally, hardware in the loop testing (HWIL) allows for verification and validation of the spacecraft hardware and software systems in a controlled laboratory environment. Such HWIL testing can expose technical faults and system integration problems saving considerable project financial and personnel resources before launching the system to space.

Basilisk is an astrodynamics framework that simulates complex spacecraft systems in the space environment. While many simulation tools possess overlapping features with Basilisk, none others possess the unique characteristics of Basilisk. These characteristics are that Basilisk is a highly modular, Python user-friendly, open-source simulation framework that provides sufficiently accurate (fidelity is configurable) coupled vehicle position and attitude dynamics, along with optional structural flexing, imbalanced momentum exchange device, and fuel slosh dynamics, with at least a 365 times speedup (one mission year in one compute time day). Furthermore, Basilisk is equally well employed during early mission design phases as it is later on during detailed design phases and further in post-launch telemetry analysis and spacecraft command sequence validation.

This paper describes the Basilisk framework in three primary sections. In Sec. 2 a brief survey of the current state-of-the-art commercial-off-the-shelf (COTS) and government-off-the-shelf (GOTS) spacecraft simulation software is given. Each of these COTS/GOTS systems posses unique feature sets. These unique feature sets predispose each tool excel when applied to different classes of spacecraft system analysis.

The remaining section of the paper are dedicated to an overview of the Basilisk framework. Beginning with Section 3, the Basilisk framework's software stack is introduced. Section 4 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 fundamental building blocks which govern simulation execution, the Basilisk message system which facilitates data passing between models, and the Basilisk spacecraft dynamics implementation. Section 5 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 their simulation design, integration rates and message passing. Sections 6 and 7 and provide an overview of the Basilisk multi-processing Monte Carlo tools and the ability to log, process, and analyze large (multi-gigabyte) data sets. In the final section of this paper a simple example Basilisk simulation configuration will be presented from the perspective

[*]Graduate Research Assistant, Smead Aerospace Engineering Sciences Department

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

[‡]Professor, Glenn L. Murphy Endowed Chair, Smead Aerospace Engineering Sciences Department

of the end user engineer to illustrate the basic functionality of Basilisk's modular design.

2. ASTRODYNAMICS SIMULATION TOOLS

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 naissance in the GOTS category and subsequently moved to the open source category. A list of popular tools to which are referred is given below.

- MATLAB/Simulink [1]
- Analytic Graphics Inc (AGI) Systems Tool Kit (STK) [2]
- a.i. FreeFlyer [3]
- NASA General Mission Analysis Tool (GMAT) [4]
- NASA Trick [5]
- OreKit [6]
- Jet Propulsion Laboratory (JPL), Dynamics Algorithms for Real-Time Simulation (DARTS)/Dshell [7]
- NASA 42 [8]

Each tool is developed with a specific subset of astrodynamics 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 multi body, gravity models, drag, solar radiation pressure and orbit determination tools. For example the Orekit tool includes six optional methods to model atmospheric density and therefore drag effects, from simple exponential models to empirical predictive models such as the Marshall Solar Activity Future Estimation [9].

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 how this impacts the modularity of the implementation. For example, tools such as OreKit and STK have increased their ability to accommodate spacecraft attitude. STK can be paired with the SOLIS plugin, a commercial plugin to the STK which models spacecraft translational and attitude dynamics. And while the SOLIS plugin enhances STK's spacecraft dynamics, it does not model disturbances which may alter the spacecraft's center of mass[10]. Similarly, OreKit models the spacecraft as a rigid body, and the dynamics are primarily focused on defining perturbations as external forces and torques.

Two tools which do provide increased modularity and the ability to customize the spacecraft dynamics are JPL's DARTS environment and NASA's "42" software package [11, 8]. 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 [12]. Similarly, 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 seem an unreasonable goal to expect a tool, which simulates the high complexity of a spacecraft system, to accommodate all possible missions configurations and spacecraft subtleties as out-of-the-box modeling functionality . On this basis, it is reasoned that extensibility of a simulation tool via means of scripting and custom code development is needed to allow engineers to adapt the tool to the particular specification and requirements of their mission. All of the tools listed include some basic level of scriptability while 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 completely custom model to any part of the simulation framework. This may include a model of the flexible 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 typically support installation on only one or two of the three big OS variants of macOS, Windows and Linux (MATLAB excepted).

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

3. SOFTWARE STACK AND BUILD

The core Basilisk architectural components and most modules are written in C++ to allow for object-oriented development and fast execution speed. However, Basilisk Modules can also be developed using Python, for easy and rapid prototyping, C (to allow flight software modules to be easily ported directly to flight targets) and Fortran (to accommodate legacy space environment models).

Whereas Basilisk Modules are developed in a number of programming languages, Basilisk users interact and develop simulation scenarios using the Python programming language. Python bindings are available for all Modules and supporting simulation utilities and core functionality as in-

dicated in Fig. 1. The Python bindings are auto-generated at build time using the Software Interface Generator (SWIG) tool. A typical Module is defined by a header file (.h) a source file (.c/cpp) and most importantly a SWIG interface file (.i). The SWIG interface file contains compiler directives, which at compile time are parsed and determine the class interface in resulting target language (Python). At compile time three build products are produced for Module compilation. These three build products are a library of the Module's source (.so or .dll), a Python interface to the underlying library (.py), and a Python-to-source language translation file (.cxx). The Python interface mirror 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 below in Listing. 3.

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

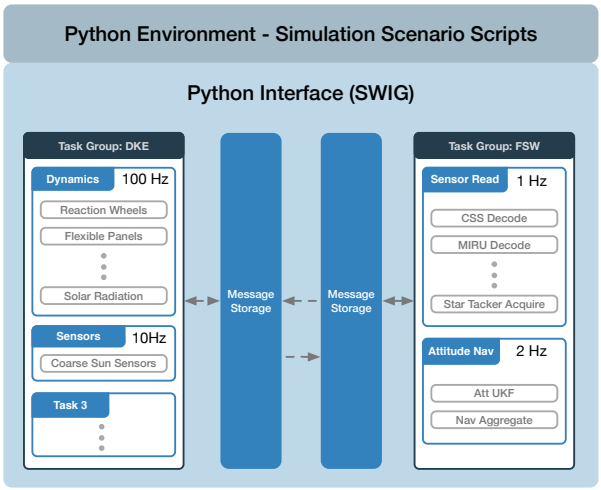


Fig. 1. An example layout of a complete Basilisk simulation where each element of the system has SWIG generated Python interfaces available in the Python environment.

The modularity enabled by the SWIG generated Python wrappers results in no compile time or run time dependencies between one Module and another. This modularity relives the user from needing any software compile knowledge and provides the ability to rapidly develop and reconfigure their simulation scenario solely in the Python language.

4. MODULARITY IN BASILISK

The types of missions that Basilisk can be used to simulate lay on a spectrum with earth orbiting cubesats at one end and interplanetary probes and spacecraft constellations at the other. The hallmark of the Basilisk framework is its highly modular system architecture. Modular design has been the guiding principle throughout Basilisk's development. The result is that Basilisk implements only two core system components, the Basilisk message system and Basilisk simulation controller. These two components are the only components required to begin building a Basilisk simulation scenario. Basilisk's modular 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 managing Module input and output data and inter-Module data requirements. Finally, a Dynamics Manager is implemented to manage the fully-coupled nature of a spacecraft rigid body dynamics.

4.1. Components

Spacecraft onboard computers typically employ real-time operating system which 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 (EOM). 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.

Basilisk's structure is built upon three components. These components are Modules, Tasks and Task Groups, and they are depicted in their relationship to each other in Fig. 2. A Basilisk Module is stand-alone code which 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 command voltage). Modules receive input data as messages by subscribing to desired messages available from the Messaging System. Similarly, a Module publishes output data as messages to the Messaging System.

Tasks are groupings of Modules. A Task has a set integration rate which directs the update rate of all Modules assigned to that Task. Each Task has an individually set integration rate. As a result a simulation may group modules with different integration rates according to desired fidelity. Furthermore, the set Task integration rate can be adjusted during a simulation to capture increased resolution for a particular duration. For example an user may increase the integration rate for the Task containing a set of spacecraft dynamics modules in order to capture the high-frequency dynamics of flexing solar panels and thruster firings during Mars Orbit Insertion (MOI). Yet the integration time step may be kept to a longer duration during the less dynamically active mission phases such as 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. This feature is particularly useful for enabling or disabling FSW focused Modules in a Task related to the simulated spacecraft mode e.g. Safe Mode, 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. 4.2. Task Groups can be considered silos of Tasks and the messages published and subscribed to by Modules within the Task Group.

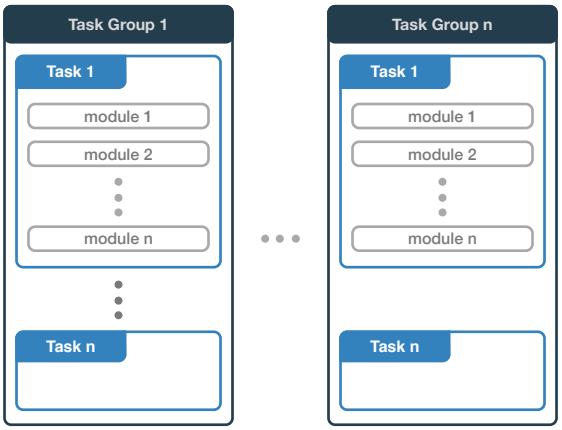


Fig. 2. Basilisk Task Group, Tasks and Module layout.

4.2. Message System

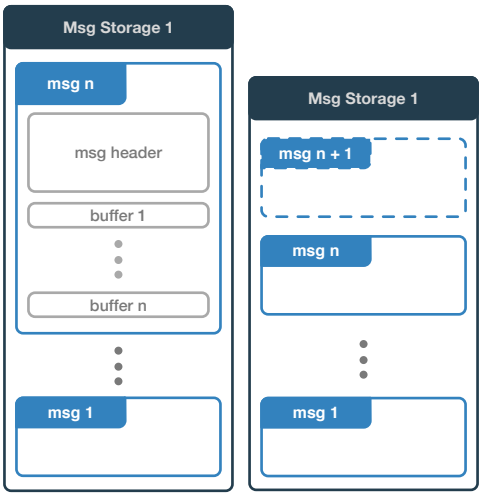
The Basilisk messaging system facilitates the input and output of data between simulation Modules. The messaging system decouples the data flow between Modules and Task Groups and removes explicit inter-Module dependency thus further supporting the modularity of the Basilisk architecture.

A Basilisk Module reads input messages and writes output messages to the Basilisk messaging system. The message system acts as a message broker for a Basilisk simulation.

The Basilisk messaging system manages the trafficking of messages and employs a publisher-subscriber message passing nomenclature. A single module may read and write any number of messages. A module that writes output data, registers the 'publication' of that message by creating a new message entry with the message exchange. Conversely, a module that requires data output by another Module subscribes to the message published by the other Module. The messaging system then maintains the messages read and written by all Modules and the network of publishing and subscribing Modules.

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

The messaging system implements the message storage as directly managed memory. As shown in Fig. 3(a) a region of memory is allocated and managed as 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 messages, their associated headers and the number of message buffers allocated for each message. It is important to note, that all messages are double buffered in the messaging system. For example, when a Module writes an updated message to the messaging system, the messaging system will alternate writing between the two buffers. This helps to protect data integrity during message writes and facilitates the, albeit rare, use case in which two modules must write a single message or when Basilisk operates in a multi-process/threaded configuration. However, as shown in Fig. 3(a), a Module can declare to increase the number of buffers for a specific message.



(a) Message system memory layout

(b) Message system memory layout upon new message creation

Fig. 3. Basilisk messaging system memory layout and organization.

A message is created in the message system when a Module invokes the call, shown in Listing 4.2, on the SystemMessaging singleton instance. This function call takes a unique message name, the maxSize in bytes of the message payload struct, the number of buffers into which an entry of the message may be written, the type of message payload struct and the identifier of the module creating the new message. As demonstrated by Fig. 3(b), the memory allocated is increased and existing message entries are moved within the allocated memory to accommodate the new message 'msg n + 1'.

```
SystemMessaging::CreateNewMessage(
    std::string messageName,
    uint64_t maxSize,
```

```
    uint64_t numMessageBuffers,
    std::string messageStruct,
    int64_t moduleID)
```

A Module 'creates' a message in the message system by passing a message payload type and a unique name and receives in return a unique message ID generated by the message exchange. It is this message ID with which a Module will reference its writing or reading of an updated message.

At simulation initialization a three stage process resolves the message subscription and publication pairs. Simulation initialization and the associated resolving of message pub-sub pairs is discussed in greater detail in Sec. 5. However, the functionality of a Task Group Interface (a unidirectional message exchange from one Task Group to a second Task Group) is described here. Each Task Group has a single associated message storage container. This one-to-one design seeks to accommodate simulation configurations 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 Module's execute on a desktop computer while the FSW executes on a separate flight target processor or processor emulator. However, there are further less stereotypical instances in which a simulation developer would like for messages in one Task Group to be available to Modules in a second 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, Modules in the second Task Group to subscribe to messages published in the first Task Group.

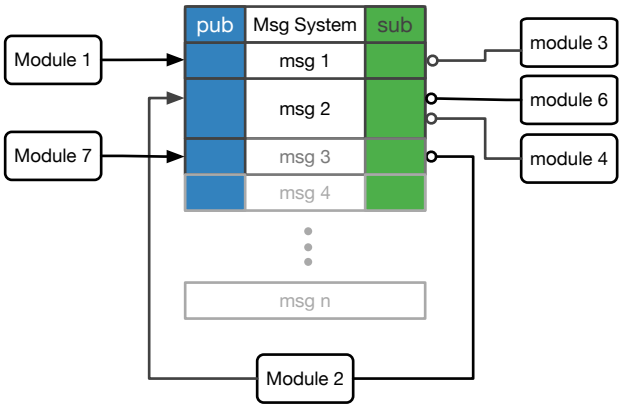


Fig. 4. A notional messaging system publish and subscribe map for a message storage container of a single Task Group.

4.3. 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 multi-body dynamics with the generalized EOMs being applicable to a wide range of spacecraft configurations. The implementation, as detailed in reference [13], uses a back-substitution 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 concept of an "Effector" is used to define Modules that model an effect on the spacecraft's dynamics. Effectors belong to one of two groups; either State Effectors or Dynamic Effectors. State Effectors are those Modules which have dynamics 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 (VSCMGs) and fuel slosh. In contrast, Dynamic Effectors are Modules which implement the phenomena that result in external forces or torques being applied to the spacecraft. Examples of Dynamic Effectors include: gravity, thrusters, solar radiation pressure (SRP) 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 following three functions;

- `updateEffectorMassProperties()` - provide contributions to the spacecraft's mass and inertia properties
- `updateContributions()` - provide coupled contributions to the back-substitution matrices
- `computeDerivatives()` - compute the Module's own state derivatives

For the non-coupled Dynamics Effector a developer must provide a single custom implementation for;

- `computeBodyForceTorque()` - compute the body or inertial frame force and/or torque due to the Effector.

The Dynamics Manager transparently organizes and aggregates the various dynamic contribution of each Effector Module in a simulation ensuring 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.

5. EXECUTION CONTROL

A Basilisk simulation steps through a number of distinct initialization, integration, and shut down phases. The high level flow of control for a Basilisk simulation is shown in Fig. 5.

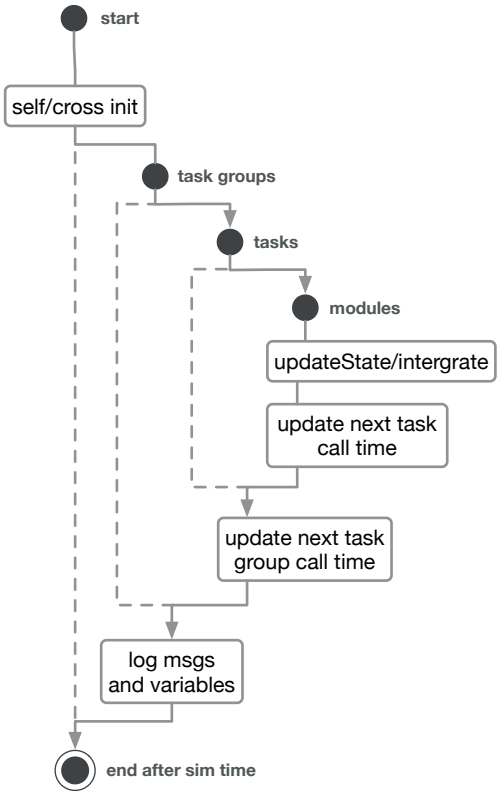


Fig. 5. Basilisk high level flow of control for simulation execution.

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` 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.

```
1    virtual void SelfInit();
2    virtual void CrossInit();
3    virtual void UpdateState(uint64_t
    currentSimNanos);
4    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 perform any internal setup and register the messages it intends to publish with the messaging system. Next, during cross initialization each Module's `crossInit()` function is called allowing a Module to subscribe to messages that were made available as published messages in the previous self initialization stage.

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 within each Task, all Modules within a task are iterated through according to their priority within their 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 common sequence of many `updateState()` function is to read subscribed input message, perform a computation defined by the Module and then write published output messages 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 Group and Task loops, the next call time for a Task and Task Group are set. This is required because each Task with in a Task Group may have a different update rates and Tasks may be enabled or disabled at various times during the simulation. As a result, the next call time for a Task and Task Group and therefore the modules can change from loop to loop and updating the next call time allows the simulation to skip forward in time according to the combined Task update rates.

After all Tasks and Modules within a Task Group have been updated the message logger copies all messages and Module variables and saves those for that time step.

6. 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 is determined prior to a simulation run where a user may specify complete messages, a single variable within a message 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. 5, the simulation Data Logger reads the requested messages and variables at the end of each loop through all Task Groups. At the con-

clusion of the simulation the user may retrieve the data with each message and variable made available as a time stamped series. This returned data format may be directly used in post processing scripts developed in Python (Numpy, PANDAS).

7. MONTE CARLO

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. The Monte Carlo simulation can be executed in a serial of multi-processing fashion. As a multi-processing 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 is 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 both, Cartesian variable, Euler angle, and MRP descriptions. However, each of these individual base dispersion can be inherited by a user's custom dispersion implementation allowing users to generate dispersions for of 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 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.

Multi-process capability is a key benefit of the Monte Carlo tools. The Monte Carlo controller uses the Python module named Multiprocessing to spawn and manage as many Python Basilisk simulation processes as the user or host machine allows. For example a computer with a 4 core CPU, each with two virtual cores will be used by the Monte Carlo controller as a machine with 8 processors. The controller will launch 8 simulations at once and continue to provide simulations to the worker pool of processes until all work is done. Each simulation execution is handled individually with data logging, initial conditions and failures all logged for later analysis. Post processing 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 data sets is achieved using the DataShaders plugin to the Bokeh plotting library. This module employs a rasterized plotting approach allowing for the generation plots of extremely large data set in a matter of seconds.

8. DEVELOPMENT APPROACH - OPEN SOURCE

Basilisk's naissance is in the support of 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 flght 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. Basilisk has been utilized in all these mission phases. Basilisk's increasing utility has prompted the original development team at LASP and the AVS Lab to make the project available as an open source project. Basilisk uses an Internet System Consortium (ISC) License which is a permissive software license simply requiring attribution and relinquishing the creator of liability [14]. It is anticipated that such a permissive license will help to encourage experimentation and contribution back to the main Basilisk project. Basilisk is available for download from bitbucket. The project employs a gitflow [15] development process where decisions about architecture changes and release cycles occur within the combined AVS Lab and LASP team.

Basilisk has undergone an internal verification and validation effort within LASP and the AVS Lab, and by comparison to flight data from previous missions. As an open source project, Basilisk will benefit from the strong ongoing engagement from a wide community of users. The community shall provide further validation, bug fixes and functionality additions. In the short time that Basilisk has been openly available a number of fixes and functionality additions have been made.

9. EXAMPLE BASILISK SIMULATION CONFIGURATIONS

Constructing a Basilisk simulation scenario requires the creation of Task Groups, assigning Tasks to these Task Groups, and the instantiation of desired Modules within the desired Task.

The following example demonstrates the important Basilisk function calls which configures a simple Earth orbiting spacecraft with multiple gravity bodies and interacting with a SPICE interface governing the bodies' positions and the spacecraft's initial state. The spacecraft initial position and velocity are extracted from the SPICE ephemeris file for the Hubble Space Telescope. A conceptual simulation configuration is presented in Fig. 6 showing the Modules, Task, and Task Group configuration.

The first function call sets up a new Basilisk simulation.

```
scSim = SimulationBaseClass.SimBaseClass()
```

Following this at least a single Task Group and Task must be created and linked. This is done by creating a Task Group also referred to as a Process and then adding a Task to this

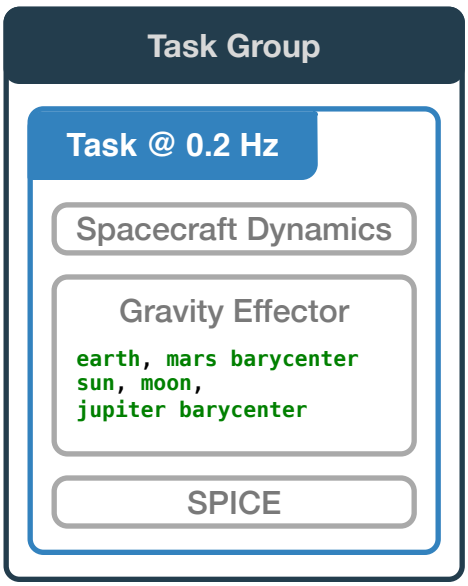


Fig. 6. Concept diagram of simple multi body gravity orbiter Basilisk simulation configuration.

Task Group. The Task integration rate is set to 5 seconds. However, in Basilisk, the base time scale is nano seconds and so the sec2nanos() utility is used for this conversion.

```
dynProcess = scSim.CreateNewProcess(
    simProcessName)
dynProcess.addTask(scSim.CreateNewTask(
    simTaskName, sec2nanos(5)))
```

Now the core Basilisk structures are instantiated and one begins to populate the simulation with various Basilisk Modules. The first Module here is the `SpacecraftPlus` Module which instantiates the rigid body hub to which other stateEffectors and dynamicEffectors can be associated. The `SpacecraftPlus()` object is added to the single Task.

```
scObject = spacecraftPlus.SpacecraftPlus
    ()
scSim.AddModelToTask(simTaskName,
    scObject, None, 1)
```

A number of gravity bodies (dynamicEffectors) are instantiated and added to the `scObject`'s `gravBodies` list.

```
gravBodies = gravFactory.createBodies(['
    earth', 'mars_barycenter', 'sun', '
    moon', 'jupiter_barycenter'])
scObject.gravField.gravBodies =
    spacecraftPlus.GravBodyVector(
    gravFactory.gravBodies.values())
```

Finally, a SPICE Module is created using the convenience function available in the gravity body factory class. This SPICE Module is also added to the Task.

```
gravFactory.createSpiceInterface(bskPath
    +'/supportData/EphemerisData/',
    timeInitString)
scSim.AddModelToTask(simTaskName,
    gravFactory.spiceObject, None, -1)
```

To begin the simulation three function calls are required. The first initializes the Task Groups, Tasks and Modules added to the simulation calling the selfInit, crossInit, resetInit functions. Following this, the stop time is set and then the simulation is launched.

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

Simulations can be executed for a specified duration after which configuration changes are made and the simulation continued further. This can be useful to simulate specific spacecraft sequence instruction sets. Additionally, Events objects are available and can be set to trigger a custom user provided function. This custom user provided function means that that the developer can trigger and change any variable/state in the simulation that is available through the Python interface of each Basilisk Module.

```
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()
```

An example plot generated from Basilisk data using the Python post-processing tools is shown in Fig. 7 shows the evolution of the spacecraft inertial position. Additionally, Fig. 8 demonstrates visually the replication of the Hubble SPICE trajectory overlayed with the Basilisk integrated trajectory. Finally, Fig. 9 demonstrates the difference and therefore close agreement of the RK4 integrated trajectory and the Hubble SPICE trajectory.

10. CONCLUSION

The Basilisk astrodynamics framework provides a new open source alternative for fully coupled spacecraft dynamics mission simulation. 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 Python user interface

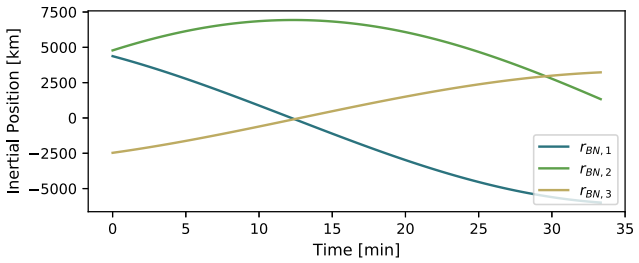


Fig. 7. Inertial position of the Hubble spacecraft simulation.

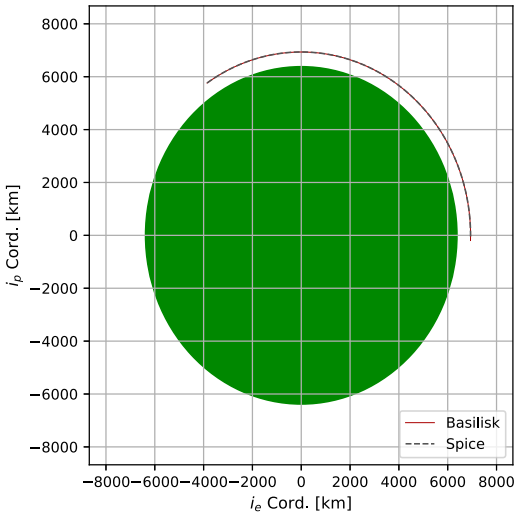


Fig. 8. Overlay of the Hubble SPICE trajectory with the Basilisk rk4 integrated trajectory.

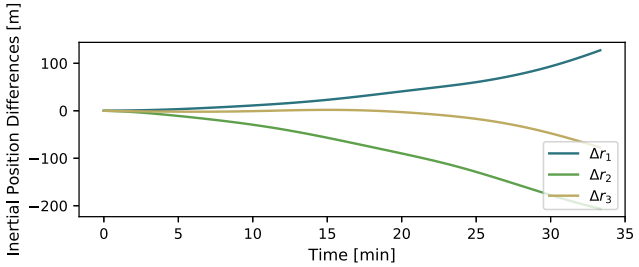


Fig. 9. The difference in inertial position between the Hubble SPICE trajectory with the Basilisk trajectory.

for each Basilisk component, allowing user 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.

11. REFERENCES

[1] MathWorks, "Matlab/simulink," https://www.mathworks.com/products/matlab.html, Accessed 23 Oct 2018.

[2] Analytic Graphics Inc (AGI), "Systems tool kit (stk)," https://www.agi.com/products/engineering-tools, Accessed 20 Oct 2018.

[3] a.i. Solutions, "Freeflyer," https://ai-solutions.com/freeflyer/, Accessed 21 Oct 2018.

[4] NASA Goddard Space Flight Center, "General mission analysis tool (gmat)," https://software.nasa.gov/software/GSC-17177-1, Accessed 27 Oct 2018.

[5] NASA Johnson Space Center, "Trick simulation environment," https://github.com/nasa/trick/wiki/FAQ, Accessed 20 Oct 2018.

[6] CS Systèmes d'Information, "Orekit: An accurate and efficient core layer for space flight dynamics applications," https://www.orekit.org, Accessed 15 Oct 2018.

[7] Jet Propulsion Lab DARTS Lab, "Darts shell (dshell)," https://dartslab.jpl.nasa.gov, Accessed 1 Oct 2018.

[8] NASA Goddard Space Flight Center, "42: A comprehensive general-purpose simulation of attitude and trajectory dynamics and control of multiple spacecraft composed of multiple rigid or flexible bodies," https://software.nasa.gov/software/GSC-16720-1, Oct 2018, Accessed 2018-10-1.

[9] Marshall Space Flight Center, "Marshall solar activity future estimation," https://sail.msfc.nasa.gov, Accessed 1 Oct 2018.

[10] Advanced Solutions, "Stk solis: Commercial plug-in to the analytical graphics, inc (agi) systems toolkit (stk)," http://www.go-asi.com/solutions/stk-solis/, Oct 2018, Accessed 1 Oct 2018.

[11] A. Jain C. Lim, "Dshell++: A component based, reusable space system simulation framework," in *IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT 2009)*, Pasadena, CA, July 19 - 23 2009, IEEE.

[12] A. Jain and G. Rodriguez, "Recursive flexible multibody system dynamics using spatial operators," *Journal of Guidance, Control, and Dynamics*, vol. 15, no. 6, pp. 1453–1466, Nov 1992.

[13] Cody Allard, Manuel Diaz Ramos, Hanspeter Schaub, Patrick Kenneally, and Scott Piggott, "Modular Software Architecture for Fully Coupled Spacecraft Simulations," *Journal of Aerospace Information Systems*, pp. 1–14, oct 2018.

[14] Open Source Initiative, "Isc license (isc)," https://opensource.org/faq, Accessed 15 Oct 2018.

[15] Vincent Driessen, "A successful git branching model," https://nvie.com/posts/a-successful-git-branching-model/, January 2010, Accessed 27 Oct 2018.