68th International Astronautical Congress, Adelaide, Australia.

IAC-17,C1,IP,22,x38512

Software Architecture for Deep-Space Navigation Filter Development

Maria Cols Margenet[a], Andrew Harris[b,*], Dr. Hanspeter Schaub[c]

[a] *Research Assistant,Department of Aerospace Engineering Sciences, University of Colorado, 431 UCB, Colorado Center for Astrodynamics Research, Boulder, CO 80309-0431.*, maria.colsmargenet@colorado.edu

[b] *Research Assistant, Department of Aerospace Engineering Sciences, University of Colorado, 431 UCB, Colorado Center for Astrodynamics Research, Boulder, CO 80309-0431.*, andrew.t.harris@colorado.edu

[c] *Glenn L. Murphy Professor of Engineering, Department of Aerospace Engineering Sciences, University of Colorado, 431 UCB, Colorado Center for Astrodynamics Research, Boulder, CO 80309-0431.*, hanspeter.schaub@colorado.edu

* Corresponding Author

Abstract

Autonomous navigation is essential for next generation missions in deep space where ground interaction is infeasible. Missions involving small-body flybys, target tracking and surface feature detection, autonomous landing, or touch-and-go maneuvers provide examples of applications that demand autonomous navigation. Additional interest has arisen in performing these missions with low- cost spacecraft in CubeSat or small satellite form-factors, which present additional constraints on the navigation problem. Intrinsic limitations of CubeSats involving onboard instrumentation, power and computational capabilities demand creative solutions to solve effectively the problem of estimating the spacecraft's rotational and translational states. In this context, implementing analysis tools that rapidly assess the performance of given navigation hardware and software combinations, and iterate upon them, is critical for reducing mission design time and cost. The focus of the present work is to explore the design and trade space for specific software implementations of navigation filter architectures, their respective strengths and weaknesses, and future paths forward for the field. Strong arguments are made in favour of using a modular navigation scheme in order to decompose the complex process of state estimation into a series of simpler steps and exchangeable components. In these lines, the Basilisk astrodynamics framework appears as a very attractive platform for the design of flexible navigation algorithms and its migration to actual flight software.

1. Introduction

The successful performance of autonomous navigation without ground in the loop requires sophisticated estimation capabilities. Estimation of complex, fully coupled states poses, in turn, new challenges on the implementation of navigation algorithms that are flexible and reusable across multiple mission profiles.

Determining a spacecraft's position and attitude remain core requirements for space mission software packages. From a theoretical perspective, the navigation problem has received extensive treatment from a theoretical perspective over the last five decades [1, 2, 3, 4, 5]; however, the manner in which these theoretical advances have been implemented remains strikingly similar to the techniques used since the early model-driven software designs of the 1960s and 70s. As mission planners increasingly seek cheaper, faster, and smaller platforms to accomplish "more with less," the techniques and tools used to implement algorithms to conduct filtering for deep-space navigation have remained stagnant. This work aims to identify the new challenges arising from small satellite platforms in the autonomous deep-space navigation design environment, shortcomings in existing software implementation methodologies, and potential new solutions leveraging modular programming environments, in particular the Basilisk astrodynamics software framework.

High launch costs, spare lift capacity, and improving miniaturization have increasingly pushed spacecraft designers towards the use of small satellites (1-100 kg) in place of more traditional designs. Typical small satellites are tightly scoped around one or two science objectives and instruments and around the natural size, weight, volume, and power constraints of small satellite platforms. These constraints limit the possible scope of a given small satellite mission. While this carries obvious disadvantages, the hard limits on small satellite mission scope also serve to reduce mission complexity, with knock-on benefits for mission costs and schedules, while enabling designers to tailor missions around specific instrument requirements. In this sense, "small" can be considered a fourth and necessary member of the phrase "faster, better, cheaper." The aforementioned benefits and drivers

have led to a dramatic rise in the number of small satellite launches over the last two decades.

Increasing interest in small satellites has also driven the examination of small satellites for deep-space missions. JPL's MARCO, which will fly to Mars as a secondary payload on the Mars 2020 mission as a technology pathfinder, University of Colorado Boulder (CU)'s CU-E3, which will fly on the first SLS flight in 2019, represent two in-development examples; many more have been either proposed or remain in the concept design phase. The low costs of small satellites make them natural choices for high-risk technology development missions; several commercial companies, including Planetary Resources and Deep Space Industries, have proposed the use of small satellites for asteroid exploration and mining.

This push for reduced-cost deep space exploration is additionally tied to improvements in space vehicle autonomy. At present, few deep space missions have made use of on-board autonomy due to its perceived lack of robustness, leading to high operational costs and large staffing needs. Reducing the dependency of space missions on contact with Earth will reduce mission cost and has been cited as a major technology development goal by various space agencies. However, software for autonomous systems is substantially more complex and therefore expensive to develop than for non-autonomous systems, compounding the cost and difficulty of development and testing cycles.

While historically aerospace software has been developed to be mission-specific, modular designs and shared standards adopted in the recent decades have shown to improve efficiency [6]. The development of inflexible, mission-specific flight algorithms is a recurrent and problematic pattern in the aerospace industry that needs to be addressed [7]. Reference [8] remarks that, although aerospace software is intended to support the rest of the system for which it was designed, it often disrupts system schedules and budgets due to lack of architecture and improper implementation.

One solution to manage the development of such software suites effectively is through incremental development, a technique enabled by modular software frameworks. To this end, a novel platform for agile flight software (FSW) development called Basilisk is under development at CU's Autonomous Vehicle Systems (AVS) Laboratory and the Laboratory for Atmospheric and Space Physics (LASP) to support agile, modular aerospace software development. The present work aims to identify specific challenges of autonomous small satellite deep space navigation, issues in existing approaches, and methods of modular software development featuring the Basilisk astrodynamics framework.

2. Deep Space Cubesat Navigation System Overview

Cubesats are a specific and recurring area of interest for deep-space exploration due to their low cost, somewhat standardized components, and ease of integration alongside larger primary payloads. As such, we develop our approach to navigation in the specific context of deep-space interplanetary navigation within a cubesat platform. Additionally, we aim to incorporate as much autonomy into the navigation suite as possible to support missions for which non-autonomous solutions are either technically or monetarily infeasible.

In doing so, we take the capabilities of other space navigation suites–such as JPL's AutoNav–as guidelines. The onboard navigation system is aimed to detect and isolate known celestial bodies (e.g. planets, moons, asteroids, etc.) in images obtained from its optical sensors, derive the associated navigation measurements and process those measurements to estimate the CubeSat's position, velocity and dynamical model parameters. The final product is required to be a tool that allows the user to simulate the navigation system and assess the performance for these types of missions, while having the freedom to explore the mission design trade space (e.g. orbit type, satellite model, optical sensor properties, star tracker properties, onboard clock stability, etc). Such a flight software system is broken down into the following components:

Navigation Flight Algorithms: involve estimation of translational states (position and velocity), attitude estimation (orientation and angular rate), onboard clock drift estimation (to be corrected with periodic ground updates) as well as anomaly detection to monitor system performance and take action to avoid estimate divergence (e.g. detected maneuvers, measurement outliers, etc).

Measurement Processing Flight Algorithms: celestial body beacons captured in an image are used to determine the spacecraft's translational states. If the lighting conditions and exposure times are appropriate, background stars contained in the same images can be simultaneously utilized in the attitude estimation algorithm. A different approach is to take background star observations at a different scheduled rate than beacons (i.e. single optical camera operating in separate star tracker and beacon sensor modes). This later solution may be the only option in proximity operations where illumination and time exposure for large bodies conflicts with stars (light point source) observation.

Onboard Dynamics Model and Simplified Propagator: Propagates all the spacecraft dynamic states relevant to the navigation problem. It is critical to understand that this is the dynamic model considered in flight

software and the integration scheme that the onboard computer will be using. There is a clear distinction between the flight software models that are to be embedded into the flight computer, and the simulation DKE (Dynamics,Kinematic and Environment) software models that are used to simulate the physical spacecraft behavior in replace to real hardware and real space environment conditions. Whereas it is desired to have simple onboard dynamics models and to use low-order integration schemes within FSW in order to reduce computational expenses, the DKE simulation models must be as accurate as possible in order to replicate the real physics reliably. Further insight in the distinction between FSW and DKE simulation is provided in the section that overviews the Basilisk Software Framework.

Onboard Ephemeris Storage: Contains ephemeris information of the various celestial bodies that are relevant to the navigation problem during the mission time-span. Once more, this corresponds to the ephemeris stored on the onboard computer. Therefore, standard ground ephemeris files, like JPL's SPICE kernels, are not suitable to be loaded and queried onboard. SPICE, in particular, is a very heavy package that would significantly impact execution times of FSW tasks.

These myriad software components are delineated to demonstrate the variety of differing software components that must be developed for deep space navigation missions to function.

3. FSW Development Strategies

3.1 COTS Software Model-Based Development

The complete engineering process to develop an aerospace FSW suite and test it on the flight processor is typically achieved by taking the following 3 steps:

1. **Develop and test flight algorithms in the desktop environtment:** The first step consists of developing a set of flight software algorithms suitable for the mission being considered. Dynamics, Kinematic and Environment (DKE) models are also built with the purpose of testing the FSW algorithm set in a simulated closed-loop until the desired capability is achieved and mission-specific requirements are met. Architecture design and modeling of both software functions and hardware subsystems is often performed using block-diagram programming software tools. Platforms commonly used for building quick models of control systems are Mathworks's Simulink and National Instruments LabVIEW. Reference [9] provides a comprehensive review of the strengths and weaknesses of aerospace COTS software packages that are currently available.
2. **Auto-generate code in the required programming language:** The next step is typically to select an automated source-code generation software tool that is compatible with the block-diagram modeling tools selected above and that auto-generates source code in the required programming language (aka autocoding). Both Simulink and LabVIEW software can produce C code directly from their drag and drop environment with the use of add-on packages.
3. **Define the flight target:** Finally, the flight target needs to be defined: autogenerated code can be targeted to a specific processor, Real Time Operating System (RTOS), or a publish/subscribe middleware layer.

Along the path that has just been presented, there are several concerning points that deserve a closer look:

- **DKE modelling:** State-of-the-art COTS softwares each have unique strengths, but provide limited capability to provide a complete physically realistic dynamical representation of a spacecraft for the purpose of ADCS design analysis, while allowing user-friendly, platform independent interaction. Additionally, many of these software solutions are prohibitively expensive for low-budget missions or student development. Open-source softwares/freeware may be poorly maintained and/or not user friendly, requiring more time to setup and learn than it is available for a particular mission.
- **Auto-coding:** Automatically generated code is usually less efficient, in either size or execution, than optimized hand-written code, and proves to be very challenging to reverify and debug due to the lack of readability. Although some code generators incorporate their own optimization features, the challenges remain.

These challenges have motivated a search for alternative design methodologies based on more contemporary software development strategies.

3.2 The Basilisk Software Framework

The AVS Laboratory and LASP are collaborating on a software development testbed named Basilisk. Basilisk seeks to capitalize on the potential of using Python as a testbed for FSW development provided that the simulation and flight algorithm code are written exclusively in C/C++, and then automatically wrapped into Python for simulation setup, analysis, and testing.

The architecture of the Basilisk software framework, as depicted in Fig. 1, is decomposed into two main blocks: a high-fidelity simulation of the physical spacecraft (DKE models) and a flight software set (on-board GN&C algorithms suite). Both the simulation and flight software

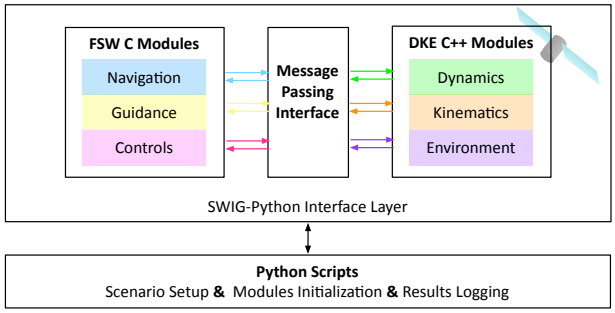

Fig. 1: Architecture of the Basilisk astrodynamics platform.

processes are developed in a modular architecture using C/C++ modules that communicate with each other through a Message Passing Interface. The modularity of the system implies that each process is decomposed into a series of simpler steps and exchangeable components, and the cascading of modules is set at the Python level, allowing different levels of simulation fidelity and flight software sophistication. The proposed modular scheme is a convenient strategy for missions with changing and evolving requirements and provides a systematic framework to scale mission complexity in a controlled manner that developers can manage.

In summary, the Basilisk platform is an excellent option for flight software prototyping and development that is specially suitable for, although not restricted to, low-cost missions for several reasons:

- **A generic set of FSW algorithms is already available:** The mix-and-match strategy of the Basilisk FSW architecture allows to suit a wide range of mission profiles with the already existing modules. Furthermore, the flight software set is easily scalable and dovetails very well with the creation of mission specific modules to satisfy particular requirements.

- **Reconfigurability and user-friendly analysis environment:** The construction and testing of the several FSW rate groups and of different modes of the flight application is handled through the high-level Python language, which is recognized as an excellent scripting environment and development testbed. Furthermore, Python-standard analysis products like numpy and matplotlib are readily available to facilitate rapid and complex analysis of data obtained in a simulation run without having to stop and export to an external tool.

- **High-fidelity DKE models are available:** The Basilisk simulation engine provides a complete, physically realistic dynamic representation of the spacecraft. Just to provide a few examples, it is possible to run simulations that include higher order gravitational effects, flexing dynamics[10] or solar radiation pressure effects[11].

- **Speed:** The fact that the underlying simulation executes entirely in C/C++ allows for faster than real-time simulation with built-in repeatable Monte Carlo capability.

- **Ease of transition from desktop testbed environment to the flight target:**

  Being open-source and cross-platform in nature (currently supported in macOS, Linux and Windows), the Basilisk package can be easily deployed and built in a desktop environment. With the generic FSW and DKE algorithms suite provided out-of-the-box, the user can readily run closed-loop dynamics numerical simulations. Furthermore, it is also possible to use Basilisk for hardware-in-the-loop (HWIL) testing, running the Basilisk flight algorithms on a flight processor and the DKE suite on separate hardware. Reference [12] shows a HWIL setup using the ARM processor of a Raspberry Pi, where Basilisk flight algorithms are readily compiled and deployed with no modifications. Reference [13] demonstrates how Basilisk-developed algorithms are integrated to NASA's middleware, the Core Flight System (CFS), and embedded onto a flight target running a standard RTOS. Transitioning from the Basilisk environment to the CFS is the strategy currently being targeted by LASP for a subsystem of a mission in development.

4. Space Zoo: Basilisk and DINO C-REx

Due to the pre-existing strengths afforded by Basilisk, it was selected to form the backbone of the Deep-space Interplanetary Navigation Optical Colorado Research Explorer (DINO C-REx) project, which aims to provide a complete set of tools for designing future deep-space navigation suites for small satellites using optical sensors. It provides a case study to describe the implementation of various components of the CubeSat problem. Figure 2 shows how DINO C-REx requirements fit into the the Basilisk framework:

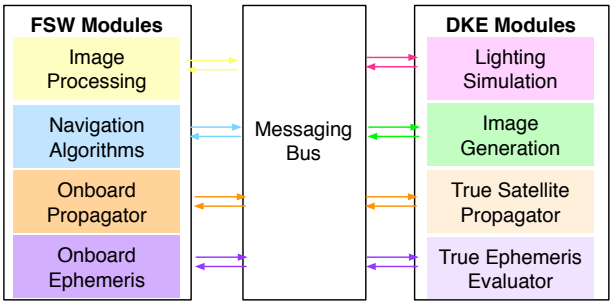

Fig. 2: The DINO project requirements framed in the Basilisk simulation architecture.

Having identified the requirements, the Basilisk framework can be even further exploited to yield a more com-

prehensive simulation context that allows the design and implementation of a realistic FSW executive system. A complete scheme of the envisioned FSW system functionality is provided in Fig. 3. The different parts of the executive are described next:

FSW Application: Main GN&C system. Includes image processing, filtering (rotational and translational state estimation) and onboard filter calibration for state-reduction.

FDIR: Global fault detection system.

Sequencing subsystem: Responsible for the commanding of the Imaging, ADC and Propulsion subsystems.

FSW Real Time: Time management system. Includes onboard ephemeris maintenance and clock drift compensation.

With the presented big picture in mind, the rest of the paper will discuss the challenges of the filtering navigation scheme that is to be implemented within Basilisk over the upcoming months. As highlighted in Fig. 3, the navigation algorithms are part of the FSW mission-specific application.

5. Modular Fractionation and its Application to the Navigation Problem

Recall that in the Basilisk framework flight algorithms are built through layers of modules with atomic functionality where each individual module can be added, replaced or swapped in a lego-like fashion. A key benefit is that the software is built through frequent increments rather than in a singular large effort. Every increment fulfills a well-defined functionality that contributes to the overall requirements. Encapsulating the GN&C functionalities in completely independent modules instead of monolithic algorithms is a key aspect in terms of software safety. There have been several instances of critical anomalies arising in complex software due to unexpected behavior of commercial off-the-shelf software[14]. With the aim of bringing down mission risks, the suggested staging of independent guidance modules allows scaling up the functionality in a safe and systematic manner. Further, modular designs allow direct reuse of functional algorithm code while avoiding duplication. Because the same module can potentially be used in very different mission-scenario contexts, there is an overall reduction of lines of code to maintain and to validate, which in turn translates into reduced V&V costs.

Reference [15] presents a modular scheme for the generation of attitude guidance algorithms that is flexible and numerical simulations in the form of integrated tests proved the validity of the architecture for the guidance case. Now, a similar modular technique is being developed to solve the computationally expensive problem of OD and attitude estimation (i.e. translational and rotational navigation). Some considerations to beware of in the new context of estimation are the challenges of a fractionated approach and the trade-off between a flexible analysis tool and ease of transition to actual flight software.

Next, three different approaches for solving the navigation/estimation problem are discussed: a monolithic approach, a modular "fractionated" approach and the so-called "state-manager" approach. The monolithic technique refers to the conventional way of performing orbit determination with a dedicated, stand-alone, OD-tool. It is discussed only for putting the reader into context.

5.1 Monolithic Approach

In principle, monolithic OD tools could be used jointly with the Basilisk framework. The strategy in this case would be to generate true orbit and dynamic data, and corrupt it if necessary, within the Basilisk engine. These data can then be pulled out of the Basilisk messaging system and used a posteriori to obtain estimation solutions with the user-custom OD tool. While this approach works for the purpose of developing and testing monolithic OD estimation algorithms, it is limited by the lack of ability to feed these OD outputs back to the Basilisk simulation dynamically in order to test other intrinsic components of FSW, like controls, in closed loop.

Indeed, the use of monolithic OD tools where state determination is the focus of the product, and other spacecraft events and FSW functionality are build as *attrezzo* around it, can complicate the development of an integral flight software system beyond navigation. The DINO-Basilisk toolset aims to build the flight algorithms for estimation and navigation as a single piece that fits into a bigger puzzle, in accordance to the concept illustrated in Fig. 3 .

A critical difference between Basilisk and most OD-focused tools is that within Basilisk there is a clear delineation between flight software and dynamics simulation. In conventional estimation terms, the dynamics simulation of Basilisk stands for the "true" model while the Basilisk flight algorithms are the simplified dynamics counterpart. Because the concept of "state to be estimated" belongs exclusively to flight software, it is not embedded into the true dynamics to keep track of the derivatives of the state that FSW cares about. Instead, flight software must have its own dynamic model, and operate upon that for state estimation.

5.2 Fractionated Approach

Splitting the development of GN&C algorithms into smaller, tightly-scoped submodules–a "fractionated" approach to software development– can simplify implementation and testing. However, it could be argued that mono-

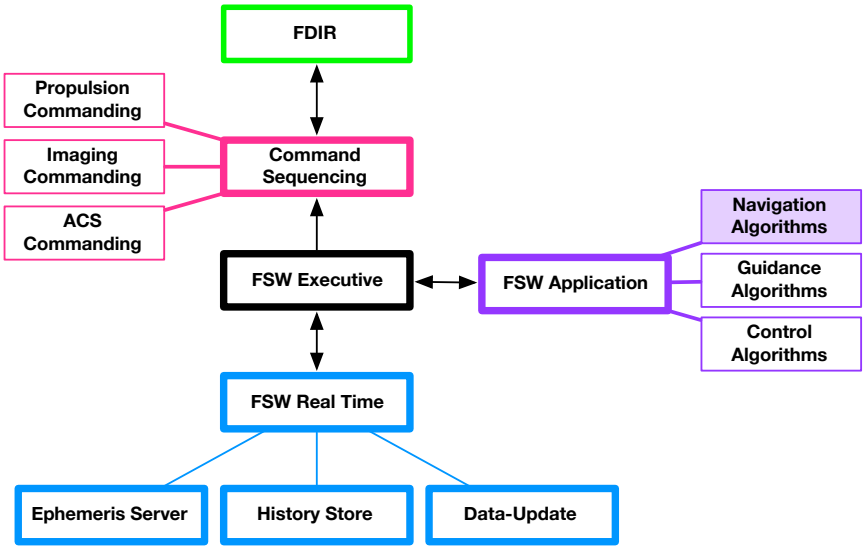

Fig. 3: Illustration of the overall FSW executive system highlighting the navigation algorithms, focus of this work.

lithic custom algorithms can reduce computational cost due to communication overhead. In the context of low power missions, like those involving cubesats or smallsat form factors, the computational load of a fractionated implementation must be considered for flight implementation.

While in Ref. [15] it is determined that, as far as the guidance is concerned, a large number of sub-components would be necessary in order for the fractionated approach to have a significant impact on speed, this statement cannot be readily extrapolated to estimation. In state estimation the cons of a fractionated technique could increase and therefore must be analyzed thoroughly, since inverting covariances and solving for states can be computationally intensive already in a single, custom, optimized algorithm. It is in the scope of the proposed work to compare the power consumption and CPU usage of a fractionated navigation algorithm over its monolithic counterpart.

5.3 State-Manager Approach

An alternative to the aforementioned architectures is an scheme involving a dedicated "state-manager." This is currently the basis of the underlying Basilisk "truth dynamics" engine, which is described briefly here for reference to inform the analogous estimation architecture. It is important to recall that this dynamics engine is separate from any means of propagation used in "flight software" tasks.

The Basilisk true dynamics architecture provides a flexible, fast means of propagating the dynamics of a spacecraft. The objective of the Basilisk dynamics architecture is to propagate the "core" states of a given spacecraft (translational and rotational) and any additional states necessary for other "dynamic" components (such as reaction wheels). This is accomplished through a so-called "state-manager" which tracks the system states necessary for the dynamics propagation. By default, the instantiation of a spacecraft within Basilisk creates a state manager that tracks its translational and rotational states, providing a core around which other states can be added. Physical forces that can affect these states are represented in software as "effector" objects, which are typically used to represent external forces and torques acting on a given spacecraft. Additional states can be added through a special class of effectors, which in turn extend the number of states tracked by the state manager. This is used in cases where additional dynamical states are required to accurately represent the spacecraft's motion, such as in the presence of reaction wheels. These additional states can have dependencies on other states through linkages managed by the state manager, enabling the development of extremely complex, coupled dynamics models through very simple building blocks.

This design has both advantages and disadvantages over the message-passing interface used by the fractionated approach. Using a dedicated state manager and specifying specific interfaces for models of phenomena that affect the spacecraft's dynamics retains modularity while simultaneously enabling the use of algorithm-speeding techniques like back-substitution to reduce the computational impact of cross-coupling. Standard interfaces to the dynamical equations of motion through the state manager simplifies both the addition of new dynamical models and the integration of those models.

While this architecture adds software complexity versus simple or hard-coded dynamic models, its flexibility and modularity is clear and has proven itself valuable to mission analysts. New models of dynamic phenomenon,

such as atmospheric drag or fuel slosh, have been rapidly developed and folded into Basilisk under this architecture. It enables the combination of flexibility, computational efficiency, and high fidelity that the Basilisk team has aimed for from the start.

The advantages of the "state-manager" approach for flexibility and speed make it attractive for navigation techniques, which require the use of myriad sensors and are typically relatively computationally expensive. In this architecture, sensors are represented as effectors with specified relationships between their readings and the states desired to be estimated. Sensor parameters, such as bias and drift, could be estimated alongside core spacecraft states in a manner similar to that accomplished in the existing state manager architecture.

A key difficulty in the extension of this architecture to the coupled estimation problem is the tracking of state variances and covariances. Because the states are dynamically initialized, tracking the covariance matrix for the entire system will be challenging. Further, because the idea of the state manager exploits the advantages of object-oriented programming or OOP (possible with languages like C++ or Python), the concept is not easily portable to embedded C flight software.

6. High-level Modular FSW Stack

To demonstrate the benefits of a fractionated approach, a fractionated navigation system is presented here as a "representative" suite for a small satellite using optical navigation. Individual modules are tightly scoped by function, and communicate with one another using a general message passing interface.

The idea is to use optical measurements of nearby "beacon" celestial objects and background stars to solve the coupled attitude-orbit determination problem. The system's capabilities include onboard ephemeris services, onboard trajectory propagation as well as generation of observables and partials. A key concept is that all these functionalities are encapsulated in plug-and-play modules that are completely independent of the filtering strategy downstream: for the linearized case, either a batch or sequential filter could take advantage of the same outlined modules. The case of higher order filters is simpler only in the sense that partials are not even required, but is not the focus of the present study because of their higher computational requirements, likely to conflict with CubeSat processor limitations.

6.1 Onboard Ephemeris Handling: Frames and Time

An important requirement of any estimation tool is to handle nicely the switching between reference frames and to keep track of the ephemeris time after compensating for the onboard clock drift. A sample modular way of fulfilling this requirement is depicted in Fig. 4. The modules involved on ephemeris services are described next.

6.1.1 Clock Module

It reads the spacecraft time (second elapsed since system boot) from the physical spacecraft clock and, if provided with an estimate of the clock drift, it is able to infer a value for the ephemeris time (i.e. Julian Date) to be considered in flight. By breaking this into a separate module, developers can simulate system responses to clock inaccuracy without using a physical hardware clock; at the same time, by treating the clock input as an abstract input to the system, the rest of the navigation system can be transitioned to flight without much complexity.

6.1.2 Celestial Body Ephemeris Module

Each celestial body to be considered by FSW has an individual instantiation of the same Ephemeris Module. In essence, the Ephemeris Module is an storage buffer for the position and velocity of a celestial body (planets, Sun or beacons) during the time arch of the mission. Early in development, this module would likely use "full" ephemeride solutions like SPICE for ease of use; however, the modular interface to the ephemeris output could readily be replaced with a more flight-suitable ephemeris solution (such as Chebyshev polynomials) when the rest of the navigation system is transitioned into flight. Once again, the application of tightly-scoped modules reduces headaches down the development timeline.

6.1.3 Ephemeris Converter Module

Handles the change of base reference frame to be used onboard, according to the requirements of each mission phase. Inputs to the module are: set of ephemeris data with respect to an initial base reference and the new base reference desired. The module output is the set of ephemeris data mapped to the new base reference (body correlated ephemeris). Figure 4 shows that the initial base reference is heliocentric (i.e. some Sun-centered frame). If the new base reference corresponds, for instance, to an Earth centered frame, FSW would now provide a pre-Copernican view of the word. Breaking out this functionality explicitly simplifies debugging during development and testing after development is completed, as the ephemeris conversion happens only in one specific component of the software suite.

6.2 Dynamics and Measurement Models

The problem of coupled orbit determination and attitude estimation can be considered either hard-coupled or soft-coupled. Hard-coupling implies having a single state that encompasses both translational and rotational variables which are hence estimated within the same filter. Soft-coupling allows two different states and filters for spacecraft's attitude and OD running at different rates and feeding each other.

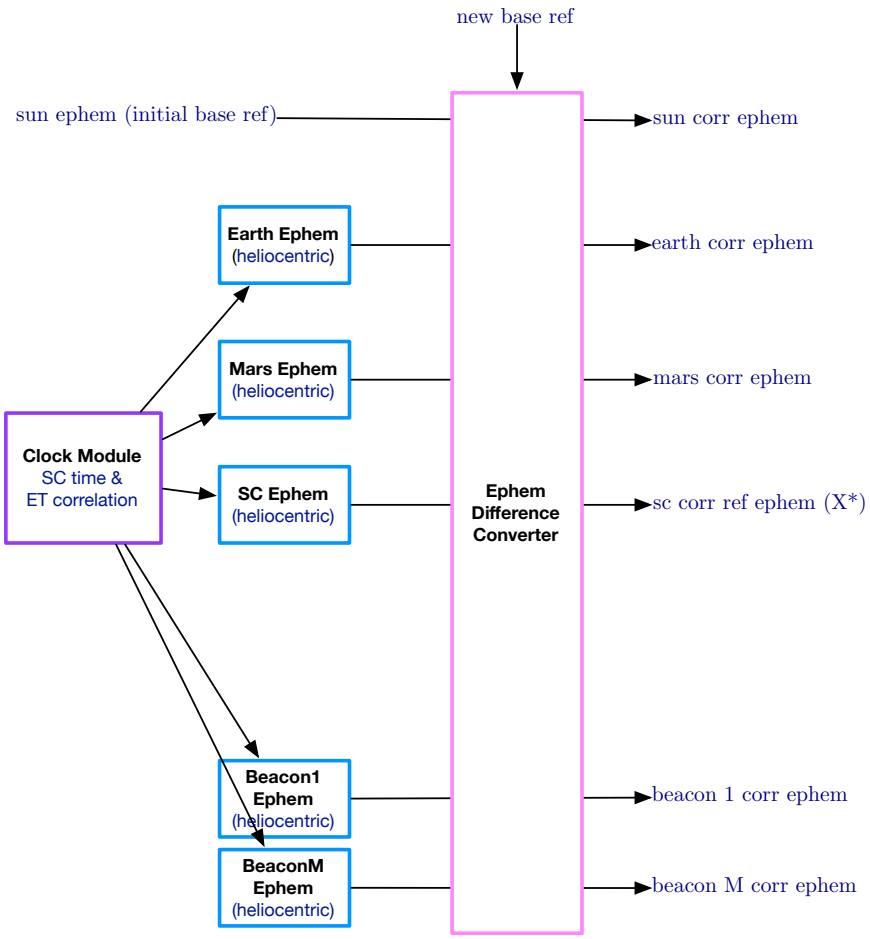

Fig. 4: Modular handling/switching of onboard reference frames, with initial ephemeris storage for Earth, Mars and the spacecraft's reference trajectory in an heliocentric frame.

Figure 5 shows a modular scheme of the orbit estimation process using optical beacons. The mission phase considered is an Earth-Mars interplanetary cruise, and a suitable dynamics model is considered Because the orientation of the on-board optical sensor depends on the spacecraft's attitude, rotational and translational states become coupled. In Fig. 5, the spacecraft attitude variable ("SC ATTITUDE") is highlighted in capital cyan letters to mark its role within the navigation scheme. The high-level architecture depicted in Fig. 5 leaves open the soft or hard coupling character of the navigation solution. Upcoming work will focus on detailing aspects of the filtering architecture to allow exploration of both coupling approaches while maintaining the plug-and-play modularity. It is important to highlight that Fig. 5 is showing generic modules configured for a specific mission phase that drives the selection of the onboard dynamic and measurement models. In this sense, "Mars-Earth Cruise Dynamics Module" corresponds to a specific instantiation of a generic Onboard Dynamic Module, whereas "Pixel & Line Mapping Module" corresponds to a specific instantiation of a generic Measurement Module. The generic modules for onboard dynamic and measurement modelling are described next. Some of the mathematics related to the particular instantiations showcased in Fig. 5 is also derived in order to put the reader into more technical context.

6.2.1 Onboard Dynamics Module

Generic Functionality:
This module is aimed to define the dynamic forces $\boldsymbol{f}_i$ to be modeled on-board for state estimation purposes. If linearization is to be applied in the filtering strategy downstream, the Jacobian of the state, defined as $\boldsymbol{A} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{X}}$ will be needed. Without loss in generality, Fig. 6 displays the aforementioned functionality.

General forces $\boldsymbol{f}_i$ to potentially be considered are follows:

$$\boldsymbol{f} = \boldsymbol{f}_{\text{m}} + \boldsymbol{f}_{\text{sh}} + \boldsymbol{f}_{\text{3b}} + \boldsymbol{f}_{\text{rel}} + \boldsymbol{f}_{\text{d}} + \boldsymbol{f}_{\text{srp}} + \boldsymbol{f}_{\text{srp}} + \boldsymbol{f}_{\text{other}} \quad (1)$$

where $\boldsymbol{f}_{\text{m}}$ is the point mass acceleration of the orbited planet, $\boldsymbol{f}_{\text{sh}}$ is the acceleration due to spherical harmonics, $\boldsymbol{f}_{\text{3b}}$ are third-body perturbations, $\boldsymbol{f}_{\text{rel}}$ are relativistic

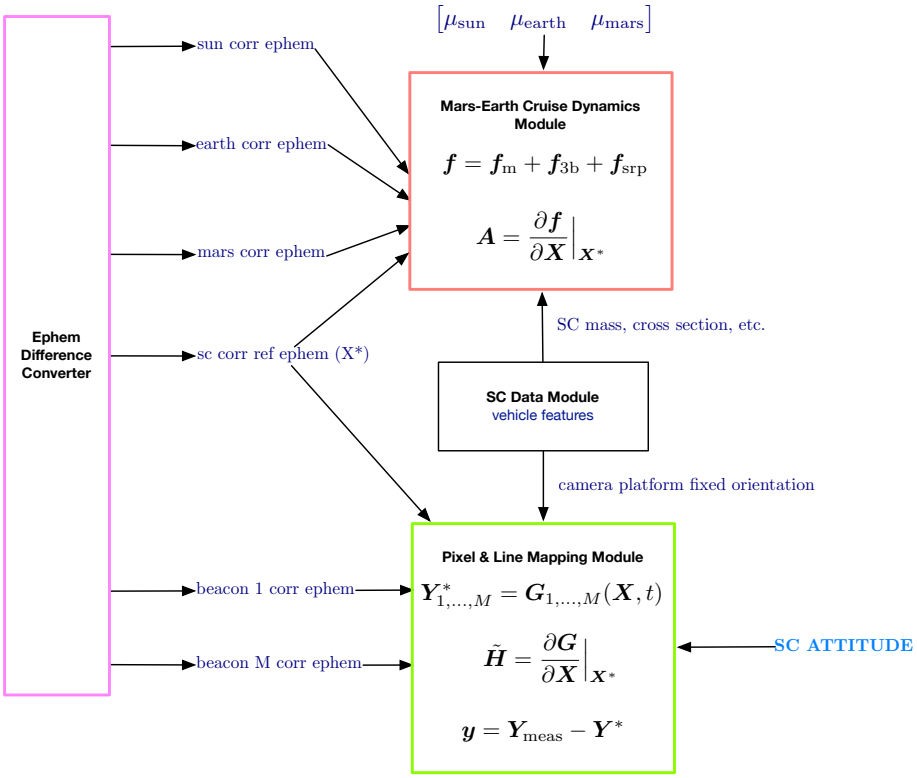

Fig. 5: Sample fractionated architecture. Onboard Dynamics Module appears in the red box, being initialized for the particular phase of Earth-Mars interplanetary cruise. Onboard Measurement

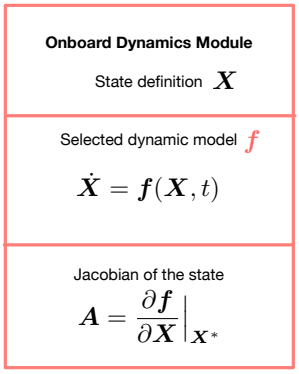

Fig. 6: Generic Onboard Dynamics Module.

effects, $\boldsymbol{f}_{\text{d}}$ is the atmospheric drag effect, $\boldsymbol{f}_{\text{srp}}$ is solar radiation pressure, $\boldsymbol{f}_{\text{arp}}$ is albedo radiation pressure, and $\boldsymbol{f}_{\text{other}}$ correspond to unknown acceleration terms that can be modelled for instance with polynomials.

The accuracy of the selected onboard dynamic model depends on the requirements of the mission phase: it is usual to neglect some dynamic effects for the sake of onboard computational speed.

Particular Instantiation:
For instance, during an interplanetary cruise Earth-Mars, the following reduced dynamic model would be a sensible choice:

$$\boldsymbol{f} = \boldsymbol{f}_{\text{m}} + \boldsymbol{f}_{3\text{b}} + \boldsymbol{f}_{\text{srp}} \tag{2}$$

This is the example dynamic model selection of Fig. 5. Developing the terms of Eq.(2) further and using a simple cannonball model for $\boldsymbol{f}_{\text{srp}}$:

$$\boldsymbol{f}_{\text{m}} = -\mu_N \frac{\boldsymbol{r}_{B/N}}{r^3_{B/N}} \tag{3a}$$

$$\boldsymbol{f}_{3\text{b}} = -\sum_{i=1, i\neq N}^{n_b} \mu_i \left(\frac{\boldsymbol{r}_{B/N} - \boldsymbol{r}_{i/N}}{|\boldsymbol{r}_{B/N} - \boldsymbol{r}_{i/N}|^3} + \frac{\boldsymbol{r}_{i/N}}{r^3_{i/N}} \right) \tag{3b}$$

$$\boldsymbol{f}_{\text{srp}} = C_R K_{\text{srp}} \frac{\boldsymbol{r}_{B/N} - \boldsymbol{r}_{\text{Sun}/N}}{|\boldsymbol{r}_{B/N} - \boldsymbol{r}_{\text{Sun}/N}|^3} \tag{3c}$$

where C_R is the SRP coefficient, n_b is the total number of celestial bodies ($n_b = 3$ accounting for Earth, Marth, Sun), and K_{srp} is a constant scale factor defined as follows:

$$K_{\text{srp}} = \ell \frac{A_{sc}}{m_{sc}} (149,597,870\text{km})^2 \left(P_{\text{srp, AU}} \frac{1\text{km}}{1000\text{m}} \right) \tag{4}$$

with ℓ being a factor that can be applied to represent the fraction of Sun that is visible (currently always set to $\ell = 1$) and with

$$P_{\text{srp, AU}} = \frac{\Phi}{c} = \frac{1358}{299,792,458} \approx 4.53 \cdot 10^{-6}\text{Pa} \tag{5}$$

Such a module would then require, as an input, ephemeris information data (output of the Ephemeris

Converter Module) to evaluate $\boldsymbol{f}_{\text{m}}$ and $\boldsymbol{f}_{\text{3b}}$ as well as vehicle properties data to evaluate $\boldsymbol{f}_{\text{srp}}$. Note that, in Eq.(3), the spacecraft position $\boldsymbol{r}_{B/N}$ and the SRP coefficient C_R are typically variables to be estimated in the filter.

Once again, breaking out this specific filter feature - of modelling dynamics onboard - as a separate module simplifies development and testing. Developers can rapidly determine how filter performance changes in response to changing dynamic models without large system-level overhauls by simply swapping out the underlying dynamic forces within the model. This also allows for trades to be readily made between computational efficiency and filter accuracy under different dynamic models. A specific example of this type of trade is the selection of an integration technique for the dynamics; algorithm designers could select between simple Euler schemes or more complex RK methods.

6.2.2 Measurement Definition Module

Generic Functionality:
Every type of measurement processed by FSW, in order to estimate the spacecraft's translational and rotational states, will own an individual instantiation of the same module class. The covariance matrix R and, for a linearized filter case, the Jacobian of the measurement $\tilde{\boldsymbol{H}}$ and the measurement residual $\boldsymbol{y}$ are required by the filter modules downstream. Without loss in generality, Fig. 7 displays the aforementioned functionality.

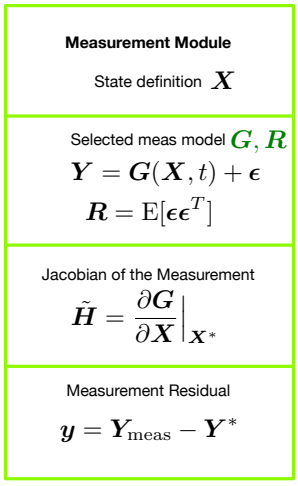

Fig. 7: Generic Measurement Module.

Particular Instantiation:
In the optical image case for OD, the type of measurement considered is the pixel and line position of imaged asteroid beacons. Therefore, a specific instantiation of such a measurement model would be the so-called "Pixel and Line Mapping Module". As mentioned, if linearization is to be applied in the on-board filtering strategy, a residual vector between imaged and predicted coordinates is required. In this context, it is desirable to generate the residual vector by mapping the predicted location of beacons (according to onboard ephemeris) into line-of-sight information (azimuth-elevation or pixel-line) that can then be compared to the imaged data, and not the other way around. The rationale for this is that selected beacons whom ephemeris is loaded onboard are limited In contrast, the number of candidate point sources in the image can be larger due to the presence of outliers.

The process for mapping the predicted location of beacon asteroids (according to onboard ephemeris) into line-of-sight information (pixel and line) is outlined following.

1. Retrieve inertial line-of-sight (LOS) vector of beacon asteroid $\boldsymbol{\theta}_N$ from the onboard ephemeris.

2. Compute Direction Cosine Matrix that maps from inertial frame $\mathcal{N}$ to camera platform frame C.

$$[CN] = [CB][BN]$$

   where $[CB]$ is the constant attitude offset between the camera platform and the spacecraft's principal body frame (assuming a rigid-body system), and $[BN]$ is the spacecraft's instantaneous attitude as provided by the Attitude Control Subsystem.

3. Map inertial predicted LOS vector $\boldsymbol{\theta}_N$ into camera-frame coordinates:

$$\boldsymbol{\theta}_C = \begin{bmatrix} \theta_{x,c} \\ \theta_{y,c} \\ \theta_{z,c} \end{bmatrix} = [CN]\boldsymbol{\theta}_N$$

4. Transform $\boldsymbol{\theta}_C$ into 2D camera plane:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \frac{f}{\theta_{z,c}} \begin{bmatrix} \theta_{x,c} \\ \theta_{y,c} \end{bmatrix}$$

   where f is the camera focal length in mm, and (x, y) is the projection of the LOS vector into focal plane coordinates.

5. Convert from rectangular coordinates (x, y) to pixel and line (p, l):

$$\begin{bmatrix} p \\ l \end{bmatrix} = \begin{bmatrix} k_x & k_{xy} & k_{xxy} \\ k_{yx} & k_y & k_{yxy} \end{bmatrix} \begin{bmatrix} x \\ y \\ xy \end{bmatrix} + \begin{bmatrix} p_0 \\ l_0 \end{bmatrix}$$

   where (p, l) is the center pixel and line of CCD, and k_{ij} are camera parameters that are calibrated on flight. Nominally, k_x and k_y are set to match f ands the off-diagonal terms are set to zero.

Eventually, the computed pixel and line values of each beacon are differenced with their respective observed values to get a residual matrix.

6.3 Filtering

The illustration in Fig. 5 does not particularize on any specific filter strategy (i.e. batch or sequential) beyond linearization. Indeed, the overall idea of the navigation architecture under construction is to provide the ability to plug-and-play with different linear filters without affecting the internals of the other navigation modules. Here, the asynchronous nature of the messaging architecture is useful. The update module could store sensor inputs over many time instances (when operating in a batch mode), or instead opt to process whatever measurements were provided most recently (as in a sequential filter). The rest of the system modules are not tightly coupled to the implementation of this specific step; they can output at whichever frequency they want by design. While this creates an impetus for system designers to carefully craft the runtime of each module, it provides a clear framework in which timing considerations can be considered and optimized at the level of tightly-scoped modules instead of monolithic software packages.

7. Conclusion and Future Work

Navigation filter development seems well-suited to modular software development frameworks. Fractionated MPI-based techniques, which are currently supported by existing software frameworks and which have long histories of success in other mediums, could be used immediately to bring the benefits of modularity to navigation filter development and implementation. The detailed navigation example provided demonstrates some of the qualitative benefits of such an architecture for both development and testing processes. More complex state management techniques may be less valuable for flight software, but offers substantial benefits to early mission designers whose hardware and requirement environments change rapidly. Both classes of filtering architecture established in this work are supported within the Basilisk astrodynamics software framework.

Naturally, the efficacy of these architectures depends heavily on their relative computational efficiency, ability to deliver on promised modularity, and ease of transition to flight software. In light of these necessary metrics, future work for these efforts is centered around the implementation of these modular navigation filter architectures for comparison with existing techniques.

REFERENCES

[1] Lefferts, E. J., Markley, F. L., and Shuster, M. D., "Kalman Filtering for Spacecraft Attitude Estimation," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 5, No. 5, 1981, pp. 417–429.

[2] Crassidis, J. L. and Markley, F. L., "Survey of Nonlinear Attiude Estimation Methods," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 30, No. 1, 2007, pp. 12–28.

[3] Crassidis, J. L. and Markley, F. L., "Unscented Filtering for Spacecraft Attitude Estimation," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 26, No. 4, July–Aug. 2003, pp. 536–542.

[4] Karlgaard, C. D. and Schaub, H., "Nonsingular Attitude Filtering Using Modified Rodrigues Parameters," *Journal of the Astronautical Sciences*, Vol. 57, No. 4, Oct.–Dec. 2010, pp. 777–791.

[5] O'Keefe, S. A. and Schaub, H., "Shadow Set Considerations For Modified Rodrigues Parameter Attitude Filtering," *AIAA Journal of Guidance, Control, and Dynamics*, Vol. 37, No. 6, 2014, pp. 2030–2035.

[6] Royce, W., "Managing the Development of Large Software Systems," Technical Papers of Western Electronic Show and Convention (WesCon), IEEE, 1970, pp. 1–9.

[7] Rarick, H. L., Godfrey, S. H., amd R. T. Crumbley, J. C. K., and Wilf, J. M., "NASA Software Engineering Benchmarking Study," SP 2013-604, NASA, May 2013.

[8] Blanchette, S., "Giant Slayer: Will You Let Software be David to Your Goliath System?" *Journal of Aerospace Information Systems*, Vol. 13, No. 10, 2016, pp. 407–417.

[9] Alcorn, J., Schaub, H., Piggott, S., and Kubitschek, D., "Simulating Attitude Actuation Options Using the Basilisk Astrodynamics Software Architecture," *67th International Astronautical Congress*, Guadalajara, Mexico, Sept. 26–30 2016.

[10] Allard, C., Diaz-Ramos, M., and Schaub, H., "Spacecraft Dynamics Integrating Hinged Solar Panels and Lumped-Mass Fuel Slosh Model," *AIAA/AAS Astrodynamics Specialist Conference*, Sept. 12–15 2016.

[11] Kenneally, P. W. and Schaub, H., "High Geometric Fidelity Modeling of Solar Radiation Pressure Using Graphics Processing Unit," *AAS/AIAA Spaceflight Mechanics Meeting*, Napa Valley, CA, Feb. 14–18 2016.

[12] Cols Margenet, M., Schaub, H., and Piggott, S., "Modular Platform for Hardware-in-the-Loop Testing of Autonomous Flight Algorithms," *International Symposium on Space Flight Dynamics*, Matsuyama-Ehime, Japan, June 3–9 2017.

[13] Piggott, S., Alcorn, J., Margenet, M. C., Kenneally, P. W., and Schaub, H., "Flight Software Development Through Python," *2016 Workshop on Spacecraft Flight Software*, JPL, California, Dec. 13–15 2016.

[14] Leveson, N. G., "The Role of Software in Spacecraft Accidents," *AIAA Journal of Spacecraft and Rockets*.

[15] Cols Margenet, M., Schaub, H., and Piggott, S., "Modular Attitude Guidance Development using the Basilisk Software Framework," *AIAA/AAS Astrodynamics Specialist Conference*, Sept. 12–15 2016.