Modular Platform for Hardware-in-the-Loop Testing of Autonomous Flight Algorithms

By Mar COLS-MARGENET,[1)] Hanspeter SCHAUB,[1)] and Scott PIGGOTT[2)]

[1)]*Autonomous Vehicle Systems Laboratory, University of Colorado Boulder, Boulder, Colorado, United States*
[2)]*Laboratory for Atmospheric and Space Physics, Boulder, Colorado, United States*

(Received April 17th, 2017)

A novel and flexible platform for prototyping flight software and testing with hardware-in-the-loop is proposed based on the Basilisk software and the Raspberry Pi hardware. The modular and scalable nature of Basilisk and the performance and affordability of the Raspberry Pi are described as a powerful combination for cost-effective mission development that is ideal for, although not restricted to, CubeSats or small satellite form-factors. A numerical simulation of an autonomous guidance maneuver where GN&C flight algorithms run on the flight processor in a closed-loop dynamics simulation, with realistic TCP/IP communication between hardware and software, is used as a proof of concept to demonstrate the validity of the architecture. Given that on-board autonomous capabilities demand high-performance processor capability and would highly benefit from agile flight software development, the present technology demonstration ultimately represents an effective strategy towards the embracement of full autonomy for next-generation spacecraft missions.

Key Words: Autonomy, Commercial Products, Basilisk Software Framework, Raspberry Pi

Nomenclature

cFE	:	Core Flight Executive
CFS	:	Core Flight System
CM	:	Compute Module
COTS	:	commercial off-the-shelf
CPU	:	central processing unit
DKE	:	dynamics, kinematics and environment
DSN	:	deep space networks
FSW	:	flight software
GN&C	:	guidance, navigation and controls
HW	:	hardware
IO	:	input-output
IP	:	internet protocol
MPI	:	message passing interface
MRP	:	modified Rodrigues parameter
OBC	:	on-board computer
PCB	:	printed circuit board
RTOS	:	real time operation system
SW	:	software
TCP	:	transmission control protocol
V&V	:	verification and validation

1. Introduction

Space missions rely highly on the efficiency and reliability of the on-board flight software in order to perform autonomous attitude control or orbit corrections. These critical software functions undergo a stringent review and validation process prior to flight, which can be both costly and time consuming. The complete engineering process to develop an aerospace Flight Software or FSW system encompasses an involved path starting from a preliminary desktop design and analysis all the way to testing on the flight hardware. The present work focuses, in particular, on the step of testing and analyzing the flight algorithms on the on-board flight processor. While multiple challenges exist in terms of both hardware and software requirements, an attractive, cost-effective approach is explored and demonstrated in the present work that combines commercial hardware with agile FSW development.

The paper is outlined as follows: In section **2.**, the importance of autonomy in deep-space missions in particular is highlighted, including a brief description and reasoning about the limited status quo. A critical path towards embracing full autonomy is then proposed that takes advantage of the flexibility offered by low-cost space exploration through small satellite form-factors. Section **3.** provides an overview of the commercial-of-the-shelf (COTS) products in terms of hardware and software that are commonly adopted in the context of small spacecraft mission design. Special attention is paid to the problems entailed by traditional space hardware and conventional software development approaches. In Section **4.** a novel proposal for low-cost FSW prototyping and flight targeting is made. This proposal addresses effectively all the problems and challenges that were identified in the previous section, and involves combining the Basilisk Software Framework and the Raspberry Pi hardware. The underlying reasons for this particular choice of hardware and software are provided, and a comparison to a similar project by NASA is included. In Section **5.** a numerical simulation is presented that serves as a technology demonstration for testing Basilisk-developed flight algorithms in a flight processor, the Raspberry Pi, with high-fidelity simulated closed-loop dynamics, all in a user-friendly and flexible environment.

2. Autonomy in Deep Space Missions

Autonomous capabilities are essential for the next generation of deep space missions where the light time delay makes ground interaction infeasible. Examples of applications that demand autonomy include missions involving small-body fly-

bys,[1] target tracking and relative navigation,[2] surface feature detection,[3] autonomous landing[4] , or touch-and-go maneuvers.[5] Although the applications and benefits of autonomy are varied and numerous, the number of missions flown to date with on-board autonomic capability remains small. Instead, it is common to include ground-in-the-loop and transfer responsibilities to ground whenever possible. This practical but expensive approach frames the way in which missions are designed, and strongly limits the amount and quality of science that can be returned for the following reasons:

- Limited quantity of science: Reducing the time spent relaying data to Earth and waiting for commands could enable more time spent doing science if the spacecraft was able to process data on-board.
- Limited quality of science: Limiting the spacecraft data collection to executing pre-defined sequences forces the science return to be limited by a priori knowledge rather than the evolving state.

With these considerations in mind, future exploration of icy moons like Europa or interstellar targets will require, nonetheless, unprecedented autonomy. In this context, developing full hardware and software capabilities for spacecraft to operate increasingly autonomously is a milestone to maximizing the science return of current missions and expanding the horizons of space exploration.

2.1. Underlying Reasons of Limited Autonomy

There are two main underlying reasons limiting the implementation of autonomy in the development of space systems: the need of **sophisticated software (SW)** and the need of **powerful hardware (HW)**.

Firstly, the successful performance of autonomous maneuvers without ground in the loop requires sophisticated flight software capabilities: the spacecraft would no longer just be following uplinked commands and sequences, but intensive computations involving data processing and decision-making would happen now on-board. Often, the flight software for new missions is developed from the ground up. In these cases, implementing complex FSW algorithms, either from scratch or by heavily refactoring inherited code, might be regarded as not enough worth the one-effort development (i.e. mission-specific designs that are meant to satisfy specific requirements and do not contemplate future software reusability). It is important to highlight, though, that 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, indeed, a recurrent problematic pattern in the aerospace industry that needs to be addressed.[7] To this end, a novel platform for agile FSW development called Basilisk is later presented that would leverage the handling of software complexity demanded by autonomous applications.

Secondly, and most prominently, aerospace technology lags state-of-the-art consumer technology due to flight heritage and radiation-hardening requirements. While commercial devices, specifications and capabilities are driven by the consumer market, radiation hardened electronics are domestically driven by US Government needs and requirements, lagging commercial development by about 10 years.[8] Specifically, the flight processors are the most critical to properly shield to ensure uninterrupted operation. Some of the most common radiation-hardened processors used in space are RAD750, Coldfire or Leon3, all of them being very expensive and presenting similar limited performance. Still, there are numerous motivations for using high performance processors capable of performing calculation-intensive tasks. Currently, the objectives and strategies of NASA's Exploration Systems Mission Directorate are constrained by the computing capability and power efficiency of the processors used in space.[8] Missions involving sophisticated autonomous vehicle operations, autonomous rendezvous and docking, vision systems, and precision landing systems, are all predicted to require processor capabilities that far exceed the current RAD750's performance specifications.

2.2. Full Autonomy through Low-cost Exploration

If the novel mission concepts on the drawing board are to become the reality, it is clear that a dramatic change in terms of both hardware use and software design is needed. A very attractive path forward is to **combine COTS hardware with agile FSW development.** Adopting commercial hardware is the first step towards a new level of autonomy, since it would allow the capability of onboard flight software to increase dramatically. Increased software abilities translate, in turn, into higher complexity and the need of being able to handle it efficiently in order to provide the desired reliability and reusability.[9]

Recently, additional interest has arisen in performing deep-space missions with low-cost spacecraft in CubeSat or small satellite form-factors. CubeSat frameworks allow for lower cost threshold for design, construction and launch, therefore opening the door completely to the actual integration of both COTS hardware and software in spacecraft systems.

Regarding **hardware**, the use of low-cost spacecraft for deep-space exploration dovetails perfectly with the adoption of state-of-the-art technology to overcome current limitations of traditional, expensive, radiation-hardened processors. Consumer technology is not only cheaper and more powerful, but its integration also enables the possibility to exploit the advantages of distributed systems. Furthermore, low-cost mission designs often escape flight heritage requirements, which would otherwise imply that only HW that has previously been flown and tested in space can be used.

In terms of **flight software** development, most low-cost missions do not have at their disposal inherited, previously flown, flight algorithms. Furthermore, in the low-cost framework, the resources to built a new flight software set from scratch are limited. In this context, the availability of software prototyping platforms that are flexible and reliable is highly valuable. While the advantages of flexible software architectures apply to all kind of missions, they are further compounded by proposals involving small spacecraft, whose intrinsic mass, power, and volume constraints require creative GN&C solutions. An example of this can be found in the Deep Impact mission,[10] which used a science instrument for navigation during approach phase and as a backup sensor during operations. This new paradigm of cost-limited space exploration demands agile flight software development.[11]

3. Commercial Products in Space

Next the *status quo* of COTS hardware processors in low-cost space missions is considered. Then, the common three-step process of developing a FSW space system (aka model-based development)[12] is revised. Although model-based development is an approach commonly adopted by both standard missions and low-cost missions, it is in the low-cost framework where available COTS software packages are mostly used and therefore where the most relevant aspects for the present work lie.

3.1. COTS Hardware

Space is a harsh environment where it is difficult to ensure that a computer will operate reliably for an extended period of time. Cosmic radiation interferes with transistors and can bit-flip computer memory (single event upset crash). Generally, two approaches may be employed (independently or in combination) to protect the spacecraft's electronic systems in the radiation environment: 1) commercial parts (COTS) in redundant and duplicative configurations and 2) Electronics hardened for radiation and environmental exposure. Current NASA investigations are assessing the pros and cons of these methods.[8] While solving this problem through radiation-hardening by electronics is traditionally highly expensive, using COTS technologies (i.e. radiation-hardening by software architecture and redundancy) has been proven as an effective method in reducing these costs.

Among the commercial tech products that could be suitable for low-cost space exploration, NASA has considered the use of Arduino platforms and Raspberry Pi's.[13] NASA's Arduino-based CubeSATs were launched in 2013 and the Lunar Sail program, which is based on the Raspberry Pi, is still under development. In a slightly different context (technology developed for ground testing and simulation rather than flight), the Pi-Sat project is another initiative from NASA Goddard that proposes the Pi as a flight-simulation testbed.[14] The Pi-Sat concept is of special relevance to the present work and it is detailed and contrasted in later sections of this paper.

Next a brief comparison between the very popular hobbyist hardware devices Arduino and Raspberry Pi is provided. While the Arduino platform is designed around a relatively low power micro-controller that gives the user complete control of its hardware, devices like the Raspberry Pi (or similarly BeagleBoard) are designed to function on a much higher level: with already integrated hardware that takes care of things like ethernet, large quantities of RAM and an almost unlimited amount of storage space, they are really mini-computers. Hence, it is possible to run complete operating systems, like Linux and Android, and develop programs within those operating systems that can control the systems functions and the input-output (IO) ports.

The general-purpose-computer features of the Raspberry Pi are highly meaningful to the present work. Before discussing this particular hardware in more detail, though, let us revise the standard approaches for flight software development and hardware-in-the-loop testing.

3.2. COTS Software Model-Based Development

The complete engineering process to develop an aerospace FSW suite and test it on the flight processor is standardly 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 15) 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 auto-coding). 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. The advantage of targeting a middleware is that this process ensures portability among different processors and RTOS. Nevertheless, it also has the most overhead and therefore it is not usually used in small missions. The Core Flight System (CFS) is an open-source pub/sub middleware provided by NASA Goddard Spaceflight Center. The CFS is a common tool used in any mission that chooses to incorporate a middleware layer and it has inherited contributions from multiple NASA centers and previous flight missions.

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 present 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.
- **Mission-specific flight target:** Target processor boards

are selected based upon allocated performance, memory and IO requirements. It is then necessary to integrate (or create) an Integrated Development Environment that supports the specific target processor with compiler, linker/loader, onboard debugger, profiler, board-support package and real-time operating system. This effort is linked exclusively to the specific processor and operating system chosen, and although it is less cumbersome than targeting a middleware layer, it is totally inflexible.

- **CFS:** The abstraction from a specific RTOS and processor can be handled through middleware. With such capability, the CFS successfully saves missions the cost of dealing with middleware tasks themselves. Nevertheless, the CFS also presents some caveats: Firstly, it requires the mission-specific FSW applications to be written in C (C++ is not natively supported). Secondly, it demands considerable efforts to integrate a new mission FSW application due to some inflexibilities on the architectural design of the CFS's executive layer, the core Flight Executive (cFE). For more details on the CFS and cFE, the reader should refer to Ref. 16) and Ref. 17).

4. Basilisk-Pi Based Development

The goal of the present work is to propose an effective approach to substantially reduce costs of space exploration by combining COTS technology to increase computational power with agile FSW development to simplify algorithm implementation and reduce expenses on the validation and verification, while ensuring robustness and reliability of the complete flight application. To this end, it is proposed to use the Raspberry Pi as the flight target (aka on-board computer or OBC) and the Basilisk Software Framework as the tool to develop and test the flight algorithms. The combination of these two tools dovetails into a powerful solution to address the current limitations of traditional aerospace processors (relevant to the pursuit of autonomy) as well as all the concerning points of FSW development and flight targeting that have been highlighted in the previous section.

4.1. The Raspberry Pi On-Board Computer

Being small, powerful, and low-cost, with large community support, Raspberry Pi's are a very attractive candidate for low-cost space exploration, provided that they can operate reliably in space. Therefore, the Raspberry Pi hardware is the choice of the present proposal to be used as the on-board computer, knowing that two main challenges still need to be adressed in future work: Firstly, there is the fact that the processor of the Pi is not proven to be radiation tolerant and secondly there is the fact that the Operating System running on the Pi is based on the Linux distribution, which is not real-time. Since the focus of the present work are deep-space missions that are not intended for human-spaceflight, the need of going hard real-time does not seem as relevant as ensuring robustness to recover from time-lapses. As mentioned earlier, one way of assessing both problems and making sure that the Raspberry Pi can operate reliably in space is through software redundancy: if multiple modules are used, then if one of them fails, another can take over (similar to the redundancy approach used for processors in human spaceflight).

Figure 1 displays the flagship Raspberry Pi model B that will be used in the technology demonstration of the present work. The question that still remains is whether a CubeSat mission

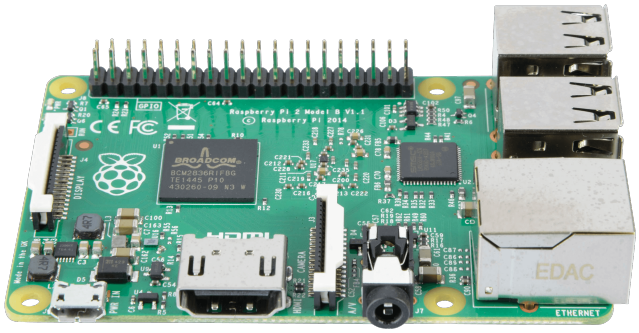

Fig. 1. The flagship Raspberry Pi model B.

would actually fly the full board that appears in Fig. 1 or not. And the answer is, of course, not. We must recall that an aerospace flight system is an embedded system, which means that the onboard computer is just a part of a larger, more complex system that includes other hardware and mechanical parts.

The good news are that, with the Raspberry Pi platform being now grown and matured and its software full-featured and stable, the Pi foundation has recently made a step forward towards industrial application by releasing the so-called Compute Module (CM). The CM is essentially a Raspberry Pi in a more flexible form factor, suitable to be embedded into larger systems and commercial products - like aerospace flight systems! In more technical detail, the CM contains the guts of a Raspberry Pi (in terms of the processor and Mbyte of RAM) with onboard memory (Flash memory device connected), all integrated onto a small 67.6×30mm board whose connectors you can customize for your own needs.

The CM is primarily designed for those who are going to create their own Printed Circuit Board (PCB), which is what spacecraft missions would do. Nevertheless, the purpose of the present work is to provide a technology demonstration of the Basilisk-developed flight algorithms running on the on-board computer (i.e. the flight processor). Given that the flagship Raspberry Pi model B and the Compute Module share the same processor, the validity of the results is not dependant on using either one or the other. The advantages of using the full model B in the present experiment, and also during the software verification part of any mission, is that the outputs of the flight software algorithms can be easily pulled out and analyzed. This strategy will be shown in the numerical simulation of section **5.**.

4.2. The Basilisk Software Framework

Next an overview of the Basilisk Software Framework is provided and, following, the main advantages of the tool in a practical level are summarized.

4.2.1. General Overview

The Autonomous Vehicle System (AVS) Laboratory at the University of Colorado and the Laboratory for Atmospheric and Space Physics (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 writ-

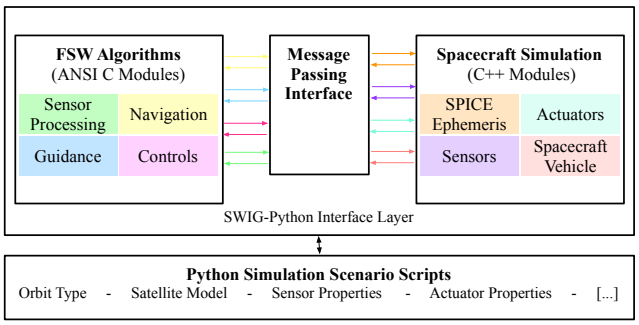

Fig. 2. Architecture of the Basilisk astrodynamics platform.

ten 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. 2, 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 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.

4.2.2. Advantages of Basilisk for FSW Mission Development

In overall, 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 and layers 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[18] or solar radiation pressure effects.[19]
- **Speed:** The fact that the underlying simulation executes entirely in C/C++ allows for maximum execution speed in faster than realtime simulation with built-in repeatable Monte Carlo capability.

As a matter of fact, the Basilisk software framework is currently being applied for: advanced astrodynamics student research; a CubeSat feasibility analysis project between the University of Colorado Boulder and an industrial partner; and a current mission that the Laboratory for Atmospheric and Space Physics is participating in.

4.3. Targeting Basilisk to the Raspberry Pi

The Raspberry Pi has a built-in ARM processor and comes with the Linux Operating System out-of-the-box. A main advantage of developing the flight algorithms with the Basilisk framework and then targeting them on the Raspberry Pi is that no auto-coding step is necessary, since Basilisk is cross-platform in nature and runs very well on the Pi. Therefore, the ability to run on the flight target the same exact hand-written, optimized, algorithms developed in the desktop environment becomes *da facto*. This approach is more reliable and efficient than the traditional GN&C model-based development, avoiding all the cumbersome tasks of targeting a traditional processor and RTOS.

Previous work has demonstrated how Basilisk-developed algorithms can also be integrated to the CFS middleware and embedded onto a flight target running a standard RTOS with closed-looped dynamics.[20] Transitioning from the Basilisk environment to the CFS is the strategy currently being targeted by LASP for a subsystem of a mission in development. In the context of low-cost missions, though, the present proposal involving the Raspberry Pi is more suitable, being straight-forward to implement and saving considerable efforts. Therefore, it is the focus of the present work.

4.3.1. Technology Demonstration

A novel low-cost platform for FSW development and flight targeting has been proposed. Now it will be explained, more technically, how the flight algorithms targeted to the OBC (Raspberry Pi) can be realistically tested and analyzed in a closed-loop dynamic simulation. In summary, this technology demonstration features:

- **Onboard Computer:** out-of-the box Raspberry Pi.
- **Onboard FSW:** Basilisk FSW class (flight software architecture that is easy reconfigurable and scalable), running on the Pi.
- **DKE Simulation Models:** Basilisk Dynamics engine (simulation architecture that is natively modular and contains high-fidelity models), running on a separate host computer.
- **Communication:** in addition to the publish/subscribe messaging bus that is used by all C/C++ modules, a TCP/IP interface is included to enable asynchronous communication between the FSW process (running on the Pi's processor) and the dynamics simulation process (running on a separate host computer). The communication between the on-board computer and the simulated hardware takes place via a message router that allows synchronization to realtime via software-based clock tracking modules.

The fully-realized system just described is depicted in Fig. 3

and its validity is demonstrated through a numerical simulation in section **5.**.

At this point it seems important to mention that both the Basilisk dynamics engine and the FSW process can perfectly run on the Raspberry Pi. Of course, the maximum speed feasible on the Pi is significantly less than the speed achievable on a fully-sized desktop computer. Ideally, the GN&C algorithms are first developed and tested in closed-loop dynamics simulation at full speed, by running both FSW and dynamics processes on the desktop computer synchronously (with no TCP client nor server). Once the required functionality of the flight algorithms is achieved, tested and verified, then it makes sense to separate the process by setting up a TCP communication with a dynamics server and a FSW client that asynchronously connects to it, and to use a processor for the flight algorithms that resembles one you would actually fly. Only at this stage it is sensible to include as well a CPU-based clock tracking module that ensures real time, deterministic behavior (emulating realistic asynchronous software-hardware interfacing).

4.3.2. A Comparison: targeting the CFS to the Pi

Previously, the Pi-Sat project has been briefly mentioned. The concept behind this initiative is very similar to the one proposed in the Basilisk-Pi platform, although there are major pragmatical differences in terms of their actual implementation. Now that all the characteristics of the Basilisk-Pi technology demonstration have been revealed and that all the critical concepts regarding COTS software and hardware have been presented, it makes sense to provide a technical comparison between the Pi-Sat project (where the CFS is targeted to the Raspberry Pi) and the Basilisk-Pi project.

Let us first recall about the Pi-Sat platform: the Pi-Sat is an initiative recently taken by the Flight Software System branch at NASA Goddard Space Flight Center that encompasses the incorporation of Raspberry Pi hardware into a version of their CubeSats. The new Pi-Sat is a low cost platform that combines a credit card-sized ARM processor (the Raspberry Pi's), a suite of low-cost sensors, a 3D-printed enclosure and battery as well as the Core Flight System as the on-board FSW set. Three different designs of the Pi-Sat have been created for different purposes:

- **Pi-Sat Cube:** 1U CubeSat prototype with the CPU of the Raspberry Pi model B.
- **Pi-Sat Wirless Node:** Wireless mesh network with peer-to-peer communication between the Raspberry Pi and a ground-system prototype.
- **Pumpkin Pi Card:** Processor card of a 1U CubeSat prototype: the CPU is the Compute Module. Due to the use of the CM, the Pumpkin Pi Card is more realistic than the Pi-Sat Cube, and gets closer to integrating into a real CubeSat stack.

The Pi-Sat project has a couple of characteristics that make it intrinsically distinct from the proposed Basilisk-Pi platform: On the FSW side, all the Pi-Sat designs use the CFS framework, which includes the cFE executive layer and inherited mission-specific applications. While the Basilisk FSW architecture has been designed in a modular fashion that allows combining already existing modules with own-developed ones, the CFS mission-specific applications that are available are few and monolithic (mainly inherited stand-alone FSW sets from previous missions). In terms of the spacecraft's dynamics representation, there are no DKE simulations models on the Pi-Sat platform, and a suite of sensors is integrated instead. The sensors approach, though, poses challenges in the testing of the flight algorithms because expensive testbeds would be needed in order to replicate the space environment in a comprehensive and realistic manner (e.g. recreation of dynamic forces and torques either from spacecraft actuators or from environmental dynamics). In the context of Cube-Sat missions where resources are scarce, the requirement of testbeds is potentially problematic, even if the sensors themselves are low-cost. Therefore, the simulated hardware approach, together with the clock-tracking modules and a realistic TCP/IP interface, seems more suitable for the purposes of low-cost FSW prototyping and flight targeting.

5. Numerical Simulation

A simple numerical simulation is presented next that serves to demonstrate the validity of the platform architecture described in the previous section. Figure 3 shows the different processes (FSW process, Spacecraft Simulation process and Clock Synchronization process) and the particular modules included in this setup. The messaging connections between the modules are also represented in Fig. 3 through arrows. The Message Passing Interface or MPI is used for communication between modules running in the same process, whereas the TCP server/client connection is used to interface the Raspberry Pi and the host computer asynchronously in real-time.

A key validation step of the platform depicted in Fig. 3 is to show that the results from the numerical simulation in the asynchronous dual-process system, i.e. Raspberry Pi (FSW algorithms) - host computer (spacecraft simulation), coincide exactly with the results obtained in a single-process system where FSW and simulation run synchronized in the desktop host computer.

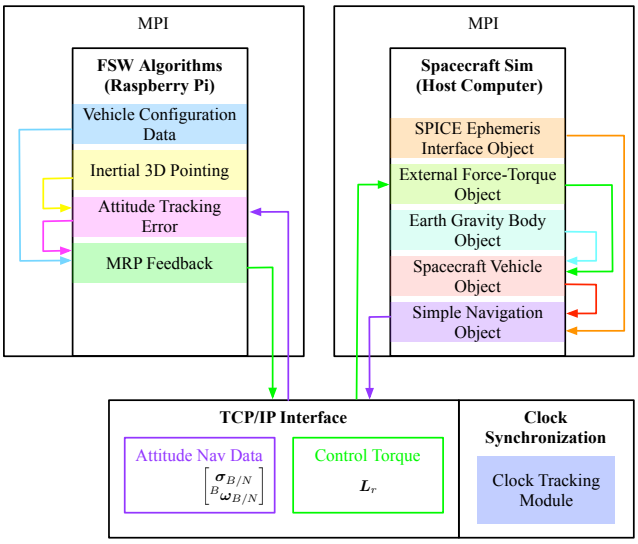

Fig. 3. Numerical simulation setup.

5.1. Scenario

The spacecraft is simulated to be orbiting Earth with the orbital parameters in Table 1, and the spacecraft's Attitude, Dynamics and Control (ADC) parameters in Table 2. Note that the Modified Rodrigues Parameter (MRP) sets, designed as $\boldsymbol{\sigma}$, are used to represent attitude states throughout the simulation.

Table 1. Initial Orbital Elements

Parameter	Value	Units
Semi-major Axis	10,000	km
Eccentricity	0.2	
Inclination	0.0	deg
Longitude of Ascendant Node	0.0	deg
Argument of Perigee	0.0	deg
True Anomaly	280.0	deg

Table 2. ADC Parameters

Parameter	Value	Units
Attitude Error Gain K	3.5	$\frac{\text{kg}\cdot\text{m}^2}{\text{s}}$
Rate Error Gain P	30.0	$\frac{\text{kg}\cdot\text{m}^2}{\text{s}}$
I_1	900	$\text{kg}\cdot\text{m}^2$
I_2	800	$\text{kg}\cdot\text{m}^2$
I_3	600.0	$\text{kg}\cdot\text{m}^2$
Inertial Reference Attitude $\boldsymbol{\sigma}_{R/N}$	$[0.1, 0.2, -0.3]$	
Inertial Reference Rate $\boldsymbol{\omega}_{R/N}$	$[0.0, 0.0, 0.0]$	rad/s

The spacecraft's inertia is a diagonal matrix with principal components I_1, I_2, I_3 as defined in Table 2. The initial conditions of the spacecraft are given by

$$\boldsymbol{\sigma}_{B/N}(t=0) = [0.1, 0.2, -0.3] \tag{1a}$$
$$\boldsymbol{\omega}_{B/N}(t=0) = [0.001, -0.01, 0.03] \text{ rad/s} \tag{1b}$$

where $\mathcal{B}$ refers the spacecraft's principal body frame and $\mathcal{N}$ indicates a generic inertial frame. The simulation time is $t_{\text{sim}} = 6$ min and the goal of the simulation is to perform a detumbling maneuver and align the spacecraft with a fixed inertial reference attitude described by $\boldsymbol{\sigma}_{R/N}$. Note that here the subscript $\mathcal{R}$ indicates the reference.

5.2. Dynamics

The rigid body equations of motion in Eq. (2) are numerically integrated within the Basilisk dynamics engine.[21)]

$$\dot{\boldsymbol{\sigma}} = \frac{1}{4}\left((1-\sigma^2)[I_{3\times3} - 2[\tilde{\boldsymbol{\sigma}}] + 2\boldsymbol{\sigma}\boldsymbol{\sigma}^T\right)\boldsymbol{\omega} \tag{2a}$$
$$[I]\dot{\boldsymbol{\omega}} = -\boldsymbol{\omega}\times[I]\boldsymbol{\omega} + \boldsymbol{L} - \boldsymbol{L}_r \tag{2b}$$

where $\boldsymbol{\sigma} \equiv \boldsymbol{\sigma}_{B/N}$ is the attitude of the spacecraft's principal body frame $\mathcal{B}$ with respect to the inertial $\mathcal{N}$ frame, $\boldsymbol{\omega} \equiv \boldsymbol{\omega}_{B/N}$ is the spacecraft's inertial angular rate, $[I]$ is the spacecraft inertia tensor, $\boldsymbol{L}$ is an external torque and $\boldsymbol{L}_r$ is the applied closed-loop control torque.

The dynamic states are integrated using a 4$^{\text{th}}$ order Runge-Kutta scheme running at 10 Hz. The closed-loop control torque is directly applied to the spacecraft (i.e. not mapped to the set of spacecraft's effectors) for the sake of simulation simplicity. The Earth is the only celestial body affecting the spacecraft's orbit and spherical harmonics are disabled in the present numerical simulation.

5.3. Flight Algorithms

For the purpose of testing the FSW modules, the following MRP feedback law is implemented that is globally asymptotically stabilizing:[21)]

$$\boldsymbol{L}_r = K\boldsymbol{\sigma}_{B/R} + [P]\boldsymbol{\omega}_{B/R} - \boldsymbol{\omega}_{R/N}\times[I]\boldsymbol{\omega}_{B/N} + \\ + [I](\boldsymbol{\omega}_{B/N}\times\boldsymbol{\omega}_{R/N} - \dot{\boldsymbol{\omega}}_{R/N}) + \boldsymbol{L} \tag{3}$$

Here $\boldsymbol{L}_r$ is the control torque being computed, $\boldsymbol{\sigma}_{B/R}$ is the spacecraft attitude tracking error, $\boldsymbol{\omega}_{B/R}$ is the spacecraft rate error, $\boldsymbol{\omega}_{R/N}$ is the reference inertial angular rate, $\dot{\boldsymbol{\omega}}_{R/N}$ is the reference inertial angular acceleration, K is the attitude error gain and $[P]$ is the rate error gain matrix.

The simulated navigation data $\boldsymbol{\sigma}_{B/N}$ and $\boldsymbol{\omega}_{B/N}$ is without any sensor corruptions to better illustrate that the control law does achieve asymptotic tracking of the reference motion.

5.4. Plots from the Asynchronous Dual-Process System

In this section plots of the dynamic simulation and of the GN&C performance are presented when the dual-process system is used, i.e the host computer (where the dynamic simulation runs) and the Raspberry Pi (where the GN&C algorithms run) communicate **asynchronously**. The TCP interface is used and communication is controlled to be in **real time**.

The results of the physical dynamic simulation of the spacecraft are pulled from the C++ level to the Python interface in the host computer and automatically plotted. The simulation analysis plots showing the spacecraft's orbit and the spacecraft's rotational states appear in Fig. 4, Fig. 5 and Fig. 6.

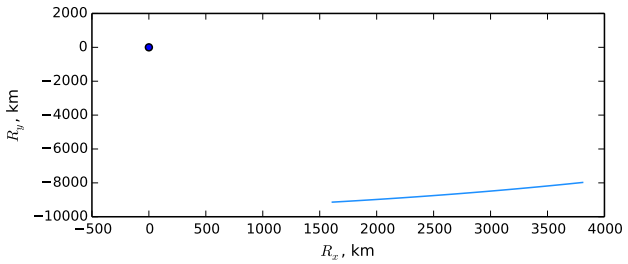

Fig. 4. Spacecraft's orbit. Results plotted on the host computer from the asynchronous dual-process run.

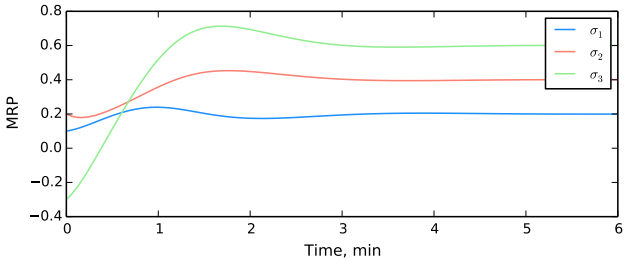

Fig. 5. Spacecraft's inertial MRP attitude set, $\boldsymbol{\sigma}_{B/N}$. Results plotted on the host computer from the asynchronous dual-process run.

Similarly, the output of the flight software algorithms running on the Raspberry Pi are retrieved from the C level to the Raspberry's built-in Python interface and plotted for analysis. The flight software plots showing the performance of the GN&C algorithms appear in Fig. 7, Fig. 8 and Fig. 9.

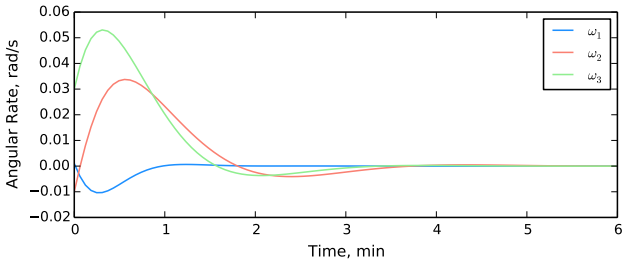

Fig. 6. Spacecraft's inertial angular rate, ${}^{B}\boldsymbol{\omega}_{B/N}$. Results plotted on the host computer from the asynchronous dual-process run.

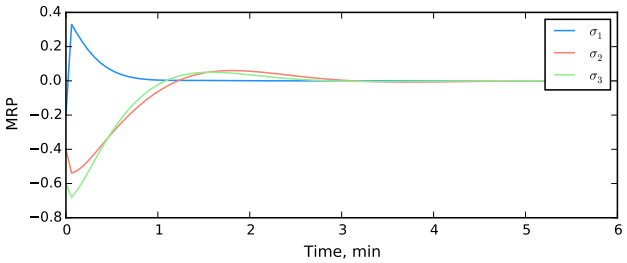

Fig. 7. Attitude tracking error MRP set, $\boldsymbol{\sigma}_{B/R}$. Results plotted on the Pi from the asynchronous dual-process run.

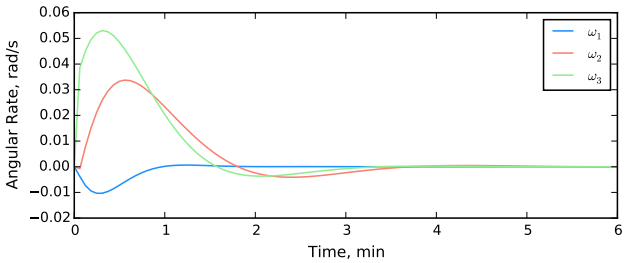

Fig. 8. Angular rate error, ${}^{B}\boldsymbol{\omega}_{B/R}$. Results plotted on the Pi from the asynchronous dual-process run.

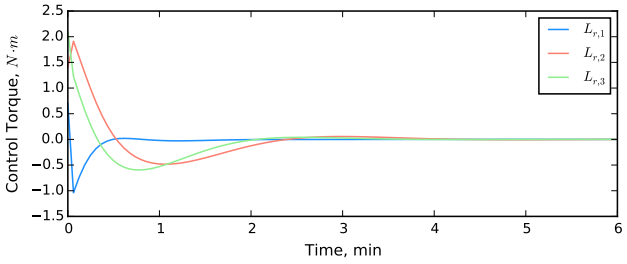

Fig. 9. Control torque, $\boldsymbol{L}_r$. Results plotted on the Pi from the asynchronous dual process run.

5.5. Plots from the Synchronous Single-Process System

In this section plots of the GN&C performance are presented when the single-process system is used, i.e both the dynamic simulation and the GN&C FSW tasks run **synchronously** at **full speed** in the desktop environment (host computer) and no TCP interface is used.

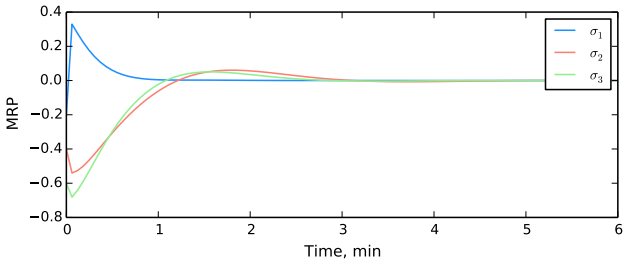

Fig. 10. Attitude tracking error MRP set, $\boldsymbol{\sigma}_{B/R}$. Results plotted on the host computer from the single-process run.

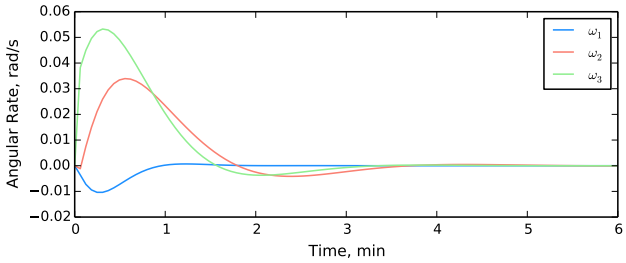

Fig. 11. Angular rate error, ${}^{B}\boldsymbol{\omega}_{B/R}$. Results plotted on the host computer from the single-process run.

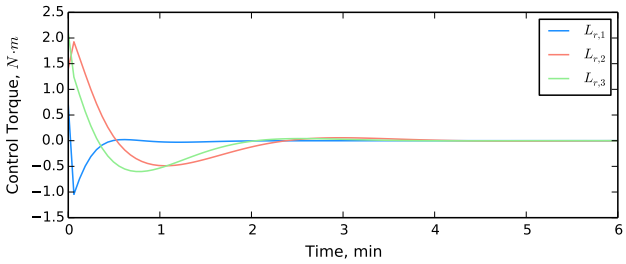

Fig. 12. Control torque, $\boldsymbol{L}_r$. Results plotted on the the host computer from the single-process run.

5.6. Results Discussion

The FSW analysis plots in Fig. 7, Fig. 8 and Fig. 9 showcase how the autonomous guidance maneuver is successfully achieved by bringing the spacecraft from its initial tumbling state to the desired inertial fixed-pointing attitude. In particular, this numerical scenario in the asynchronous dual-process mode shows how the flight software algorithms running on the Pi are able to infer/read the spacecraft's states (attitude and rate) from the dynamics engine running on the host computer, using the Message Passing Interface (MPI) and TCP interface. According to the current spacecraft's states and the desired reference states, a tracking error is computed on the Pi flight algorithms and a control torque is derived. In turn, the computed control torque on the Pi is then commanded, through the MPI and TCP communication, to the simulated spacecraft on the host computer. In this way it is demonstrated how the flight algorithms on the OBC are readily tested in a highly realistic closed-loop dynamics simulation.

Finally, a key strength of the proposed Basilisk-Pi platform is revealed when comparing the GN&C FSW plots in the asynchronous dual-process system (i.e. Fig. 7, Fig. 8 and Fig. 9) with the GN&C FSW plots in the synchronous single-process system (i.e. Fig. 10, Fig. 11 and Fig. 12). By comparison it is shown that running the two different systems yields the same

exact results. Obtaining identical results proves the validity of the architecture including the TCP/IP interface and shows that the performance of the flight algorithms is preserved when they are targeted to the flight processor; as pointed out earlier, since the Basilisk-Pi approach does not encompass any auto-coding step, there is then no lost in efficiency of the flight algorithms. Further, since the Pi runs human-written algorithms, it is possible to make changes directly on the FSW suite of the on-board processor, which allows for faster and more flexible testing. Last but not least, the parity of the results provides higher credibility and reliability to the full-speed Monte Carlo simulations that can be ran on the desktop environment with the purpose of testing a wide range of scenarios and what-if situations, as long as the Raspberry Pi is indeed the flight target.

Conclusions

The paper demonstrates how the Basilisk framework can be effectively used to first test and analyze flight software in a very user-friendly software environment and, then, easily integrate it into a flight target, the Raspberry Pi, for the next phase of software-hardware testing with high fidelity, real-time, closed-looped dynamics. Future work encompasses hardening the internals of the Basilisk-Pi system to ensure robustness against space radiation through software redundancy and to address real-time metrics reliability on the Pi on-board computer.

Acknowledgments

Thanks to Andrew T. Harris for his support and insight on the workings of the Linux Operating System.

References

1) Nickolaos Mastrodemos, a. O. J. and Rush, B., "Optical Navigation for the Rosetta mission," *Annual AAS Guidance and Control Conference*, Breckenridge, CO, 2015.
2) Bhaskaran, S., Riedel, J. E., and Synnott, S. P., "Autonomous target tracking of small bodies during flybys," *AAS/AIAA Spaceflight Mechanics Meeting*, Maui, Hawaii, United States, Feb 8–12 2004.
3) Fuchs, T., Thompson, D. R., Bue, B., Castillo, J., Chien, S., Gharibian, D., and Wagstaff, K., "Enhanced Flyby Science with Onboard Computer Vision: Tracking and Surface Feature Detection at Small Bodies," Tech. rep., Earth and Space Science, 2015.
4) Kominato, T., Matsuoka, M., Uo, M., Kawaguchi, J., Kubota, T., and Hashimoto, T., "HAYABUSA's Optical Hybrid Navigation for Approaching to and Stationkeeping around Asteroid ITOKAWA," *The Journal of Space Technology and Science*, 2006.
5) Olds, R., May, A., Mario, C., Hamilton, R., Debrunner, C., and Anderson, K., "The Application of Optical Tracking to OSIRIS-REx Asteroid," *AAS*, 2015.
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) Keys, A., Watson, M., Frazier, D., Adams, J., Johnson, M., and Kolawa, E., "High Performance, Radiation-Hardened Electronics for Space Environments," *5th International Planetary Probes Workshop*, Bordeaux, France, June 28 2007.
9) Royce, W., "Current Problems," *Aerospace Software Engineering: A Collection of Concepts*, edited by C. Anderson and M. Dorfman, Vol. 136, Progress in Aeronautics and Astronautics, AIAA, 1991, pp. 5–15.
10) Mastrodemos, N., Kubitschek, D., and Synnott, S., *Deep Impact Mission: Looking Beneath the Surface of a Cometary Nucleus*, chap. Autonomous Navigation for the Deep Impact Mission Encounter with Comet Tempel 1, Springer Netherlands, 2005, pp. 95–121.
11) Dingsoyr, T., Moe, N. B., Nerur, S. P., and Balijepally, V., "A decade of agile methodologies: Towards explaining agile software development," *Journal of Systems and Software*, Vol. 85, June 2012, pp. 1213–1221.
12) Briggs, M., Benz, N., and Forman, D., "Simulation-Centric Model-Based Development for Spacecraft and Small Launch Vehicles," *32nd Space Symposium*, Colorado Springs, Colorado, April 11–12 2016.
13) Violette, D., "Arduino/Raspberry Pi: Hobbyist Hardware and Radiation Total Dose Degradation," *EEE Parts for Small Missions*, Greenbelt, MD, September 10-11 2014.
14) Cudmore, A., "Pi-Sat: A Low Cost Small Satellite and Distributed Spacecraft Mission System Test Platform," Tech. rep., NASA Goddard Space Flight Center, Greenbelt, MD, September 9 2015.
15) 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.
16) Cudmore, A., "NASA/GSFC's Flight Software Architecture: Core Flight Executive and Core Flight System," *Flight Software Workshop*, 2011.
17) McComas, D., "NASA/GSFC' Flight Software Core Flight System," *Flight Software Workshop*, San Antonio, TX, Nov. 7–9 2012.
18) 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.
19) 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.
20) 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.
21) Schaub, H. and Junkins, J. L., *Analytical Mechanics of Space Systems*, AIAA Education Series, Reston, VA, 3rd ed., 2014.