

**End-to-End Flight Software Development and Testing:  
Modularity, Transparency and Scalability across Testbeds**

by

**Mar Cols-Margenet**

B.S., Polytechnic University of Catalonia, 2015

Doctoral committee members: Prof. Hanspeter Schaub, Prof. Daniel Kubitschek, Prof. Jay  
McMahon, Prof. Marcus Holzinger, Prof. Hans-Juergen Herpel

A thesis submitted to the  
Faculty of the Graduate School of the  
University of Colorado in partial fulfillment  
of the requirements for the degree of  
Doctor of Philosophy  
Department of Aerospace Engineering Sciences  
2020

Cols-Margenet, Mar (Ph.D., Aerospace Engineering Sciences)

End-to-End Flight Software Development and Testing: Modularity, Transparency and Scalability  
across Testbeds

Thesis directed by Prof. Hanspeter Schaub

This thesis investigates end-to-end flight software (FSW) development strategies and working implementations that support having both desktop and embedded environments separately while ensuring 1) transparent migration of the flight application and 2) consistent testing throughout different testbeds. In order for the flight algorithm migration to be transparent, it is critical that the source code remains unchanged. Regarding consistent and high-fidelity testing throughout environments, such an endeavour can be effectively achieved by making use of a distributed communication architecture. The idea behind a distributed architecture is to allow integration of heterogeneous and independent mission components into a single simulation run that works seamlessly whether FSW executes from desktop or embedded environments.

The term end-to-end used in this thesis implies that the entire FSW development cycle is covered: starting from a preliminary desktop design and analysis all the way to testing on the flight hardware –or rather, its emulated counterpart. The emulation of embedded systems is particularly interesting because it provides pure software substitutions for expensive hardware components of limited quantity that might be needed simultaneously for testing by different mission groups.

The different aspects of the FSW development process that are covered in this work are briefly outlined following: desktop flight algorithm design and testing, migration of the flight application into several flight targets (commercial processors as well as middleware layers), distributed closed-loop simulations by means of a modern communication architecture, embedded development and testing in a realistically emulated flat-sat and, finally, profiling of memory and CPU resources for a modern yet embeddable FSW application.

All the development proposals and strategies pursued in this work consider exclusively open-



source products and strive for the embedded system to be as close as possible to the desktop testbed in terms of user friendliness and interaction functionalities, while still adhering to the needs of space: determinism, concurrency and low use of resources. Currently, deploying an embedded flight system and migrating flight algorithms on it is not an easy task. However, many small-satellite missions or start-up companies without extensive FSW legacy would highly benefit from having available an end-to-end FSW development tool suite like the one designed and proved in this thesis.

## Dedication

To Pete Balsells, a true source of inspiration both professionally and personally, who has opened the doors of the University of Colorado Boulder to many citizens of Catalonia like myself through the Balsells fellowship. To Dean Wang, the most influential English professor I have ever had and without whom I would, most likely, not be in the United States today. To Hanspeter Schaub, who gave me the unique opportunity to become a PhD student in the Autonomous Vehicle Systems laboratory. To Scott Piggott, my technical mentor, from whom I have learnt the most valuable skills I have today. To Matt Kappnius, who has taught me how to find comfort in discomfort and how to bring awareness into my own life. To Chris Kiehl, who keeps inviting me to step out of my comfort zone and from whom I have learnt that most important things in life are not taught in school. To my mother, for her unconditional support in every single stage of my life.

## Acknowledgements

Thanks to Scott Piggott for his technical advice, supervision and support throughout every single endeavour pursuit and presented in this thesis. Scott, you are an amazing flight software engineer and without you none of this work would have been possible. Thanks to Hanspeter Schaub for all his insight on spacecraft dynamics and controls, for sending me to conferences across the globe to present my research and for believing in my potential until the end, despite all the ups and downs. Thanks to Patrick Kenneally, with whom I had the pleasure to begin the Black Lion project. Thanks to Wayne Sidney and Brian Kirby, users of the emulated flat-sat and contributors to the tests performed.

## Contents

### Chapter

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background: FSW Development Environments . . . . .	4
1.1.1	Desktop Development Environment . . . . .	4
1.1.2	Embedded Environment . . . . .	11
1.1.3	Middleware Layers . . . . .	13
1.2	Literature Review: State-of-the-art Tools for FSW Development . . . . .	14
1.2.1	Modular Desktop Development Tools . . . . .	14
1.2.2	Cross-Environment Development Tool Suites . . . . .	16
1.3	Outline . . . . .	18
<b>2</b>	<b>Desktop Flight Algorithm Development: Modular ADCS</b>	<b>21</b>
2.1	The Basilisk Testbed . . . . .	21
2.2	Rotational Reference Motions for Distinct Guidance Profiles . . . . .	23
2.2.1	Problem Statement . . . . .	24
2.2.2	Software Modules and Mathematical Development . . . . .	26
2.2.3	Numerical Simulations . . . . .	34
2.3	Summary . . . . .	44
<b>3</b>	<b>Flight Algorithm Migration into Commercial Flight Targets</b>	<b>45</b>
3.1	The Raspberry Pi for Space Applications . . . . .	47

3.2	Distributed Basilisk Simulation using the Raspberry Pi . . . . .	47
3.3	Summary . . . . .	51
<b>4</b>	<b>Flight Algorithm Migration into the core Flight System</b>	<b>53</b>
4.1	The cFS Middleware . . . . .	54
4.2	From Basilisk into a cFS-FSW Application . . . . .	55
4.2.1	Setup Item: C Module Initialization . . . . .	55
4.2.2	Setup Item: Task Groups and Rates . . . . .	57
4.3	Translation Mechanism: the Auto-Setter . . . . .	58
4.4	Summary . . . . .	61
<b>5</b>	<b>Emulated Flat-Sat Testing of cFS-FSW through Distributed Communication</b>	<b>63</b>
5.1	Emulated Flat-Sat . . . . .	64
5.2	The Black Lion Communication Architecture . . . . .	67
5.2.1	Design and Architecture . . . . .	67
5.2.2	Data Transfer and Synchronization . . . . .	69
5.2.3	Applications . . . . .	71
5.3	FPGA registers and Avionics Hardware Modeling . . . . .	73
5.3.1	Register Space . . . . .	74
5.3.2	Avionic Models . . . . .	76
5.4	Emulated Flat-Sat Simulations . . . . .	76
5.4.1	Spacecraft Pointing Commands . . . . .	77
5.4.2	Mars Orbit Insertion . . . . .	80
5.4.3	Coarse Sun Sensors Corruption/Miscompare . . . . .	83
5.5	Formation Flying Simulation through Black Lion . . . . .	85
5.6	Summary . . . . .	90

<b>6</b>	<b>Flight Algorithm Migration into MicroPython</b>	<b>93</b>
6.1	From Basilisk into a MicroPython application . . . . .	94
6.2	Migration Mechanism: the Auto-Wrapper . . . . .	96
6.3	Post-Processing MicroPython Results . . . . .	97
6.4	Numerical Simulation: Testing Basilisk-MicroPython FSW . . . . .	99
6.5	Summary . . . . .	100
<b>7</b>	<b>Basilisk-MicroPython for Embedded Systems</b>	<b>103</b>
7.1	Port of MicroPython to the LEON Flight Target . . . . .	104
7.1.1	Building the RTEMS toolchain and LEON BSP . . . . .	106
7.1.2	Building the MicroPython executable . . . . .	107
7.1.3	Summary . . . . .	110
7.2	Basilisk-MicroPython Profiling: Use of Resources . . . . .	111
7.2.1	RAM Usage . . . . .	112
7.2.2	ROM Usage . . . . .	125
7.2.3	CPU Usage . . . . .	126
7.2.4	Summary . . . . .	127
<b>8</b>	<b>Conclusions</b>	<b>129</b>
	<b>Bibliography</b>	<b>135</b>
	<b>Appendix</b>	
<b>A</b>	<b>Modular Attitude Guidance</b>	<b>139</b>
A.1	Attitude Control MRP Feedback . . . . .	139
A.2	Additional Base Pointing Modules . . . . .	141
A.2.1	Inertial Pointing . . . . .	141

A.2.2	Hill and Velocity Pointing . . . . .	141
<b>B</b>	Python-based Introspection Tools	<b>145</b>
B.1	Auto-setter . . . . .	145
B.2	Auto-wrapper . . . . .	148
<b>C</b>	Black Lion Data Transfer: ZMQ	<b>153</b>
C.1	Socket Patterns . . . . .	153
C.2	Connection Types . . . . .	154
C.3	Controller Requests and Node-Delegate Replies . . . . .	156
<b>D</b>	Building the Basilisk-MicroPython FSW System for Unix	<b>157</b>
<b>E</b>	Benchmarking Heap Memory Usage	<b>160</b>
E.1	Sensor example in Python . . . . .	162
E.2	Sensor example in MicroPython . . . . .	163
E.2.1	MicroPython with Garbage Collector . . . . .	163
E.2.2	MicroPython without Garbage Collector . . . . .	164
E.3	Sensor example in Basilisk-MicroPython without GC . . . . .	166

# **Tables**

## **Table**

2.1	Initial orbital elements . . . . .	35
2.2	Control and spacecraft parameters . . . . .	35
2.3	Configuration data for the Euler angle rotation module . . . . .	36
2.4	Configuration data for the raster manager module . . . . .	39



## Figures

### Figure

1.1	Flight algorithm targets: desktop computer, single-board computer (SBC) emulation and SBC hardware . . . . .	2
1.2	Concept of emulated flat-sat with three components: ground system emulator, SBC emulator and spacecraft physical simulation . . . . .	3
1.3	Evolution of software architectures . . . . .	6
1.4	Model-based development: from model in the loop (MIL), to software in the loop (SIL), up into hardware in the loop (HWIL) . . . . .	8
1.5	Python wrapper with underlying C/C++ code: straight from software in the loop (SIL) to hardware in the loop (HIL) . . . . .	9
1.6	Radiation-hardened microprocessors. Image extracted from [29] . . . . .	12
2.1	Basilisk (BSK) desktop environment . . . . .	22
2.2	Break-down of sample mission profiles . . . . .	24
2.3	Attitude guidance generation . . . . .	24
2.4	Inputs and outputs of a compounded attitude reference chain . . . . .	25
2.5	Constrained celestial two-body pointing scenario . . . . .	27
2.6	Sample scanning patterns . . . . .	30
2.7	Principal body frame (blue) and control frame (magenta) . . . . .	33
2.8	Nadir-spinning stack . . . . .	36

2.9	Nadir spinning: cascaded attitude sets . . . . .	37
2.10	Inertial asterisk scanning stack . . . . .	38
2.11	Asterisk scanning: attitude tracking error and control torque . . . . .	40
2.12	Simulated asterisk scanning patterns . . . . .	40
2.13	Stack of modules for a spiral scanning maneuver . . . . .	41
2.14	Spiral scanning: nominal pointing vs. actual . . . . .	42
2.15	Spiral scanning: tracking error and wheel torques . . . . .	42
2.16	Triple-spiral scanning: nominal pointing vs. actual . . . . .	43
2.17	Stack of modules for a triple-spiral scan . . . . .	43
3.1	Migration of the flight application . . . . .	46
3.2	Numerical simulation setup . . . . .	49
3.3	Results from the spacecraft physical simulation . . . . .	50
3.4	Results from the FSW process on the Raspberry Pi . . . . .	51
4.1	Architecture of the core Flight System . . . . .	54
4.2	Translation of setup code from Python to C . . . . .	58
4.3	Embedded FSW application . . . . .	60
4.4	Emulated FPGA register space . . . . .	60
5.1	Emulated flat-sat components . . . . .	65
5.2	Heterogeneity of flat-sat models . . . . .	66
5.3	Black Lion interfaces: <b>Central Controller</b> and, for each node, <b>Delegate</b> and <b>Router</b> APsd . . . . .	68
5.4	“Tick-tock” synchronization . . . . .	69
5.5	Black Lion application: emulated flat-sat . . . . .	71
5.6	Black Lion application: testing of formation flying concepts . . . . .	72
5.7	cFS-FSW interaction: emulated FPGA registers and avionics hardware models . . .	73

5.8	Detailed view of the emulated FPGA registers and avionics hardware models . . . .	75
5.9	Closed-loop response: spacecraft's main body attitude . . . . .	77
5.10	Closed-loop response: reaction wheel speeds . . . . .	78
5.11	Closed-loop response: instruments pointing . . . . .	79
5.12	Mars orbit insertion scenario . . . . .	81
5.13	Addition of CFDP node for realistic uplink and downlink of data . . . . .	82
5.14	CSS analog-to-digital converters: signal miscompare . . . . .	84
5.15	FDP telemetry packets: faulted converter 1 and primary converter 2 . . . . .	85
5.16	Attitude guidance reference generation: modular vs. distributed . . . . .	86
5.17	Distributed guidance: concept of operations and simulation setup . . . . .	88
5.18	Distributed commanded guidance: connections between modules of the chief and the deputy simulations . . . . .	89
5.19	Stack of attitude guidance modules for the chief and deputy FSW suites . . . . .	89
5.20	Deputy true attitude states . . . . .	90
5.21	Deputy FSW states: tacking error and control torque . . . . .	91
5.22	Relative asterisk pattern: commanded vs. true pointing angles . . . . .	91
6.1	Flight algorithm migration into MicroPython through the <b>AutoWrapper</b> . . . . .	97
6.2	Post-processing Python (desktop) and MicroPython (embedded) execution runs. . .	98
6.3	Closed-loop testing of MicroPython flight algorithms . . . . .	99
6.4	Closed-loop testing of MicroPython-FSW: flight algorithm plots . . . . .	101
7.1	Basilisk-MicroPython: system-level architecture for different targets . . . . .	105
7.2	Port of the stand-alone MicroPython to the RTEMS-LEON target . . . . .	106
7.3	Sample executable for RTEMS-LEON . . . . .	108
7.4	MicroPython executable for RTEMS-LEON . . . . .	108
7.5	MicroPython Makefiles: adapting QEMU-ARM port to QEMU-LEON . . . . .	109
7.6	MicroPython kernel for QEMU's emulation of LEON . . . . .	110

7.7	Heap memory: fragmented vs. compacted . . . . .	113
7.8	View of the Basilisk-MicroPython system and breakdown of the FSW application scripts . . . . .	115
7.9	Sample models, tasks and events . . . . .	116
7.10	Heap memory used at initialization of the FSW application . . . . .	117
7.11	Heap memory used during execution of the FSW application for 40 virtual minutes .	118
7.12	Output of memory-profiler for the <b>Execute</b> call . . . . .	121
7.13	Heap memory used during execution of the FSW application for 400 virtual minutes after removing use of Python dictionaries . . . . .	123
7.14	Basilisk-MicroPython open-loop simulation: inertial spinning . . . . .	124
A.1	Flow between Guidance and Control Blocks . . . . .	139
A.2	Illustration of the Hill and Velocity Orbit Frames . . . . .	142
C.1	Socket patterns between the <b>Central Controller</b> and the nodes' <b>Delegate</b> . . . . .	154
C.2	Socket connections types: binding vs. connecting . . . . .	155
E.1	Sensor example in Python 3 . . . . .	162
E.2	Sensor example in MicroPython with GC . . . . .	164
E.3	Sensor example in MicroPython without GC . . . . .	165
E.4	Sensor example in Basilisk-MicroPython without GC . . . . .	166

## Chapter 1

### Introduction

Space missions rely highly on the efficiency and reliability of the on-board flight algorithms 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 cycle to develop a FSW system encompasses an involved path of deploying and running the flight algorithms within different testbed environments. In a standard spacecraft mission there are three distinct computing environments to consider as flight algorithm targets: desktop computer (for algorithm prototyping and rapid iteration), hardware flight processor (for flat-sat testing and eventually flying) and emulated flight processor in a virtual machine (for emulated flat-sat testing). These environments are illustrated in Fig. 1.1, where the term single board computer (SBC) is used to refer to the flight processor. The two latter environments (i.e. hardware and emulated flight processors) are considered to be embedded. Since a regular desktop computer environment and an embedded flight processor environment are very different in terms of resources, capabilities and end-user programmability, migrating the flight algorithms from one environment to the other generally demands a significant engineering effort. Further, there is also a disparity in the testing tools and procedures that each testbed currently allows. This thesis investigates end-to-end FSW development strategies and working implementations that support having both desktop and embedded environments separately while minimizing the existing gap between them in order to ensure:

- (1) Transparent migration of the flight application.

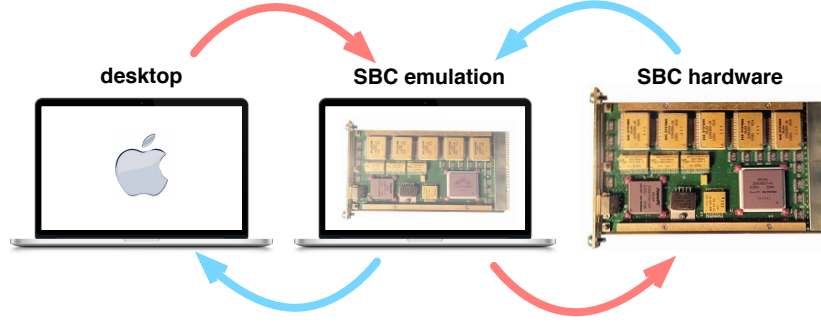


Figure 1.1: Flight algorithm targets: desktop computer, single-board computer (SBC) emulation and SBC hardware

(2) Consistent testing throughout the different testbeds.

The term *end-to-end* used in this thesis implies that the entire FSW development cycle is covered: starting from a preliminary desktop design and analysis all the way to testing on the flight hardware.

Regarding desktop FSW development, this thesis focuses particularly on architecting flight algorithms through modular designs and shared coding standards. At its aim, FSW is intended to support the rest of the system for which it has been designed yet, in practice, it often ends up slaying the other system components due to lacking architecture and proper implementation[5]. The development of inflexible, mission-specific flight algorithms is, indeed, a recurrent and problematic pattern in the aerospace industry that needs to be addressed[41]. Architectural design of flight algorithms takes place in the desktop prototyping phase and the design decisions made here impact portability and testability across all testbeds.

Regarding flight hardware, this thesis puts special emphasis on emulated flat-sat testing of the embedded FSW. The emulation of embedded systems is particularly interesting because it provides pure software substitutions for expensive hardware components of limited quantity that might be needed simultaneously for testing by different mission groups[13, 31, 34].

The transition between desktop and embedded environments is a critical step in which continuity and homogeneity of the FSW testing process is often overlooked. Continuity can be enhanced by ensuring that the migration process is transparent and the algorithm source code remains un-

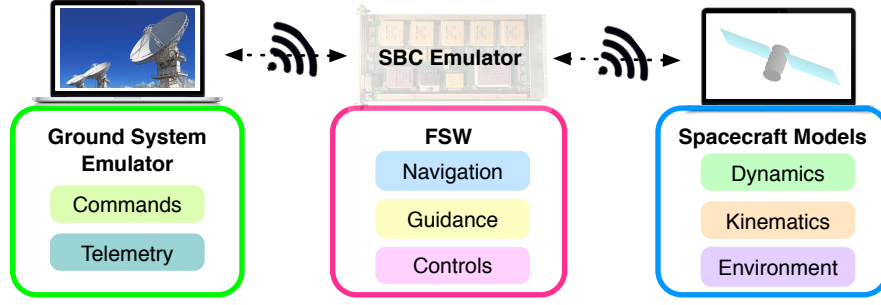


Figure 1.2: Concept of emulated flat-sat with three components: ground system emulator, SBC emulator and spacecraft physical simulation

changed. The underlying idea being to stay as close as possible to the long-held NASA saying of: “testing what you fly, flying what you test” –since the first day of development until the last one, from the desktop all the way into the embedded environment.

Regarding consistent and high-fidelity testing throughout environments, such endeavour can be effectively achieved by making use of a distributed communication architecture. Alongside the prototyping of flight algorithms for a given mission, spacecraft physical models are also built with the purpose of testing the FSW algorithm set in a simulated closed-loop. In a flat-sat configuration, there are additional external models interacting with FSW, like the ground system for example. On these lines, using a distributed communication architecture allows integrating independent mission components into a single simulation run which works seamlessly whether FSW executes from either the desktop or the embedded environment. Having a testbed that is agnostic to FSW being embedded or not is key to homogeneity and continuity of the FSW testing process. In turn, the agnosticism of the testbed is granted by the distributed nature of the underlying communication architecture. The testbed could be composed of hardware-only components, software-only components or a combination of both. Figure 1.2 shows an emulated flat-sat conformed exclusively by software components.

The end-to-end FSW development strategies pursued in this thesis consider exclusively open-source products and strive for the embedded system to be as close as possible to the desktop testbed in terms of user-friendliness and interaction functionalities, while still adhering to the needs of

space: determinism, concurrency and low use of resources. Currently, deploying an embedded flight system and migrating flight algorithms on it is not an easy task. However, many small-satellite missions or start-up companies without extensive FSW legacy would highly benefit from having available an end-to-end FSW development tool suite. An interesting new trend in some missions is to use commercial processors in redundant configurations instead of a single radiation hardened processor[9, 17]. The increasing interest on alternatives to classic radiation-hardened processors reveals the need for improvement in existing embedded flight systems.

As further developed in later sections, the use of middleware can aid portability of the flight application across different targets. Other than middleware layers, there are different frameworks that have been developed in the recent years in order to test FSW in the desktop environment yet in a flight-like manner. For example, in the context of robotic FSW for Mars surface exploration missions, JPL has developed the Surface Simulation (SSim)[46], which uses actual FSW instead of a simplified model to perform rapid desktop testing. A similar testbed developed also at JPL is F-Prime[6], which is specifically designed for small-scale flight systems. This manuscript investigates, in particular, the use of classic and modern middleware layers.

## 1.1 Background: FSW Development Environments

This section aims to provide the reader with further context on desktop and embedded systems. First, the general features of desktop development environments are described and two different methodologies for desktop FSW prototyping are discussed: model-based development and the use of Python wrapping C/C++ flight algorithm code. Next, the features of embedded environments are presented and the concept of middleware is finally developed.

### 1.1.1 Desktop Development Environment

Desktop computers are the most flexible of the environments thanks to the use of state-of-the-art processors and operating systems. This flexibility is shown in terms of computing speed, memory availability, deployability and user friendliness among other. Because of its flexibility, the desktop



environment is used in the preliminary step of prototyping mission-specific flight algorithms. These FSW algorithms are usually tested in closed-loop dynamics simulations with spacecraft physical models until the desired algorithm performance is achieved and mission-specific requirements are met. During the desktop FSW development phase, there are three different aspects that are worth discussing:

- (1) The importance of modular algorithm designs.
- (2) The convenience of using scripting languages for rapid prototyping and iteration.
- (3) The two main implementation approaches adopted: model-based development or Python interface with underlying C/C++ code.

Each of these items is further explained next.

#### **1.1.1.1 Importance of Modular Designs**

As mentioned earlier, architectural design of flight algorithms takes place in the desktop prototyping phase and the design decisions made here impact portability and testability across all testbeds. Therefore, it becomes paramount to start the FSW design process by thinking about its architecture. Figure 1.3 provide a very illustrative metaphor of the evolution of software architectures throughout the past few decades: starting from a complete lack of architecture, moving to monolithic algorithm architectures and finally transitioning to an architecture of micro-services or atomic, individual and independent modules.

These generic software architectures apply to spacecraft FSW as well. Encapsulating GN&C functionalities in completely independent modules, instead of using monolithic algorithms, is a key aspect in terms of software safety; addition of independent FSW modules allows scaling up functionality in a more safe and systematic manner. Complexity is built through layers of atomic modules and the decoupling between these units simplifies the verification and validation process because they can be individually tested and analyzed. While the verification of the individual components by itself does not guarantee the combined algorithm is without errors, modularity

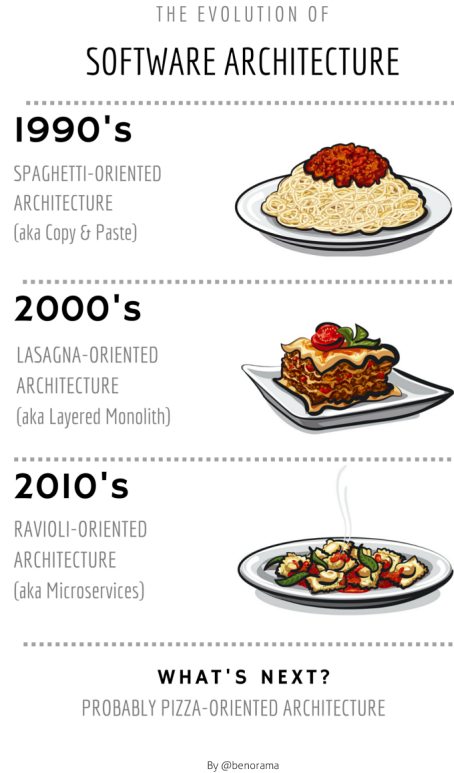


Figure 1.3: Evolution of software architectures

helps greatly on isolating errors effectively and identifying the root cause of emerging behaviors in complex GN&C sequences.

#### 1.1.1.2 Use of Scripting Languages

For the purposes of prototyping FSW in a desktop environment, the use of high-level scripting languages like Python or Matlab is extremely convenient as it enables rapid development and iteration. However, regular desktop scripting languages are not suitable for embedded flight applications requiring low memory footprint and deterministic use of resources (CPU and RAM). For this reason, if flight algorithm source code is firstly prototyped in the desktop environment using desktop scripting languages, it is then usually translated into programming languages like Fortran, C or C++ for migration into an embedded flight target.

Note that the previous statement refers specifically to “regular desktop scripting” languages,

in the sense that they are meant for general-purpose programming (like Python, Matlab, Ruby, Perl, etc.). In the field of space engineering, command sequencing languages like VML[27], PLEXIL[47] or Timeliner[8] are often also referred to as scripting languages because of their high-level nature. Sequencing languages are used onboard the spacecraft to simplify operations as well as to provide autonomy for onboard decision making. Although these languages express spacecraft commands using high-level concepts, the languages themselves are simplified to ensure that spacecraft safety can be guaranteed under all execution paths. In order to fit modestly sized flight processors, they also present reduced memory footprint. In this manuscript, the term “scripting” language is used to refer specifically to the former type, i.e. general-purpose desktop scripting language, which does not present memory constraints nor provides guarantees on real-time determinism.

For proper deterministic programs, one needs both a language that guarantees determinism on its operations (assuming the underlying machine is deterministic) as well as some rules for writing programs. Scripting languages are high-level and, by definition, they have a large “distance” from the underlying machine instructions. In contrast and for example, an operation in C generally compiles down to one or a handful of machine instructions. This makes it highly deterministic, since the machine itself is usually hard real-time. A common feature of desktop scripting languages is that they rely heavily on dynamic memory allocation and garbage collection. However, resource-limited systems cannot afford a non-deterministic call to the garbage collector. And if dynamic allocation cannot be used because of the lack of memory, it is very important to have other mechanisms of memory management, like placing data in custom addresses, as C pointers allow.

Going back to desktop prototyping, the two different approaches that are most commonly adopted in the aerospace community for this kind of preliminary development are discussed next: model-based development (MBD) and Python interface with underlying C/C++ code.

#### **1.1.1.3 Approach 1: Model-based development**

The MBD approach consists on performing architecture design and modeling of both software functions and hardware subsystems using block-diagram programming software tools like,



Figure 1.4: Model-based development: from model in the loop (MIL), to software in the loop (SIL), up into hardware in the loop (HWIL)

for example, Mathworks’s Simulink<sup>1</sup> and National Instruments LabVIEW<sup>2</sup>. Next, an automated source-code-generation software tool is used to translate the graphic design into programming source code. This step is often known as auto-coding. The MBD process is depicted in Fig. 1.4. In spite of its convenience, MBD introduces another step in the flow of flight algorithms between environments that adds on into the continuity problem: FSW validity from model-in-the loop (MIL) simulations to software-in-the-loop (SIL) simulations cannot be readily inferred without further testing[4]. Additional challenges with automatically generated code are that: 1) it can be less efficient in either size or execution than optimized hand-written code and 2) it can be very challenging to edit and debug due to lack of readability[7].

#### 1.1.1.4 Approach 2: Python interface with underlying C/C++ code

An alternative to MBD is the use of Python wrapping C/C++ source code. The Python language is recognized as an excellent scripting environment and code-development testbed that would lend itself very well to the FSW development process if the code could run as FSW. However, as a scripting language, the Python runtime is generally too slow and insufficiently well-controlled for time-critical applications like those required for aerospace FSW. Without loss in generality, runtime limitations are a tradeoff that the Python language pays in exchange for its dynamic nature and versatility. Lack of timing controllability comes in the form of processes like dynamic memory allocation and garbage collection, which take up variable amounts of time to complete. Regarding Python’s speed limitations, there are multiple reasons behind the fact: it uses a global

<sup>1</sup> <https://www.mathworks.com/products/simulink.html>

<sup>2</sup> <http://www.ni.com/en-us/shop/labview.html>

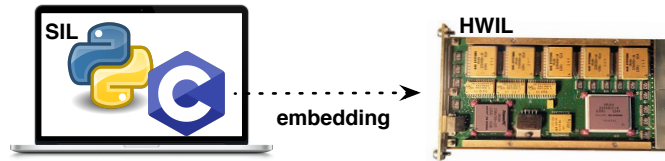


Figure 1.5: Python wrapper with underlying C/C++ code: straight from software in the loop (SIL) to hardware in the loop (HWIL)

interpreter lock that can only execute one operation at a time; it is interpreted and not compiled; and it is a dynamically typed language.

Having said that, looking at the internals of the Python language itself reveals that most built-in modules requiring speed are actually written in C/C++ and then wrapped into Python using Python-language bindings. Using this logic, it makes good sense that Python could serve as an excellent testbed for FSW development if the FSW code is written exclusively in C/C++ and then wrapped into Python for simulation, analysis, and testing. Such development proposal is depicted in Fig. 1.5. In the desktop environment, there are several ways to extend the Python language with custom C/C++ modules. CPython is the native way of creating these bindings but there are also higher level libraries like SWIG (Simplified Wrapper and Interface Generator)<sup>3</sup> that conveniently handle this extension.

The advantage of this approach is that that there is no MIL development and, from a testing perspective, the transition from MIL to SIL is skipped. While this improvement in continuity comes at the expense of developers writing the algorithm source code directly in C/C++, testing and post-processing can be done entirely in Python; hence, taking advantage of built-in libraries and other optimized mechanisms that scripting languages provide for these very specific purposes. For migration into the embedded target, the C/C++ source-code portion remains unmodified and the Python portion in Fig. 1.5 is simply removed.

None of the desktop development approaches discussed (i.e. MBD and Python wrapping C/C++ source code) provides a clear advantage for transitioning from SIL into hardware-in-the-

---

<sup>3</sup> <http://swig.org>

loop (HWIL). The migration into a flight target can actually be facilitated by the use of middleware, but performance and compatibility issues still need to be addressed separately. Even with identical source code and libraries, building an application for a new target introduces different configurations and numerical reproducibility challenges that require further verification. Reference [4] provides several guidelines to improve reproducibility and portability of numerical computations between desktop host computers and flight targets. The point being made in this manuscript is that adopting a Python-interface approach offers certain niceties shared with MBD (like scripting interface for the user) while ensuring a more homogenous environment for FSW development. The environment is more homogenous in the sense that the source code and mission-specific libraries are at least the same for both desktop host computer and embedded flight target.

The most compelling part of the Python-wrapping-C++ approach is that it takes advantage of two extremely powerful (and also extremely different) languages: Python and C/C++. Because their differences are not strictly advantageous in one direction or the other (i.e. it depends on the particular application), the idea is to exploit each language for the features that suit most the particular endeavor of FSW development. The main differences between these languages can be summarized in terms of: memory management, type declaration, language complexity and language implementation (i.e. interpreted vs. compiled). Specific advantages of Python are the following:

- (1) It presents an especially clean, straightforward syntax (which leads to faster development and less mental overhead)
- (2) It has a very large standard library
- (3) Because Python is generally not compiled, Python interpreters are great for rapid testing and exploration.

In turn, some advantages of C++ over Python are the following:

- (1) The runtime performance is better and more predictable (since it does not have garbage collection and encourages use of raw pointers to manage and access memory)

- (2) It can target just about every known platform including embedded systems (since it is a low-level language)

With all this in mind, the generalized interest in Python as a wrapper for C/C++ flight code is not surprising and there are many state-of-the-art FSW tools that are adopting this approach. For instance, the Basilisk astrodynamics framework is a desktop FSW testbed that seeks to capitalize on the potential of using Python as a wrapper for flight algorithm source code that is actually written in C/C++[39, 1, 18]. In the context of deep-space navigation design and analysis, there is MONTE[44], which has been developed by JPL and consists of low-level C++ astrodynamics libraries that are presented to the end user as an importable Python-language module. A different tool developed by JPL which also leverages the use of Python as a C++ wrapper is Dshell[10]: a physical simulator that supports both robotic and spacecraft simulations, software and hardware-in-the-loop testing as well as mission telemetry visualization.

### 1.1.2 Embedded Environment

Time-critical applications like those of FSW usually demand the use of onboard processors with drastically fewer resources available than a typical desktop computer. Therefore, FSW systems are said to be constrained or embedded. Embedded environments are in essence electronic systems that are managed by a microprocessor (like a hardware flight processor) or a micro-controller that operates the whole system with precise timing. Embedded flight processor environments are defined by the selection of two items: the microprocessor board and the real-time operating system (RTOS).

When programmed appropriately, a real-time system can guarantee that tasks execute consistently in a specified time constraint. Determinism is, precisely, the characteristic that describes how consistently a system executes tasks within a time constraint. A perfectly deterministic system would experience no variation in timing for tasks. Typically, flight systems demand determinism in both operations and CPU cycles. In addition, they present reduced memory availability (RAM/ROM).

For the purposes of embedded testing, an alternative to using a hardware flight processor

is to emulate it on a virtual machine. Examples of processor board emulators are the open-source QEMU<sup>4</sup> or the closed-source EMU[25] created by the ESOC center of the European Space Agency. As mentioned earlier, the advantage of using an emulation is that it provides a pure-software substitution for an expensive and limited piece of hardware which, otherwise, cannot be used simultaneously by different mission groups. Regardless of the flight processor board being physical or emulated, it still represents an embedded environment. Embedded flight processors lag state-of-the-art processors (like those in a desktop computer) by about 10 years due to flight heritage and radiation-hardening requirements[29]. Radiation hardening of processors is important in order to ensure their un-interrupted operation over long durations in the harsh environment of space. Figure 1.6 shows several radiation-hardened processors commonly used for space exploration (RAD750, ColdFire, LEON, etc.); all of them being very expensive and presenting similar limitations in performance.

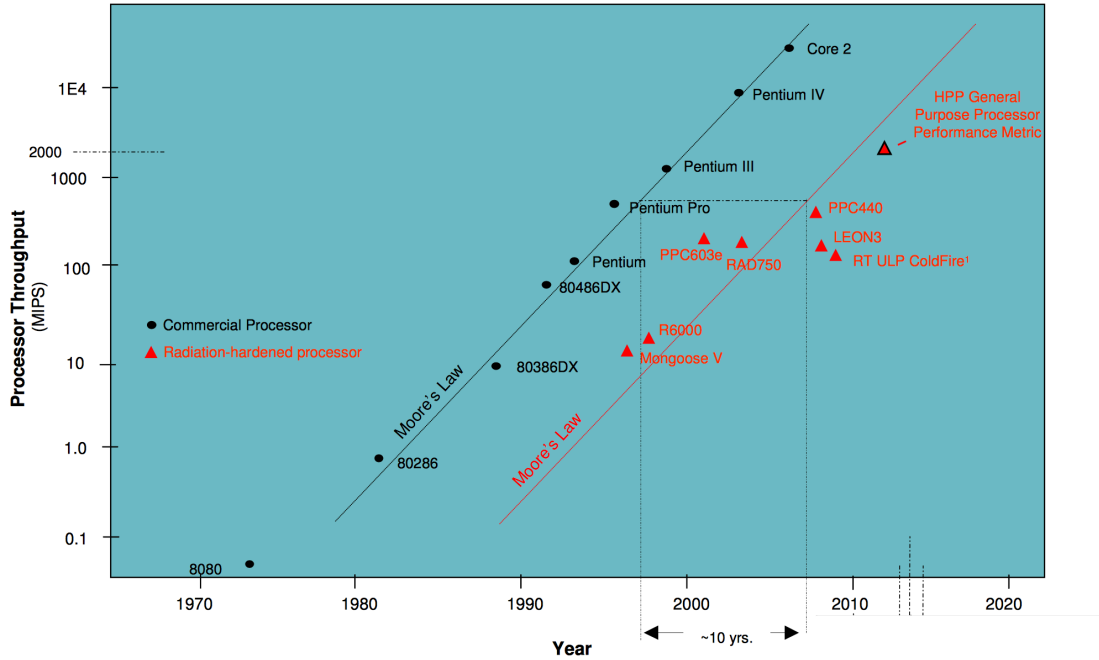


Figure 1.6: Radiation-hardened microprocessors. Image extracted from [29]

<sup>4</sup> <http://qemu.org>



### 1.1.3 Middleware Layers

Because a regular desktop computer environment and a flight processor environment operate differently, migrating the flight application from one to another demands a significant migration effort. Furthermore, this effort is intrinsically linked to the specific processor board and RTOS chosen, tending to be mission-specific. An alternative target for the flight algorithms is a middleware layer. Middleware can be regarded as an abstraction layer or “glue code” that ensures portability of the flight application among different processors and RTOS. An example of middleware is the core Flight System (cFS)[36, 21], which is an open-source product provided by NASA Goddard Spaceflight Center. While targeting middleware can be worthwhile in the long run to ensure portability of the flight application, small missions do not tend to follow this approach given the complexity and steep learning curve of the work entailed[7]. However, if a user-friendly, easily-deployable middleware layer existed, the number of missions embracing reusability through middleware would most likely increase.

Recently, a lean (i.e. memory lightweight), efficient (i.e. with fast execution) and highly portable implementation of the Python 3 programming language has been developed. This new implementation is named MicroPython and it is very compelling for use in embedded FSW systems as it includes a small subset of the Python standard library and it is optimized to run on microcontrollers and in constrained environments. The difference between MicroPython and conventional programming languages is that it provides many advanced features (characteristic of scripting languages) while having little memory footprint and being extremely compact (characteristic of embedded programming languages). Regarding MicroPython’s portability, it currently supports about 15 different ports available on GitHub.<sup>5</sup> Some of these ports include: Unix, Windows, STM32, QEMU-ARM, bare-ARM or EST32. This promising language is already being investigated by ESA for Onboard Control Procedures (OBCPs) in spacecraft payloads [26, 30]. OBCPs are flight procedures that provide a flexible way to operate the spacecraft by either extending the onboard software functionality or modifying the behavior of onboard applications. Since OBCPs operate

---

<sup>5</sup> <https://github.com/micropython/micropython/tree/master/ports>

separately from the main onboard FSW, this functionality is different from the one proposed in this manuscript: the use of MicroPython is as a middleware layer for porting the main onboard FSW into multiple flight targets.

## **1.2 Literature Review: State-of-the-art Tools for FSW Development**

This section provides an overview of state-of-the-art software tools that are commonly used in the aerospace sector for FSW development. Firstly, several frameworks that are suitable for desktop algorithm design are reviewed. During this initial prototyping phase, it is critical to use spacecraft astrodynamics models that are realistic enough to provide valuable information about the flight algorithm performance through closed-loop tests. Secondly, the discussion is extended to common combinations of software tools that can support cross-environment testing (i.e. from desktop simulations to embedded ones).

### **1.2.1 Modular Desktop Development Tools**

Desktop analysis through closed-loop dynamic simulations has become an essential part of flight algorithm development, especially when it comes to the attitude determination and control system (ADCS). There are numerous spacecraft astrodynamics frameworks in the market and each of them presents its own strengths and weaknesses[1]. In general, though, these frameworks have limited capability to represent a complete, physically realistic, dynamical representation of a spacecraft that can be used for accurate ADCS analysis.

Frameworks that are not open source can easily become prohibitively expensive for low-budget missions or student development. In turn, open-source products are –more often than not– poorly maintained and not user-friendly enough to learn them in the timeline of a particular mission. In addition to features like open sourcing and user friendliness, it is a plus for software frameworks to be cross platform (i.e. suitable for Mac, Linux and Windows). For the specific purposes of ADCS design and analysis, it is also critical that the simulation environment can support both real-time HWIL testing and faster-than-real-time pure software testing with built-in repeatable Monte Carlo

options.

Mathworks’s Simulink and National Instruments LabVIEW, which were introduced earlier, provide excellent platforms for building quick models of control systems for faster-than-real-time and HWIL simulations. Downsides of these softwares are:

- (1) Lack of a built-in visualization tool that allows fast and intuitive analysis.
- (2) Cost of the tools, since they are not open source.
- (3) Necessity of a fairly experienced user.

Softwares such as AGI’s Systems Toolkit (STK),<sup>6</sup> a.i. solutions’ Freeflyer,<sup>7</sup> and Applied Defense Solutions’ Spacecraft Design Tool<sup>8</sup> all provide beautiful visualizations and highly efficient simulation tools but they are oriented towards orbit and mission design rather than ADCS. STK does provide a six-degree-of-freedom option for propagation of attitude states but relies on external torque generation and has no way of implementing accurate sensor or actuator models during its dynamic simulation. Furthermore, none of these three software packages are conditioned for HWIL testing. Softwares such as NASA’s Trick<sup>9</sup> and CS’s Orekit/Rugged<sup>10</sup> are both specifically made for spacecraft attitude dynamics simulation, but they still fall short in providing a fully-coupled dynamic representation of a spacecraft with complex actuators such as reaction wheels. Various other open-source softwares exist for attitude dynamics simulation but are either poorly documented, unmaintained, or lack critical features for all-in-one ADCS development[45].

After providing a brief review of state-of-the-art spacecraft simulation frameworks, the dynamic’s simulation side of the Basilisk testbed is introduced. Basilisk’s modeling of the spacecraft’s physics is particularly relevant in the context of this thesis because it is used to design and test ADCS algorithms in closed-loop dynamics. Basilisk provides a fully coupled dynamical representation of spacecraft attitude states, allowing for complex dynamics behaviors such as imbalanced

---

<sup>6</sup> <http://www.agi.com/products/stk/>

<sup>7</sup> <https://ai-solutions.com/freeflyer/>

<sup>8</sup> <http://www.applieddefense.com/products-and-services/sdt/>

<sup>9</sup> <https://github.com/nasa/trick>

<sup>10</sup> <https://www.orekit.org/>

reaction wheels, solar panel hinging[3] and even fuel slosh[2]. Basilisk is an open-source and cross-platform product, it is extensively documented and it offers an add-on visualization tool. It runs natively faster than real time and it includes built-in Monte Carlo capabilities. As shown later in this thesis, soft real-time numerical simulations with HWIL are also supported.

The next subsection discusses how these desktop tools can be combined with other frameworks for cross-environment testing.

### 1.2.2 Cross-Environment Development Tool Suites

Currently, several software tools –or rather combinations of them– exist that support the entire FSW development cycle (i.e. from desktop prototyping up to embedded integration and testing). With some generalization, FSW development architectures are characterized by the degree to which system components are coupled. The coupling between simulation components is manifested by the simulation structure, where the overall system may be: integrated as a single system of required components; integrated as a modular system with optional components; or developed as a group of cooperative yet stand-alone components. Some of the most comprehensive tool suites include: MAX Flight Software<sup>11</sup> together with the On-Board Dynamic Simulation System (ODySSy)[23], both provided by Advanced Solutions Inc (ASI); Matlab’s Simulink<sup>12</sup> and Matlab’s Embedded Coder<sup>13</sup> together with NASA’s Trick engine[38]; JPL’s Dshell system[10] together with the NASA Operational Simulator (NOS)<sup>14</sup>; or the Basilisk desktop testbed<sup>15</sup> together with the Black Lion communication architecture[13].

These state-of-the-art options differ on the specific tools they use and on the degree of coupling between them. ASI’s proposal is an example of a tightly coupled tool suite, where MAX Flight Software is used for FSW design and ODySSy is used for closed-loop testing. While this option provides testability in both desktop and embedded environments, it does so by requiring these tools

<sup>11</sup> <http://www.go-asi.com/solutions/max-flight-software>

<sup>12</sup> <https://www.mathworks.com/products/simulink.html>

<sup>13</sup> <https://www.mathworks.com/products/embedded-coder.html>

<sup>14</sup> [https://www.nasa.gov/centers/ivv/jstar/jstar\\_simulation.html](https://www.nasa.gov/centers/ivv/jstar/jstar_simulation.html)

<sup>15</sup> <https://hanspeterschaub.info/bskMain.html>

specifically and there are minimal options to substitute one component with another which was not intended to operate with the ASI's system. A more flexible combination is the one encompassing the use of Mathwork's Simulink for FSW design, Mathwork's Embedded Coder for FSW migration and NASA's Trick engine for closed-loop testing. This specific tool suite is currently being applied to the Orion mission[37]. The main caveat of either of these proposals (i.e. ASI's and Orion's) is that they rely on an MBD approach for desktop development, which does not respect the principle of migration transparency. Within ASI's proposal, auto-coding is performed by MAX Flight Software; within the Orion mission proposal, auto-coding is performed by Mathwork's Embedded Coder. In addition, none of these FSW development proposals addresses the integration of code into flight targets beyond auto-coding. Note that auto-coding simply produces source code in the target programming language, but this code still needs to be integrated into either a middleware layer or a specific board and RTOS. Further, these tool suites do not natively offer a distributed testing environment: both ODySSy and Trick (i.e. the spacecraft physical simulations) are meant to migrate with FSW when the latter is embedded. Given the computing limitations of traditional flight processors, running an embedded spacecraft simulation raises questions about the fidelity of its physical models.

A software suite which demonstrates increasing modularity in its architecture is JPL's proposal through Dshell and NOS. While the Dshell system is a physical simulator, NOS constitutes the communication framework to run distributed software simulations of independent mission components[50, 28], with Dshell being one of its potential components. Although the combination of Dshell and NOS conform a very compelling option for distributed closed-loop testing, which would work seamlessly whether FSW resides in the desktop or in the embedded environment, these software tools are not open source. Further, this tool suite does not provide a FSW development environment on itself, which is let to be chosen by the specific mission.

In overall, the state-of-the-art tool suites that have been reviewed fall short in one or more of the following categories: completeness of the tool suite as a FSW testbed, transparency of the flight algorithm flow between testbeds, architectural flexibility to include external models for testing,

support of distributed simulations and open sourcing of the tools to the community.

The niche identified in existing end-to-end FSW development tool suites has motivated the development of the Basilisk software testbed and the Black Lion communication architecture. This thesis has contributed to both. As a quick recapitulation, Basilisk is an open-source, cross-platform, desktop testbed for designing flight algorithms and testing them in closed-loop dynamics simulations. It leverages Python’s ease of use as a testbed for FSW development provided that the spacecraft models and the flight algorithm code are written exclusively in C/C++ and, then, wrapped into Python for custom setup, desktop execution and post-processing. In turn, Black Lion is a purely software-based communication architecture aimed at integrated testing of independent spacecraft mission models –with Basilisk being one of its potential components. Black Lion is architected to be reconfigurable and scalable, allowing for any number of heterogenous software models, across one or multiple computing platforms, to be integrated into a single spacecraft simulation run. Such architecture enables, for instance, seamless integration of legacy software models that were never designed to work together. Both Basilisk and Black Lion are currently being implemented by the Autonomous Vehicle Systems (AVS) laboratory at the University of Colorado Boulder and the Laboratory for Atmospheric and Space Physics (LASP) in support of an ongoing interplanetary spacecraft mission. Yet both tools are being built under the principles of flexibility and reusability and, as demonstrated in this thesis, their application extends far beyond this particular mission.

### 1.3 Outline

The different phases of the FSW development process that are covered in this thesis, as well as their scope, are outlined next:

- (1) **Desktop flight algorithm development: modular attitude guidance reference generation for distinct mission profiles.** The relevance of architecting flight algorithms through modular designs and shared coding standards is showcased through a guidance

application: onboard attitude reference generation in a modular fashion. The proposed work involves: 1) Breaking up the mathematical transformations required to build up complex guidance motions into atomic functions; 2) developing a suite of atomic guidance software modules; and 3) integrating these modules in Basilisk for simulated closed-loop testing. This work has been published into the AIAA Journal of Aerospace Information Systems[18].

- (2) **Flight algorithm migration into commercial flight targets.** This work describes the migration of flight algorithms from the Basilisk desktop prototyping environment into a commercial processor that has been considered for low-cost space applications: the Raspberry Pi. For the first time, two Basilisk processes running on different platforms are shown to close the loop in a distributed fashion. These contents were presented in the 26th International Symposium on Space Flight Dynamics[17].
- (3) **Flight algorithm migration into the core Flight System.** The technical steps required to migrate Basilisk-developed flight algorithms into a pure-C cFS application are described. The resulting cFS-FSW application can then be ported across different embedded systems (i.e. processor boards and RTOS). The novelty of the work lies in the migration mechanism: the transition out of the Basilisk desktop environment is achieved by automatically generating the integration code required to integrate the unmodified C/C++ flight algorithm code into the cFS embeddable environment. The C integration code, which is minimal and completely human-readable, is generated through Python’s introspection capabilities while the actual source code remains unchanged. This work was presented in the 24th DASIA conference[15] and it is currently undergoing the second review for publication into the AIAA Journal of Aerospace Information Systems[19].
- (4) **Emulated flat-sat testing of cFS-FSW through distributed communication.** The cFS-FSW application is tested on an emulated flat-sat conformed by heterogeneous mission models that were never designed to work together. The Black Lion communication archi-

ture, which has been built to enable integration of all these components into a single, distributed simulation run, is described. In addition, FPGA registers and avionic component models are emulated to allow reading and writing of the cFS-FSW states from and to the external world. The unprecedented level of fidelity inherent to these emulations makes it possible for FSW to believe it is running on a hardware processor board, although it is actually running on its emulated counterpart. Numerical simulations show various tests performed on the emulated flat-sat as well as other applications of the Black Lion communication architecture. The work associated to Black Lion was presented in the 2018 John L. Junkins Dynamical Systems Symposium[14] and it has been submitted for publication into the AIAA Journal of Aerospace Information Systems[13]

- (5) **Flight algorithm migration into MicroPython.** The technical steps involved in migrating Basilisk-developed flight algorithms into the modern MicroPython are described. In addition, tools aimed at automatizing this migration process are implemented and explained. The use-case of MicroPython is novel in the sense that, to the author’s knowledge, it has not yet been considered as a potential middleware layer to ensure portability of the onboard FSW application among different RTOS and flight processor boards. The first proof of concept of the Basilisk-MicroPython flight system is showcased in the form of a distributed numerical simulation. This work has been presented and submitted for publication together with cFS-related work mentioned earlier[15, 19].

- (6) **Basilisk-MicroPython for Embedded Systems.** The suitability of the Basilisk - MicroPython system for constrained environments (i.e. with limited resources and deterministic requirements) is analyzed. This work involves targeting a Basilisk-MicroPython FSW application into 32-bit processors (like the family of LEON boards) as well as profiling and optimizing the memory and CPU usage of the FSW application on Unix (with the aim of targeting a constrained Unix environment that could be ready for flight). It is envisioned to present this work in the 25th DASIA conference.



## Chapter 2

### Desktop Flight Algorithm Development: Modular ADCS

Architectural design of flight algorithms takes place in the desktop prototyping phase and it is enabled by flexible development environments which, ideally, allow complete customization of algorithm structure and contents while leveraging the recurrent yet time-consuming tasks of deploying, building and compiling the executable to test. To this end, the Basilisk environment is presented as a flexible, desktop FSW testbed that incarnates the desktop development proposal of using Python as a user-interface language for prototyping and testing flight algorithm code that is actually written in C/C++. Upon Basilisk, a modular scheme for generating attitude reference motions is architected and implemented. The layered approach of building an attitude reference is interesting because it promotes code reusability in a topic area that tends to be highly mission specific: the generation of rotational guidance profiles. Once proved, the validity of the modular guidance approach still holds out of the Basilisk environment.

#### 2.1 The Basilisk Testbed

Basilisk is an open-source, cross-platform, desktop testbed for designing flight algorithms and testing them in closed-loop dynamics simulations. Basilisk is architected in a modular and highly reconfigurable fashion using C++ modules that perform spacecraft physical simulation tasks and C modules that perform mission-specific GN&C tasks. The SWIG library is used to wrap the C/C++ modules and make them available at the Python layer for **setup, desktop execution** and **post-processing**. Some of the advantages of using Python as the user-facing interface

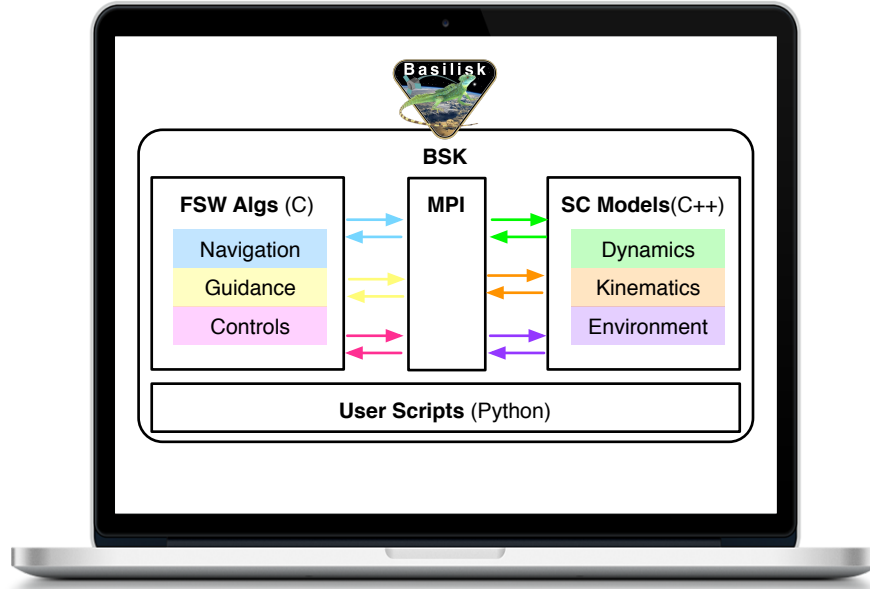


Figure 2.1: Basilisk (BSK) desktop environment

are: ease of data analysis (which is comfortably leveraged through built-in libraries like NumPy, Matplotlib and Pandas among other), capability of automated regression tests (via PyTest) and rapid Monte-Carlo handling.

Figure 2.1 illustrates the nominal setup –but not necessary required– of a Basilisk desktop simulation. This setup is conformed by two main processes: a high-fidelity simulation of the physical spacecraft and a suite of GN&C flight algorithms. During a simulation run, the C and C++ modules from the different processes communicate with each other through a custom Message Passing Interface (labeled as “MPI” in Fig. 2.1). For the sake of clearness, in this manuscript the acronym MPI is used to refer to the Basilisk custom message passing interface, which is completely unrelated to the well-known MPI used in parallel computing.<sup>1</sup> Basilisk’s MPI is written in C/C++ and it is based on a publish-subscribe pattern. The beauty of using a message interface is that it delineates a very clean separation between the different processes. As shown in later section of this manuscript, the clear separation of processes facilitates, in turn, the migration of the FSW application into a different processor.

<sup>1</sup> <https://mpi.org>

The modularity of the Basilisk system implies that each process is decomposed into a series of simpler steps and exchangeable components. The cascading of modules is set at the Python level, allowing different levels of simulation fidelity and flight software sophistication. Such lego-like architecture allows, for example, the modular guidance application that is presented in the next section.

## 2.2 Rotational Reference Motions for Distinct Guidance Profiles

Mission GN&C flight algorithms undergo a stringent review and validation process prior to flight, which can be both costly and time consuming. Developing flight algorithms through modular designs (i.e. breaking up functionality into a series of simple independent pieces or modules instead of having one large software piece that performs a complex function) has shown to improve efficiency in terms of implementation and testing. Generally, talking about modularity of GN&C would mean that there are separate modules used for sensory input, parameter identification, reference trajectory selection/generation, position error determination, control, and output. This kind of split is seen, for example, in the GN&C sequences presented in [24, 43]. Note that “selecting” a reference implies that the motion is predefined by ground and uploaded onboard. In turn, “generating” a reference implies computing the desired motion autonomously onboard and this is, precisely, the concept of operations addressed in this work. The novelty lays on bringing modularity one step further by dividing the reference generation into multiple, exchangeable sub-components. The advantage of fractionating the reference generation is that, for a given set of core modules, a wide variety of guidance behaviors can be achieved through combination and distinct initialization. Figure 2.2 illustrates that common spacecraft attitude guidance profiles tend to share core functionalities and this is the fact to be exploited in this work.

Without loss in generality, this work investigates a methodology to modularize any attitude reference motion into three atomic parts: a base pointing reference, a dynamic reference that is relative to the base, and an attitude offset. The final, desired reference motion is a result of cascading these three attitude reference parts, as depicted in Fig. 2.3. Different combinations of

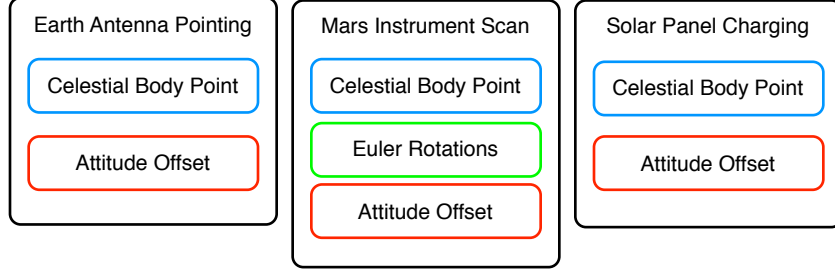


Figure 2.2: Break-down of sample mission profiles

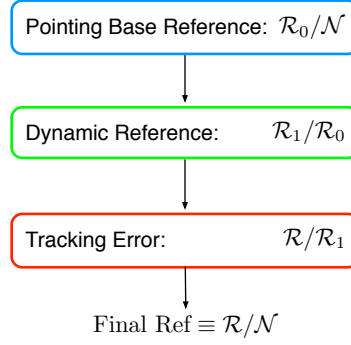


Figure 2.3: Attitude guidance generation

pointing and dynamic references yield guidance scenarios of distinct complexity. All proposed guidance schemes are applicable to any type of Keplerian orbit, including elliptic circumnavigation and hyperbolic fly-by trajectories.

### 2.2.1 Problem Statement

For a given spacecraft, the goal of the onboard GN&C flight software is to estimate the current state of the spacecraft body  $\mathcal{B}$  (navigation task), generate a reference state  $\mathcal{R}$  that can be time-varying or not (guidance task), derive the attitude tracking error between the current state  $\mathcal{B}$  and the desired one  $\mathcal{R}$  (also guidance task), and apply the required control torque to align  $\mathcal{B}$  with  $\mathcal{R}$  (control task). Both the spacecraft-body state  $\mathcal{B}$  and the reference state  $\mathcal{R}$  computed onboard are expressed with respect to an inertial frame  $\mathcal{N}$ . Note that  $\mathcal{N}$  is kept as a generic inertial frame to be chosen (it could be J2000, Earth fixed-frame, Mars centered frame, etc.) since all that matters is consistency throughout the entire flight algorithm suite. The computed reference state  $\mathcal{R}$  is, in

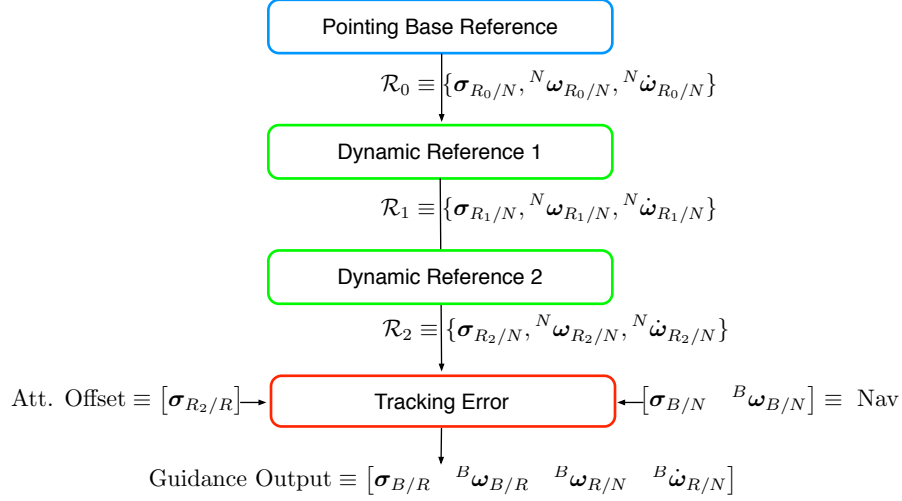


Figure 2.4: Inputs and outputs of a compounded attitude reference chain

this work, composed of three parameters:

$$\mathcal{R} = [\sigma_{R/N}, {}^{\mathcal{N}}\omega_{R/N}, {}^{\mathcal{N}}\dot{\omega}_{R/N}] \quad (2.1)$$

These parameters are: an inertial attitude measure, denoted through the Modified Rodrigues Parameters (MRP) set  $\sigma_{R/N}$  [42], an inertial angular rate vector  ${}^{\mathcal{N}}\omega_{R/N}$  expressed in inertial frame  $\mathcal{N}$  components, and an inertial angular acceleration vector  ${}^{\mathcal{N}}\dot{\omega}_{R/N}$  also in  $\mathcal{N}$ -frame components. The left-superscript denotes the frame with respect to which the vector components are taken. The use of MRPs in this development is simply a convenient choice but, as a matter of fact, any other attitude description like quaternions or Direction Cosine Matrices (DCM) could be used instead. The use of MRPs is attractive because they are a non-redundant set (composed by 3 components only) and their singularities can be avoided by switching to the so-called shadow set[42].

Figure 2.4 illustrates the generation of a compounded final reference  $\mathcal{R}$ . The final reference is computed through addition of a base pointing reference, two dynamic reference motions (each one relative to the former one) and an attitude offset. The tracking error module is always the last component in the guidance chain. This module reads the output of the latest reference module and adds an attitude offset if applicable. An attitude offset is applied when the generated attitude reference is meant for a spacecraft fixed frame that is not the main one. The

output of the tracking error module is the output of the entire attitude guidance block and it is used to feed the control block next. Note that any number of dynamic references modules could be sequentially chained to create increasingly complex rotational patterns. The key to this scalability is that, as shown in Fig. 2.4, all the reference generation modules (both pointing base and dynamic) output the same message structure defined in Eq. (2.1).

## 2.2.2 Software Modules and Mathematical Development

For each of the aforementioned core functionalities (base pointing reference, dynamic reference, and attitude offset) several software modules are implemented, which can be plugged-and-played as lego-like pieces in order to achieve very different rotational patterns. The mathematical derivation for each of the implemented modules is presented next.

### 2.2.2.1 Base Reference Modules

The first guidance stage consists of modules that generate a base pointing reference  $\mathcal{R}_0$  that can be either inertial or non-inertial. The common feature of the base modules is that the generated reference does not depend on any prior reference frame. The base reference modules developed and later integrated to the Basilisk software framework are: inertial pointing, Hill orbit pointing, velocity orbit pointing, and constrained two-body pointing. Both the inertial and orbit frame references are widely used and well documented, but the novelty here lies on the scheme upon which they are architected: through the modular stack and interface definition, base modules can be used in stand-alone mode or as the base of complex dynamic behaviors. For this reason, their development is presented in Appendix A. In turn, the constrained two-body pointing module is a novel kinematic solution to a constrained attitude pointing requirement. The mathematical transformations associated to the derivation of this module are presented next.

Within the constrained two-body pointing module, a base reference frame  $\mathcal{R}_0 : \{\hat{\mathbf{r}}_1, \hat{\mathbf{r}}_2, \hat{\mathbf{r}}_3\}$  is generated that tracks the center of a primary celestial target (e.g. pointing the communication antenna at the Earth) and tries to align the second reference axis towards a second celestial body

as best as possible (e.g. pointing a solar panel normal axis to the sun). Two attitude conditions in a three-dimensional space compose an overdetermined problem. Hence, a TRIAD-like approach is used such that the main constraint is always prioritized over the secondary one.[42]

Figure 2.5 depicts the desired reference frame  $\mathcal{R}_0 : \{\hat{\mathbf{r}}_1, \hat{\mathbf{r}}_1, \hat{\mathbf{r}}_2\}$  and the inertial frame  $\mathcal{N} : \{\hat{\mathbf{n}}_1, \hat{\mathbf{n}}_1, \hat{\mathbf{n}}_2\}$ . The  $\mathcal{R}$  frame has its origin in the spacecraft body  $\mathcal{B}$ . The points  $\mathcal{P}_1$  and  $\mathcal{P}_2$  are the primary and secondary celestial targets respectively. The initial states known by the module are the position vector  $\mathbf{R}_{i/N}$ , velocity vector  $\dot{\mathbf{R}}_{i/N}$ , and acceleration vector  $\ddot{\mathbf{R}}_{i/N}$  of the spacecraft ( $i = \mathcal{B}$ ) and of the celestial bodies ( $i = \mathcal{P}_1, \mathcal{P}_2$ ) with respect to the inertial frame.

The normal vector  $\mathbf{R}_n$  is perpendicular to the plane defined by the two celestial targets and the spacecraft location, and it is expressed as

$$\mathbf{R}_n = \mathbf{R}_{P_1/B} \times \mathbf{R}_{P_2/B} \quad (2.2)$$

The desired base reference frame  $\mathcal{R}$  is computed such that the first unit base vector  $\hat{\mathbf{r}}_1$  points to the primary target. The third base vector  $\hat{\mathbf{r}}_3$  is aligned with  $\mathbf{R}_n$  while the last base vector  $\hat{\mathbf{r}}_2$  completes the right-handed triplet.

$$\hat{\mathbf{r}}_1 = \frac{\mathbf{R}_{P_1/B}}{|\mathbf{R}_{P_1/B}|} \quad (2.3a)$$

$$\hat{\mathbf{r}}_3 = \frac{\mathbf{R}_n}{|\mathbf{R}_n|} \quad (2.3b)$$

$$\hat{\mathbf{r}}_2 = \hat{\mathbf{r}}_3 \times \hat{\mathbf{r}}_1 \quad (2.3c)$$

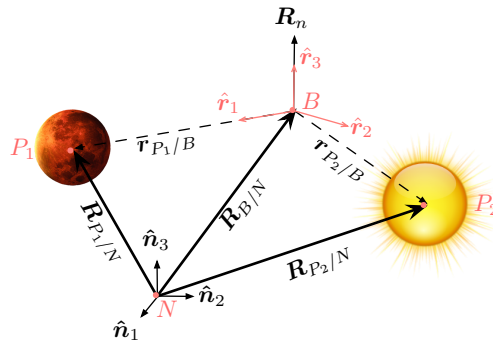


Figure 2.5: Constrained celestial two-body pointing scenario

The celestial object locations relative to the spacecraft are found through

$$\mathbf{R}_{P_1/B} = \mathbf{R}_{P_1/N} - \mathbf{R}_{B/N} \quad (2.4a)$$

$$\mathbf{R}_{P_2/B} = \mathbf{R}_{P_2/N} - \mathbf{R}_{B/N} \quad (2.4b)$$

This setup aligns  $\hat{\mathbf{r}}_2$  as closely as possible with  $\mathbf{R}_{P_2/B}$ . However, for general configurations it is not possible to align these vectors perfectly while meeting the primary constraint at the same time.

The DCM that maps from the inertial frame  $\mathcal{N}$  to the desired reference frame  $\mathcal{R}$  is given by:

$$[RN] = \begin{bmatrix} \mathcal{N}\hat{\mathbf{r}}_1^T \\ \mathcal{N}\hat{\mathbf{r}}_2^T \\ \mathcal{N}\hat{\mathbf{r}}_3^T \end{bmatrix} \quad (2.5)$$

The desired MRP attitude set  $\boldsymbol{\sigma}_{R_0/N}$  can then be directly obtained from  $[R_0N]$ . The angular velocity  $\boldsymbol{\omega}_{R_0/N}$  and acceleration  $\dot{\boldsymbol{\omega}}_{R_0/N}$  still have to be computed. In order to do so, the time derivatives of the reference base vectors are needed. The following expressions are found for the first inertial time derivatives:

$$\dot{\hat{\mathbf{r}}}_1 = ([I_{3 \times 3}] - \hat{\mathbf{r}}_1 \hat{\mathbf{r}}_1^T) \frac{\dot{\mathbf{R}}_{P_1/B}}{|\mathbf{R}_{P_1/B}|} \quad (2.6a)$$

$$\dot{\hat{\mathbf{r}}}_3 = ([I_{3 \times 3}] - \hat{\mathbf{r}}_3 \hat{\mathbf{r}}_3^T) \frac{\dot{\mathbf{R}}_n}{|\mathbf{R}_n|} \quad (2.6b)$$

$$\dot{\hat{\mathbf{r}}}_2 = \dot{\hat{\mathbf{r}}}_3 \times \mathbf{r}_1 + \mathbf{r}_n \times \dot{\hat{\mathbf{r}}}_3 \quad (2.6c)$$

where the first and second inertial time derivatives of  $\mathbf{R}_n$  are

$$\dot{\mathbf{R}}_n = \dot{\mathbf{R}}_{P_1/B} \times \mathbf{R}_{P_2/B} + \mathbf{R}_{P_1/B} \times \dot{\mathbf{R}}_{P_2/B} \quad (2.7)$$

$$\ddot{\mathbf{R}}_n = \ddot{\mathbf{R}}_{P_1/B} \times \mathbf{R}_{P_2/B} + 2\dot{\mathbf{R}}_{P_1/B} \times \dot{\mathbf{R}}_{P_2/B} + \mathbf{R}_{P_1/B} \times \ddot{\mathbf{R}}_{P_2/B} \quad (2.8)$$

Differentiating the unit vectors in Eqs. (2.6) yields:

$$\ddot{\hat{\mathbf{r}}}_1 = \frac{1}{|\mathbf{R}_{P_1/B}|} (([I_{3 \times 3}] - \hat{\mathbf{r}}_1 \hat{\mathbf{r}}_1^T) \ddot{\mathbf{R}}_{P_1/B} - 2\dot{\hat{\mathbf{r}}}_1(\hat{\mathbf{r}}_1 \cdot \dot{\mathbf{R}}_{P_1/B}) - \hat{\mathbf{r}}_1(\dot{\hat{\mathbf{r}}}_1 \cdot \dot{\mathbf{R}}_{P_1/B})) \quad (2.9a)$$

$$\ddot{\hat{\mathbf{r}}}_3 = \frac{1}{|\mathbf{R}_n|} (([I_{3 \times 3}] - \hat{\mathbf{r}}_3 \hat{\mathbf{r}}_3^T) \ddot{\mathbf{R}}_n - 2\dot{\hat{\mathbf{r}}}_3(\hat{\mathbf{r}}_3 \cdot \dot{\mathbf{R}}_n) - \hat{\mathbf{r}}_3(\dot{\hat{\mathbf{r}}}_3 \cdot \dot{\mathbf{R}}_n)) \quad (2.9b)$$

$$\ddot{\hat{\mathbf{r}}}_2 = \ddot{\hat{\mathbf{r}}}_3 \times \mathbf{r}_1 + \mathbf{r}_n \times \ddot{\hat{\mathbf{r}}}_3 + 2\dot{\hat{\mathbf{r}}}_3 \cdot \dot{\hat{\mathbf{r}}}_1 \quad (2.9c)$$



The reference angular rate is expressed in reference frame components as:

$${}^{R_0}\boldsymbol{\omega}_{R_0/N} = \begin{bmatrix} \boldsymbol{\omega}_{R_0/N} \cdot \hat{\mathbf{r}}_1 \\ \boldsymbol{\omega}_{R_0/N} \cdot \hat{\mathbf{r}}_2 \\ \boldsymbol{\omega}_{R_0/N} \cdot \hat{\mathbf{r}}_3 \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{r}}_3 \cdot \dot{\hat{\mathbf{r}}}_2 \\ \hat{\mathbf{r}}_1 \cdot \dot{\hat{\mathbf{r}}}_3 \\ \hat{\mathbf{r}}_2 \cdot \dot{\hat{\mathbf{r}}}_1 \end{bmatrix} \quad (2.10)$$

Taking the inertial derivative of Eq. (2.10) yields

$${}^{R_0}\dot{\boldsymbol{\omega}}_{R_0/N} = \begin{bmatrix} \dot{\boldsymbol{\omega}}_{R_0/N} \cdot \hat{\mathbf{r}}_1 \\ \dot{\boldsymbol{\omega}}_{R_0/N} \cdot \hat{\mathbf{r}}_2 \\ \dot{\boldsymbol{\omega}}_{R_0/N} \cdot \hat{\mathbf{r}}_3 \end{bmatrix} = \begin{bmatrix} \dot{\hat{\mathbf{r}}}_3 \cdot \dot{\hat{\mathbf{r}}}_2 + \hat{\mathbf{r}}_3 \cdot \ddot{\hat{\mathbf{r}}}_2 - \boldsymbol{\omega}_{R/N} \cdot \dot{\hat{\mathbf{r}}}_1 \\ \dot{\hat{\mathbf{r}}}_1 \cdot \dot{\hat{\mathbf{r}}}_3 + \hat{\mathbf{r}}_1 \cdot \ddot{\hat{\mathbf{r}}}_3 - \boldsymbol{\omega}_{R/N} \cdot \dot{\hat{\mathbf{r}}}_2 \\ \dot{\hat{\mathbf{r}}}_2 \cdot \dot{\hat{\mathbf{r}}}_1 + \hat{\mathbf{r}}_2 \cdot \ddot{\hat{\mathbf{r}}}_1 - \boldsymbol{\omega}_{R/N} \cdot \dot{\hat{\mathbf{r}}}_3 \end{bmatrix} \quad (2.11)$$

Finally the angular rates and acceleration in equations Eq. (2.10) and Eq. (2.11) are mapped to the inertial frame  $\mathcal{N}$

$$\mathcal{N}\boldsymbol{\omega}_{R_0/N} = [R_0N]^T {}^{R_0}\boldsymbol{\omega}_{R_0/N} \quad (2.12a)$$

$$\mathcal{N}\dot{\boldsymbol{\omega}}_{R_0/N} = [R_0N]^T {}^{R_0}\dot{\boldsymbol{\omega}}_{R_0/N} \quad (2.12b)$$

All the variables conforming the output structure of the constrained two-body pointing module have now been derived:  $\mathcal{R}_0 = \{\boldsymbol{\sigma}_{R_0/N}, \mathcal{N}\boldsymbol{\omega}_{R_0/N}, \mathcal{N}\dot{\boldsymbol{\omega}}_{R_0/N}\}$ .

### 2.2.2.2 Dynamic Reference Modules

The second stage consists of modules that define a dynamic reference motion relative to the former one. The generated motions can be super-imposed, for instance, on top of any of the base reference frames  $\mathcal{R}_0$  shown earlier (inertial pointing, Hill-orbit pointing velocity-orbit pointing, celestial two-body pointing, etc.). The dynamic reference modules developed and integrated into Basilisk are: inertial 3D spinning and 3-2-1 Euler angle rotation.

The 3-2-1 Euler angle rotation module is particularly interesting because complex dynamic motions can be achieved through elegantly simple constant Euler rates. While Euler angles are often avoided in guidance algorithms because of their mathematical singularities, here the advantages of the Euler sets are exploited in a robust and safe manner that is free of numerical issues. Multiple

Euler modules can be cascaded with one another and, through different initialization/setup of the same module, a wide variety of rotational patterns can be achieved. Figure 2.6 illustrates three sample scanning patterns that can be performed by consecutive requests of Euler angle offsets and rates. In Fig. 2.6, the scanning is performed relative to the time-varying base reference frame  $\mathcal{R}_0 : \{\hat{\mathbf{r}}_{0,1}, \hat{\mathbf{r}}_{0,2}, \hat{\mathbf{r}}_{0,3}\}$ , which keeps track of the orbited celestial body.

Whereas the module presented uses a 3-2-1 Euler sequence, any of the twelve Euler sequences could be equally implemented. Interestingly, by staging the 3-2-1 sequence module up to three times, the twelve Euler combinations can be achieved. This is particularly relevant in terms of code reusability: chaining the same one module multiple times with different initializations on each stage provides the same functionality as developing the twelve modules of distinct sequences.

Next, the mathematical aspects of the 3-2-1 Euler angle rotation module are developed. An initial 3-2-1 Euler angle orientation  $\boldsymbol{\theta}_{R_1/R_0}$  and a constant set of rates  $\dot{\boldsymbol{\theta}}_{R_1/R_0}$  are defined as inputs to this dynamic module:

$$\boldsymbol{\theta}_{R_1/R_0} : \{\psi_0, \theta_0, \phi_0\}$$

$$\dot{\boldsymbol{\theta}}_{R_1/R_0} : \{\dot{\psi}, \dot{\theta}, \dot{\phi}\}$$

Because the module considers Euler rates that are constant, the associated differential kinematic

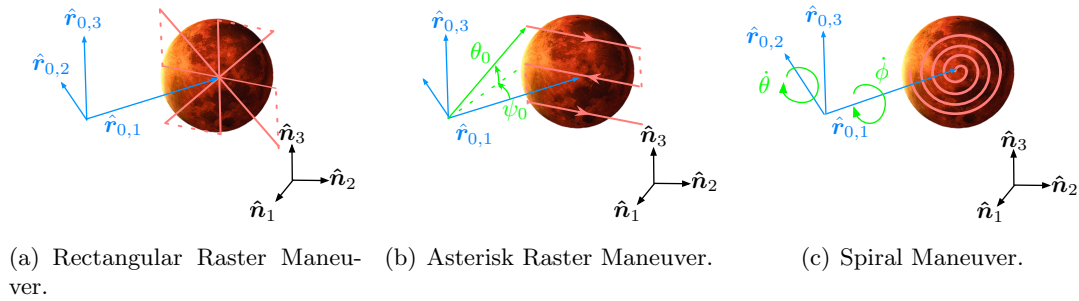


Figure 2.6: Sample scanning patterns

equations are integrable. The current Euler angles are thus expressed as

$$\psi(t) = \psi_0 + \dot{\psi}\delta t \quad (2.13a)$$

$$\theta(t) = \theta_0 + \dot{\theta}\delta t \quad (2.13b)$$

$$\phi(t) = \phi_0 + \dot{\phi}\delta t \quad (2.13c)$$

The final time-varying attitude  $[R_1N]$  is evaluated by adding the dynamic attitude  $[RR_0]$  onto the base reference attitude  $[R_0N]$ :

$$[R_1N] = [R_1R_0(\psi(t), \theta(t), \phi(t))][R_0N(\boldsymbol{\sigma}_{R_0/N})] \quad (2.14)$$

The output orientation of this dynamic module is obtained by converting the  $[RN]$  matrix into the equivalent MRP coordinates:  $[R_1N] \rightarrow \boldsymbol{\sigma}_{R_1/N}$ .

Following, the angular velocity vector  $\boldsymbol{\omega}_{R_1/N}$  and its derivative  $\dot{\boldsymbol{\omega}}_{R_1/N}$  are developed. The final angular velocity vector is defined as:

$$\boldsymbol{\omega}_{R_1/N} = \boldsymbol{\omega}_{R_1/R_0} + \boldsymbol{\omega}_{R_0/N} \quad (2.15)$$

where  $\boldsymbol{\omega}_{R_0/N}(t)$  is the angular velocity of the base reference frame (input to the module). The matrix  ${}^{R_1}\boldsymbol{\omega}_{R_1/R_0}$  is obtained from the 3-2-1 Euler angle differential kinematic equations:

$${}^{R_1}\boldsymbol{\omega}_{R_1/R_0} = \begin{bmatrix} -\sin\theta & 0 & 1 \\ \sin\phi\cos\theta & \cos\phi & 0 \\ \cos\phi\cos\theta & -\sin\phi & 0 \end{bmatrix} \begin{bmatrix} \dot{\psi} \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} \quad (2.16)$$

Substituting Eq.(2.16) into Eq. (2.15) yields the output state  $\boldsymbol{\omega}_{R_1/N}$ .

At this point, is important to follow up on the previous statement that the 3-2-1 Euler module is free of singularities. As a matter of fact, each set of Euler angles has a geometric singularity where two angles are not uniquely defined. Such geometric singularities result in mathematical singularities in the differential kinematic equations. These singularities appear always when mapping from angular body rates to Euler angle rates. However, in the present development, it is the inverse mapping takes place (i.e. the module maps Euler rates to angular velocity) and such mapping is

free of singularities. Any geometric singularity would be inevitably reflected in Eq. (2.16) and, as shown by inspection of the equation, this is not the case.

The angular acceleration of the  $\mathcal{R}_1$  reference is computed by taking the inertial derivative of Eq. (2.15). The short hand dot notation is used to denote an inertial derivative of a vector.

$$\dot{\omega}_{R_1/N} = \dot{\omega}_{R_1/R_0} + \dot{\omega}_{R_0/N} \quad (2.17)$$

The inertial derivative of the spinning vector  $\omega_{R_1/R_0}$  is evaluated using the transport theorem[42] to be

$$\dot{\omega}_{R_1/R_0} = \frac{\mathcal{N}d}{dt}(\omega_{R_1/R_0}) = \frac{R_1d}{dt}(\omega_{R_1/R_0}) + \omega_{R_1/N} \times \omega_{R_1/R_0} \quad (2.18)$$

Let us now develop the right-hand side of Eq (2.18). Making use of the fact that the defined Euler rates are constant, the following expression is obtained for the  $\mathcal{R}_1$ -frame derivative of  $\omega_{R_1/R_0}$ :

$$\frac{R_1d}{dt}(\omega_{R_1/R_0}) = \begin{bmatrix} -\dot{\theta}\dot{\psi} \cos \theta \\ (\dot{\phi} \cos \phi \cos \theta - \dot{\theta} \sin \phi \sin \theta)\dot{\psi} - \dot{\phi}\dot{\theta} \sin \phi \\ -(\dot{\phi} \sin \phi \cos \theta + \dot{\theta} \cos \phi \cos \theta)\dot{\psi} - \dot{\phi}\dot{\theta} \cos \phi \end{bmatrix} \quad (2.19)$$

Simplifying Eq. (2.18) and joining this equation with Eq. (2.17), the  $\mathcal{N}$ -frame angular acceleration is eventually obtained

$$\dot{\omega}_{R_1/N} = \frac{R_1d}{dt}(\omega_{R_1/R_0}) + \omega_{R_0/N} \times \omega_{R_1/R_0} + \dot{\omega}_{R_0/N} \quad (2.20)$$

With these, all the variables conforming the output structure of the 3-2-1 Euler rotation have been derived:  $\mathcal{R}_1 = \{\sigma_{R_1/N}, \mathcal{N}\omega_{R_1/N}, \mathcal{N}\dot{\omega}_{R_1/N}\}$ .

### 2.2.2.3 Tracking Error Module

The last component in the guidance chain is always the tracking error module, responsible of ensembling the final reference  $\mathcal{R}$  and computing the tracking errors with respect to the principal body frame  $\mathcal{B}$ . The novelty of this module is found in the strategy used for controlling a spacecraft-fixed frame  $\mathcal{B}_c$  that is not the principal one. Instead of expressing the guidance output in terms of the specific control frame  $\mathcal{B}_c$  (which would then have to be accounted for in the control law),

the offset between the main body frame  $\mathcal{B}$  and the control frame  $\mathcal{B}_c$  is added to the generated final reference  $\mathcal{R}$ . The advantage of this trick is that it reduces the number of transformations required to guide and control a generic spacecraft fixed-frame that is not the principal one.

For example, Fig. 2.7 shows the spacecraft principal body frame  $\mathcal{B}$  in blue and a body-fixed control frame  $\mathcal{B}_c$  in magenta. Here, the magenta frame is the one containing the instrument boresight that has to perform the guidance scanning maneuver. However, recall from Fig. 2.4 that the output of the guidance block is designed to be expressed in the main body frame  $\mathcal{B}$  as:  $\{\sigma_{B/R}, {}^{\mathcal{B}}\omega_{B/N}, {}^{\mathcal{B}}\omega_{R/N}, {}^{\mathcal{B}}\dot{\omega}_{R/N}\}$ .

Theoretically, the guidance output could be expressed in the specific control frame  $\mathcal{B}_c$  for each case. Nevertheless, it would then be necessary to keep track of the relationship between the principal body frame and the control body frame  $[B_c B]$  in multiple modules inside the guidance-control sequence. Looking at the control law formulation presented in Eq. (A.1), it is observed that all the vector and matrix components would then need to be mapped to this new body-fixed control frame.

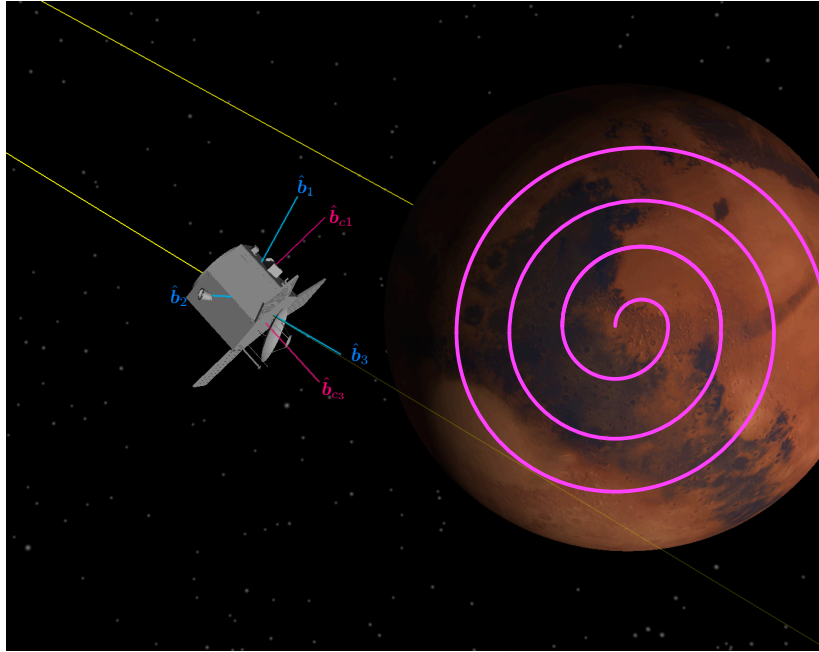


Figure 2.7: Principal body frame (blue) and control frame (magenta)

An alternative tracking error solution that is more elegant is derived following. Let us define the input reference to the attitude tracking error module as  $\mathcal{R}_i = \{\sigma_{R_i/N}, {}^N\omega_{R_i/N}, {}^N\dot{\omega}_{R_i/N}\}$ , where  $\mathcal{R}_i$  has been generated by superposition of base and dynamic reference modules. In the general case where the body-fixed frame to be controlled is not the spacecraft's primary frame  $\mathcal{B}$ , the control frame  $\mathcal{B}_c$  should be guided towards the desired reference  $\mathcal{R}_i$ . The control body-frame  $\mathcal{B}_c$  could correspond to any body-fixed frame that differs from  $\mathcal{B}$  by a constant angular offset  $[B_c B]$ . Thus, as  $\mathcal{B}_c \rightarrow \mathcal{R}_0$ , then  $\mathcal{B} \rightarrow \mathcal{R}$  if  $\mathcal{R}$  is defined such that

$$[B_c B] = [R_i R] \quad (2.21)$$

As illustrated earlier in Fig. 2.4, the attitude tracking error module receives navigation data containing the estimated spacecraft orientation  $\sigma_{B/N}$ . In addition, the offset between control and principal body frames  $[B_c B]$  is set as a configurable parameter within the tracking error module. With this information along with the input reference  $[R_i N(\sigma_{R_i/N})]$ , the orientation of the final reference frame  $\mathcal{R}$  (such that  $\mathcal{B} \rightarrow \mathcal{R}$ ) can be readily derived:

$$[RN] = [RR_i][R_i N] = [B_c B]^T [R_i N] \quad (2.22)$$

With the discussed generalization, it is possible to correct any sensor or component frame orientation using the same classical tracking error algorithms and control laws that deal with the main body frame. Therefore, the need of creating particular code for the guidance and control of different spacecraft-fixed frames is overcome.

### 2.2.3 Numerical Simulations

The rigid body equations of motion in Eq. (A.1) are numerically simulated in Basilisk to validate and illustrate the performance of the presented guidance modules. The orientation is controlled through a set of four RWs. The dynamic states are integrated using a Range-Kutta 4 scheme running at 10 Hz. The MRP feedback control in Eq. (A.4) is used to drive the spacecraft orientation towards the desired reference motion using an update rate of 2Hz. The estimated

Table 2.1: Initial orbital elements

Parameter	Value	Units
Semi-major Axis	7471.618( $\approx 2.2R_{\text{Mars}}$ )	km
Eccentricity	0.4	
Inclination	0.0	deg
Longitude of Ascendant Node	0.0	deg
Argument of Perigee	0.0	deg
True Anomaly	270.0	deg

Table 2.2: Control and spacecraft parameters

Parameter	Value	Units
Attitude Error Gain K	2.531	$\frac{\text{kg}\cdot\text{m}^2}{\text{s}}$
Rate Error Gain P	45.0	$\frac{\text{kg}\cdot\text{m}^2}{\text{s}}$
$\hat{\mathbf{g}}_{s_1}$	$[-0.5, 0.5, -\frac{\sqrt{2}}{2}]$	-
$\hat{\mathbf{g}}_{s_2}$	$[0.5, 0.5, -\frac{\sqrt{2}}{2}]$	-
$\hat{\mathbf{g}}_{s_3}$	$[0.5, -0.5, -\frac{\sqrt{2}}{2}]$	-
$\hat{\mathbf{g}}_{s_4}$	$[-0.5, -0.5, -\frac{\sqrt{2}}{2}]$	-
$J_{s_i}$	0.1591549	$\text{kg}\cdot\text{m}^2$
$I_1, I_2$	700	$\text{kg}\cdot\text{m}^2$
$I_3$	800.0	$\text{kg}\cdot\text{m}^2$

navigation data  $\sigma_{B/N}$  and  $\omega_{B/N}$  is without any sensor corruptions to better illustrate that the control law does achieve asymptotic tracking of the reference motion. Each simulation has a maneuver duration of 9600 seconds ( $\approx 2.7$  hours). The spacecraft is simulated to be orbiting Mars with the orbital parameters in Table 2.1 and the control-related parameters in Table 2.2.

The spacecraft is flying through the periapses region of a highly eccentric orbit. Consequently, there is a considerably amount of variability on the spacecraft orbit rates. In the first control scenario, asymptotically tracking this orbital motion emphasizes the challenges of properly evaluating acceleration and rates. With the chosen proportional gain  $P$ , the control time decay constant [42] is  $\tau \approx 30$  sec. The  $K$  gain is computed to yield a critically damped system. This choice of gains provides a non-aggressive control response. Although initial requests of large torques are unavoidable (i.e. detumbling phase), operating with non-saturated reaction wheels is guaranteed for the rest of the simulation time (i.e. tracking phase).

### 2.2.3.1 Orbit Axis Rotation

The first simulation consists on performing a nadir-spinning maneuver by using the stack of modules shown in Fig. 2.8. The base pointing module generates a reference that is initially aligned with the Hill-orbit frame. By introducing an attitude offset in the tracking error module, a 180 degree rotation about the third body axis  $\hat{\mathbf{b}}_3$  is achieved. In this way, the resulting frame points the first principal axis of the spacecraft body  $\hat{\mathbf{b}}_1$  towards the planet, i.e. in the nadir direction ( $-\hat{\mathbf{i}}_r$ ) instead of radially outwards. In addition, a dynamic spinning motion about  $\hat{\mathbf{i}}_r$  is superimposed on top of the base reference. With proper tracking,  $\hat{\mathbf{b}}_1$  remains consequently fix while  $\hat{\mathbf{b}}_2$  and  $\hat{\mathbf{b}}_3$  rotate in the local-horizontal orbit plane.

As shown in Fig. 2.8, both the Euler rotation module and the tracking error module need to be configured with maneuver-specific parameters. The configuration parameters for the Euler rotation module are provided in Table 2.3. In turn, the tracking error module is configured as follows:

$$\boldsymbol{\sigma}_{Bc/B} = [0.0, 0.0, 1.0] \quad (2.23)$$

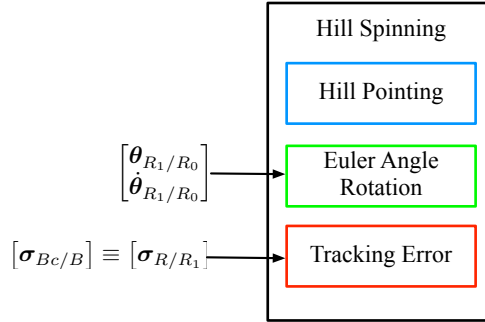


Figure 2.8: Nadir-spinning stack

Table 2.3: Configuration data for the Euler angle rotation module

Parameter	Value	Units	Description
$\boldsymbol{\theta}_{R_1/R_0} : \{\psi, \theta, \phi\}$	$[0.0, 0.0, 0.0]$	deg/s	Initial 3-2-1 set of Euler angles relative to the base $\mathcal{R}_0$ .
$\dot{\boldsymbol{\theta}}_{R_1/R_0} : \{\dot{\psi}, \dot{\theta}, \dot{\phi}\}$	$[0.0, 0.0, 0.3]$	deg/s	Desired 3-2-1 set of Euler rates relative to the base $\mathcal{R}_0$ .



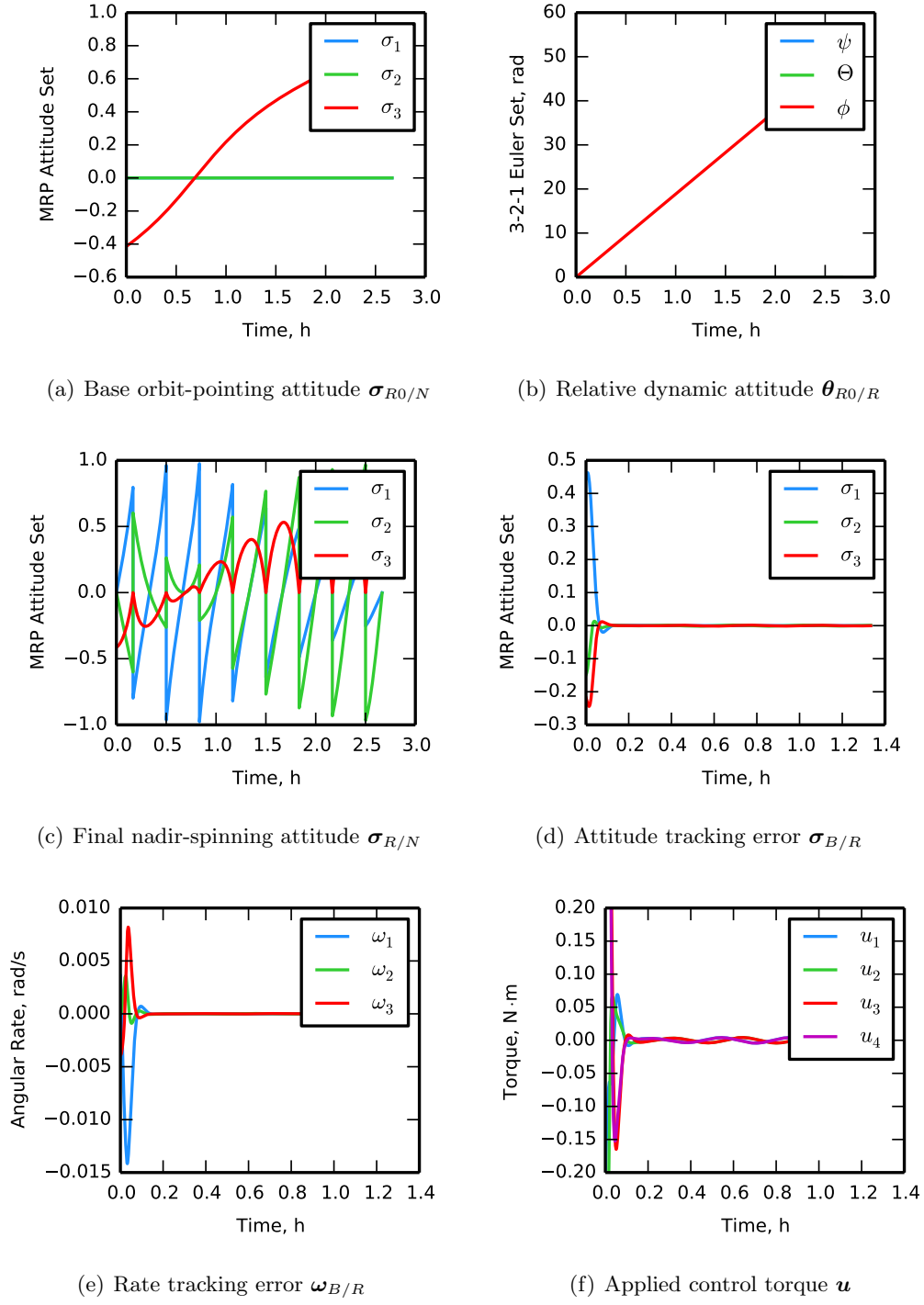


Figure 2.9: Nadir spinning: cascaded attitude sets

The first three plots in Fig. 2.9 shows the reference attitude sets generated at each stage. Figure 2.9(a) illustrates the time varying Hill-frame orientation, denoted as  $\mathcal{R}_0$  in this scenario.

The rapid change observed on the base pointing attitude is due to the spacecraft flying through the periapses of a highly elliptic orbit. The constant Euler angle rates in Fig. 2.9(b) show that only  $\dot{\phi} \neq 0$ . Cascading these two reference frames together yields the final desired attitude shown in Fig. 2.9(c). In turn, the last three plots in Fig. 2.9 displays the tracking errors and the commanded control torque. Inspection of the tracking error plots reveals that the spacecraft body frame aligns asymptotically with the desired reference frame; therefore, proving proper kinematic superposition in the assembly of reference frames. The control torque in Fig. 2.9 is capped to a maximum RW torque of 0.2 N·m. Note that the variable  $u_i$  corresponds to the torque of each wheel and wheel saturation only takes place at the very beginning of the maneuver. The commanded torques in Fig. 2.9(f) do not converge to zero and this is, indeed, expected: a nadir-spinning maneuver is not a natural equilibrium motion and, therefore, a certain torque will always be necessary to maintain it.

### 2.2.3.2 Asterisk Scanning Maneuver

The second numerical simulation consists on performing an asterisk scanning pattern. An asterisk pattern is conformed by a total of four raster lines and, by using the Euler rotation module, the complete pattern can be achieved with only four command steps. The stack of modules necessary to perform such maneuver is depicted in Fig. 2.10. Note that a raster manager module is attached to the Euler rotation module. The raster manager commands a sequence of Euler angle

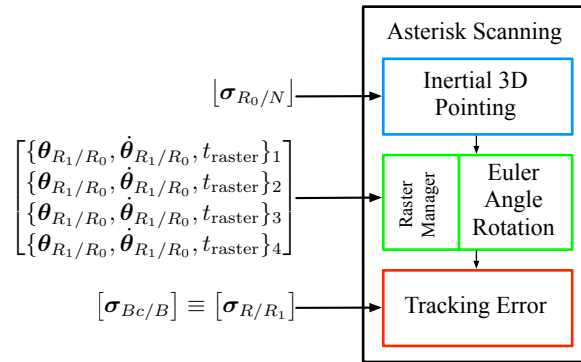


Figure 2.10: Inertial asterisk scanning stack

Table 2.4: Configuration data for the raster manager module

Parameter	Value	Units	Description
$\theta_{R_1/R_0} : \{\psi, \theta, \phi\}_1$	$[\alpha, 0.0, 0.0]$	deg	Initial 3-2-1 Euler angle set of the first raster.
$\dot{\theta}_{R_1/R_0} : \{\dot{\psi}, \dot{\theta}, \dot{\phi}\}_1$	$[-\dot{\alpha}, 0.0, 0.0]$	deg/s	3-2-1 Euler angle rates of the first raster.
$\theta_{R_1/R_0} : \{\psi, \theta, \phi\}_2$	$[-\alpha, -\alpha, 0.0]$	deg	Initial 3-2-1 Euler angle set of the first raster.
$\dot{\theta}_{R_1/R_0} : \{\dot{\psi}, \dot{\theta}, \dot{\phi}\}_2$	$[\dot{\alpha}, \dot{\alpha}, 0.0]$	deg/s	3-2-1 Euler angle rates of the second raster.
$\theta_{R_1/R_0} : \{\psi, \theta, \phi\}_3$	$[\alpha, -\alpha, 0.0]$	deg	Initial 3-2-1 Euler angle set of the third raster.
$\dot{\theta}_{R_1/R_0} : \{\dot{\psi}, \dot{\theta}, \dot{\phi}\}_3$	$[-\dot{\alpha}, \dot{\alpha}, 0.0]$	deg/s	3-2-1 Euler angle rates of the third raster.
$\theta_{R_1/R_0} : \{\psi, \theta, \phi\}_4$	$[0.0, \alpha, 0.0]$	deg	Initial 3-2-1 Euler angle set of the forth raster.
$\dot{\theta}_{R_1/R_0} : \{\dot{\psi}, \dot{\theta}, \dot{\phi}\}_4$	$[0.0, -\dot{\alpha}, 0.0]$	deg/s	3-2-1 Euler angle rates of the forth raster.
$t_{\text{raster}, i}$	1600	s	Time duration of each commanded raster maneuver.

offsets and rates at the configured raster times. Each raster lasts for 1600 seconds ( $\approx 0.45$  hours) and, as mentioned, a total of 4 Euler commands is used to complete the pattern.

The inertial pointing module and attitude tracking error module are initialized with the following parameters:

$$\sigma_{R_0/N} = [0.0, 0.0, 0.0] \quad (2.24a)$$

$$\sigma_{B_c/B} = [0.0, 0.0, 0.0] \quad (2.24b)$$

Thus, the requested inertial attitude is that of the global inertial frame with  $\mathcal{R}_0 \equiv \mathcal{N}$ . The control body frame coincides with the principal body frame through  $\mathcal{B}_c \equiv \mathcal{B}$  or, equivalently,  $[B_c B] = [I_{3 \times 3}]$ . The configuration parameters of the raster manager are provided in Table 2.4.

The parameters  $\alpha$  and  $\dot{\alpha}$  in Table 2.4 are defined as follows:

$$\alpha = 8.0 \text{ deg} \quad (2.25)$$

$$\dot{\alpha} = \frac{2\alpha}{t_{\text{raster}}} \quad (2.26)$$

Figure 2.11 shows plots of the attitude tracking error and the commanded wheel control torques during the maneuver. Each peak corresponds to the commanding of a new raster line. After each command, it is observed that the spacecraft quickly converges into the new raster.

Figure 2.12(a) compares the desired nominal raster lines (dark blue lines) with the actual yaw-pitch angles scanned in the maneuver (magenta lines). In order to get into the desired rasters

on time, a small angle offset ( $\alpha_{\text{offset}}$ ) needs to be commanded, which is also depicted with light-blue color. The value of  $\alpha_{\text{offset}}$  is a tradeoff with the picked control gains  $K, P$ . In this case, an offset of  $\alpha_{\text{offset}} = 0.5\alpha$  is found to work well for all the cases analyzed. The additional of this angular offset demands, in turn, small adjustments on the commanded maneuver times, in order to make sure

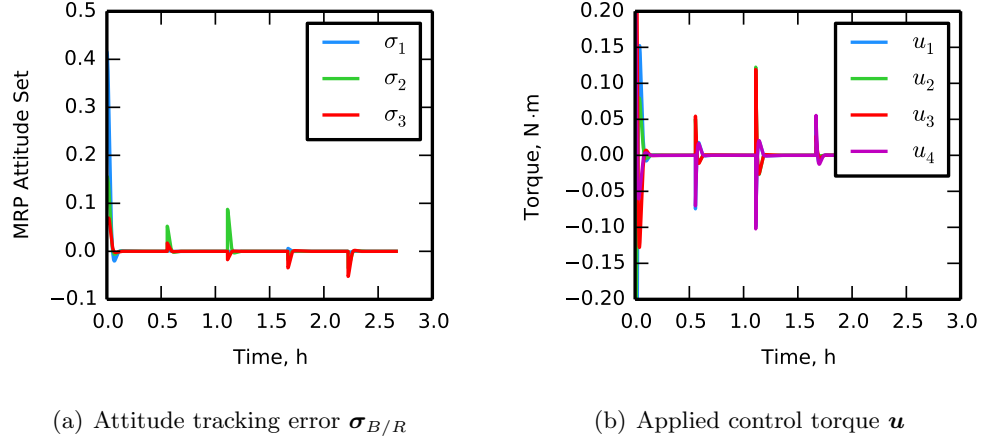
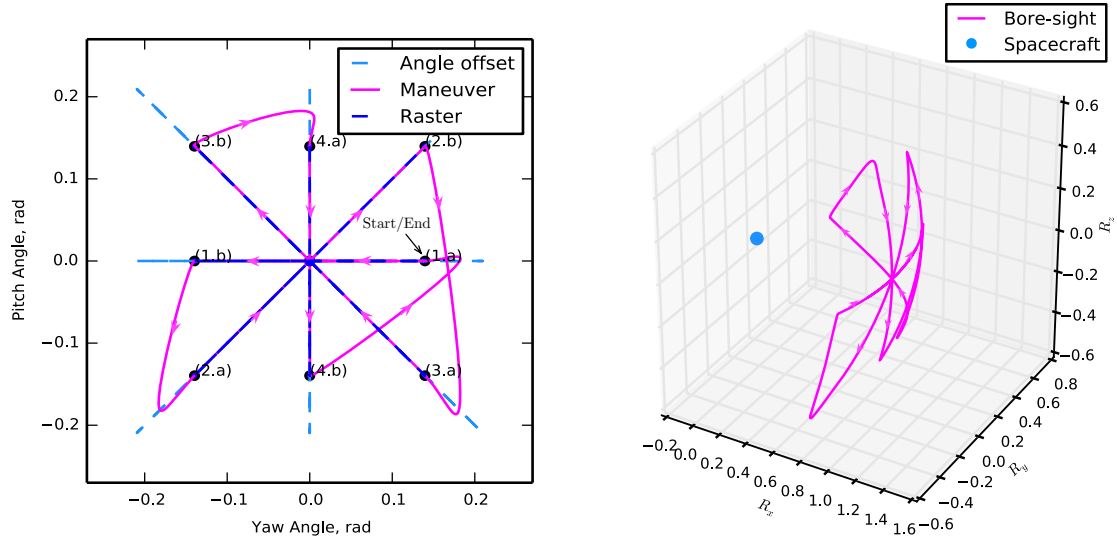


Figure 2.11: Asterisk scanning: attitude tracking error and control torque



(a) Nominal Rasters vs. Achieved Maneuver for  $\alpha = 8$  deg. (b) Achieved 3D Bore-sight Pointing for  $\alpha = 24$  deg.

Figure 2.12: Simulated asterisk scanning patterns

that the spacecraft does not move to the next raster before finishing the current one.

The raster lines in Fig. 2.12(a) are numbered from 1 to 4 according to their order of execution. In addition, the starting point of each raster line is labeled with the letter  $a$  and the ending point with the letter  $b$ . Upon arrival at the end point of the pattern (i.e. point 4( $b$ )) the spacecraft is smoothly driven back to beginning (i.e. point 1( $a$ )).

Figure 2.12(b) shows the three-dimensional view of the spacecraft's maneuver for a nominal angle of  $\alpha = 24$  degrees. Note that the coordinates used in this case are Cartesian. The single blue dot corresponds to the position of the spacecraft, from which the scanned pattern is projected on a unit sphere.

### 2.2.3.3 Spiral Scanning Maneuver

The last numerical simulation performs a scanning maneuver that draws a spiral pattern on the inertial frame. This scenario is particularly interesting to showcase the superposition of multiple dynamic reference modules. A spiral motion can be easily defined through a 1-2 sequence of constant Euler rates. The stack of modules required to perform a single-spiral maneuver is shown in Fig. 2.13. In this stack, the inertial 3D pointing module is used as the base reference for the sake of simplicity in analyzing the resulting plots. In order to simulate a 1-2 Euler sequence, two

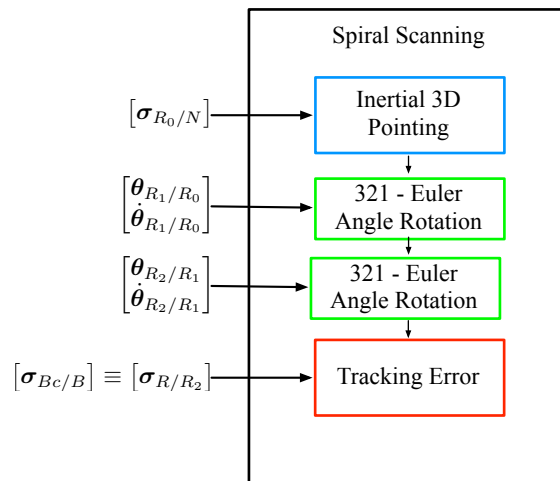


Figure 2.13: Stack of modules for a spiral scanning maneuver

Euler rotation modules are staged as dynamic references.

Figure 2.14 shows numerical results of the commanded scan versus the one actually achieved. Here, the nominal (i.e. commanded) pointing motion is depicted in magenta while the actual actual angles scanned by the bore-sight are depicted in blue. After the helix pattern is completed, the spacecraft's bore-sight is smoothly driven back to the starting scanning point at the center of the plot.

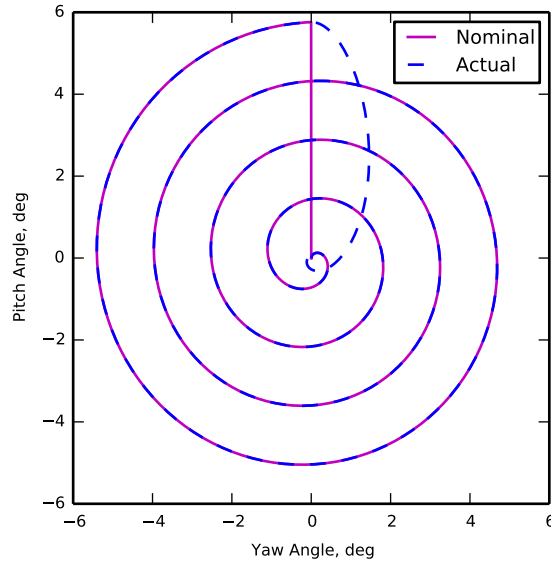
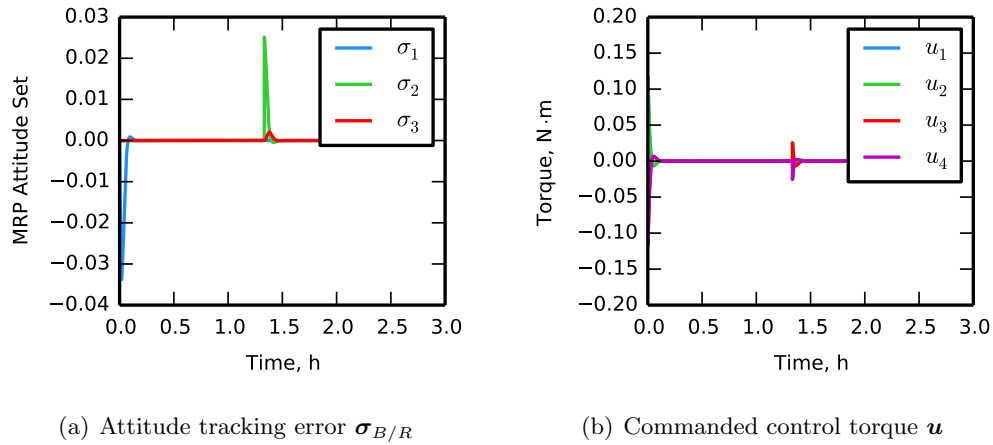


Figure 2.14: Spiral scanning: nominal pointing vs. actual



(a) Attitude tracking error  $\sigma_{B/R}$

(b) Commanded control torque  $\mathbf{u}$

Figure 2.15: Spiral scanning: tracking error and wheel torques

Figure 2.15 shows plots of the attitude tracking error and the commanded wheel torques during the maneuver. Two peaks are observed in these plots: one at the beginning of the simulation (when the spacecraft is first commanded to start the spiral pointing motion) and one in the middle (when the spacecraft has finished the spiral-pointing maneuver and it goes back to fix inertial pointing). Between the two peaks in Fig. 2.15(b), it is observed that the reference motion is perfectly tracked with small wheel torques.

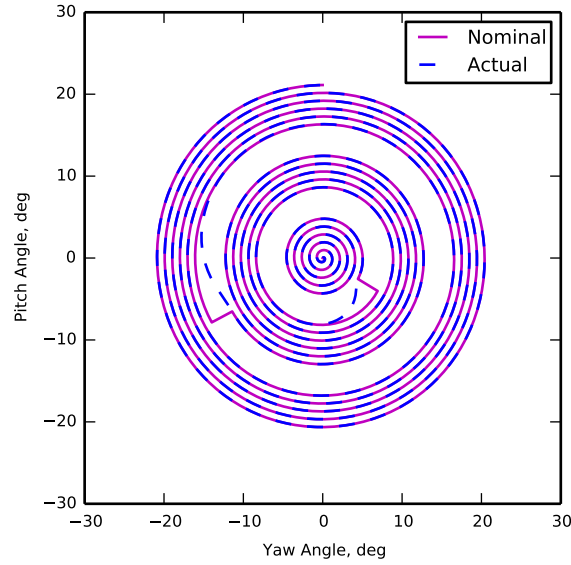


Figure 2.16: Triple-spiral scanning: nominal pointing vs. actual

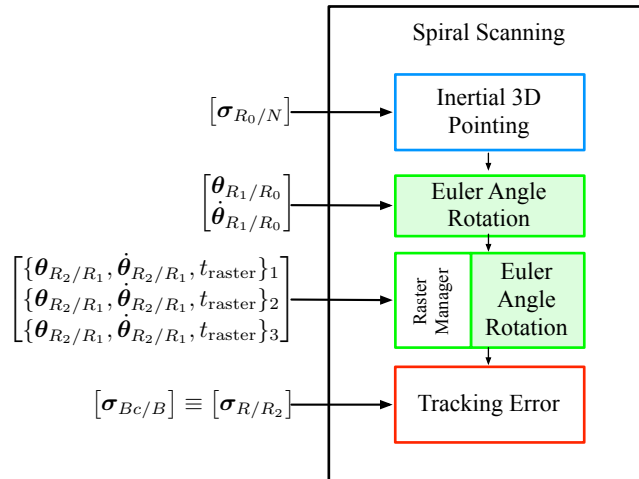


Figure 2.17: Stack of modules for a triple-spiral scan

To finalize this section, Fig. 2.16 shows how the single-spiral scanning can be extended into multiple spirals. This extension can be easily achieved by using the raster manager module introduced earlier. The corresponding stack of modules is illustrated in Fig. 2.17.

## 2.3 Summary

This chapter has introduced the Basilisk software framework, a desktop testbed for prototyping flight algorithms and testing them in closed-loop dynamic simulations. Upon Basilisk, a novel reference generation architecture is implemented where complex attitude patterns are achieved through combination of atomic guidance modules that fulfill a well-defined functionality. As a quick recapitulation, there are three core functionalities that conform all guidance reference motions: base pointing reference, dynamic reference and attitude offset. For each one of these functionalities, different software guidance modules are implemented and showcased; the mathematical equations are developed and numerical simulations illustrate how the individual components are arranged to support different rotational dynamics mission requirements. This layered strategy promotes reusability of code throughout distinct mission profiles and, as a matter of fact, it is currently being applied in an interplanetary spacecraft mission to perform science and nominal attitude maneuvers.



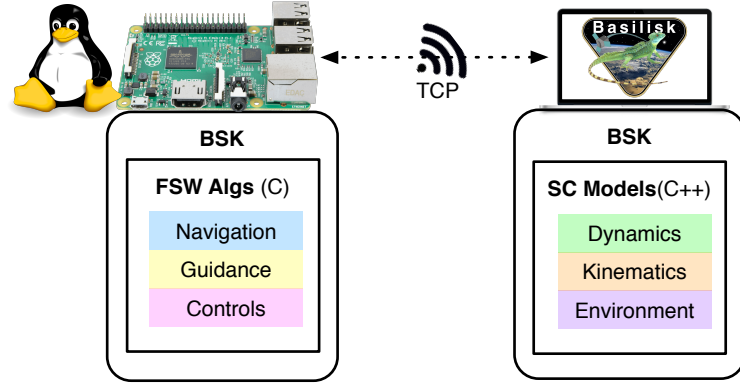
## Chapter 3

### Flight Algorithm Migration into Commercial Flight Targets

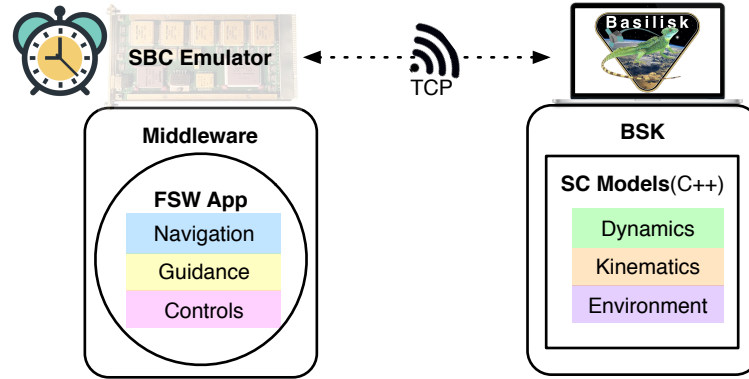
Once flight algorithms are proven to satisfy mission requirements in the desktop development environment, the next step is to migrate them into the mission flight target of choice. As a matter of fact, Basilisk-developed flight algorithms have been ported into several flight targets: commercial processors (like the Raspberry Pi) as well as embeddable middleware systems (like cFS and MicroPython). The migration into these kind of targets is illustrated in Fig. 3.1. A key aspect conveyed by Fig. 3.1 is that FSW shall migrate from the desktop environment into another target while preserving the capability of closing the loop with a dynamics simulation that remains on the desktop computer. The reason for maintaining the dynamics simulation running on a desktop computer is to preserve higher fidelity, since traditional spaceflight processors lag state-of-the-art consumer technology notably.

The idea of use using commercial processors in redundant configurations (rather than a single radiation hardened-processor) for flight applications has received increased attention and interest in the recent decades[9, 48, 22, 17]. This chapter describes, in particular, the port of the Basilisk flight architecture into the Raspberry Pi commercial hardware. The modular and scalable nature of Basilisk and the performance and affordability of the Raspberry Pi are combined into a flight system that is ideal for –although not restricted to– CubeSats and small-satellite form factors.

The idea pursuit in this chapter consists on installing the flight portion of Basilisk on the Raspberry Pi and, then, performing a closed-loop maneuver with the setup illustrated in Fig. 3.1(a) in order to prove the validity of the distributed architecture. While a Basilisk closed-loop simulation



(a) FSW on the Raspberry Pi: ARM processor and Linux OS



(b) FSW on middleware, embedded in an SBC (single-board computer) emulator

Figure 3.1: Migration of the flight application

could run entirely on the Pi platform (i.e. single platform simulation containing both the flight algorithms and the spacecraft physical simulation), the distributed configuration in Fig. 3.1(a) replicates reality better.

The chapter is outlined as follows: first, a brief motivation for using commercial processors like the Raspberry Pi in space applications is presented. Next, the challenges of porting Basilisk into the Raspberry Pi are described. Then, the implementation of a peer-to-peer router that allows distributed TCP communication is explained. Finally, numerical results of a distributed inertial guidance maneuver, in which the flight algorithms run on the Raspberry Pi and the spacecraft simulation run on a separate host computer, are shown.

### 3.1 The Raspberry Pi for Space Applications

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 (e.g. single event upset crash). Generally, two approaches are employed, independently or in combination, to protect the spacecraft’s electronic systems in the radiation environment:

- (1) Commercial-off-the-shelf (COTS) parts in redundant and duplicative configuration.
- (2) Electronics hardened for radiation and environmental exposure.

Each of these methods presents its pros and cons[29]. However, 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) is an effective method to minimize costs –hence, being very compelling for small-sat and low-cost missions.

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 hardware[48]. For example, the PhoneSat<sup>1</sup> project is a technology demonstration mission launched in 2013 with the aim of proving that smartphones could be used as avionic systems in nano-satellites and that they would survive. In the context of realistic flight-like testing, the Pi-Sat project developed by NASA Goddard proposes the Raspberry Pi as a flight computing environment on which to run the core Flight System middleware[22]. Similarly, this chapter aims to use the Raspberry Pi as the flight target in which to run Basilisk-developed FSW applications.

### 3.2 Distributed Basilisk Simulation using the Raspberry Pi

Now that the interest on using the Raspberry Pi as a flight processor has been introduced, this section explains the process of 1) building Basilisk on the Pi platform and 2) testing the system in distributed closed loop.

---

<sup>1</sup> [https://www.nasa.gov/directorates/spacetech/small\\_spacecraft/phonesat.html](https://www.nasa.gov/directorates/spacetech/small_spacecraft/phonesat.html)

First, the technical challenges of porting Basilisk on the Raspberry Pi are outlined. The Raspberry Pi has a built-in ARM processor and comes, out of the box, with the Debian operating system, which is a flavour of Linux. While Basilisk is cross-platform in nature and supports Linux, MacOS and Windows, compiling and building the Basilisk software on Debian required additional technical effort, mainly due to the SPICE<sup>2</sup> library included within Basilisk. It was necessary to build from source the Cspice Linux 32bit version but manually modifying the compilation flags in order to target the 64-bit CPU architecture of the latest Raspberry Pi hardware. All the other Basilisk libraries were linked without problems.

Once Basilisk numerical simulations are proved to run on the Raspberry Pi, the next step is to move from a single-platform simulation into a multi-platform simulation where the FSW algorithms and the spacecraft physical models are executed on different computing platforms, as in Fig. 3.1. In order to enable TCP communication, a Basilisk router module based on the Boost C++ library<sup>3</sup> was used. While Boost can handle both transport and serialization of data across platforms, its memory footprint is quite large. The router module could act either as a server or client and therefore it would only allow a peer-to-peer communication from one Basilisk process (e.g. FSW process on the Raspberry Pi) to another one (e.g. spacecraft physical simulation on the desktop computer). The one-to-one nature of this router module implies that the communication map is not scalable to more than two processes and that, similarly, multi-broadcast architectures are not supported. Despite these limitations, the router model is important for two reasons:

- (1) It allowed the very first distributed numerical simulation within Basilisk.
- (2) It paved the ground for the development of the flexible and scalable Black Lion communication architecture (see Section 5.2) .

Next, a distributed closed-loop numerical simulation is shown. In this simulation, the physical spacecraft (on the desktop computer) is initially tumbling and the FSW algorithms (on the Raspberry Pi) are responsible of bringing the spacecraft into an inertial-pointing mode. The setup

---

<sup>2</sup> <https://naif.jpl.nasa.gov/naif/toolkit.html>

<sup>3</sup> <https://www.boost.org>

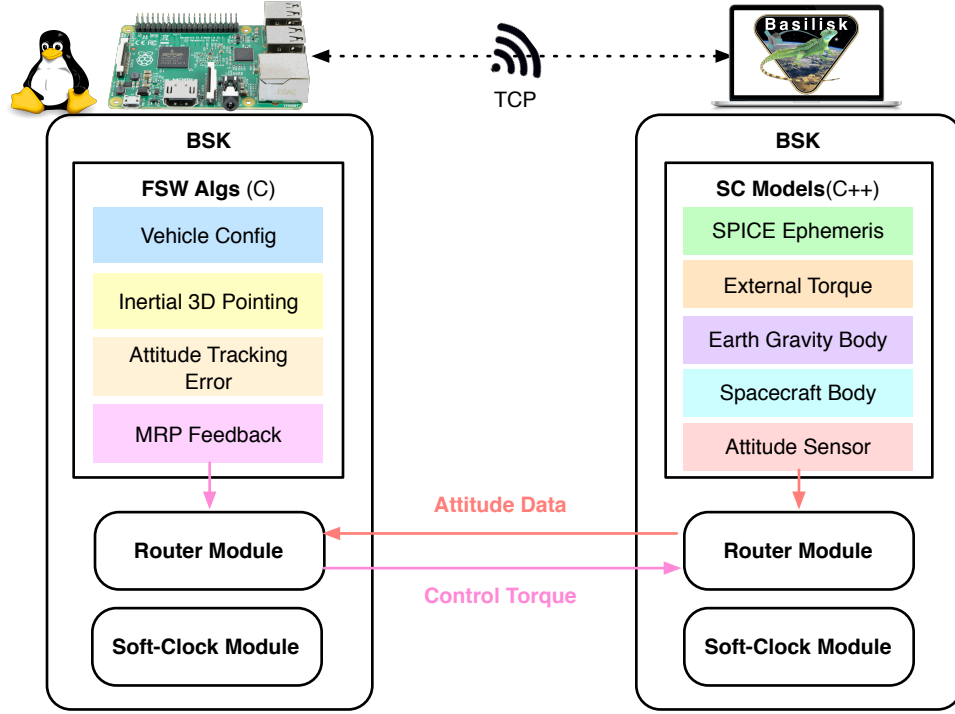
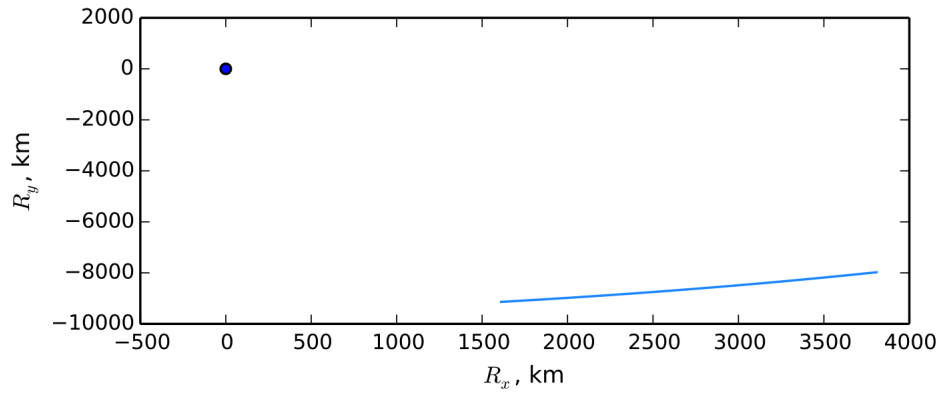


Figure 3.2: Numerical simulation setup

of the distributed numerical simulation is shown in Fig. 3.1. Both Basilisk (BSK) processes contain the corresponding modules plus a router module and a soft-clock module that allows both processes to run in synch and in soft real time. There are two messages being exchanged via TCP connection:

- (1) Sensed attitude data: from the spacecraft physical simulation to the FSW algorithms.
- (2) Commanded control torque: from the FSW algorithms to the spacecraft physical simulation.

Figure 3.3 shows plots of the spacecraft physical simulation in response to the closed loop with FSW. In turn, Fig. 3.4 shows the closed-loop evolution of FSW states. The highlight of the results is that the distributed maneuver is successfully achieved. In addition, Ref. [17] emphasizes the fact that the FSW results are exactly the same whether the closed-loop simulation is executed in a distributed multi-platform fashion (as in Fig. 3.1) or on a single computing platform.



(a) Spacecraft orbit around Earth

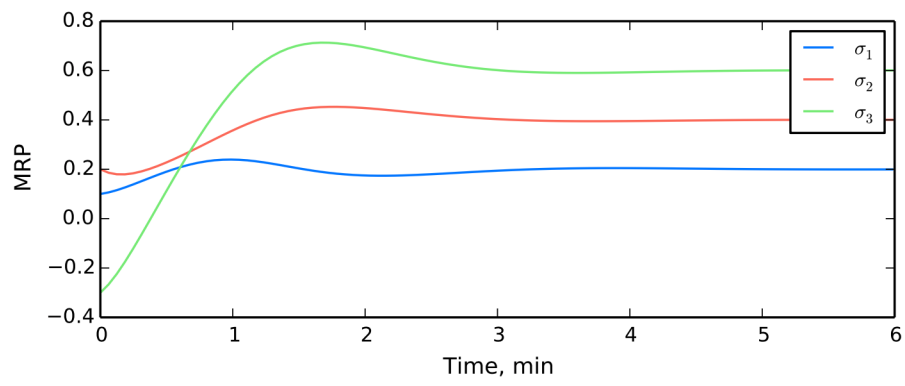
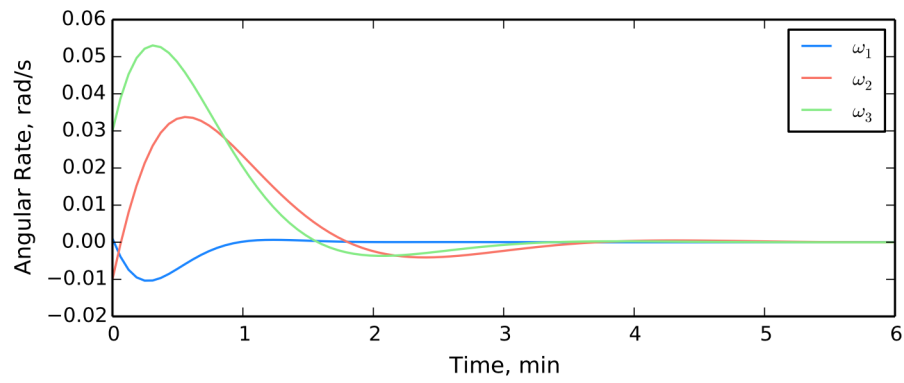
(b) Spacecraft inertial MRP attitude set,  $\sigma_{B/N}$ (c) Spacecraft inertial angular rate,  ${}^B\omega_{B/N}$ 

Figure 3.3: Results from the spacecraft physical simulation

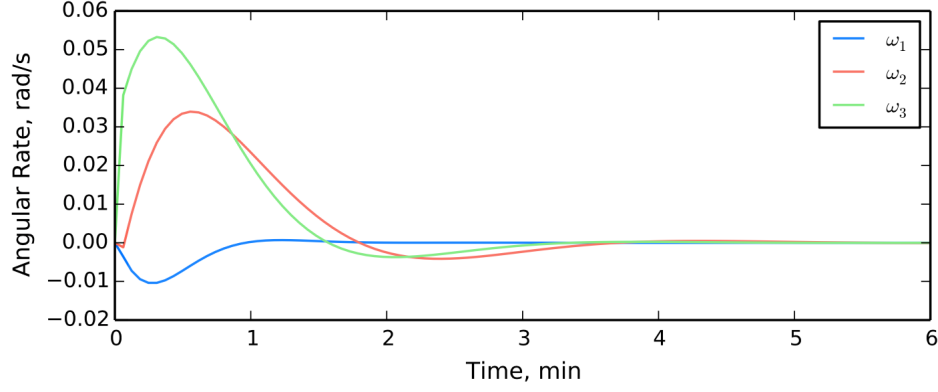
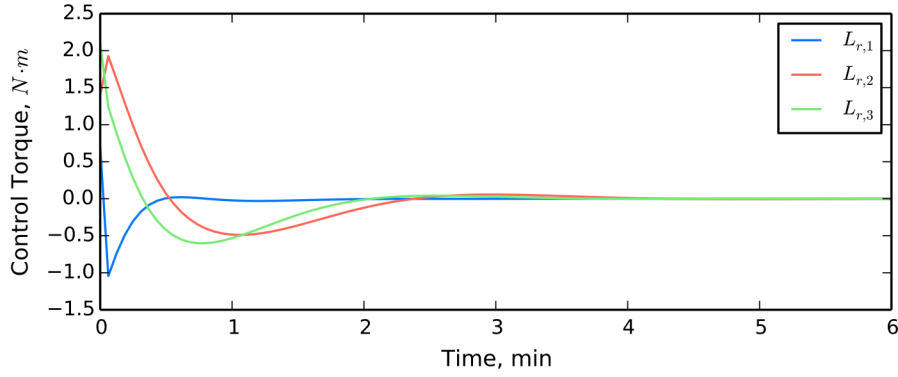
(a) Angular rate error,  ${}^B\omega_{B/R}$ (b) Control torque,  $\mathbf{L}_r$ 

Figure 3.4: Results from the FSW process on the Raspberry Pi

### 3.3 Summary

This chapter has introduced the interest on using commercial processors for flight applications. On these lines, the challenges associated into porting the Basilisk architecture into the Raspberry Pi commercial hardware have been outlined. Most importantly, the first demonstration of a distributed Basilisk simulation on a Raspberry Pi has been shown. The distributed nature of the simulation implies that the flight algorithms and the spacecraft physical simulation run on separate computing platforms, which is a key step towards more realistic, flight-like testing. In addition, the use of a peer-to-peer communication router that enables TCP communication between a client and a server has been explained. This communication router is also an initial milestone

towards the development of the flexible and multi-propose Black Lion communication architecture (described later on, in Chapter 5.2).



## Chapter 4

### Flight Algorithm Migration into the core Flight System

While the work with commercial targets shown in the previous chapter constitutes the first proof of concept towards distributed testing of Basilisk flight applications, the present chapter focuses on the migration of Basilisk flight algorithms into the cFS (i.e. a middleware target). The relevance of migrating Basilisk-developed flight algorithms into cFS is clear in that the latter is a standard and widely-used middleware layer for space missions that ensures portability of the flight application among different radiation-hardened processors and RTOS. The result of migrating Basilisk flight algorithms into the cFS middleware is a cFS-FSW application that can be readily integrated into an embedded system like an emulated flight processor (as in Fig. 3.1(b)). Such strategy is being currently employed for the development of the interplanetary mission in which LASP and the AVS laboratory are collaborating. The challenges associated to this work is that the cFS middleware is an embeddable system and, by nature, there is a considerable gap between the Basilisk desktop environment and the cFS environment.

This chapter is outlined as follows: first, the architecture of the cFS middleware is described. Next, the technical work required to migrate the Basilisk flight application (which is written in a combination of Python and C) into a pure-C cFS application is presented. Then, the tool developed to smoothen out the transition process back and forth desktop and cFS environments is showcased. This tool is named **AutoSetter**, it is currently available as an open-source tool and, in addition, pseudo-code revealing its working mechanisms is provided in Appendix B. To finalize the chapter, a summary section highlighting the shown results is included.

## 4.1 The cFS Middleware

First and foremost, let us provide further insight on the cFS itself. The cFS is a middleware layer aimed at ensuring portability of flight applications among different RTOS and processor boards. It is an open-source product developed by NASA Goddard that has inherited software from flight missions for over 20 years. It is mostly written in the C programming language and its architectural design of cFS is depicted in Fig. 4.1. Starting from the highest level of the architecture to the lowest: first, there is the application layer, which is where the mission-specific flight algorithms reside –therefore this layer is always customized by the user. Below, there is a library layer, where common components that are typically part of a FSW system are available for sharing and reuse (e.g. file delivery protocol, checksum, housekeeping, etc.). In the middle there is the core Flight Executive layer (cFE), which is the central piece of cFS and provides five core services: executive, event, software bus, table and time services. One level lower there is the platform and OS abstraction layer, which are the key pieces enabling portability. Finally, the very bottom is where the boot software resides.

Now that the cFS middleware has been properly introduced, let us dive into the technical steps required for migrating Basilisk flight algorithms into a cFS application.

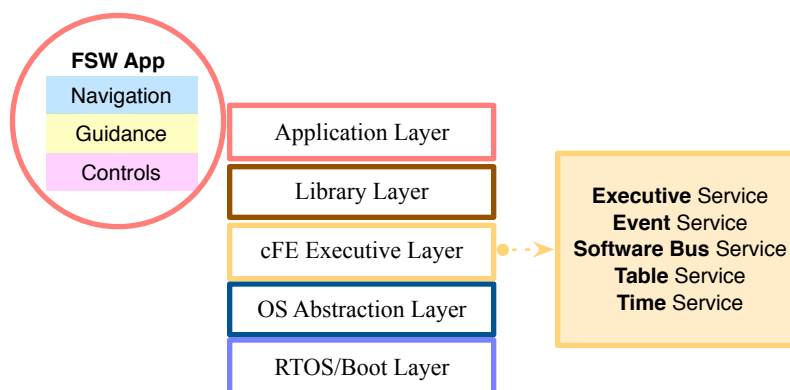


Figure 4.1: Architecture of the core Flight System

## 4.2 From Basilisk into a cFS-FSW Application

This section explains what it takes to migrate Basilisk-developed flight algorithms into a cFS application that is embeddable. Recall that Basilisk is written in a combination of Python, C and C++ whereas cFS only supports pure-C applications. In particular, Basilisk uses Python for:

- (1) Setup of the underlying C/C++ modules. It involves two items:
  - (a) Variable initialization of each individual C module.
  - (b) Grouping of modules in tasks that run at certain task rates.
- (2) Desktop execution of the FSW simulation process
- (3) Post-processing of results

There is one of these Python functionalities that needs to be translated into C in order to yield a fully functional C FSW application: the setup code for the C modules. Each setup item (i.e. module variable initialization and grouping of modules in task) is further developed next.

### 4.2.1 Setup Item: C Module Initialization

Each Basilisk C module is a stand-alone model or self-contained piece of logic. In the context of FSW, a module could be: a specific navigation filter, a control law, a torque-to-voltage converter or, simply, a container for static vehicle data (i.e. spacecraft parameters like inertia or center of mass). All Basilisk C modules are characterized for having a C configuration struct and four main methods operating on the defined struct. In functionality, these main methods are common to all modules and they perform: module self-initialization, cross-initialization, update and reset. These generic functions are externally called from Python during desktop execution. Listing 1 shows a snippet of code from a very simple module, the vehicle configuration one.

Listing 1: C module source code (vehicleConfigSource.h)

```

// Configuration struct
typedef struct{
    double ISCPntB_B[9]; // inertia
    double CoM_B[3]; // center of mass
    char outputMsgName[MAX LENGHT] // unique name for the output message
}VehicleConfigStruct;

// Main algorithms
void SelfInit_vehConfig(VehicleConfigStruct *data, ...);
void CrossInit_vehConfig(VehicleConfigStruct *data, ...);
void Update_vehConfig(VehicleConfigStruct *data, ...);
void Reset_vehConfig(VehicleConfigStruct *data, ...);

```

In the desktop environment, SWIG automatically handles the conversion of types from C and C++ into Python and, to date, there has not been any C or C++ variable type that could not be SWIGed (including nested C structures and custom C++ classes). Initializing the C and C++ variables of all the modules in Python is handy because it makes the simulation completely reconfigurable: changing the initialization values from Python does not force recompilation of the C code again. This feature is specially useful to handle Monte-Carlo testing. By decoupling the high-level Python functionality (i.e. configuration and initialization) from the low-level C module implementation (i.e. algorithm source code), the principle of dependency inversion is applied. When following this principle in object-oriented design, the conventional dependency relationships established from high-level and policy-setting functionality to low-level modules are reversed, thus rendering high-level functionality independent of the low-level module implementation details. A snippet of Python code initializing the C vehicle configuration module is shown in Listing 2.

Listing 2: Python setup code (for vehicle configuration module)

```

# Instantiate C config struct as a Python object
self.VehicleConfigObj = VehicleConfigStruct()

# Initialize variables
def SetVehicleConfig(self):
    # Define a unique model tag for the Python object
    self.ModelTag = "veh"

    # Initialize the C struct variables as if they were Python object variables
    self.VehicleConfigObj.ISCPntB_B = [600.0, 0.0, 0.0,
                                         0.0, 600.0, 0.0,
                                         0.0, 0.0, 600.0]

    self.VehicleConfigObj.CoM_B = [0.0, 0.0, 1.0]

    self.VehicleConfigObj.outputMsgName = "adcs_config_data"

    return

```

All the module variables that in the desktop environment are initialized from Python, in the cFS embedded environment must be initialized directly in C. Further, in order to keep consistency in the testing throughout both environments, the values of these variables need to match exactly.

#### 4.2.2 Setup Item: Task Groups and Rates

The other setup item leveraged from Python in the desktop environment is the definition of C/C++ tasks that run at the defined task rates. Any number of modules can be added to a task and calling priorities are also established from Python. In the desktop simulation, Python itself loops through the tasks cyclically and, for each task, calls the update method of all the modules in that task. In the embedded environment, it is desired to maintain the same task groups. Therefore, C calls need to be implemented that, for each task, contain callbacks to all the methods of the modules belonging to that task. These C callbacks should respect the module priority established

in the Python desktop script.

### 4.3 Translation Mechanism: the Auto-Setter

Now that the kind of setup code that needs to be translated from Python to C has been explained, let us describe the interesting part: the translation mechanism. Figure 4.2 illustrates the conversion of the flight application from a Basilisk desktop simulation into a pure-C application that can be readily integrated into cFS. A key remark here is that the flight algorithm source code (FSW Algs in Fig. 4.2) remains unchanged. The pure-C application is conformed by the unmodified flight algorithms plus one additional header and one source file (i.e. (`setup.h` and `setup.c`) containing the setup code written in C. The translation of the setup code from Python to C is handled automatically via an independent script written in Python: the **AutoSetter**. The beauty of the **AutoSetter** is that it is not a black box but, rather, a simple template mapping Python variable types into their C counterparts. The resulting C setup code is minimal and completely human readable. In essence, the workings of the **AutoSetter** rely on Python's introspection capa-

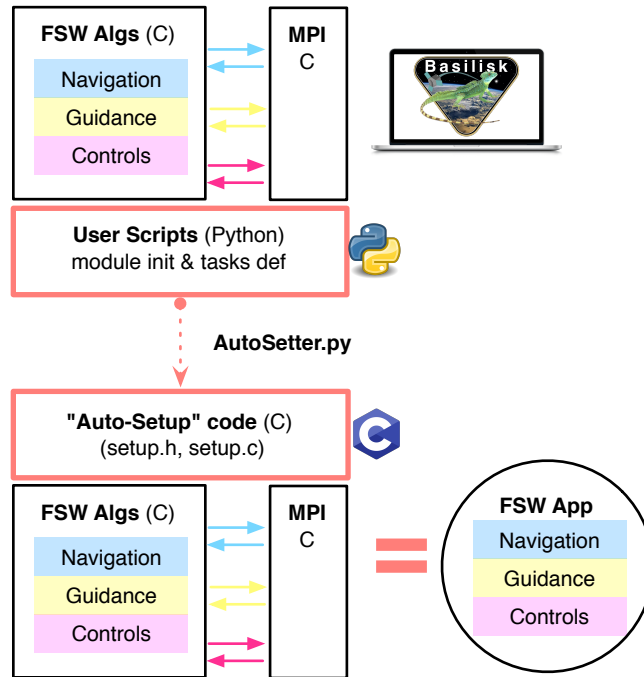


Figure 4.2: Translation of setup code from Python to C

bilities. Looking at oneself is something that neither C or C++ can accomplish without significant investment in source parsing. In contrast, Python can easily realize that, inside the FSW simulation process (written in C but wrapped in Python), there is a list of tasks. And inside each task, there is a list of modules that, despite being written in C, now appear as Python objects. Therefore, these modules now present built-in Python properties like `__module__`, `__name__`, `type()`, `dir()`, `getattr()` and so on, which are the key to introspection. Listing 3 shows a snippet of the C code automatically generated by the `AutoSetter`. Note that this C setup code (output of the `AutoSetter`) corresponds to the Python code shown previously in Listing 2 (input of the `AutoSetter`). Let us take a closer look, for instance, at the inertia variable (`ISCPntB_T` in Listing 3). In Python, the inertia is initialized as a list of 9 floats, with only 3 of them being actually non-zero values; for the `AutoSetter` this unambiguously translates into a C array of 9 doubles, with the same indices filled with non zero values as in the Python list.

Listing 3: Sample of AutoGenerated C Setup Code

```
typedef struct{ // Struct with all FSW modules
    VehicleConfigStruct veh;
    // [...] More modules below
} AllConfig;

void AllConfig_DataInit(AllConfig *data){ // Modules initialization
    memset(data, 0x0, sizeof(AllConfig));
    // VehicleConfig module init
    data->veh.CoM[1] = 1.0;
    data->veh.ISCPntB_B[0] = 600.0;
    data->veh.ISCPntB_B[4] = 600.0;
    data->veh.ISCPntB_B[8] = 600.0;
    strcpy(data->veh.outputMsgName, "adcs_config_data");
    // [...] More modules below
}
```

It is worth clarifying that absolutely no naming convention is imposed on the module variables in order for the **AutoSetter** to find them and parse them appropriately. Developing this tool was a matter of investigating which Python built-in properties would provide the information required to create C code out of the SWIGed modules instantiated and initialized in the existing desktop Python scripts. It was established since the beginning that the **AutoSetter** should not impose any *ad-hoc* restriction nor change in the original scenario scripts and source code. Appendix B contains pseudo-code for the **AutoSetter** (see Listing 5). It is important to notice that, by design, the **AutoSetter** script is specifically linked to the workings of Basilisk. Having said that, it constitutes a general proof of Python's effectiveness in introspection and parsing. Any other FSW testbed that uses Python wrapping C/C++ code could use an equivalent translation mechanism.

As a quick recapitulation, and as illustrated in Fig. 4.3, the unmodified FSW algorithms

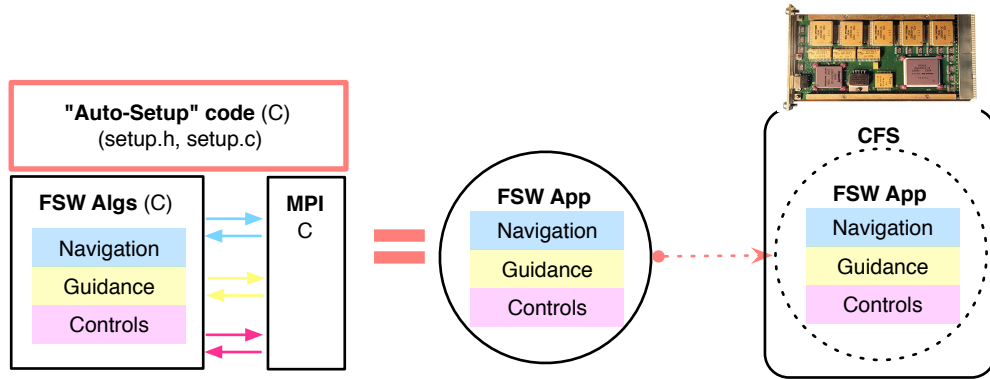


Figure 4.3: Embedded FSW application

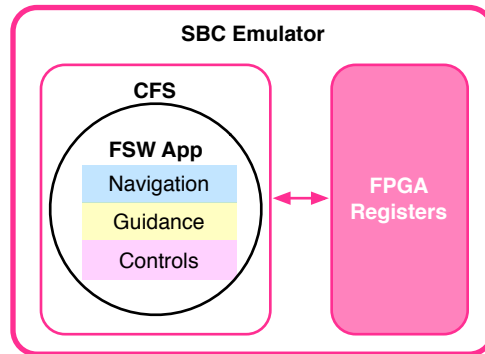


Figure 4.4: Emulated FPGA register space



plus the auto-generated C setup code constitute a cFS application that is embeddable. The embedded cFS-FSW application can be tested in an emulated flat-sat configuration. The concept of emulating a flat-sat configuration for the purposes of integrated testing is depicted earlier in Fig. 1.2. In such case, the cFS-FSW application runs within an SBC emulator, which is responsible for enabling interaction between FSW and the external world. Once FSW is embedded, however, it is no longer that simple to access the FSW states for reading and writing; in contrast to the desktop environment, now there is no longer a flexible Python layer that allows easy interaction with the external world. In order to overcome this challenge and enable communication between the embedded FSW and the external world, FPGA registers have been modeled within the SBC emulator. These registers have been modeled as a memory map for the input and output of raw binary data. The layout of the combined cFS-FSW and modeled registers within the SBC emulator is illustrated in Fig. 4.4.

## 4.4 Summary

This chapter has described the transition of the flight application from Basilisk into the cFS middleware. While Basilisk is a flexible desktop environment useful for prototyping and performing rapid testing of flight algorithms, the cFS is a middleware layer widely use in space applications that ensures portability of the flight application among different targets (flight processor boards and RTOS). The most interesting part of the migration process is the use of the **AutoSetter**, a Python tool that has been developed for the specific purposes of translating the Python portion of the Basilisk flight architecture into C code. The resulting C code, which is minimal and completely human-readable, is generated through Python’s introspection capabilities. Such transition mechanism is generally applicable to any desktop testbed that, similar to Basilisk, leverages the use of Python for wrapping underlying C/C/C++ code. The generated C code plus the original flight algorithm source code are then compiled together into a pure-C cFS FSW application that can be embedded into an emulated processor board. The need of modeling the FPGA registers within the emulated board has also been introduced. Yet, given the complex relationship between the

cFS-FSW states and the register space, this topic is treated separately later on in the manuscript (see Section 5.3). At this point and before providing further details on the modeled register space, it is convenient to step back and introduce the specific flat-sat configuration in which the cFS-FSW is to be tested. This is done in the following chapter.

## Chapter 5

### Emulated Flat-Sat Testing of cFS-FSW through Distributed Communication

The previous chapter has shown the generation of a cFS-FSW application that is actually embeddable. The embedded flight target could be a hardware processor board (for flat-sat testing and eventually flying) or its emulated counterpart (for emulated flat-sat testing). This chapter focuses on the integration of an emulated flat-sat testbed that allows realistic testing of the cFS-FSW application. The advantage of a flat-sat emulation is that it provides pure software substitutions for expensive hardware components of limited quantity that might be needed simultaneously for testing by different mission groups[13, 31, 34]. While embedded testing in an emulated flight processor does not replace actual hardware testing, its use can reduce bottlenecks and alleviate schedule constraints by allowing system-wide testing early on in a mission program's schedule. Further, if flight hardware is emulated in proper high fidelity (which, as it will be shown, is a non-trivial endeavour), the transition from emulated flat-sat testing into hardware flat-sat testing can be smooth and straightforward. An additional challenge associated to the assembly of an emulated flat-sat is the need of dealing with legacy software models that were never design to work together. Integration of independent and heterogeneous mission models into the a single integrated simulation run demands the need for a flexible communication architecture underneath.

With these considerations in mind, this chapter is outlined as follows. First, the legacy software models conforming the emulated flat-sat in which the cFS-FSW application is to be tested are described. Then, the design and implementation of the Black Lion communication architecture is presented as part of the main work in this thesis. Black Lion is designed to support any type of

distributed simulations and the discussion in this chapter covers: functionality, architecture, relevant implementation details and applications. Next, the focus goes back to the specific interaction between the cFS-FSW application and the other components of the emulated flat-sat. This interaction is enabled thanks to the modeling of the FPGA registers and the inclusion of accompanying avionics hardware models; both of which are specially interesting for their implementation challenges. Then, three numerical simulations in the emulated flat-sat are showcased, exemplifying the usage of Black Lion, the emulated registers and the avionic component models. These numerical tests involve:

- (1) Spacecraft pointing commands.
- (2) Mars orbit insertion (MOI) maneuver.
- (3) Fault-detection of coarse sun sensor (CSS) corruptions.

Next, yet another application of Black Lion is showcased: a distributed formation flying simulation. Finally, a summary section remarking the highlights of the chapter is included.

## 5.1 Emulated Flat-Sat

This section presents a specific incarnation of an emulated flat-sat in which the cFS-FSW application is to be tested. Such incarnation is illustrated in Fig. 5.7 and it is being applied to support an interplanetary spacecraft mission. In the configuration of Fig. 5.7, the cFS-FSW application corresponds to the actual onboard executable and it runs on a flight processor emulator which, in turn, interacts with external applications. As shown in Fig. 5.7, the four main components in the emulated flat-sat are the following: flight processor emulator, spacecraft physical models, ground system (GS) model and visualization tool. These components are further explained next:

**Flight-processor emulator: LEON board emulated by QEMU with RTEMS on top.**

The QEMU emulator is used to emulate a LEON-3 board; the RTEMS real time operating system runs on top. The emulated flight processor contains the cFS-FSW application as well as the modeled FPGA registers. Four different registers have been implemented: two

input/output board registers (IOB), the solid state recorder register (SSR) and the single board computer register (SBC). Through the FPGA registers, FSW reads and writes in a hardware-like fashion that also replicates interrupts.

**Spacecraft Models: Basilisk Dynamics Simulation.** Basilisk is used to compute the dynamics and kinematics of the spacecraft in the space environment as well as to simulate onboard hardware components like antennas, sensors (e.g. star tracker, gyroscope, coarse sun sensors...) and actuators (reaction wheels, attitude control thrusters,  $\Delta V$  thrusters). In addition, the spacecraft physical simulation has been expanded with mission-specific

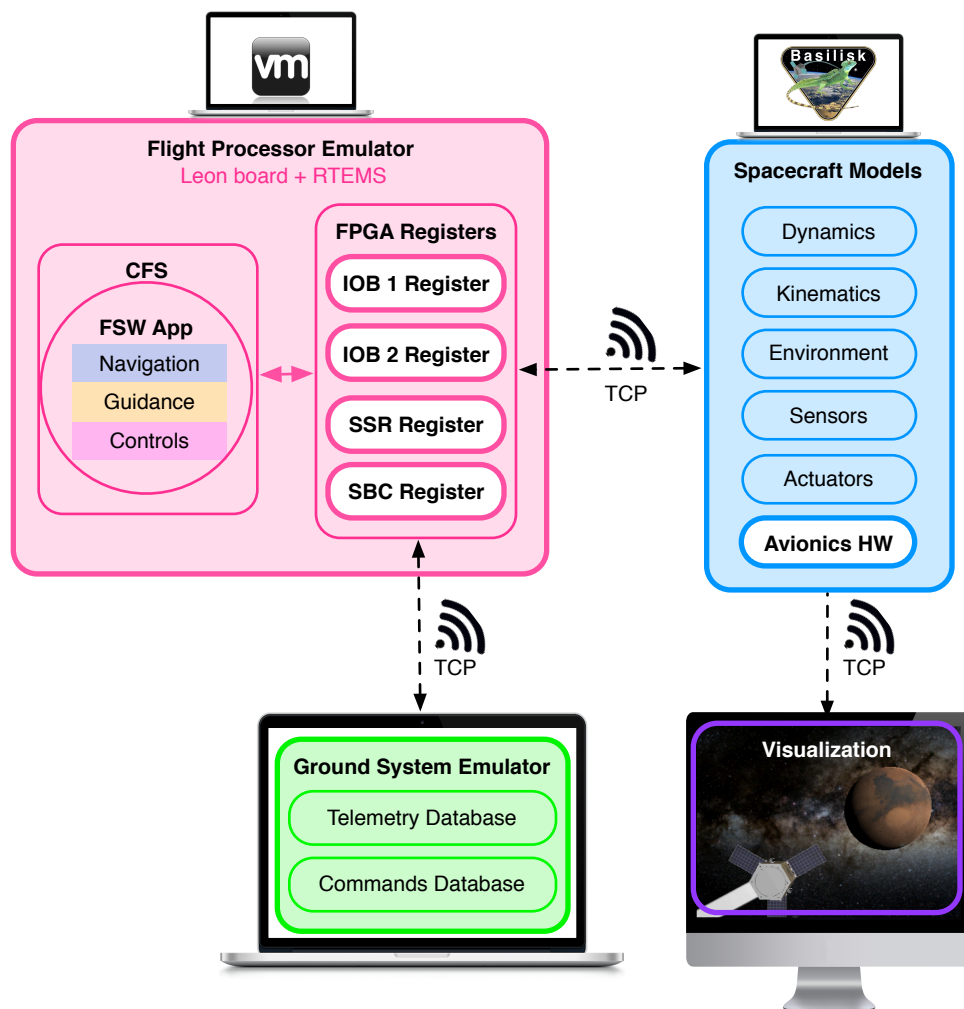


Figure 5.1: Emulated flat-sat components

avionic hardware models that are meant to interact with the FPGA registers.

**Ground System (GS) Emulator: LASP in-house model named Hydra.** It includes the command and telemetry databases for the mission. The GS system provides a graphical interface for the user to manually send commands to FSW as well as to upload tables and command sequences. The graphical interface also allows the user to monitor telemetry provided by FSW. Commands and telemetry come in and out in the form of CCSDS packets.<sup>[11]</sup>

**Visualization: Unity-based interface named Vizard.** It is an open-source graphic tool distributed together with the Basilisk framework and it allows realistic visualization of the spacecraft simulated behavior<sup>[49]</sup>.

Interestingly, all these models conforming the emulated flat-sat are independent and stand-alone applications that were never designed to work together. Therefore, they are inherently heterogeneous: they are written in different programming languages, they have different execution speeds (asynchronous vs. synchronous, faster than real time vs. slower), some are single-threaded applications while others run on multiple threads and they also present different endianness (little endian vs. big endian). The heterogeneity of the models in terms of programming languages and data structures is represented in Fig 5.2.

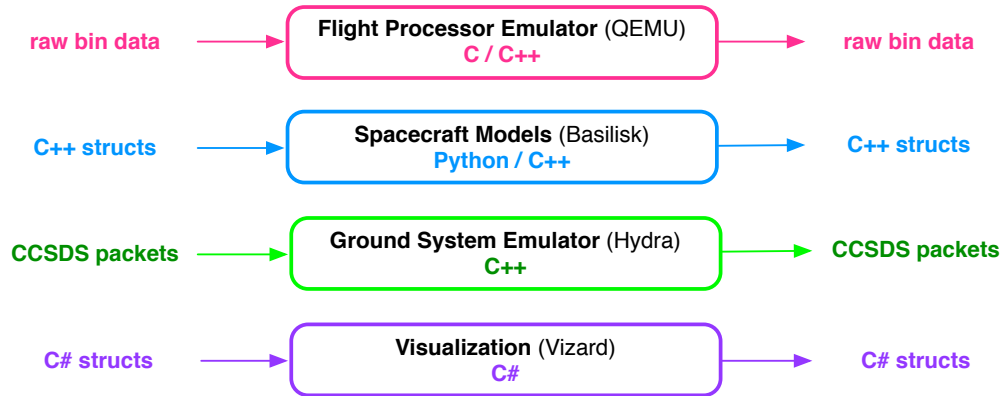


Figure 5.2: Heterogeneity of flat-sat models

Given the unique and distinct nature of each model in the flat-sat, the integration of all the

models into a single simulation run demands for a solid communication architecture underneath. On these lines, Black Lion is presented as a purely software-based distributed communication architecture that has been developed to support the aforementioned interplanetary mission[13]. The Black Lion communication architecture is described in detail in the next section.

## 5.2 The Black Lion Communication Architecture

Black Lion is a pure-software communication framework aimed at distributed testing of independent spacecraft mission models. The framework is architected to be scalable and reconfigurable, allowing for any number of heterogenous software models, across one or multiple computing platforms, to be integrated into a single simulation run. While the development of Black Lion was initially motivated to support an interplanetary spacecraft mission, the system is built under the principles of reusability and scalability and its applications extend beyond the flat-sat use case. As a communication architecture, Black Lion accomplishes four main tasks:

- (1) Transport of binary data between components/nodes.
- (2) Serialization of binary data: each node must know how to convert the received bytes into structures that can then manage internally.
- (3) Synchronization: all the nodes need to be in lock-step during a simulation run.
- (4) Dynamicity in the connections map: nodes shall not static and required components in the network but, rather, dynamic clients able to come and go on the fly.

### 5.2.1 Design and Architecture

The goal of Black Lion is to achieve the described communication tasks while being as abstracted as possible from the internals of each node. A communication layer that is transparent and abstracted from the source code of the nodes/components allows users to plug and play their models of choice. In order to achieve the desired level of abstraction, the Black Lion architecture has been designed as a single central controller and two APIs (Application Programming Interfaces)

that are attached to each node. Such architecture is depicted in Fig. 5.3 and the functionality of each Black Lion interface is defined next:

**Central Controller:** it is the only static piece in the network (i.e. it has a static IP address). It acts as a message broker for data exchange and it also governs synchronization through a “tick-tock” mechanism. The **Central Controller** is written in Python.

**Delegate API:** it is a generic interface that manages sockets and network connections with the **Central Controller**. The same script is attached to all the nodes. The **Delegate** class is currently implemented in Python, C++ and C#.

**Router API:** it is a generic interface class with node-specific callbacks. Its purpose is to route data in and out of the internals of the node. Hence, it is an intermediary between the generic **Delegate** class and the specific node application. The **Router** class is also available in Python, C++ and C#.

In order to exemplify the functionality of the different BL interfaces it is convenient to use a human-

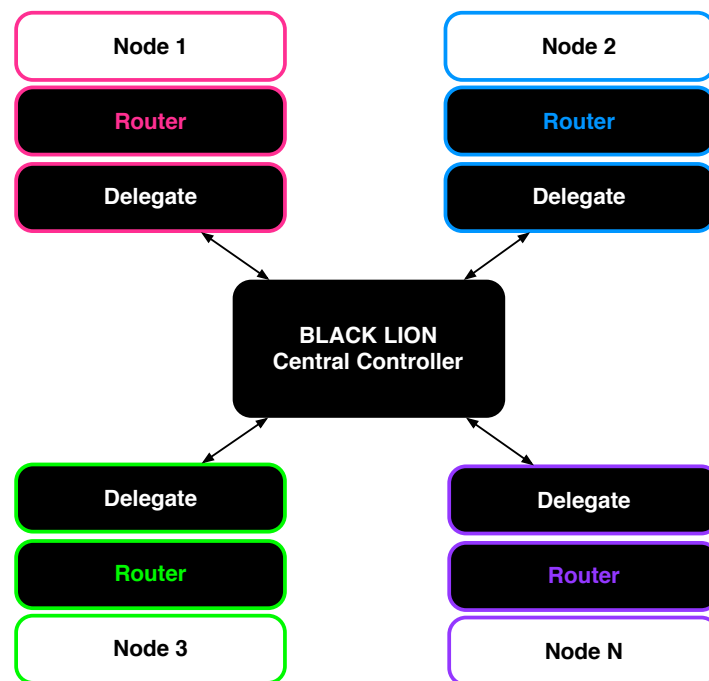


Figure 5.3: Black Lion interfaces: **Central Controller** and, for each node, **Delegate** and **Router** APsd



language analogy. Each node can be seen as an individual that speaks a different language (i.e. Spanish/French/German in the analogy or, in terms of Black Lion, Python/C/C++/C#). The **Router** acts as a translator from the individual’s language to a common standardized language (i.e. metaphorically English, and hexlified byte-string in Black Lion). If the **Router** is the translator, the **Delegate** can be seen as the communicator (i.e. the person who reads the translation out loud or, in Black Lion, the interface who sends out the standardized data through the sockets). The final result is an English conversation in which each individual node does not have to learn the particular language of every other participant in the conversation. This property of the communication architecture is key to its scalability.

### 5.2.2 Data Transfer and Synchronization

Regarding data transfer between applications, Black Lion takes advantage of the ZeroMQ (ZMQ) Message Library<sup>1</sup>. In order to fully understand how the communication hub operates, the reader is pointed to Appendix C, which describes in detail the socket types, connection types and communication commands used in the system.

<sup>1</sup> <http://zeromq.org>

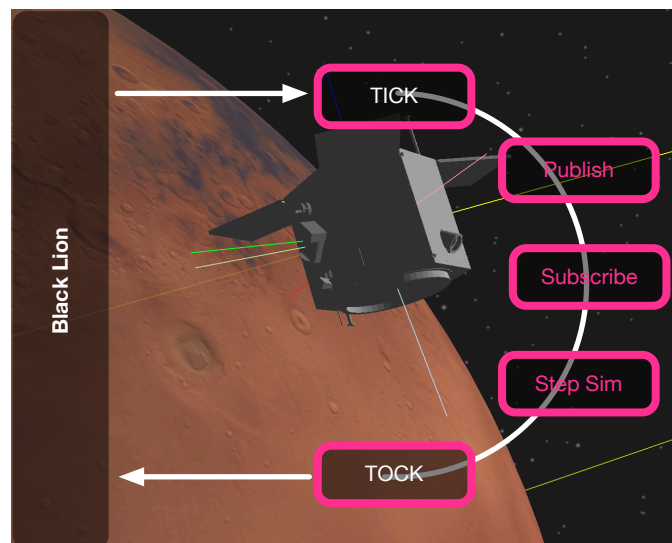


Figure 5.4: “Tick-tock” synchronization

In terms of synchronization, Black Lion uses a “tick-tock” mechanism to maintain all the applications in lock step. The synchronization is based on a request-reply pattern where the **Central Controller** requests a “tick” and the **Delegate** of each node replies a “tock” after the node has performed its duty. Figure 5.4 illustrates the actions that each node performs at each time step: **Publish**, **Subscribe** and **StepSimulation** forward. These three actions are performed sequentially upon receiving a “tick” request. They are further described next:

**Publish:** In the **Publish** internal call, the node’s **Router** collects the application internal data and makes it available to the node’s **Delegate** for publication to the controller’s frontend.

**Subscribe:** In the **Dubscribe** internal call, the node’s **Delegate** receives external data coming from the controller’s backend and passes the data to the node’s **Router**, who is responsible for writing these messages down into the internals of the application.

**Stem Simulation:** In general terms, the **StepSimulation** internal call implies executing the application forward for a  $\Delta t$  period in order to generate new data.  $\Delta t$  is basically a time step and its duration is determined by the **Controller** and sent to the node’s **Delegate** as part of the “tick” message.

Although there are nuances in the precise meaning of **StepSimulation** for nodes that are synchronous (i.e. run in cycles, like FSW or the spacecraft physical simulation) and for nodes that are asynchronous (i.e. that are event-based like the ground system), the “tick-tock” mechanism works seamlessly with either. For asynchronous applications, the Black Lion interfaces simply need to run on a separate thread than the main application.

It is relevant to mention that, during a Black Lion run, the communication commands (like the “tick” request) are sent at once by the **Central Controller** such that all the applications start running in parallel. Because the applications present different execution speeds, there are some that can step forward faster and reply back with a “tock” earlier. The trick here is for the **Central Controller** to wait until it has received the “tock” from all the applications before sending them the following “tick”. In this manner, all the components are kept in lock step and the overall Black

Lion simulation speed is driven by the slowest of the components.

In hybrid simulations that include both software and hardware in the loop, the “tick-tock” synchronization mechanism can be exploited to satisfy real-time hardware constraints. The only limitation of this strategy appears when different applications impose competing, hard time constraints. For instance, if two components had distinct time requirements, the “tick-tock” strategy could only comply with the slowest of them.

### 5.2.3 Applications

Through the described architecture, Black Lion has been able to enable communication between nodes whose heterogeneity spans from: multithreaded vs. single-threaded nodes, asyn-

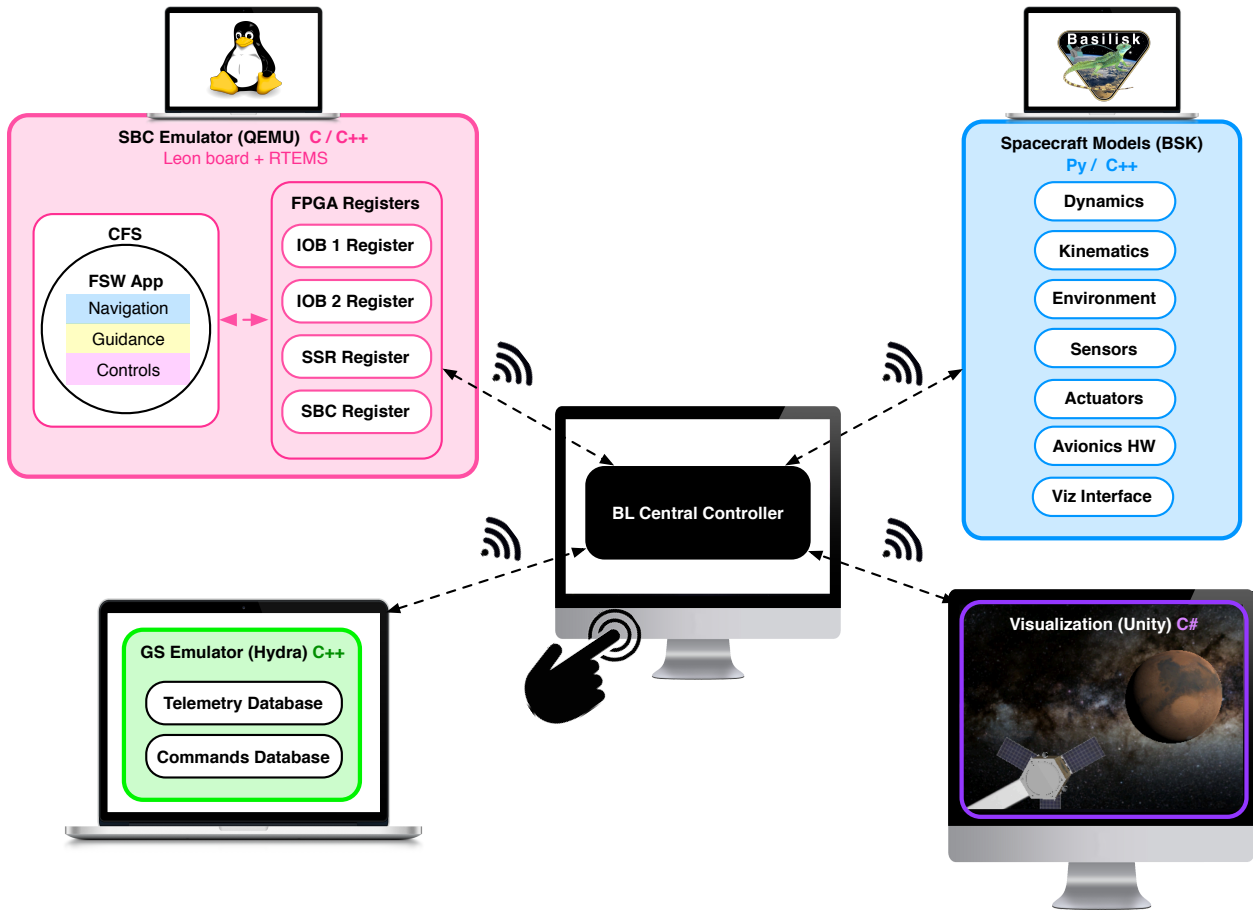


Figure 5.5: Black Lion application: emulated flat-sat

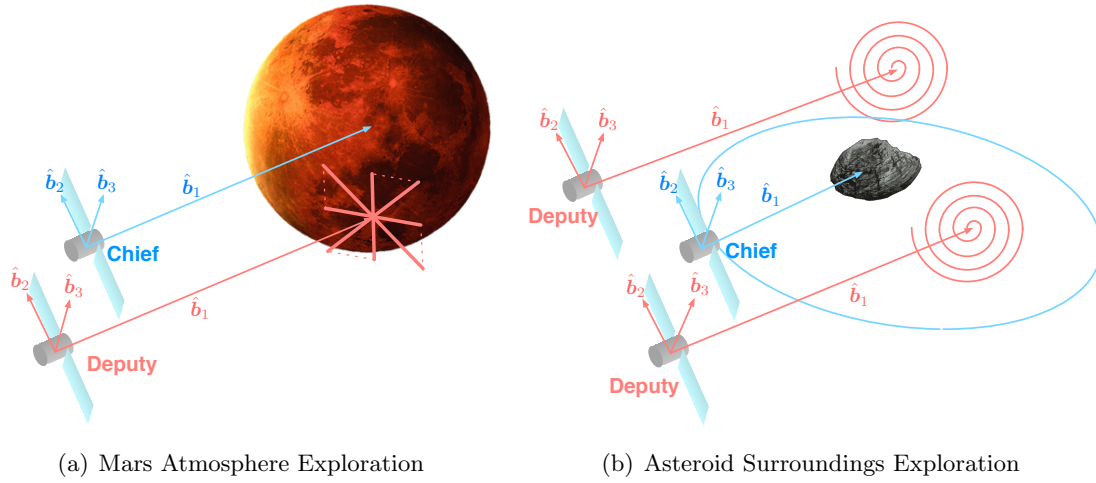


Figure 5.6: Black Lion application: testing of formation flying concepts

chronous vs. synchronous nodes, little-endian vs. big endian nodes, as well as for a variety of programming languages: Python, C, C++ and C#. The applications for Black Lion are diverse and numerous. One of the most interesting ones is the emulated flat sat in Fig. 5.5, which was indeed the driver for the development of Black Lion. Another use case of Black Lion is realistic testing of formation flying concepts. For example, the communication between a chief spacecraft and a deputy spacecraft can be realistically emulated through Black Lion. This idea is depicted in Fig. 5.6 and showcased through numerical simulation in Section 5.5

While Black Lion simulations can be executed in a completely distributed fashion using TCP communication, as in Fig. 5.5 (where each component of the flat sat runs on a separate computing platform), it is granted the all the components can also be executed from the same machine using, for example IPC communication or other local protocols. For the case in which all the Black Lion components run on the same computing platform, a **Bootstrapper** file whose function is to launch all the different components/applications at once has been developed. The **Bootstrapper** file is written in Python and uses the language's **Popen** feature to launch any external executable. This feature is extremely convenient for users to start a single-platform Black Lion simulation with a single click.

Now that the Black Lion communication has been described, let us move forward with the

flat-sat application. Next section picks up from earlier and deepens into the interaction between FSW and the external world through the FPGA registers, which store the FSW data that will then shipped across Black Lion.

### 5.3 FPGA registers and Avionics Hardware Modeling

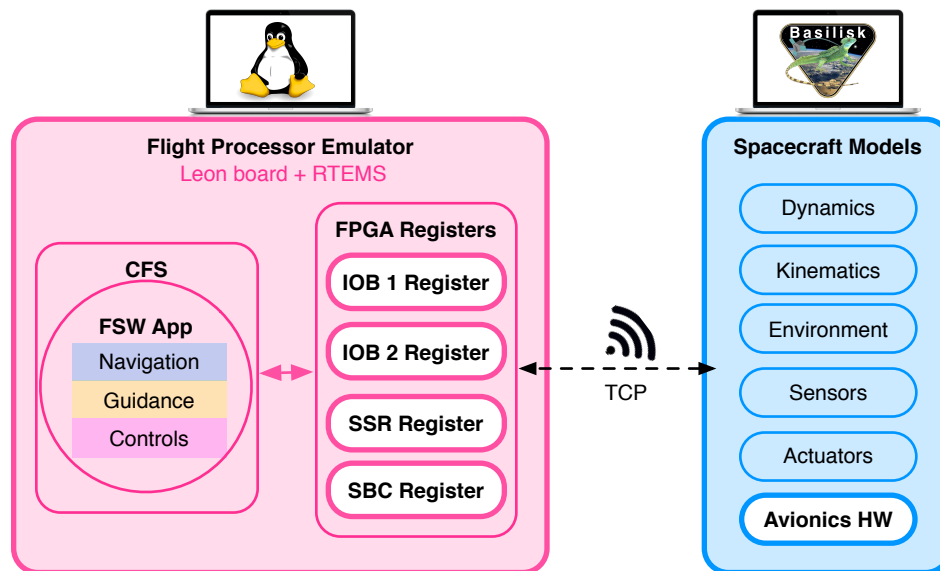


Figure 5.7: cFS-FSW interaction: emulated FPGA registers and avionics hardware models

Figure 5.7 illustrates a subset of components in the emulated flat-sat. The underlying Black Lion architecture is now simply represented as a TCP connection. The white boxes in Fig. 5.7 highlight the specific parts that have been developed for the purpose of enabling reading and writing of FSW states: FPGA registers (modeled within the processor board emulator) and accompanying avionics hardware (modeled within the spacecraft physical simulation). Through the different registers, FSW interacts with the external world by reading and writing in a hardware-like fashion that also replicates interrupts. For instance, FSW receives commands and returns telemetry from and to the GS emulator (by means of CCSDS packets that are stored directly into the registers), it commands the actuators in the spacecraft simulation and it also receives simulated sensor data through the registers. In turn, the avionics hardware models in the spacecraft physical simulation leverage complex functionality that would otherwise have to be implemented within the registers

as well.

### 5.3.1 Register Space

As mentioned earlier, four different registers have been implemented: two input/output board registers (IOB), the solid state recorder register (SSR) and the single board computer register (SBC). The idea is that each register has an associated memory buffer, and specific FSW states are mapped to specific addresses within these buffers. Not all the internal FSW states are mapped to the register's buffers, but only those that require interaction with the external world. These shared states are referred to as snorkels, in the sense that they are direct connection pipes to the internals of the cFS-FSW application. The different snorkels that have been implemented within the register space, as well as the specific connection of these snorkels to the external world, are illustrated in Fig. 5.8.

In terms of implementation, the challenge is that all these snorkels are intrinsically different from each other: some are unidirectional (they could be either reader or writers with respect to FSW) while other are bidirectional (they have both a reader and a writer associated); some operate by packet address while others require both packet and descriptor addresses (the descriptor being a separate word that describes something important about the packet); some snorkels store single-word packets while others queue them; some packets contain a fixed number of bytes while others present variable size; a few snorkels need to add or remove header bytes before providing/retrieving the packets to/from FSW; and most of the snorkels are required to handle endianness, which is a non-trivial task. Because big endian ordering has the most significant byte in the lowest address while little endian has the most significant byte at the highest byte address, sharing information between different machines can be complicated. It is necessary to know the size of the data being transferred as well as the endianness of the source and target machines. Otherwise shared information may appear to be incorrect due to data conversions being done incorrectly.

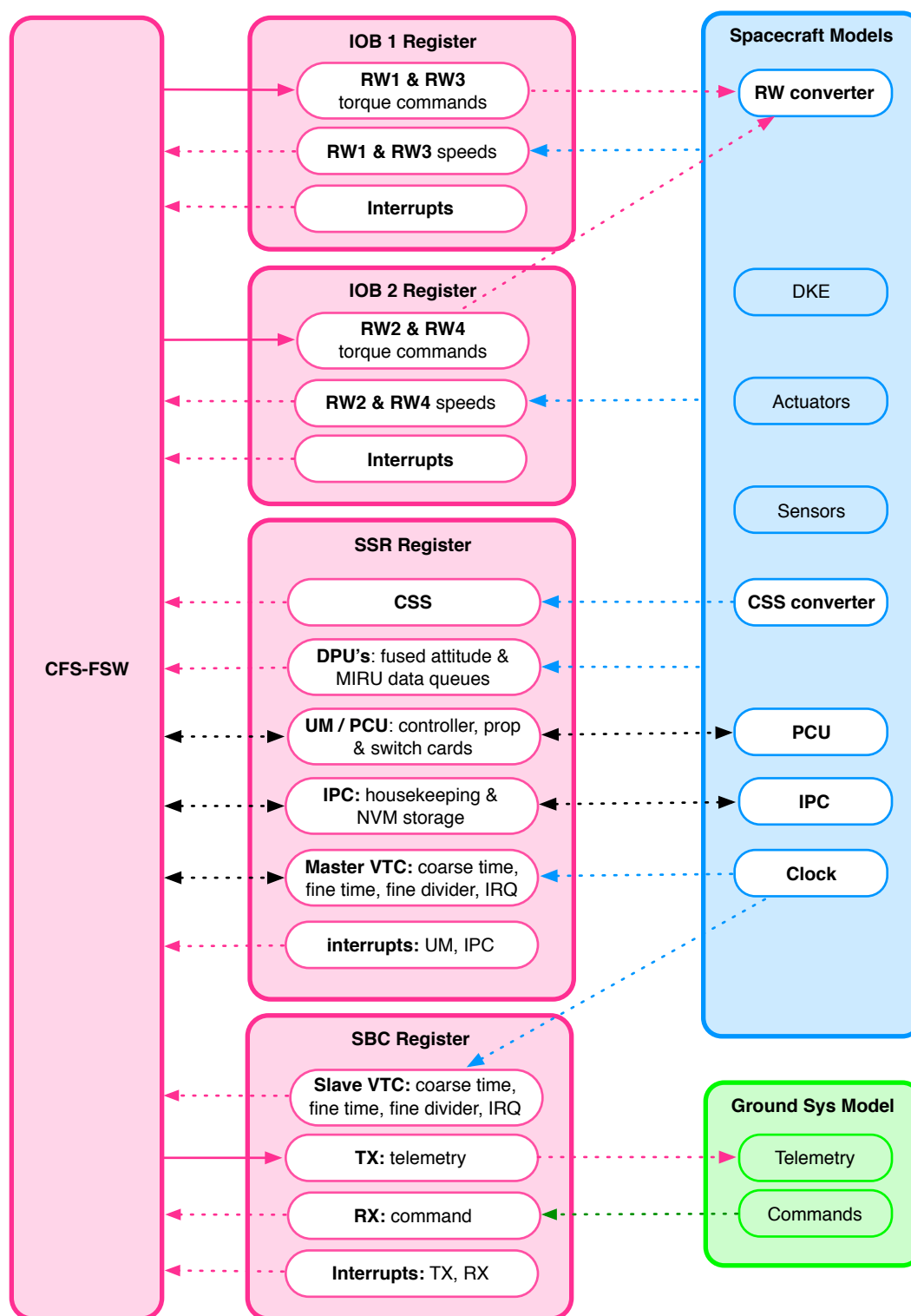


Figure 5.8: Detailed view of the emulated FPGA registers and avionics hardware models

### 5.3.2 Avionic Models

Regarding the avionics hardware models that accompany the registers' snorkels, the following models (shown in Fig. 5.8) have been implemented: a reaction wheel (RW) converter, a coarse sun sensor (CSS) converter, a clock model, the power computing unit (PCU) and the interconnect peripheral component (IPC). The PCU and IPC are particularly complex models responsible of managing avionics cards (i.e. controller, propulsion and switch cards), storing and retrieving non-volatile-memory commands, and producing house-keeping packets. While the specific snorkels and avionic hardware models described here are mission-specific, the register space with its readers and writers constitutes a generic framework that can be applied to any FSW application. The use of this same register space into a different FSW application (which is not cFS based) is shown in later chapters of this manuscript.

The next section presents several numerical simulations exemplifying the use of the emulated registers and avionic models in order to test the closed-loop behavior of the cFS-FSW application. In these simulations, FSW interacts with the other flat-sat components (i.e. GS model, spacecraft physical simulation and visualization) through the distributed Black Lion communication architecture.

## 5.4 Emulated Flat-Sat Simulations

This section showcases three different numerical simulations corresponding to different tests on the emulated flat-sat. The first numerical simulation consists on performing a series of spacecraft pointing maneuvers by sending individual commands from the ground system model. This simulation serves to show the core functionality of the modeled FPGA registers on enabling communication between FSW and the external world for the purposes of closed-loop testing. The second numerical simulation consists on performing a Mars orbit insertion by uplinking the corresponding block-command sequence. This simulation is particularly interesting for its complexity and reliance on multiple avionic hardware models. The third and last numerical simulation is a fault-detection



test which has been designed and implemented by the author in support to the aforementioned spacecraft mission. In this test, the Black Lion communication architecture is exploited to trigger a fault on the simulated coarse sun sensors in the middle of a distributed run. Then, the ground system model is used to monitor FSW telemetry and check for correct detection of the injected fault.

#### 5.4.1 Spacecraft Pointing Commands

The first numerical simulation consists on commanding the spacecraft into a series of pointing maneuvers. In this case, the user utilizes the ground system interface to manually send commands to FSW as well as to monitor the reported telemetry. The different commands issued by the user are the following:

- (1) Nav monitoring and inertial pointing command
- (2) Ephemeris correlation and Mars pointing command

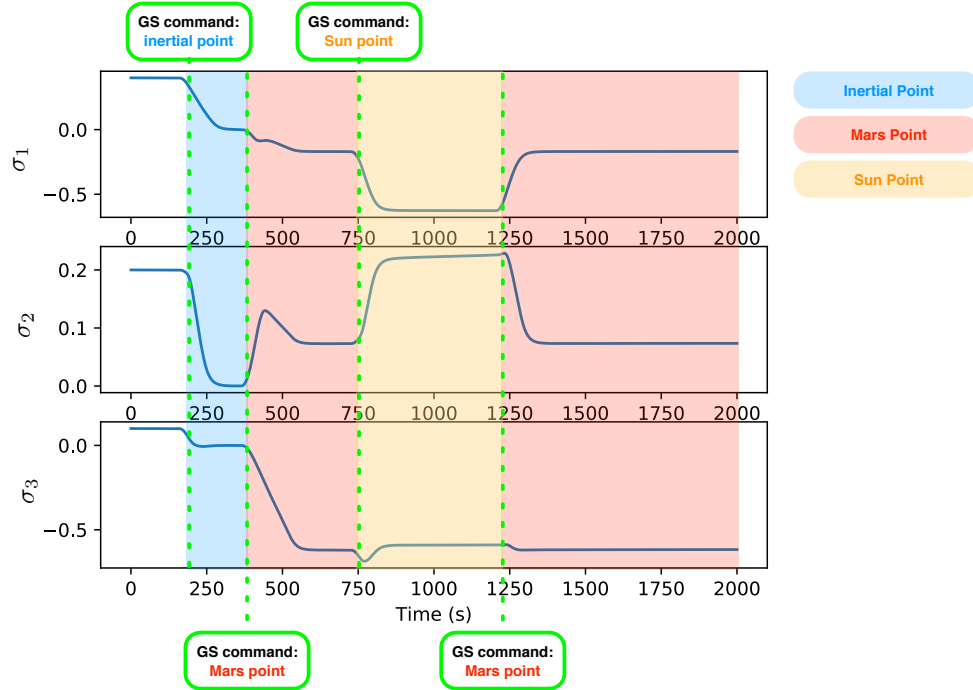


Figure 5.9: Closed-loop response: spacecraft's main body attitude

- (3) Sun pointing command
- (4) Mars pointing command

These commands are stored in the FPGA registers and picked up by the cFS-FSW application in order to reconfigure the onboard pointing mode. In the meanwhile, sensor data from the spacecraft physical simulation is being continuously updated within the registers. Once the nav monitoring command is received, FSW starts using the sensor data as an input to the navigation filters in order to achieve attitude lock. The user is able to monitor the ADC (Attitude Dynamics and Controls) packets in the telemetry stream through the GS interface; hence, keeping track of

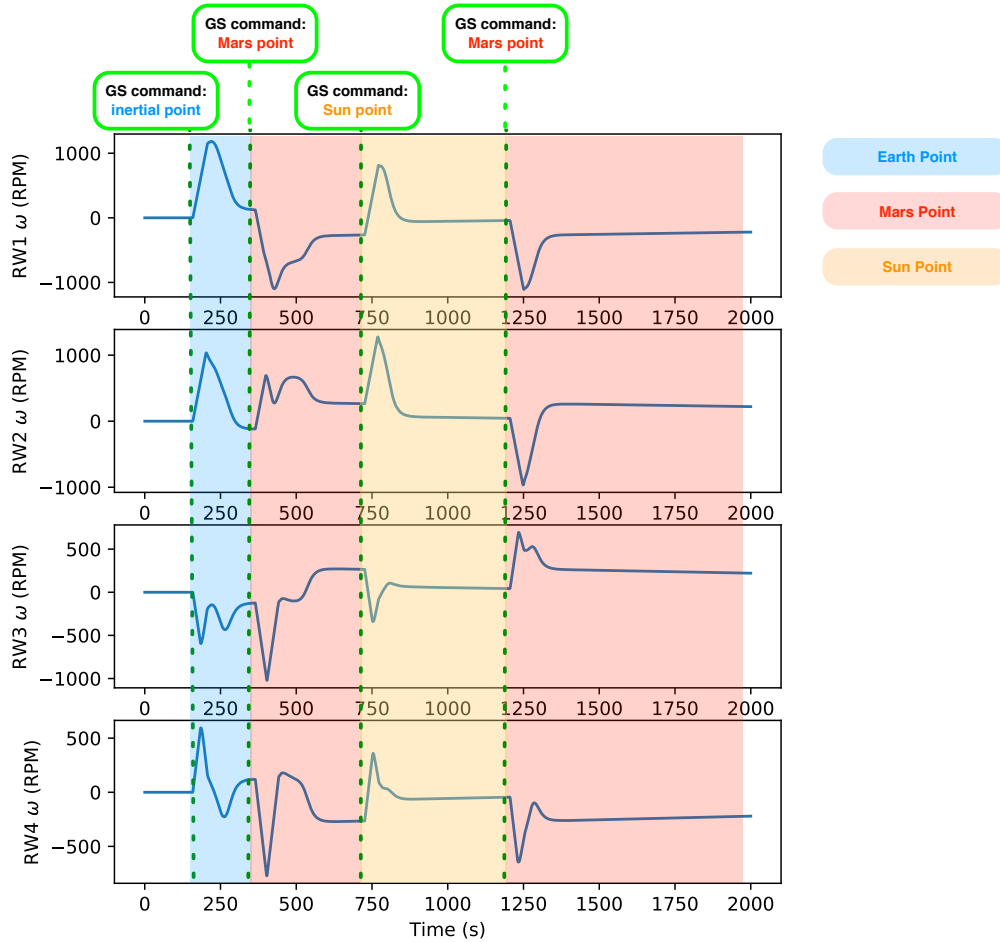


Figure 5.10: Closed-loop response: reaction wheel speeds

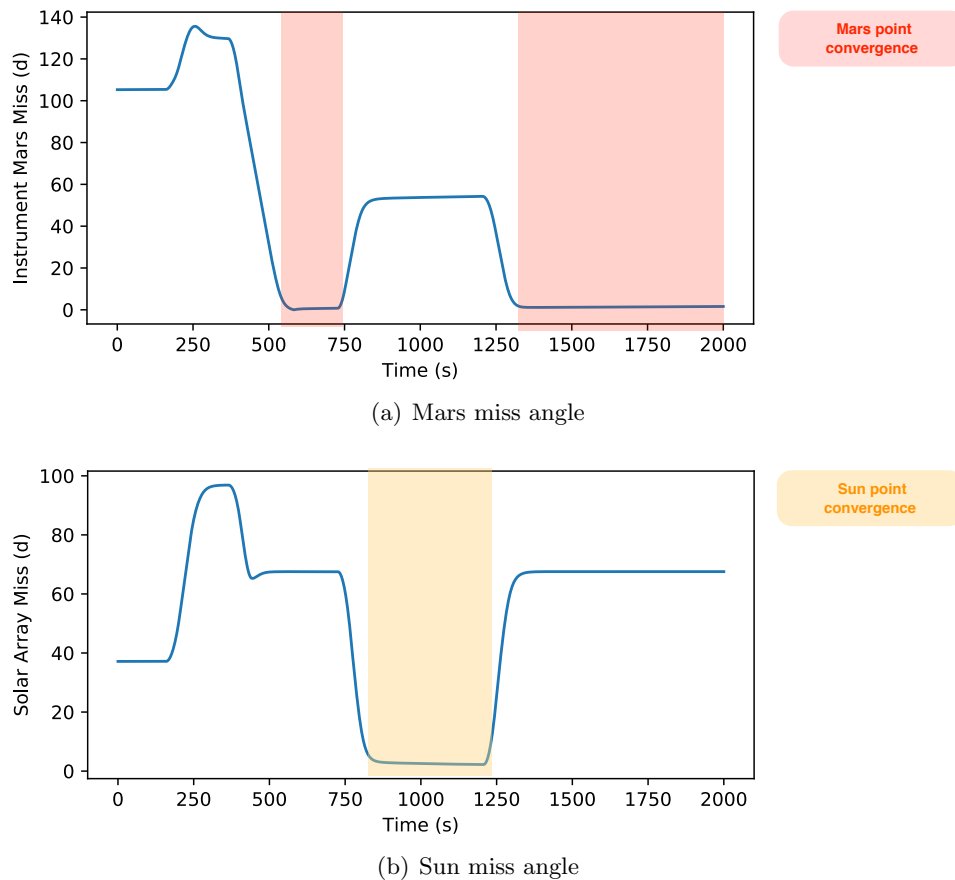


Figure 5.11: Closed-loop response: instruments pointing

the current FSW attitude states. Once attitude lock is achieved and a pointing command (e.g. inertial pointing) is issued, FSW estimates the spacecraft's current pointing attitude, derives the associated tracking errors and computes the control torques required to drive the spacecraft into the desired attitude. The control torques are stored in the registers and sent across Black Lion to the spacecraft physical simulation, where a set of four reaction wheels is used to apply the commanded control torque.

Figure 5.9, Fig. 5.10 and Fig. 5.11 show the closed-loop response of the spacecraft physical simulation. In particular, the plots in Fig. 5.9 correspond to the MRP attitude of the spacecraft's main body frame; the plots in Fig. 5.10 show the reaction wheel speeds driven by the control voltage commanded by FSW; and Fig. 5.11 displays the miss angle (in degrees) of the onboard Mars

instrument and solar arrays. These plots help testing not only the flight algorithm performance but also the validity of the GS commands. In addition, they constitute a proof of adequate register modeling and exchange of data across Black Lion.

### 5.4.2 Mars Orbit Insertion

The second integrated simulation involves a Mars orbit insertion (MOI) scenario. Validating the FSW performance in such complex and long scenario is not easily done by simply looking at plots of individual parameters. In this case, the visualization tool that is integrated as part of the emulated flat-sat turns extremely handy. A visual movie of the spacecraft can be watched live stream during a simulation run.<sup>2</sup> In addition, the tool also offers offline playback capability. Figure. 5.12 shows a series of screenshots from the visualization movie of an MOI test.

Behind the scenes of the MOI run, the register snorkels and avionics hardware models play a crucial role. In addition, another component is integrated into the emulated flat-sat: the CFDP (CCSDS File Delivery Protocol) node. Through CFDP, data files like command sequences can be uplinked to FSW in a flight-like manner. The addition of the CFDP node as a transition point between the GS model and the embedded FSW application is illustrated in Fig. 5.13.

In the MOI scenario, one of the most important models is the power computing unit (PCU), responsible of managing the propulsion cards within the Solid State Recorder register of the FPGA, in order to trigger the  $\Delta V$  and ACS thrusters to fire. As depicted earlier in Fig. 5.8, the PCU model has been implemented in two pieces: a PCU snorkel within the processor board emulator and an accompanying PCU avionic model within the spacecraft physical simulation. The snorkel and avionic model communicate through each other across Black Lion in a bidirectional fashion. The PCU avionic model initializes the bytes within the snorkel's cards. These cards are a controller card, two propulsion cards and four switch cards. In turn, when FSW commands the thrusters to fire, it does so by writing burn times in certain addresses of the register's propulsion cards. These burn times are shipped across Black Lion and received by the PCU avionic model on the spacecraft

---

<sup>2</sup> <https://hanspeterschaub.info/basilisk/Vizard/Vizard.html>

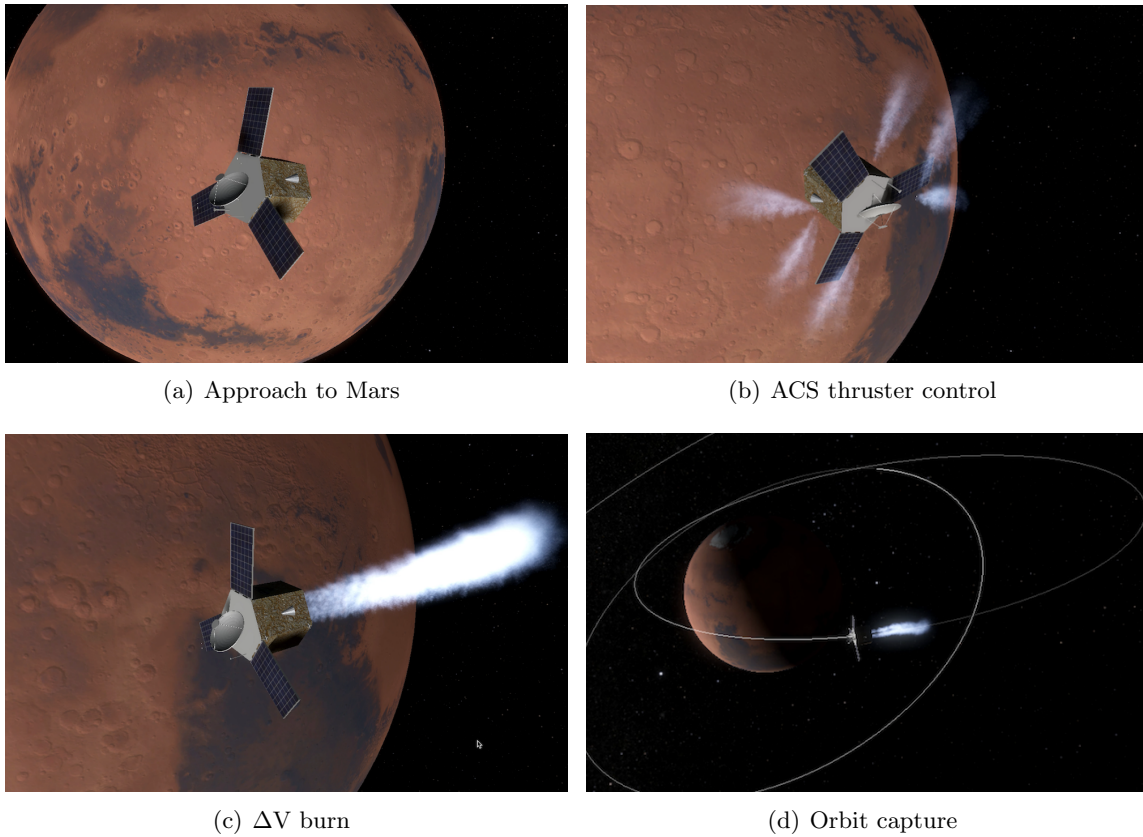


Figure 5.12: Mars orbit insertion scenario

simulation side. The avionics model transforms the received burn times into commands that the thrusters modelled within Basilisk understand. The thruster activity within the spacecraft physical simulation is observed in Fig 5.12(b) and Fig 5.12(c).

Another interesting pair of register snorkel plus avionics model, which is also used in MOI, is the spacecraft clock. In Fig. 5.8, the clock snorkel is labeled as VTC, which stands for vehicle time clock, and its function is to provide accurate time data to the onboard flight algorithms. The use of the clock interfaces allows accounting for times when FSW is asleep but the spacecraft physical simulation has to keep running. In general, it has always been a challenge to use flight software simulators when dealing with simulations that need to account for times when FSW is asleep. For instance, a rover on Mars goes to sleep at night to conserve energy, but the environment keeps going (temperature goes down, wind keeps blowing, etc.). The clock model implemented as

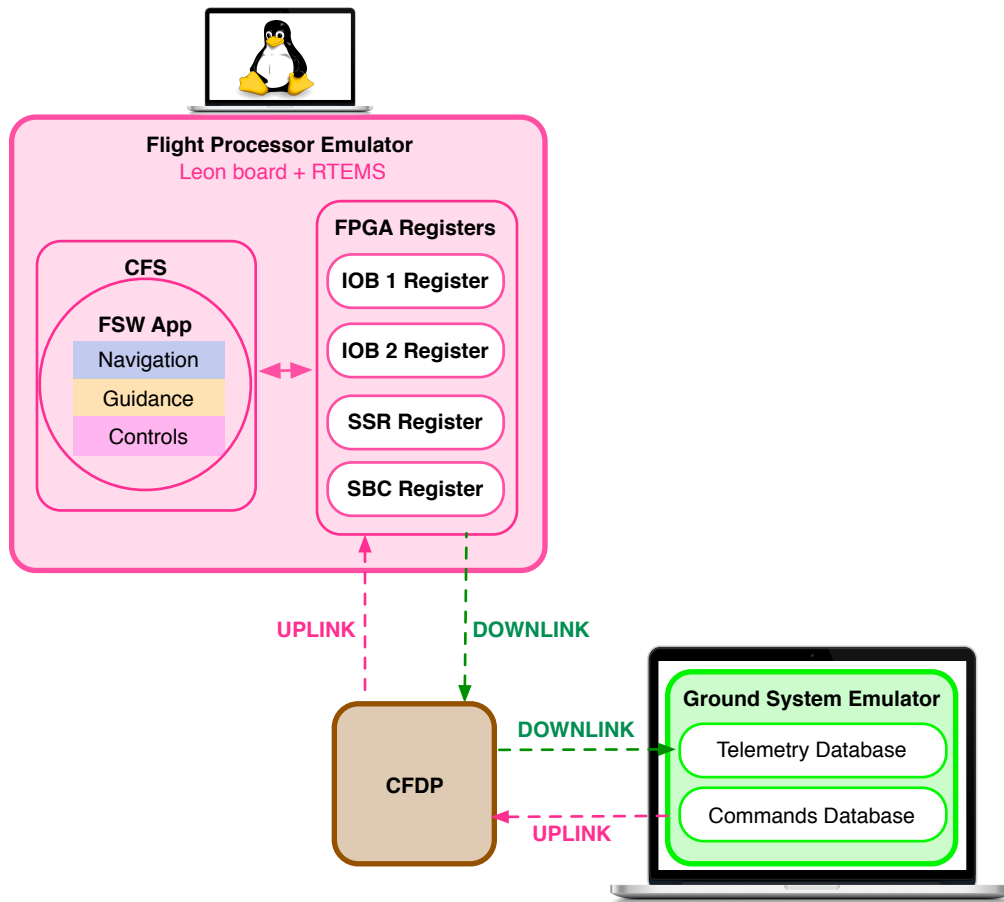


Figure 5.13: Addition of CFDP node for realistic uplink and downlink of data

half snorkel and half avionics component (as in Fig. 5.8) can gracefully handle this situation by allowing FSW time jams. The idea is that, when FSW wakes up, the flight clock needs to catch up and be synchronized with the current virtual time in the spacecraft physical simulation. In the MOI scenario of Fig. 5.12, the FSW clock time is jammed for the purpose of transitioning from the cruise period up into the time when MOI is scheduled. Integrating the capability of jamming the spacecraft clock time was more challenging than it seems at first sight because of the complex interaction between FSW and its time providers. Two different snorkels had to be implemented: a master VTC and a slave VTC. Both of them had to be separate snorkels yet in synch all the time. Some of the data provided by these snorkels includes coarse time, fine time and hardware-interrupt status among other. As a matter of fact, for most snorkels to work properly, it is necessary to

replicate and trigger hardware interrupts at the right times –the aim is to make FSW believe it is running on the hardware flight processor rather than on its emulated counterpart.

An additional capability integrated into the emulated flat-sat in the context of MOI testing has to do with FSW resets. The MOI block-command sequences uplinked to FSW actually contain FSW resets, as it will happen in flight. Rather than modifying those sequences, it was desired to add the capability of simulating FSW resets within the emulated flat-sat as well. In order to achieve this, an IPC snorkel and an IPC avionic model were implemented within the FPGA registers and the spacecraft simulation respectively. The IPC avionic model is responsible of 1) producing housekeeping packets and 2) storing non-volatile-memory (NVM) data. In turn, the IPC snorkel is responsible of 1) arranging the NVM data that FSW writes into complete packets and 2) requesting either the storage or the retrieval of the NVM packets to the IPC avionic model.

#### **5.4.3 Coarse Sun Sensors Corruption/Miscompare**

The last numerical simulation is particularly interesting because it represents a test done in support to the Fault Detection and Protection (FDP) group at LASP for the aforementioned interplanetary spacecraft mission. This test contemplates a mismatch between the data provided by the two pyramids of coarse sun sensors (CSS) onboard the spacecraft. The nature of the fault makes it impossible to run the test on the actual spacecraft hardware; the only viable way to test the FSW response in presence of a CSS corruption is by means of the emulated flat-sat.

Six different faulted cases are considered:

- (1) All sensors in one pyramid rail high.
- (2) All sensors in one pyramid rail low.
- (3) All sensors in one pyramid babble.
- (4) One sensor in one pyramid rails high.
- (5) One sensor in one pyramid rails low.
- (6) One sensor in one pyramid babbles.

These cases were assessed separately for each of the two pyramids; hence, conforming a total of 12 different tests.

Figure 5.14 shows the introduction of a fault into all the CSS in pyramid 1. Pyramid 1 is shown in blue and pyramid 2 in red. To be more precise, the plotted signals correspond to the output of the analog-to-digital converter for each CSS. In the case shown in Fig. 5.14, all the sensors in pyramid 1 start babbling. The fault is introduced into the Basilisk sensors in the middle of the simulation through Black Lion. The fault is triggered after 200 seconds of nominal operation, shortly after the spacecraft has stabilized into safe mode. In this simulation, the GS interface is

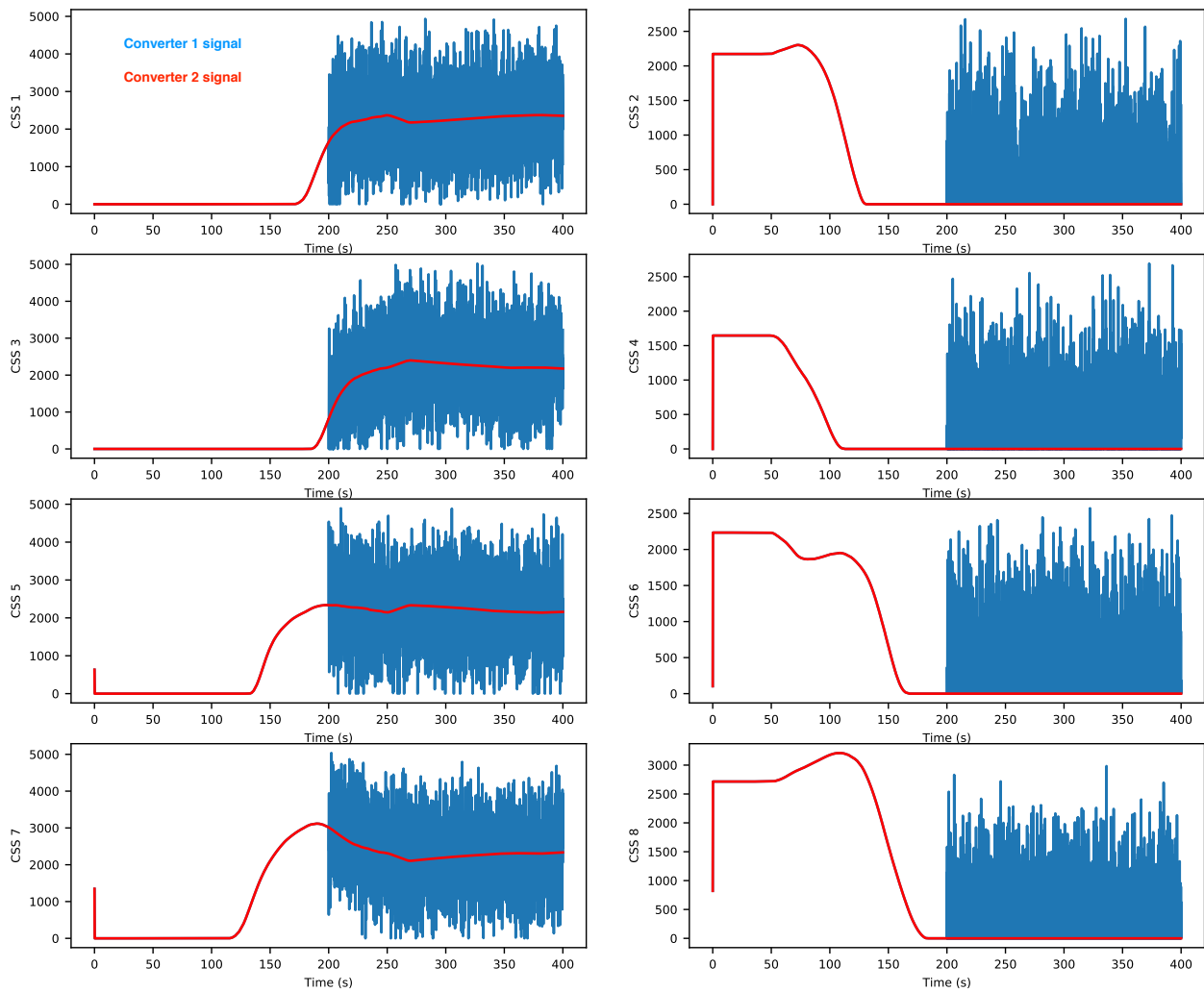
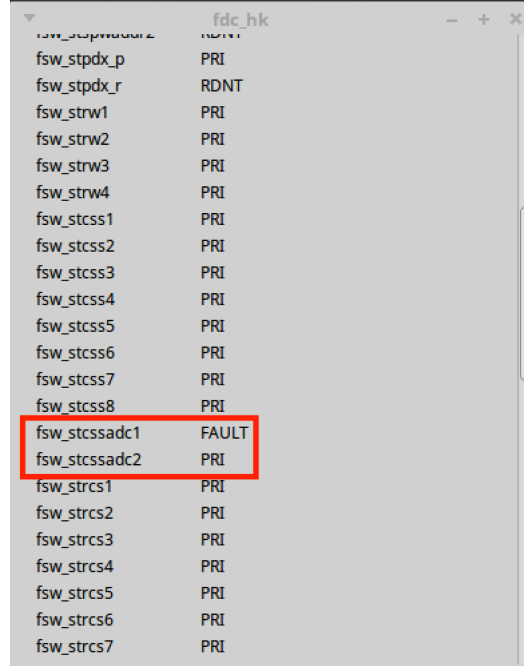


Figure 5.14: CSS analog-to-digital converters: signal miscompare





Packet Name	Status
fsw_stpdx_p	PRI
fsw_stpdx_r	RDNT
fsw_strw1	PRI
fsw_strw2	PRI
fsw_strw3	PRI
fsw_strw4	PRI
fsw_stcss1	PRI
fsw_stcss2	PRI
fsw_stcss3	PRI
fsw_stcss4	PRI
fsw_stcss5	PRI
fsw_stcss6	PRI
fsw_stcss7	PRI
fsw_stcss8	PRI
<b>fsw_stcssadc1</b>	<b>FAULT</b>
<b>fsw_stcssadc2</b>	<b>PRI</b>
fsw_strcs1	PRI
fsw_strcs2	PRI
fsw_strcs3	PRI
fsw_strcs4	PRI
fsw_strcs5	PRI
fsw_strcs6	PRI
fsw_strcs7	PRI

Figure 5.15: FDP telemetry packets: faulted converter 1 and primary converter 2

used to monitor the FDP telemetry packets provided by FSW. These packets reveal that FSW has detected a miscomparision between the signal of the two converters. In addition the packets also point to the converter that FSW believes is faulted. Figure 5.15 is a screenshot of the GS interface showing the FDP telemetry packets for the faulted case in Fig. 5.14. Note that converter 1 is correctly identified as the one being faulted.

## 5.5 Formation Flying Simulation through Black Lion

This section extends the concept of modular attitude guidance on a single spacecraft (presented earlier in Chapter 2) into distributed guidance across a constellation of satellites[12]. Such extension is achieved by means of the Black Lion communication architecture. Without loss in generality, two spacecraft are considered: a chief spacecraft and a deputy. The chief computes a nadir-pointing reference and, through Black Lion, it communicates this time-varying base reference to the deputy spacecraft, which aligns with it and superimposes a relative dynamic motion on top. The final guidance reference of the deputy is a scanning pattern on the surroundings of the

chief's pointing target. To make it even more interesting, the numerical simulation presented in this section includes a Raspberry Pi in the loop; the flight algorithms of the deputy run separately on the Raspberry Pi hardware.

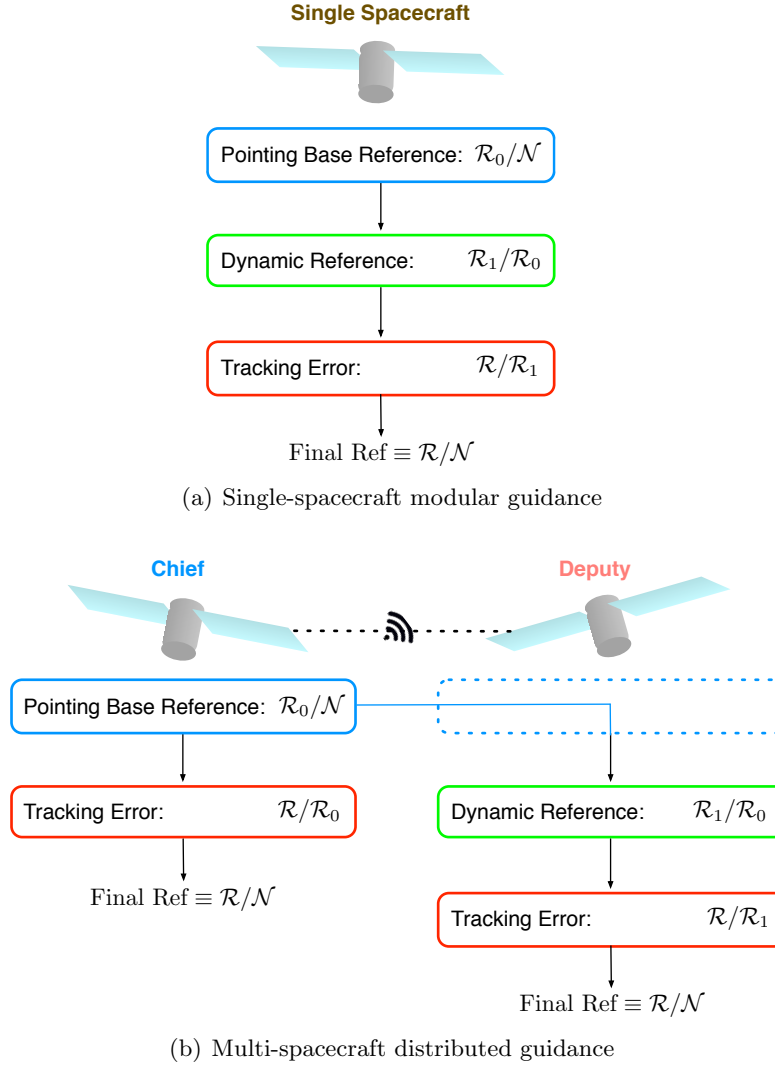


Figure 5.16: Attitude guidance reference generation: modular vs. distributed

Figure 5.16 illustrates the concept of extending modular attitude guidance maneuvers into their distributed counterpart. With the distributed scheme in Fig. 5.16(b), the deputy spacecraft has constrained autonomy in the sense that it relies on the base reference generated by the chief but not on the ground, while the chief spacecraft is completely autonomous (i.e. it builds its final

reference onboard).

Figure 5.17 shows the concept of operations as well as the setup of the distributed simulation. The idea behind the concept of operations is the following: the guidance reference for the chief spacecraft is a time-varying pointing reference, e.g. pointing towards the center of the orbited celestial body. Note that the pointed celestial object could be any target for which the chief has onboard ephemeris: a planet, moon, asteroid, etc. In turn, the goal of the deputy spacecraft is to scan the surroundings of the chief's target in search of additional science opportunities or events. As the chief keeps orbiting, the base pointing reference will change, and the deputy will realign to match this time-varying base, on top of which the scanning pattern will continue. In the orbit of Fig. 5.17, the base reference is seen in the alignment of both spacecrafts while the relative dynamic motion of the deputy is represented by the asterisk scanning pattern. Of course, the drawing in Fig. 5.17 does not illustrate the time-varying nature of the base pointing reference but it is actually simulated. An interesting aspect of this concept is that the deputy does not require any kind of knowledge about the nature of the target; it simply scans the surrounding area of the chief's pointing reference. Note also that such concept is valid for any relative distribution between chief and deputies, which could certainly be on different orbits.

The setup of the distributed numerical simulation is the following: each spacecraft has a separate FSW process (labeled as "Chief FSW" and "Deputy FSW" in Fig. 5.17) and a spacecraft physical simulation process (labeled as "Chief SC" and "Deputy SC" in Fig. 5.17). For the chief, both processes run on the same computing platform. For the deputy, the physical simulation process runs on a desktop computer while the FSW process runs on a separate commercial processor, the Raspberry Pi. Note that each physical simulation is conformed by dynamic, kinematic and environment ("DKE" in Fig. 5.17) models as well as spacecraft hardware models like sensors and actuators. In turn, each FSW process is conformed by navigation tasks (sensor processing, estimation, etc.), guidance tasks (reference generation, tracking error, etc.) and control tasks (control law, torque mapping, etc).

*A priori*, a distributed guidance maneuver could work in two different modes: it could be

either commanded or sensed. In the commanded case, the deputy receives the base pointing reference from the chief through a communication receiver (i.e. using radio frequency). In the sensed case, the deputy spacecraft uses a relative attitude sensor in order to estimate the current attitude and angular rate of the chief spacecraft. The numerical simulation presented next focuses on the former case: distributed commanded guidance. Figure 5.18 shows a more detailed version of the simulation setup used for the commanded case considered. In Fig. 5.18, the specific connections between FSW and hardware models of the chief and the deputy are specified. Here, the chief FSW computes the Mars pointing reference and transmits this guidance reference to the communication receiver on the deputy. The data received on the deputy hardware model is then transmitted to the deputy guidance algorithms.

The particular guidance modules used by the chief and the deputy to assemble their final references are shown in Fig. 5.19. The chief's final reference is nadir Mars pointing, which can be achieved by means of the Hill-orbit pointing module and the addition of an attitude offset. In turn, the deputy's final reference is an asterisk scanning relative to the chief's pointing, which can be

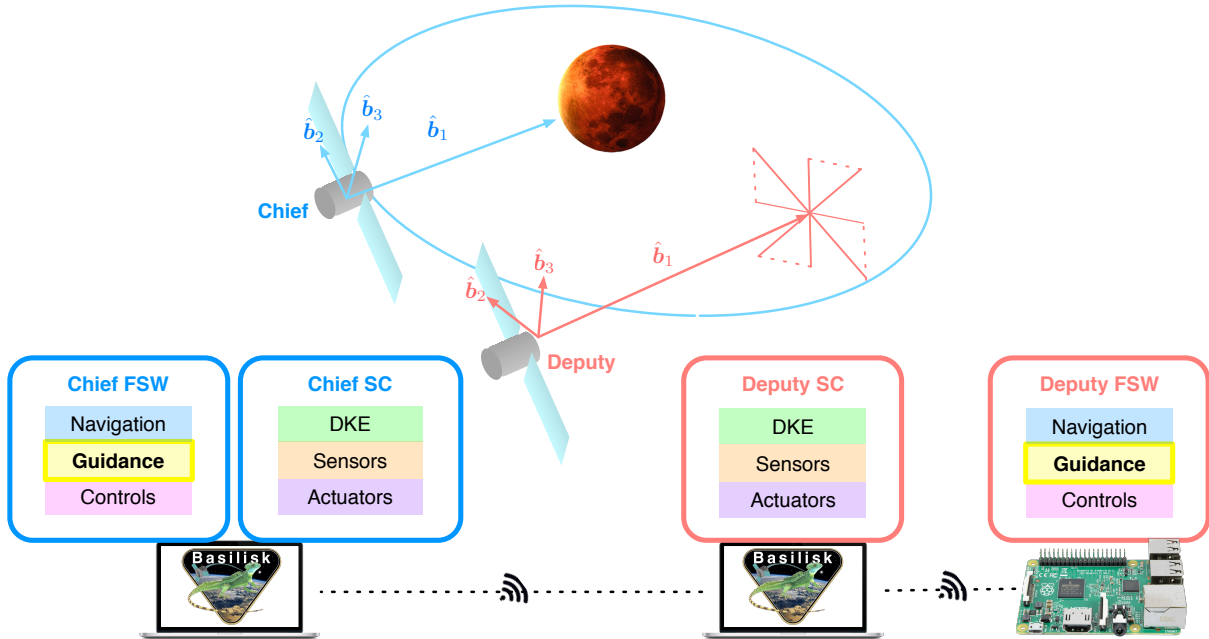


Figure 5.17: Distributed guidance: concept of operations and simulation setup

achieved through four timely commands to the Euler rotation module.

Figure 5.20 show the true attitude states (i.e. computed in the spacecraft physical simulation) of the deputy. The peaks in Fig. 5.20(b) coincide with to command of a new raster. It is observed that, after each command, the angular rates stabilize into a non-zero reference, which reveals the time-varying nature of the Mars pointing base reference. In turn, Fig. 5.21 displays the tracking

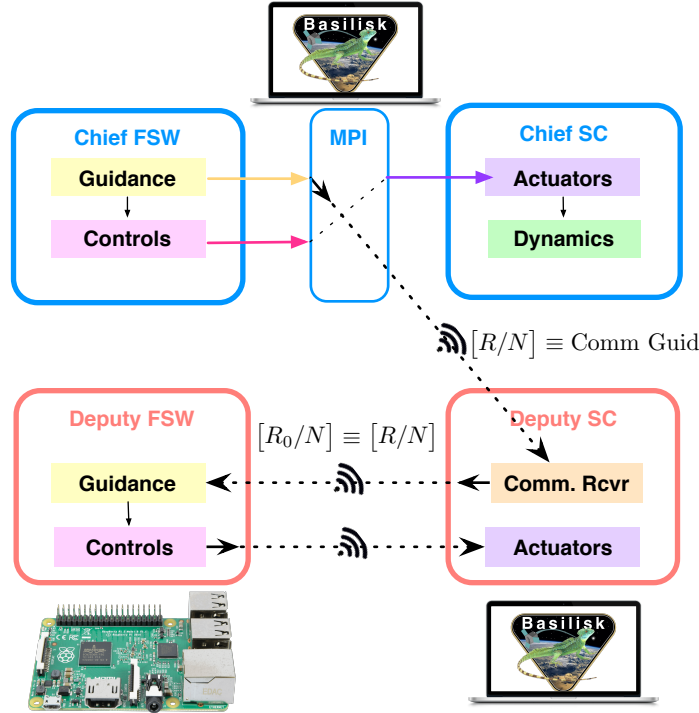


Figure 5.18: Distributed commanded guidance: connections between modules of the chief and the deputy simulations

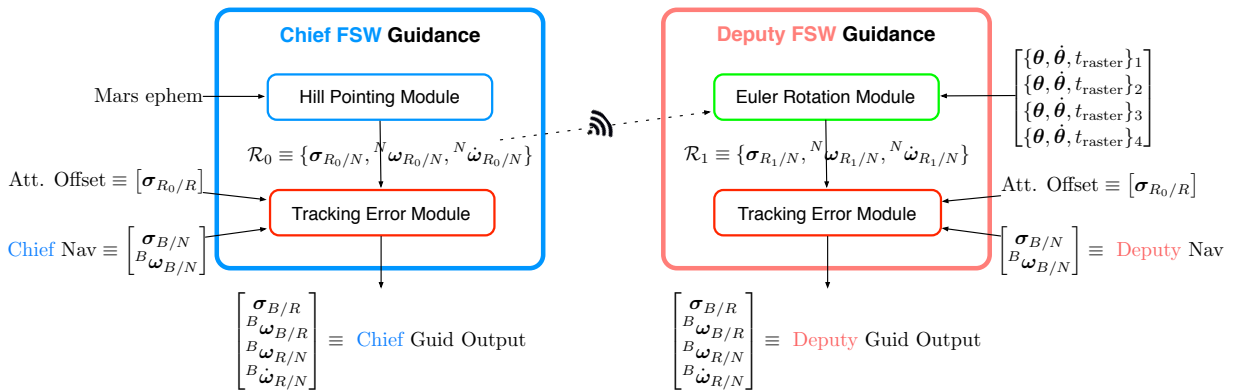


Figure 5.19: Stack of attitude guidance modules for the chief and deputy FSW suites

error and control torque computed by the deputy FSW. As expected, these converge to zero, indicating that the proper superposition of base and dynamic references –if there were discrepancies between the reference attitude states, the tracking error could not asymptotically converge to zero. Finally, Fig. 5.22 shows the relative scanning pattern: as commanded by FSW in Fig. 5.21(a) and as achieved in the spacecraft physical simulation in Fig. 5.21(b).

## 5.6 Summary

This chapter has described all the technical challenges associated to the assembling of an emulated flat-sat with unprecedented level of fidelity. Using this emulation, it is possible to test the performance of an embedded FSW application in a distributed and flight-like manner. There are two main tasks or aspects that have been critical to the successful implementation of the flat-sat emulation:

- (1) Development of the Black Lion communication architecture.
- (2) Modeling of FPGA registers and accompanying avionic hardware models.

While the development of Black Lion was initially motivated for an interplanetary mission,

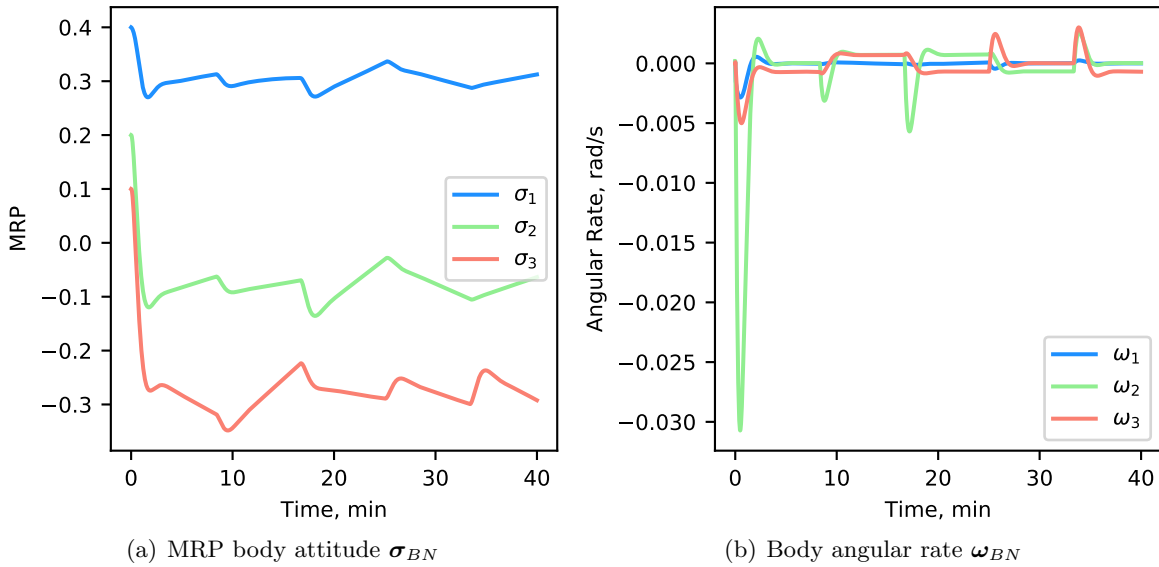


Figure 5.20: Deputy true attitude states

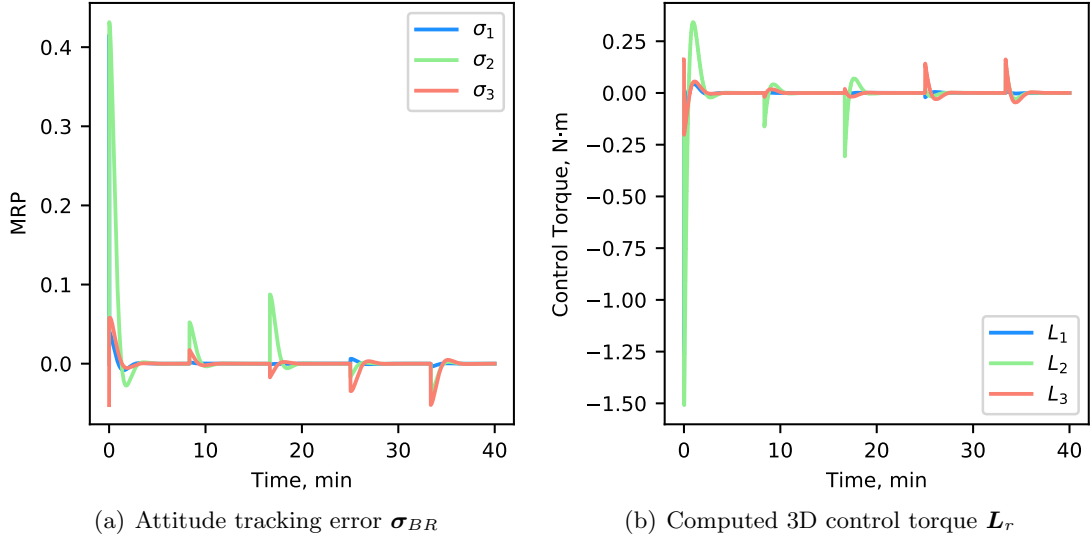


Figure 5.21: Deputy FSW states: tacking error and control torque

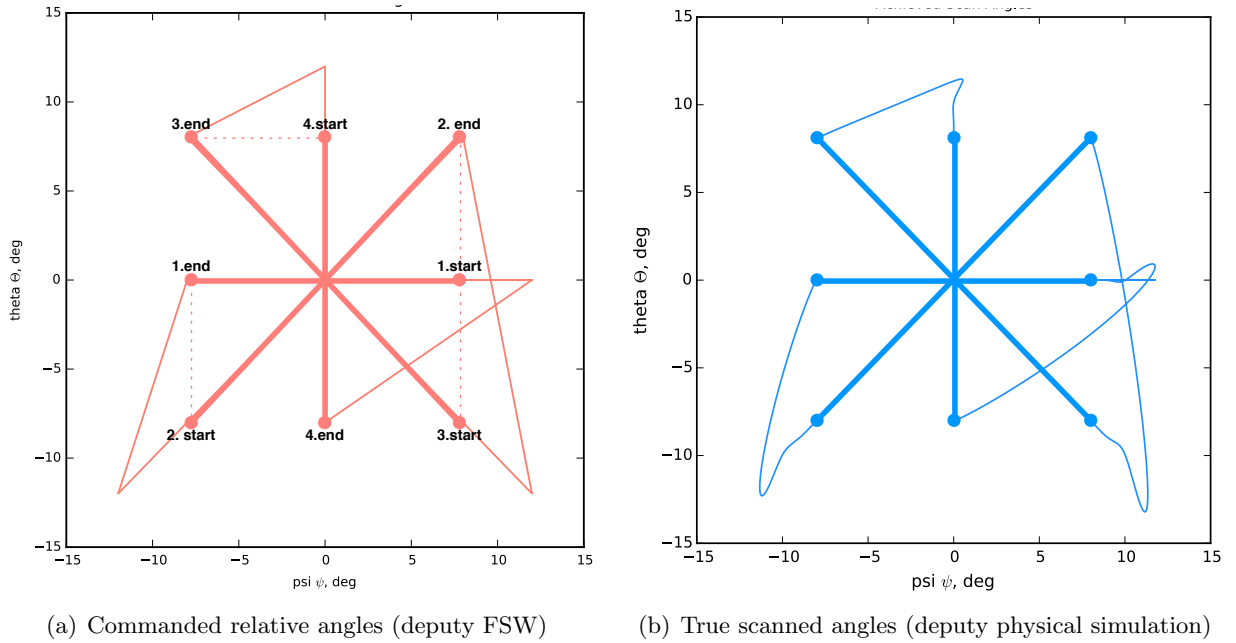


Figure 5.22: Relative asterisk pattern: commanded vs. true pointing angles

this communication system has been built under the principles of reusability and scalability and its applications extend beyond the flat-sat example discussed here. For instance, the use of Black Lion in a formation flying simulation with hardware-in-the-loop components has been demonstrated.

Additional use cases of Black Lion could involve: the integration of large clusters of spacecraft, having complex simulation components running on super-computers or cloud servers, as well as hybrid distributed simulations including both software and hardware spacecraft components.

For the aforementioned mission, emulated flat-sat testing through Black Lion has proved to be an extremely cost-effective means of performing system-wide testing early on in the mission's program, alleviating schedule constraints by using software models only. In addition, the flexibility of Black Lion has allowed running fault detection tests that could not be executed in any other testbed.

In turn, the modeling of FPGA registers and avionic components stands out for its high-level of fidelity. In addition to performing complex operations on FSW packets, these models also replicate hardware interrupts, making FSW believe it is running on an actual flight processor board—despite being actually a software emulation. While the presented avionic models and packet handlers are mission specific, the register space with its readers and writers constitutes a generic framework that can be applied to any FSW application. The registers have been designed and implemented in a generic manner where high-level handlers can be customized to satisfy particular mission needs. As a matter of fact, a simplified version of the same register space is used in the next chapter to test a different FSW application that is based on MicroPython rather than on cFS.



## Chapter 6

### Flight Algorithm Migration into MicroPython

Seeing the generalized interest in Python for desktop FSW development, it makes good sense to consider MicroPython as a middleware for embedded development. As a quick recapitulation, MicroPython is a lean and efficient implementation of the Python 3 programming language that includes a small subset of the Python standard library and that is optimized to run in micro-controllers and in other constrained environments. It presents many advanced scripting features while being compact enough to fit and run within just 256k of code space and 16k of RAM.

The compelling aspects of the modern and open-source MicroPython is that it has the potential of providing an embeddable infrastructure that is similar to desktop environments (in terms of user friendliness and interaction functionalities) while adhering to the needs of space (in terms of determinism and minimal use of resources). The endeavour pursuit in this chapter consists on building a Basilisk-based flight application on top of MicroPython and prove, through numerical simulation, that it is indeed feasible. The challenge is that these two software frameworks were never designed to work together and, therefore, their compatibility is *a priori* an open-ended question. While Basilisk uses Python, MicroPython is only a subset of the library and limitations are prompted to appear. In addition, in the Basilisk desktop environment the underlying C and C++ modules are wrapped through the SWIG library, which is as handy as it is large (and therefore cannot be used within MicroPython). While MicroPython can interface with native C code, the difficulty of creating your own custom modules still need to be assessed –recall that Basilisk C modules are complex flight algorithms and their interaction with the Python layer is also non-trivial.

With these considerations in mind, the chapter is outlined as follows: first, the technical steps required to migrate the flight algorithms out of the Basilisk desktop environment into the MicroPython embeddable environment are described. Next, a pure-Python tool developed to automatize and aid in the migration process is presented. This tool is named **AutoWrapper** and it is born from the same principle as the **AutoSetter** tool shown earlier: Python’s introspection capabilities. Then, a mechanism to post-process numerical simulation results coming from the constrained MicroPython environment is suggested and described. Numerical simulation results of a Basilisk-MicroPython FSW application tested in closed-loop are shown to demonstrate the feasibility of the system and the successful implementation of all the tools previously described in this chapter. Finally, a summary section describing the highlights of the results is included.

## 6.1 From Basilisk into a MicroPython application

The idea proposed in this manuscript is to use MicroPython for embedded setup and embedded execution of the same (unmodified) C flight algorithm code as in the Basilisk desktop environment. However, the standard way of extending MicroPython with custom C modules involves a lot of boilerplate code. For this reason, another open-source software tool is introduced: the MicroPython C++ Wrap<sup>1</sup> is a header-only C++ library that provides some interoperability between C++ and the MicroPython programming language. Using the MicroPython C++ Wrap the process of integrating C++ modules within MicroPython is drastically reduced. However, this comes at the cost of requiring all the modules to be written in C++ rather than in C; recall that, currently, all the FSW modules within Basilisk are written in C. With this in mind, the technical work required to migrate Basilisk flight algorithms into MicroPython can be broken down into three steps:

- (1) **Creating a C++ class for every C FSW module:** Since the MicroPython C++ wrapper is specially designed to wrap C++ code, the suggested approach for wrapping the unmodified C FSW algorithms, as they currently exist in Basilisk, is to create a C++

---

<sup>1</sup> <https://github.com/stinos/micropython-wrap>

wrapper class (new .hpp file) for every module (.h and .c file) there is. The C++ class is a wrapper in the sense that it does not implement new functionality: it simply provides callbacks and easy access to the underlying C variables and methods from the MicroPython layer.

- (2) **Generating integration code for every C++ class that needs to be available at the MicroPython layer.** MicroPython is meant to interact directly with the recently created C++ wrapper classes, treating them as if they were native Python modules. In order to achieve this behavior, it is necessary to recompile MicroPython after having declared and registered the different C++ classes, functions and types.
- (3) **Adapting existing desktop Python scenario scripts into MicroPython.** Since MicroPython is only a light version of the Python 3 programming language, some advanced Python functionalities and large libraries (like those usually employed for post-processing) are not supported. If this constraint is accounted for, the desktop Python scripts could, in principle, be seamlessly used within MicroPython, provided that they are written in version 3 of the language. With respect to Basilisk, however, MicroPython scripts currently import the C++ wrapper classes while desktop Python scripts import directly the C modules. In this sense, the import and instance of modules is different between MicroPython and Python scenario scrips –which is a small adjustment.

These three technical steps are necessary in order to integrate Basilisk into MicroPython. In addition to integrating these software frameworks together, it is also necessary to compile them and build them jointly. Because the Basilisk-MicroPython system encompasses the use of three distinct software repositories and tools (i.e. Basilisk, the MicroPython C++ Wrapper and MicroPython), building the system together is an involved endeavour. Appendix D provides step-by-step guidelines on how to build the Basilisk-MicroPython FSW system for Unix.

## 6.2 Migration Mechanism: the Auto-Wrapper

The interesting part of the described migration tasks is that the creation of the C++ wrapper classes and the generation of the MicroPython integration code can be handled automatically. Let us recall the introspection capabilities that are inherent to the Python language. Similarly to how the **AutoSetter** produces specific C setup code for integration within cFS (see Section 4.3 and Appendix B for reference), an equivalent script has been developed to automate the integration of Basilisk modules within MicroPython. This new introspective script is referred to as the **AutoWrapper** and it is also written in Python. Pseudo-code for the **AutoWrapper** is also presented in Appendix B.

The process of migrating Basilisk FSW modules from the desktop environment into MicroPython through the **AutoWrapper** is illustrated in Fig. 6.1. Note that the input to the **AutoWrapper** is simply a desktop Python scenario script and the output are the corresponding C++ wrapper classes and the MicroPython integration code patch. The **AutoWrapper** tool uses the same mechanism as the **AutoSetter** to figure out the variable and method names of the underlying C modules. Once introspection is granted, the code for the C++ wrapper classes and the MicroPython integration can be generated through templates. Listing 6 showcases a sample C++ wrapper class that has been automatically generated by the **AutoWrapper**, while Listing 7 provides pseudo-code for the Python template that the **AutoWrapper** uses to define how the C++ classes are to be written.

The combination of the C++ wrapper classes and the MicroPython integration code is equivalent to the functionality that SWIG provides out of the box for Python in a regular desktop environment. Thanks to this, Basilisk FSW simulations can be set up and executed from the embeddable MicroPython layer in the same way they are executed from Python in the desktop environment.

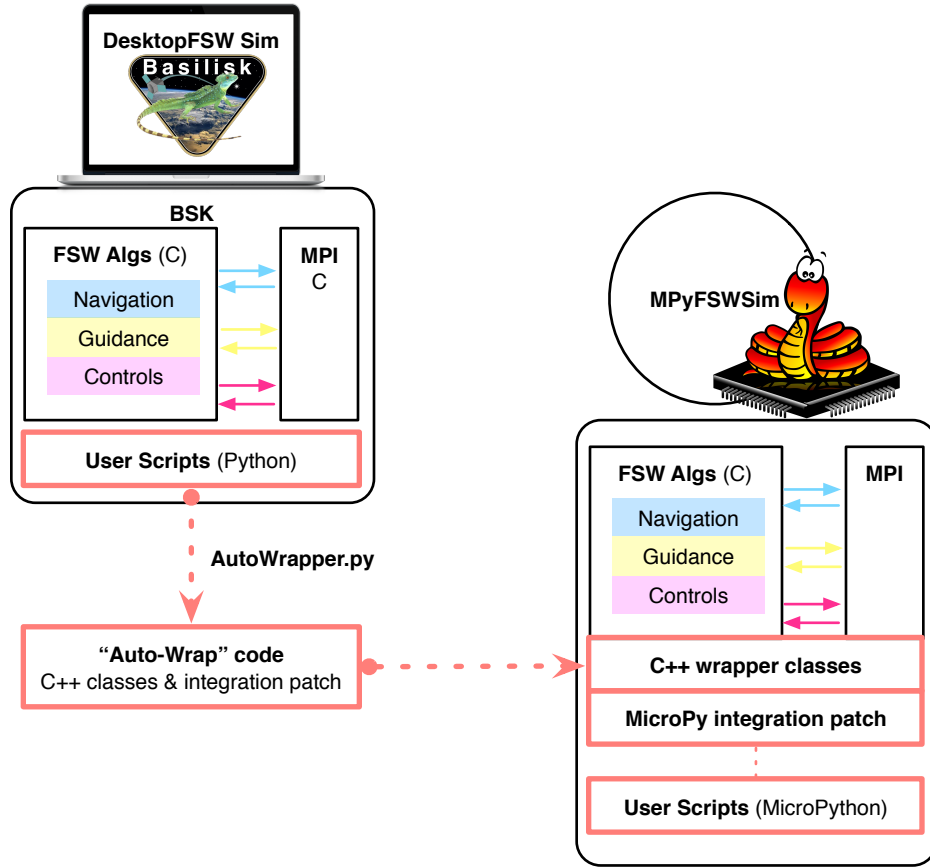


Figure 6.1: Flight algorithm migration into MicroPython through the AutoWrapper

### 6.3 Post-Processing MicroPython Results

Recall from earlier that, in the desktop prototyping environment, Python is used for 1) setup, 2) desktop execution and 3) post-processing of the simulation results. However, MicroPython cannot handle post-processing because it is meant to be embedded and, not surprisingly, large libraries for analysis and plotting are not supported. Then, the question that raises immediately is how to validate the results from a MicroPython simulation run. As a matter of fact, MicroPython is capable of logging all the data from an execution run. Since the problem is about pulling and plotting such data within the constrained environment, an alternative solution is to archive the data in a binary file. Thanks to the interoperability between MicroPython and regular Python, the archived results can be loaded without modification back into the desktop Python environment for

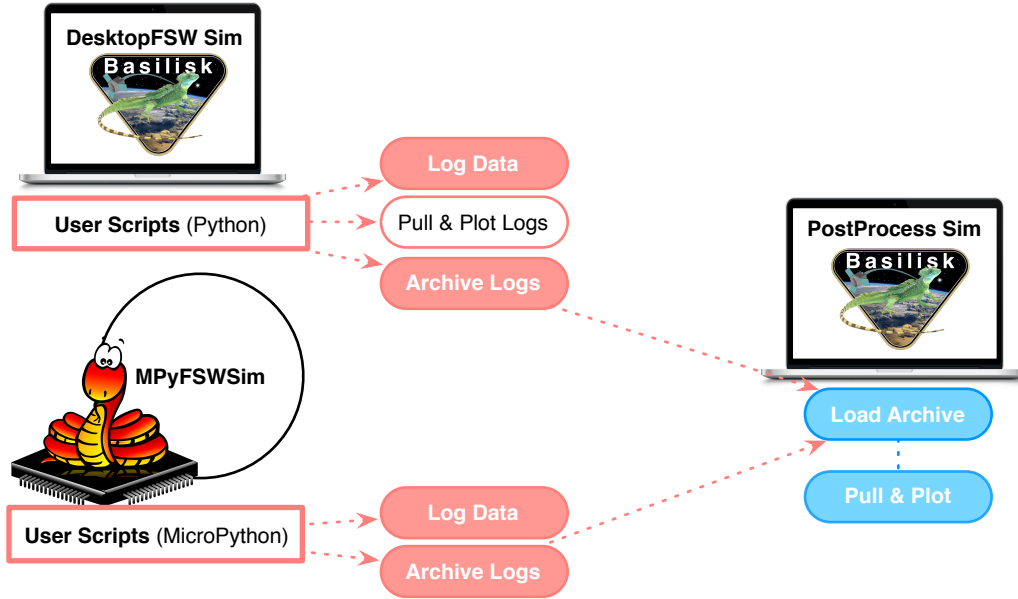


Figure 6.2: Post-processing Python (desktop) and MicroPython (embedded) execution runs.

regular post-processing. Figure 6.2 illustrates the suggested post-processing mechanism through an archived binary file. The key aspect of this approach is that the post-processing simulation is agnostic of the archive file being created out of the Python desktop simulation or out of the MicroPython embedded simulation. Such agnosticism contributes towards a more homogenous and smooth process for cross-environment testing.

In order to elaborate further on how the archived binary file is structured, it is necessary to go back to Basilisk’s architecture, since they are intrinsically linked. All the Basilisk simulations are actually instances of a simulation-base class written in C++ and this class contains a member variable that is a message logger. Recall that Basilisk is based on a publish-subscribe messaging interface; meaning that all the modules instantiated in a simulation communicate with each other through messages that have unique message names and IDs. The message logger of the simulation-base class is used at the Python level in order to define which message names are to be logged and at which logging rate. For each message name to be logged, a log element is created as a C++ class. This log element has an associated memory buffer, which increases over time as more instances of the same message are being stored. Once the simulation is over, the buffer associated to each

element/message can be retrieved back at the Python layer for post-processing. Additionally, these buffers can also be written into a file (i.e. the binary archive). The archive is simply created by looping through the different log elements (or message names) and, for each element, writing out the message data (message name, message ID, number of logs and raw buffer) into the file using I/O functions from the C++ standard library.

#### 6.4 Numerical Simulation: Testing Basilisk-MicroPython FSW

In order to fully prove the validity of MicroPython as a FSW target, it is necessary to test the performance of the embedded flight algorithms in simulated closed-loop. With this purpose in mind, several Basilisk modules have been integrated within MicroPython: a subset of the C FSW modules, the basic C++ architectural modules (e.g. process containers, task containers, messaging system) and a register-like module that enables interaction between FSW and the external world. A simplified illustration of the MicroPython-FSW setup, as well as its interaction with an external spacecraft simulation, is shown in Fig. 6.3. The register interface module in Fig. 6.3 is actually a

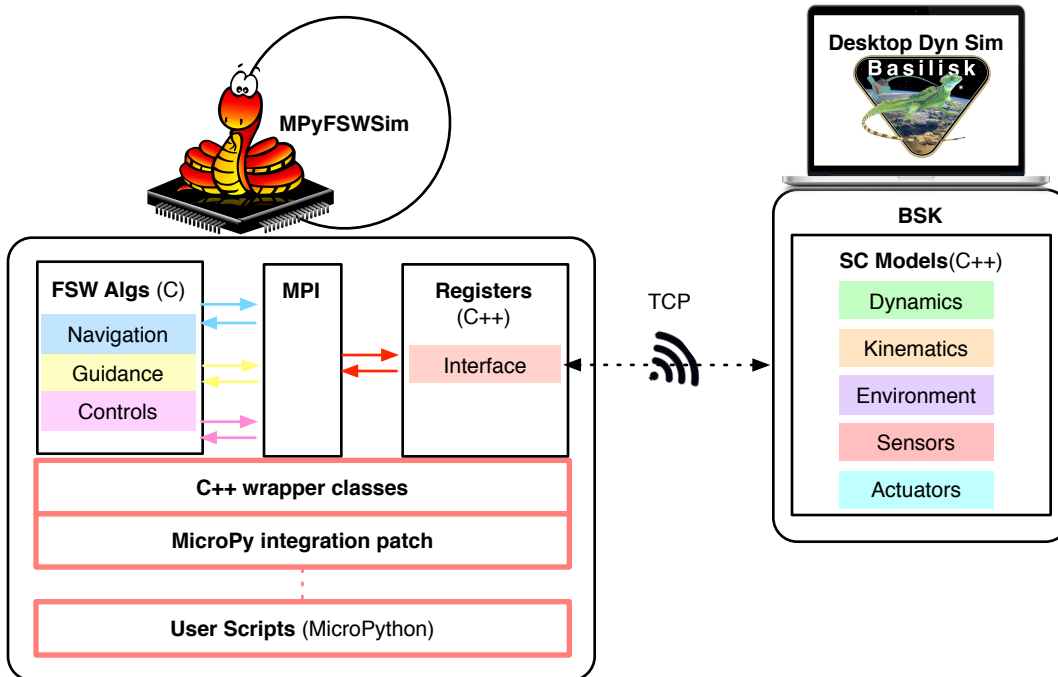


Figure 6.3: Closed-loop testing of MicroPython flight algorithms

simplified version of the complex FPGA register space implemented earlier in the context of cFS. The concept of snorkel readers and writers is exactly the same.

An inertial pointing guidance maneuver is used in order to prove the validity of MicroPython as a target for Basilisk flight algorithms. The FSW modules involved in this closed-loop maneuver are the following:

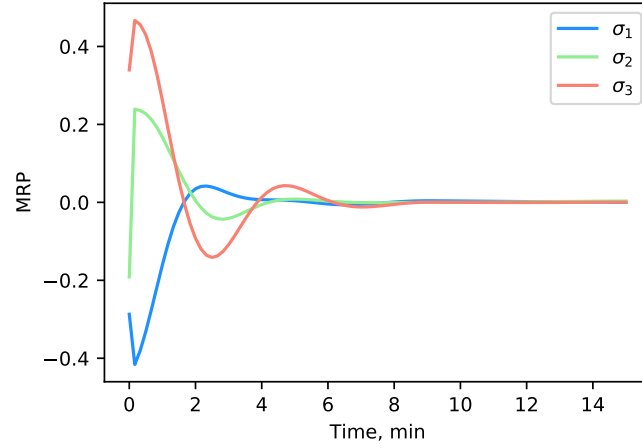
- **Vehicle configuration:** contains vehicle static data used by other modules.
- **Reaction wheel (RW) configuration:** contains RW static data used by other modules.
- **Inertial pointing (guidance module):** computes an inertial 3D reference, which is conformed by an MRP attitude set, angular rate and angular acceleration.
- **Attitude tracking error (guidance):** computes the tracking error between the current state and the desired reference.
- **MRP feedback (controls module):** computes a 3D control torque.
- **Reaction wheel motor torque (controls):** maps the 3D control torque into individual motor torques for the RW pyramid (a set of four RWs is used in the physical simulation).

Figure 6.4 illustrates numerical results from the closed-loop simulation. Figure 6.4(a) displays the attitude tracking error evolution as computed by FSW, while Fig. 6.4(b) illustrates the control torques commanded to the reaction wheel pyramid. The FSW states computed within MicroPython have been post-processed and plotted *a posteriori* on a desktop post-processing simulation as described earlier. The FSW algorithms running within MicroPython bring the spacecraft (which is initially tumbling) into an inertial 3D pointing state. The spacecraft simulation, which runs on a separate platform, is set in a Mars orbit.

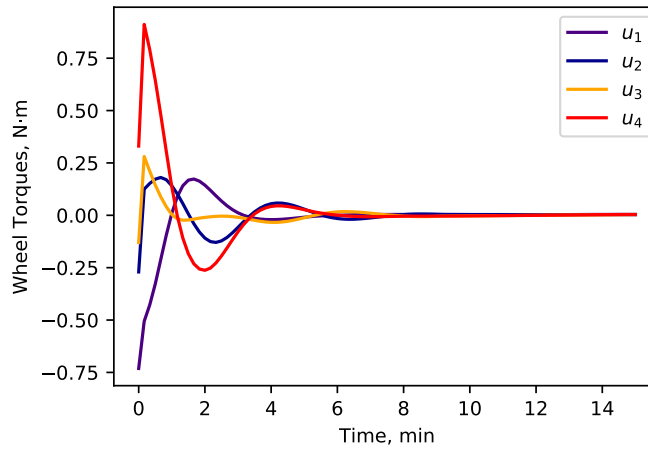
## 6.5 Summary

The previous results constitute an initial feasibility analysis for running Basilisk-developed flight algorithms within MicroPython. In this first proof of concept, the potential of MicroPython as a middleware for space applications is already showing. Therefore, early comparisons between





(a) MRP attitude tracking error



(b) Reaction wheel commands

Figure 6.4: Closed-loop testing of MicroPython-FSW: flight algorithm plots

MicroPython and cFS can be drawn.

- (1) **Reduced migration effort:** by means of MicroPython, the migration effort is reduced in the sense that the generated integration code is no longer specific but reconfigurable. The C setup code generated by the `AutoSetter` for cFS integration is specific to every single FSW configuration in a given Python scenario script; in contrast, the FSW C++ classes and MicroPython integration code generated by `AutoWrapper` are only written once. At this point, all the FSW states become fully reconfigurable from the MicroPython layer without need of recompiling the source code again.

- (2) **FSW states access:** Because MicroPython has full access to the message passing interface of the FSW application, it is possible to fully capture all the FSW states at any point in a simulation run. Further, the modeling of FPGA registers can be greatly simplified for the purpose of emulated flat-sat testing.
- (3) **Portability:** Last but not least, MicroPython guarantees the portability of a middleware layer without the replicated functionality imposed by cFS.

## Chapter 7

### Basilisk-MicroPython for Embedded Systems

The previous chapter has shown the first proof of concept for using the Basilisk flight architecture together with a MicroPython interpreter in order to yield a user-friendly, light-weight and flexible flight operating system that can seamlessly run in desktop environments and, potentially, in constrained flight environments. In the initial proof of concept, MicroPython runs on top of the Unix operating system but, indeed, the real use case of the Basilisk-MicroPython FSW application is to run in a constrained environment on top of an RTOS. Yet, the previous technical demonstration served to show that, by making use of the MicroPython C++ Wrap library, MicroPython can easily interface with Basilisk's C/C++ algorithm source code. As stated earlier, common requirements of spacecraft FSW involve:

- Interfacing to native C and C++ code.
- Real-time determinism: consistent and repeatable execution within a time constraint.
- Concurrency: processing multiple streams of events at the same time.
- Low use of resources: RAM, ROM and CPU.

The next sections of this manuscript analyze, precisely, the suitability of the Basilisk - MicroPython application to run in constrained environments with limited resources and deterministic requirements. This work involves targeting the application into 32-bit processors (like the family of LEON boards) as well as profiling and optimizing the memory and CPU usage of the

Basilisk-MicroPython application on Unix (with the aim of targeting a constrained Unix environment).

Figure 7.2 displays a system-level view of the Basilisk-MicroPython system for three different targets: unconstrained Unix (on the left), constrained Unix (in the middle) and embedded RTEMS-LEON. The system structure is the following: the highest level corresponds to the FSW application/executable, which in this case is conformed by the Basilisk C/C++ flight algorithms and the MicroPython scripts. The layer below contains 3rd party libraries, like the MicroPython C++ Wrapper (which is common in all three targets) and ZMQ. The ZMQ messaging library is used in the unconstrained unix target in order to perform closed-loop testing. The work in this chapter makes use of open-loop FSW simulations in order to analyze the suitability of the system in constrained environments. Because ZMQ would not be part of the onboard executable, there is no need to include it.

This chapter is outlined as follows: firstly, the port of MicroPython to the LEON target is presented. The different steps taken during the porting process are described and challenges are outlined. This first section also provides a summary describing the current limitations faced when attempting to integrate the Basilisk flight algorithms on top of MicroPython on 32-bit platforms like LEON. The second section in this chapter profiles the use of resources demanded by the Basilisk-MicroPython system when running a sample FSW application. In addition, optimizations are made to avoid dynamic memory allocation.

## 7.1 Port of MicroPython to the LEON Flight Target

MicroPython supports, out of the box, 64-bit architectures and a few 32-bit architectures. Each of these MicroPython ports is hardware specific and creating a port to a new architecture is a non-trivial task, specially when it comes to 32-bit targets. Currently, the family of LEON boards does not have a port available on the MicroPython repository. For this reason, before targeting the Basilisk-MicroPython application into LEON, there is a natural intermediate step that consists on porting the stand-alone MicroPython to LEON. The challenges of building a Basilisk-based

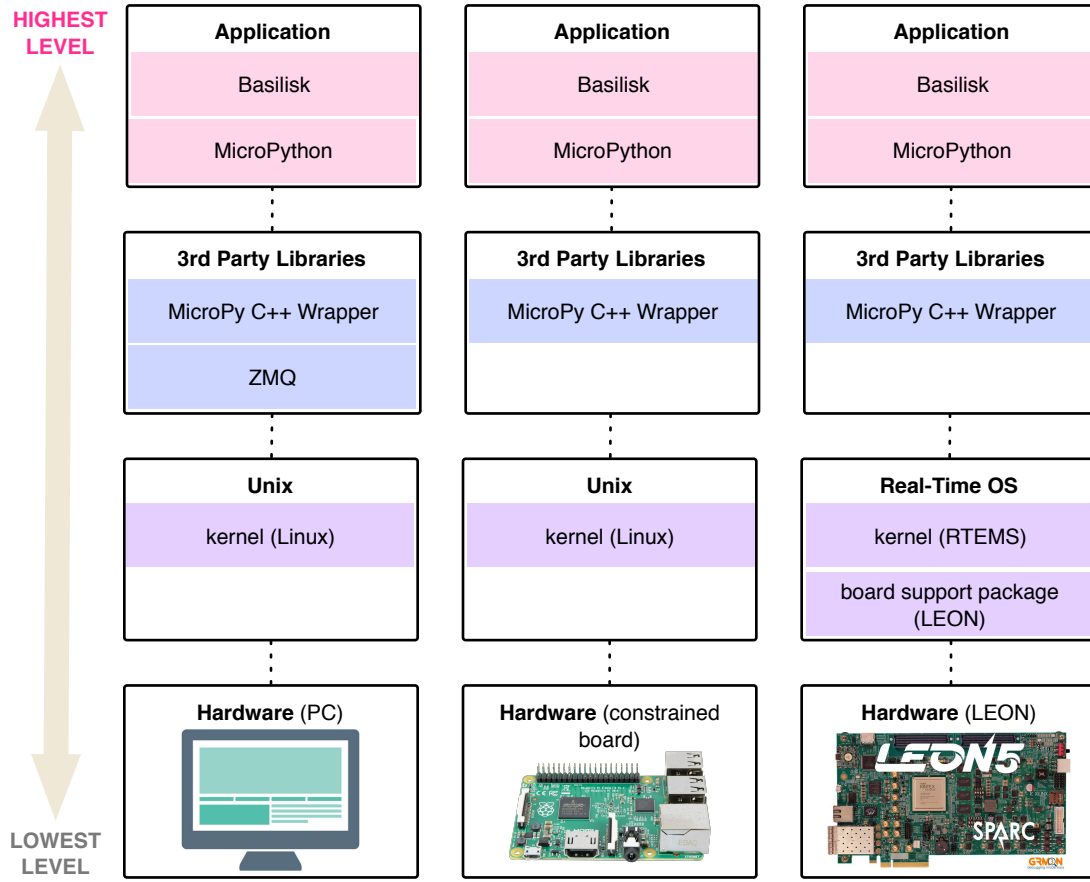


Figure 7.1: Basilisk-MicroPython: system-level architecture for different targets

application on top of MicroPython for any 32-bit target are described at the end of this section. For the moment, the focus is kept on building the stand-alone MicroPython for a new 32-bit port: LEON. The port to LEON is particularly interesting in the context of this thesis because these boards are widely used in flight applications. In addition, recall that an emulated LEON board was also used in the emulated flat-sat presented earlier.

As a matter of fact, ESA has also considered the use of MicroPython on LEON[33], but their work is only available under an ESA Community License of Type 3; implying that only companies belonging to the European Union can access this software repository. This thesis also aims to port MicroPython to LEON (or, equivalently, its virtual counterpart) and report the lessons learnt. Once MicroPython is granted to run on LEON, then the integration of Basilisk flight algorithms

can be considered.

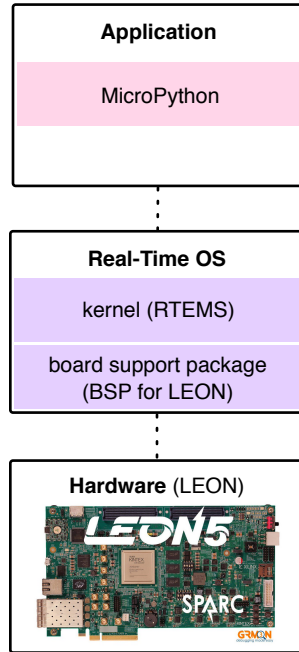


Figure 7.2: Port of the stand-alone MicroPython to the RTEMS-LEON target

Figure 7.2 shows the architecture for the port of the stand-alone MicroPython to RTEMS-LEON. The realization of such architecture can be decomposed into two main steps:

- (1) Build the RTEMS toolchain and the LEON board support package (BSP)
- (2) Build the MicroPython executable using the RTEMS-LEON toolchain in the previous step

Each of this steps is further discussed next.

### 7.1.1 Building the RTEMS toolchain and LEON BSP

Firstly, the concepts of toolchain and board support package are explained. The notion of a toolchain is usually used in the area of embedded systems and it is defined as a set of distinct software tools chained together in order to compile, link, and deploy software from a development host to a target device. In most embedded devices, the device itself does not have enough capability to support development directly on the device –therefore, a toolchain is needed. Generally, a toolchain will contain a cross-compiler and linker for the target, possibly a debugger (to allow

embedded on-device debugging), possibly a simulator for testing on the host, and a mechanism for deploying software to the device. The toolchain itself is often presented as a collection of command-line tools to use in the host; the RTEMS toolchain used in this work is indeed a suite of command-line tools for Unix. The RTEMS sources<sup>1</sup> are available on Git and, in order to build the toolchain, it is necessary to clone and bootstrap the RTEMS kernel and the RTEMS source-builder.[40]

In turn, in embedded systems, a board support package (BSP) is the layer of software containing hardware-specific drivers and other routines that allow a particular operating system (usually an RTOS) to function in a particular hardware environment (a computer or CPU card), integrated with the RTOS itself. In order to support a specific RTOS on a particular hardware, it is necessary to create a BSP that allows that RTOS to run on the target platform. In this thesis the RTEMS toolchain is used to build a BSP for the LEON target.

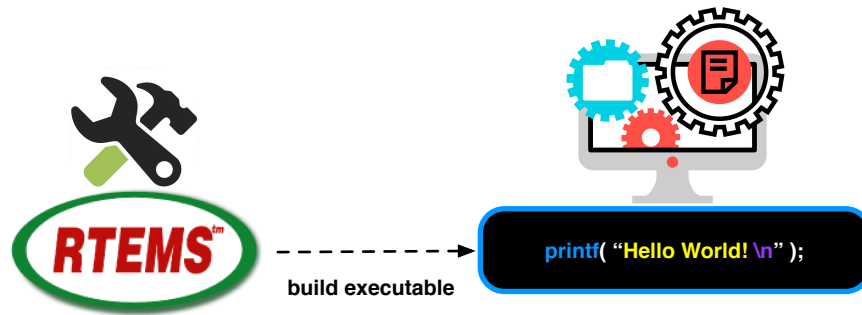
With the RTEMS toolchain and the LEON BSP bundled together, it is then possible to compile and link an executable that can run on a LEON board (or its emulated counterpart) with the RTEMS operating system on top. In order to test the bundle, a sample hello-world application is built. The resulting kernel is then executed as an RTEMS task within an emulated LEON board. This test is graphically illustrated in Fig. 7.3.

### 7.1.2 Building the MicroPython executable

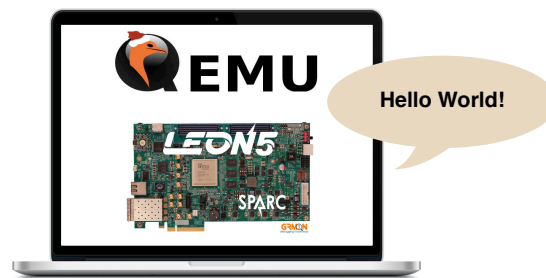
Once the created RTEMS toolchain and LEON BSP are proven to work, the next step is to use them to compile and link the MicroPython executable. Such endeavour is depicted in Fig. 7.4. Since MicroPython is much more complex than a sample hello-world application written in C, the question is where to start in the building process. As mentioned earlier, in the current MicroPython repository there are several ports available that target 32-bit processors; for instance, there is a bare-ARM port, a QEMU-ARM port and an STM32 port among other. The target platform of this work is LEON, which implements a SPARC V8 instruction set. The SPARC architecture is a

---

<sup>1</sup> <https://github.com/RTEMS>



(a) Hello-world application built with the RTEMS toolchain and LEON BSP



(b) Hello-world application running as an RTEMS task on QEMU's emulation of LEON

Figure 7.3: Sample executable for RTEMS-LEON

32-bit RISC machine that is very similar to the ARM architecture. Therefore, the *modus operandi* used in this work is to take the Makefile from MicroPython's QEMU-ARM port as a start point and make the edits required for SPARC. This process is depicted in Fig. 7.5

Two set of edits are required in the Makefile and associated source code. The first set of edits

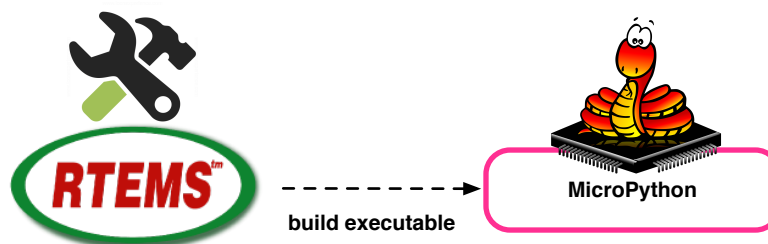


Figure 7.4: MicroPython executable for RTEMS-LEON



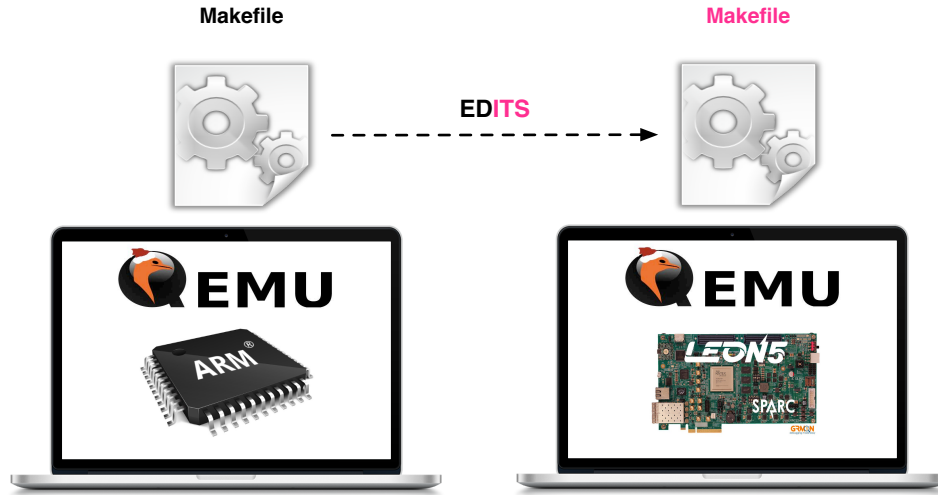


Figure 7.5: MicroPython Makefiles: adapting QEMU-ARM port to QEMU-LEON

consists on removing and/or replacing ARM specific code:

- UART (universal asynchronous receiver/transmitter) file: since UART ports are hardware specific and such interface is not required for QEMU's emulation of LEON, the UART source file within MicroPython's QEMU-ARM port is simply removed.
- Startup file: the startup file is used to initialize the given hardware board (or, in this case, its emulated counterpart). The file contains handlers for: bootstrapping the system, calling MicroPython's main, exiting, resetting and defining the interrupt service routine. Some of these handlers are embedded assembly functions and, therefore, they are target specific. It is then necessary to adapt the startup-related assembly functions from ARM to SPARC.

The second set of edits is related to certain shortfalls of the RTEMS toolchain with respect to the GNU ARM embedded toolchain. The RTEMS version used in this work is the latest one available, which is version 5. The shortfalls are the following:

- Assert function (`__assert_func`): MicroPython uses `__assert_func` in order to catch run-time errors. Since the RTEMS 5 toolchain does not come with an assertion source file, a custom `assert.c` for SPARC needs to be added.

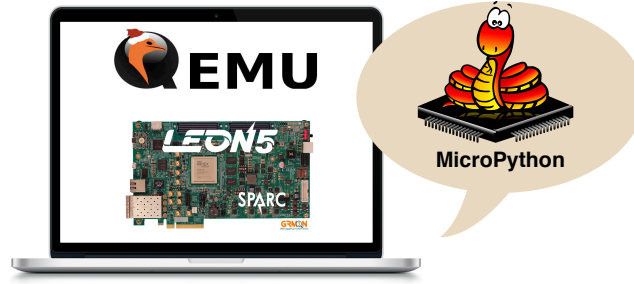


Figure 7.6: MicroPython kernel for QEMU's emulation of LEON

- Jump calls (`setjmp` and `longjmp`): as pointed out in Ref. [33], MicroPython requires a non-local jump mechanism to handle exceptions. This can be provided by the standard C `setjmp/longjmp` calls, or by some custom code, which usually needs to be written in assembler. Because RTEMS 5 does not include `setjmp/longjmp`, a custom version for SPARC needs to be provided. Ref. [33] provides pseudo-code for the implementation of the `setjmp/longjmp` assembler functions.

After integrating the described changes into the Makefile, the MicroPython executable is compiled and linked using the RTEMS-LEON toolchain (as in Fig. 7.4). Then, the resulting executable is loaded as a kernel for QEMU's emulation of LEON as in Fig. 7.6. The relevance of the porting of MicroPython to LEON and the challenges and limitations associated to integrating Basilisk flight modules on top are discussed in the next summary section.

### 7.1.3 Summary

This section has shed light on the challenging steps required to port MicroPython to LEON platforms. Whereas this endeavour has also been accomplished by ESA[33], their software repository is not open to access; therefore, it seems crucial to report and document the details of this technical task in the hope that future researchers can build up from here.

Within the scope of this thesis, it was desired to port not only MicroPython to LEON but also the MicroPython C++ Wrapper and the Basilisk flight modules, as in Fig. 7.2. In the pursuit of this objective, it has been attempted to compile and link together MicroPython, the C++ Wrapper

and Basilisk for several 32-bit ports: the recently created QEMU-LEON port as well as the QEMU-ARM and STM32 ports that are already available in the MicroPython repository. However, the MicroPython C++ wrapper library has many dependencies on advanced C++ functionality, which is not easily supported by embedded toolchains like RTEMS-LEON and GNU ARM. As a matter of fact, the MicroPython C++ wrapper library, which acts as the glue between MicroPython and Basilisk, is still a project in beta stage and it has only been tested for Unix and Windows. Although a lot of effort has gone into solving the encountered compile errors, the full Basilisk-MicroPython application could not be successfully built for any of the considered 32-bit targets.

## 7.2 Basilisk-MicroPython Profiling: Use of Resources

This section aims to profile (and constrain if necessary) the resources that the Basilisk-MicroPython application takes up when running on the Unix environment. The proposed analysis is relevant for different reasons:

- (1) Although traditional flight processors are 32-bit platforms, it is still possible to determine in a 64-bit platform whether a certain FSW application meets the constraints of an embedded system or not. For example, the LEON board that is considered in previous sections runs at 80MHz and has at least 8 megabytes of RAM memory. Determining how much resources are allocated for running the Basilisk-MicroPython FSW application on Unix can tell us if the complete FSW application could fit into LEON. The main advantage of profiling the use of resources in Unix is that there are many tools available to do it. For instance, Python's `memory-profiler`<sup>2</sup> module, Valgrind's `Massif`<sup>3</sup> or Valgrind's `Callgrind`.<sup>4</sup> The Valgrind tools are applicable to any kind of executable (i.e. they are not linked to a specific programming language) and therefore can be perfectly used for MicroPython. In turn, Python's `memory-profiler` can be used to perform support analysis on the Python desktop scripts that are equivalent to those in MicroPython (recall Fig. 6.2 for reference).

---

<sup>2</sup> <https://pypi.org/project/memory-profiler/>

<sup>3</sup> <https://valgrind.org/docs/manual/ms-manual.html>

<sup>4</sup> <https://valgrind.org/docs/manual/cl-manual.html>

- (2) As mentioned earlier, there is an increasing interest on using commercial-off-the-shelf processors for flight applications[9, 17]. In the lines of more modern approaches to space missions, the Basilisk-MicroPython application could be considered ready for flight if adapted to a constrained Unix environment.

In order to determine the suitability of an application/executable for a given processor board, there are three main resources to look at: RAM (random-access memory) usage, ROM (read-only memory) usage and CPU usage. Each of these resources is analyzed separately in the upcoming sections. While the resources demanded by the stand-alone MicroPython executable are already studied in Ref. [33], this thesis focuses on determining the use of resources demanded by the Basilisk-MicroPython FSW application. As pointed out in Ref. [33], in space applications minimizing RAM usage is usually a priority above all other design considerations. It is also important to keep in mind that, when optimizing for minimal resource usage, RAM, ROM and CPU optimizations usually play off against each other. With all these considerations, the usage of the different resources is discussed next.

### 7.2.1 RAM Usage

Allocation of RAM memory comes in two different flavours: static memory (data stored in the stack) and dynamic memory (data stored in the heap). Whereas stack usage is fixed (i.e. memory is allocated when the program is compiled), heap memory is allocated at run time. The size of the heap is limited by the size of virtual memory. Synchronous applications like the onboard FSW executable run in cycles and, therefore, heap memory allocation (if any) occurs in a control loop. The problem of using a finite resource (heap memory) in an “infinite” control loop becomes obvious. Having said that, because data on the heap can be accessed randomly at any time (i.e. it is RAM), it is also possible to allocate and deallocate (or free) blocks of memory at any time.

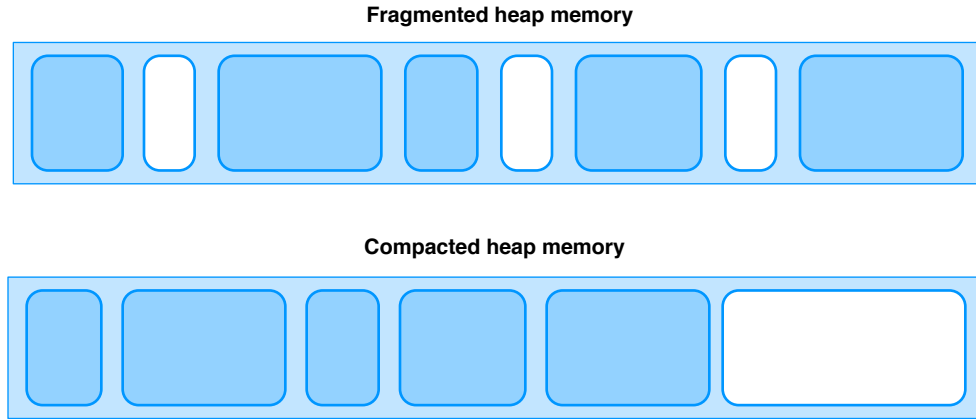


Figure 7.7: Heap memory: fragmented vs. compacted

#### 7.2.1.1 Garbage Collection and Alternatives

In the context of a control loop, the function of the garbage collector (GC) is precisely to free resources such that they can be reused the next time around. However, keeping track of which parts of the heap are allocated and which are free is a non-trivial task and the question boils down to what to free and when to do it. The non-precise nature of most garbage collectors implies that determinism is not guaranteed: freeing memory takes up variable amounts of time to complete and these calls are not made in a fixed pattern. Another potential problem associated to garbage collection is the fragmentation of the heap, which can happen after lots of allocations and deallocations. This problem is illustrated in Fig. 7.7. For example, the heap might have  $x$  words free which are spread out as one word in  $x$  places; this means that any allocation request greater than one word would fail.

As described earlier in Section 1.1.1.2, dynamic memory allocation and garbage collection are features of scripting languages like Python. However, MicroPython is capable of performing many operations without allocating heap memory. In the scope of this thesis, it is desired to determine when heap memory allocation is needed and when is not. In the light of all the challenges associated with implementing a deterministic garbage collector[33], the proposed approach is to remove completely garbage collection and, instead, work towards the development of a Basilisk-

MicroPython FSW application that does not allocate heap memory beyond initialization.

Appendix E provides some benchmarks for the heap memory consumption of different applications when running the same sample script. The different applications analyzed are the following:

- (1) Python 3
- (2) MicroPython with GC
- (3) MicroPython without GC
- (4) Basilisk-MicroPython

The results presented in Appendix E motivate the use of MicroPython without GC and showcase that the addition of the Basilisk modules to the MicroPython built does not suppose a performance hit in terms of RAM usage.

#### 7.2.1.2 Deterministic Basilisk-MicroPython FSW Application

This section aims to analyze the RAM usage of the Basilisk-MicroPython system when running a sample FSW application. For the purposes of profiling RAM, the FSW application is tested in open loop, meaning that the Black Lion communication architecture and the spacecraft physical simulation are, in this case, not present. In this application, FSW performs a guidance maneuver that consists on spinning inertially. The reason for using an open-loop simulation is that its closed-loop counterpart would require the integration of the ZMQ message library and of the Black Lion communication interfaces. However, in real flight, none of them would be present. Therefore, they should not be included in the profiling analysis.

The profiling of heap usage is achieved using Valgrind’s Massif tool, as in Appendix E. Before showing these results, it is critical to understand both the script that is being executed (i.e. the FSW application script written in MicroPython) as well as the system that is running it (i.e. Basilisk-MicroPython). Figure 7.8 illustrates the Basilisk-MicroPython system on the left and a breakdown of the FSW application scripts on the right. It is relevant to mention that the overall structure

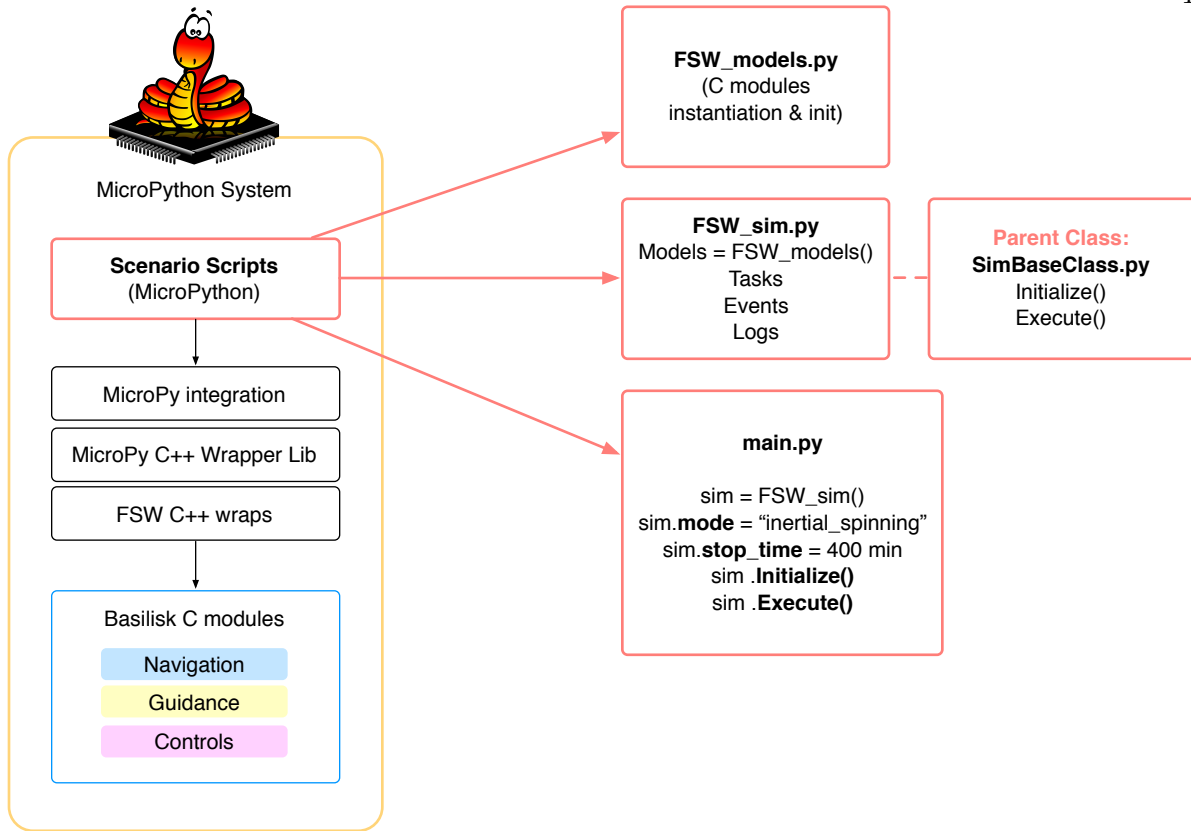


Figure 7.8: View of the Basilisk-MicroPython system and breakdown of the FSW application scripts

of the FSW application scripts is not tied to the specific sample maneuver being considered in this demonstration. The breakdown shown in Fig. 7.8 is the same for all Basilisk-based scenarios. As a quick recapitulation, the Basilisk C modules (blue box) run within MicroPython (yellow box); thanks to the integration of the MicroPython C++ Wrapper library and the additional “glue code” generated by the **AutoWrapper** (i.e. FSW C++ wraps and MicroPython integration patch in Fig. 7.8), the Basilisk C modules appear as importable modules within MicroPython. The FSW scenario scripts written in MicroPython (orange boxes) are then used to initialize and execute cyclically the underlying C flight algorithms. Without loss in generality, the scenario scripts can be divided into three categories: **FSW\_models**, **FSW\_sim** (which is a child class of a generic Python class named **SimBaseClass**) and the **main**. Each of these scripts and associated classes are reviewed next:

- **FSW\_models**: this MicroPython script instantiates the FSW C++ wraps and initializes the underlying C modules in a given configuration. These models are tied to specific FSW application/scenario being considered and, therefore, they are mission specific.
- **FSW\_sim**: this MicroPython script defines the scenario to be tested by: importing the desired **FSW\_models**, arranging them in tasks, creating FSW events (which in turn activate certain task groups) and defining the messages to be logged for testing purposes. Figure 7.9 exemplifies the arrangement of Basilisk C modules into tasks as well as the definition of FSW events. The FSW simulation itself is also mission specific and meant to be customized by the user. However, all the simulations are child classes of the same parent class: **SimBaseClass**. This parent class defines the skeleton of all Basilisk-based simulations and, therefore, it is the critical file to be profiled.
- **main**: this MicroPython script runs the actual scenario: it imports the **FSW\_sim** to be executed, it defines the FSW event to be tested, it defines the test duration, it initializes the system and it executes the control loop. Note that in Fig. 7.8 the **initialize** and **execute** calls belong to **SimBaseClass**. These are precisely the function calls that need to be analyzed in terms of RAM usage.

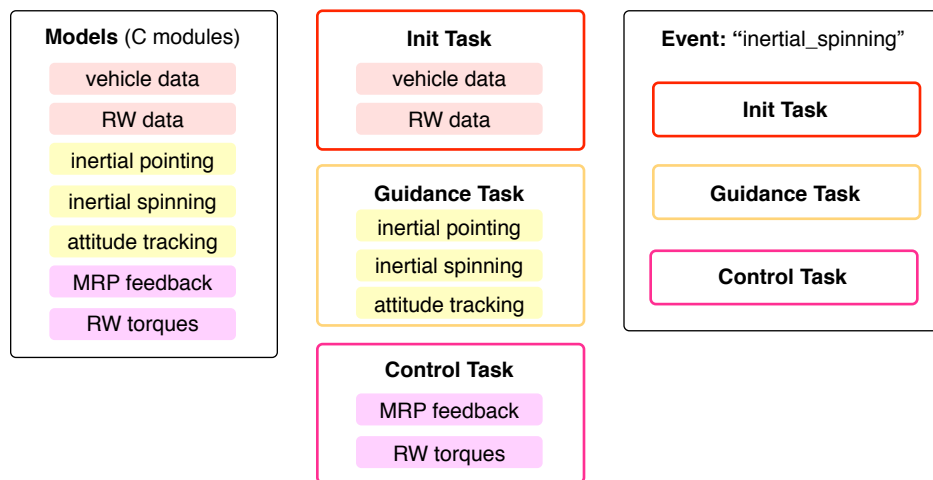


Figure 7.9: Sample models, tasks and events



Valgrind's Massif is used in order to figure out how much RAM memory is consumed during, firstly, initialization of the FSW application and, secondly, execution. Figure 7.10 shows the memory consumed up to initialization. Note that the graphic shows the heap memory usage in function of “time in B”, meaning that the time unit corresponds to the number of bytes allocated/deallocated on the heap and stack(s). In addition, the legend shows the principal sources of memory consumption. However, this thesis is only interested in the total memory heap consumption rather

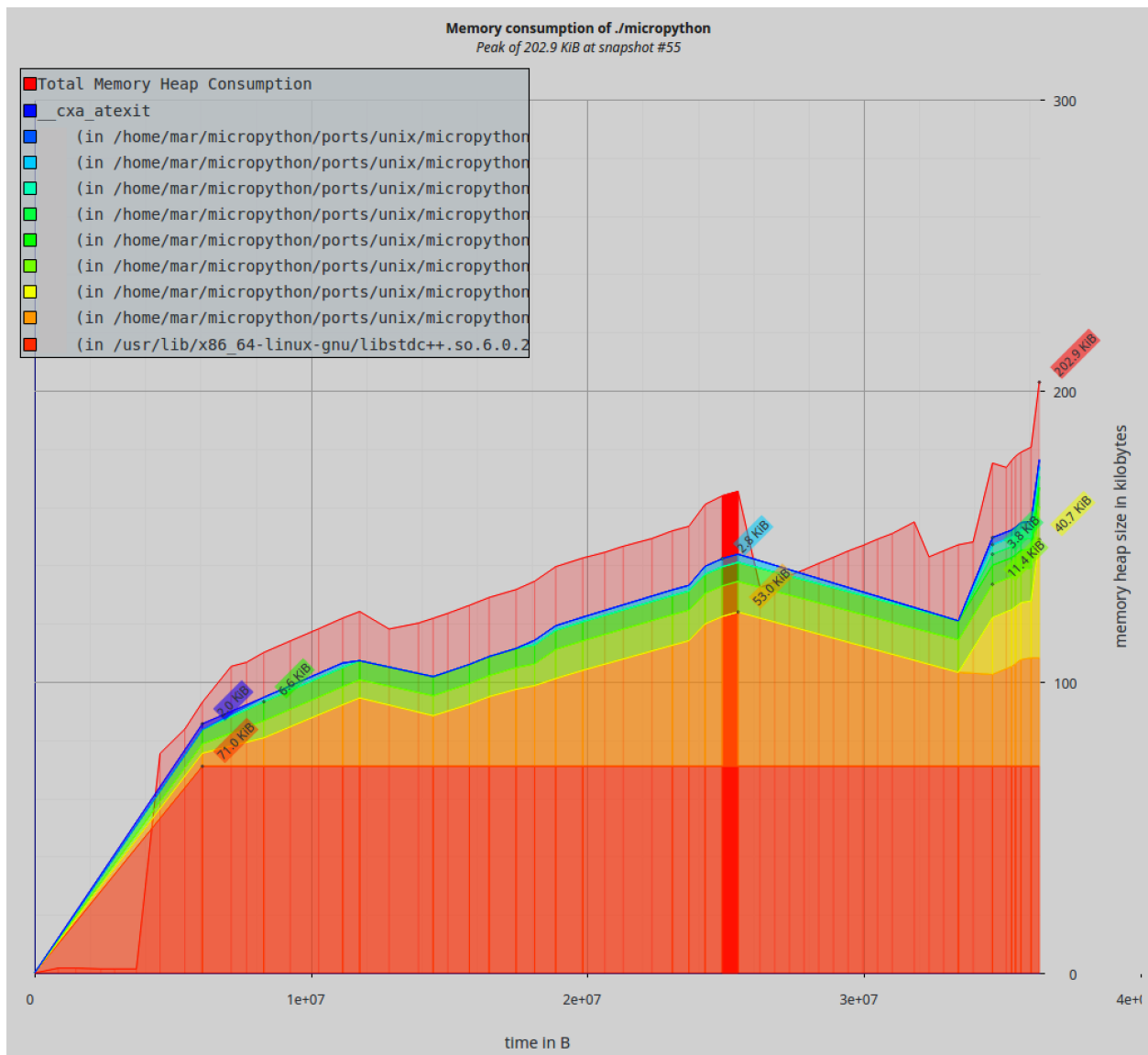


Figure 7.10: Heap memory used at initialization of the FSW application

than on its breakdown. Looking at the total memory consumption in Fig. 7.10, it is observed that the peak at 202.0 KiB coincides with `initialize` call from `SimBaseClass.py`. The earlier peak highlighted in red (at about 165 KiB) corresponds to the memory required just to bring up the Basilisk-MicroPython system (i.e. memory used previously to the instantiation of any of the FSW modules). To contextualize, for the application being profiled, the seven modules shown in Fig. 7.9 are instantiated, initialized and arranged in tasks and events during the `initialize` call.

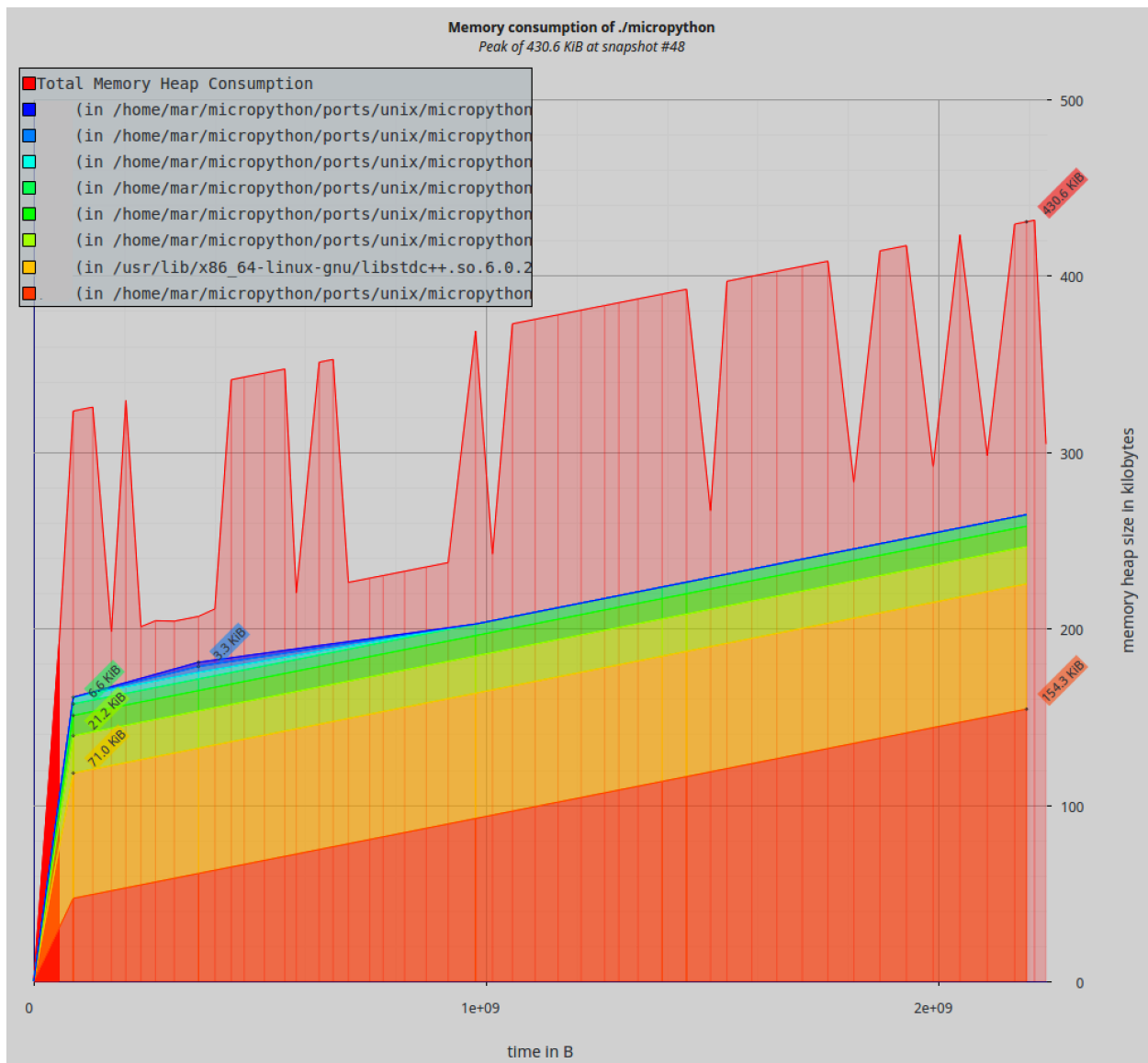


Figure 7.11: Heap memory used during execution of the FSW application for 40 virtual minutes

In turn, Fig. 7.11 shows the memory consumed by the FSW application when executing for a simulated time of 40 minutes. It is observed that the memory usage grows linearly with time, indicating that at every cycle there are Python objects being created and, since the garbage collector is not there to collect them, they keep accumulating despite being out of scope. In the light of the results in Fig. 7.11, the next steps consist on:

- (1) Tracking down where the leak is happening with the MicroPython scenario scripts.
- (2) Identifying which operations in MicroPython require heap memory allocation and should be therefore avoided.

In order to track down all the potential memory leaks in a reliable and efficient manner, the PyPi memory-profiler<sup>5</sup> comes in handy. The memory-profiler is a pure python module for monitoring memory consumption of a process as well as line-by-line analysis of memory consumption for python programs. The memory-profiler can be used to track memory consumption in the desktop Python simulation and, in this way, reveal where Python (and equivalently MicroPython) are consuming heap memory during the `execute` call. Extrapolating analysis results (from the desktop Python simulation into the constrained MicroPython simulation) is possible thanks to the equivalence between the FSW scripts in both the Basilisk-Python and the Basilisk-MicroPython systems.

For reference, pseudo-code for the `execute` method in `SimBaseClass.py` is provided in Listing 4. The operations happening within `execute` are non-trivial in the sense that they involve calls to both Python and C code. Similarly, some of the parameters governing the while loop are changed by the Python layer and, under the hood, by the C layer as well. The code in Listing 4 is only provided to showcase how the memory-profiler works; basically, the output of the profiler is a description of how much memory is used in executing each line of code. This output is shown in Fig. 7.12.

Listing 4: Code for the `Execute()` call within `SimBaseClass.py`

---

<sup>5</sup> <https://pypi.org/project/memory-profiler/>

```

def ExecuteSim(self):
    # Initialize control-loop parameters

    self.initializeEventChecks()

    nextStopTime = self.TotalSim.NextTaskTime
    nextPriority = -1

    # Execute control loop until the end of the simulation
    while(self.TotalSim.NextTaskTime <= self.StopTime):
        # Figure out time of next event

        if (self.nextEventTime <= self.TotalSim.CurrentNanos and self.nextEventTime >= 0):
            self.nextEventTime = self.CheckEvents()

            self.nextEventTime = self.nextEventTime if \
                self.nextEventTime >= self.TotalSim.NextTaskTime \
            else self.TotalSim.NextTaskTime

        # Figure out next stop time

        if (self.nextEventTime >= 0 and self.nextEventTime < nextStopTime):
            nextStopTime = self.nextEventTime
            nextPriority = -1

        # Execute the C flight algorithms (TotalSim is actually a C++ FSW process class)
        self.TotalSim.StepUntilStop(nextStopTime, nextPriority)

        # Set stop time for next time around

        nextPriority = -1
        nextStopTime = self.StopTime
        nextStopTime = nextStopTime if nextStopTime >= self.TotalSim.NextTaskTime \
        else self.TotalSim.NextTaskTime

```

As revealed by the Increment column in Fig. 7.12, there are only two lines of code within the while loop that use up memory:

- (1) `checkEvents`: this is a call to another Python function, which needs to be profiled separately.
- (2) `TotalSim.StepUntilStop`: this is actually a call to the underlying C++ code; `TotalSim`

Line #	Mem usage	Increment	Line Contents
370	67.133 MiB	67.133 MiB	@profile
371			def ExecuteSim(self):
372	67.152 MiB	0.020 MiB	self.initializeEventChecks()
373	67.152 MiB	0.000 MiB	nextStopTime = self.TotalSim.NextTaskTime
374	67.152 MiB	0.000 MiB	nextPriority = -1
375	67.242 MiB	0.000 MiB	while (self.TotalSim.NextTaskTime <= self.StopTime):
376	67.242 MiB	0.000 MiB	if(self.nextEventTime <= self.TotalSim.CurrentNanos and self.nextEventTime >= 0):
377	67.242 MiB	67.242 MiB	self.nextEventTime = self.checkEvents()
378			self.nextEventTime = self.nextEventTime if \
379	67.242 MiB	0.000 MiB	self.nextEventTime >= self.TotalSim.NextTaskTime \
380			else self.TotalSim.NextTaskTime
381	67.242 MiB	0.000 MiB	if(self.nextEventTime >= 0 and self.nextEventTime < nextStopTime):
382	67.242 MiB	0.000 MiB	nextStopTime = self.nextEventTime
383	67.242 MiB	0.000 MiB	nextPriority = -1
384	67.242 MiB	67.242 MiB	self.TotalSim.StepUntilStop(nextStopTime, nextPriority)
385	67.242 MiB	0.000 MiB	nextPriority = -1
386	67.242 MiB	0.000 MiB	nextStopTime = self.StopTime
387	67.242 MiB	0.000 MiB	nextStopTime = nextStopTime if nextStopTime >= self.TotalSim.NextTaskTime \
388	67.242 MiB	0.000 MiB	else self.TotalSim.NextTaskTime

Figure 7.12: Output of memory-profiler for the `Execute` call

is a C++ class wrapped through SWIG in the Python desktop environment and wrapped through the MicroPython C++ Wrapper in the MicroPython constrained environment. As a matter of fact, SWIG returns the underlying C++ class every time that the `StepUntilStop` method is called. Therefore, objects are being generated at the Python layer at every cycle. Recall that the profile in Fig. 7.12 is for the Python desktop environment (because the memory-profiler does not work with MicroPython). Having said that, the MicroPython C++ Wrapper does not return the underlying C++ class when it is being called. Therefore, this memory leak that shows up in Python is not present in MicroPython. The only remaining culprit is then the `checkEvents` call.

After tracing down and separately analyzing all the Python operations that take place underneath the call to `checkEvents`, it was discovered that the memory leak was being caused by the use of dictionaries in the Python layer. Python dictionaries are implemented using hash tables, which make look-up's fast at the cost of using up memory. When the garbage collector is not around, iterating over dictionary entries, accessing dictionary keys by name and updating dictionary values at every time step increases the memory consumption over time. Under the light of this finding, the FSW application scripts written in MicroPython have been modified to not use dictionaries. In the heap-consuming application profiled in Fig. 7.11, dictionaries were being used for two different

purposes:

- (1) Mark FSW events as active
- (2) Check which message names had to be logged for testing purposes

However, none of this purposes is critical to the proper execution of the control loop. Figure 7.13 displays the memory consumed by the Basilisk-MicroPython system when running the FSW application after removing the use of dictionaries. Now the heap memory consumption is flat and stabilized at 320 KiB.

Although this FSW application only includes a subset of C FSW modules and it is therefore smaller than a regular onboard executable, the results are very compelling. Because all the simulation structure required to run any FSW scenario is already in place (and profiled in Fig. 7.13), expanding the FSW application to include more modules, tasks and events will have a relatively small memory impact. The execution of the control loop, in particular, is agnostic to the number of modules present (in terms of RAM usage).

Having now proved that, after editing the FSW scripts adequately, the heap memory usage does not increase during execution of the control loop, it is important to showcase that the FSW process is actually running underneath. With this purpose in mind, the same exact FSW simulation (performing an inertial spinning maneuver) is executed again but, this time, logging messages and archiving them in a binary file.

The results obtained from post-processing the binary file back in the desktop environment are illustrated in Fig. 7.14. In particular, Fig. 7.14(a) shows the MRP attitude reference  $\sigma_{R/N}$  resulting from superimposing the inertial pointing and inertial spinning guidance modules. Next, Fig. 7.14(b) shows the attitude tracking error  $\sigma_{B/R}$ . Note that the attitude reference and the tracking error only differ in the sign of the components. This is because this maneuver is being tested in open-loop and, therefore, the sensed/filtered body attitude  $\sigma_{B/N}$  is simply zero. Finally, Fig. 7.14(c) shows the torques commanded by FSW to the pyramid of reaction wheels. Note that these torques are actually quite large; yet, this is only because the maneuver is in open loop.

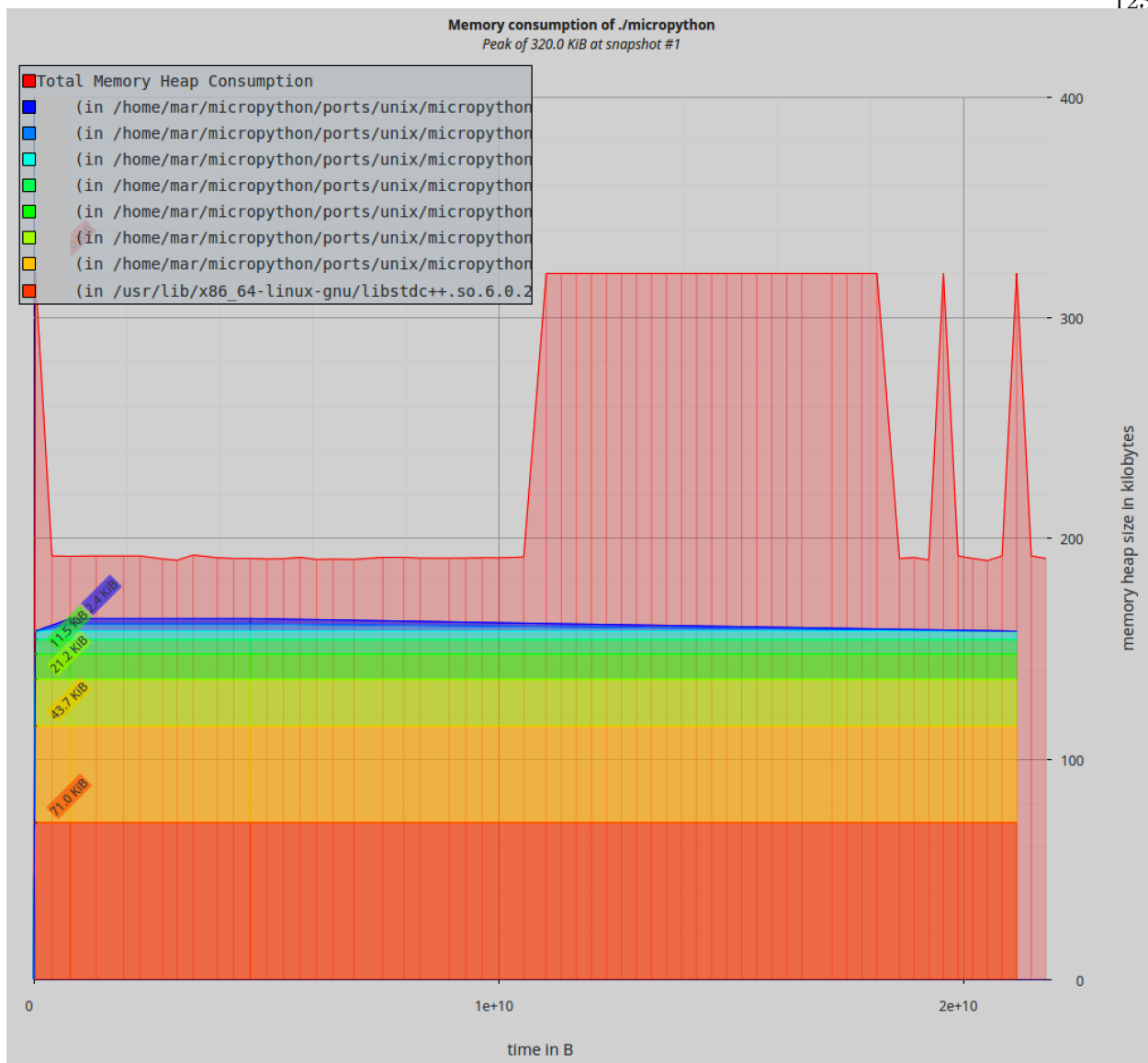
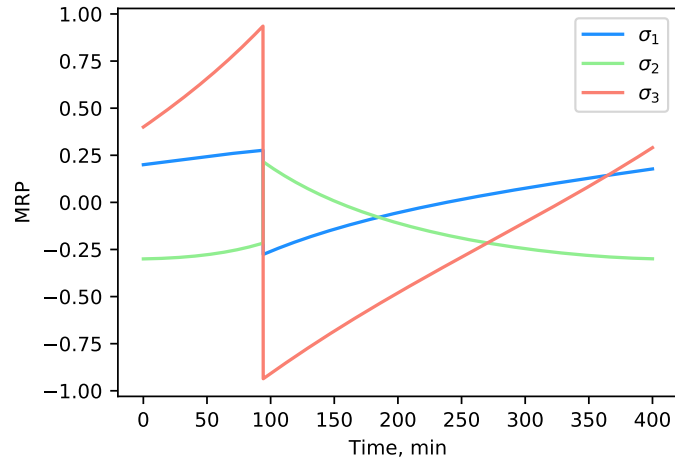
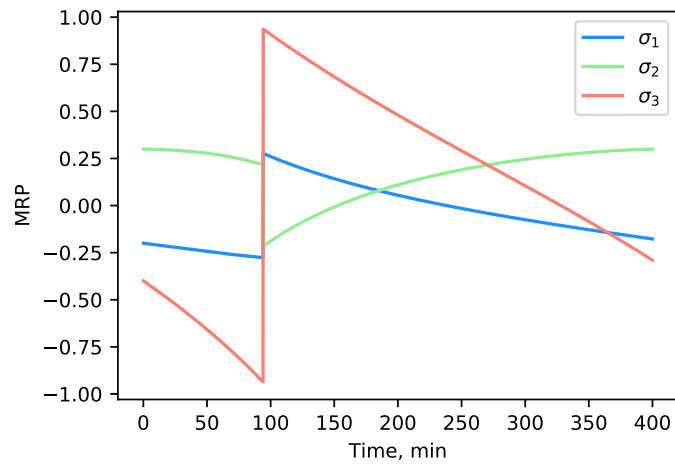


Figure 7.13: Heap memory used during execution of the FSW application for 400 virtual minutes after removing use of Python dictionaries

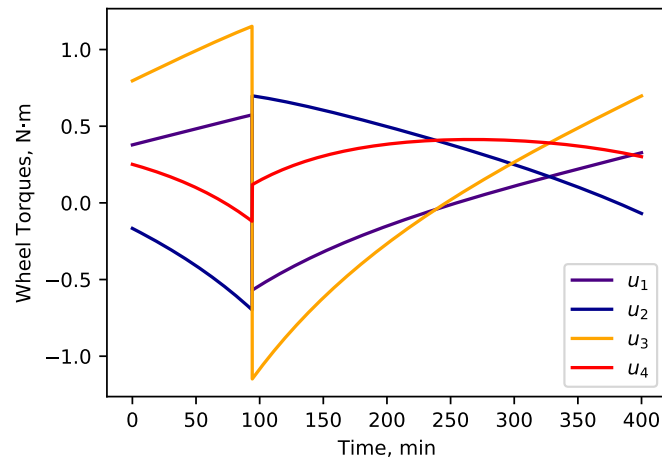
In summary, the continuous nature of the plots in Fig. 7.14 confirm that the underlying flight algorithms are performing its computations and exchanging messages properly. After having successfully profiled and optimized the RAM memory usage, it is interesting to look at ROM and CPU usage as well.



(a) MRP attitude reference



(b) MRP tracking error



(c) Commanded wheel torques

Figure 7.14: Basilisk-MicroPython open-loop simulation: inertial spinning



### 7.2.2 ROM Usage

ROM memory (or flash) provides permanent storage and it is where the executable firmware is stored. ROM chips often have a storage capacity of 4 to 8 MB. The amount of ROM necessary to store the Basilisk-MicroPython executable is obtained by looking at the code size. For comparison, it is also interesting to look at the code size of the stand-alone MicroPython executable when built for different ports: the Unix port, the so-called minimal-Unix port and the QEMU-ARM port (all of them being available in the MicroPython repository). Both the minimal-Unix and QEMU-ARM ports are compiled for 32-bit platforms. In addition, minimal-Unix only compiles a small subset of the MicroPython modules that are included in the regular 64-bit Unix port. As mentioned earlier, MicroPython provides many compile-time configuration options to enable or disable features of the Python language. By disabling unnecessary features, the code size can be drastically reduced; although this comes at the cost of less Python compatibility between desktop and embeddable environments. The code size of the stand-alone MicroPython executable in the aforementioned ports as well as the code size of the Basilisk-MicroPython executable in the Unix port are provided next:

- Basilisk-MicroPython executable for Unix: 1,691,816 bytes
- MicroPython executable for Unix: 408,976 bytes
- MicroPython executable for minimal-Unix: 170,216 bytes
- MicroPython executable for QEMU-ARM: 364,860 bytes

In the Unix port, building Basilisk on top of MicroPython almost triplicates the code size. Given that the integrated Basilisk C code is pretty minimal, the observed code-size increase can be attributed to the integration of the MicroPython C++ Wrapper library. Further optimization in code space could be achieved by:

- (1) Removing all unused functionality (and associated code) within the MicroPython C++ Wrapper library.

- (2) Finding a middle ground between the Unix and minimal-Unix ports in terms of enabled features. As it currently stands, the minimal-Unix port is too minimal to support the integration of the MicroPython C++ Wrapper library along with the Basilisk C modules.

Having said that, the 1.7 MB of code space required by the Basilisk-MicroPython executable are still far from the storage limit of ROM chips, between 4 and 8 MB.

### 7.2.3 CPU Usage

CPU time is the time between the start and the end of execution of a given program and it is a true measure of processor/memory performance. CPU time depends on the program being executed, including:

- (1) Number of instructions executed.
- (2) Types of instructions executed and their frequency of usage.

Next, the time that it takes for different systems to execute the same FSW application script is analyzed. The systems are Basilisk-MicroPython without GC, Basilisk-MicroPython with GC and Basilisk-Python. Once more, the FSW application consists on performing an inertial spinning maneuver for a simulated time of 400 min. The time-average results are the following:

- **Basilisk-MicroPython without GC:**  $t = 649.29$  ms
- **Basilisk-MicroPython with GC:**  $t = 720.1$  ms
- **Basilisk-Python (with GC):**  $t = 824.37$  ms

It is not surprising that the execution time is faster when the GC is not present, since calls to the GC actually take variable amounts of time to complete. In addition, calling the GC implies that a higher number of instructions is executed; therefore, it only makes sense that the time increases. By using Valgrind's Callgrind tool it has been observed that, in each run of the sample maneuver, 3.5 billion instructions are read without the GC and 3.8 billion when the GC is present (over the execution times reported above).

The fact that MicroPython is faster than Python is an achievement in the sense that MicroPython is specifically optimized to reduce RAM usage and, as stated earlier, resource optimizations usually demand a tradeoff between each other. It is also interesting to realize that all the average times are actually in the same order of magnitude. Very likely, this could be driven by the fact that the underlying Basilisk flight algorithms are actually implemented in C and they are the same for all the systems. In this sense, the more complex computations (which drive the overall execution speed) are being performed at the C level and Python/MicroPython simply acts as the executive layer.

Another relevant and very compelling advantage of running Basilisk-MicroPython without GC on specifically Unix is that, for testing purposes, it runs much faster than real time (400 simulated minutes run in 0.65 seconds). In turn, running in soft real-time could be easily achieved as shown earlier in Section 3.2, where a Basilisk FSW application on the Raspberry Pi runs in synch with a software-based clock.

#### 7.2.4 Summary

This section has profiled the use of resources (RAM, ROM and CPU) demanded by a sample Basilisk-MicroPython FSW application. In addition, the FSW scripts that are core to all Basilisk simulations have been optimized to avoid using up heap memory in the control loop when the GC is not present. This enhancements are not tied to the particular scenario being executed and they would hold true for any suite of maneuvers.

Being able to execute the Basilisk-MicroPython system without the GC is a key capability towards ensuring determinism on timing (in the sense that non-deterministic calls the GC do not take place) and proving that memory is not allocated dynamically (avoiding also the problem of heap fragmentation). Hence, making the system flight-like. In summary, flying Basilisk on top of MicroPython for Unix is very promising in that:

- (1) Basilisk FSW executions happen in a predictable manner that excludes the GC.

- (2) The simulation speed is roughly the same –actually, slightly faster– than in the Python desktop environment, which is extremely convenient for testing.
- (3) The memory usage is drastically reduced, hence fitting very moderate-size processors.

With increasing small-sat missions using commercial-off-the-shelf hardware and software solutions for flight exploration, running a Basilisk-MicroPython FSW application on top of Unix in flight could be a very interesting option. Future work to be done in this line consists on making the Unix operating system to behave in real time, which could be achieved by using a real-time Linux kernel.<sup>6</sup> Such enhancement would guarantee that the Basilisk-MicroPython system is real-time deterministic not only in terms of the executable (as proved in this section) but also in terms of the underlying operating system.

As a final and quick recapitulation, for any reader who might be interested on building upon the work presented, the Basilisk-MicroPython system can be integrated together as described in Section 6.1, built as explained in Appendix D and optimized as described in Section 7.2. With this basic infrastructure in place, the users/readers can customize the system with their own Basilisk FSW modules.

---

<sup>6</sup> <http://wiki.linuxfoundation.org/realtime/start>

## Chapter 8

### Conclusions

This thesis has designed and implemented end-to-end FSW development strategies and tools that guarantee migration transparency and testing continuity across testbeds: starting with flight algorithm prototyping on desktop environments, transitioning into middleware integration and ending with embedded testing in constrained environments. The strategies presented in this work are all based upon the principles of flexibility, scalability and reusability. In addition, the results and tools that are the outcomes of this thesis' research have been all documented and made available as open-source products. On these lines, the niche identified among state-of-the-art FSW development tool suites has been filled, satisfying: completeness of the tool suite as a FSW testbed, transparency of the flight algorithm flow between testbeds, architectural flexibility to include external models for testing, support of distributed simulations and open sourcing of the tools to the community. In order to conclude, it becomes useful to provide a brief summary of the particular highlights of each chapter.

Chapter 2 is dedicated to the desktop prototyping phase of the FSW development process. This chapter introduces the Basilisk software framework (a desktop testbed for prototyping flight algorithms and testing them in closed-loop dynamic simulations), upon which a novel strategy to autonomously generate attitude guidance references is implemented. Using the proposed guidance approach, complex attitude patterns are achieved through combination of atomic guidance modules that fulfill a well-defined functionality. As a quick recapitulation, there are three core functionalities that conform all guidance reference motions: base pointing reference, dynamic reference and

attitude offset. For each one of these functionalities, different software modules are implemented in the C language, tested and verified. In this manuscript, the mathematical equations behind each software module/algorithm are developed and numerical simulations illustrate how the individual components can be arranged to support different rotational dynamics mission requirements. This layered strategy for building an attitude guidance reference promotes code reusability throughout distinct mission profiles and, as a matter of fact, it is currently being applied to perform science and nominal attitude maneuvers in an actual interplanetary spacecraft mission.

Chapter 3 focuses on the migration of Basilisk-developed flight algorithms into commercial processors, in particular, the Raspberry Pi. This chapter emphasizes the rising interest on using commercial processors for flight applications. In addition, the challenges associated into porting the Basilisk architecture into the Raspberry Pi commercial hardware are explained. One of the most important parts of this chapter is the demonstration of the very first distributed Basilisk simulation. The distributed nature of the simulation implies that the flight algorithms and the spacecraft physical simulation run on separate computing platforms, which is a key step towards more realistic, flight-like testing. In addition, the use of a peer-to-peer communication router that enables TCP communication between a client and a server is introduced. This communication router is, actually, an initial milestone towards the development of the flexible and multi-propose Black Lion communication architecture (one of the core tools developed within the scope of this thesis).

Chapter 4 describes the transition of the flight application from Basilisk into the core Flight System (cFS) middleware. While Basilisk is a flexible desktop environment that becomes extremely handy for prototyping and performing rapid testing of flight algorithms, the cFS is a widely-used middleware layer for space applications that ensures portability of the flight application among different targets (i.e. flight processor boards and RTOS). The most interesting part of the migration process is the use of the **AutoSetter**, a Python tool that has been developed for the specific purposes of translating the Python portion of the Basilisk flight architecture into C code. The resulting C code, which is minimal and completely human-readable, is generated through Python's

introspection capabilities. Such transition mechanism is generally applicable to any desktop testbed that, similar to Basilisk, leverages the use of Python for wrapping underlying C/C++ code. The generated C code plus the original flight algorithm source code are then compiled together into a pure-C cFS FSW application that can be embedded into an emulated processor board.

Chapter 5 describes all the technical challenges entailed in assembling an emulated flat-sat with unprecedented level of fidelity. Using this emulation, it is possible to test the performance of an embedded FSW application in a distributed and flight-like manner. There are two main tasks/aspects that have been critical to the successful implementation of the flat-sat emulation: 1) development of the Black Lion communication architecture and 2) modeling of FPGA registers and accompanying avionic hardware models. Emulated flat-sat testing through Black Lion has proved to be an extremely cost-effective means of performing system-wide testing early on in the mission's program, alleviating schedule constraints by using software models only. In addition, the flexibility of Black Lion has allowed running fault detection tests that could not be executed in any other testbed. In turn, the modeling of FPGA registers and avionic components stands out for its high-level of fidelity. In addition to performing complex operations on FSW packets, these models also replicate hardware interrupts, making FSW believe it is running on hardware –although it is actually a software emulation. While the presented avionic models and packet handlers are mission specific, the register space with its readers and writers constitutes a generic framework that can be applied to any FSW application: these registers have been designed and implemented in an abstract manner where high-level handlers can be customized to satisfy particular mission needs.

Chapter 6 presents an initial feasibility analysis for running Basilisk-developed flight algorithms within the modern and promising MicroPython. This analysis is accompanied by the development of another Python-based introspection tool that facilitates the migration across environments: the **AutoWrapper**. In the first proof of concept presented in this chapter, the potential of MicroPython as a middleware for space applications already shows and certain advantages of MicroPython over cFS can be drawn in terms of: migration effort, FSW states accessibility and application portability.

Chapter 7 is aimed at analyzing the suitability of the Basilisk-MicroPython system for constrained environments with reduced resources and real-time determinism needs. This chapter is divided into two main sections. The first section sheds light on the challenging steps required to port MicroPython to LEON platforms. Whereas this endeavour has also been accomplished by ESA[33], their software repository is not open to access; therefore, it seemed crucial to report and document the lessons learnt. Within the scope of this thesis, it was desired to port not only MicroPython to LEON but also the MicroPython C++ Wrapper and the Basilisk flight modules. However, the MicroPython C++ wrapper library has many dependencies on advanced C++ functionality, which is not easily supported by embedded toolchains like RTEMS-LEON. Suggested future work is to collaborate with the MicroPython C++ Wrapper project to make it suitable for 32-bit platforms. The second section of Chapter 7 profiles the use of resources (RAM, ROM and CPU) demanded by a sample Basilisk-MicroPython FSW application. In addition, the FSW scripts that are core to all Basilisk simulations are optimized to avoid heap memory usage in the control loop when the garbage collector (GC) is not present. Being able to execute the Basilisk-MicroPython system without the GC is a key capability towards making the system flight-like. With increasing small-sat missions using commercial-off-the-shelf hardware and software solutions for flight exploration, running a Basilisk-MicroPython FSW application on top of Unix in flight could be a very interesting option.

Potential future work that could be built upon all the material presented is the following:

- (1) Enhancement of Black Lion to support multiple time constraints. With the current “tick-tock” synchronization mechanism (see Section 5.2.2), the overall simulation speed is driven by the slowest of the components. Therefore, only the time requirement of the slowest component can be satisfied. This limitation is not a problem as long as the components either present the same timing constraint (e.g. real time) or have adjustable execution speeds.
- (2) Investigation and/or development of alternatives to QEMU for emulating flight processor



boards. The rational behind this task is to avoid the main caveats of QEMU, which is mostly written in C (and therefore not object oriented nor modern), it is not user friendly, it has a steep learning curve and, in addition, its execution speed for the SPARC emulation is slower than real time (which strongly limits the amount of testing that can be accomplished).

- (3) Generalization of the FSW snorkels and avionic hardware models used for emulated flat-sat testing (see Section 5.3). While the idea of an FPGA register space modeled as a memory map for the input and output of raw binary data constitutes a generic framework that can be applied to any FSW application, the specific handlers (i.e. snorkels) and avionic hardware models that have been implemented are actually mission specific. Yet, because many spacecraft have similar avionic systems, it is left for future work to generalize the aforementioned models such that they can be customized through initialization/configuration rather than containing mission-specific source code.
- (4) Automatization of the building process of the Basilisk-MicroPython system. Although guidelines for building MicroPython together with Basilisk FSW modules through the MicroPython C++ Wrapper are provided in Appendix D, this process is rather involved. As future work, it would be ideal to find a way to deliver the system in a single and complete bundle that can be directly customized for user-specific applications without having to worry about integration and joint compilation of the different software tools (i.e. Basilisk, MicroPython and the MicroPython C++ Wrapper).
- (5) Collaboration with the developers of the MicroPython C++ Wrapper library in order to target 32-bit ports. As explained in Section 7.1.3, after all the work of porting the standalone MicroPython to LEON platforms, it was not possible to get the MicroPython C++ Wrapper to build properly for LEON nor for any other 32-bit target platform. The reason behind this limitation is that the MicroPython C++ Wrapper has many dependencies on advanced C++ functionality, which is not easily supported by embedded toolchains like RTEMS-LEON and GNU ARM. As a matter of fact, the MicroPython C++ Wrapper

library, which acts as the glue between MicroPython and Basilisk, is still a project in beta stage and it has only been tested for Unix and Windows in 64-bit targets. For future applications, it would be desirable to collaborate with the developers of the MicroPython C++ Wrapper tool to include real-time operating systems and 32-bit platforms among their tested targets.

- (6) Run and test the Basilisk-MicroPython system on a constrained Unix environment that operates in real time. Recalling from Section 7.2.4, this could be achieved by applying the real-time kernel patch for Linux.

## Bibliography

- [1] John Alcorn, Hanspeter Schaub, Scott Piggott, and Daniel Kubitschek. Simulating attitude actuation options using the Basilisk astrodynamics software architecture. In 67th International Astronautical Congress, Guadalajara, Mexico, Sept. 26–30 2016.
- [2] C. Allard, M. Diaz-Ramos, and H. Schaub. Spacecraft dynamics integrating hinged solar panels and lumped-mass fuel slosh model. In AIAA SPACE, Long Beach, CA, Sep. 13–16 2016.
- [3] C. Allard, H. Schaub, and S. Piggott. General hinged solar panel dynamics approximating first-order spacecraft flexing. In AAS Guidance and Control Conference, Breckenridge, CO, Feb. 5–10 2016.
- [4] Andoni Arregi and Fabian Schriever. Numerical reproducibility for model-based software-engineering. In DASIA, 2019.
- [5] S. Blanchette. Giant slayer: Will you let software be David to your Goliath system? Journal of Aerospace Information Systems, 13(10):407–417, 2016.
- [6] Robert Bocchino, Timothy Canham, Garth Watney, Leonard Reder, and Jeffrey Levison. F prime: An open-source framework for small-scale flight software systems. In AIAA/USU Conference on Small Satellites, 2018.
- [7] Mike Briggs, Nathaniel Benz, and Douglas Forman. Simulation-centric model-based development for spacecraft and small launch vehicles. In 32nd Space Symposium, Colorado Springs, Colorado, April 11–12 2016.
- [8] J. Busa, E. Braunstein, R. Brunet, R. Grace, T. Vu, R. Brown, and W. Dwyer. Timeliner: automating procedures on the ISS. In SpaceOps, 2002.
- [9] Stephan Busch, Philip Bangert, Slavi Dombrovski, and Klaus Schilling. UWE-3, in-orbit performance and lessons learned of a modular and flexible satellite bus for future pico-satellite formations. In Acta Astronautica, volume 117, pages 73–89, December 2015.
- [10] Jonathan Cameron, Abhinandan Jain, Burkhardt Dan, Erik Bailey, J Balaram, Eugene Bonfiglio, Havard Grip, Mark Ivanov, and Evgeniy Sklyanskiy. DSEDS: Multi-mission flight dynamics simulator for NASA missions. AIAA Space 2016, (September):1–18, 2016.
- [11] CCSDS Secretariat, , Office of Space Communication . CCSDS recommendation for space packet protocol. Technical report, National Aeronautics and Space Administration, Washington, DC, September 2003.

- [12] M. Cols-Margenet, H. Schaub, and S. Piggott. Sequentially distributed attitude guidance across a spacecraft formation. In International Workshop on Satellite Constellations and Formation Flying, University of Strathclyde, Glasgow, Scotland, July 2019.
- [13] Mar Cols Margenet, Patrick Kenneally, Hanspeter Schaub, and Scott Piggott. Distributed simulation of heterogeneous mission subsystems through the Black Lion framework. Submitted to AIAA Journal of Aerospace Information Systems, 2019.
- [14] Mar Cols Margenet, Patrick W. Kenneally, Hanspeter Schaub, and Scott Piggott. Simulation of heterogeneous spacecraft and mission components through the Black Lion framework. In John L. Junkins Dynamical Systems Symposium, number 7, College Station, TX, May 20–21 2018.
- [15] Mar Cols Margenet, Hanspeter Schaub, and Scott Piggott. An end-to-end FSW development approach using Micropython and the Basilisk software testbed. In DASIA (Data Systems In Aerospace), Torremolinos, Spain, June 4–6 2019.
- [16] Mar Cols Margenet, Hanspeter Schaub, and Scott Piggott. Modular attitude guidance development using the Basilisk software framework. In AIAA/AAS Astrodynamics Specialist Conference, Sept. 12–15 2016.
- [17] Mar Cols Margenet, Hanspeter Schaub, and Scott Piggott. Modular platform for hardware-in-the-loop testing of autonomous flight algorithms. In International Symposium on Space Flight Dynamics, Matsuyama-Ehime, Japan, June 3–9 2017.
- [18] Mar Cols-Margenet, Hanspeter Schaub, and Scott Piggott. Modular attitude guidance: Generating rotational reference motions for distinct mission profiles. AIAA Journal of Aerospace Information Systems, 15(6):335–352, June 2018.
- [19] Mar Cols Margenet, Hanspeter Schaub, and Scott Piggott. Flight software development, migration and testing in desktop and embedded environments. Submitted to AIAA Journal of Aerospace Information Systems, 2019.
- [20] Mar Cols Margenet, Hanspeter Schaub, and Scott Piggott. Avionics hardware modeling and embedded flight software testing in an emulated flat-sat. In AAS Guidance, Navigation and Control Conference, Breckenridge, CO, Jan. 30–Feb. 5 2020.
- [21] Alan Cudmore. NASA/GSFC’s flight software architecture: core Flight Executive and core Flight System. In Flight Software Workshop, Johns Hopkins University Applied Physics Laboratory, MD, 2011.
- [22] Alan Cudmore. Pi-sat: A low cost small satellite and distributed spacecraft mission system test platform. Technical report, NASA Goddard Space Flight Center, Greenbelt, MD, September 9 2015.
- [23] John Cuseo. STK/SOLIS and STK/ODYSSY flight software: Supporting the entire spacecraft lifecycle. In 2011 Workshops on Spacecraft Flight Software, Johns Hopkins University Applied Physics Laboratory, Laurel, MD, October 19-21 2011.
- [24] Jean de Lafontaine, Jeroen Buijs, Pierrik Vuilleumier, Pieter Van den Braembussche, and Karim Mellab. Development of the PROBA attitude control and navigation software. 2000.

- [25] JR García-Blanco, Beatriz Lacruz-Alcaraz, Nuno Santos, and Daniel Silveira. Increasing representativeness of SIL VV simulators. In Dasia, 2019.
- [26] Damien George, David Sanchez de la Llana, and Tiago Jorge. Porting of Micropython to LEON platforms. Technical report, George Robotics Ltd. and ESA ESTEC, 2016.
- [27] Christopher Grasso. The fully programmable spacecraft: procedural sequencing for JPL deep space missions using VML (virtual machine language). In Aerospace Conference, 2002.
- [28] Matthew Grubb, Justin Morris, Scott Zemerick, and John Lucas. NASA operational simulator for small satellites (NOS3): Tools for software-based validation and verification of small satellites. In Proceedings of the AIAA/USU Conference on Small Satellites, Logan, Utah, August 2016.
- [29] Andrew Keys, Michael Watson, Donald Frazier, James Adams, Michael Johnson, and Elizabeth Kolawa. High performance, radiation-hardened electronics for space environments. In 5th International Planetary Probes Workshop, Bordeaux, France, June 28 2007.
- [30] Thomas Laroche, Pierre Denis, Paul Parisi, Damien George, David Sanchez de la Llana, and Thanassis Tsiodras. Micropython virtual machine for on board control procedures. In Dasia, 2018.
- [31] D.S. Lauretta. OSIRIS-REx Asteroid Sample-Return Mission, volume Handbook of Cosmic Hazards and Planetary Defense. Springer, 2015.
- [32] Christopher Lim and Abhinandan Jain. Dshell++: A component based, reusable space system simulation framework. In SMC-IT, 2009.
- [33] George Robotics Limited. Porting of Micropython to LEON platforms. Technical report, ESA contract, June 2017.
- [34] Mark Mangieri and Jason Vice. Kedalion: NASA’s adaptable and agile hardware/software integration and test lab. In AIAA SPACE, Long Beach, CA, 2011.
- [35] Robert Martin. Clean Architecture: A craftsman’s Guide to Software Structure and Design. Prentice Hall Press, 2017.
- [36] David McComas. NASA/GSFC’s flight software core Flight System. In Flight Software Workshop, San Antonio, TX, Nov. 7–9 2012.
- [37] Ryan Odegard, Joel Henry, Zoran Milenkovic, and Michael Buttacoli. Model-based GNC simulation and flight software development for Orion missions beyond LEO. In IEEE Aerospace Conference, Big Sky, Montana, 2014.
- [38] John Penn and Alexander Lin. The Trick simulation toolkit: A NASA open-source framework for running time based physics models. In AIAA Modeling and Simulation Technologies Conference (Sci-Tech), San Diego, CA, 2016.
- [39] Scott Piggott, John Alcorn, Mar Cols Margenet, Patrick W. Kenneally, and Hanspeter Schaub. Flight software development through Python. In 2016 Workshop on Spacecraft Flight Software, JPL, California, Dec. 13–15 2016.

- [40] RTEMS Project and Contributors. RTEMS user manual. Technical Report 5.a23b1fb, 17th May 2020.
- [41] H. L. Rarick, S. H. Godfrey, J. C. Kelly and R. T. Crumbley, and J. M. Wilf. Nasa software engineering benchmarking study. SP 2013-604, NASA, May 2013.
- [42] Hanspeter Schaub and John L. Junkins. Analytical Mechanics of Space Systems. AIAA Education Series, Reston, VA, 4th edition, 2018.
- [43] Peter Z. Schulte and David A. Spencer. Development of an integrated spacecraft guidance, navigation, and control subsystem for automated proximity operations. October 2014.
- [44] Jonathon Smith, William Taber, Theodore Drain, Scott Evans, James Evans, Michelle Guevara, William Schulze, Richard Sunseri, and Hsi-Cheng Wu. MONTE Python for deep space navigation. In Proceedings of the 15th Python in Science Conference (SCIPY), 2016.
- [45] Andrew Turner. An open-source, extensible spacecraft simulation and modeling environment framework. PhD thesis, Citeseer, 2003.
- [46] Chris Leger Vandi Verma. SSim: NASA Mars rover robotics flight software simulation. In IEEE Aerospace Conference, 2019.
- [47] V. Verma, A. Jónsson, C. Pasareanu, and M. Iatauro. Universal executive and PLEXIL: Engine and language for robust spacecraft control and operations. In AIAA Space, 2006.
- [48] Daniel Violette. Arduino/Raspberry Pi: Hobbyist hardware and radiation total dose degradation. In EEE Parts for Small Missions, Greenbelt, MD, September 10-11 2014.
- [49] J. Wood, M. Cols-Margenet, P. Kenneally, H. Schaub, and S. Piggott. Flexible Basilisk astrodynamics visualization software using the Unity rendering engine. In AAS Guidance and Control Conference, Breckenridge, CO, February 2–7 2018.
- [50] Scott A. Zemerick, Justin R. Morris, and Brandon T. Bailey. NASA operational simulator (NOS) for V and V of complex systems. 875205(May 2013):875205, 2013.

## Appendix A

### Modular Attitude Guidance

This appendix is aimed to support to the work presented in Chapter 2.

#### A.1 Attitude Control MRP Feedback

This section is related to controls rather than guidance and, therefore, it is not a core part of the work in Section 2.2. Yet, it is relevant to allow replication of the numerical simulations presented in Section 2.2. Seeing the control equations presented here is also useful to realize how the final output of the attitude reference chain is used downstream in the FSW process. Figure A.1

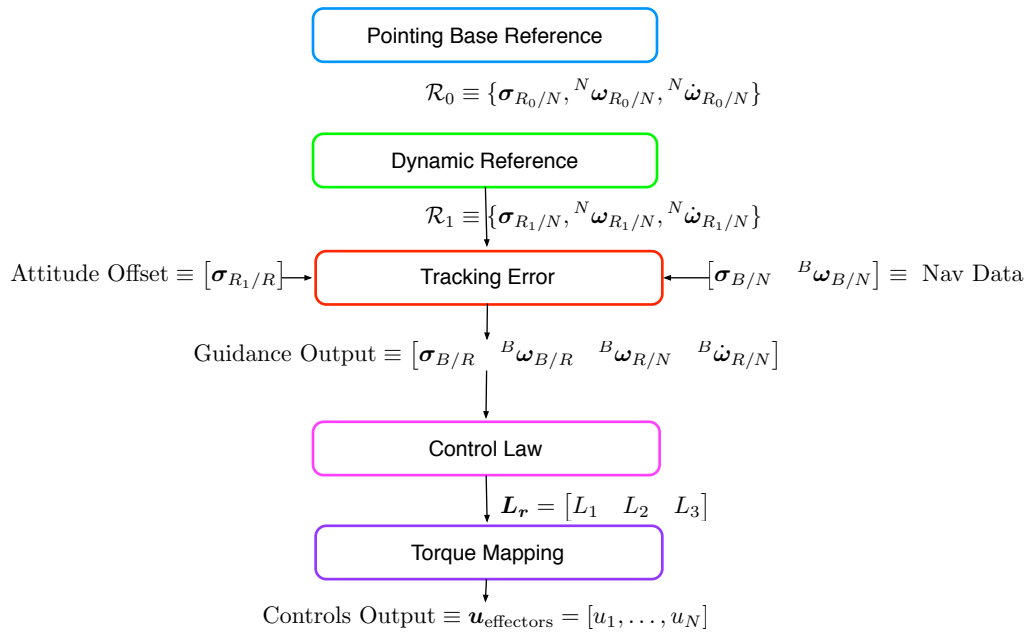


Figure A.1: Flow between Guidance and Control Blocks

depicts the flow from the attitude guidance block to the attitude control block. The equations outlined next belong to the algorithms that compute a control torque (i.e. “Control Law” module in Fig. A.1) and map it to the set of chosen spacecraft actuators (i.e. “Torque Mapping” module).

In the numerical simulations presented in Section 2.2, a rigid spacecraft with  $N = 4$  reaction wheels (RWs) is modeled. The associated differential equations of motion (EOM) are the following:

$$[I]\dot{\boldsymbol{\omega}}_{B/N} = -[\tilde{\boldsymbol{\omega}}_{B/N}]([I]\boldsymbol{\omega}_{B/N} + [G_s]\mathbf{h}_s) - [G_s]\mathbf{u}_s + \mathbf{L} \quad (\text{A.1})$$

where  $[I]$  is the spacecraft inertia tensor,  $\mathbf{L}$  is an external torque and  $\mathbf{u}_s$  is the set of RW motor torques. The RW spin axis are defined in the  $3 \times N$  projection matrix

$$[G_s] = \begin{bmatrix} \hat{\mathbf{g}}_{s_1} & \cdots & \hat{\mathbf{g}}_{s_N} \end{bmatrix} \quad (\text{A.2})$$

with  $\hat{\mathbf{g}}_{s_i}$  being the  $i^{\text{th}}$  RW spin axis. The  $N \times 1$  RW inertial angular momentum matrix  $\mathbf{h}_s$  is

$$\mathbf{h}_s = \begin{bmatrix} J_{s_1}(\boldsymbol{\omega}_{B/N} \cdot \hat{\mathbf{g}}_{s_1} + \Omega_1) \\ \vdots \\ J_{s_N}(\boldsymbol{\omega}_{B/N} \cdot \hat{\mathbf{g}}_{s_N} + \Omega_N) \end{bmatrix} \quad (\text{A.3})$$

where  $J_{s_i}$  is the RW spin axis inertia.

Given the EOM in Eq. (A.1), the control law implemented is an MRP feedback law that is globally asymptotically stabilizing:

$$[G_s]\mathbf{u}_s = K\boldsymbol{\sigma}_{B/R} + [P]\boldsymbol{\omega}_{B/R} - \boldsymbol{\omega}_{R/N} \times ([I]\boldsymbol{\omega}_{B/N} + [G_s]\mathbf{h}_s) + [I](\boldsymbol{\omega}_{B/N} \times \boldsymbol{\omega}_{R/N} - \dot{\boldsymbol{\omega}}_{R/N}) + \mathbf{L} \quad (\text{A.4})$$

Here  $\mathbf{u}_s$  is the control torque being computed,  $K$  is the attitude error gain and  $[P]$  is the rate error gain matrix. The control block is fed with the guidance output, as illustrated in Fig. A.1. This information includes the MRP attitude error  $\boldsymbol{\sigma}_{B/R}$ , the body rate error  ${}^{\mathcal{B}}\boldsymbol{\omega}_{B/N}$ , the reference rate  ${}^{\mathcal{B}}\boldsymbol{\omega}_{R/N}$  and the reference inertial acceleration  ${}^{\mathcal{B}}\dot{\boldsymbol{\omega}}_{R/N}$ .

Note that the vectors and matrices in in Eq. (A.4) are expressed in body  $\mathcal{B}$ -frame components. This implies that controlling a spacecraft frame that is not the main body  $\mathcal{B}$ -frame (e.g. star tracker component frame,  $\mathcal{B}_c$ ), has an impact in the guidance-control sequence. A common approach is to



map the vector and tensors of Eq. (A.4) into the component body  $\mathcal{B}_c$ -frame aimed to be guided and controlled. In this work, a different strategy is proposed that is more efficient: the offset from the main body frame  $\mathcal{B}$  to the control frame  $\mathcal{B}_c$  is added to the generated reference. This function is performed as part of the “Attitude Tracking Error” (see Section 2.2 for more details).

On a final note, it is important to remark that the particular feedback law outlined in Eq. (A.4) is not critical to the presented work. Any other asymptotically stable attitude control law could be used instead without impacting the guidance results.

## A.2 Additional Base Pointing Modules

This section provides the mathematical derivations associated to the implementation of the following base pointing modules: inertial pointing, Hill-orbit pointing and velocity-orbit pointing. Both the inertial and orbit frame references are widely used and well documented, but the novelty here lies on the scheme upon which they are architected: through the modular stack and interface definition, base modules can be used in stand-alone mode or as the base of complex dynamic behaviors.

### A.2.1 Inertial Pointing

The inertial pointing module is the simplest one. Here the constant reference frame  $\mathcal{R}_0$  is in a fixed general orientation relative to the inertial frame  $\mathcal{N}$ . The desired inertial orientation is given through an input MRP set  $\sigma_{R_0/N}$ , while the reference frame rates and accelerations are internally set to zero.

$$\omega_{R_0/N} = \dot{\omega}_{R_0/N} = \mathbf{0} \quad (\text{A.5})$$

### A.2.2 Hill and Velocity Pointing

The Hill and velocity base modules are strongly related and therefore presented jointly. Assume the spacecraft is to align with the orbit Hill frame  $\mathcal{R}_0 \equiv \mathcal{H} : \{\hat{\mathbf{i}}_r, \hat{\mathbf{i}}_\theta, \hat{\mathbf{i}}_h\}$  or the velocity frame  $\mathcal{R}_0 \equiv \mathcal{V} : \{\hat{\mathbf{i}}_n, \hat{\mathbf{i}}_v, \hat{\mathbf{i}}_h\}$ . Both frames are completely defined by the position and velocity vectors

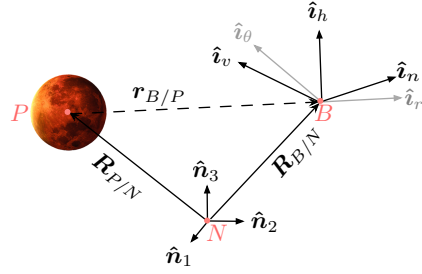


Figure A.2: Illustration of the Hill and Velocity Orbit Frames

of the spacecraft. The frames  $\mathcal{H}$  and  $\mathcal{V}$  are each conformed by their own right-handed set of axes where:  $\hat{\mathbf{i}}_r$  is the nadir axis pointing radially outward,  $\hat{\mathbf{i}}_v$  is tangent to the orbit and parallel to the velocity vector,  $\hat{\mathbf{i}}_h$  is defined normal to the orbital plane in the direction of the angular momentum, and finally  $\hat{\mathbf{i}}_\theta$  and  $\hat{\mathbf{i}}_n$  complete their respective right-handed triplet.

Figure A.2 illustrates the Hill and velocity frame orientations, each having their origin on the spacecraft location. The inertial frame  $\mathcal{N} : \{\hat{\mathbf{n}}_1, \hat{\mathbf{n}}_2, \hat{\mathbf{n}}_3\}$  is also depicted. The inertial position and velocity vectors of the spacecraft ( $\mathbf{R}_{B/N}, \mathbf{v}_{B/N}$ ) and the celestial body ( $\mathbf{R}_{P/N}, \mathbf{v}_{P/N}$ ) are the only variables assumed to be known by the module. The relative position of the spacecraft with respect to the planet  $\mathbf{r}_{B/P}$  and relative velocity  $\mathbf{v}_{B/P}$ , are obtained through

$$\mathbf{r}_{B/P} = \mathbf{R}_B - \mathbf{R}_P \quad (\text{A.6a})$$

$$\mathbf{v}_{B/P} = \mathbf{v}_B - \mathbf{v}_P \quad (\text{A.6b})$$

The Hill  $\mathcal{H}$  and velocity  $\mathcal{V}$  frame orientations with respect to the inertial frame  $\mathcal{N}$  are defined through the following Direction Cosine Matrices (DCMs):

$$[HN] = \begin{bmatrix} \mathcal{H}_{\hat{\mathbf{i}}_r}^T \\ \mathcal{H}_{\hat{\mathbf{i}}_\theta}^T \\ \mathcal{H}_{\hat{\mathbf{i}}_h}^T \end{bmatrix} \quad [VN] = \begin{bmatrix} \mathcal{V}_{\hat{\mathbf{i}}_n}^T \\ \mathcal{V}_{\hat{\mathbf{i}}_v}^T \\ \mathcal{V}_{\hat{\mathbf{i}}_h}^T \end{bmatrix} \quad (\text{A.7})$$

where the associated unit direction vectors are defined as:

$$\hat{\mathbf{i}}_r = \frac{\mathbf{r}_{B/P}}{|\mathbf{r}_{B/P}|} \quad (\text{A.8a})$$

$$\hat{\mathbf{i}}_v = \frac{\mathbf{v}_{B/P}}{|\mathbf{v}_{B/P}|} \quad (\text{A.8b})$$

$$\hat{\mathbf{i}}_h = \frac{\mathbf{r}_{B/P} \times \mathbf{v}_{B/P}}{|\mathbf{r}_{B/P} \times \mathbf{v}_{B/P}|} \quad (\text{A.8c})$$

$$\hat{\mathbf{i}}_\theta = \hat{\mathbf{i}}_h \times \hat{\mathbf{i}}_r \quad (\text{A.8d})$$

$$\hat{\mathbf{i}}_n = \hat{\mathbf{i}}_h \times \hat{\mathbf{i}}_v \quad (\text{A.8e})$$

The corresponding Hill orbit and velocity orbit MRP attitude sets can be directly obtained from their corresponding DCM (see Ref. [42] for details on the mapping):

$$[HN] \rightarrow \boldsymbol{\sigma}_{H/N} \equiv \boldsymbol{\sigma}_{R_0/N} \quad \text{or} \quad [VN] \rightarrow \boldsymbol{\sigma}_{V/N} \equiv \boldsymbol{\sigma}_{R_0/N} \quad (\text{A.9})$$

Next, the reference frame rates and accelerations are determined. In the case of the Hill frame  $\mathcal{H}$ , the angular rate of the reference is that of the orbital motion:

$$\boldsymbol{\omega}_{R_0/N} = \boldsymbol{\omega}_{H/N} = \dot{f} \hat{\mathbf{i}}_h \quad (\text{A.10a})$$

$$\dot{\boldsymbol{\omega}}_{R_0/N} = \dot{\boldsymbol{\omega}}_{H/N} = \ddot{f} \hat{\mathbf{i}}_h \quad (\text{A.10b})$$

Where  $f$  is the true anomaly angle, whose variation is expressed through the following general astrodynamics relation:

$$\dot{f} = \frac{\mathbf{r}_{B/P} \times \mathbf{v}_{B/P}}{\mathbf{r}_{B/P} \cdot \mathbf{r}_{B/P}} \quad (\text{A.11a})$$

$$\ddot{f} = -2 \frac{\mathbf{v}_{B/P} \cdot \hat{\mathbf{i}}_r}{|\mathbf{r}_{B/P}|} \dot{f} \quad (\text{A.11b})$$

The velocity frame  $\mathcal{V}$  orientation differs from the Hill frame orientation by a single-axis rotation of angle  $-\beta$  in the orbital plane about  $\hat{\mathbf{i}}_h$ . The DCM that maps from  $\mathcal{H}$  to  $\mathcal{V}$  is expressed in terms of the flight path angle  $\beta$  or the classical set of orbital elements as follows:

$$[VH] = \begin{bmatrix} \cos \beta & -\sin \beta & 0 \\ \sin \beta & \cos \beta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1 + e \cos f}{\sqrt{1 + e^2 + 2e \cos f}} & -\frac{e \sin f}{\sqrt{1 + e^2 + 2e \cos f}} & 0 \\ \frac{e \sin f}{\sqrt{1 + e^2 + 2e \cos f}} & \frac{1 + e \cos f}{\sqrt{1 + e^2 + 2e \cos f}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (\text{A.12})$$

The inertial angular rate and acceleration of the velocity frame  $\mathcal{V}$  are obtained by

$$\boldsymbol{\omega}_{V/N} = \boldsymbol{\omega}_{V/H} + \boldsymbol{\omega}_{H/N} \quad (\text{A.13a})$$

$$\dot{\boldsymbol{\omega}}_{V/N} = \dot{\boldsymbol{\omega}}_{V/H} + \dot{\boldsymbol{\omega}}_{H/N} \quad (\text{A.13b})$$

Where

$$\boldsymbol{\omega}_{V/H} = -\dot{\beta} \hat{\mathbf{i}}_h \quad (\text{A.14a})$$

$$\dot{\boldsymbol{\omega}}_{V/H} = -\ddot{\beta} \hat{\mathbf{i}}_h \quad (\text{A.14b})$$

An analytical expression for  $\beta$  is derived from Eq. (A.12), whose inertial time derivatives are:

$$\begin{aligned} \dot{\beta} &= \frac{e(e + \cos f)}{1 + e^2 + 2e \cos f} \dot{f} \\ \ddot{\beta} &= \frac{e(e + \cos f)}{1 + e^2 + 2e \cos f} \ddot{f} + \frac{e(e^2 - 1) \sin f}{(1 + e^2 + 2e \cos f)^2} \dot{f}^2 \end{aligned}$$

The velocity-frame base pointing module rates and accelerations are thus defined as

$$\boldsymbol{\omega}_{R_0/N} = \boldsymbol{\omega}_{V/N} = (\dot{f} - \dot{\beta}) \hat{\mathbf{i}}_h \quad (\text{A.16})$$

$$\dot{\boldsymbol{\omega}}_{R_0/N} = \dot{\boldsymbol{\omega}}_{V/N} = (\ddot{f} - \ddot{\beta}) \hat{\mathbf{i}}_h \quad (\text{A.17})$$

All the variables conforming the output structure of the orbit pointing modules have now been derived:  $\mathcal{R}_0 = \{\boldsymbol{\sigma}_{R_0/N}, {}^{\mathcal{N}}\boldsymbol{\omega}_{R_0/N}, {}^{\mathcal{N}}\dot{\boldsymbol{\omega}}_{R_0/N}\}$ .

## Appendix B

### Python-based Introspection Tools

#### B.1 Auto-setter

Listing 5 provides pseudo-code for the **AutoSetter** tool, showing its working mechanisms: looping through the C modules of each FSW task defined in a given Python scenario and parsing the modules' main algorithms as well as their variables and values. Note that, in order to handle nested structures and arrays, the variables need to be parsed recursively. Listing 5 also includes comments exemplifying the parsing of the vehicle configuration module (defined earlier in Listing 1 and initialized in Listing 2). The pseudo-code provided in Listing 5 focuses, particularly, on the introspection part of the tool. Once introspection is granted, C output can be generated by defining output templates. The template strategy is shown for the **AutoWrapper** in Listing 6 and Listing 7.

Listing 5: Pseudo-code for the AutoSetter.py

```
def main():  
    # FSW process containing GN&C tasks and modules  
    sim = FSW_process()  
  
    # Define the specific tasks to be handled by the autosetter.  
    # These correspond to a subset of all the tasks created in the sim  
    task_list = ["initialization", "sensor_read", "inertial_point", "feedback_control"]
```

```

# Run the autoserter

parse_modules(sim, task_list, output_file_name="c_setup_code")

# Look for the tasks in the sim that are also defined in the task_list.
# For each of these tasks, start looping through the modules contained in the task

def parse_modules(sim, task_list, output_file_name):

    source_file = open(output_file_name '.c', w+) # create a C source file

    for i in range(0, len(sim.tasks)):

        task = sim.tasks[i]

        if task.name in task_list:

            for j in range(0, len(task.models)):

                model = task.models[j] # refers to the Py object wrapping the C module
                # e.g. model = Basilisk.vehicleConfigSource.VehicleConfigStruct;
                # proxy of <Swig Object of type 'VehicleConfigStruct *' at address..

                model_tag = task.models[j].ModelTag # e.g. model_tag = "veh"

                autocode_variables(model, model_tag, source_file)

                autocode_methods(model, model_tag, source_file)

# This function looks for the specific names of the main methods in each module
# (e.g. SelfInit_vehConfig, CrossInit_vehConfig, Update_vehConfig and Reset_vehConfig)

def autocode_methods(model, model_tag, source_file):

    # Get the name of the C module underneath the Python/SWIG layers

    c_module_name = model.__module__ # Basilisk.vehicleConfigSource

    # Create the header line to include int the AutoSetter output file

    split_names = module.split('.') # e.g. [Basilisk, vehicleConfigSource]

    create_header(split_names) # e.g. '#include "Basilisk/vehicleConfigSource.h"'

    # Get the name of the C structure

    c_struct_name = str(type(model).__name__) # e.g. VehicleConfigStruct

    # Get the actual C module to perform some more introspection

    system_model = sys.modules[c_module_name]

```

```

# sys.modules above is a Python built-in dictionary

methods_list = dir(system_model)

# dir() provides the Python and C functions of the module
#e.g. methods_list = [elfInit_VehicleConfigData, _file_, ...]

for method_name in methods_list: # parse the methods

    method_object = eval('sys.modules["' + module + '"].' + method_name)

    if type(method_object).__name__ == "SwigPyObject":

        # then you have found the name of the module-specific methods
        # e.g. "SelfInit_VehicleConfig"

...

# Once the main method names in a module are known,
# they can be used in a C template to create part of the AutoSetter's output
...

return(...)

# This is a recursive method evaluating which (and how)
# variables are to be translated into C

def autocode_variables(model, model_tag, source_file):

    # e.g. model = Basilisk.vehicleConfigSource.VehicleConfigStruct;
    # e.g. model_tag = "veh"

    field_names = dir(module) # dir provides all module variables (C and Python)
    # e.g. field_names = ["CoM", "ISCPntB_B", "_class_", ..., "outputMsgName"]

    for k in range(0, len(field_names)):

        field = field_names[i] # e.g. field = "CoM"

        field_value = getattr(module, field) # e.g. field_value = [0.0, 0.0, 1.0]

        field_type = type(field_value).__name__ # e.g. field_type = "list"

        # Here we parse the current field and decide whether to auto-code it or not

        if <the field is a Python/SWIG built-in variable>:

            # these variables either have names starting with "__" or "this->"

```

```

    # or the field_type is "SwigPyObject" or "instancemethod"

    continue # ignore this variable and move on to the next one

elif field_type=="class":

    # recursion:

    nested_model_name = model_tag + field_name

    autocode_model(model=field_value, model_tag=nested_model_name, ...)

elif field_type=="list" and type(field_value[0])=="class":

    # recursion: it's a list of class/struct objects

    for <each class element in the field_value list>:

        autocode_model(...)

elif field_type=="list": # numeric lists

    # Translate non-zero elements of the list into C

    for m in range(0, len(field_value)):

        if field_value[m] != 0: # e.g. for CoM, field_value[1] = 1.0

            write_val_to_source(...) # Output example: veh.CoM[1] = 1.0;

elif field_type=="str": # character array

    write_str_to_source(...)

    #Output example: strcpy(veh_model_tag.outputMsgName,"adcs_config_data");

else: # non-array type. E.g. X=2.0, letter="a", etc.

    write_val_to_source(...)

```

## B.2 Auto-wrapper

As mentioned earlier, the objectives of the **AutoWrapper** tool are to generate a C++ wrapper class around each C FSW module and to generate the integration (or glue) code between MicroPython and the C++ classes. Listing 6 shows the C++ wrapper class that has been automatically generated for the vehicle configuration C module. This C++ wrapper class is described next. The C++ class contains the original C struct of the vehicle configuration module as a private variable. For reference, recall that this C struct is provided earlier in Listing 1. For each member in the C struct, a setter function and a getter function are created in the C++ wrapper class. The reason



for this is that the MicroPython C++ Wrapper library does not support direct interoperability between MicroPython and C++ class variables (only between MicroPython and C++ class functions). In addition, the C++ class in Listing 6 also contains callbacks to the main four C algorithms of the vehicle configuration module. Recall that these main calls (i.e. self-init, cross-init, update and reset) are used for execution in the Python desktop environment, and they will also be used for execution in MicroPython.

The combination of the C++ wrapper classes with their setters and getters and the MicroPython integration code, which is not shown in this manuscript, is equivalent to the functionality that SWIG provides out of the box for Python in a regular desktop environment. Although the functionality achieved is the same, the memory footprint with the MicroPython wrapping approach is drastically reduced. The **AutoWrapper** tool uses the same mechanism as the **AutoSetter** to figure out the variable and method names of the underlying C modules. Once introspection is granted, the code for the C++ wrapper class and the MicroPython integration patch can be generated through templates. Pseudo-code for the Python template describing how to generate the C++ wrapper class of a C module is provided in Listing 7.

Listing 6: AutoGenerated C++wrapper class (vehicleConfigSource.hpp)

```

#ifndef WRAP_vehConfigData_HPP
#define WRAP_vehConfigData_HPP

#include <iostream>

#include "utilities/linearAlgebra.h"
#include "_GeneralModuleFiles/sys_model.h"
#include "vehicleConfigData/vehicleConfigData.h"

class vehClass: public SysModel {
public:
    /* Constructor: memset 0 the C struct member variable */

```

```

vehClass(){ memset(&this->config_data, 0x0, sizeof(VehicleConfig));}

~ vehClass(){return;}

/* Callbacks to the C model's generic algorithms */

void SelfInit(){ SelfInit_vehicleConfig(&(this->config_data), ...); }
void CrossInit(){ CrossInit_vehicleConfig(&(this->config_data), ...); }
void UpdateState(uint64_t callTime){ Update_vehicleConfig(&(this->config_data), ...); }
void Reset(uint64_t callTime){ Reset_vehicleConfig(&(this->config_data), ...); }

/* Setter and getter for the "outputPropsName" variable of the C struct */
void Set_outputPropsName(std::string new_outputPropsName){
    memset(this->config_data.outputPropsName, '\0', sizeof(char) * MAX_STAT_MSG_LENGTH);
    strncpy(this->config_data.outputPropsName, new_outputPropsName.c_str(), ...);
}

std::string Get_outputPropsName() const{
    std::string local_outputPropsName(this->config_data.outputPropsName);
    return(local_outputPropsName);
}

/* Setter and getter for the inertia "ISCPntB_B" variable of the C struct */
void Set_ISCPntB_B(std::vector<double>new_ISCPntB_B) {
    m33Copy(RECAST3X3 new_ISCPntB_B.data(), RECAST3X3 this->config_data.ISCPntB_B);
}

std::vector<double> Get_ISCPntB_B() const {
    std::vector<double> local_ISCPntB_B(this->config_data.ISCPntB_B, ...);
    return (local_ISCPntB_B);
}

/* Setter and getter for the center of mass "CoM_B" variable of the C struct */
void Set_CoM_B(std::vector<double>new_CoM_B) {
    v3Copy(new_CoM_B.data(), this->config_data.CoM_B);
}

std::vector<double> Get_CoM_B() const {
    std::vector<double> local_CoM_B(this->config_data.CoM_B, ... );
}

```

```

        return (local_CoM_B);
    }

private:
    /* Define the model's C struct as a private member variable */
    VehicleConfig config_data;
};

#endif

```

Listing 7: Python pseudo-code for the C++ templates

```

class CppWrapClassTemplate(object):
    def __init__(self):
        ...

    def create_new_template(self, model_tag, header_line, c_struct_name, algs_dict, hpp_line):
        # e.g. model_tag = "veh"
        # hpp_line = '#include "vehicleConfigSource.h"'
        # c_struct_name = "VehicleConfigStruct"
        # algs_dict = ["CrossInit_vehicleConfig", "SelfInit_vehicleConfig", ...]

        self.current_model = model_tag

        # Create C++ class
        compile_def_name = 'WRAP_%s_HPP' % model_tag
        str_compile_def = '#ifndef ' + compile_def_name + '\n' + \
            '#define ' + compile_def_name + '\n\n'

        class_name = model_tag + "Class" // e.g. class_name = "vehClass"
        str_class = 'class %s: public SysModel {\n' % class_name + \
            # Constructor
            'public: \n' + \
            '\t%s(){ memset(&this->config_data, 0x0, sizeof(%s));}\n' % \
            (class_name, c_struct_name) + \
            # Destructor
            '\t~%s(){return;}\n' % class_name

```

```

...

str_c_data = 'private: \n' + \
             '\t%s config_data;' % c_struct_name

str_end = '\n}; \n\n#endif'

def add_string_property(self, field_name): #e.g. field_name = "outputPropsName"
    # Getter

    getter_str = "\tstd::string Get_%s() const{\n" % field_name + \
                 "\t\tstd::string local_%s(this->config_data.%s);\n" % \
                 (field_name, field_name) + \
                 "\t\treturn(local_%s);\n" % field_name + \
                 "\t}\n"

    # Setter (...)

    return (...)

```

## Appendix C

### Black Lion Data Transfer: ZMQ

Black Lion takes advantage of the ZeroMQ (ZMQ) Message Library<sup>1</sup> in order to transfer data between applications. ZeroMQ is a high-performance asynchronous messaging library aimed at use in distributed or concurrent applications. It allows the transport of data to be fast, reliable and protocol independent. The ZMQ interfaces are available in a wide range of programming languages, which can perfectly interact with each other.

#### C.1 Socket Patterns

In order to understand how the data transfer work, it is critical to first explain upfront the socket types and connection types used in the system. Two types of ZMQ socket patterns are used to transport data: the request-reply pattern and the publish-subscribe pattern. Within the Black Lion architecture, the publish-subscribe pattern is applied in two different flavors. , as described next:

**Request (REQ) - Reply (REP):** the Central Controller has a REQ socket for each node instantiated in the simulation. As the name indicates, REQ sockets are used to make requests, which demand a reply before the program can continue its execution. In turn, each node has a REP socket that receives and parses the request, performs the commanded task, and replies back indicating accomplishment.

---

<sup>1</sup> <http://zeromq.org>

**Publish (PUB) - Subscribe (FRONTEND SUB):** Every node has a PUB socket to share its own internal data through broadcasting or publications. In turn, the **Central Controller** has a frontend with a SUB socket that subscribes to the publications from all nodes.

**Publish (BACKEND PUB) - Subscribe (SUB):** Additionally, the **Central Controller** has a SUB-frontend and a PUB-backend. The messages received at the frontend are internally routed to the backend, which then re-publishes the data. In turn, each node has a SUB socket that subscribes to the messages of interest coming from the controller's backend.

The relationship between sockets just described is exemplified in Fig.C.1. The figure depicts the **Central Controller** in the middle and two sample nodes highlighted in magenta and blue. As shown in Fig. C.1, the sockets are encapsulated by the **Delegate** API.

## C.2 Connection Types

Now that the socket types are defined, the connections of these sockets to a given IP address and port are discussed. All the socket connections in the system fall into either one of these categories: static connection (i.e. **binding** type in ZMQ terms) or dynamic type (i.e. **connecting** type in ZMQ terms). The static connections are all associated to sockets in the **Central Controller**, while the dynamic connections are associated to the sockets in each of the nodes' **Delegate** API.

**Central Controller:** it is the only static piece in the network thanks to the frontend-backend

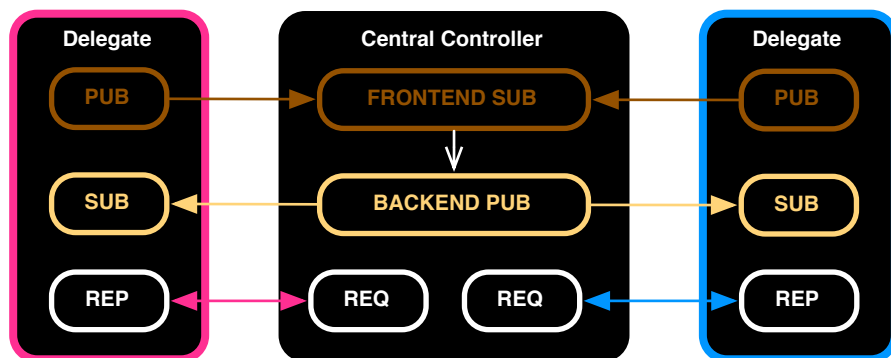


Figure C.1: Socket patterns between the **Central Controller** and the nodes' **Delegate**

(broker) approach. The controller acts as a server in the sense that it **binds** to a static IP address. With the same address, it uses a total of  $(2 + N)$  ports, where  $N$  is the number of nodes instantiated: One port for the frontend, one port for the backend, and a command port for each of the node-request sockets.

**Nodes' Delegate API:** through the **Delegate** API attached to each one of the nodes, the nodes become dynamic clients that can come and leave without bringing down the rest of the system. This dynamicity is reflected in the fact that the nodes only **connect** to an address and port, rather than bind.

Through the described strategy, the server (i.e. **Central Controller**) is always required and the clients are independent entities that do not intrinsically rely on each other. The use of ZMQ also allows all the connections to be protocol independent (TCP, IPC, etc.). The idea of socket binding (static nature) versus socket connecting (dynamic nature) is illustrated in the topology showcased in Fig. C.2. The figure also reflects the fact that there is only one static IP address in the entire Black Lion system and multiple ports are associated with that IP address. As before, the figure displays the **Central Controller** (server) in the middle and two sample nodes (clients) highlighted in magenta and blue colors.

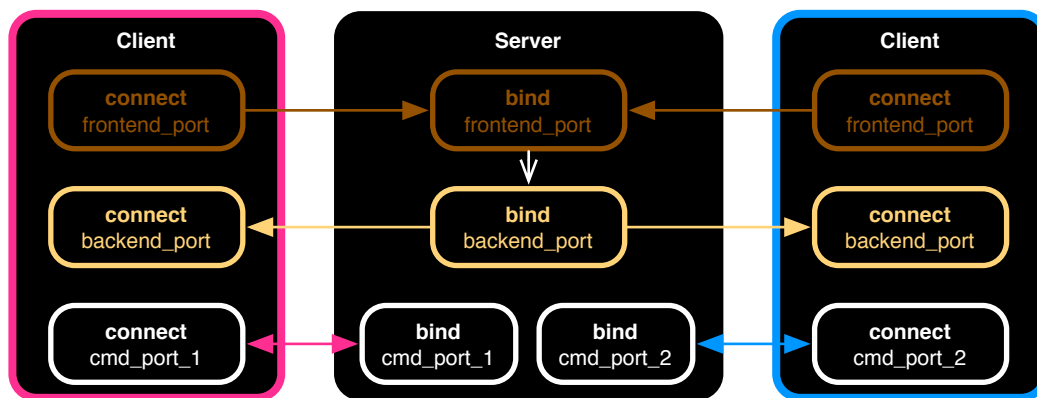


Figure C.2: Socket connections types: binding vs. connecting

### C.3 Controller Requests and Node-Delegate Replies

The **Central Controller** makes five types of requests to the **Delegate** of each node. Some of these request come in the form of multi-part ZMQ messages.

- (1) **“Initialize” request:** it is a multi-part message containing the “Initialize” signal, the controller’s frontend address and port and the controller’s backend address and port. The actions taken by the requested node are: self initialization, connect its pub-socket to the controller’s frontend and connect its sub-socket to the controller’s backend.
- (2) **“Provide Desired Message Names” request:** it instructs each node to report all the message names to which the node wishes to subscribe.
- (3) **“Match Message Names” request:** it is a multi-part message with the ”Match” signal and a list of all the message names for which the other nodes have asked. The requested node returns a reduced list with the message names for which it has found an internal match.
- (4) **“Tick” request:** it is used at every time-step of the SW-sim run for synchronization purposes. This request contains the time duration of the next time-step (i.e.  $\Delta t$ ). Once the requested node has accomplished all the tasks that must happen after a “Tick”, it sends back a “Tock” reply.
- (5) **“Finish” request:** it is a signal for the node to close the sockets, clean up and shut down.



## Appendix D

### Building the Basilisk-MicroPython FSW System for Unix

This chapter provides the reader with guidelines on how to build the Basilisk-MicroPython system for Unix. Such built exemplifies an effective manner of integrating custom C modules (e.g. FSW modules) into MicroPython by means of the MicroPython C++ Wrapper tool. The technical aspects described in this chapter aim to compensate for the fact that the Basilisk-MicroPython system cannot be easily made open source to the community as a single and complete bundle due to the use of multiple independent software repositories. While Basilisk, MicroPython and the MicroPython C++ Wrapper are all free software tools, their corresponding repositories are managed independently by different individuals and groups. This thesis has combined them together in order to come up with a flexible, lightweight and portable FSW system.

Dovetailing together Basilisk and MicroPython consists of, essentially, an integration process and a build process. The integration process is greatly simplified by the use of the **AutoWrapper** tool introduced earlier, which is now available as an open-source product through the Basilisk repository. While the **AutoWrapper** tool automatically generates the “glue” code between Basilisk and MicroPython, this integration code then needs to be added into the proper places for building –and this is, in part, what this chapter describes. Despite building the Basilisk-MicroPython system is not a simple process, the system can be replicated and built by following the guidelines provided next.

- (1) The first step consists on cloning the MicroPython repository and making sure that the **Unix port** builds successfully. Such built is done with only a few command lines that are

outlined [here](#). Then, the MicroPython executable can be launched from command line by typing `./micropython` within the Unix port. After having tested that MicroPython runs on Unix, it is suggested to clean the build –since it is going to be the MicroPython C++ Wrapper which builds the MicroPython executable next.

- (2) The second step consists on cloning the MicroPython C++ Wrapper repository and calling its `Makefile` to build MicroPython’s `Unix port` together with the C++ Wrapper, which is in turn linked as a static library.

To provide context and as explained in this [example](#), the core of the MicroPython C++ Wrapper library is its `module.cpp` file, which creates a MicroPython module that is named `upywraptest` and to which several C++ classes and functions are added. In order for this module to become actually importable from MicroPython, it is necessary to modify the `main.c` file in the MicroPython’s Unix port. In particular, there are three edits required around the `main_` call of the `main.c` file. Pseudo-code for the `main_` call is shown in Listing 8 and the edits correspond to lines 1, 2 and 6. The function of these edits is to declare and register the `upywraptest` module such that it can then be imported from MicroPython by simply typing `import upywraptest`. Then, all the C++ classes and functions implemented within this module can be used as if they were MicroPython objects.

With these edits of MicroPython’s main in place, the user can build the MicroPython executable together with the C++ Wrapper static library by `cd`-ing into the `micropython-wrap` directory and typing `make staticlib`.

- (3) The final step is customization, which involves: implementing your own C++ modules, adding them to the Wrapper’s `Makefile` to compile and link (in the same way that is currently done for `module.cpp`) and then editing the `module.cpp` file to define not only the `upywraptest` module but also your custom MicroPython module (e.g. `upywrap_fsw`). The idea is that within `upywrap_fsw` the different C++ FSW classes and functions that need to be available at the MicroPython layer are declared. For the case of the Basilisk FSW

modules, these declarations are described as “MicroPython integration patch” in Fig. 6.1 and they are generated automatically by the **AutoWrapper** tool. The generated integration code is meant to be added underneath the custom MicroPython module (i.e. `upywrap_fsw`) within the `module.cpp` file.

After customization, the system can be built again as described in Item (2).

Listing 8: Edits in MicroPython’s Unix main.c file

```
#include <py/objmodule.h> // line 1
extern mp_obj_module_t* init_upywraptest(); // line 2
MP_NOINLINE int main_(int argc, char **argv) { // line 3
    ...
    mp_init(); // line 5
    mp_module_register(qstr_from_str("upywraptest"), init_upywraptest()); // line 6
    ...
}
```

## Appendix E

### Benchmarking Heap Memory Usage

This section aims to provide reference memory benchmarks by comparing the heap memory consumption of different systems when running the same script. The different systems to be analyzed are the following:

- (1) Python 3
- (2) MicroPython with garbage collector (GC)
- (3) MicroPython without GC
- (4) Basilisk-MicroPython

The generic script to use is a sensor example extracted from Ref. [33]. Pseudo-code for this sensor example is presented in Listing 9. The script is conformed by two parts: initialization and execution (or control loop). The interesting aspect is that the control loop does not allocate heap memory: objects, classes and data structures are preallocated during initialization. Then, the control loop only uses a restricted set of constructs that do not require heap allocation. In MicroPython, the following operations are known to not require heap usage:

- Small integers, which are stored in object pointers.
- Functions, which use the C stack and do not need binding to be called.
- For loops over a range of integers.

Operations that would require heap usage are, for instance:

- Floating point number arithmetics.
- General iteration construct (other than iteration over integers).

The sensor example script in Listing 9 is deterministic in the sense that it does not allocate memory dynamically beyond initialization. Therefore, the heap memory consumed by the different systems when executing the main loop is supposed to be a flat line. Having said that, these flat lines will present offsets between each other and these offsets are precisely what reveals the performance of each system. The Massif<sup>1</sup> tool provided by Valgrind is used to profile the heap memory for each application.

Listing 9: Deterministic Sensor Example. Extracted from Ref. [33].

```
class Sensor:
    def __init__(self, ...): ...
    def sample(self): return(...)

class Indicator:
    def __init__(self, ...): ...
    def toggle(self): ...

def create():
    sensors = {"sensor_1": Sensor(), "sensor_2": Sensor()}
    indicators = {"indic_1": Indicator(), "indic_2": Indicator()}
    return sensors, indicators

def loop(sensors, indicators):
    for i in range(2000): # iteration over range of ints
        num = sensors["sensor_1"].sample()+["sensor_2"].sample()
        if num > 50: # integer condition
            indicators["indic_1"].toggle()
```

---

<sup>1</sup> <https://valgrind.org/docs/manual/ms-manual.html>

```

else: indicators["indic_2"].toggle()

def main():

    sensors, indicators = create() # preallocate objects

    loop(sensors, indicators) # control loop

```

## E.1 Sensor example in Python

Figure E.1 shows the heap memory consumed by Python 3 when running the script in Listing 9. This plot is obtained using Massif's visualizer and the time is represented in number of bytes

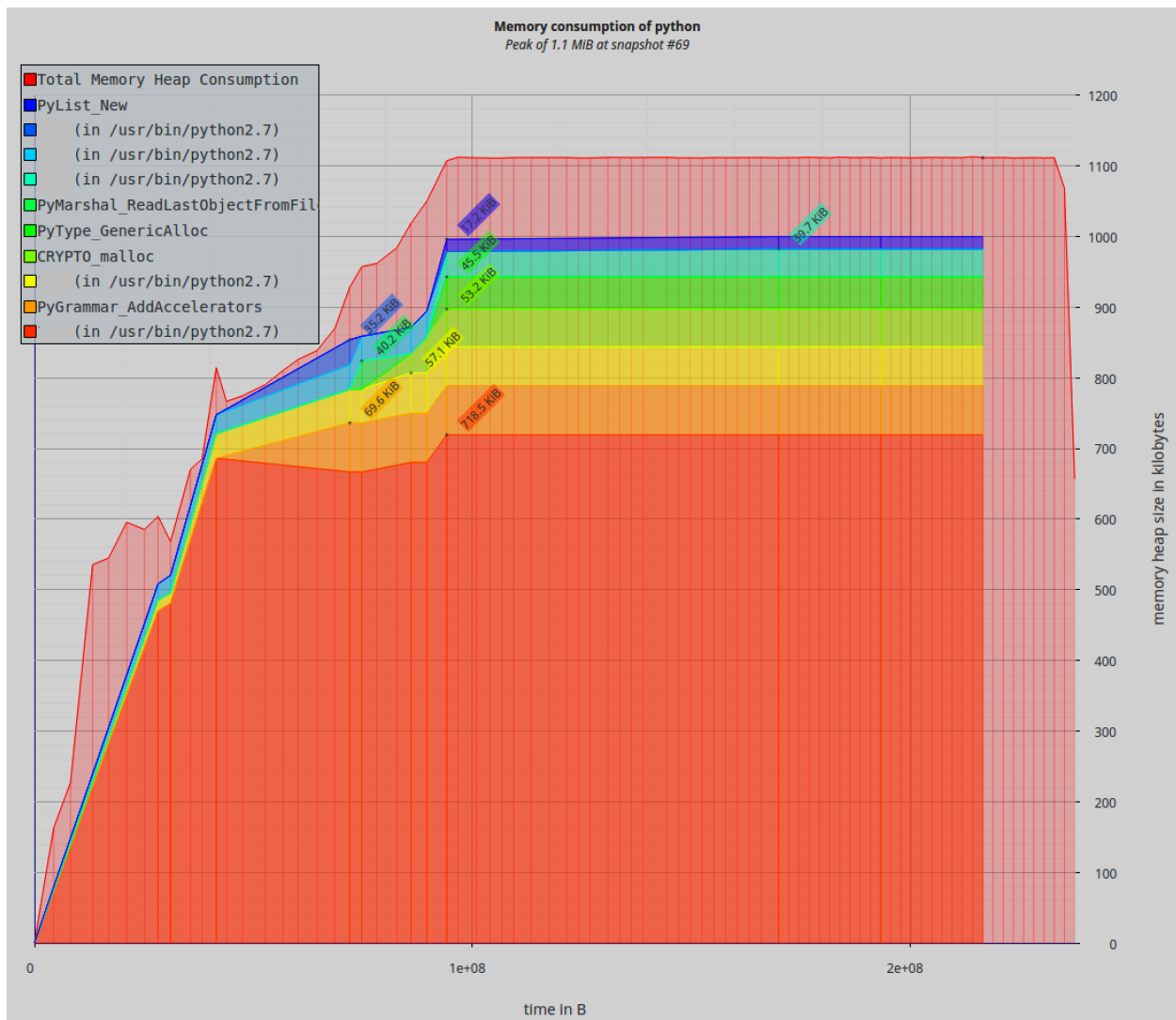


Figure E.1: Sensor example in Python 3

allocated/deallocated on the heap and stack (i.e. time in B). The peak heap consumption coincides with the flat consumption at 1.1 MebiByte (MiB).

It is relevant to mention that all Python 3 installations come with built-in garbage collector algorithms: a reference counting collector (which is fundamental to Python workings and cannot be disabled) and a cyclic garbage collector (i.e. the `gc` module, which is optional and can be enabled/disabled manually). For the execution profiled in Fig. E.1, garbage collector algorithms were present (i.e. built within Python 3) although not used because the program/script does not allocate memory dynamically.

In order to discuss results further, it is convenient to see how Python compares with MicroPython. The profiling results for MicroPython are presented next.

## E.2 Sensor example in MicroPython

In contrast to Python, MicroPython does not come as a complete bundle ready to be installed; instead, it is meant to be built for specific targets and customized for specific needs. On these lines, MicroPython provides many compile-time configuration options to fine tune the build of the MicroPython executable. One of these options is to build MicroPython without the GC. Next, the heap usage of MicroPython when running the script in Listing 9 is profiled in two different cases: when MicroPython is built with and without the GC respectively.

### E.2.1 MicroPython with Garbage Collector

Figure E.2 shows the heap memory usage of MicroPython, built with the GC, when running the sensor example. The heap usage is, as expected, constant and the consumption peak equals the flat value of the main loop (at 2.1 MiB). Surprisingly, MicroPython with GC demands 1MiB more than regular Python3. However, this is only because of the presence of the GC which, as seen next, is stored in RAM.

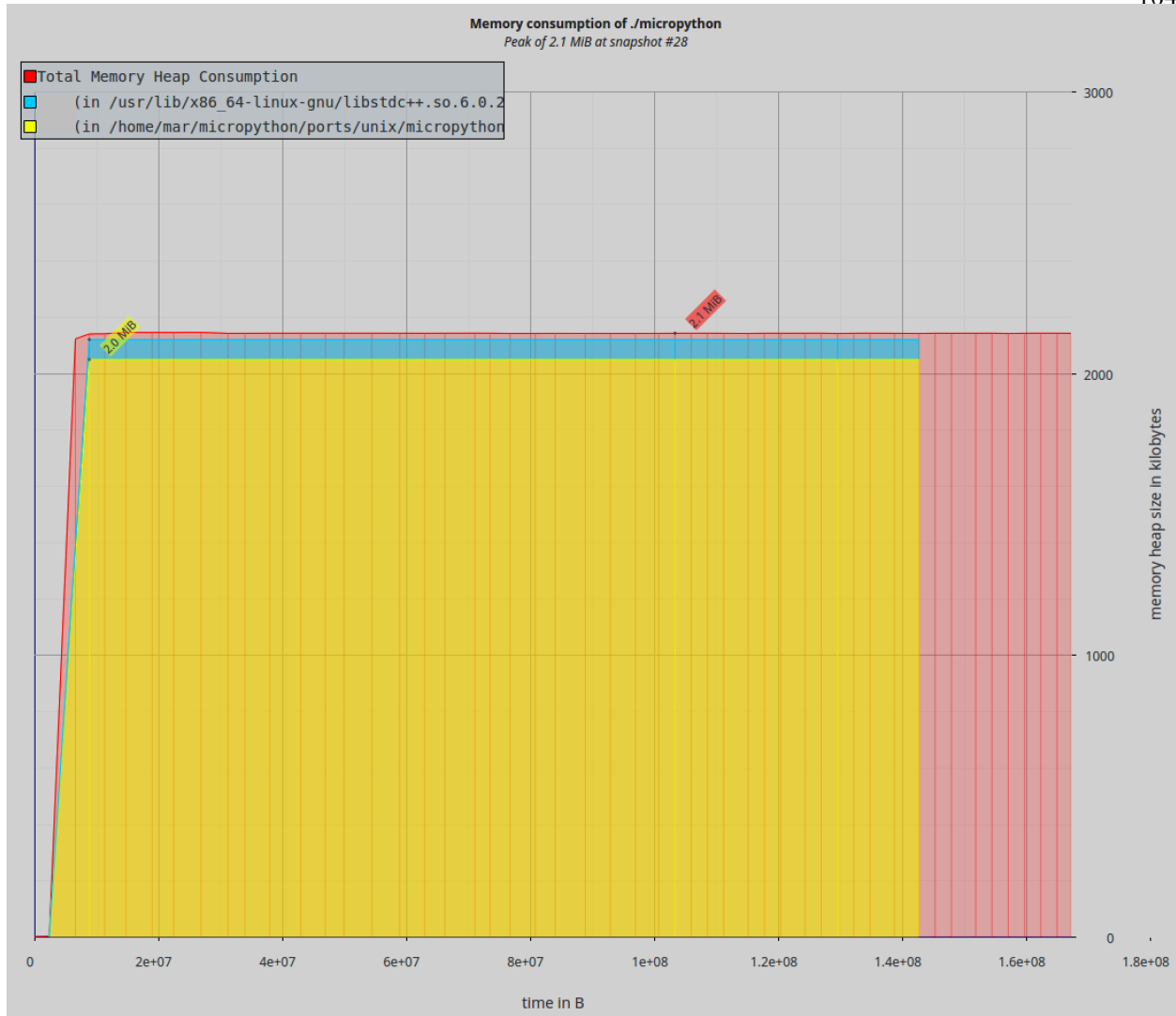


Figure E.2: Sensor example in MicroPython with GC

### E.2.2 MicroPython without Garbage Collector

Figure E.3 shows the heap memory usage of MicroPython, built without the GC, when running the same sensor example script. In this case the memory peak is 104.8 KiB, which happens at initialization. The memory consumption is flat during the control-loop execution and has a constant value of 57 KiB approximately. Comparing now the memory usage of Python 3 versus MicroPython without GC, the peak consumption is reduced by one order of magnitude.

The profile in Fig. E.3 also proves that, indeed, the script in Listing 9 does not allocate heap



memory in the control loop. If runtime objects were being created cyclically, the heap usage would increase proportionally to the number of cycles –since the GC is not present to free resources that run out of scope.

In MicroPython the GC is stored in RAM. Therefore, getting rid of it is a great way of optimizing the RAM usage, provided that certain rules are followed when it comes to writing programs in MicroPython.

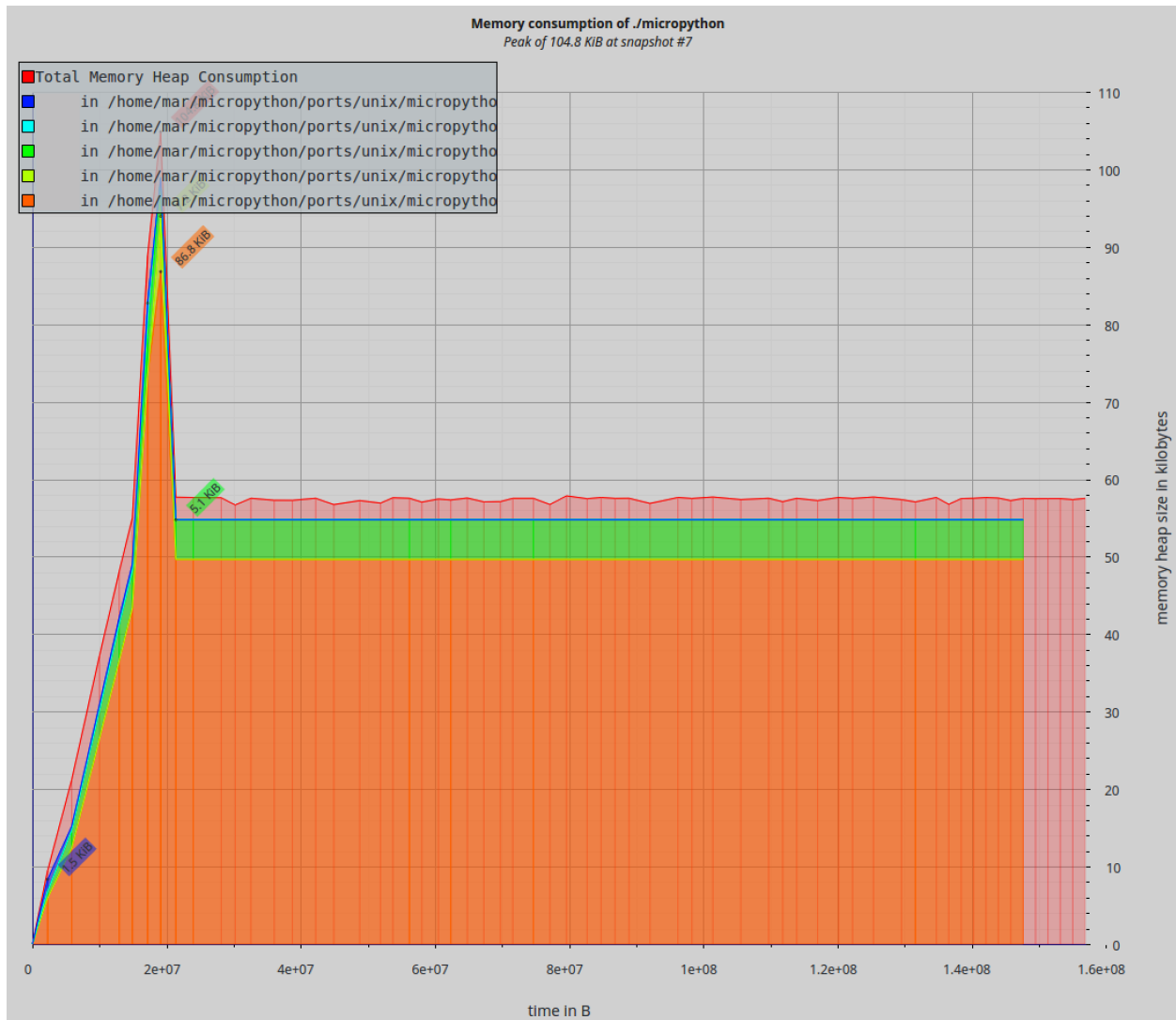


Figure E.3: Sensor example in MicroPython without GC

### E.3 Sensor example in Basilisk-MicroPython without GC

Seeing that building MicroPython with the GC supposes a considerable hit in RAM usage, the next question to address is: how much RAM memory is required to build MicroPython together with the Basilisk FSW modules and the MicroPython C++ Wrapper? Figure E.4 shows the memory consumed by Basilisk-MicroPython (without the GC) when running the example sensor script in Listing 9. Note that, in this case, the Basilisk FSW modules are present but they are not used. The peak memory consumption happens at initialization and has a value of 206.8 KiB, while the

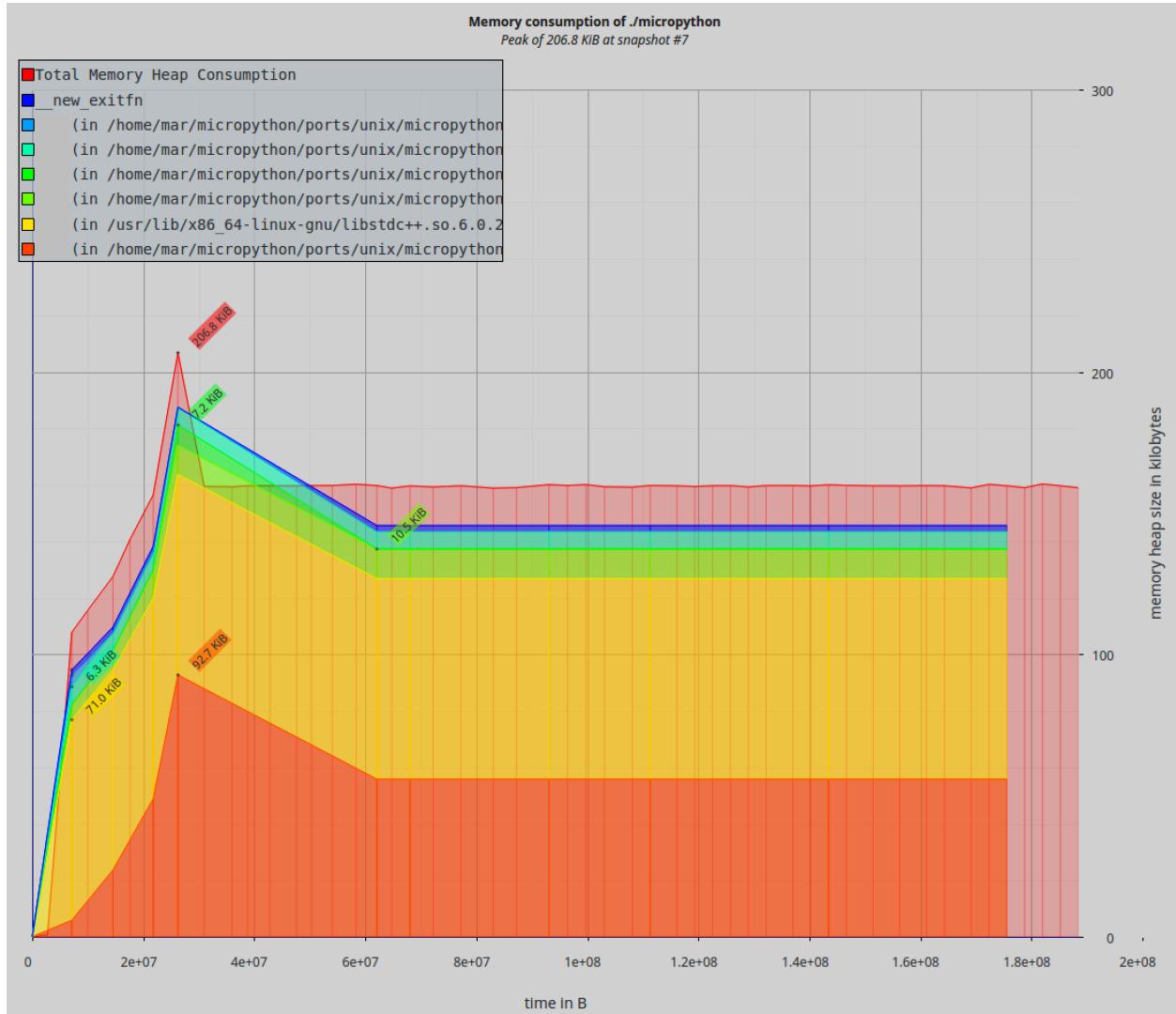


Figure E.4: Sensor example in Basilisk-MicroPython without GC

flat consumption in the control loop settles at 160 KiB approximately. Hence, building Basilisk together with MicroPython requires only 100 KiB of RAM.