POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico
di Torino

Master's Degree Thesis

Functional and extra functional simulation of a RISC-V based unmanned aerial vehicle through SystemC-AMS

Supervisors

Prof. Sara VINCO

Prof. Daniele Jahier PAGLIARI

Prof. Alessio BURELLO

Dr. Giovanni POLLO

Dr. Mohamed Amine HAMDI

Candidate

Pietro FURBATTO

October 2024

Summary

Nowadays, the rise of the RISC-V instruction set architecture (ISA) has led to an increasing interest towards the development of royalty-free and open-source products. Thanks to RISC-V, hardware architectures can be customised to suit specific product needs, offering a flexible and scalable approach that benefits both Original Equipment Manufacturer (OEM) producers and final consumers.

To design products, companies strongly rely on simulations to explore the various design choices and the possible flaws of a design, reducing both the development times and costs. Being a rather recent technology, however, there is still a lack of comprehensive simulation frameworks for RISC-V cores, especially for ones targeting both functional and extra-functional aspects such as power consumption.

In the landscape of functional simulation, GVSoC represents one of the most noteworthy options, allowing for a highly configurable and timing-accurate simulation for GAP8, a powerful RISC-V based IoT-oriented processor. GVSoC enables practical functional and performance analysis at the full-platform level, but lacks the support for any other external components and extra functional properties.

To overcome these shortcomings, the MESSY framework has recently been developed: by combining GVSoC and the expressive power of SystemC-AMS, this tool provides a scalable yet easily customizable solution to satisfy the need of a system-level simulation framework for embedded to industrial applications. Although the software has already reached a quite mature state, very few tests have been conducted to evaluate the effectiveness of the system in simulating properties of real, complex products.

This thesis thus focuses on modelling a complex robotic system within MESSY. To model the complex mechanics of a robotic system without impacting the complexity of simulation setup, MESSY has been connected to Webots, an open source application which provides a complete development environment to model, program and simulate robots and that can provide realistic sensor data to further enhance the realism of the simulation environment.

The target platform for this study is the GAP8-based Crazyflie 2.1 nano drone. Nano drones have recently received a lot of interest in the academic world due to their versatility and the widespread of on-edge Artificial Intelligence (AI). As

battery-powered devices, the interest in extra functional properties is extremely valuable, as simulations allow for estimating the impact on battery life of various control algorithms and the exploration of several hardware choices. Additionally, nano drones present unique challenges due to their complex control and the need for multiple sensors, making them ideal candidates for the purpose of this work.

All of these aspects have been integrated in a virtual platform that combines MESSY and Webots, in order to achieve a "digital twin" version of the Crazyflie drone that closely represents its architecture behaviour and power consumption.

The thesis successfully demonstrates the effectiveness of the system-level simulation capabilities offered by MESSY, along with its minimal impact on the Webots simulation times. Furthermore, the analyses of the impact on flight time of different batteries, control algorithms and environment conditions serve as an example to highlight some of the meaningful insights that can be gathered from such simulation framework, showcasing its potential for comprehensive performance evaluation and optimization.

Acknowledgements

Questa tesi rappresenta il culmine, ma anche il termine, della mia lunga carriera universitaria, oltre che da studente in generale. Un percorso come quello di tanti altri, certo, che però confesso ha avuto qualcosa di unico ed irripetibile. Guardandomi indietro, mi è impossibile non accorgermi di quante difficolà ho incontrato e superato, soprattuto grazie all'aiuto che ho ricevuto da tantissime persone, a ciascuna delle quali questo capitolo è dedicato.
Un grazie in primis a Giovanni, Amine, Daniele, Sara ed Alessio. La vostra costanza e il vostro interesse nella risucita di questa tesi è senza dubbio stato di enorme aiuto: avete reso tutto più leggero e stimolante, consentendomi di arrivare qui oggi con la consapevolezza di avere concluso questo percorso nel migliore dei modi. Un grazie a Matteo, compagno di tesi e amico da sempre. Oltre alle partite a Clash, ti sono debitore di parecchie dritte per questo lavoro... ma non solo.
Grazie alla mia famiglia. Un ENORME grazie a voi. Mamma, papà, Anna, Lorenzo e nonna: siete stati la mia bussola in ogni circostanza, e il vostro affetto e supporto è sempre stato fondamentale per la buona riuscita di questo e dei miei altri traguardi.
E un altrettanto gigantesco grazie a chi mi è stato vicino in questi ultimi anni.
Per primi ci siete voi, gli autodefiniti "amici di classe Z". Crupi, Gianci e Faggio, a voi devo davvero tanto di chi sono ora. Siamo stati assieme da quando eravamo ragazzini, e porterò per sempre con me le nostre risate e ogni avventura e riflessione che abbiamo condiviso. Un altro super grazie lo devo agli amici del mare: abbiamo trascorso di tutto, di bello e di meno bello, ma siete stati un punto saldo di questi ultimi 9 (si, davvero 9) anni e la vera anima di tutte le mie estati. Come non ringraziare poi Ale, Gabri, Bis e Parry: non ci sono riusciti la fine del liceo nè il covid a separarci, e mi auguro che la nostra amicizia possa durare ancora a lungo. Grazie anche a te Cassi, e ai nostri interminabili audio dove abbiamo condiviso le nostre cose più intime.
Infine, un grazie specifico lo devo forse soprattutto a voi, amici dell'uni. Eleonora, Mattias, Giorgio, Angelo, Fabrizio, Filippo, Elena, Samuel, Valentina e Jessica. Questi anni li avete davvero alleggeriti, e spero che, tra le fatiche dei progetti e le nostre uscite, ciascuno di voi conservi gli stessi bei ricordi che porterò con me. E, in particolare, grazie anche a voi, Peppe e Combe, per l'enorme aiuto che mi/ci

avete dato, e per la vostra risata tremendamente contagiosa.
Un enorme e sentito grazie va a tutte quelle persone con cui ho condiviso anni, giorni, o anche solo un istante di tutto questo percorso. Un gesto gentile, una chiacchierata piacevole, un'uscita per un caffè. Tanti di voi non li ho citati, di altri non so nemmeno il nome... ma avete davvero contribuito a rendere ciascuno di questi giorni più facile e speciale, facendomi affrontare ogni sfida con una carica in più.

Insomma, un grazie profondo a chi ci è stato. Non so cosa mi riserverà la vita e come continuerà la mia strada. Ma la forza che mi avete donato mi aiuterà a vivere le cose come ho fatto in questi anni: con tanta voglia di continuare e sempre con un semplice, ma sincero, sorriso.

Table of Contents

List of Tables IX

List of Figures X

Acronyms XIII

1 Introduction 1

2 Background 5
2.1 RISC-V . . . 5
2.1.1 PULP . . . 6
2.1.2 GAP8 . . . 7
2.2 System simulation for virtual prototyping . . . 8
2.2.1 Functional simulation . . . 9
2.2.2 Extra-functional simulation . . . 10
2.3 Inter-Process Communication through network sockets . . . 10
2.3.1 UNIX sockets . . . 12
2.4 Unmanned aerial vehicles . . . 13
2.4.1 Introduction to drones and nanodrones . . . 13
2.4.2 Drone controllers . . . 17
2.4.3 The Crazyflie 2.1 nanodrone . . . 19

3 Related works 25
3.1 Functional ISS simulation . . . 25
3.1.1 GVSoC . . . 26
3.1.2 SystemC . . . 27
3.2 System-level simulation . . . 28
3.2.1 SystemC-AMS . . . 28
3.2.2 MESSY . . . 29
3.3 Robotics simulation . . . 31
3.3.1 ROS . . . 31

3.3.2 Webots . . . 32
3.4 Drone power models . . . 33
3.4.1 Overview of theoretical models . . . 34
3.4.2 Final considerations . . . 36
3.4.3 Empirical model . . . 37

4 Methodologies 39
4.1 Establishing the connection between MESSY and Webots . . . 39
4.1.1 The VirtualConnector library . . . 39
4.2 Crazyflie architecture modelling in MESSY . . . 41
4.2.1 Sensors and SoC: camera and STM32 microprocessor . . . 45
4.2.2 PID controller . . . 53
4.2.3 Handling the simulation from the `main` . . . 59
4.2.4 General improvements . . . 60
4.3 GVSoC program . . . 64
4.4 The Webots controller . . . 68
4.5 Power modelling . . . 71
4.5.1 Sensors . . . 75
4.5.2 Power buses . . . 78
4.5.3 Energy consumption models . . . 78
4.5.4 Battery and battery converter . . . 80

5 Experimental Results 83
5.1 Scenario overview and power models comparison . . . 83
5.2 System-level simulation overhead and time offset . . . 87
5.3 Changing simulation parameters . . . 90
5.4 Testing different batteries . . . 93

6 Conclusions and future works 100

A Additional power models 102

B VirtualConnector class 103

C PID iteration function 105

D GVSoC program - Initial definitions and most important methods 108

E Webots controller - `main` and `sampleRun` methods 112

Bibliography 117

List of Tables

3.1 Summary of energy consumption models and their feasibility 37
3.2 Power consumption measurements for the Crazyflie drone 38

4.1 Camera Registers 47
4.2 STM32 Registers 49
4.3 STM32 control/status register commands 50
4.4 Parameters for D'Andrea model. 79
4.5 Parameters for Stolaroff model. 80

5.1 Simulation time measurements excluding MESSY (data in ms). . . 89
5.2 Simulation time measurements including MESSY (data in ms). . . 89
5.3 4 corners time measurements excluding MESSY (data in ms). . . . 91
5.4 4 corners time measurements including MESSY (data in ms). . . . 91

List of Figures

2.1 Overview of client and server operations in typical UNIX domain socket connection . . . 14
2.2 Growth trajectories in UAV research directions from January 2020 to December 2022 [28] . . . 15
2.3 PID controller scheme [32]. . . . 18
2.4 The Crazyflie 2.1 drone by Bitcraze . . . 20
2.5 Propellers direction of motion . . . 21
2.6 Useful terms about UAV motion . . . 24

3.1 Example of an LFS block. . . . 29

4.1 Visual representation of the MESSY default flow. . . . 46
4.2 Visual representation of the bus-camera-Webots interaction. . . . 48
4.3 Battery connector scheme. . . . 73
4.4 Motor schematics. . . . 73
4.5 Overview of the connection between battery and motors. . . . 73
4.6 STM32 pins. . . . 74
4.7 VCC reference circuit. . . . 74
4.8 Overview of the use of VCC and its generation. . . . 74
4.9 LPHD6520030 charge-voltage discharge graph [61]. . . . 81

5.1 Overview of Webots 3D environment. . . . 83
5.2 The drone at the beginning of the simulation. . . . 84
5.3 The drone passing a gate. . . . 84
5.4 Shots of the Crazyflie drone during the simulation. . . . 84
5.5 Power and battery information shown at the end of the simulation. 85
5.6 Comparison of the battery discharge curves. . . . 87
5.7 Power and battery information (empirical model) for 4 corners simulation. . . . 92
5.8 8 ms timestep. . . . 93
5.9 16 ms timestep. . . . 93

5.10 32 ms timestep. 93
5.11 Battery discharge curves for different timesteps. 93
5.12 UFX402525 battery discharge curves. 94
5.13 UFX 250 mAh battery results. 95
5.14 Cyclone 300 battery discharge curves. 96
5.15 Cyclone 300 mAh battery results. 97
5.16 LiPol 350 mAh battery results. 98
5.17 Comparison of the battery discharge with the three batteries. 99

Acronyms

AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AMS	Analog/Mixed-Signal
API	Application Programming Interface
ARM	Advanced RISC Machines
BLE	Bluetooth Low Energy
CAD	Computer-Aided Design
CARE	Continuous-time Algebraic Riccati Equation
CCW	Counter-Clockwise
CNN	Convolutional Neural Network
CMOS	Complementary Metal-Oxide-Semiconductor
CISC	Complex Instruction Set Computer
CPI	Camera Parallel Interface
CPU	Central Processing Unit
CW	Clockwise

DC	Direct Current
DMA	Direct Memory Access
DSE	Design Space Exploration
ELN	Electrical Linear Networks
EMI	Electromagnetic Interference
FC	Fabric Controller
FP	Floating Point
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
GPS	Global Positioning System
GUI	Graphical User Interface
HDL	Hardware Description Language
I2C	Inter-Integrated Circuit
I2S	Inter-IC Sound
IMU	Inertial Measurement Unit
IP	Intellectual Property
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
ISS	Instruction Set Simulator

IoT	Internet of Things
L1	Level 1
L2	Level 2
LDO	Low DropOut
LED	Light Emitting Diode
LiDAR	Laser Imaging Detection and Ranging
LiPo	Lithium Polymer
LQR	Linear Quadratic Regulator
LSF	Linear Signal Flow
MAC	Multiply-ACcumulate
MCU	Micro-Controller Unit
ML	Machine Learning
MPU	Memory Protection Unit
NN	Neural Network
OS	Operating System
PID	Proportional-Integral-Derivative
PULP	Parallel Ultra Low Power
PWM	Pulse Width Modulation
RAM	Random Access Memory

RISC	Reduced Instruction Set Computer
ROS	Robot Operating System
RPC	Remote Procedure Call
RPM	Rotations Per Minute
RTL	Register Transfer Level
SLAM	Simultaneous Localization and Mapping
SOC	State Of Charge
SoC	System on Chip
SP	Set Point
SPI	Serial Peripheral Interface
SW	Software
TDF	Timed Data Flow
TLM	Transaction-Level Modeling
ToF	Time of Flight
UAV	Unmanned Aerial Vehicle
UART	Universal Asynchronous Receiver-Transmitter

Chapter 1

Introduction

In the latest years, the interest towards unmanned air vehicles (UAVs) has taken off quite significantly. The appeal of drones is in fact both industrial and academic, due to their great versatility and cheap cost. They are able to provide significant advantages over any other aerial vehicle in several fields, ranging from agriculture analytics, package delivery and military defense, to rescue operations, cinema, and weather forecasting[1].

In essence, drones are a category of aircrafts that do not carry a human operator, and make use of the airfoil principle to provide vehicle lift. They are either autonomous or remotely controlled, making them also very appealing to research and other specialised purposes. The great adoption of such technology has led to different kinds of drones, developed to suit specific size and speed needs. Among those, one of the latest and most interesting classes of vehicles is the one of nano drones.

Also known as miniature or small unmanned aerial vehicles, nano drones are, to put it simply, a downscaled version of standard drones, offering as a result a unique combination of size, agility, and affordability. These tiny, lightweight drones are designed for specific tasks (such as reconnaissance, surveillance, and communication relay[2]), and their desirability has recently increased even further thanks to the spread of on-edge Artificial Intelligence (AI) technology [1], allowing for example the autonomous exploration of complex and dynamic spaces, or the use of several of those in drone swarms and other recreational flying applications in general.

Being particularly small, however, nano drones are able to support only small battery packages, which results in having at their disposal a very limited flight time.

---

[1]https://www.cbinsights.com/research/drone-impact-society-uav/

[2]https://www.stratechos.com/techtonics/nano-drones%3A-small-size%2C-big-impact-in-modern-industries

Nevertheless, this drawback can be easily studied prior to the drone production thanks to the availability of simulation softwares, which allow to emulate the impact on flight time of several batteries, different scenarios and other drone characteristics.

More broadly speaking, the ability of using simulations to investigate possible flaws and enhancements during the design phase of a product is a highly valuable strategy for companies in general: it allows in fact to explore the design space without the need of developing physical prototypes, resulting in huge time and cost savings. This is especially true for hardware companies, where rigorous logic verifications have to be carried out before moving into the production phase of processors and other high-end electronic components. To accommodate this need for hardware simulation, several softwares have emerged in the past years. As better detailed later in the document, these include programs such as Simulink [2], QEMU [3] and RENODE [4], which share the ability of enabling detailed functional simulations per different architectures (each focusing for slightly different use cases).

In this context, the rise of the RISC-V architecture [5] opened a new set of advantages for functional simulation: its open-source nature as well as its extensible and customizable instruction set makes it particularly suited for simulation environments, allowing for rapid prototyping and testing of hardware designs without the constraints imposed by proprietary architectures. This customizability allows developers and researchers to ease and accelerate the exploration process[3], giving even more importance to the simulation stage. Hardware emulation is also much more accurate than others, as it allows for a very precise modelling of internal delays and other timing related characteristics.

But the concept of functional simulation is widely spread at higher levels as well. In the field of robotics, for example, Webots [6] stands out as a very powerful piece of software providing a convenient way to create testing environments for a vast quantity of robotic systems. In this case, the simulation involves the model of one (or more) robots through their sensors (such as cameras, LiDARs, GPS, microphones, etc.) and actuators (like linear and rotational motors, propellers, brakes, etc.). These are then placed inside a realistic 3D environment, which can be customised with models of both static and dynamic objects to replicate with great accuracy specific industrial or everyday scenarios. The robot(s) can then be programmed to interact with the surroundings, enabling developers to gather relevant information about its behavior and to collect the data recorded by the device sensors.

Hardware and functional simulation are however just one of the two most important aspects when dealing with electronic devices, especially when considering

[3]https://www.ust.com/en/insights/risc-v-the-underdog-chip-poised-to-reshape-the-tech-landscape

battery powered ones such as nano drones. In fact, as said, being able to carry only a limited battery capacity is one of the main drawbacks of these vehicles: hence, it is evident how the simulation of a robot may profit from the inclusion of extra-functional aspects such as power consumption. The need to model energy intake may also be extremely useful in other scenarios too, such as when dealing with several robots attached to the power line in an industrial plant.

Among the several solutions that condense both the functional and the extra-functional aspects inside a unique platform, SystemC [7] and its expansion SystemC-AMS [8] are particularly noteworthy. They provide a comprehensive environment that allows to model and simulate both digital and analog/mixed-signal systems. By enabling the co-simulation of functional behaviors (such as logic operations and control algorithms) along with extra-functional aspects like power consumption, timing, and signal integrity, they represent very ideal tools to model complex systems like nano drones and robots in general.

To ease the development and use of such SystemC-based simulations, the MESSY framework [9] was recently developed by Politecnico di Torino. This tool exploits the expressive power of SystemC for functional and extra-functional aspects and provides the user with a customizable interface to automatically generate ready-to-use codes for sensors, actuators, batteries and more. Moreover, it enhances the simulation capabilities of robotic systems by integrating GVSoC [10], a powerful and almost cycle-accurate functional hardware simulator for GAP8, which is, in turn, a versatile RISC-V processor tailored for Internet of Things (IoT) and embedded applications. By combining the two, MESSY is able to provide a comprehensive simulation framework for a wide range of RISC-V based devices, allowing for precise system-level simulations.

However, being MESSY a recent development, no attempt has ever been carried out to test the actual capabilities of the software, especially when considering the simulation of real, complex products. In parallel with other projects, the following thesis will thus address the challenge of using MESSY to build a simulation environment to model as accurately as possible the functional and extra-functional characteristics of a RISC-V based nano drone: the Crazyflie 2.1 [11] by BitCraze. In order to make the simulation even more realistic, MESSY will be integrated with Webots: making use of its robust physical engine, the resulting system will enable the testing of control algorithms and potential neural networks running on the drone onboard processor while estimating in real time the battery performance and the underlying hardware dynamics.

To describe the development process of such a system, the following chapters are structured as follows. Chapter 2 delves deeper into the fundamental topics the whole thesis is based on: starting from a more in-depth introduction on the RISC-V architecture, the interest shifts towards sockets (the inter-process communication mechanism to establish the communication between MESSY and Webots), system

simulations and the nano drone structure, control and hardware architecture. Chapter 3 provides an overview at the current state-of-art of Instruction Set Simulators (ISSs), functional and robotics simulations, and later moves onto the description of the drone power models available in literature. Moving to the core part of the thesis, Chapter 4 illustrates how the connection of MESSY and Webots can be achieved. The discussion then documents the modelling of the Crazyflie nano drone in the two softwares, starting from the control logic and its sensors up to the energy consumption evaluation models. In Chapter 5, several tests illustrate the low overhead of the system with respect to purely functional simulations and better highlight its potential for Design Space Exploration (DSE). Finally, Chapter 6 recaps the major benefits and limits of the developed environment, focusing on the several possible improvements that could be considered to further enhance the simulation on both the functional and power perspectives.

Chapter 2

Background

The following chapter will introduce all the background knowledge necessary to fully comprehend the thesis topics. In particular, great attention will be posed on the RISC-V [5] architecture in Section 2.1, in order to grasp the necessary details about this quickly growing instruction set and better identify the potentials and limits of its adoption. Secondly, a general overview of simulations will be proposed in Section 2.2 to make the reader aware of their several benefits and their importance in this context and in other fields as well. In Section 2.3, the interest will shift towards sockets, and especially UNIX sockets, as those are the underlying mechanism through which the communication between MESSY [9] and Webots [6] will be established. Some general information about drones will be given in Section 2.4 and, lastly, the spotlight will be pointed towards the Crazyflie 2.1 [11] architecture.

2.1 RISC-V

Being one of the most promising projects in the field of computer architecture, RISC-V [5] is an open standard instruction set architecture (ISA) born in 2010 at the University of California, Berkeley. Differently to most of its competitors, this design is provided under royalty-free open-source licenses, which make it very appealing for both research and hardware production purposes.

As also stated in its name, RISC-V belongs to the family of Reduced Instruction Set Computer (RISC) architectures, whose focus is providing an instruction set with fewer and faster (ideally, lasting one clock cycle only) operations. Rather than CISC (Complex Instruction Set Computer), which focuses on completing a task with the fewest lines possible, the approach for RISC-V is to divide complex tasks in simpler flows of instructions by exploiting the so called "load-store"

architecture: operations belonging to the ALU (Arithmetic and Logic Unit, the section of the processor devoted to arithmetical and logical operations) support only register-register operands, meaning there is a complete separation between ALU and memory operations. As a consequence, stored operands should first be loaded from memory to register, and vice versa to save the result. While CISC dates back to 1970s and has dominated the market for both consumer and server computing due to its small code size and lower reliance on memory and on software, the RISC approach is recently gaining more and more attention[1] due to its simplicity, scalability and general efficiency. As an example, this is the approach towards which the personal computer market is moving to[2], on both MacOS (with the M series chips) and Windows (with the very recent Snapdragon X Elite series) devices. RISC is also the basis of ARM (formerly an acronym for Advanced RISC Machines, now part of the British semiconductor company Arm Ltd.) chips, which have vastly dominated[3] both embedded and mobile (smartphones, tablets, etc.) markets in the last decade, due again to their power efficiency and cost advantages.

The problem, however, is the fact that all of these solutions are either closed source or protected by license: this is the case for the ARM designs, which are sold as IPs (Intellectual Properties) to third parties through licensing agreements. The reader should hence be aware of the several benefits of developing an open RISC hardware: the open-source nature of RISC-V introduces an unmatched flexibility and ease of customizability, allowing developers to optimize the architecture to suit a specific application. The modular design of RISC-V facilitates its implementation also on FPGAs (Field Programmable Gate Arrays), enabling the possibility of prototyping even entire processors without the need of physical models. Its adaptability makes RISC-V particularly well-suited for embedded systems as well, where power efficiency and tailored performance are the main concerns. For more complex designs, several RISC-V cores can be integrated on the same chips and interfaced with custom hardware (AI accelerators, I/O controllers, etc.) , expanding its possibilities even further.

2.1.1 PULP

It should be underlined, however, that RISC-V ISA is defined in a way which is deliberately not focused on hardware implementation, but rather seeks to provide an efficient and flexible approach that can be adapted across a wide range of

[1] https://riscv.org/blog/2024/04/risc-v-impact-on-technology-and-innovation/

[2] https://www.nextmsc.com/report/arm-based-pc-processors-market

[3] https://www.statista.com/statistics/1132112/arm-market-share-targets/

applications. One noteworthy example of how this flexibility is leveraged in practice is the PULP (Parallel Ultra-Low Power) [12, 13] platform, which is a project that implements the RISC-V architecture in a way that is optimized for energy efficiency and performance for low-power embedded systems and IoT (Internet of Things) applications. PULP is a joint project between the Integrated Systems Laboratory (IIS) of ETH Zurich and the Energy-efficient Embedded Systems (EEES) group of University of Bologna that was born to develop an open, scalable hardware and software research platform with the goal to break the pJ/op barrier. The project consists of a set of IPs described in the SystemVerilog language, the related simulation and synthesis scripts, as well as the runtime software written in C and RISC-V assembly. Several are the platforms that have already been developed to suite different application scenarios. The most notable ones are: PULP (multi-core, organized in clusters of RISC-V cores which share together a tightly-coupled data memory), PULPino [14] (single-core, similar to a microcontroller without caches nor DMA), PULPissimo [15] (single-core, which is a "boosted" version of PULPino, including additional peripherals such as micro-DMA) and RI5CY [16] (32-bit, in-order RISC-V core with a 4-stage pipeline that implements the RV32I ISA and some PULP custom extensions). The latter is of particular interest in the context of this document, as it provides the basis to the GAP8 [17] processor, the multi-core RISC-V chip that can be installed on the Crazyflie 2.1 drone.

2.1.2 GAP8

GAP8 [17, 18] is a fully programmable RISC-V based IoT-edge computing engine developed by GreenWave Technologies. The PULP platform is the core basis of the processing power of the chip: the design features in fact 8 RI5CY cores, plus one additional high performance RISC-V based core. The two have different purposes: the 8-cores cluster can execute in parallel, and provide high performance calculation for image processing, audio processing, signal modulation, etc., while the single core, referred to as the “Fabric Controller” or simply FC, is used as micro-controller, meaning it is in charge of controlling all the operations of GAP8 like the micro-DMA. Completing the architecture, GAP8 also features: 64 KB of level 1 (L1) memory shared by all the cores in the cluster, 8 KB of L1 memory owned by the FC, 512 KB of L2 cache for all cores, a lightweight and completely autonomous micro-DMA, a multi-channel cluster-DMA that controls the transactions between the L2 and L1 memory, 2 programmable clocks and a Memory Protection Unit (MPU). In addition to all of this, this SoC (System-on-a-Chip) features a custom CNN (Convolutional Neural Network) accelerator unit named HWCE, which is suitable for accelerating convolution operations. As a result, GAP8 can deliver up to 10 GMAC/s for CNN inference (90 MHz, 1.0V) at the

energy efficiency of 600 GMAC/s/W within a worst-case power envelope of 75 mW[4].

Lastly, it is important to notice the support of communication protocols such as I2C (Inter-Integrated Circuit), I2S (Inter-IC Sound), CPI (Camera Parallel Interface), SPI (Serial Peripheral Interface), and UART (Universal Asynchronous Receiver-Transmitter), which are enabled by several on-board interfaces. The presence of these, along with the micro-DMA and the CNN accelerator, make this chip very suitable for nano drones, as they allow for a energy-aware hardware support for the image processing and neural network inference tasks that have to be performed in general AI-related applications. This is even enhanced by the fact that all the cores and peripherals are power switchable and their voltages and frequencies are adjustable on demand, allowing GAP8 to adapt extremely quickly to the processing/energy requirements of a running application. The biggest limit of the chip is arguably the lack of floating point (FP) operations support, which would however impose a significantly higher power consumption. Moreover, CNNs can actually be run in fixed point without significant loss of precision, limiting the usefulness of a dedicated FP unit.

2.2 System simulation for virtual prototyping

As stated earlier in Section 1, simulation is a fundamental concept and a very important step in the design and development of complex systems. In the field of engineering, simulation can be thought of as the process of creating a virtual model of a real entity inside a software with the scope of testing it against some conditions and verifying some of its characteristics. The reader should be aware of the fact that using simulations helps in multiple ways: it does not limit itself to help identifying potential flaws, but it also allows to explore various design alternatives and make informed decisions, all while reducing time-to-market and development costs. Of course, these benefits led to a wide spread of simulations in several other fields, such as aerodynamics [19], economics [20], architecture [21], medicine[5] and biology [22]. The thesis however will deal with Instruction Set Simulators (or ISSs).

As suggested by its name, ISS is a general term for a software which mimics the behavior of a processor at the instruction set level. It can be used to develop and debug code without the need of having access to the hardware, possibly offering speedups and cost savings. This is even more true considering the fact that such simulation can be performed at lower levels of accuracy, or may be used to model

[4]http://asic.ethz.ch/2017/GAP8.html

[5]https://cambridgemedicine.org/doi/cmj.2021.08.001

slower embedded processors on much powerful machines. They are available for a wide range of platforms, from mainstream to specific ones[6].

2.2.1 Functional simulation

In a lot of cases, simulators are used for functional purposes only. A simulation of a design is considered functional when the focus is purely oriented towards logic rather than timing or other parameters. In a functional simulation, delays and latencies may not be accurately reflected in results, as the interest is to check only the "behavior" of the object being tested. An excellent example of functional only simulation is the one that is obtained when using an Hardware Description Language (HDL) to create a Register Transfer Level (RTL) model of a circuit design. In such a case, the simulation of a component can be carried out by means of a testbench and finally loaded onto some simulator softwares to check the intended behavior of the device. Later in the development, such circuit undergoes the synthesis stage, after which it is possible to obtain also useful information about the circuit timings. For the scope of the following, it should be highlighted that also Webots provides what can be considered a functional simulation environment, despite the fact that robots are the actual entities whose behavior is going to be verified rather than circuits.

As a general idea, three are the main approaches through which the simulation of an instruction set can be achieved. Those are, namely, the following:

- **interpretation**: each of the program lines is decoded and executed once at a time, resulting in an easier overall implementation which however can be relatively slow (due to the time required to process instructions individually);
- **virtualization**: a simulated environment is created in which code can be executed as close as possible to the real hardware, due to the implementation of the ISS at the OS (Operating System) level. In this case, the code (which can also be precompiled) can execute much faster, at the expense of the complexity of virtual environment resource requirements;
- **just-in-time**: a middle way approach that dynamically translates the program code into the host (i.e, the machine "hosting" the simulation) native language, which balances the advantages and trade-offs of the former.

On a last note, it should be evidenced that these programs may help emulate the behavior of various system components too such as memory, bus, I/O devices, user input and so on, further enhancing the simulation capabilities.

[6]Some examples for RISC, ARM and MIPS can be found at http://www.fast-iss.org/

2.2.2 Extra-functional simulation

As underlined earlier in the document, the pure behavior of a device is certainly not the only characteristic that may be of interest to simulate. Speaking of circuits, it was already mentioned the fact that timing simulations may be of great interest, for example when checking the occurrence of problems in a circuit at different clock frequencies [23]. This concept however can be expanded for other properties too, with the main ones being signal integrity, thermal efficiency, electromagnetic interference (EMI) and, most importantly, power consumption. Considering again the circuit design flow example, this set of properties can be tested after the place and route phase: at this stage, knowing the physical placement of cells on the die, it is possible to use software to evaluate these parameters with close-to-real-hardware simulations, to identify for example signal distortions or possible inefficiencies in the thermal distribution. Some power analysis can be performed too, but with the catch that, along with the static/leakage power draw, the dynamic consumption can only be estimated on the basis of some general indications about the switching activities of the inputs.

In general, those properties are not included inside ISS simulations, with the exception of timing: the ultimate goal of an ISS would be in fact to simulate a micro architecture in a cycle-accurate way, mimicking the exact timing and behavior of the original hardware.

Along with timing, one the most crucial factors to be included in hardware simulators is power consumption. Being able to dynamically evaluate the consumption of a chip has a great impact on the accuracy in estimating either battery life or the running cost of a device, especially in the embedded environment [24]. Of course, this would imply an additional overhead in the simulation, introducing the need of finding a good trade off between simulation time and accuracy.

2.3 Inter-Process Communication through network sockets

Any computer program can be run on a machine by means of a process, which is basically the instance of such program being executed by one or many threads. A process can be either independent or co-operating with other processes, meaning in the latter case that its execution is affected by other running processes, which can be done, as an example, for data exchange or synchronization scopes. Inter-process communication (IPC) is the mechanism that allows processes to communicate with each other and synchronize their actions. The communication takes place using one

of two general approaches: either the two share a common reference to a memory area or a connection channel is first established and messages are then exchanged on such medium. Nonetheless, it would be wrong to categorise the IPC techniques in these two sets only, as their boundary line is not always sufficiently clear. Hence, follows a list of the main IPC mechanisms, outlining on a case by case basis the working principle through which the communication is achieved:

- **Pipes**: one-way communication mode that exploits system calls provided by an OS. Simpler, but slower and uni-directional;
- **FIFOs**: half-duplex version of pipes (also known as "named pipes"), in which communication happens through a specific file handled by the OS;
- **Signals**: mechanism used to send asynchronous notifications (signals) across processes. Generally used to handle events rather than transferring data;
- **Message Queues**: it allows messages to be passed using either a single or several message queues. It is managed by the system kernel, and the messages are coordinated using an API (Application Programming Interface);
- **Shared memory**: it consists of using a shared RAM memory region, where multiple processes can access the same data and perform read/write operations. This is a very fast approach, but requires a synchronization mechanism between the processes (such as semaphores or mutexes);
- **Memory-Mapped Files**: mapping between a file and part of the OS address space, which enables multiple processes to modify the file by reading and writing directly to the memory, improving read and write speeds especially for larger files. They can also be persistent, meaning they are associated with a source file on a disk;
- **Network domain sockets**: data is sent over a network interface, offering a connection which is computer and OS independent. They are mostly used to communicate over a network, but can also be used for processes on the same machine;
- **UNIX domain socket**: very similar to sockets, but in this case all communications occurs within the kernel of a machine. Domain sockets use the file system as their address space, so they can avoid some checks and operations (like routing), which makes them faster and lighter;
- **Remote Procedure Calls (RPCs)**: in a distributed network, RPCs happen when a computer program causes a procedure (subroutine) to execute in a different address space (either on the same or on a remote computer) as if they were local.

Further distinctions can be made between blocking and non-blocking mechanisms. Blocking mechanisms, such as some implementations of pipes or message queues, cause the process to wait until the operation is complete, ensuring synchronization but potentially leading to inefficiencies. Non-blocking mechanisms, instead, allow the process to continue executing other tasks while waiting for the communication to complete, providing better performance but requiring more complex error handling and synchronization strategies.

Follows now a deeper focus on the working principle of UNIX sockets (the IPC technique this thesis will mainly deal with), explaining to the reader how the connection is established and managed from a higher level perspective.

2.3.1 UNIX sockets

To introduce the working details of UNIX sockets [25], it must be first specified that the connection is composed of a client and server. In practice, both of them create a socket and the client then *connects* to the one of the server, which in turn *accepts* the request.

The entire list of operations that is done by the **server** in the lifetime of a socket is represented in the list below. Be aware that some coding-like names for the system calls will be used, to make the reader familiarise with some of the terms which will be used later in the discussion of the C++ code that controls both MESSY and Webots.

- `socket()`: the first command to be issued is the `socket()` system call, which returns a file descriptor that can be used for future references to the socket;
- `bind()`: follows the binding of the socket to a known address, so that the client can later connect to it;
- `listen()`: with this command, the socket is marked as "passive", i.e., it is listening for incoming connection requests;
- `accept()`: a blocking step that halts the code execution until a request has arrived. When so, it accepts such request and returns a new file descriptor, different from the one obtained with the `socket()` call, that shall be used for any communication with the client;
- `read()` and `write()`: once the connection is established, these two system calls can be used to exchange data with the client. Note that the `read()` operation can be either blocking or non-blocking;
- `close()`: at the end of the program, it is good practice to use this command to release the file descriptors and make available again the used resources.

On the **client** side, instead, the standard actions to be performed are the following:

- `socket()`: as per the prior case, a socket is first created on the client side too;
- `connect()`: it is used to connect to the server passive socket, which is reachable thanks to its well-known address;
- `read()` and `write()`: once again, read and write operations can now be performed on the socket;
- `close()`: finally, the connection is closed in order to de-allocate the used resources.

To ease the understanding, Figure 2.1 recaps the general flow of execution on both the client and the server sides. As a last addition, the reader should be informed that the server socket can accept many client connections. This is the reason why the server side uses the file descriptor returned by `accept()` rather than the one obtained after the `socket()` system call.

2.4 Unmanned aerial vehicles

2.4.1 Introduction to drones and nanodrones

As stated earlier in Chapter 1, drones are a class of aerial vehicles that can fly either autonomously or remotely controlled. The absence of a human pilot on board is the reason why this class of aircrafts is also known as unmanned (meaning, without a human pilot) aerial vehicles.

Drones are suitable for a wide spectrum of tasks, but, in essence, two are the main contexts the drones are utilised: transport and camera/sensors monitoring. About the former, the convenience of drones is quite evident: they provide a cost-effective way to reach even remote or narrow spots, making it ideal for example in rescue missions and package delivery in bigger cities. This has led to the creation of largely sized drones, some weighing even more than 150 kg. About the other applications, such as defense or infrastructure inspection, drones provide an unmatched combination of efficiency and agility, making it easy to both monitor small and wider spaces with cameras (e.g., for agriculture and military purposes) and gathering data from LiDAR (Laser Imaging Detection and Ranging), radars,

[7] https://medium.com/swlh/getting-started-with-unix-domain-sockets-4472c0db4eb1

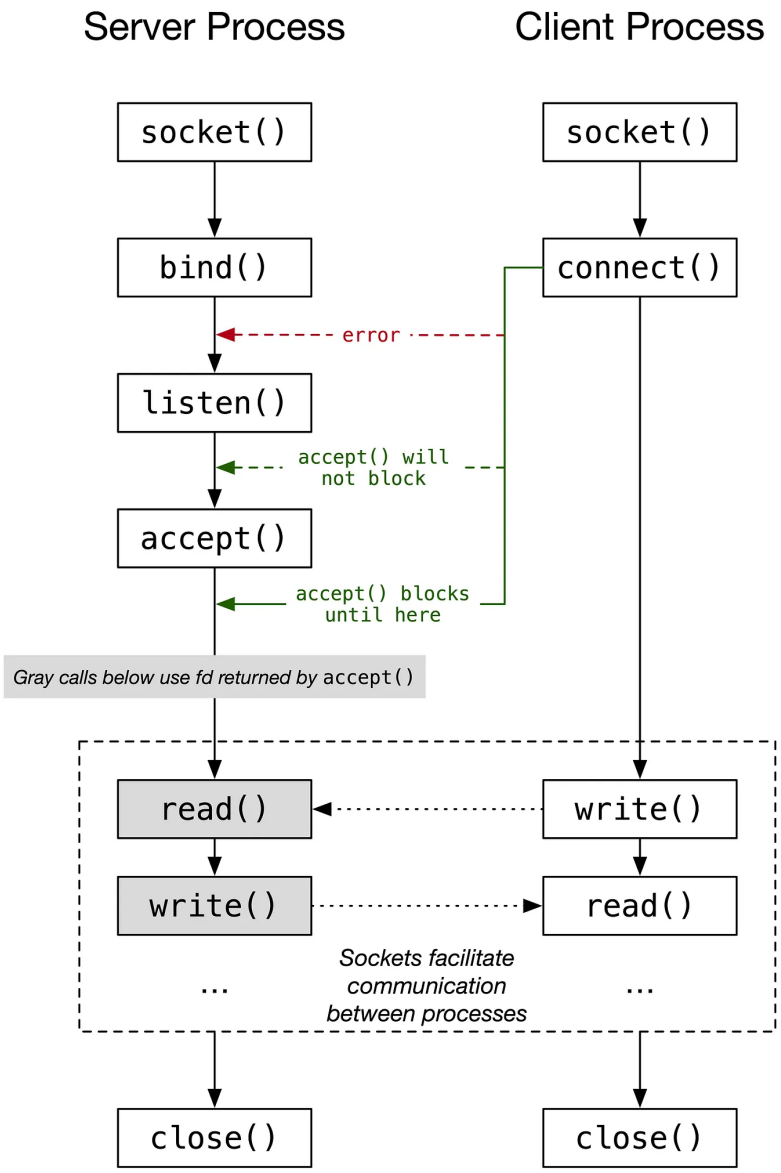
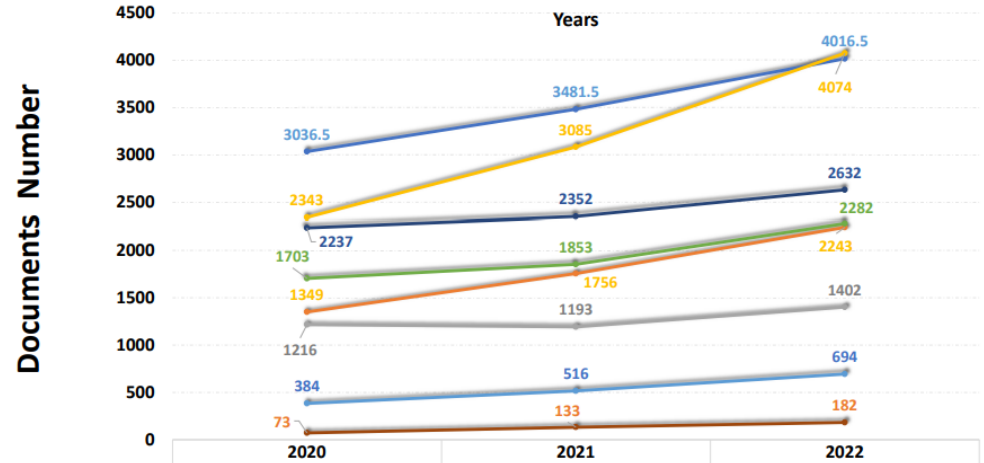


Figure 2.1: Overview of client and server operations in typical UNIX domain socket connection[7].

pressure sensors, etc. for IoT, weather forecasting, and so on[8].

[8]https://www.cbinsights.com/research/drone-impact-society-uav/

The interest in UAVs has surely grown even more with the spread of AI technologies. Several peer review studies were conducted in the past years to analyze the state-of-art of research in the field of drones [26, 27], with [28] being one of the most noteworthy. One of the analyses proposed by the paper goes over the number of drone-related publications in each research direction. As reported by the results, depicted in Figure 2.2, the growth of AI is the most significant, surpassing even the one of antennas (which has been the most attractive as the transmission and reception of signals are essential for the teleoperation of UAVs).



	2020	2021	2022
Antenna x 0.5	3036.5	3481.5	4016.5
Aircraft Detection	1349	1756	2243
Remote Sensing	1216	1193	1402
AI	2343	3085	4074
IoT	384	516	694
Energy-Efficient	1703	1853	2282
Control	2237	2352	2632
Swarm Behavior	73	133	182

Figure 2.2: Growth trajectories in UAV research directions from January 2020 to December 2022 [28]

It is in fact true that AI algorithms are revolutionising the way in which UAVs operate. By using these, UAVs can achieve improved autonomous flight and decision-making capabilities, especially when combining together data from cameras, LiDARs and other sensors. As an example, computer vision can enable the UAV to recognize objects and people in its environment, which can be used to improve safety and prevent collisions, meanwhile, LiDARs can provide detailed information about the UAV surroundings, including the distance, size, and speed of the objects, which can be used to more effectively navigate and model complex

environments such as for SLAM (Simultaneous Localization And Mapping) algorithms. Another way AI can be used in UAVs is through the optimization of flight paths: by using Neural Networks (NNs) , UAVs can learn from past flights and optimize their routes to reduce energy consumption and increase efficiency [29, 30]. This can be achieved by analyzing data such as wind speed, temperature, and other environmental factors.

As said, the many possibilities opened by drones and the integration of AI would come useful in several scenarios. Being potentially very different from each other, several models of drones have been proposed, with changes in speed, maximum payload, range, and, most importantly, weight and size. Drones can be in fact categorised as follows:

- **Nano** if the weight is lower than 250 g;
- **Micro** if the weight is between 250 g and 2 kg;
- **Small** if the weight is between 2 kg and 25 kg;
- **Medium** if the weight is between 25 kg and 150 kg;
- **Large** if the weight is higher than 150 kg;

Once more, it is important to stress that the gross differences in size will reflect the very different capabilities, consumption and applications of each of these classes.

As stated earlier, the following document will deal with a nano drone, a device belonging to the most lightweight class of drones. These devices share some common characteristics: they are all battery powered (while other drones may use fuel or solar panels), all have more than one rotational motor, they are usually not build to transport any additional payloads (other than the drone weight itself), and they have low range and low altitude capabilities. This category of devices is very suitable for some specific tasks, mainly the ones that require a great agility and maneuverability, especially in restricted spaces. Among the most important, nano drones are used for surveillance purposes[9], inspections of machines and pipes[10], drone racing and even very specific ones like artificial pollination [31]. Needless to say, combining drones with AI will surely benefit any of those, allowing to perform the tasks in a completely automated manner. However, it is important to

[9] https://www.stratechos.com/techtonics/nano-drones%3A-small-size%2C-big-impact-in-modern-industries

[10] https://candrone.com/blogs/news/revolutionizing-pipeline-inspections\-the-role-of-drone-technology

consider that, being nano drones significantly smaller than the others, less power will be consumed by the motors (as they need to produce less lift) and the energy contribution of a neural network will be significantly more relevant.

2.4.2 Drone controllers

Because of their small size and weight, the task of controlling a UAV presents some great challenges: the dynamics of drones have to face problems as underactuation (robots with fewer actuators than degrees of freedom), non-linearity, static instability, and strong coupling between the dynamic states. The discussion slightly changes when limiting it to devices operating with rotatory motors: this category of vehicles, which share two or multiple pairs of identical fixed pitch propellers, allow a much higher maneuverability, leading to an overall greater range of movement and ease of programming.

These aircrafts may be controlled by means of either linear or nonlinear controllers. About the former, this set of algorithms assume that the drone can be accurately approximated by means of linear equations. Two are the most worth mentioning techniques: PID (Partial Integral Derivative) and LQR (Linear Quadratic Regulator) controllers [32].

As suggested by its name, a PID control function works on the basis of three main components: a partial (P), an integral (I) and a derivative (D) one. The function exploits a closed-loop feedback system that continuously calculates an error value $e(t)$ as the difference between a desired setpoint (SP, $r(t)$) and a measured process variable (PV, $y(t)$), which gives the information about the current status of the system. Once $e(t) = r(t) - y(t)$ has been evaluated, it applies a correction to the system based on P, I, D terms. As a general idea, the P term is proportional to the current value of the error signal, the I term accounts for past values of the error and integrates them over time, while the D coefficient (also known as "anticipatory control") provides a mechanism to estimate the future trend of the error based on its current rate. These three contributions are taken into account by multiplying the error, its integral in time and its derivative with respect to time with the K_p, K_i and K_d coefficients, respectively. The system, whose graphical representation is reported in Figure 2.3, will thus output the following $u(t)$ signal:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(t)dt + K_d \cdot \frac{de(t)}{dt} \tag{2.1}$$

Of course, some test shall first be carried out before implementing a PID controller,

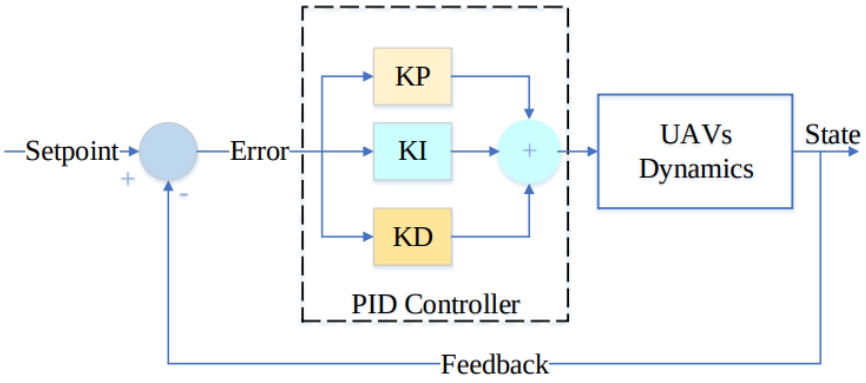


Figure 2.3: PID controller scheme [32].

in order to tune the three coefficients with optimum values for the desired control response. Being so adjustable, PIDs are a very general method that finds application in several other fields as well, such as in speed control in electric vehicles [33] and temperature management in electric furnaces [34].

On the other hand, LQR controllers represent another valid type of functions that can be used to command multirotors [35]. They describe a system with a set of linear differential equations plus a quadratic cost function. The core idea is to minimise this cost function, which typically consists of parameters such as the distance from the desired state and the energy consumed to perform corrective actions. As per the PID, this kind of controller works continuously by sensing the current conditions of the drone (again, forming a feedback loop) and producing at each step a new set of control inputs for the drone. One key difference, however, is that LQR controllers focus on finding the optimal strategy to minimise the cost function, rather than simply reacting to errors.

An LQR cost function generally takes a form like the following:

$$J = \int_0^{inf} (u^T(t) \cdot R \cdot u(t) + x^T(t) \cdot Q \cdot x(t))dt \tag{2.2}$$

where J is the (quadratic) cost function, u is the vector of the control input, x is the state vector, Q is a weight matrix that penalizes the state deviation from the desired one, and R is another matrix that penalized the control effort. The solution of such equation can then be found by evaluating $u(t)$ on the basis of some other known equations (such as the Continuous-time version of the Algebraic Riccati Equation, CARE). This method proves to be both robust and flexible, with the

downside of requiring a very difficult tuning.

About the nonlinear models instead, these controlling algorithms are derived from the original dynamic model of the UAVs, which leads to the overall best performance. Among these, feedback linearization, backstepping, sliding mode control, adaptive control, and model predictive control are the ones that have lately received the most attention [32]. However, as this kind of controllers results in more complex models, a more in-depth discussion will be omitted for the sake of brevity.

For the scope of this thesis, a PID controller will be considered throughout all the tests. Despite not being the most optimized or adaptive approach, PIDs have proven to behave sufficiently well in tasks such as stabilization and hovering motion, showing good robustness and performance while providing an extreme ease of implementation.

2.4.3 The Crazyflie 2.1 nanodrone

As previously anticipated, the case-study for this work will be the Crazyflie 2.1, shown in Figure 2.4. This nanodrone (which is going to be referred to as "Crazyflie" for convenience) stands out due to its modular and open source architecture, which allows for high flexibility and adaptability in various applications, especially in research. This robot was developed by Bitcraze, a company which has its roots precisely in the development and manufacturing of the Crazyflie series and their expansion kits, which were created with the goal of developing an "electronic board that flies", and make it available as an open source development platform.

This drone is a *quadrotor* (alternatively called *quadrucopter*), meaning it possesses four rotors (i.e., rotatory motors) with the following configuration: looking from the top of the drone, two rotors on a diagonal spin clockwise (CW) and the other two counterclockwise (CCW). This is done because if they all shared the same direction the drone body would receive a reaction torque from the motion of the propellers in the opposite direction. A scheme of the rotation of the four propellers is reported in Figure 2.5.

Since it belongs to the nanodrones category, it is no surprise that the drone structure is considerably small indeed: looking from the top, the distance between

[11] https://www.bitcraze.io/products/old-products/crazyflie-2-1/

[12] https://www.bitcraze.io/documentation/tutorials/getting-started-with-crazyflie-2-x/

Figure 2.4: The Crazyflie 2.1 drone by Bitcraze[11].

the rotors is less than 10 cm (92 mm per each pair of adjacent motors), with a total height of just 29 mm. As a result, the drone weight is extremely contained too, settling at just 27 grams. The robot has sufficient power to lift with an additional payload, which however must not exceed 15 g.

At its core, the drone architecture is based on the cooperation of two different low-power MCUs (MicroController Units):

- **STM32F405**[13]: a chip from ST Microelectronics featuring a single core Cortex-M4, running at 168 MHz, with 192 kB SRAM, 1 MB flash and 15 communication interfaces including USART and SPI. This chip provides a good compromise between performance and power consumption, since, due to the 90 nm manufacturing and the dynamic power scaling, can reach a consumption as low as 238 µA/MHz. It is used to run the main firmware, which is in charge of several tasks: motor and flight control, sensor reading,

[13] `https://www.st.com/en/microcontrollers-microprocessors/stm32f405-415.html`

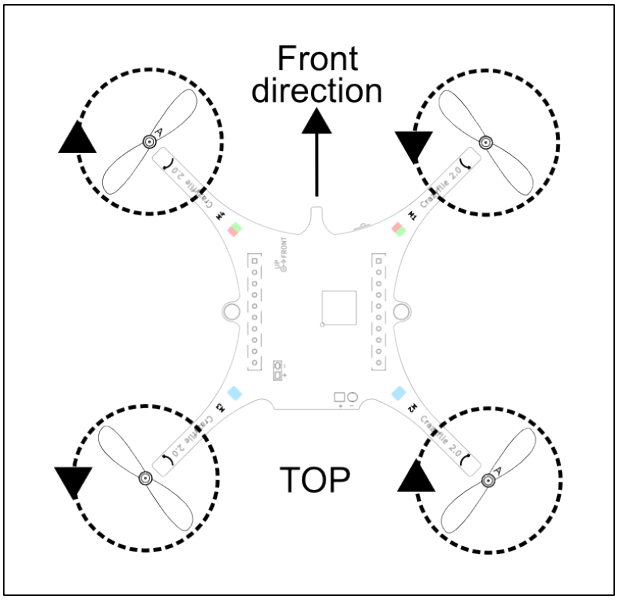


Figure 2.5: Propellers direction of motion[12].

telemetry and battery tracking. In addition, this is also the target chip for any user application;

- **nRF51822**[14]: a radio and power management MCU. It is a single core, ultra-low power SoC that uses a Cortex-M0 running at a maximum of 32 MHz with 16 kB SRAM and 128 kB flash. One of the most noteworthy characteristics of this chip is the support for Bluetooth®Low Energy (BLE) and other 2.4 GHz protocol stacks. In the context of the Crazyflie, this chip is in charge of handling the ON/OFF logic, enabling power to the rest of the system (STM32, sensors, etc.), manage battery charging, radio, BLE communication and detect the installed expansion boards.

About the latters, one of the most remarkable features of the Crazyflie drone is its support for the so-called *expansion decks* or *expansion boards*. These decks are modular add-ons that can be attached to the drone to extend its capabilities. The barebone drone in fact only has one IMU (Inertial Measurement Unit) sensor, which consists of a 3 axis accelerometer/gyroscope (BMI088[15]), and a high precision pressure sensor (BMP388[16]). With the help of such expansions, the drone can be

[14] `https://www.nordicsemi.com/Products/nRF51822`

[15] `https://www.bosch-sensortec.com/products/motion-sensors/imus/bmi088/`

[16] `https://www.bosch-sensortec.com/products/environmental-sensors/pressure-sensors/bmp388/`

equipped with several other components to enhance its features. Some expansion deck examples are the Flow Deck, which adds a ToF (Time of Flight) sensor (which measures the distance to the ground with high precision) and an optical flow sensor (which measures movements in relation to the ground); the Multiranger Deck, which adds five ToF sensors (to measure distance up to 4 meters within a few millimeters in five directions) and the Z-Ranger Deck, which adds a laser sensor to measure distance from ground. Other decks provide additional LEDs, buzzers and better positioning systems, which collectively increase the range of possibilities of the drone and allow its customization on the basis of the specific application needs.

For the scope of this thesis, only one deck will be considered: the AI deck. This expansion kit integrates a RISC-V processor and a camera in a 4.4 g package, allowing the Crazyflie to run AI algorithms directly on-board and enabling advanced functionalities such as computer vision and object detection. The processor embedded in the kit is the GAP8 processor, described earlier in section 2.1.2. As just said, this SoC is paired with a camera, which in this case is an HM01BO[17] sensor from HIMAX. This ultra-low-power CMOS sensor is capable of capturing 320x320 (pixels) images at a maximum frame rate of 51 FPS (Frames Per Second), all within a package of 5 mm by 5 mm. Moreover, the kit also features a ESP32 MCU (the NINA-W102) which enables the WiFi connectivity. Together, this combination of chips extends the computational capabilities of the Crazyflie and enables complex artificial intelligence-based workloads to run onboard, with the possibility of achieving fully autonomous navigation capabilities.

As a last detail, the drone is equipped with a 250 mAh LiPo battery, which provides approximately 7 minutes of hovering. No information is given by the company about the specific model, and the only known information are the dimensions (code 682030, hence it is a 6.8x20x30 mm (HxWxL) battery) nominal voltage (3.7 V), and the maximum charge (2 C) and discharge (15 C) rates.

Note that the drone was very recently discontinued in favour of an updated version (named Crazyflie 2.1+), which features an upgraded battery and different propellers which are able to provide about 15% more battery life. Hence, it must be specified that the model that is going to be used for this thesis is the original Crazyflie 2.1 version. Nonetheless, both the updates introduced in the + version would not have a significant impact on the actual simulations, as the modifications could be easily taken into account with minor adjustments.

[17] `https://www.himax.com.tw/products/cmos-image-sensor/always-on-vision-sensors/hm01b0/`

Before moving to section 3, it may be useful to familiarise with some basic terms of the UAVs motion. In particular, drones have 3 axes, one per couple of degrees of freedom. Those are:

- **Longitudinal (roll)**: has origin at the centre of gravity and is directed parallel to the "forward" direction of the drone. Motion about this axis, called roll, allows for the right and left movements. To achieve it, the rotational velocity of the motors towards the right direction (the two rightmost motors of Figure 2.5) has to be changed with respect to the other, in order to generate a different intensity for the lift for the two sides of the drone;
- **Transverse (pitch)**: has again origin at the centre of gravity but is directed on the axis orthogonal to the roll axis on the plane formed by the propellers. In this case, the motion along this, called pitch, allows for the forward and backwards movements. To achieve it, it is necessary to change the rotational velocity of the motors towards the front with respect to the ones towards the back;
- **Vertical (yaw)**: once again, has its origin at the center of gravity and is directed towards the top of the aircraft, perpendicularly to the propellers. The rotation about this axis, which provides the in-place rotation of the drone, is referred to as yaw. To achieve it, it is necessary to change the speed of either the CW or the CCW propellers in order to make the net torque on the drone body not null.

Note that having same-spin-direction motors on the two diagonals prevents the drone from unwanted rotations during roll and pitch changes, as during these maneuvers two adjacent motors will always have the same absolute speed leading to a zero net torque.

Lastly, one additional term that the reader must be aware of is **thrust**, meaning the net vertical force produced by the propellers that generates the lift of the drone. To summarise, a graphical representation of these terms on a Crazyflie drone is reported in Figure 2.6.

[18]https://labs.dese.iisc.ac.in/embeddedlab/flying-crazyflie-using-hand-gestures/

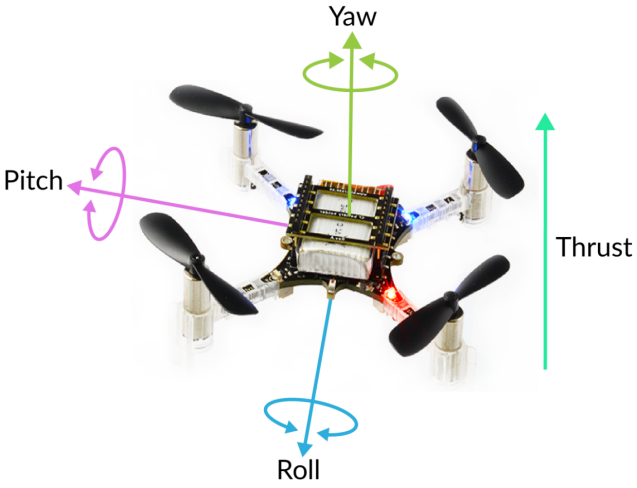


Figure 2.6: Useful terms about UAV motion[18].

Chapter 3

Related works

This chapter has the goal of providing an overview of the most important resources that contributed directly and indirectly to this work, evidencing the current state-of-art in the field of ISS, extra-functional and robotic simulations, along with the most known power consumption models for UAVs. The focus will be oriented towards the works that are the most related to the discussion topics, in order to further clarify their benefits and the limits that this thesis aims to overcome.

3.1 Functional ISS simulation

As mentioned aforehand, ISS simulators mimic the behavior of processors at the instruction set level. This implies that the simulation is limited to a purely functional one, meaning that the sole purpose of these programs is to allow programmers to develop software without the need of hardware prototypes, representing for companies a huge saving in time and cost terms. Among the most important ones, it is worth mentioning:

- **QEMU** [3]: QEMU is a highly versatile emulator that can simulate multiple processor architectures (such as x86, ARM, MIPS, RISC-V) and even full systems including peripherals. It allows to run entire OSs or applications on virtual hardware, making it a powerful cross-architecture ISS that is however best suited for virtualization purposes rather than simulations.

- **RENODE** [4]: RENODE, on the other hand, is a system simulator with ISS capabilities specialised in multi-core embedded systems (mainly ARM, RISC-V, and PowerPC) and their peripherals. It can simulate instruction execution along with elements such as sensors, timers, and communications buses, making it more targeted for systems involving multiple interacting components.

- **Spike** [36]: Spike, instead, is the official simulator for the RISC-V architecture. It is a pure ISS targeting RISC-V based chips only. Contrary to the previous two, it does not simulate any peripheral devices or full systems, hence it is often used in the early stages of development to verify the correctness of software targeting RISC-V cores.

Several higher level simulators exist, such as Simulink [2]. However, Simulink simulates the behavior of systems with much more abstraction, focusing on block-based models rather than instruction-level. This kind of simulators should thus not be included in the list, as they do not simulate the low-level execution of machine code on specific hardware architectures like a typical ISS.

3.1.1 GVSoC

Among the other options, GVSoC [10] is of particular interest for the scope of this thesis, as it is one of the core building blocks of MESSY. It is an ISS developed by GreenWaves Technology that was built to simulate the GAP8 processor (presented earlier in Section 2.1.2).

GVSoC is functionally equivalent to the real RISC-V platform. The compiled code that can run on one of these chip will also run as is on the simulator[1], which is indeed one of the main benefit of an ISS.

However, unlike some higher-level softwares that focus only on instruction execution, this program goes further by providing cycle-level accuracy, meaning that it models the timing and behavior of the system at each clock cycle. This makes it able to perform analysis at the best possible detail level. Moreover, beyond just the cores, GVSoC simulates peripherals, memories, and interconnections, making it a comprehensive simulator that allows to test how software interacts with the entire system, including hardware components like timers, GPIO, SPI, etc.

As a general idea, the system to be simulated is modeled using a component-based approach. Each piece of hardware is a component and is interacting with other components only through bindings. Bindings are connections between a master port of a component and a slave port of another connection. They allow both sides to interact together using function calls. Moreover, each port has a signature, which is the set of methods that a component can call on the other ones.

This component-based architecture extends to both the RISC-V cores and the peripherals: each of those is implemented as a distinct module, communicating through defined interfaces. Cores are connected to memory controllers, I/O peripherals, and other system components via these bindings, enabling a detailed

[1] https://greenwaves-technologies.com/gvsoc-the-full-system-simulator-for-profiling-gap-applications/

interaction and data transfer modeling. The event-driven simulation model ensures that these interactions are synchronized with the system clock, handling in an accurate way events such as instruction executions, memory accesses, and interrupt signals, which have all been timed to match the ones of the real platform.

Overall, GreenWaves declares that the software is able to maintain a timing error lower than 10% for nominal cases, and that this does not exceed 20% even for corner cases[2]. GVSoC is also able to estimate the chip power consumption interpolating the temperature, voltage and frequency data, with a maximum error of 20% with respect to real values[2]. In addition, according to the company, it can simulate up to 20 million instructions per second, which is approximately 10 times less than the actual performance of GAP8[2].

3.1.2 SystemC

SystemC [7] is not an ISS by itself, but rather a language to describe, model and simulate/verify hardware. Nonetheless, it strongly contributes to the functional verification aspect of MESSY.

The language is built by extending the standard C++ with some open-source class libraries. In essence, these modifications allow developers to design entire hardware systems using a component-based approach, where each module represents a specific block (such as a processor, memory, or bus). These modules communicate through ports and signals, which enable interaction between different parts of the system using interfaces. Each component can execute independently, synchronized through the SystemC kernel, which is in charge of managing the scheduling of the events and the simulation time.

The simulation model is event-driven, allowing SystemC to simulate the behavior of these components at various levels of abstraction. In particular, designers can rely on a wide spectrum of approaches, ranging from transaction-level modeling (TLM, where interactions between hardware components are represented as transactions rather than individual signals), to RTL (lower-level technique used for describing digital circuits in terms of registers and data transfers). This makes SystemC particularly effective for functional simulation (or better, verification) of digital systems: through TLM models, designers can verify communication protocols, data transfers, and overall system behavior without needing a fully detailed implementation. This accelerates the prototyping and testing of hardware-software interfaces, facilitating detection of flaws and performance bottlenecks earlier in the design phase, before considering any actual hardware implementation.

[2]https://pulp-platform.org/docs/lugano2023/gvsoc_pulp_anniversary.pdf

3.2 System-level simulation

Despite the usefulness of functional simulators, these kinds of softwares do not include any information about other relevant extra-functional aspects, such as thermal distribution and energy dissipation. The following section explains what SystemC-AMS is and how it can be used to overcome some of these limitations. Then, the discussion shifts to the introduction of the MESSY framework, and how the integration of SystemC-AMS makes the software able to provide a mechanism to model power consumption.

3.2.1 SystemC-AMS

The SystemC-AMS standard [8] introduces system-level design and modeling of embedded Analog/Mixed-Signal (AMS) systems, i.e. heterogeneous systems that involve both digital and analog components. It is an extension of SystemC but, while SystemC focuses on digital hardware and software design and verification, SystemC-AMS expands its capabilities to include analog, electrical, and mixed-signal behaviors, allowing designers to simulate extra-functional properties like power consumption, signal integrity, noise, thermal effects and even mechanical dynamics. These properties are critical in analog and mixed-signal circuits, where the performance of the overall system can be affected by physical factors (which is not usually the case for purely digital systems).

SystemC-AMS uses combination of discrete-time static non-linear and continuous time dynamic linear model abstractions to offer three modeling frameworks:

- **Timed Data Flow (TDF)**: a high-level modeling style that represents data flow between modules with time annotation, useful for modeling continuous signals (e.g., audio, RF signals) and interactions between analog and digital components in a discrete-time modeling style. In TDF, signals are considered as sampled data at specific time intervals, with each sample carrying discrete or continuous values such as amplitudes.

- **Electrical Linear Networks (ELN)**: allows the modeling of linear electrical circuits such as resistors, capacitors, and inductors, and their interaction with digital control circuits thanks to some predefined linear network primitives. It fills the gap between abstract dataflow and more detailed circuit simulation, making it possible to represent both high-level system behavior and specific electrical characteristics in a unified manner.

- **Linear Signal Flow (LSF)**: model of computation that allows the modeling of AMS behaviors using relationships defined by sets of linear differential algebraic equations. It is a continuous-time modeling style that uses discrete,

real-valued signals, resulting in a non-conservative system description that represents the equation system. Signal flow models can be visualized as block diagrams (such as the one reported in Figure 3.1), where each block represents a basic function or component. The connections between blocks define mathematical relationships, which are solved to simulate the system behavior.

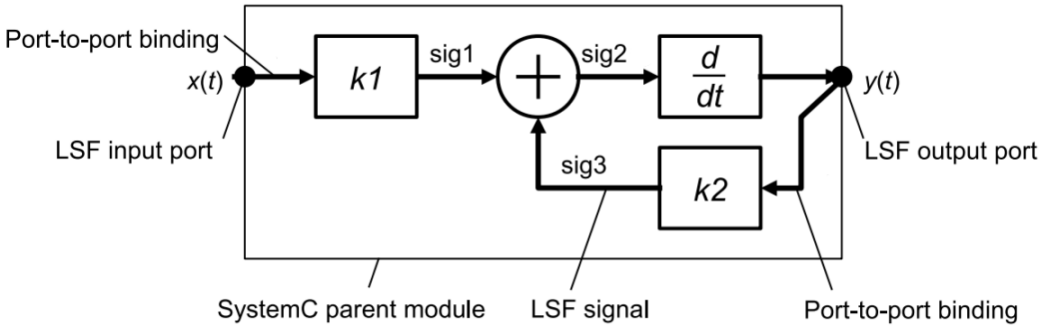


Figure 3.1: Example of an LFS block.

Considering the timing aspect, instead, the SystemC-AMS simulation kernel enhances the standard SystemC kernel through the incorporation of a couple of additional elements: a TDF scheduler (which organizes TDF modules into clusters, and then constructs a static schedule for each cluster) and a linear Differential-Algebraic Equation solver (to manage ELN and LSF models). In the end, a synchronization layer utilises the activation time step of each module, primitive, and cluster to integrate the execution of SystemC-AMS elements into the conventional SystemC simulation flow. The effectiveness of these mechanisms, combined together, made SystemC-AMS a worldwide standard for system-level design and modeling of embedded analog/mixed-signal systems at higher levels of abstraction, resulting in a good balance of modeling accuracy, fidelity and simulation speed.

3.2.2 MESSY

As previously mentioned, the MESSY framework [9] is a recently developed, open-source tool that allows for system-level simulations of RISC-V based chips. At its core, it integrates GVSoC and SystemC-AMS. As seen, the former is used to obtain the functional simulation capabilities for the software running on the chip, while

the latter is included to model extra-functional aspects, with a focus on power storage and distribution.

In order to provide both functionalities, the internal architecture of MESSY exploits a bus-centric paradigm, splitting the two properties in different, interconnected domains. As a consequence, the platform features one bus for the functional aspect (`functional_bus`) and another one for extra-functional ones (`power_bus`). Each system component is connected to each bus by implementing a different model. For example, a sensor will have a couple of implementations: a functional implementation (`_functional`), that describes instruction processing connected to the functional bus and handles the timing behavior of the system, and a power model (`_power`), which estimates the corresponding power demand and exports it (through a DC-DC converter) to the power bus. Speaking of the latter, MESSY uses the power bus to model all the power flow as an aggregation of all the load demands on the bus, in order to determine the amount of current required from the energy sources (batteries, harvesters, etc.).

The functional bus, on the other hand, is principally used for the flow of information when the program executing in GVSoC needs to communicate with one of the components (such as a sensor). In this case, the bus broadcasts the necessary pieces of information (i.e., address, control signals, data), which are then intercepted by the appropriate receiver. Every functional instance of the peripherals can execute different types of actions, each of which is associated with a different power consumption as indicated in the previous paragraph.

The mutual interdependence between the functional and the power models is achieved through an information exchange that is implemented using direct SystemC ports and signals.

Moreover, the SystemC core unit handles the simulation and facilitates the interaction with GVSoC, which is organised as follows. At the beginning, the `sc_main` function (the entry point of SystemC programs) instantiates GVSoC and all SystemC-AMS components. Then, GVSoC initialises its queues by starting SW execution, while SystemC initialises them by executing all components exactly once. From that point on, the simulation proceeds by alternating the execution of GVSoC and SystemC-AMS: first, the functional model of the core invokes the `step_until()` function of GVSoC (which executes the SoC functionality and returns the timestamp t of the following GVSoC event); secondly, it executes a SystemC-AMS wait() until time t (to allow the execution of other SystemC-AMS components while maintaining the temporal alignment among the two simulators). These steps are repeated until the end of the simulation, when control is given back to SystemC to correctly exit the simulation.

All of these features are already integrated in MESSY, and the user is given some useful Python scripts and indications to interact with the framework. In particular, the tool is able to generate ready-to-deploy code by requiring a single

JSON configuration file, limiting the user job to the sole customization of each component/power behavior and the GVSoC program.

3.3 Robotics simulation

Simulations have a critical role also in the fields of robotics. They are used to explore control algorithms and approaches to solve problems, training AI models and, most importantly, mimic the interaction between a robot and its surroundings [2]. As previously discussed, the availability of virtual models for the robots prevents the need of physical machines, thus saving cost and time in the development and test phases of products. In some cases, such applications can be transferred onto a physical robot (or rebuilt) without modification, while in others they serve for preliminary verifications on the robot behavior.

The main use of robotics simulators is to create 3D models of robots and their environments. These simulators provide virtual robots that can mimic the movements of real ones in realistic settings. However, their focus on extra-functional aspects is primarily geared towards simulating external factors like aerodynamics and physical interactions, rather than accurately modeling the internal behavior of the underlying hardware.

Some good examples of general robotics simulators are : Gazebo [37], RoboDK [38] and Webots [6]. In the past years, however, several additional softwares have emerged to better suit the need for modelling of specific types of robots. In the case of UAVs, for example, various softwares have been made available, such as Microsoft AirSim [39] and FlightGear [40], with several studies going over their respective advantages and disadvantages [41].

3.3.1 ROS

Robot Operating System (ROS) [42] is a flexible, open-source framework that provides a collection of tools and libraries for building robotic applications. ROS is designed to handle the complexities of software development for robotics by offering essential capabilities such as hardware abstraction, device drivers, libraries for commonly used functionalities, and tools for system integration, testing, and debugging. One of its core features is the ability to manage distributed computing systems where different parts of the robot (like sensors, actuators, and control logic) can be developed and run separately, often across multiple computers, while communicating efficiently.

[2] https://blogs.nvidia.com/blog/what-is-robotics-simulation/

ROS operates on a publish/subscribe messaging architecture, where different components of a robot, known as *nodes*, exchange information through topics. Each node performs a specific function (e.g., sensor data processing or motor control) and communicates asynchronously with other nodes. A node can publish data (such as sensor readings) to a topic, and other nodes can subscribe to this topic to receive the data. This architecture makes ROS very modular and allows easy integration of various hardware and software components.

Its support for hardware abstraction and sensor integration is particularly useful to create very effective models for complex robots. For example, the Crazyflie drone can be simulated using ROS with tools that mimic its flight dynamics, sensor input, and control systems. However, while ROS is excellent for system-level simulation and software testing, it has some drawbacks when simulating detailed low-level hardware or real-time constraints that may arise in a deeply embedded environment like the Crazyflie. Being the integration of these extra-functional aspects one of the main scopes of this thesis, ROS was thus excluded from the virtual platform, making the MESSY-Webots combination the most advantageous choice. Webots in fact offers more accurate simulations of real-world physics, timing, and resource management. Moreover, it is specifically designed for real-time embedded systems, and hence more suitable for low-level control and hardware/software co-design.

3.3.2 Webots

Webots [6] is an open-source robot simulation project that enables the modeling and simulation of complex robotic systems in highly detailed 3D environments. It is widely used for educational, research, and industrial purposes due to its ability to simulate realistic physical interactions between robots and their environments.

Thanks to its physics engine (fork of ODE) and an OpenGL 3.3 rendering engine (wren), Webots allows for an easy creation of diverse simulation scenarios, ranging from industrial settings to everyday environments, where the behavior and performance of robots can be tested before real-world deployment. All of this can be achieved through the provided asset library, which includes robots, sensors, actuators, objects and materials. In addition to this, Webots allows importing custom CAD (Computer-Aided Design) models of robots, from Blender or from URDF (Unified Robot Description Format, an XML format for representing a robot model). Each robot, sensor, and any object within the environment is a so called *node*, and presents a set of properties that allow users to adapt it to their specific requirements. These properties can be easily modified through the GUI (Graphical User Interface) or within the *Scene* description file (`.wbt`).

A key feature of Webots is the concept of controllers, which are the softwares that govern the robot actions during the simulation. A Webots controller can be written in multiple programming languages, including C++, Python, Java,

and MATLAB, providing flexibility in how the robotic system is programmed and controlled. The controller allows a user to simulate sensor inputs, process data, and define motor commands, offering a realistic platform to evaluate robotic performance in dynamic environments.

Time in Webots is handled using discrete time steps, where each step represents a fixed time interval. This approach allows for step-by-step advancement of the simulation, ensuring that each physical interaction, sensor reading, or control action is updated synchronously. The length of the time step can be adjusted depending on the level of accuracy and performance desired for the simulation: a smaller time step leads to more precise results but requires more computational resources, while a larger time step increases speed but may compromise accuracy. The user can configure the time step by adjusting the `basicTimeStep` parameter (expressed in ms) at the very beginning of the simulation, and this cannot be changed once the simulation is started.

Webots supports ROS, but in this case all the control and handling of the drone sensors was chosen to be given to MESSY: as anticipated in one of the previous paragraphs, there is in fact no need of ROS at all, as MESSY already provides the necessary control and resource management capabilities needed for the chosen scenario. With it, Webots becomes the ideal tool for creating a virtual simulation platform for the Crazyflie, as its ability of creating a sufficiently detailed 3D environments and the possibility of gathering realistic sensor data give to MESSY all the necessary items to model both the functional and extra-functional aspects.

3.4 Drone power models

The following section will present several possible approaches to model the power consumption of the Crazyflie motors. It should be kept in mind that the development of power consumption models for UAVs has primarily been driven by the need to estimate energy usage for transport and delivery applications [43, 44]. Many of the models available in the literature are tailored for larger drones, with a focus on payload capacity and long-range flights. However, these models often do not provide accurate results when applied to smaller UAVs, such as nano-drones, where different physical factors, such as aerodynamic drag and motor efficiency, become more significant in determining energy consumption. However, to the best of the author's knowledge, only very few models target drones of such small dimensions and all of these are generally impractical as they require access to very specific data about the drone structure and dynamics.

This chapter provides a comprehensive review of the existing power models, with a focus on their applicability across different UAV sizes and missions. The discussion will be divided into two parts: the first part will present the main

power models from the literature, categorizing them based on their complexity and the number of input parameters required; while the second one focuses on the development of an empirical model, tailored for the Crazyflie drone, based on direct measurements taken by the company from the robot itself.

3.4.1 Overview of theoretical models

Theoretical energy consumption models are generally based on physical principles such as aerodynamics, battery characteristics, and flight dynamics. Most theoretical approaches consider variables like payload, flight speed, altitude and atmospheric resistance, with the most complex ones adding contributions such as drag coefficients, inertial matrixes and motor efficiency. This chapter reviews the most prominent theoretical models, emphasising their assumptions, input requirements, and applicability. Then, the discussion makes use of a table to evidence whether these models provide a sufficient balance between accuracy and practicality, in order to identify the ones that could be actually selected to be implemented in the power model of the Crazyflie motors.

Note that some more niche models were not included in the discussion because of their higher complexity, but have been briefly described in Appendix A.

D'Andrea (2014)

D'Andrea's model [45] is one of the simplest, focusing on the total mass and the gravitational force acting on the drone, scaled up by the efficiency of the system. It is a foundational equation for estimating energy needs based on basic physical principles, and posed the basis for further models.

$$Power = \frac{\sum_{k=1}^{3} m_k v_a}{\eta r} + p$$

The mass is obtained as the sum of the one of the drone, of the payload and the battery, v_a is the velocity, while η represents the power transfer efficiency for propellers. r is the lift-to-drag ratio and p the power consumption of the electronics.

Dorling et al. (2017)

Dorling et al. [46] expanded the previous model by incorporating aerodynamic factors, particularly air density and velocity, making it more suitable for drones operating at various altitudes. The equation factors in the dynamic nature of flight, emphasising velocity and air resistance, which becomes critical for delivery drones that travel long distances.

$$Power = \sqrt{\frac{T^3}{2n\rho\beta}} = \sqrt{\frac{\left(\sum_{k=1}^{3} m_k g\right)^3}{2n\rho\beta}}$$

In this case, it is considered again the total mass, as well as the number of rotors n, the air density ρ (equal to 1.225 kg/m^3) and the area swept by the propeller blade of one motor (β). This model is one of the most commonly used, such as in [47] and [48]. Note however that it does not depend on velocity: this is due to the assumption that most of the power consumption is caused by the hovering motion of the drone, which has to generate a thrust greater than the gravitational force generated towards the Earth.

Stolaroff et al. (2018)

Stolaroff et al. [49] push the complexity further, modeling energy consumption by considering aerodynamic drag, thrust, and angles of flight. This model takes into account more specific flight mechanics, including induced velocity and angle of attack, and is particularly useful in assessing energy requirements in varying flight conditions.

$$Power = \frac{T^{3/2}}{\sqrt{\frac{1}{2}\pi D^2 n\rho} \cdot \eta}$$

Where

$$T = g\sum_{k=1}^{3} m_k + \frac{1}{2}\rho\sum_{k=1}^{3} C_{d_k} A_k v_a^2$$

and

$$C_D = \frac{2 \cdot m_{\text{body}} \cdot g \cdot \tan(\alpha)}{\rho \cdot v_a^2 \cdot A_{\text{body}}}$$

Moreover, the model says that at high velocities, the following equation is valid:

$$Power = T(v_a \sin\alpha + v_i)$$

Where v_i is the induced velocity and can be calculated as:

$$v_i = \frac{2T}{\pi n D^2 \rho\sqrt{(v\cos(\alpha))^2 + (v\sin(\alpha) + v)^2}}$$

C_D is the drag coefficient, ρ the air density, and A_{body} the area projected by the drone. α is the angle of attack (i.e., the pitch angle), D the diameter of the propeller, n the number of rotors and the other letters resemble the same quantities as per the other equations.

Kirschstein (2020)

Kirschstein [50] equations were built on top of the previous models by integrating additional environmental factors and drag coefficients into the calculation, offering a highly detailed approach that takes into account the drone aerodynamic profile. In this case, the model is expressed as E_{ppm}, i.e. the energy required for steady flight per unit distance [J/m].

$$E_{ppm} = \frac{1}{\eta}\left(\frac{\kappa T W}{v_a} + \frac{1}{2}\rho \sum_{k=1}^{3} C_{d_k} A_k v_a^2 + \frac{\kappa_2 \left(\sum_{k=1}^{3} m_k g\right)^{1.5}}{v_a} + \kappa_3 \left(\sum_{k=1}^{3} m_k g\right)^{0.5} v_a\right)$$

Where:

$$T = \sqrt{\left(\sum_{k=1}^{3} m_k g\right)^2 + \left(\frac{1}{2}\rho \sum_{k=1}^{3} C_{d_k} A_k v_a^2\right)^2}$$

κ is the lifting power markup, W the downwash coefficient and $\kappa 2$ and $\kappa 3$ reflect details of the rotors and environment.

Tseng (2017b)

Lastly, Tseng's model focuses on small payload drones and is tailored for low-speed scenarios. This has been reported as one of the empirical models that can be found in literature. The equation provided is relatively simple (it is made by numerically interpolating measured data) but is highly specialised, focusing on specific drone sizes and operational conditions. It was obtained from a nine-term nonlinear regression model that includes horizontal and vertical speeds and acceleration. Again, it is expressed in E_{ppm}, and it targets the SDR Solo drone.

$$E_{ppm} = -2.595 + \frac{0.197 m_2 + 251.7}{v_a}$$

3.4.2 Final considerations

From a practical standpoint, only the models by D'Andrea, Dorling, and Stolaroff are feasible for the selected use case, as Kirschstein, Tseng and all the others either require specific simulation parameters or are designed for very particular scenarios that do not align with the selected case study. Moreover, it should be considered that some data about the Crazyflie is not publically available. Despite the presence of few resources [51] that detail some of the mechanical and dynamics characteristics of the drone, it is not always possible to fulfill the need of all the data, excluding other model options such as [52] and [53].

Model Name	Input Parameters	Feasibility
D'Andrea (2014)	Mass, Gravity, Efficiency	Yes
Dorling et al. (2017)	Mass, Gravity, Velocity, Air Density	Yes
Stolaroff et al. (2018)	Thrust, Angle, Induced Velocity, Efficiency	Yes
Kirschstein (2020)	Drag Coefficient, Area, Mass, Velocity	No
Tseng (2017)	Payload Mass, Velocity	No

Table 3.1: Summary of energy consumption models and their feasibility

3.4.3 Empirical model

A different approach to power modelling is represented by empirical models, which are based on the availability of the actual drone consumption data for different velocities of the motor. For the case of the Crazyflie drone, Bitcraze itself has proposed on its website an article [54] in which are reported measurements of the drawn current and battery voltage for different RPM (Rotations Per Minute) values of the motors. The team used optical switches to measure motor RPM by logging the timing data at a high sampling rate (500 Hz). For thrust, a simple setup was built where the drone was held down on a scale, allowing the lift to be measured by the weight reduction. Current (amps) was measured with a multimeter in series with the power source, while the firmware was modified to gradually increase thrust and log key metrics like voltage, PWM (Pulse Width Modulation), and RPM.

The results obtained by the company are reported in Table 3.2.

Knowing the power consumption data of the Crazyflie drone from the table, it was possible to create the empirical power model. As a first factor, it should be considered that the relation between drained current and velocity is very close to a linear one, hence a quadratic regression was done on the two variables in order to obtain a function that outputs with a sufficiently good approximation the measured current drain. Then, the relationship between the consumed current and the battery voltage was approximated. This, instead, can be modeled linearly, as the battery voltage drops due to internal resistance: when the current increases, the voltage drop becomes larger, making this linear relationship useful for predictions.

Additionally, a correction factor must be applied because the RPM values simulated in Webots do not perfectly match the real-world RPM. To address this, the drone hovering point (i.e., where the generated lift equals the drone weight) was used to find the correct RPM for hover. This allows the introduction of a multiplicative factor that aligns the simulation RPM with the actual flight behavior.

As a result, the obtained empirical model was the one composed of the following equations:

Amps	Thrust (g)	Voltage	PWM (%)	Average RPM
0.24	0.0	4.01	0	0
0.37	1.6	3.98	6.25	4485
0.56	4.8	3.95	12.5	7570
0.75	7.9	3.92	18.75	9374
0.94	10.9	3.88	25	10885
1.15	13.9	3.84	31.25	12277
1.37	17.3	3.80	37.5	13522
1.59	21.0	3.76	43.25	14691
1.83	24.4	3.71	50	15924
2.11	28.6	3.67	56.25	17174
2.39	32.8	3.65	62.5	18179
2.71	37.3	3.62	68.75	19397
3.06	41.7	3.56	75	20539
3.46	46.0	3.48	81.25	21692
3.88	51.9	3.40	87.5	22598
4.44	57.9	3.30	93.75	23882

Table 3.2: Power consumption measurements for the Crazyflie drone

$$I_{drain} = (0.00100706) \cdot RPM^2 + (-0.0143) \cdot RPM + 0.32618 \tag{3.1}$$

$$V_{battery} = (-0.16302) \cdot I_{drain} + 4.03495 \tag{3.2}$$

$$Power = V_{battery} \cdot I_{drain} \tag{3.3}$$

Note that the equations output the I_{drain} and $V_{battery}$ variables expressed in A and V, respectively.

Chapter 4

Methodologies

4.1 Establishing the connection between MESSY and Webots

As anticipated in Chapter 1, part of the discussion is going to be dedicated to explaining the establishment of the connection between Webots and MESSY. In this case, the mechanism used to create the communication channel was blocking UNIX sockets. Such choice can be easily justified considering three main aspects: first, the two programs are meant to be run on the same machine, excluding the need of other slower mechanisms suitable for computer networks; secondly, the connection is meant to be half duplex, meaning that only one of the two will send data to the other during the execution, excluding the need of pipes; thirdly, they are much easier to implement than others, exploiting simple library functions rather than the more complex synchronisation mechanisms used for shared memory and similar approaches.

In particular, the communication may be established through the use of the methods contained in a user defined library, named `VirtualConnector`, which is described in the following.

4.1.1 The VirtualConnector library

The VirtualConnector library consists of two main classes, `ConnectionConfig` and `VirtualConnector`, each of which plays a different role in managing the communication between the software and Webots. Note that these have been developed in C++, as this is the language used on both MESSY and Webots.

The `ConnectionConfig` class is responsible for setting up and managing the UNIX socket connection. It provides several methods: `initialize_connection` to establish a connection to the server, `get_server_FD` and `get_connection_FD`

to retrieve the file descriptors, and `close_connection` to terminate the connection. This set of functions is rather simple, using only a few system calls that operate on the two file descriptors (one for the server, and one for the connection) objects directly contained in the class. Among those, the `initialize_connection` is surely the most important. Its code, reported in the Listing 4.1, works as the client side explained earlier in Section 2.3:

- it creates a UNIX socket with the `socket(AF_UNIX, SOCK_STREAM, 0)` command;
- initialises the address with the `sockaddr_un` and sets the type to `AF_UNIX`;
- sets the socket path by copying `socket_path` in `server_address.sun_path`;
- attempts the connection to the server with the `connect` method.

In case any error arises, the method returns a -1 value.

Listing 4.1: Implementation of the connection initialization method

```
int ConnectionConfig::initialize_connection() {
    struct sockaddr_un server_address;

    if ((this->server_fd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1)
        return -1;

    memset(&server_address, 0, sizeof(server_address));
    server_address.sun_family = AF_UNIX;
    strncpy(server_address.sun_path, this->socket_path, sizeof(
    server_address.sun_path)-1);

    if ((this->connection_fd = connect(this->server_fd, (struct
    sockaddr*) &server_address, sizeof(server_address))) == -1)
        return -1;

    return 0;
}
```

Together, this and the other `ConnectionConfig` functions, which will not be commented further for brevity concerns, ease the management of the socket, while the actual data exchange (specifically, reading and writing JSON-formatted messages through the socket) is instead handled by the other class, `VirtualConnector`. In particular, this class uses the JSON library (the nlohmann/json library [55]) to receive and send JSON-formatted data. The two functionalities are achieved using two methods, included in Appendix B, which are described in the following paragraphs.

The first method, `read_from_channel`, is used to read data from the socket. In particular, it reads the length of the incoming packet, dynamically allocates

memory for the message to be received, and parses it into a JSON object. More in detail, the algorithm (whose pseudo code is reported in Algorithm 1) works in 6 fundamental steps. First, after initializing some variables, it gets from the channel the amount of bytes to be read and saves it in `rd_len_buffer`. This value is then converted into an integer in step 2 and it is used in step 3 to reserve in memory (in the space pointed by `rd_json_buffer`) the correct amount of bytes. Next, a `while` loop performs several reads on the socket channel up to when either the number of read bytes exceeds the expected one or if no byte was received at the last read. Note that the reading functionality must be split in several iterations in order to ensure the correct receival of the packet also for messages exceeding the maximum possible dimension per single transfer (which depends on several factors). As last steps, the code parses the received message into a JSON object (returned at the end of the function) and frees the memory allocated to accommodate the packet.

Conversely, `write_on_channel` converts a JSON object into a string and sends first its length and then its content via the socket (functioning in a way that is complementary to the `read_from_channel` explained earlier). The function, whose pseudo code has been reported in Algorithm 2, works in 5 steps. In the first one, the `data` JSON object passed to the function is converted into a string, in order to be inserted in a packet, and its length is evaluated. Then, this length is used to create a 4-byte value in step 2, which is sent on the socket on the next step. A new memory section is allocated knowing the number of bytes to be sent and, after copying the `data` JSON object to this space, the information contained in such buffer is sent on the socket. As a last step, the memory is freed and returned to the OS.

The modularity of the code, along with the divisions between socket management (`ConnectionConfig`) and data handling (`VirtualConnector`), enhances the flexibility and reusability of such functions inside all the components that must communicate with Webots through the socket. Moreover, the inclusion of JSON further simplifies this standardised exchange of data, making the communication between the software and Webots more efficient and human readable.

4.2 Crazyflie architecture modelling in MESSY

As anticipated earlier in Chapter 2.4.3, the processing power of the Crazyflie 2.1 drone is composed of a couple of chips: an STM32 and an nRF51822. The STM chip handles most of the drone computational tasks, including the flight control, the sensor data processing, and the execution of the main flight algorithms. It runs the core firmware and manages the motors and communication interfaces. On the other hand, the nRF chip is primarily responsible for wireless communication and power management. It uses Bluetooth and radio protocols to enable remote control

Algorithm 1 VirtualConnector::read_from_channel pseudo code

```
1: procedure READ_FROM_CHANNEL(fd)
2:     ▷ Initialize variables
3:     num_read ← 0, counter_read ← 0, string_length ← 0
4:     rd_len_buffer ← [0, 0, 0, 0]
5:
6:     ▷ Step 1: Read the length of the incoming JSON packet
7:     num_read ← READ(fd, rd_len_buffer, 4)
8:
9:     ▷ Step 2: Calculate the JSON string length
10:    for i = 0 to num_read do
11:        string_length ← string_length | (rd_len_buffer[i] ≪ (8 ∗ i))
12:    end for
13:
14:    ▷ Step 3: Allocate memory for the JSON string
15:    rd_json_buffer ← ALLOCATE_MEMORY(string_length + 1)
16:
17:    ▷ Step 4: Read the actual JSON string
18:    while counter_read < string_length do
19:        num_read ← READ(fd, rd_json_buffer + counter_read, string_length
   - counter_read)
20:        counter_read ← counter_read + num_read
21:        if num_read == 0 then
22:            break
23:        end if
24:    end while
25:    rd_json_buffer[counter_read] ←′ \0′          ▷ Null-terminate the string
26:
27:    ▷ Step 5: Parse the JSON string
28:    data ← PARSE_JSON(rd_json_buffer)
29:
30:    ▷ Step 6: Free memory and return parsed data
31:    FREE_MEMORY(rd_json_buffer)
32:    return data
33: end procedure
```

Algorithm 2 VirtualConnector::write_from_channel pseudo code

```
1: procedure WRITE_ON_CHANNEL(fd, data)
2:     ▷ Step 1: Convert the JSON object to a string
3:     json_string ← DATA.TO_STRING
4:     json_length ← LENGTH(json_string)
5:
6:     ▷ Step 2: Prepare the length of the JSON string to be sent
7:     length_buffer[0] ← json_length&0xFF
8:     length_buffer[1] ← (json_length ≫ 8)&0xFF
9:     length_buffer[2] ← (json_length ≫ 16)&0xFF
10:    length_buffer[3] ← (json_length ≫ 24)&0xFF
11:
12:    ▷ Step 3: Send the length of the JSON string
13:    WRITE(fd, length_buffer, 4)
14:
15:    ▷ Step 4: Send the JSON string itself
16:    wr_buffer ← ALLOCATE_MEMORY(json_length)
17:    COPY_TO_BUFFER(wr_buffer, json_string, json_length)
18:    WRITE(fd, wr_buffer, json_length)
19:
20:    ▷ Step 5: Free the allocated memory
21:    FREE_MEMORY(wr_buffer)
22: end procedure
```

and data transmission between the drone and a mobile device.

However, in this work it was also considered the presence of the AI deck, which introduces a GAP8 chip and a camera. This AI deck allows the Crazyflie to perform more complex tasks, and it should be included as it is needed for the case study scenario of this thesis, which is described in the following.

This work aims at simulating a scenario in which a Crazyflie drone is autonomously navigating in a drone racing environment, where it must start itself, fly through a sequence of gates and finally come back to the start position and stop the motors. Considering this setting, the main computational task is handled by the GAP8 chip, which processes images captured by the onboard camera using a neural network. Based on the results of this processing, the GAP8 determines the next drone movement coordinates, which are then transmitted to the STM32 chip. In turn, this is responsible for converting them into actual motor commands, allowing the drone to adjust its flight and pass through the gates accurately.

The GAP8 chip is emulated using GVSoC, allowing for a reproduction of its behavior in the real hardware. Meanwhile, MESSY is used to model both the power consumption and the functional behavior of other key components, such as the camera sensor, the STM32 chip, the battery, and the motors (motors are also critical, as they receive the processed flight commands and execute the physical movements of the drone). However, not all the real components of the Crazyflie have been modeled accurately: for example, the nRF chip was excluded, due to the difficulties that were encountered when modelling its power consumption (which is strongly correlated to the firmware running on the chip itself). Nonetheless, its contribution to the energy balance of the device was taken into account in the consumption model of the motors.

Webots, instead, plays the role of handling the drone interaction within the 3D environment. In addition to simulating the physical movements of the drone, Webots is also responsible for emulating certain sensors that are necessary for the drone navigation but for which precise power consumption data (which are needed for a proper model on MESSY) is not available. In particular, those are the GPS (Global Positioning System), the gyroscope and the IMU (Inertial Measurement Unit), which are essential for the functioning of the PID.

Another goal of this setup is to ensure that the entire system operates in real-time, with a target frame rate of 30 FPS (Frames Per Second). To meet this requirement, the process of capturing an image, running the NN inference on the GAP8, and translating the resulting coordinates into motor commands must be completed within a 33 ms time window. This ensures that the drone can timely react to environmental changes and maintain a smooth and efficient flight path during the race.

This chapter presents an overview of the development process for each of the drone parts to be modeled on MESSY, grouping them in categories according to

the similar logic functions. Before moving on, note that the discussion is going to deal only with the modifications introduced to tailor the simulation environment for the Crazyflie drone. Hence, no in-depth description will be proposed for the modules that were left untouched with respect to the automatically generated ones, such as the core and some battery submodules.

4.2.1 Sensors and SoC: camera and STM32 microprocessor

Sensors introduction and simulation overview

Most of the drone components are simulated by means of customized sensors. In MESSY, sensors are the main entities that can be used to simulate the various subparts of an embedded system, for which a code can automatically be generated starting from the JSON configuration file. As partially anticipated in Section 3.2.2, every sensor is composed of two distinguished parts: a **functional** and a **power** one. Each of those is associated with a different piece of code, which is automatically generated and integrated with the rest of the system.

Upon generation, in fact, MESSY creates per each of the sensors configured in the JSON file a new SystemC module for the functional part and a SystemC-AMS module for the power part. The former is responsible for the definition of the internal characteristics of a sensor. It is a standard `SC_MODULE` already equipped with a series of signals that integrate it with the functional bus, in order to enable the communication with the core (for which a different couple of functional/power codes are also generated). Along with those, a `sensor_logic` method is created, as better explained later in the document.

Speaking of the power side, a `SCA_TDF_MODULE` is used instead. This code in fact has to include a set of analog-mixed signals that connect to the power bus, and which are used to represent the behavior of the voltage and current of the device in order for the power source(s) to accurately simulate the power consumption. The power side also controls the "status" of the sensor, as better detailed later in the discussion.

As a general idea, the flow of operations of any simulation on MESSY has been reported in Figure 4.1.

The `sc_main` program initializes all the modules (the core, the sensors, the buses, etc.), connects them with the signals and starts the simulation with a `sc_start` command. At this point, the core (i.e., the GAP8 chip simulated by GVSoC) runs the main program and interacts with the various peripherals/sensors by accessing the associated memory space. Once this happens, a transaction on the functional bus is created and the functional part of the involved sensor retrieves the details of the request, such as the address, read/write flag, and data. Once parsed, the request is handled by the sensor logic which, at the current state of MESSY, is composed by default of a set of "statuses", identified with sequences of

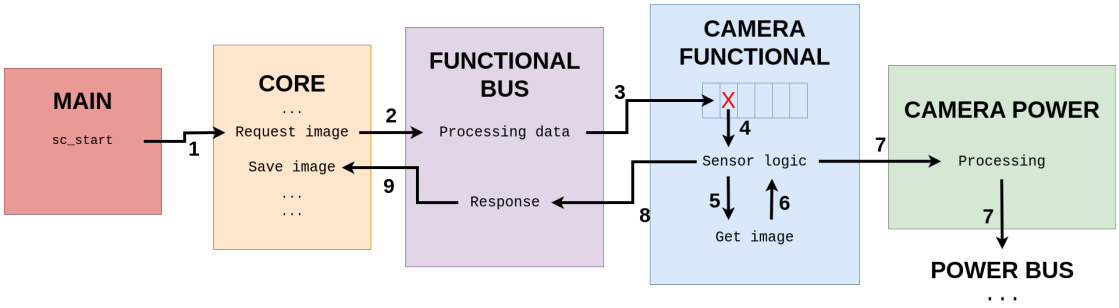


Figure 4.1: Visual representation of the MESSY default flow.

`if` statements, which trigger a certain logic reaction on the basis of the request that has been received. After this step, the power instance is activated. Each status is associated with a specific current draw from the power source(s) and a specific delay: this information is used to evaluate the energy that is associated to that specific status of the sensor, while also advancing the SystemC simulation core to mimic the time taken to handle the request. After these operations, the power instance sets an IDLE state and gives the control back to the functional instance, which can then prepare the output signals accordingly to reply to the core.

The simulation keeps running in this manner until the end of the GVSoC program has been reached. Once the remaining pending operations on the SystemC side have also been completed, the simulation stops and `sc_main` returns.

As said in the previous paragraph, the core has to target a certain address to interact with a specific sensor. In MESSY, each sensor can be configured to have a specific amount of memory, which is mainly used to mimic the presence of status registers, control registers, and the internal memory. The capacity of each sensor is defined in the JSON configuration file, with the value expressing how many bytes (8-bit registers) are at disposal of such peripheral. Once MESSY is instructed to build the SystemC code on the basis of the given configuration, it creates per each sensor a `register_memory` private `uint8_t` array variable inside the functional part of the code. This memory is the same one used by the automatically generated `sensor_logic` method, which is able to understand when and where a read/write operation has occurred, and instructs how to react accordingly (this is in fact the function that needs to be targeted by the developers in order to specify any of the sensor functionalities).

The focus is now going to shift towards the specific cases of the camera and STM32, highlighting how their `register_memory` is organized and how the `sensor_logic` was modified to provide the wanted functionality on the basis of the register the core is interacting with.

Camera

The camera mounted on the AI deck expansion of the Crazyflie drone is an HIMAX HM01B0-MNA-00FT870[1]. As previously noticed, the camera produces 320p monochrome images that are supposed to undergo the NN inference to produce the next target coordinates.

To model the behavior of the real camera, the simulation environment has to work as follows. First, the program running on GVSoC should be able to inform the camera sensor that an image is needed. Then, the camera module of MESSY (`sensor_camera_functional`) should react and request an image from the camera sensor present in Webots. Once the photo is shot from the sensor, this has to be sent back to the camera module of MESSY, which in turn informs GVSoC of the availability of the image. In order to do so, the `sensor_camera_functional` module should have a control register to handle the image request, a status register to be polled by GVSoC to inform when the image is ready and a memory space to save the image information, in order for the GAP8 to retrieve it. The register structure was thus organized as shown in Table 4.1.

Reg	Description	Functionality
0	Control register	IF 1: GET IMAGE, asks the drone to capture an image
1	Status register	IF 1: GET IMAGE COMPLETED, image was successfully taken
2-102401	Image registers	Store the image information

Table 4.1: Camera Registers

A 1-byte control register was created for the request to happen. When the GAP8 chip writes a 1 in such register, it triggers the `sensor_logic` function, which handles the process of getting the image from Webots. Once this has been obtained, the image is stored in the registers from 2 to 102401 (which have a total capacity of 102400 bytes, i.e. the space needed for a 320x320 monochrome image) and the 1-byte status register is updated with a 1 to flag the completion of the process.

In order for the module to provide the functionality of getting the image, two additional methods were created: `get_camera_image` and `json_parser`. As the `sensor_logic` function senses the `enable` and `ready` signals to be high and the `flag_wr` to be low (meaning the sensor is enabled and a read is being performed on it), it extracts the `add` address and uses it to populate its memory with the

[1] `https://mm.digikey.com/Volume0/opasdata/d220001/medias/docus/4886/HM01B0-MNA-00FT870.pdf`

passed data. If the target of the message is the control register and a GET IMAGE command was issued, the module reacts by calling the `get_camera_image` function.

This, in turn, writes on the socket a JSON packet with a specific request (`command = GET_IMAGE`) that has to be sent to Webots. Then, it brings the camera module in a wait state by using a blocking read until the image arrival. As the picture has been obtained from Webots by means of another JSON packet, it calls the `json_parser` method. This loops through the JSON object searching for the key "camera" and, when this key is found, it retrieves the width and height of the image followed by the data from the image itself (stored as an array of unsigned characters). The image is retrieved (with an index that uses the width and height to loop along the pixels) and then transferred into the `register_memory`, where each byte is stored sequentially starting from the third index (i.e., skipping the control and status registers). At the end of the transaction, the control is handed back to the `sensor_logic` function and other signals are activated to inform the power instance of the sensor about the current consumption. Once the function completes, the sensor goes back to the `wait` state. From that point on, any change in the `ready` signal will trigger again the execution of `sensor_logic`: this happens because of the inclusion of such signal inside the sensitivity list of the module. To ease the reading, a scheme of the communication happening between the functional bus, the camera and Webots has been reported in Figure 4.2.

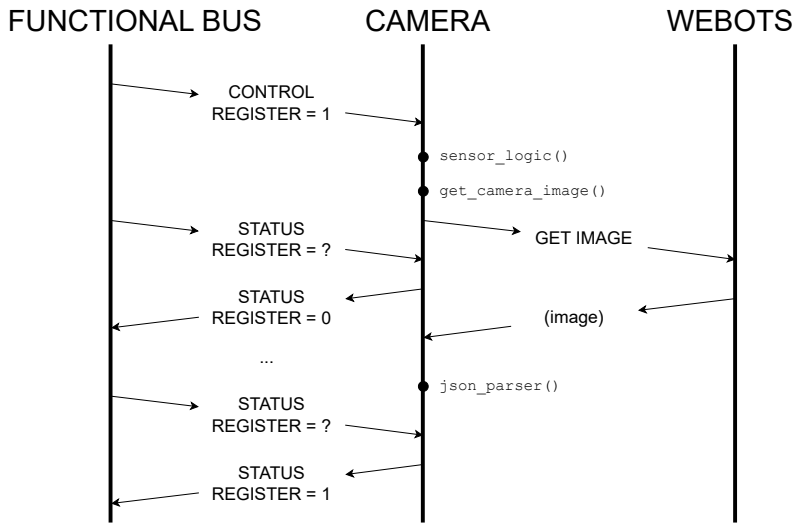


Figure 4.2: Visual representation of the bus-camera-Webots interaction.

Note that the `sensor_logic` function also advances the simulation by the amount of time needed by the real sensor to capture an image. In this case, a value of 16 *ms* was chosen, in order to be compliant with the 60 FPS specification of

the real camera. By default, MESSY also asks to specify a delay for the cases in which GVSoC performs a write on the registers with respect to a read. Its value (set to 4 ms) is not necessary nor relevant for the scope of this thesis, and was included simply to maintain the original code structure. In this case, the function calls the `advance_simul` method, whose details are going to be described later in the chapter.

The module also has a `set_connection_parameters` function and a `ConnectionConfig` attribute: these are needed in order to ensure that the camera is able to correctly receive and save the socket file descriptor during the execution of the `sc_main`.

As a final remark, be aware that the set of operations described for the `sensor_logic` function, the presence of the `advance_simul` and `set_connection_parameters` methods are going to be common to every sensor, hence will be omitted in the next descriptions for brevity concerns.

STM32

The STM32 microcontroller is responsible for managing the control of the drone motors, handling the received commands from GVSoC, and processing sensor data. More in detail, the tasks of this module are setting the target positions, initializing the PID controller for motor control, and managing the power consumption based on the drone operations. On MESSY, the architecture of this sensor can be implemented by simply including a control register and a status register (allowing to check which type of operation is needed and to signal the completion of a task), as well as a memory region to store the target positions that have to be passed to the PID when setting a new destination. About the latter, three double (8 bytes) values are needed to store the X, the Y and the Z coordinates. The resulting register configuration of the sensor has been reported in Table 4.2.

Reg	**Description**	**Functionality**
0	Control register	See Table 4.3
1	Status register	See Table 4.3
2-9	Target Position X	Target position for the X coordinate
10-17	Target Position Y	Target position for the Y coordinate
17-25	Target Position Z	Target position for the Z coordinate

Table 4.2: STM32 Registers

Due to the large quantities of operations to be performed by the STM32 module, a different table (Table 4.3) was created to recap those and the associated control/status register codes. Similarly to the prior case, a name is associated to

each command, in order to easily identify it (in the code, this corresponds to the usage of several `#define` statements).

Code	Name	Control Register Function	Status Register Function
1	INIT PID	Runs the function to initialize the PID parameters	PID initialization was completed successfully
2	GET STABLE	Drone lifts from ground and reaches a height of 1 m	Drone is now stable at 1 m +/- tolerance
3	SET TARGET	Sets the program to update the target	Target was correctly set
4	RUN SINGLE STEP PID	Runs a single step of the PID	PID single iteration has ended
5	RUN PID	Runs PID until the target reached	Target reached (continuous run of the PID case)
6	TARGET REACHED	—	Target has been reached (step-by-step run of the PID case)
7	GO DOWN	Signals the STM32 the need for landing the drone	Landing has completed
8	PRINT SIMUL TIME	Prints the current simulation time + offset	—
9	BATTERY LOW	—	Battery is too low
10	ITER ADVANCE	Advances the Webots simulation	Simulation was correctly advanced

Table 4.3: STM32 control/status register commands

Follows a high-level description of the functions implemented inside the STM32 module.

As per the camera, the `sensor_logic` function continuously monitors the control register of the STM32 to handle the requests from GVSoC, reading the address to which they are destinated as well as the data. Depending on the command that was written on the control register (like `INIT PID`, `GET STABLE`, or `SET TARGET`), it triggers the corresponding actions, which is often associated to the execution of one of the other method contained in the STM32 class. At the completion of such method, the status register is updated with the correct code in order to inform GVSoC.

In case a PID INIT command is received, `PID_init` is called. The function initializes the PID controller by setting the initial conditions for the motors PID. It also communicates with Webots to retrieve some necessary data: among all,

this function obtains the simulation timestep, which is fundamental for both the PID and the correct compensation of the sensor delay offsets. In the meanwhile, it ensures the drone remains stable at a defined target position until new instructions are received.

Once the initialization is completed, `get_stable` can be called: this forces the drone to lift from ground and reach a stable altitude of 1 m by adjusting the PID controller accordingly.

Once it has reached a stable position, the drone can start receiving commands to move around the environment. This functionality can be achieved by means of several methods, and it may happen in a controlled or partially autonomous manner. In the former case, the PID iterations are executed once at a time in a controlled manner, in order to simulate the single iterations that would occur in the real case scenario of the drone continuously capturing images and moving one step at a time. In the latter case, instead, the iterations are called repetitively until the drone has reached the target position, without further intervention from GVSoC. That happens if a RUN PID command is sensed at the control register, which triggers the `PID_loop` function.

In both cases, however, the central method that runs the de-facto PID is `PID_iteration`. This function will be better detailed later in Section 4.2.2, but for now the reader should be informed of the fact that it uses `get_sensor_data` to read some data from the Webots simulation, including its current position, orientation, and altitude. This information is then used to generate the motor commands by calculating the desired adjustments to the drone position. Moreover, it then updates the motor power values accordingly to the `send_data_to_motor`, which sends the updated motor velocity values to the motors (which in turn sends them to Webots and waits for an acknowledgment before completing the command). This function exploits a direct connection with the motors, and uses the `send_command` port to inform the presence of new motor values and the `send_completed` one to understand when the information was correctly received and processed. `PID_iteration` is either triggered by the `PID_loop` or the receival of a RUN SINGLE STEP PID: in the latter case, before returning, the `sensor_logic` function checks both if the battery is low (setting the status to BATTERY LOW) or if the target position of the PID was met (in such case, the target position of the PID is updated according to the logic explained in section 4.2.2).

In case GVSoC wants to advance the simulation by a specific amount of time (in order for example to simulate the time elapsed during the running of the inference of the neural network), it can do so by writing in the control register of the STM32 sensor a number greater than ITER ADVANCE (which corresponds to a 10). This value, which is going to be decreased by 10, represents the number of ms that both the MESSY and the Webots simulation should advance of.

As a target position is reached (for example, a gate was just traversed) and

there is the need of setting a new target destination for the PID controller, this can be achieved by writing a SET TARGET command. In this case, the `sensor_logic` function retrieves the data from the target position registers (2-25) and sets the internal target values of the PID, resetting the target reached flag.

Once the drone has completed the race, it can reach the ground again and stop the motors. This action is triggered by the GO DOWN command: in this case, the `sensor_logic function` communicates with Webots to set the new target coordinates to the ones of the ground. This action is split in two steps (setting the target position slightly above ground first, then to the height of 0 m) in order for the drone to reach the ground with a lower speed and avoid damages.

Finally, as suggested by the name, the `end_simulation` function is called to communicate the stop of the simulation to Webots. In turn, Webots slows down the motors to a null velocity and answers back once it confirms that those have been successfully powered off. The `end_simulation` function also handles the cleanup of the simulation and synchronisation between MESSY and Webots.

Once the simulation has correctly ended, GVSoC can call a PRINT SIMUL TIME to launch the `time_print` function, which outputs the current time and estimated error between MESSY and Webots time to assess simulation accuracy.

It should be evident how the connection with Webots is necessary for the simulation flow: the PID needs to receive constant data from some sensors, and variables such as the position in space of the drone are essential pieces of information for the STM32 sensor in general.

For this reason, it might be noteworthy to briefly mention how the packets exchanged with Webots are generally formatted. The messages that go through the socket use the JSON format, and they all contain a `command` field in which the type of operation that is needed is reported: Webots checks this field to interpret the received message and plan its actions accordingly. Then, depending on the function, several other entries are added to this, such as `COORDINATE_X` to request the X position of the drone from the GPS or `ROLL` to get the rolling angle.

Before moving on, it should be noted that, as said, sensors are supposed to advance the simulation time while performing some actions. Different actions should correspond to different times: the execution time of a single calculation may greatly differ from the run of a complex algorithm for example. Unfortunately, due to the impossibility of knowing the time consumed by the STM chip to perform any of the actions considered in the simulation (such as converting the coordinates into motor commands, and the time taken to inform the motors), the timing information was excluded from most of the operations. As a result, the points in which the simulation is advanced are:

- during the initialization of the PID, as Webots requires a 2000 ms delay when starting the simulation to ensure the synchronization of the system components;

- during the set of the target position, with a delay of 5 ms;
- during the receival of an ITER ADVANCE;
- during `end_simulation`, where Webots informs MESSY of the time that has elapsed while shutting down the motors

Also in this case, the `sensor_logic` function includes the case in which a write is performed on a sensor. This action is set to last 1 ms, even if, in reality, this operation can not happen.

4.2.2 PID controller

By default, the firmware running on the real Crazyflie 2.1 drone utilizes a PID controller to manage the drone state. More in detail, it employs distinct PID controllers for each control level: position, velocity, attitude, and attitude rate. The attitude refers to the orientation of the drone in space, including its roll, pitch, and yaw angles. In contrast, attitude rate is the rate at which these orientation angles change over time, i.e. measuring how quickly the drone is adjusting its roll, pitch, or yaw.

The output of each controller feeds into the input of the next one, forming a cascaded PID structure. Depending on the control mode, different setpoints can be fed into the system, influencing which PID controllers are activated. For instance, when using attitude rate setpoints, only the attitude rate PID controller is active; for attitude setpoints, both the attitude and attitude rate PID controllers are used; and so on for velocity and position setpoints. Ultimately, regardless of the control mode, the angle rate controller translates the desired angle rates into PWM commands for the motors.

In the simulation environment considered in this work, instead, the control of the Crazyflie drone was given to a single PID function, which is in practice the `PID_iteration` function contained in the STM32 module. In order for this to work, a couple of files were also needed: `pid_controller.hpp` and `pid_controller.cpp`. These two contain some classes, methods and attributes definitions which provide the complete set of items that one may need to control the simulated version of the device, and partially resemble the ones running on the actual firmware of the drone. Note that these files were created on the basis of the Webots controller files already made at disposal by the Bitcraze company itself [56]. However, some important modifications were introduced to convert the given C code in a C++ class (`PIDController`) with additional control functionalities. In order for it to be included in the STM32 component, it was simply needed to include the two files and instantiate an object of the `PIDController` class into the set of the private attributes of the module.

To start, it is important to recall that the `PID_init` function must first be called before any usage of the PID: this function is used to initialize variables such as the current position of the robot in the world, the various velocities and angles, the PID gains (the proportional (`kp`), derivative (`kd`), and integral (`ki`) gains for velocity and attitude), the current simulation time and the timestep. From that point on, the PID iterations may be run consecutively to move the drone. The position that is used as a current destination is specified by the TARGET (x, y, z) coordinates: together, these represent the point in the Webots simulation environment that the PID uses to compute the error signal. Remember in fact that the idea is that, at each iteration of the PID, the STM32 sensor computes the error between the current and the target position and uses it to evaluate the velocities to be applied to the motors at that specific timestep to smoothly reach the destination.

The algorithm running on the STM32 module has been reported in the form of a pseudo code in Algorithm 3, while the complete version may be found in Appendix C instead. Note that the code presents some variables which are local and some others belonging to the `PIDController` class, which are used by the module thanks to the inclusion of a `pid` attribute among its private objects.

First, the needed sensor data is obtained through the `get_sensor_data` method, and variables like the global X and Y velocities (`vx_global` and `vy_global`) are computed based on the change in position over the time elapsed since the last iteration. Then, the global velocities are transformed into the drone body-fixed frame using its yaw (rotation), with the cosine and sine of the yaw angle to compute the body-fixed velocities `vx` and `vy`. Next, the set of desired state variables for the drone movement (roll, pitch, velocity, yaw rate) is initialized to zero. At this point, the algorithm computes the difference between the drone current position (`x_global`, `y_global`, `altitude`) and the already-set target position (`targetposition`) for the three axes. This information is used to set the desired state variables, which are updated depending on whether the difference on each axis is positive or negative. Now, the `pidVelocityFixedHeightController` can be called passing these values. The function computes the motor power values (`m1`, `m2`, `m3`, `m4`), which are then sent to the motors module that forwards them to Webots. Finally, the function finishes by updating the past state values (`past_time`, `past_x_global`, `past_y_global`) for the next iteration.

In order to work, the `pidVelocityFixedHeightController` makes use of the functions included in the `PIDController` class. In detail, it uses the following sequence of methods. First, it adjusts the horizontal velocities with `pidHorizontalVelocityController`, which works by adjusting the roll and pitch based on the velocity errors in the X and Y directions. Then, it handles the altitude with `pidFixedHeightController`: this function computes the error between the desired and actual altitude, applies PID control, and adjusts the altitude. Next, the attitude is changed by means of `pidAttitudeController`, which adjusts the

Algorithm 3 PID_iteration pseudo code

procedure PID_ITERATION
    ▷ Step 1: Get sensor data
    GET_SENSOR_DATA

    ▷ Step 2: Calculate time difference and velocities
    $pid.time\_dt \leftarrow pid.currenttime - pid.past\_time$
    $vx\_global \leftarrow \frac{pid.x\_global - pid.past\_x\_global}{pid.time\_dt}$
    $vy\_global \leftarrow \frac{pid.y\_global - pid.past\_y\_global}{pid.time\_dt}$

    ▷ Step 3: Compute body-fixed velocities
    $cos\_yaw \leftarrow \cos(pid.actualYaw)$
    $sin\_yaw \leftarrow \sin(pid.actualYaw)$
    $pid.actualstate.vx \leftarrow vx\_global \cdot cos\_yaw + vy\_global \cdot sin\_yaw$
    $pid.actualstate.vy \leftarrow -vx\_global \cdot sin\_yaw + vy\_global \cdot cos\_yaw$

    ▷ Step 4: Initialize desired state values
    Initialize *pid.desiredstate* variables to 0

    ▷ Step 5: Calculate differences between current and target positions
    ▷ Step 6: Determine the maximum axis deviation
    See Listing 4.2

    ▷ Step 7: Compute movement coefficients for shortest path
    ▷ Step 8: Update desired velocities based on position difference
    ▷ Step 9: Check if the target is reached
    See Listing 4.3

    ▷ Step 10: Generate control values through the PID
    PIDVELOCITYFIXEDHEIGHTCONTROLLER

    ▷ Step 11: Communicate the velocities to motors
    Write m1, m2, m3, m4 to the ports communicating with the motors
    SEND_DATA_TO_MOTOR

    ▷ Step 12: Update some variables for the next iteration
    $pid.past\_time \leftarrow pid.currenttime$;
    $pid.past\_x\_global \leftarrow pid.x\_global$;
    $pid.past\_y\_global \leftarrow pid.y\_global$;
end procedure

roll, pitch, and yaw using PID control based on the errors between the desired and actual attitudes. As a final step, it converts the roll, pitch, yaw, and altitude commands into the individual motor values with `motorMixing`.

It is important to evidence that, while the PID is able to adjusts the drone altitude, roll, pitch, and yaw using the position/velocity feedback and driving motor commands based on the difference between the current and desired states, no mechanism is present in order to know when the target position is met. Moreover, no control is performed to ensure that the drone follows the shortest trajectory to the destination and some errors may generate in case the target position may not be reached, for example, due to obstacles. To solve these issues, the following modifications were introduced into the algorithm.

In order to ensure that the drone follows the most direct path (i.e., the shortest) to the target, the algorithm was modified as shown in Listing 4.2

Listing 4.2: Implementation of the straight path

```
1   ...
2   double x_diff = pid.x_global - pid.targetposition.x;
3   double y_diff = pid.y_global - pid.targetposition.y;
4   double z_diff = pid.actualstate.altitude - pid.targetposition.z;
5
6   double x_abs = abs(x_diff);
7   double y_abs = abs(y_diff);
8   double z_abs = abs(z_diff);
9   double max_abs = 0;
10
11  if (x_abs > y_abs) {
12      if (x_abs > z_abs) {
13          max_abs = x_abs;
14      }
15      else {
16          max_abs = z_abs;
17      }
18  }
19  else {
20      if (y_abs > z_abs) {
21          max_abs = y_abs;
22      }
23      else {
24          max_abs = z_abs;
25      }
26  }
27
28  double x_coeff = x_abs/max_abs;
29  double y_coeff = y_abs/max_abs;
30  double z_coeff = z_abs/max_abs;
31  ...
```

In essence, after having evaluated the differences from the current and the target position in `x_diff`, `y_diff` and `z_diff`, the absolute value of these is calculated, and the direction with the greatest value is saved in `max_abs`. Then, a new coefficient is generated per each axis by scaling the (absolute) difference with respect to the maximum one: in this way, the PID makes sure to orient its corrections on the axis that needs it the most, producing a response that is proportional to the magnitude of the distance of each axis from the target position. Conveniently enough, these modifications allow to generate the linear trajectory to the destination, as the normalised values are then used to evaluate the magnitude of the desired movements (sideways, forward, height difference).

To make the system able to evaluate whether the destination position is met or not, instead, the concept of `tolerance` was introduced. The drone receives a target position, and the system considers it to be effectively reached only if the drone has remained for enough time in the close proximity of such coordinate. In order words, the target position is not a point anymore, but the set of points that are within a distance of `tolerance` from the destination. In fact, it would be impossible for the drone to remain stable at the very exact destination point, as even the slightest changes in the motor velocities would make it oscillate around that position. Moreover, it is necessary to ensure that the drone has remained still for some time, as, in the current implementation, the PID does not make the drone slow down before the target position is met, meaning that, if the target was a specified coordinate, the drone would traverse it with a certain speed rather than stopping at it. Using a tolerance also allows the drone to start the deceleration as it is `tolerance` far from the destination, allowing it to settle in the close proximity of the original target.

To implement these corrections in the algorithm, an additional `temp_counter` variable was inserted in the STM32 code, as shown in the code snippet reported in Listing 4.3. This is set to 0 at the beginning of each iteration, and is incremented if the difference from the current drone position and the target coordinate (contained in the `x_diff`, `y_diff` and `z_diff` variables) is found to be larger than `tolerance` in any direction. If at the end of these checks the variable is still 0, meaning that the drone is within `tolerance` distance from the target in any direction, the `target_counter` variable is incremented. Once the drone has remained close to the target for enough cycles (denoted by `TARGET_CYCLES`), the position is marked as reached: the `target_reached` variable is set to `true` and the status register is updated with the correct code.

Note that both the `target_counter` and the `tolerance` variables were inserted in the `PIDController` class, as they are properties of the PID that should be maintained throughout the simulation.

Listing 4.3: Implementation of the tolerance and the target reached command

```
...
if(x_diff > pid.tolerance)  {
    sideways_desired = x_coeff * 0.5;
    temp_counter++;
}
if(x_diff < -pid.tolerance) {
    sideways_desired = - x_coeff * 0.5;
    temp_counter++;
}
if(y_diff > pid.tolerance)  {
    forward_desired = - y_coeff * 0.5;
    temp_counter++;
}
if(y_diff < -pid.tolerance) {
    forward_desired = y_coeff *  0.5;
    temp_counter++;
}
if(z_diff > pid.tolerance)  {
    height_diff_desired = - z_coeff * 0.2;
    temp_counter++;
}
if(z_diff < -pid.tolerance) {
    height_diff_desired = z_coeff * 0.2;
    temp_counter++;
}

if(temp_counter == 0)    {
    pid.target_counter += 1;
    if(pid.target_counter >= TARGET_CYCLES){
        pid.target_reached = true;
        *(register_memory + STATUS_REG_OFF) = TARGET_REACHED;
        pid.num_iteration = 0;
    }
}
pid.num_iteration += 1;
if( pid.num_iteration >= UNLOCK_CYCLES*32/webots_timestep)  {
    pid.target_reached = true;
    *(register_memory + STATUS_REG_OFF) = TARGET_REACHED;
    pid.num_iteration = 0;
}
...
```

Moreover, note the few lines at the bottom. These were inserted in order to allow for the drone to unlock the simulation in the case the target position could not be reached (for example, due to the presence of an obstacle). In order to do so, the simulation keeps track of the number of iterations performed with `num_iteration` and, if these exceed a specified number of cycles (`UNLOCK_CYCLES`,

which is a configurable variable that has to be set bigger than `TARGET_CYCLES`), the position is marked as reached as well. The number of cycles needed to unlock the simulation is scaled to the Webots timestep, in order to avoid any possible misbehavior with smaller time granularities.

As a final remark, it should be evidenced that the PID is able to adapt to different timesteps and different delays between the iterations, in order not to generate problems, for example, when the asynchronous delay of some sensors have advanced the simulation before the next PID iteration. To do so, it is simply necessary to evaluate the time difference from the last iteration by saving the previous simulation time in `past_time` and by obtaining at the beginning of the iteration the current simulation time from Webots (achieved by means of the `get_sensor_data` function).

4.2.3 Handling the simulation from the `main`

The code that serves as a wrapper for all the systems and actually makes the simulation start is the `main.cpp` file. This contains the `sc_main` function: while a normal C++ uses `int main` as its entry point, SystemC programs use instead the `int sc_main` method. This is because the SystemC library has the `main` function already defined in its kernel, which internally calls `sc_main` passing (if necessary) the needed command-line parameters.

The changes that were needed to be introduced in the `sc_main` are related to the connection of the custom ports that were created for the communication between the STM32 and the motors components, and the ones that enable the connection between the functional and power modules of the motors. Furthermore, a few additional lines were inserted in order to make the method able to open the connection with Webots. To do so, it was simply needed to create a new object of type `ConnectionConfig` (whose class has been defined in the VirtualConnector library as seen in section 4.1.1) and call the `initialize_connection` method. Once the connection is successfully established, all the parameters related to the socket are passed to the individual sensors by using their `set_connection_parameters` methods.

Next, the `open_files` method of the motors component is called. As better described later, this allows the module to keep track of the measured consumption for the selected power models in several files. Then, the `sc_start(SIM_LEN , SIM_RESOLUTION)` command is executed to start the simulation with a predefined resolution (`SIM_RESOLUTION`) for a maximum amount of time (`SIM_LEN`). As the simulation has ended, the socket connection is closed (`cc.close_connection`), as well as the motors and trace files.

4.2.4 General improvements

This section reports a set of meaningful modifications that were introduced to either tailor the simulation environment for the selected scenario or to improve the user interaction with MESSY.

Time accuracy

A particular attention was given in order to ensure that the SystemC core time and the Webots time were aligned during the simulation. This problem might look trivial, but there are multiple ways in which the two may diverge from each other. In this subsection, the several developed mechanisms that ensure the maximum synchronism possible are presented, along with a final remark on the impossibility of having a perfect match between the two.

The most important modification that was introduced to solve this issue was the inclusion of a custom `advance_simul()` method in each of the sensors, which is called instead of the standard `request_delay` function provided by MESSY. This function was created to correctly keep track of any discrepancy between the two simulation times which may arise due to the different timesteps that the Webots simulation may use. In fact, while it is possible to configure MESSY to use any time resolution without significant drawbacks, a specific timestep should be set on Webots in order to strike the best balance between accuracy and simulation time. In other words, the Webots timestep (meaning the minimum unit of time that the simulation can be advanced of) may be chosen to be more or less accurate on the basis of the simulation complexity and execution time. As a result, while MESSY can have a finer-grained resolution (in this work, the accuracy is set to 1 ms), the other program may be forced to advance simulations with a different granularity, such as 4, 8, 16 or 32 ms. This surely creates a problem when the sensors delays are not multiples of the timestep: misalignments with the time of MESSY may occur, and, for example, smaller delays may not be modeled on 16 or 32 ms simulations while would be considered for 2 and 4 ms. Hence, this function was created in order to maintain the best synchronisation possible. For the reader's convenience, a simplified version of the algorithm has been reported in Listing 4.4, and it is going to be explained in the following paragraph.

Once a certain action has to be performed on the sensor, the SystemC simulation kernel is advanced by the correct amount of time (`sensor_time`) using a `request_delay` function (which basically is a method of the core made at disposal by MESSY that allows the GVSoC program to keep executing its instructions in that interval of time in order to simulate the concurrency of the real device). Then, the closest timestep multiple is evaluated in this way.

Consider this numerical example. If the `sensor_time` to be simulated is 29 ms and the `webots_timestep` is 8 ms, the closest multiple to correctly represent such

delay in Webots is 32. In this case, `q`, meaning the multiple of the timesteps that the simulation should be advanced of, is 4, and `r`, meaning the difference between the time elapsed on MESSY (29 ms) with respect to the one of Webots (32 ms) is -3 ms. In the first step (lines 11-12), the `q` and `r` parameters are evaluated as the simple quotient and remainder of the division between the sensor delay and the Webots timestep. Then, a corrective procedure (lines 14 to 17) is performed: if the reminder is greater than half of the timestep (in other words, the following multiple is distant less that `webots_timestep`/2), the next multiple should be considered (`q = q+1`) and the reminder should be updated accordingly.

This approach allows to work with any couple of sensor delay and timestep, making it suitable to work also in the presence of a timestep equal to 1 ms (requiring `r` to be strictly greater than half of the timestep rather than greater or equal).

As an additional note, be aware that a check is done in order to ensure that the Webots time step is non-zero. If that is not the case, that means that the simulation has already ended or not started at all, meaning that the time offset should not be sent to Webots but should be compensated at the final calculation of the Webots time.

Listing 4.4: Example of Webots timestep control

```
1     ...
2     core->request_delay(current_time, sensor_time, SIM_RESOLUTION);
3     ...
4
5     if (webots_timestep == 0) {
6         sensors_offset = sensors_offset + sensor_time;
7         cout << "updated offset (camera)" << endl;
8         return;
9     }
10
11    // Advance Webots by the closest multiple of the timestep
12    int q = sensor_time / webots_timestep; // q = quotient
13    int r = sensor_time % webots_timestep; // r = remainder
14
15    if (r > (webots_timestep / 2)) {
16        q = q + 1;
17        r = r - webots_timestep;
18    }
19
20    request["command"] = "ADVANCE_TIME";
21    request["time"] = q * webots_timestep;
22    sensors_offset = sensors_offset + r;
23    VirtualConnector::write_on_channel(connection->get_server_FD(),
    request);
```

In addition to this, the algorithm also serves as an example to show how the communication with Webots occurs: the sensors use the `read_from_channel` and

`write_from_channel` methods explained earlier in order to send a JSON packet with a certain request in the `command` field and then wait (blocking wait) for the answer.

This mechanism allows to correctly compensate for the offset due to the sensors delays, which are accumulated in the global `sensor_offset` variable. However, there is still an important error source that should be considered. When performing any `request_delay` action with a specified t time, MESSY actually does not advance the simulation by t, but rather gives the control to the core in order to keep executing the actions present in its queue. Each action is associated to a certain execution time related to the one of the real chip, and it is allowed to dispatch those until time t has been reached. However, as a result, the execution of these operations usually goes over t by some μs, generating very small time offsets that become non negligible as they accumulate. To solve this issue, a simple modification can be inserted per each call of the `request_delay` function, as shown in Listing 4.5.

Listing 4.5: Adjusting possible time offsets deriving from `request_delay`

```
1  double time1 = sc_time_stamp().to_double()/1000000000;
2  core->request_delay(current_time,time,SIM_RESOLUTION);
3  double time2 = sc_time_stamp().to_double()/1000000000;
4  sim_off_accum = sim_off_accum + time2 - time1 - time;
```

In this approach, the current simulation time is saved before (`time1`) and after (`time2`) the command that is issued to advance the simulation. Then, a simple difference is evaluated between the difference of the old time minus the new one (`time2- time1`) and the time that was meant to elapse (`time`). Any residual is saved in the `sim_off_accum` global variable, which accumulates this kind of time errors during the course of the whole simulation.

These two mechanisms allow the two simulations to be almost perfectly aligned. Unfortunately, there is still going to be a difference due to some delays originating in GVSoC. While executing its instructions, in fact, a small time error is generated, and this is not captured with ps precision on the MESSY side. Despite the use of several strategies to investigate the origin of the problem, it was impossible to point out its exact cause. As a simple observation, it should be evidenced that this delay seems proportional with the number of instructions executed by the GAP8 program. Nonetheless, these appear to impact the simulation in a very minor way ($< 0.1\%$), as shown by the measurements reported in Section 5.

Configuration files and debug mode

Along with the `config.hpp` configuration file provided by MESSY, a new one was created to allow the user to easily customize the simulation. This file, named `simul_configs.hpp`, can be used to set the power model to be considered, the

debugging level, the battery to be used and a few variables, namely the `tolerance` of the PID (expressed in m), the number of cycles of the PID to consider the position reached (`TARGET_CYCLES`) and the one needed unlock the simulation if the target was not met yet (`UNLOCK_CYCLES`).

Among the configurations provided by `simul_configs.hpp`, lies the setting of the multi-level debugging. In general, the code was developed in order to include meaningful messages that could be used during the simulation to check the correct functioning of the virtual simulation platform. The presence of several levels allows the user to choose which type of information should be included:

- `DEBUG_LEVEL_NONE` is the lowest level, providing only the core information;
- `DEBUG_LEVEL_LOW` includes few details, such as the received image width/height and the operations detected on the bus;
- `DEBUG_LEVEL_MED` exposes a big number of information, including the Webots requests and the sensor data during PID execution;
- `DEBUG_LEVEL_HIGH` provides instead the user with all the possible information, including all the packets exchanged through the socket.

Of course, the inclusion of the debug messages has a negative impact on the simulation performance, and it is hence advised to be used only during development.

Adapting the simulation to the use-case scenario

Finally, it is important to evidence one last modification that was introduced in order to prepare the MESSY-Webots virtual platform for the Crazyflie racing environment scenario. In particular, it was said that this setting would have involved the drone to perform, every 33 ms, a sequence of actions: getting a photo from the camera, running an inference, passing the target coordinates to the STM32 chip and translating these into motor commands. This, however, creates a problem with certain Webots timesteps. Consider this numerical example: the camera takes 7 ms to capture an image, the inference lasts 12 ms, the translation into PWM commands takes the remaining 14 ms, and the Webots simulation timestep is set to 32 ms. If the delays of each step were modeled individually, the simulation would never be able to advance: each of them is lower than `webots_timestep`/2, meaning that the closest multiple of the timestep is 0 ms. In order to solve this issue, the delays generated by the various sensors in a single iteration were accumulated in a specific variable, named `iter_delay`.

Basically, at the beginning of each iteration, the camera module captures the image and sets `iter_delay` to 16 ms (i.e., the delay associated with the image capturing process). Then, GVSoC triggers the ITER ADVANCE function on the

STM32 to simulate the time taken to run the inference on the network. Such time is extracted and added to the `iter_delay` (in the considered scenario, this is equal to 10 ms). Finally, 7 ms are added to account for the time that passes between the translation of the GVSoC coordinates to the PWM commands reaching the motors after having executed the `PID_iteration` function. Note that these delays were inserted to be as realistic as possible, but do not match any measured value. For the camera, its delay value was chosen knowing the fact that is able to run at a maximum of 60 FPS, while a measurement performed by GreenWaves Technology on some CNN network [2] was used as the reference for the inference time (note in fact that the ISS was not meant to simulate a real inference due to lack of a proper trained network that calculates the target gate position on the basis of a Webots image). The remaining ms were used to model the other operations, which in the real drone would depend on the firmware of the STM32 chip and on the switching time of the gates connecting its GPIO (General Purpose Input/Ouput) pins to the motors.

Finally, note that both at the beginning and at the end of the simulation the `iter_delay` variable is set to be equal to the timestep: this choice was done in order to prevent the rise of possible errors, and to maintain the maximum synchronism between MESSY and Webots.

4.3 GVSoC program

As said multiple times, the system is mainly controlled by the program running on GVSoC, the GAP8 emulator. The code, written in C, is compiled with the libraries already present in MESSY and used as the main program during the simulation. A brief description of the code is reported below, and a schematic recap of the main functionalities is reported in Algorithm 4 (due to the complexity of the program, only the three main methods have been reported). Moreover, the original versions of those methods have also been reported in their entirety in Appendix D.

The program starts with the definition of a series of macros. In particular, these are used to define the basis address of the peripherals, the offset to be applied to reach the various registers of the camera and STM32 modules, as well as the commands reported earlier in Table 4.3. The code continues with the definition of the function prototypes, and the initialization of the pointers that are going to be needed to access the individual registers and other variables needed for the simulation.

[2] `https://greenwaves-technologies.com/gap8-versus-arm-m7-embedded-cnns/`

Algorithm 4 GVSoC program pseudo code

```
Initialize macros for the basis address of the peripherals, the register offsets, and commands
Function prototypes
Initialize registers pointers
procedure MAIN
    if DEBUG_MODE enabled then
        Check camera and STM32 registers
    end if
    Call init_pid and get_stable
    if CORNERS_TEST enabled then
        Set and reach targets at defined corners
    else
        for i = 1 to NUM_OF_CIRCUIT_LOOPS do
            Set and reach various simulation targets
        end for
    end if
    Call go_down and print completion message
    Return errors
end procedure
procedure SET_TARGET(x, y, z)
    if low_battery is false then
        Set new target coordinates for STM32
        while target not reached and battery not low do
            Call get_image, advance_simul(10), and run_single_pid_iter
        end while
        Reset control and status registers
    else
        Skip target due to low battery
    end if
end procedure
procedure RUN_SINGLE_PID_ITER
    Send command to STM32 to run a single PID step
    while STM32 status does not indicate completion do
        if battery is low then
            Set low_battery flag and return
        else if target is reached then
            Set target_reached flag and return
        end if
    end while
end procedure
```

Listing 4.6: Scheme of general operation execution for GVSoC code

```
1    *(module_control_reg) = TARGET_OP;
2    while(*(module_status_reg) != TARGET_OP);
3    *(module_status_reg) = 0;
4    *(module_control_reg) = 0;
```

Pointers are used to reference the control and status registers, and each operation works as shown in Listing 4.6: the opcode of the wanted operations is written on the control register, then the status register is polled until the same code is retrieved. Once so, both registers are reset back to 0 to clear them and prevent possible errors. This approach is used for several of the methods that are called by the program. Namely, those are:

- `get_image`, used to force the camera to capture an image;
- `init_pid`, used to call the `PID_init` function of the STM32 sensor;
- `get_stable`, used to call the `get_stable` function of the STM32 sensor;
- `advance_simul`, used to simulate the presence of an inference and advance the simulation by a specific amount of time;
- `go_down`, used to trigger the actions to make the drone descend to ground at the end of the simulation;
- `run_pid`, used to call the `PID_run` function of the STM32 sensor;
- `print_simul_time`, used at the end of the program to print the current simulation time (in this case, no reset of the control/status register is performed, as the simulation is almost finished).

A couple of other methods have been inserted to complete the program functionalities: those are `set_target` and `run_single_pid_iter`.

The first method is the one that has to be used inside the main function to set and reach a new destination. This method receives as inputs the space coordinate of the destination and, after writing them inside the proper registers of the STM32, informs the SystemC module that new target coordinates have been provided. In turn, this updates the local variables of the `pid` object, and sets the status register to inform of the task completion. Once that condition is sensed by `set_target`, the function enters a `while` loop that performs the three main actions of the drone to be performed sequentially throughout the simulation: gets an image from the camera (`get_image`), simulates the inference of the image (`advance_simul`) and finally runs the PID (`run_single_pid_iter`). The `while` loop terminates in two conditions: either the position was effectively reached (with the local `target_reached` variable being set to 1), or the battery level was found to be too

low to continue (`low_battery` = 1). When so happens, it resets `target_reached` as well as the STM32 control and status registers to prepare for the next actions. If the battery was found to be too low when entering `set_target`, the function simply skips its action in order for the simulation to terminate as soon as possible.

As said, by using `run_single_pid_iter`, GVSoC is able to run an iteration of the PID controller. This operation is performed in a way similar to the other methods: the specific operation code (RUN SINGLE STEP PID) is written on the control register, and the status register is polled until the completion of the task on the SystemC side is signaled by the same code. However, the status register content is checked also against two other codes: LOW BATTERY (which triggers the update with a 1 of `low_battery` and breaks from the loop) and TARGET REACHED (which, along with signalling the completion of the single PID iteration, also makes GVSoC aware that the target was reached by updating the `target_reached` to 1). With these two functions, GVSoC is enabled to be fully aware of the current state of the simulation, and can plan its reaction accordingly.

Having discussed all the possible operations, it is now fundamental to explain how these are used by the `main` function to obtain a working simulation. The order of operations is the following. After a few messages are printed on the terminal for the user to know that the simulation has started, the program issues a **`init_pid`** command to initialize the system. Follows **`get_stable`**, which makes the Crazyflie lift and reach a stable position. Now, the drone is ready to traverse the 10 gates: with a sequence of 10 **`set_target`** functions, the program gives to the device the coordinates of the points that it must reach in order to pass through all the targets. This process is repeated a number of times, selectable by the user with the `NUM_OF_CIRCUIT_LOOPS` macro. Finally, the drone is brought to ground with **`go_down`**, which also shuts down the drone motors and stops the Webots simulation. Before concluding, GVSoC issues a **`print_simul_time`** to make the STM32 sensor print the current simulation time.

Note that the code also supports a debug mode. If enabled (i.e., if a `DEBUG_MODE` macro is defined), it shows at the very beginning of the program the values of the two peripherals registers, by calling specific functions (`check_camera_registers` and `check_stm32_registers`) that access them and print on terminal their content. Moreover, the presence of the `CORNERS_TEST` macro triggers a different simulation type: instead of making the drone traversing the gates, it pilots the device to the four edges of the map as many times as the value of the macro. This function was included to provide a "debugging" simulation type, that can be used to simulate a much more linear (steady) flight, which helps focusing on simpler scenarios (very useful when there is the need of testing new control algorithms and the effectiveness of several batteries).

4.4 The Webots controller

Before moving to the modelling of the power consumption, the Webots controller should be described in order to complete the overview of the functional aspect of the simulation. The code, which again is used to answer the requests of GVSoC and to handle the 3D environment, is rather complex, and for this reason a pseudo code version has been reported in Algorithm 5. Note that just a couple of the original methods have been reported, and the reader can consult the original version in Appendix E.

The controller, developed in C++, was structured as follows. First, a set of `#include` commands is used to include all the libraries, along with several `#define`s to create some useful macros. Among these, lies `SIM_ACCURACY`, used to configure the timestep: depending on this value, the global variable `simul_timestep` is updated with the number of *ms* corresponding to the smallest unit of time of the simulation. Note that this value should match the one of `basicTimeStep`, an attribute of the `WorldInfo` node that should be set manually through the GUI. After the function prototypes definitions, lies the `main` function. This is the entry point of the simulation, and it performs the following actions. First, it initializes the robot and creates the socket connection. Being the server, the actions to be performed are the ones seen in Section 2.3. Those have been reported in the code snippet of Listing 4.7: the socket is created, its address is filled with the right information, it gets binned to a path, then it listens to any process that wants to communicate with it and, as a client is detected, it accepts the connection. Each operation is associated with an error management strategy, that makes the `main` function return a -1 in presence of any error.

Listing 4.7: Server side socket creation and connection establishment

```
1  server_socket = socket(AF_UNIX, SOCK_STREAM, 0);
2  if (server_socket == -1) {...}
3
4  memset(&address, 0, sizeof(address));
5  address.sun_family = AF_UNIX;
6  strcpy(address.sun_path, socket_path.data());
7  unlink(socket_path.data());
8
9  if (bind(server_socket, (struct sockaddr*) &address, sizeof(
   address)) == -1) {...}
10
11 if (listen(server_socket, 1) == -1) {... }
12
13 client_socket = accept(server_socket, NULL, NULL);
14 if (client_socket == -1){...}
```

If the connection was successfully established, the motors are initialised by

Algorithm 5 Webots controller pseudo code

```
Libraries inclusion, macro definitions and function prototypes
function MAIN
    Initialize Webots environment, create an instance of the robot

    Create the socket connection
    Connect the client

    Initialize all the motors and the sensor
    Enable IMU, GPS, gyro, camera, and distance sensors

    Wait for 2 seconds
    while stop is not active do
        Run sampleRun
    end while
    Close the connection
    Clean up the resources
end function

function sampleRun
    Receive command from the MESSY client in JSON format
    if command == "GET_INIT_DATA" then
        Send initial data
    else if command == "GET_IMAGE" then
        Send camera image
    else if command == "GET_SENSORS" then
        Send sensor data
    else if command == "SET_VELOCITY" then
        Set motor velocities
    else if command == "GET_DATA_2" then
        Send data for model 2
    else if command == "GET_DATA_3" then
        Send data for model 3
    else if command == "STOP" then
        Stop the simulation and send elapsed time
    else if command == "ADVANCE_TIME" then
        Advance the simulation
    end if
end function
Other methods definition
...
```

setting the internal Webots PID associated to each of them to have a certain initial velocity and not to have any initial target position. Then, the sensors are initialised and set to operate at the same timestep of the simulation. At this point, 2 seconds are waited, in order for the sensors to align and for the Webots PIDs to make the motors reach the initial velocity. Now, the program can effectively start. The run of all the functions is handled by the `sampleRun` function, which is repetitively called until the simulation has stopped (signalled by the global `stop` variable being found equal to 1). When so happens, the controller closes the connection, deletes the robot and terminates the simulation.

It should be evident how `sampleRun` plays a central role in the simulation. This function, which receives as inputs the pointers to every component of the robot, basically parses the JSON packet received from MESSY and performs a sequence of actions on the basis of the content of the `command` field, calling other methods if necessary. The function either returns (ready for the next iteration) or sends back to Webots some data. Follows a high-level description of the eight possible commands and how these are handled by the function.

If a GET_ command (either a **`GET_INIT_DATA`**, a **`GET_SENSORS`**, a **`GET_DATA_2`** or a **`GET_DATA_3`**) has been received, Webots answers back to MESSY with a packet containing the required sensor data, putting each piece of information in a different field. As an example, it returns the GPS coordinates, the time and the timestep at the receival of a `GET_INIT_DATA` and the horizontal plane velocity in the case of `GET_DATA_2`. While the first command is requested by the `PID_init` function to get the initial data (and is handled by the `getInitData` Webots method), `GET_SENSORS` is received at each step of the PID to provide the necessary data for the current iteration to occur. In this case, the operations are handled by `getSensors`. The other two `GET_DATA` commands are instead sent from the MESSY motors, which have the need of gathering data for the energy consumption evaluation performed by two distinct power models. On the Webots side, the packet containing the necessary information is created by means of `send_data_model_2` and `send_data_model_3`.

If a **`GET_IMAGE`** is received, Webots should return to MESSY the image currently visualised by the camera sensor. This is handled by the `getCameraImage` method, described in the following. In order to work with the camera, however, this has first to be characterized in a way that is fully compliant with the real device specifications. As it is not possible to force the camera to be monochrome, the only modification needed for the `camera` node was the setting of the image width and height (320x320). The conversion into a monochrome image was instead performed inside the function by using the following approach. First, the raw image is retrieved from the camera using the `camera->getImage` function. A gray-scale transformation is then applied to each pixel of the image: this is done by iterating through every pixel in the image using the width and height information and applying the function `camera->imageGetGray` to extract the gray scale value of

each single pixel. The value, which represents the average color intensity of the pixel, is stored in a dynamically allocated memory buffer (`imageData`), which can then be sent on the socket by means of a JSON packet.

In the case of a **`SET_VELOCITY`** command, Webots extracts the velocities to be applied to the four rotors and calls the `setMotors` function to apply each of them to the corresponding actuator present on the simulated version of the drone. As it does so, the simulation is advanced by one timestep: in case multiple timesteps are needed to simulate the 33 ms loop, the command is simply sent another q-1 times by the MESSY motors.

When the command is sensed to be **`ADVANCE_TIME`**, the controller extracts the amount of time that should elapse, and advances the simulation accordingly.

Finally, the **`STOP`** command (called by the `end_simulation` method of the STM32 sensor) instructs Webots to stop the drone motors and end the simulation. To do so, the controller calls the `stop_simul` method, which basically waits for the drone to reach an effective speed of 0 and reports the elapsed time back to MESSY to keep the time alignment. The reader should in fact be reminded that the model of the motors present in Webots has an internal PID, completely independent from the one of the STM32. When setting the speed to 0, the velocity drop is not instantaneous, but rather decreases in time according to the internal PID parameters. As the motors are stopped, the `stop` variable is set to 1 in order to break from the `while` loop that encapsulates `sampleRun`.

Note that the communication with Webots happens through JSON packets, which are received and sent with the very same functions (called `receiveMessage` and `sendMessage`, respectively) contained in the VirtualConnector library on the MESSY side.

As a final addition, a debug mode was also inserted on the Webots side. In this case, this enables the logging of each of the exchanged packets, plus the print of some additional messages related to the simulation. The debug mode can be enabled by simply defining a `DEBUG_MODE` macro.

4.5 Power modelling

This section describes the modelling of the most significant extra-functional property: power consumption. Due to the openness of the Crazyflie architecture, most of the information needed to deploy such model have already been made at disposal by the BitCraze company itself, while some approximations are going to be introduced to deal with the aspects not covered by the drone components datasheets.

In particular, the main contributors to the energy consumption of the device are the core (whose contribution is automatically taken into account on MESSY), the camera, the STM32 chip, the nRF chip and, most importantly, the motors. Other

sources, such as the ones of the Bluetooth/WiFi antennas (supposed to be shut off) and the ones of the individual chips (like GPS, IMU, etc., for which it was impossible to gather detailed consumption information) were not considered. In addition to this, it should also be taken into account the fact that the transmission of the power is not ideal, hence MESSY simulates the efficiencies of the various DC-DC converters, as well as the internal resistance of the battery, to provide the best accuracy possible.

The standard configuration of MESSY to evaluate power consumption is the following. Each component (namely, the core and the sensors) is connected to a power bus, which has a fixed associated voltage and which periodically (by default, every 1 ms) collects all the individual contributions to create the total current drain. This current request is then forwarded to the power sources. MESSY supports two types of energy harvesters: batteries (which are more complex and are defined by means of SystemC-AMS) and sources (defined in a much simpler way, and which represent elements that can provide a current that dispenses the system to take it from the batteries). In this work, only one battery was considered to model the real drone setting.

While the core has its own set of states and associated power consumption, each sensor status is generally associated to a current consumption, specified in the JSON configuration file. In particular, MESSY identifies 3 statuses for any sensor (reading, writing and idle, which are the fundamental ones), but the user may specify his own set of status-consumption pairs, which comes especially useful when modelling sensors with complex behaviors.

For both the core and the sensors, the connection to the power bus is obtained by means of a DC-DC converter. In essence, each of the system components has an additional module, named as `module_x_converter`, which connects to the power instance of the device to produce at the output the effective current consumed by that module on the basis of a customizable efficiency value. By default, the converter gets from the power instance the voltage specified in the JSON file, as well as the current consumed by the current sensor state. Moreover, the simulation automatically sets the efficiency to 1 in the `sc_main` by using the `set_efficiency` methods included in the modules themselves.

Also the battery has an inbuilt DC-DC converter, `battery_converter`, that simulates the efficiency of the battery with respect to the drained current exploiting a LUT (Look Up Table) based approach: it uses some of the known relations between the battery voltage (or current) and efficiency of the real device to accurately evaluate the inefficiencies.

Before going over the single components, however, it is important to have a look at the architecture of the real drone in order to understand which modifications are needed to be introduced in the MESSY automatic configuration to suit the Crazyflie. In particular, some useful pieces of information may be taken from the

official website [57] and the schematics provided the company [58], which help to have an idea of the other sources of power inefficiencies and dissipation.

Speaking of the drone schematics, these reveal a couple of very important details. The first one regards the relation of a motor with the battery, whose schematics has been reported in Figure 4.5. As evidenced by the image, rather than being

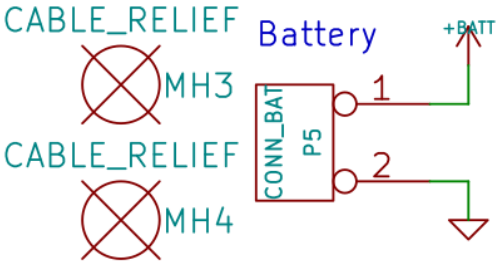


Figure 4.3: Battery connector scheme.

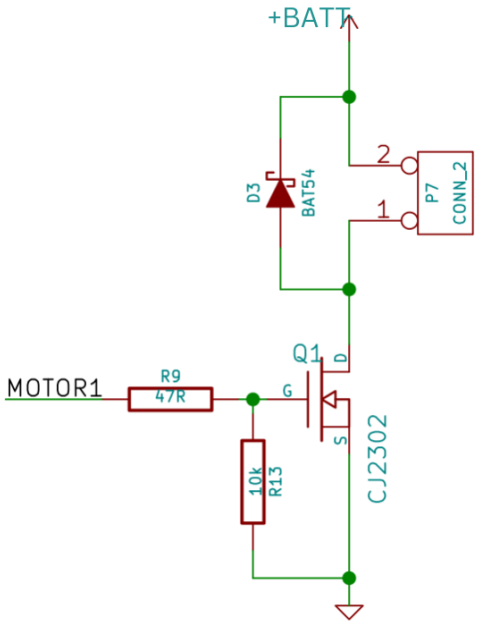


Figure 4.4: Motor schematics.

Figure 4.5: Overview of the connection between battery and motors.

connected to VCC (which is a regulated power reference also used by other system components), the motors are exposed to the raw battery voltage. In fact, excluding the presence of the Schottky diode (which protects the circuit from the voltage spikes generated by the inductive load of the motor when it turns off) and the low-side switch N-MOS (which enables/stops the current to flow through the motor), the motors are directly inserted across the positive battery tension and the (virtual) ground of the Drain (D) terminal of the transistor. This means that a second power bus needs to be introduced in MESSY, to which the sole motors are attached, and whose voltage value dynamically varies with the one of the battery.

Having a look at Figure 4.8, instead, it is possible to see that the STM32 chip reference voltage is VCC and how this is obtained from the battery voltage +BATT.

The generation of VCC (which is a stable 3.0 V that feeds the input of the STM32 chip and the camera) and the other references voltages are obtained by

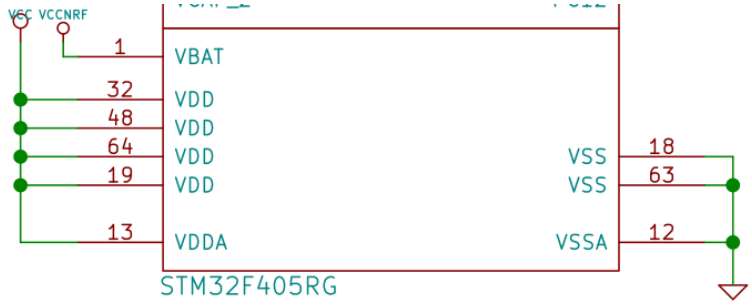


Figure 4.6: STM32 pins.

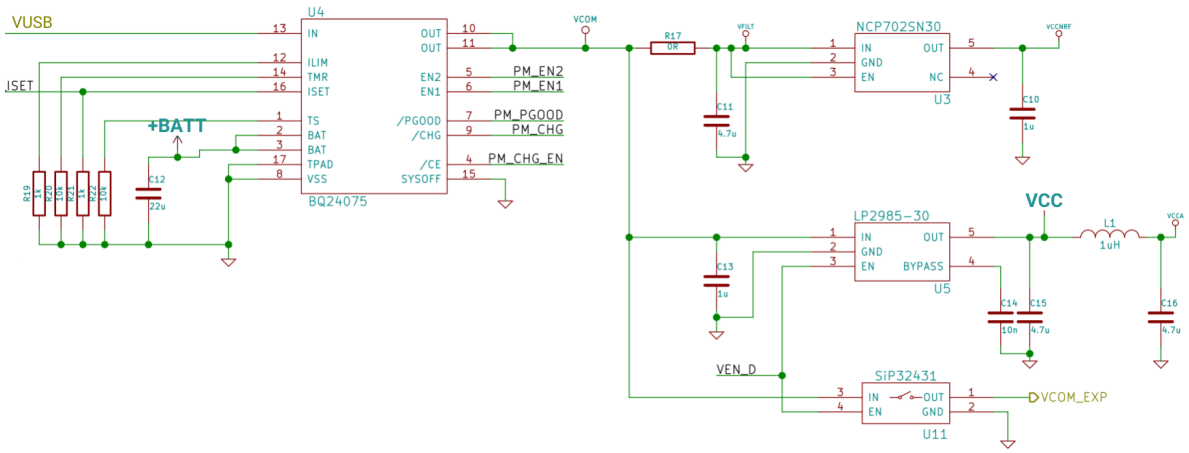


Figure 4.7: VCC reference circuit.

Figure 4.8: Overview of the use of VCC and its generation.

means of a couple of devices: a BQ24075 battery charger and a LP2985-30 Low DropOut voltage (LDO) regulator. The first is the chip that connects the battery to the charger port to allow for the correct charging operation when an USB (Universal Serial Bus) power source (in the range of 4.5 - 5.5 V) is connected. Moreover, the chip is used to track the battery status and to protect the circuitry from overcurrents or other malfunctions. According to its datasheet[3], this component is able to produce a maximum output current of 5 A with a dropout voltage of 0.1 V with respect to the one of the battery when the charger is disconnected. In doing so, a very small quiescent current is consumed by the device (max: 1.1 mA).

The second is instead a LDO regulator, which has the role of stabilizing its

[3] https://www.ti.com/lit/ds/symlink/bq24073.pdf?ts=1727457139132

output voltage at a fixed 3.0 V independently on the input one (which, in the Crazyflie, is the one output by the battery charger). Accordingly to the datasheet[4], this component is able to stabilize the input up to the point at which this has reached a value between 3.15 and 3.10 V (being 0.12/0.145 V the typical and maximum dropout voltage for an output current of 50 mA). Apart from that, the DC-DC conversion produces a very low quiescent current, which sits at about 750 μA in the maximum output current condition.

Considering these aspects, it is possible to evaluate the efficiency of the power transmission as follows. The efficiency η is the ratio between the output and the input power, which is in turn the product of the input and output currents and voltages. Since I_{in} and I_{out} differ from each other by just the leakage current I_{leak} (which is negligible for both the charger and the LDO), the two terms can be cancelled out, and the efficiency can be simply modeled as the ratio between the output and input voltage. To ease the reading, this process has been resumed in Equation 4.1.

$$\eta = \frac{P_{out}}{P_{in}} = \frac{V_{out} \cdot I_{out}}{V_{in} \cdot I_{in}} = \frac{V_{out} \cdot I_{out}}{V_{in} \cdot (I_{out} + I_{leakage})} = \frac{V_{out}}{V_{in}} \tag{4.1}$$

The insertion of this inefficiency in the power transmission from the battery to the STM32 and the camera is the second modification to be introduced in order to create a sufficiently accurate modelling of the drone.

Note that, for a true, complete estimate of the power consumption, all the quiescent currents of the passive components should also be included, as well as the inefficiencies of the few other chips (such as the RF frontend and the IMU sensors). However, being these contributions virtually insignificant with respect to the others, they were excluded from the model.

4.5.1 Sensors

The power model of the three sensors (the STM32, the camera and the motors) mainly involved the customization of their power modules and converters.

All power instances have three default methods: `set_attributes`, `initialize` and `processing`. These are part of the standard methods used for TDF computations. In all cases, `set_attributes` is used to define the timestep and the delay of the ports, while the specific behavior of the module is specified in `processing`. About the latter, this method simply senses the value at the `func_signal` port to understand the sensor status, and communicates both the associated current and voltage to the converter. In turn, this module (which is also composed of the very

[4] `https://www.ti.com/lit/ds/symlink/lp2985a.pdf?ts=1720429702974`

same three methods) senses the current, scales it according to the wanted efficiency (defaulted to 1) and returns it to the power bus.

Camera

To adjust the camera consumption for the reading (which corresponds to the capture of an image), writing and idle states, the datasheet was used. The camera contribution was set at 1.75 mA, 1.75 mA, 0.14 mA for the three cases, respectively. The first is equal to the current actively consumed by the device when operating QVGA quality at 60 FPS, while the third one was set to be equal to the typical standby current in the worst operating conditions. The second value was simply copied from the first one, as the writing condition is never used inside this simulation (as noted in previous chapters).

STM32

Unfortunately, the consumption of the STM32 chip would strongly depend on the firmware. This would of course change depending on the actual operations to be executed (running a single PID step would require a different energy with respect to the setting of the targets, for example). For this reason, the current for the reading, writing and idle status was set respectively to 150 mA, 150 mA, 60 mA: the first two values represent the 102 mA consumed by the STM32 in run mode at 168 MHz (the maximum frequency) plus the contributions of the IMU sensor, while the last was chosen according to the maximum typical current consumed in standby at 168 MHz. These values were obtained from the component datasheet [5] as per the camera. Note that these values were meant also to include an estimate of the power consumption of the nRF chip. Since a detailed modelling of this chip was out of the scope of this thesis, its consumption (the chip is effectively active most of the time to manage the power delivery) was aggregated to the one of the STM32.

Motors

Being strongly related to the power aspect, no information was yet given about the motors. Hence, an overview of both its functional and extra functional modules is presented in the next paragraph.

Starting with the functional side, the motors expose the following functions, grouped accordingly to their similarities:

[5] `https://www.st.com/resource/en/datasheet/stm32f415rg.pdf`

- `sensor_logic`: this senses the presence of a new set of commands for the motors on the `send_command` port and calls `send_data` and `consumed_power` to send them to Webots and evaluate the current power consumption. Once so, it informs the motors that the send has been completed by raising `send_completed` for a single delta cycle;

- `open_files` and `close_files`: these functions handle the opening and the closure of three files, which are used by the motors to keep track of the instantaneous power consumption of the three chosen power models. As seen, both are called inside the `main`;

- `send_data`: this method prepares the packet containing the `SET_VELOCITY` command and writes it on the socket. As seen, this command is repeated q times, as Webots advances the simulation one timestep at a time. Note that the simulation is advanced by `iter_delay`-1 ms to compensate for the 1 ms required (by one of the next methods) to send the current value to the motor power instance. Furthermore, several mechanisms described in Section 4.2.4 are present to keep track of the offsets due to `iter_delay` and `request_delay`;

- `consumed_power`: as better explained in 4.5.3, this method calculates the current power consumption according to the three chosen models;

- `send_to_power_model`: it is the function that communicates the drawn current to the power instance. This is done by a simple protocol, which writes the value on the `current_value` port and raises the `send_current` to high to signal the presence of new data. A delay of 1 ms was inserted before the down raise of that signal, as that is the minimum sampling resolution of the power instance;

- `get_data_model_1`, `get_data_model_2` and `get_data_model_3`: these methods are called by `consumed_power` to obtain from Webots the necessary data for all the power models.

On the power side, motors maintain a structure similar to the ones of the other sensors: the sampling resolution of the ports is set by the `set_attributes` function while `processing` writes the current voltage and current values to the motors converter (which shares the same properties as the others). The only changes with respect to the other two modules are represented by the presence of the two ports that enable the direct communication with the motor functional instance.

4.5.2 Power buses

As said, two power buses should be modeled in the system: the standard VCC one, which provides power to the STM32 chip and the camera, and the bus of the battery voltage, used to feed the motors.

The two were organized in the following way. The new power bus (`Motor_power_bus`) was created on the basis of the standard one. It shared the very same structure as the other, but without the core contribution: 2 input ports are present to obtain the voltage and current values from the motor converter, and the `current_batteries` output port informs the battery converter of the motor current draw.

Conversely, the VCC power bus was modified from the automatically generated one to exclude the signals related to the motors. As a result, the standard power bus takes the power information from the core, the camera and the STM32 sending the sum of their current draws to the battery converter.

The latter, which is the module supposed to inform the battery about the total contribution, collects current from both buses by using an efficiency parameter of 1 for the motor current and the $\frac{3.0V}{V_{sensors}}$ efficiency derived in Section 4.1 for the ones coming from the other sensors.

Note that the voltage information received by the VCC power bus comes from the very same signal used by the power instances of the modules to inform their sensor converters of the current voltage.

4.5.3 Energy consumption models

Among all the contributions, the energy drawn by the motors is surely the most relevant. To evaluate it, several models were previously proposed in Section 3.4, but only three of those could actually be implemented in the considered scenario, either due to the lack of simulation/drone data or due to their complexity. Those models are the empirical one, based on the current measurements performed on the real drone, the D'Andrea one and the one from Stolaroff. About the latter, there was no need to include the correction for high velocities, as that aspect is not part of the considered scenario. Those models were included inside the functional instance of the motors, specifically in the `consumed_power` method, in order to be called as soon as a new motor velocity is received.

About the empirical model, no data is actually needed from Webots, as the consumption is evaluated by means of a quadratic relation between the current and the velocity values produced by the PID. The current is evaluated in mA, while the motor velocity is expressed in rad/s. The velocity value fed into the equation is simply obtained as the average velocity from the four motors (calculated in `get_data_model_1`). Knowing the current drain, the battery voltage was estimated by means of the linear equation mentioned earlier in Section 3.4.3.

The D'Andrea model required instead the declaration of a few variables inside the set of private attributes of the functional instance of the motors. In particular, the most important ones have been reported in Table 4.4.

Variable	Value [Unit]	Description
`mass_drone`	0.0199 [kg]	Drone mass
`mass_aideck`	0.0044 [kg]	AI deck mass
`mass_battery`	0.0071 [kg]	Battery mass
`g`	9.81 [m/s^2]	Gravity acceleration
`r`	3 [(adimensional)]	Lift-to-drag ratio
`efficiency`	0.9 [(adimensional)]	Efficiency of power transfer from battery to propellers
`P_avio`	0.350 [W]	Power due to electronics
`P_hover`	8.7 [W]	Power consumed by the motors during hover

Table 4.4: Parameters for D'Andrea model.

The mass was set considering the drone structure (19.9 g) plus the weight of the battery (7.1 g) and the AI deck (4.4 g). Following the author's suggestions, the lift-to-drag ratio was set to 3, a pessimistic estimate for vehicles that are capable of vertical takeoff and landing. The efficiency was set to 0.8 instead, in order to model any power loss due to thermal, vibration and noise, while still considering the direct connection of the batteries and the propellers (for example, D'Andrea estimates this to be about 0.5, while Stolaroff 0.7). The power taken from the electronics was set to always be 350 mW, rather than taking the one from the STM32 component and the camera. This was done for three primary reasons. First, the power modelling of both sensors is rather approximate due to the impossibility of performing direct measurements on the real devices while testing the algorithms. Secondly, opening the communication with the sensors would have implied a significant increase on the setup complexity and general orchestration. Finally, the Bitcraze forum already provides some values for the consumption of the drone while hovering [59, 60], providing a much more accurate way of estimating the real one. The same reasoning goes for the hovering power, which was again estimated on the basis of the data provided by the company itself: the 8.7 W values was obtained by having a look at the measurements on drone consumption for different thrust values (the one for 32 g was chosen). Note that this value was not mentioned in the original model: the D'Andrea equation targets drones of much larger sizes, in which consumption can be modeled to be linearly related to velocity. In this case, the power consumption introduced by hovering was added to the velocity dependent term to obtain more accurate results. From Webots, this model only needs the current drone velocity,

which is obtained by means of `get_data_model_2`.

The parameters presented in Table 4.5 were instead used to characterize the missing parameters of the Stolaroff model.

Variable Name	Value [Unit]	Description
`n`	4 [(adimensional)]	Number of rotors
`beta`	0.001589 [m^2]	Area of spinning blade
`ro`	1.225 [kg/m^3]	Air density
`A`	0.008464 [m^2]	Projected area

Table 4.5: Parameters for Stolaroff model.

The drone is a quadrotor (`n` = 4), and β was evaluated as the area of a single spinning blade ($A = \pi \cdot r^2$, where r is 22.5 mm, i.e. half of the propeller diameter[6]). For air density, the value was taken considering the sea level conditions with 0% humidity and a temperature of 15° C. The projected body area, instead, was considered to be the product between the drone width and length (92 mm by 92 mm). This ignores the presence of some empty space between the propellers, and possible modifications due to the battery installation, which were however quite complex to model without the possibility of performing measurements on the real device. Finally, note that the model also needs the horizontal velocity of the drone, obtained through the `get_data_model_3` method.

Along with the current consumption, the battery voltage estimate obtained for the first model was also used to compute the power for each model. These values were then logged in the three different files opened by the `open_files` method in order to ease the comparison process performed at the end of the simulation.

As a last addition, remind that, among the others, these models were the least complex ones. Despite that, the previous paragraphs should have evidenced how many approximations were already needed to create a realistic set of parameters, and how the introduction of further complexity would not have benefited the accuracy of the results.

4.5.4 Battery and battery converter

The Crazyflie drone mounts by default a 250 mAh battery. This single cell battery features a nominal voltage of 3.7 V and a maximum charge/discharge current of 2C/15C (0.5 A / 3.75 A) in a 682030 package (WxHxD: 20x7x30 mm). Considering the integrated Molex connector, the battery weighs about 7.1 g and allows the

[6] `https://store.bitcraze.io/collections/spare-parts/products/propeller-pack`

drone to hover for about 7 minutes[7].

In order to model the battery in MESSY, its discharge curves are needed, in particular the ones revealing the relation between the capacity and the voltage. Unfortunately, Bitcraze does not declare the manufacturer of this component, making it impossible to know the performance of the real one. To compensate for this problem, however, batteries with similar specifications could be used. Among these, the LPHD6520030 battery [61] from *LiPol Battery* seems to match in its entirety the original model: it is a 250 mAH battery, capable of a 15C discharge, with same nominal voltage and package dimensions. The curve of such battery, made available by the producer, has been reported in Figure 4.9.

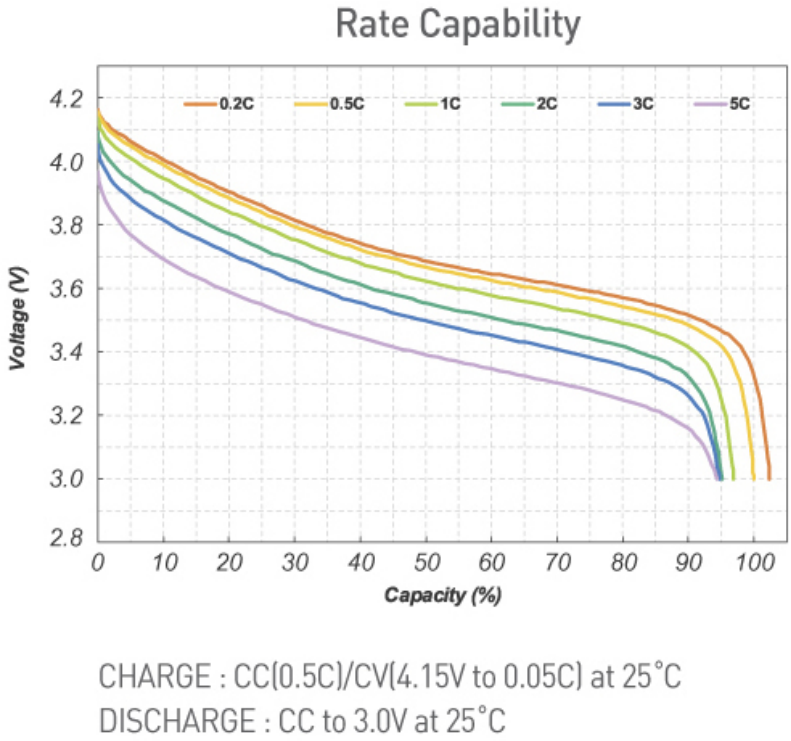


Figure 4.9: LPHD6520030 charge-voltage discharge graph [61].

To use these curves, some samples were taken from different current rates and then interpolated by means of a fourth order polynomial. To obtain the samples, the PlotDigitizer app [62] was used. Then, these were fed into the Digipyze program [63] to obtain the coefficients. The resulting equations made it possible to model the relations between the SOC (State Of Charge) with the battery open circuit

[7] `https://www.bitcraze.io/support/f-a-q/`

voltage v_{oc} (useful to estimate the remaining battery charge) and internal resistance r_s (useful to model the ohmic voltage drop when a current flows). These have been reported in Equations 4.2 and 4.3.

$$\begin{aligned} v_{oc} = 1.39609 \cdot SOC^4 - 2.95149 \cdot SOC^3 + \\ 1.60142 \cdot SOC^2 - 0.10038 \cdot SOC + 4.15517 \end{aligned} \tag{4.2}$$

$$\begin{aligned} r_s = 0.0027310 \cdot SOC^4 - 0.0058076 \cdot SOC^3 + \\ 0.0031560 \cdot SOC^2 + 0.00005570 \cdot SOC + 0.00001875 \end{aligned} \tag{4.3}$$

These two could then be used in MESSY to simulate the real-time behavior of the battery. To handle battery operations, the platform provides three different SystemC-AMS modules, which are able to accurately estimate the remaining battery capacity, the discharge current, the battery voltage, the v_{oc} and the r_s. The module that has to be customized with such equation is the `Harvester_battery_battery_voc`. This class is the one handling the core battery functionality, which includes updating the SOC based on the current drawn (received from the battery converter through the `i` port) and using it to estimate the voltage and internal resistance. Note that the SOC is updated continuously by solving an equation that takes into account the current drawn in the current and previous time instants. Moreover, this equation was split in several sub operations in order to prevent overflows, which appeared in the original implementation for battery capacities exceeding 250 mAh.

Additionally, this module is also used in order to check the current status of the battery. In particular, if the SOC falls below 10%, the system triggers an alert, which reaches GVSoC. As seen in Section 4.3, this condition instructs the program to force the drone to go down, and the simulation to terminate. This approach ensures a more realistic testing condition, preventing the battery from falling below 0%.

Finally, it is worth mentioning that the `battery_converter` works as previously discussed: it gets all the system currents and calculates the total one as the sum of the motors contribution (efficiency = 1 due to the direct connection) and the current coming from the sensors (scaled by the efficiency, which is found to be $\frac{V_{out}}{V_{in}}$ as shown in Equation 4.1). The latter is also checked for being in the range of 0 to 1: if it does exceed these boundaries, it is saturated to either 0 or 1, respectively, in order to avoid possible errors.

Chapter 5

Experimental Results

5.1 Scenario overview and power models comparison

Once the architecture was correctly developed, it was possible to test the virtual platform with some simulations to verify its correctness. In order to do so, a new world file was first created in Webots. This was then setup: a sequence of gates were added, along with the drone, in order to match the wanted scenario. The result of the 3D simulation environment has been reported in Figure 5.1. As

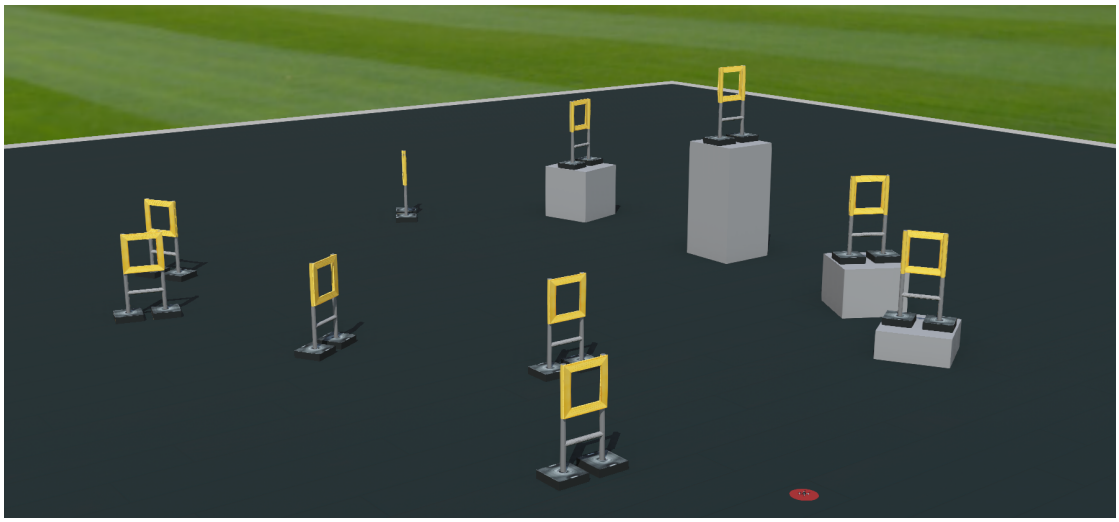


Figure 5.1: Overview of Webots 3D environment.

previously highlighted, the drone is meant to start from ground (red circle on the bottom of the image), lift up and then traverse the gates (the yellow squares) in

order. Then, the start position is reached again and the drone is brought down to ground to stop the motors.

The simulation was first performed using the default Webots timestep, which is automatically set to 32 ms. This value was manually set in the Webots controller to ensure the alignment between the two. On the MESSY side, instead, the files for the core, sensors, etc. were created with the `codegen` command. After porting all the necessary modifications (the ones evidenced throughout Chapter 4), the simulation was finally started by compiling the GVSoC program and launching it. Note that a value of 0.1 m was chosen for the `tolerance`, the TARGET CYCLES were set to 100 and the UNLOCK CYCLES to 5000. Debug mode was set to level NONE to maximise the performances.

During the run of the application, it was possible to see the drone starting from the initial position and traversing the gates one by one, as shown in Figure 5.4. At

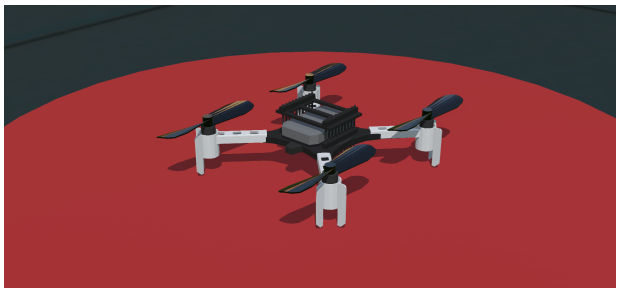


Figure 5.2: The drone at the beginning of the simulation.

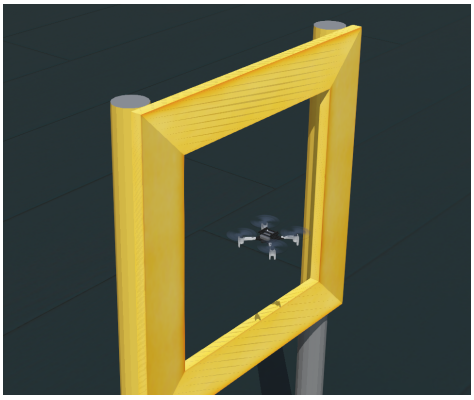


Figure 5.3: The drone passing a gate.

Figure 5.4: Shots of the Crazyflie drone during the simulation.

the very end of the simulation, the Webots controller was instructed to print the total simulation time on the terminal, and the same was done by MESSY. In the latter case, two values were actually printed: the simulation time of the SystemC core and the estimated Webots time (which differs from the real one due to the errors introduced by GVSoC, as seen in 4.2.4). These information were printed in order to be collected and analyzed, as seen in the next paragraphs.

In addition to this, a Python script was set to be automatically executed at the end of the simulation. This collects the power and battery data from the traces and the motors log files in order to plot the estimated power consumption of the three models, as well as the battery level throughout the simulation. For the specific configuration settings specified at the beginning of this chapter, the

obtained graphs were the ones reported in Figure 5.5. In particular, the chosen battery model (despite logging each of them, only one is selected for the evaluation of the battery charge) was set to the empirical one (explained earlier in Section 3.4.3).

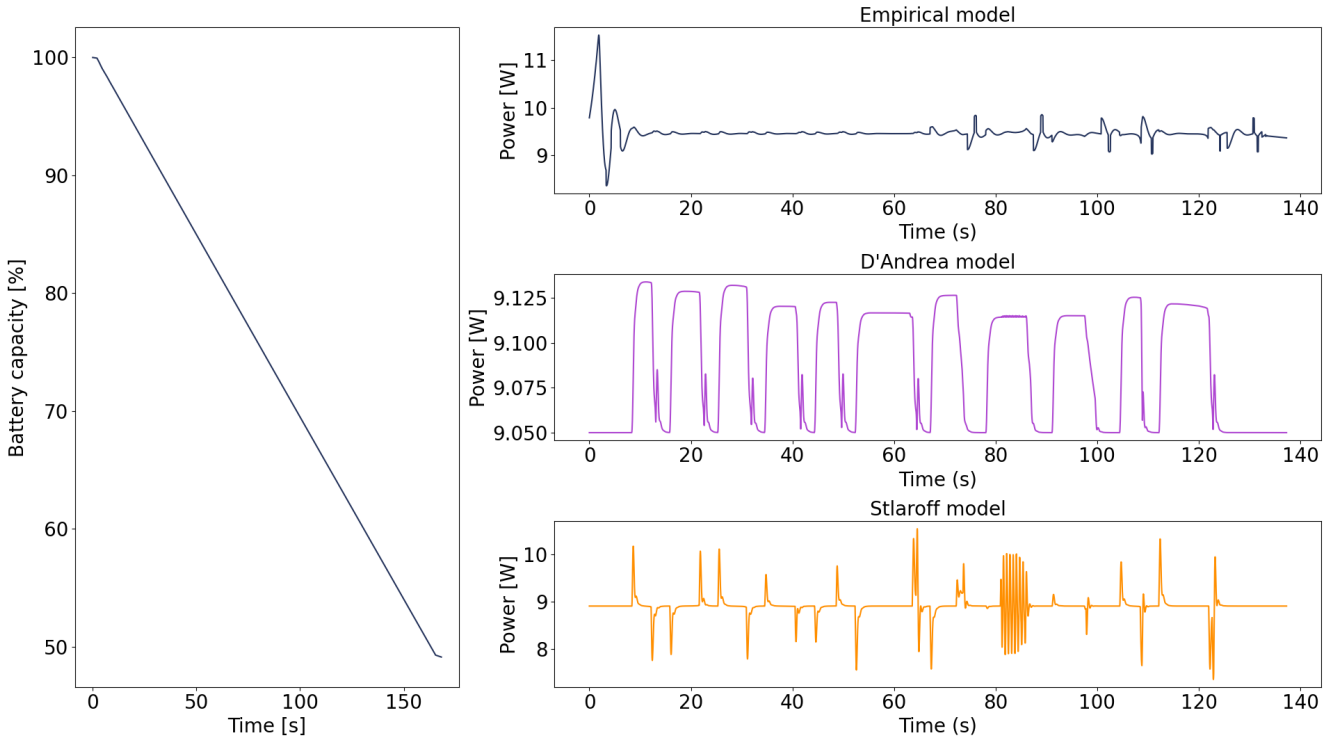


Figure 5.5: Power and battery information shown at the end of the simulation.

By looking at the X-axis of the battery graphs, the simulation lasted about 150 *s*. However, there is a difference in the range of this graph and the ones of the battery models, with the latter reporting a time of about 140 *s*. Two are the main causes of this gap. First, it should be noted that, while the battery graph is based on the data coming from MESSY, the others reports are based on the Webots simulation, which is called to terminate before the one of MESSY. Moreover, the collection of the data for the motor log files is set to exclude the cases in which the velocity of the drone propellers is 0: this data is not useful to be plot and would also cause issues in the readability of the graphs, greatly increasing the Y-axis scale range. This implies that the represented values exclude the few seconds at the very beginning and at the very end of the simulation.

By looking at these graphs, few observations can already be made. For example,

Bitcraze declares a battery life of about 7 minutes (420 s) while hovering[1] for the drone. The current simulation seems not to diverge too far from that value, as about 50% of the battery charge was consumed in about 160 s while moving, leading to an estimated battery life of about 330 s.

The battery discharge is almost linear, with a different slope in the very last instants: this is because in this time the four motor values are set to 0, and the only power drawn from the battery comes from the electronics. The power models instead show evident spikes in several points of the simulation.

The first model, the empirical one, shows a great increase in the power during the initial lift (as the motors are set to higher RPM values to reach in the fastest way possible the stable altitude), with only minor oscillations during the drone movement and descent. This was sort of expected, as the only variable factored into the equation is the motor velocity, which is just slightly modified during the drone movement.

In the D'Andrea model (the second one), it is possible to notice 11 spikes, corresponding to the time instants in which the drone was traversing the 10 gates and then returning to the initial point. The rest of the time, where the drone simply hovers, the consumption remains flat to the 8.7 W indicated by the `P_hover` variable: this is because this model is directly proportional to the drone velocity in the X/Y plane, obtained by the Webots GPS sensor.

Lastly, the Stolaroff model reported significant transients in multiple points of the simulation. These were found to be much shorter in time than the ones from D'Andrea, as the model is greatly influenced by the pitch angle which varies significantly especially during accelerations and decelerations. In a specific point of the simulation, the graph is particularly messy: this is due to the corrective maneuvers performed by the PID when the drone falls out of the `tolerance` region, which makes it rapidly take the opposite direction causing significant oscillations before it is effectively able to stabilise itself. Note however that the data shown for the Stolaroff model was obtained by multiplying by a factor of 5 the original power consumption points to obtain a plausible range of values. Unfortunately, being the model targeted to a very different kind of drones, it was not able to provide realistic data for nano drones. Nevertheless, it was still worth it to include it as a comparison and to understand how the drone maneuvers affect its consumption.

This correction allows to obtain, per each model, data that is in the range of expected values and close to the measurements performed on the physical drone model [54]. As a result, also the battery discharge curves of the three models are very similar between each other, as reported in Figure 5.6.

[1] `https://www.bitcraze.io/support/f-a-q/`

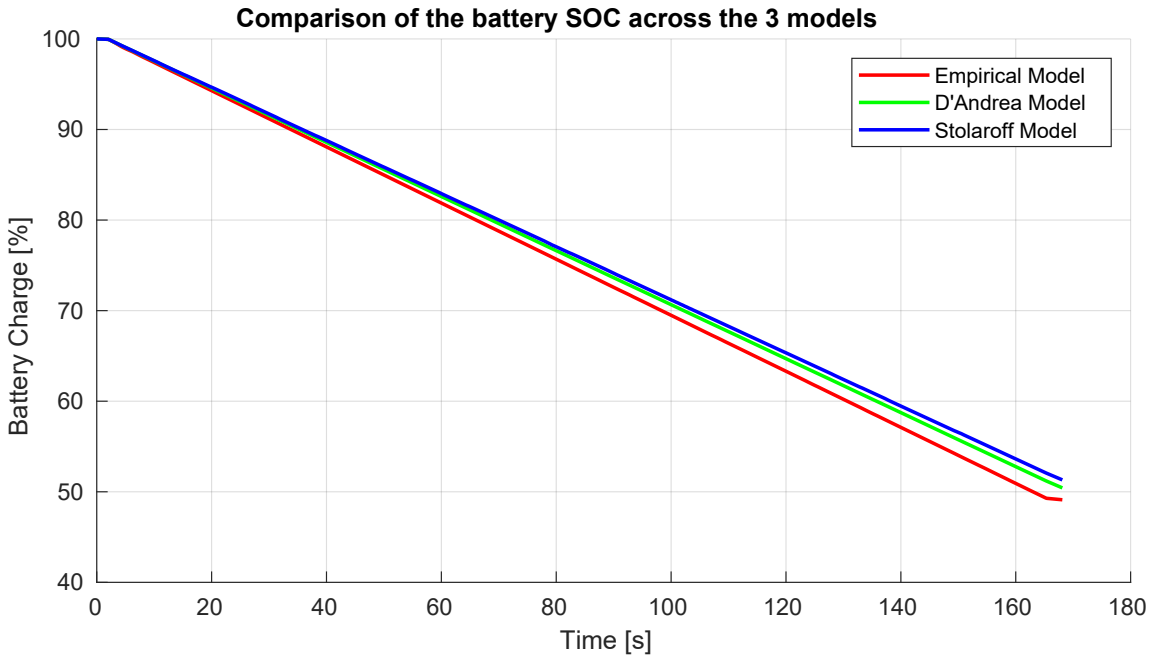


Figure 5.6: Comparison of the battery discharge curves.

5.2 System-level simulation overhead and time offset

Once that the simulation was effectively validated, it was possible to start with more focused tests. In particular, the first ones were oriented towards the evaluation of the impact on performance that the introduction of MESSY has brought into Webots, with the introduction of the UNIX socket communication, the simulation of the real hardware, and the power consumption estimation. To do so, the GVSoC program, as well as the functions of the STM32 and the camera, were brought inside a new controller and adapted in order to perform the same operations using only Webots. By comparing the execution times of the two simulations, it was then possible to estimate the overhead of the virtual platform approach.

To perform the tests, a Linux machine was used. The system was powered by a quad core Intel Core i5-8250U processor, which has an integrated graphics (Intel UHD Graphics 620), and 8 GB of DDR4 RAM running at 2400 MHz. Before each test, the CPU (Central Processing Unit) load was taken to the 10-15% range by closing any other user process.

To measure the execution time of the Webots simulation, the command reported in Listing 5.1 was launched.

Listing 5.1: Command to measure Webots execution time

```
time webots --log-performance=/home/performance_log.txt --no-rendering --stdout --minimize --mode=fast /home/user/Desktop/Crazyflie\_gates\_world.wbt
```

The UNIX `time` command was used to execute the program while tracking the elapsed time from start to finish (using the *real* value, which included the time spent waiting for I/O and other processes). Webots was then launched with a series of arguments:

- `-log-performance`, which makes it possible to track the simulation performance in a custom file;
- `-no_rendering`, which disables the 3D rendering of the scene, to ease the load on the system;
- `-stdout`, to redirect the Webots console output to the standard output (useful to manually terminate the program at its completion);
- `-minimize`, to let the process start in background, saving a bit of system resources;
- `-mode=fast` to enable the simulation to go at the fastest speed possible rather than 1x (i.e., real time).

Unfortunately, the Webots process does not terminate itself as the controller returns, so the program was manually stopped as soon as the last simulation prints appeared on the console output. To mitigate the possible errors due to human reaction times and other conditions affecting the system speed, each measurement was repeated three times and an average of those was used as the final value.

The tests on the performance drop were done on the basis of different timesteps to test very different simulation times (with finer time granularity, the number of PID iterations, on the Webots side, is much higher). Moreover, this approach also allowed to verify the discontinuities between Webots and MESSY time. In fact, at the end of the simulation MESSY also produces the estimated Webots time, which however is different from the real one due to some unknown GVSoC-related delays, as formerly explained in Section 4.2.4. It is important to underline that these tests were performed on the exact same 3D environment, by properly adjusting the `basicTimeStep` variable at each iteration and by recompiling the controller to use that same value as timestep.

The system was tested on the 8, 16, 24 and 32 *ms* timestep. Running the Webots-only simulation led to the values of Table 5.1. This includes the effective user execution time (*Exec. time*) as well as the total duration of the simulation (*Webots*). The times obtained including MESSY, instead, were reported in Table

5.2. The columns represent, in order, the timestep, the user execution time, the SystemC core time, the Webots time, the estimate of the Webots time performed by MESSY and error between the latter two (expressed as a percentage of the real value).

Timestep	**Exec. time**	**Webots**
8	71239	108896
16	42108	122208
24	30536	132264
32	25212	145592

Table 5.1: Simulation time measurements excluding MESSY (data in ms).

Timestep	**Exec. time**	**MESSY**	**Webots**	**Webots est.**	**Webots est.-real (%)**
8	-	-	-	-	-
16	212343	163732	137644	137730	0.0625
24	241893	200416	131736	131818	0.0622
32	212006	168113	142144	142210	0.0464

Table 5.2: Simulation time measurements including MESSY (data in ms).

Speaking of the first table, it is possible to see how the execution time was found to be linearly related to the timestep: since the simulation is composed of more iterations, even if the effective time to be simulated is lower, the number of advancements to be performed increases as the timestep decreases. On this note, the reader should be reminded of the fact that the Webots simulation lasts less for lower timesteps, as the PID is able to perform a more accurate tuning of the motor values to reach the destination, which leads to the need of less cycles.

Comparing the Webots results with the ones of the other table, the values were found to be quite consistent. The few s differences between the Webots time of the two simulations may be imputable to the sensor delays, which also influence the PID response and, consecutively, the behavior of the drone. On the topic of the PID, note that no value is present for the 8 ms timestep: being the single PID iterations dependent on the time elapsed from the previous one, changing timestep leads to different time differences, which generates different commands for the motors. During such simulation, the generated motor velocities made the drone collide with one of the gates and flip over itself. Being the current versions of the PID not able to handle such situations, it was impossible to terminate the simulation. This same observation was also true for timesteps greater than 32 ms: with the provided C code, it was not possible to build a "universal" controller, requiring the developer to modify the position of the gates or to further adjust the

PID gains with a set of trial and errors in case other simulation timesteps were needed.

The two tables also allow to get an idea of the performance loss that are introduced in the simulation by including MESSY. In particular, it is possible to see that the time taken by the system to simulate the standard scenario varies from 30 to 70 s, while about 220 s are needed to run the simulation with MESSY: in this case, the performance hence degraded by a factor that oscillates between 66% and 88%. This result was sort of expected: with MESSY, several processes are run in parallel and should be able to communicate a lot of data (e.g., camera images, which weight over 0.1 MB each, are sent about 4000 times) between each other, while that is not the case for the Webots-only simulation. On top of that, the compiler may be doing some optimization on the Webots code, as elements such as the camera images are generated but never used by that controller.

Finally, it was possible to see that the misalignment that forms between the real and the estimated Webots time is very marginal, with values being always lower than 0.1%. As previously noted, several modifications were introduced during the tests to identify the origin of such a problem. In particular, it was found that this offset seemed not to appear when the drone was set to run continuously, i.e. without needing it to send an image and perform one PID iteration every 33 ms. This approach eliminates the loop used for the PID on the GVSoC side and transfers it to MESSY (which does it using the `PID_loop` function), making the former program execute drastically less instructions. Since this modification allowed to have a ms precision of the Webots estimation time, the most probable cause of the offset is hence related to some small time deviations that arise during the simulation of the GVSoC instructions.

5.3 Changing simulation parameters

The simulation was then changed in order to verify the system behavior in other conditions. In particular, the `CORNER_MODE` macro was activated in the GVSoC program to evaluate again the execution times, the offsets of the two simulations and the degradation of the system performances with a different kind of drone trajectory. In particular, the code was set so that the drone is instructed to reach the four corner points of the map and return back to the start position. The new set of measurements for the Webots-only simulation led to the values of Table 5.3, while Table 5.4 contains the ones that were obtained by including MESSY.

With respect to before, the simulation lasted longer, due to the different path taken by the drone. Comparing the obtained results with the ones of Table 5.1 and Table 5.2, it is possible to see that, while for the Webots-only simulations the execution time is linearly dependent on the timestep (i.e., on the number of

Timestep	Exec. time	Webots
8	124399	184504
16	65863	191472
24	45971	197952
32	36173	204096

Table 5.3: 4 corners time measurements excluding MESSY (data in ms).

Timestep	Exec. time	MESSY	Webots	Webots est.	Webots est.-real (%)
8	348069	235223	195632	195731	0.0506
16	342202	239287	199616	199715	0.0496
24	441168	307773	197736	197864	0.0647
32	340446	203872	203872	203970	0.0481

Table 5.4: 4 corners time measurements including MESSY (data in ms).

advancements performed during the simulation), that was not the case for the ones run with MESSY. In the latter case, the machine time seems to be completely independent: this is probably related to the fact that the CPU is very constrained by the socket connection, and, since the number of PID iterations remained almost the same per each case (simulations take very similar times, and the operations are always performed at a 33 ms rate), the advancement of multiple timesteps at a time has almost no impact on the performances. In both scenarios, an exception should be made for the 24 ms case: this simulation always took longer to complete, probably due to the fact that this is the only number not belonging to the set of powers of 2, which makes computations less efficient on the MESSY side. Note also that, in this case, the absence of obstacles made it possible to test the simulation performances with the 8 ms timesteps, as the drone could not collide with objects anymore.

Another meaningful observation is the fact that execution time does not depend on the used power model, as the simulation kernel is advanced of the same amount of time independently on the operations to be performed. The models may have an impact on the time taken by the machine to run the program, due to the specific calculations being performed per each model. However, since standard C++ operators were used for the motor consumption equations, no practical difference was found.

Speaking of the actual performance of the simulation, these were found to be consistent with the ones of Section 5.1: the drop in performance with respect to the Webots only simulation (expressed as a percentage) was in the order of 64% - 90%. The result again would have greatly improved by limiting the amount of

images sent over the socket with the usage of the `PID_loop` function rather than the current setup.

This new simulation also confirmed two important observations: the estimated Webots time is evaluated very accurately by MESSY (with an error that is always lower than 0.1%), and there is a strong consistency between that value and the real one, with minor changes imputable to the differences in the generated motor values.

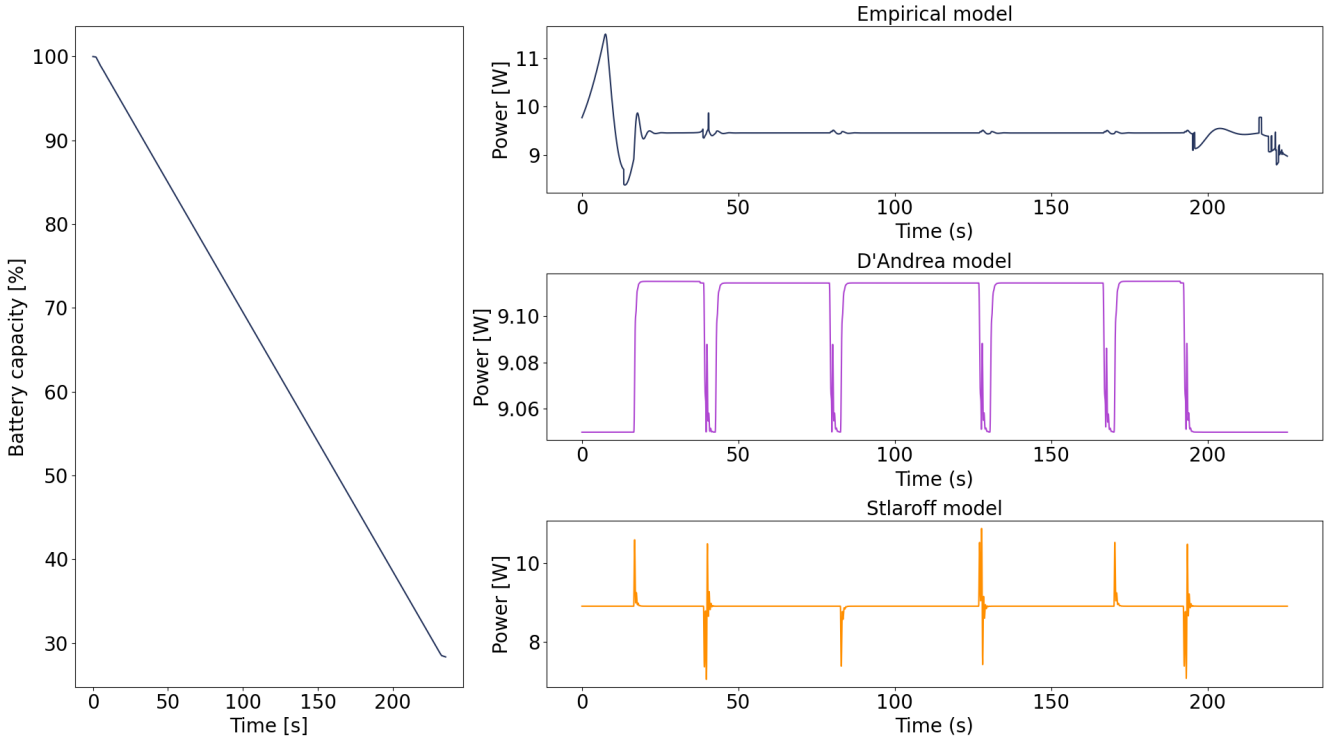


Figure 5.7: Power and battery information (empirical model) for 4 corners simulation.

Having a look at the consumption data (reported in Figure 5.7 considering the empirical model with the 8 *ms* timestep), it is possible to notice that the longer simulation led to a higher battery consumption, as expected. Considering the three power models in general, less peaks are found (only five, one per each of the four targets and the final one to reach the start position). Also the oscillations have reduced drastically, especially when considering the Stolaroff model: the velocity is maintained constant most of the time, without rapid direction changes as relative accelerations/decelerations. Moreover, the battery discharge was found to be very consistent between the three models, and among the different timesteps as well. As an example, the discharge curve of the battery (according to the empirical

model) for the 8 ms, 16 ms and 32 ms timesteps have been reported in Figure 5.11. The resulting charge drop is slightly dissimilar, which is imputable to the different length of the simulation given by the different motor commands produced by the PID: as seen in Section 5.1, in fact, the tests proved again that lower simulation times are obtained with lower timesteps.

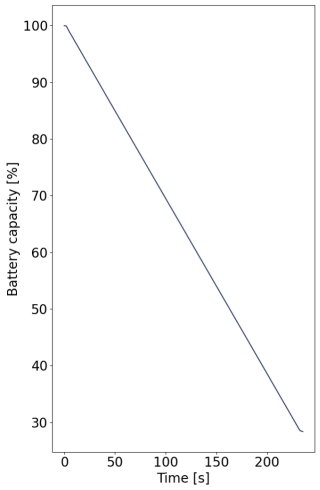


Figure 5.8: 8 ms timestep.

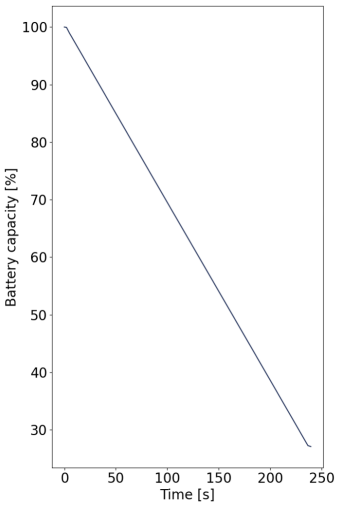


Figure 5.9: 16 ms timestep.

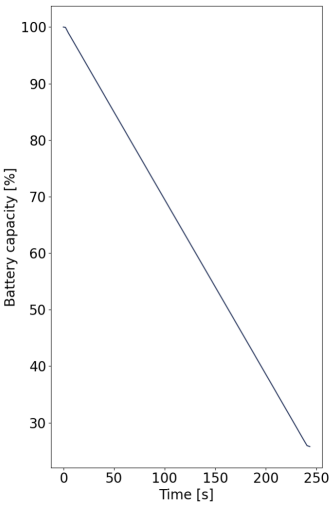


Figure 5.10: 32 ms timestep.

Figure 5.11: Battery discharge curves for different timesteps.

5.4 Testing different batteries

In this paragraph, a deeper focus on power is presented, in order to provide an example of the usefulness of the developed virtual simulation platform for DSE. Among the different choices that could be made to improve the drone performance, the selection of the battery is surely one of the most noteworthy. Larger batteries provide additional power to the system, but their additional weight may counterbalance the benefits. Moreover, larger batteries may be more expensive, and being able to simulate their behavior is thus a very valuable strategy for saving unnecessary costs.

Three batteries were chosen to be tested against the original one. Note that, to change battery model, only the battery capacity, its mass and the fourth-order equations contained in the `Harvester_battery_battery_voc` should be updated with the new ones. As per the prior case, the latter may be obtained from the respective discharge curves by means of the Digipyze program [63]. The obtained

values for the two equations were then included in this module, in order for the user to select which battery to simulate by adjusting the `BATTERY_MODEL` macro defined in the `simulation_configs.cpp` file. Speaking of the configuration file, note that the same environment was used throughout the various simulations, with the same settings specified in Section 5.1 (drone traversing 10 gates, 32 ms timestep).

The chosen battery model was the Stolaroff one. This was selected as it is going to be the most accurate between the three: despite being based on a corrective coefficient, is the only one that truly takes into account the effect of mass, as the D'Andrea model includes a fixed hovering power (which however should change linearly with the mass) and the empirical one does not factor it in at all.

The first battery that was tested was the UFX402525 250 mAh 3.7 V 25C battery from *Guangdong UFine New Energy Co.* [64]. This was chosen to see if any effect can be noticed by using a battery with identical specifications coming from another manufacturer. With respect to the previous one, this battery can be bought for cheaper, and it weighs less than the original one (5.8 g versus 7.1 g). The scope of the simulation was then to check whether the choice of this model is truly more convenient, or not. Its discharge curve is represented in Figure 5.12, while Equation 5.1 and Equation 5.2 report the models (equations) for the internal series resistance and open circuit voltage.

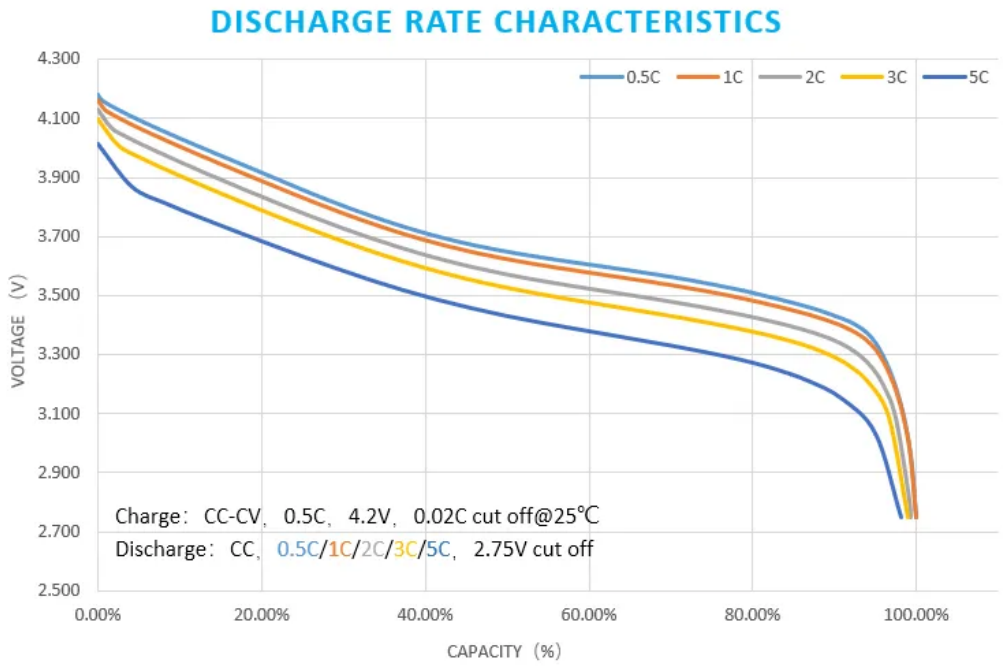


Figure 5.12: UFX402525 battery discharge curves.

$$v_{oc} = -3.55202 \cdot SOC^4 + 5.59181 \cdot SOC^3 - 1.95993 \cdot SOC^2 - 0.98731 \cdot SOC + 4.16759 \tag{5.1}$$

$$r_s = 0.0013721 \cdot SOC^4 - 0.0018010 \cdot SOC^3 + 0.0006004 \cdot SOC^2 + 0.00000473 \cdot SOC + 0.00017039 \tag{5.2}$$

The power information obtained after the simulation is reported in Figure 5.13.

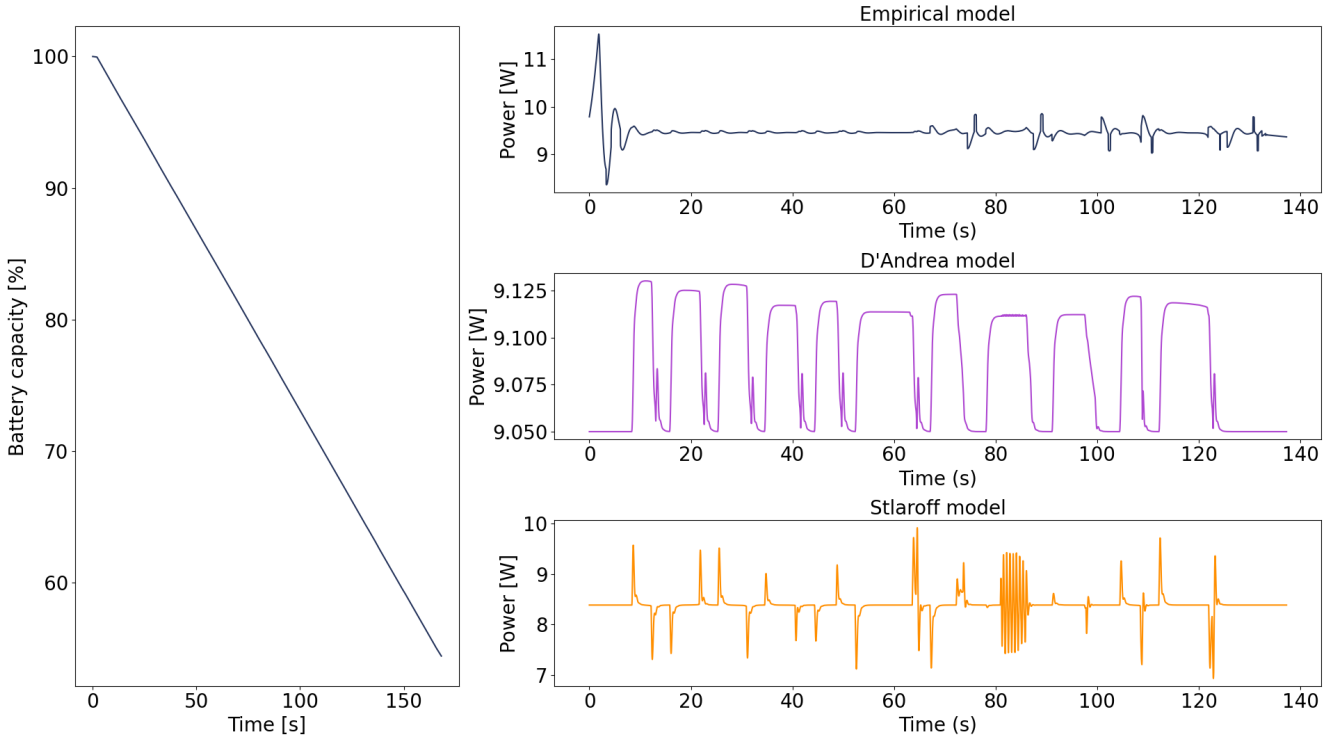


Figure 5.13: UFX 250 mAh battery results.

By comparing the performances of this battery with the original ones, it is possible to state that the two showed a very comparable behavior. However, the UFX battery seemed to have a slight advantage in terms of efficiency: the battery discharged by 3% less at the end of the simulation, making it de-facto the best possible choice when comparing it to the original one considering both performances and cost. The effectiveness of the battery is surely related to its lower weight: by looking at the consumption data, it was possible to see that per each model less power was consumed with the UFX one. This advantage greatly reduces the needed

amount of power, making the differences introduced by the internal behavior of the battery less significant.

A second battery was included in order to verify the system behavior in the presence of a higher capacity. The model that was considered was the 300 mAh 3.7 V 20C battery from *Cyclone* [65]. In particular, this was chosen in order to test whether the additional energy would benefit the simulation, by considering a product from yet another manufacturer. The battery weighs 8.1 g, and its dimensions are not too far off from the ones of the original one (WxHxD = 30x3.5x48 mm). Figure 5.14 represent the relation between battery charge and voltage, while the corresponding equations have been reported in Equation 5.3 and Equation 5.4.

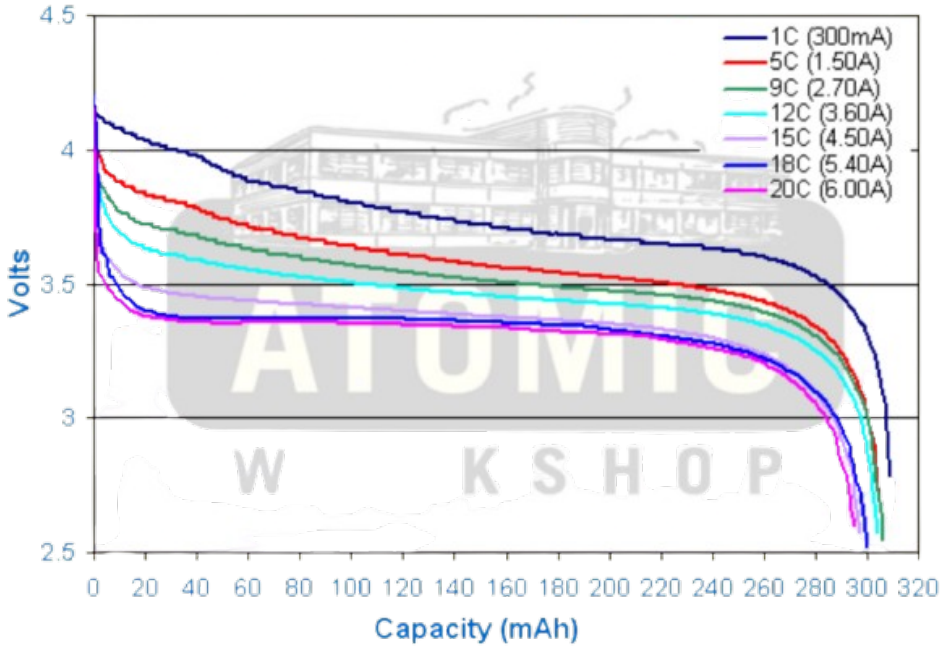


Figure 5.14: Cyclone 300 battery discharge curves.

$$v_{oc} = -2.88116 \cdot SOC^4 + 3.95055 \cdot SOC^3 - 0.75681 \cdot SOC^2 - 1.06078 \cdot SOC + 4.15788 \tag{5.3}$$

$$r_s = 0.00065739 \cdot SOC^4 - 0.0008854 \cdot SOC^3 + 0.00043203 \cdot SOC^2 - 0.0001729 \cdot SOC + 0.0001722 \tag{5.4}$$

The results have been reported in Figure 5.15. Also in this case, the selected battery appeared to overcome the performance of the original one. In this scenario,

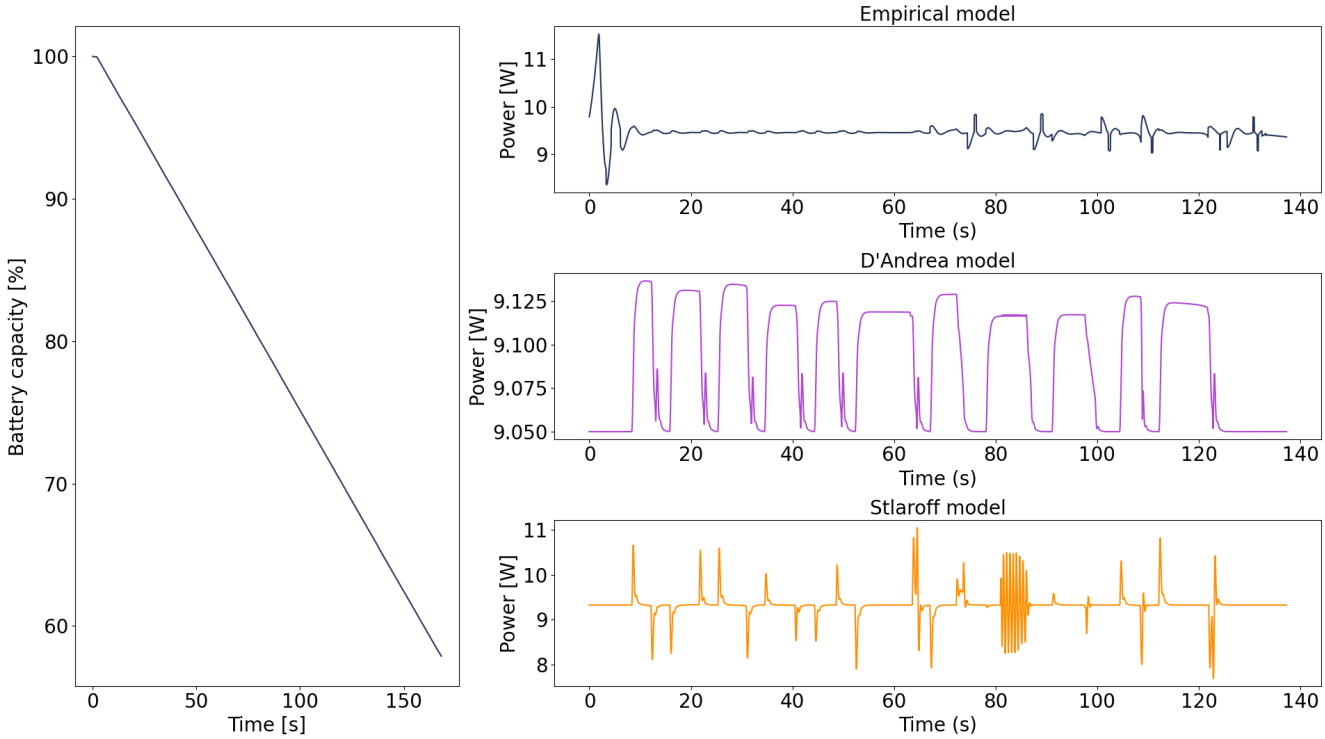


Figure 5.15: Cyclone 300 mAh battery results.

the additional capacity made the difference: the higher weight (which is however limited to 1 additional gram) was counterbalanced by the extra amount of energy made at disposal by the Cyclone battery, leading to better overall results. This comes at a greater expense however, opening the need to find the best ratio between performances and costs.

Finally, a third battery was tested to simulate the Crazyflie behavior for an even higher capacity. This was done as Bitcraze also sells 350 mAh batteries from *Tattu* that can be used to improve the hover-time (up to about 3 additional minutes [2]) at a higher cost. Unfortunately, it was impossible to gather the needed curves for the original battery (very few LiPo manufacturers actually make these public), but the LPHD7820030 model from *LiPol Battery* was close enough to the Tattu one, considering both dimensions(20x8x33 vs 20x7.8x30 mm), capacity, voltage (3.7 V), weight (9.1 g vs 9.2 g), number of cells (1) and C-rate (15C). The company provides for this model the same battery discharge curves that have been reported

[2] `https://store.bitcraze.io/collections/accessories/products/350mah-lipo-battery`

in Figure 4.9, while the v_{oc} and r_s equations were calculated accordingly to the new capacity and are reported in Equation 5.5 and Equation 5.6, respectively.

$$v_{oc} = 1.39609 \cdot SOC^4 - 2.95149 \cdot SOC^3 + \\ 1.60142 \cdot SOC^2 - 0.10038 \cdot SOC + 4.15517 \tag{5.5}$$

$$r_s = 0.0019507 \cdot SOC^4 - 0.0041482 \cdot SOC^3 + \\ 0.0022543 \cdot SOC^2 + 0.00003978 \cdot SOC + 0.00001339 \tag{5.6}$$

The obtained performances have been reported in Figure 5.16. Also in this case,

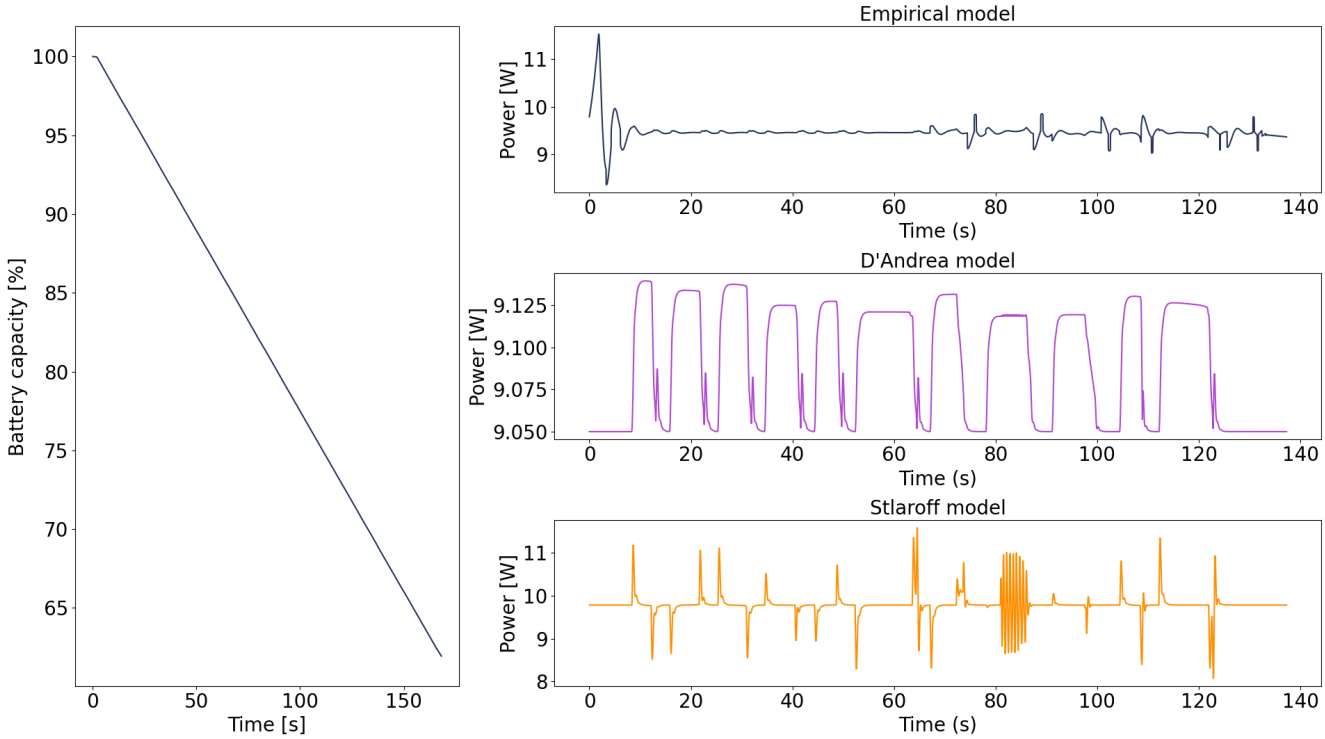


Figure 5.16: LiPol 350 mAh battery results.

the additional capacity made it worth the upgrade: the battery was able to last even more than the previous one, leading to an estimated flight time that was a bit lower than the declared one (450 s instead of 600 s). Hence, this test led to another Pareto point, as this battery has proved to provide the best performance at the highest cost. A company should carefully evaluate these aspects to identify the best choice, along with considering the scope of the product. For example,

remember that the Crazyflie drone was built as a modular device, with several decks that can be purchased to enhance the system functionalities. Hence, it is not always worth the extra battery life: the additional grams introduced by the battery would also limit the payload that could be transported by the drone, which is not necessarily the best option.

As a final comparison, a curve including the battery discharges per each of the three batteries has been reported in Figure 5.17.

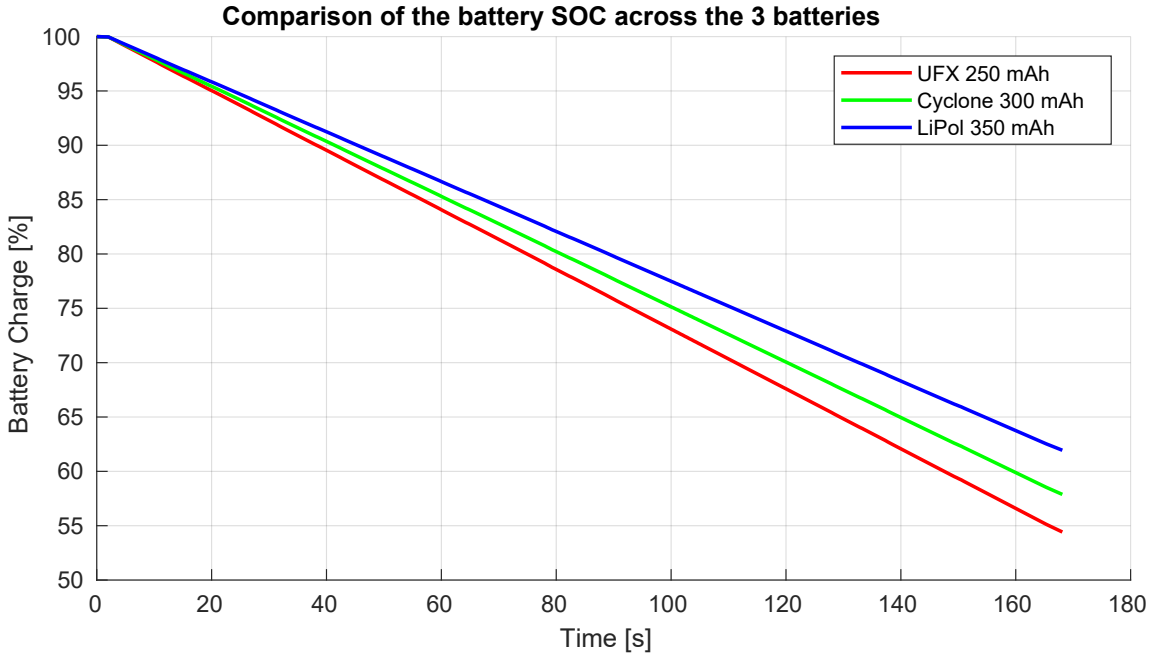


Figure 5.17: Comparison of the battery discharge with the three batteries.

These simulations surely allow companies to make more informed choices, but may benefit the customer as well, as they could be used to evaluate if the battery upgrade is worth for the desired use case or not.

Chapter 6

Conclusions and future works

This thesis has successfully proven the effectiveness of MESSY in modelling functional and extra-functional parameters of a complex robotic device. The inclusion of Webots allowed the virtual platform to provide a complete test environment to verify the Crazyflie behaviour, as well as its power performance. Nonetheless, the system may benefit from further works in multiple aspects.

The first one is related to the improvement of the drone implementation. In particular, at its current version, the drone PID controller suffers some known limits that were also evidenced during the tests. Being a very simple implementation, this control algorithm is not capable of handling the yaw rotation correctly, meaning that the drone is able to change its altitude and speed, but struggles when dealing with rotations. Moreover, this issue makes the drone not capable of recovering from drone collisions or rapid speed changes, making the system response potentially unbound and hence more difficult to perform simulations with different timesteps. On the topic of the PID, the partial and integral PID gains were experimentally adjusted to provide a sufficiently good response, but additional tests may help in further tuning these parameters.

On the MESSY side, the simulation capabilities could be further enhanced by adding the models of the other sensors of the Crazyflie, such as the GPS or the IMU. A deeper study on the firmware could also be beneficial, in order to model more accurately both the functional and the power specifications of the STM32, the camera and the nRF chip. Moreover, BitCraze actually provides some Python bindings for the real firmware [66], which could be integrated in MESSY to have a true representation of the performed operations.

On the GVSoC side, instead, a neural network could be developed to generate the target coordinates on the basis of the image captured by the camera. This

approach would help to simulate with a much higher accuracy the capabilities of the algorithm, and would be more realistic especially for the extra-functional aspects.

Another point to consider regards the power models that were used to estimate the motor power consumption. Despite being sufficiently accurate for the purposes of this thesis, those still did not consider various aerodynamic and physics effects which would significantly influence the total consumption. As a result, even the most accurate model of the onboard electronics would become meaningless, as the contribution of the four propellers is undoubtedly the biggest. Models such as [53] or [50] could be implemented to mitigate or even solve these issues, allowing to obtain a true and accurate functional and extra-functional representation of the system performance.

In addition to all of this, other IPC mechanisms could be put under test in order to explore the performance of the other approaches, which may be found to be more effective than the UNIX sockets and would help in reducing the performance gaps. Some speedups may also be obtained by opting for much simpler packets, as the inclusion of the JSON library surely introduced a significant overhead, especially for smaller messages.

Nevertheless, the obtained virtual platform already produced satisfactory and meaningful results. An example of verification and DSE was provided in this thesis, and the system showed to be entirely functional with output results closely matching the real model data.

Together, these considerations underline the potential and the ease of customization provided by MESSY, showcasing its capabilities as a very powerful functional and extra-functional simulation environment for RISC-V based systems.

Appendix A

Additional power models

Gong et al.

This model [67] is much more detailed with respect to the considered ones, with strong focus on nanodrones with multirotors characteristics. It combines the different stages of the drone flight: ascend, horizontal flight and descent.

Chan and Kam

Chan and Kam [68] propose instead a systematic evaluation of the contribution of propellers and electric parts (focusing in this case on the losses along the power path). Also in this case, the contributions are splitted across the different phases of the drone flight, such as hover, tilted flight and descent.

Jacewicz et al.

In this case [52], the model considers aspects such as momentum of inertia (measured experimentally) and non-linear aerodynamics aspects.

Phung and Moring

The model proposed by Phange and Moring [69] targets the Vertical Take Off and Landing (VTOL) UAVs and was developed combining the study of propeller profiles, momentum and blade theory.

Michel et al.

Finally, this model [53] tries to incorporate all the contributions in one single system of equations. The included aspects are physics, aerodynamics of the rotor-propeller assembly, battery, electro-mechanical dynamics of motors and motor controller, plus the rigid body dynamic of the airframe.

Appendix B

VirtualConnector class

```
1     json VirtualConnector::read_from_channel(int fd) {
2
3     int num_read, counter_read, string_length;
4     unsigned char rd_len_buffer[4];
5     char *rd_json_buffer;
6     json data;
7
8     // Read how many characters the json will be long
9     string_length = 0;
10    num_read = read(fd, &rd_len_buffer, 4);
11
12    for (int i = 0; i < num_read; i++)
13        string_length = string_length | (rd_len_buffer[i] << (8 * i))
      ;
14
15    #if DEBUG_LEVEL >= DEBUG_LEVEL_HIGH
16    std::cout << "Incoming packet of length: " << string_length <<
      endl;
17    #endif
18
19    // Use the number to allocate memory required to store the json
      string
20    num_read = 0;
21    counter_read = 0;
22    rd_json_buffer = (char *) malloc((string_length + 1 )* sizeof(
      char));
23
24    while (counter_read < string_length) {
25        num_read = read(fd, rd_json_buffer + counter_read,
      string_length - counter_read);
26        counter_read += num_read;
```

```
27
28          if (num_read == 0)
29              break;
30
31      }
32      rd_json_buffer[counter_read] = '\0';
33
34
35      #if DEBUG_LEVEL >= DEBUG_LEVEL_HIGH
36      std::cout << "Packet received with length: " << counter_read <<
        endl;
37      #endif
38
39      // Parse the json
40      data = json::parse(rd_json_buffer);
41      free(rd_json_buffer);
42      return data;
43  }
44
45  void VirtualConnector::write_on_channel(int fd, json data) {
46      unsigned char length_buffer[4];
47      char *wr_buffer;
48      int json_length;
49      std::string json_string;
50
51      json_string = data.dump();
52      json_length = json_string.size();
53
54      #if DEBUG_LEVEL >= DEBUG_LEVEL_HIGH
55      std::cout << json_string << endl;
56      #endif
57
58      length_buffer[0] = json_length & 0x000000FF;
59      length_buffer[1] = (json_length >> 8) & 0x000000FF;
60      length_buffer[2] = (json_length >> 16) & 0x000000FF;
61      length_buffer[3] = (json_length >> 24) & 0x000000FF;
62      write(fd, length_buffer, sizeof(length_buffer));
63
64      wr_buffer = (char *) malloc(json_length * sizeof(char));
65      strncpy(wr_buffer, json_string.c_str(), json_length);
66      write(fd, wr_buffer, json_length);
67      free(wr_buffer);
68  }
```

Appendix C

PID iteration function

```
void Sensor_stm32_wrapper_functional::PID_iteration() {

    Sensor_stm32_wrapper_functional::get_sensor_data();

    pid.time_dt = pid.currenttime - pid.past_time;
    double vx_global = (pid.x_global - pid.past_x_global) / pid.
    time_dt;
    double vy_global = (pid.y_global - pid.past_y_global) / pid.
    time_dt;

    // Get body fixed velocities
    double cosyaw = std::cos(pid.actualYaw);
    double sinyaw = std::sin(pid.actualYaw);
    pid.actualstate.vx = vx_global * cosyaw + vy_global * sinyaw;
    pid.actualstate.vy = -vx_global * sinyaw + vy_global * cosyaw;

    // Initialize other values
    pid.desiredstate.roll = 0;
    pid.desiredstate.pitch = 0;
    pid.desiredstate.vx = 0;
    pid.desiredstate.vy = 0;
    pid.desiredstate.yaw_rate = 0;
    double forward_desired = 0;
    double sideways_desired = 0;
    double yaw_desired = 0;
    double height_diff_desired = 0;

    // Logic for generation of next movement based on current status
    with respect to target
    double x_diff = pid.x_global - pid.targetposition.x;
    double y_diff = pid.y_global - pid.targetposition.y;
```

```
29      double z_diff = pid.actualstate.altitude - pid.targetposition.z;
30
31      // Logic to make the drone follow the shortest path to reach the
        target destination
32      double x_abs = abs(x_diff);
33      double y_abs = abs(y_diff);
34      double z_abs = abs(z_diff);
35      double max_abs = 0;
36
37      if (x_abs > y_abs)  {
38          if (x_abs > z_abs)  {
39              max_abs = x_abs;
40          }
41          else {
42              max_abs = z_abs;
43          }
44      }
45      else {
46          if (y_abs > z_abs)  {
47              max_abs = y_abs;
48          }
49          else {
50              max_abs = z_abs;
51          }
52      }
53
54      double x_coeff = x_abs/max_abs;
55      double y_coeff = y_abs/max_abs;
56      double z_coeff = z_abs/max_abs;
57
58      int temp_counter = 0;
59      if(x_diff > pid.tolerance)  {
60          sideways_desired = x_coeff * 0.5;
61          temp_counter++;
62      }
63      if(x_diff < -pid.tolerance) {
64          sideways_desired = - x_coeff * 0.5;
65          temp_counter++;
66      }
67      if(y_diff > pid.tolerance)  {
68          forward_desired = - y_coeff * 0.5;
69          temp_counter++;
70      }
71      if(y_diff < -pid.tolerance) {
72          forward_desired = y_coeff *  0.5;
73          temp_counter++;
74      }
75      if(z_diff > pid.tolerance)  {
76          height_diff_desired = - z_coeff * 0.2;
```

```
77          temp_counter++;
78      }
79      if(z_diff < -pid.tolerance) {
80          height_diff_desired = z_coeff * 0.2;
81          temp_counter++;
82      }
83
84      if(temp_counter == 0)    { // Meaning: if all the conditions are
    valid (drone is whithin tolerance distance from the target in all
    directions)
85          pid.target_counter += 1;
86          if(pid.target_counter >= TARGET_CYCLES){
87              pid.target_reached = true;
88              *(register_memory + STATUS_REG_OFF) = TARGET_REACHED;
89              pid.num_iteration = 0;
90          }
91      }
92      pid.num_iteration += 1;
93      if( pid.num_iteration >= UNLOCK_CYCLES*32/webots_timestep)  { //
    Scaling the number of cycles to the timestep to avoid possible
    misbehaviors with smaller timesteps
94          pid.target_reached = true;
95          *(register_memory + STATUS_REG_OFF) = TARGET_REACHED;
96          cout << "Warning: target position could not be reached!
    Bypassing to the next one... " << endl;
97          pid.num_iteration = 0;
98      }
99
100     // Updating the desired values
101     pid.height_desired += height_diff_desired * pid.time_dt;
102     pid.desiredstate.yaw_rate = yaw_desired;
103     pid.desiredstate.vy = sideways_desired;
104     pid.desiredstate.vx = forward_desired;
105     pid.desiredstate.altitude = pid.height_desired;
106
107     pid.pidVelocityFixedHeightController(pid.actualstate, &pid.
    desiredstate, pid.pidgains, pid.time_dt, &pid.motorpower);
108
109     // Communicating the values to be applied to the motors, and
    preparing for the next iteration
110     mot1.write(pid.motorpower.m1);
111     mot2.write(pid.motorpower.m2);
112     mot3.write(pid.motorpower.m3);
113     mot4.write(pid.motorpower.m4);
114     Sensor_stm32_wrapper_functional::send_data_to_motor();
115     pid.past_time = pid.currenttime;
116     pid.past_x_global = pid.x_global;
117     pid.past_y_global = pid.y_global;
118 }
```

Appendix D

GVSoC program - Initial definitions and most important methods

```
#include "pmsis.h"
#ifdef GAP_SDK
#define AXI_BASE 0x80000000
#else
#define AXI_BASE 0x20000000
#endif

#define NUM_ITERS 10
#define CAMERA_CTRL_REG_OFFSET 0
#define CAMERA_IMAGE_OFFSET 2
#define STM32_CTRL_REG_OFFSET 102402
#define STM32_STAT_REG_OFFSET 102403
#define STM32_TARGET_POS_X_OFFSET 102404
#define STM32_TARGET_POS_Y_OFFSET 102412
#define STM32_TARGET_POS_Z_OFFSET 102420

#define GET_IMAGE 1

#define INIT_PID 1
#define GET_STABLE 2
#define SET_TARGET 3
#define RUN_SINGLE_STEP_PID 4
#define RUN_PID 5
#define TARGET_REACHED 6
#define GO_DOWN 7
```

```
26 #define PRINT_SIMUL_TIME 8
27 #define BATTERY_LOW 9
28 #define ITER_ADVANCE 10
29
30 // Uncomment the specific line to use the wanted debug feature
31 // #define DEBUG_MODE
32 // #define CORNERS_TEST 4
33
34 // Define the number of times the circuit is repeated throughout the
       simulation
35 #define NUM_OF_CIRCUIT_LOOPS 1
36
37 // Main variables
38 char* camera_control = (volatile char *)AXI_BASE +
       CAMERA_CTRL_REG_OFFSET;
39 char* camera_image = (volatile char *)AXI_BASE + CAMERA_IMAGE_OFFSET;
40 char* stm32_control = (volatile char *)AXI_BASE +
       STM32_CTRL_REG_OFFSET;
41 char* stm32_status = (volatile char *)AXI_BASE +
       STM32_STAT_REG_OFFSET ;
42 double* stm32_x = (volatile double *)(AXI_BASE +
       STM32_TARGET_POS_X_OFFSET);
43 double* stm32_y = (volatile double *)(AXI_BASE +
       STM32_TARGET_POS_Y_OFFSET);
44 double* stm32_z = (volatile double *)(AXI_BASE +
       STM32_TARGET_POS_Z_OFFSET);
45 int target_reached = 0;
46 int low_battery = 0;
47 int target_point = 1;
48
49 int main(void)
50 {
51     uint32_t errors = 10;
52     uint32_t core_id = pi_core_id(), cluster_id = pi_cluster_id();
53
54     #ifdef DEBUG_MODE
55     // Initial check
56     printf("[GVoC] List of sensors and respective initial data:");
57     check_camera_registers();
58     check_stm32_registers();
59     #endif
60
61     init_pid();
62     get_stable();
63
64     // Debug mode - Corners simulation
65     #ifdef CORNERS_TEST
66
67     #if CORNERS_TEST > 0
```

```
68      printf("[GVoC] Reaching the first target corner\n");
69      set_target(-1, 11, 1.15);
70      #endif
71
72      #if CORNERS_TEST > 1
73      printf("[GVoC] Reaching the second target corner\n");
74      set_target(17, 11, 1.15);
75      #endif
76
77      #if CORNERS_TEST > 2
78      printf("[GVoC] Reaching the third target corner\n");
79      set_target(17, -11, 1.15);
80      #endif
81
82      #if CORNERS_TEST > 3
83      printf("[GVoC] Reaching the fourth target corner\n");
84      set_target(-1, -11, 1.15);
85      set_target(-2.6, 0, 1);
86      #endif
87
88      #endif
89
90      // Standard mode - gate traversing simulation
91      #ifndef CORNERS_TEST
92      #ifdef NUM_OF_CIRCUIT_LOOPS
93      for(int i = 0; i < NUM_OF_CIRCUIT_LOOPS; i++) {
94
95          set_target(-0.8, 1.5, 1.00);
96          set_target(2, -0.4, 1.00);
97          set_target(4.7, 1.7, 1.00);
98          set_target(7.7, 3.1, 1.00);
99          set_target(9.8, 2, 1.00);
100         set_target(11.3, -3.5, 1.00);
101         set_target(9.4, -6.4, 2.15);
102         set_target(5.3, -6.5, 3.15);
103         set_target(1.6, -5.8, 1.65);
104         set_target(-0.3, -4.7, 1.35);
105         set_target(-2.6, 0, 1);
106     }
107     #endif
108     #endif
109
110     printf("\n[GVoC] Making the drone descend... ");
111     go_down();
112     printf("ground reached!\n");
113
114     printf("\n[GVoC] End of simulation! \n[GVoC] Bye from GVSoC!");
115     print_simul_time();
116
```

```
117     return errors;
118 }
119
120 void set_target(double x, double y, double z) {
121
122     if(!low_battery)    {
123         printf("\n[GVoC] New destination: point %d...", target_point)
    ;
124         target_point +=1;
125         *stm32_x = x;
126         *stm32_y = y;
127         *stm32_z = z;
128         perform(SET_TARGET);
129
130         while(!target_reached && !low_battery) {
131             get_image();
132             advance_simul(10); // Advancing the simulation by 20 ms
    to emulate the neural network inference time
133             run_single_pid_iter();
134         }
135         target_reached = 0;
136         *(stm32_control) = 0;
137         *(stm32_status) = 0;
138         printf("reached!\n");
139     }
140     if(low_battery) {
141         printf("\n[GVoC] Skipping gate %d: LOW BATTERY", target_point
    );
142         target_point +=1;
143     }
144 }
145
146 void run_single_pid_iter() {
147
148     *(stm32_control) = RUN_SINGLE_STEP_PID;
149     while(*(stm32_status) != RUN_SINGLE_STEP_PID) {
150         if (*(stm32_status) == BATTERY_LOW ) {
151             low_battery = 1;
152             printf("\n[GVoC] Low battery !\n");
153             return;
154         }
155         if (*(stm32_status) == TARGET_REACHED ) {
156             target_reached = 1;
157             return;
158         }
159     }
160 }
```

Appendix E

Webots controller - `main` and `sampleRun` methods

```
1   int main() {
2
3   int server_socket, client_socket;
4   struct sockaddr_un address;
5   Robot *robot = new Robot();
6
7   #ifdef DEBUG_MODE
8   logfile.open("simulation.log");
9   if (!logfile) {
10     cerr << "An error has occurred while opening the logfile!" <<
    endl;
11     return -1;
12  }
13  cout << " " << endl;
14  #endif
15
16  // Socket creation
17  #ifdef DEBUG_MODE
18  logfile << "Sockets creation..." << endl;
19  #endif
20  server_socket = socket(AF_UNIX, SOCK_STREAM, 0);
21  if (server_socket == -1) {
22      cout << "> Cannot create the socket! Exiting... " << endl;
23      return -1;
24  }
25
26  // Fill in socket address
27  memset(&address, 0, sizeof(address));
```

```
28      address.sun_family = AF_UNIX;
29      strcpy(address.sun_path, socket_path.data());
30      unlink(socket_path.data());
31
32      // Bind the socket to a path
33      if (bind(server_socket, (struct sockaddr*) &address, sizeof(
        address)) == -1) {
34          cout << "> Cannot bind path to the socket! Exiting... " <<
        endl;
35          close(server_socket);
36          return -1;
37      }
38
39      // Listen to any process who wants to communicate
40      if (listen(server_socket, 1) == -1) {
41          cout << "> Cannot listen to any connection! Exiting... " <<
        endl;
42          close(server_socket);
43          return -1;
44      }
45
46      // Wait for a client to connect
47      client_socket = accept(server_socket, NULL, NULL);
48      if (client_socket == -1){
49          cout << "Error while accepting connection! Exiting... " <<
        endl;
50          close(server_socket);
51          return -1;
52      }
53
54      // Initialize motors
55      Motor *m1_motor = robot->getMotor("m1_motor");
56      m1_motor->setPosition(INFINITY);
57      m1_motor->setVelocity(-1.0);
58      Motor *m2_motor = robot->getMotor("m2_motor");
59      m2_motor->setPosition(INFINITY);
60      m2_motor->setVelocity(1.0);
61      Motor *m3_motor = robot->getMotor("m3_motor");
62      m3_motor->setPosition(INFINITY);
63      m3_motor->setVelocity(-1.0);
64      Motor *m4_motor = robot->getMotor("m4_motor");
65      m4_motor->setPosition(INFINITY);
66      m4_motor->setVelocity(1.0);
67
68      // Initialize sensors
69      InertialUnit *imu = robot->getInertialUnit("inertial_unit");
70      imu->enable(simul_timestep);
71      GPS *gps = robot->getGPS("gps");
72      gps->enable(simul_timestep);
```

```
73      Gyro *gyro = robot->getGyro("gyro");
74      gyro->enable(simul_timestep);
75      Camera *camera = robot->getCamera("camera");
76      camera->enable(simul_timestep);
77      DistanceSensor *range_front = robot->getDistanceSensor("
    range_front");
78      range_front->enable(simul_timestep);
79      DistanceSensor *range_left = robot->getDistanceSensor("range_left
    ");
80      range_left->enable(simul_timestep);
81      DistanceSensor *range_back = robot->getDistanceSensor("range_back
    ");
82      range_back->enable(simul_timestep);
83      DistanceSensor *range_right = robot->getDistanceSensor("
    range_right");
84      range_right->enable(simul_timestep);
85
86      // Wait for 2 seconds (required by the current implementation of
    get_stable_position)
87      robot->step(2000);
88
89      int width = camera->getWidth();
90      int height = camera->getHeight();
91
92      while(!stop){
93          sampleRun(client_socket, camera, width, height, m1_motor,
    m2_motor, m3_motor, m4_motor, robot, imu, gps, gyro);
94      }
95
96      #ifdef DEBUG_MODE
97      logfile << "Closing sockets and cleanup..." << endl;
98      logfile.close();
99      #endif
100
101     close(client_socket);
102     close(server_socket);
103
104     delete robot;
105     return 0;
106 }
107
108 void sampleRun(int client_socket, Camera *camera, int width, int
    height, Motor *m1, Motor *m2, Motor *m3, Motor *m4, Robot *robot,
    InertialUnit *imu, GPS *gps, Gyro *gyro) {
109
110     json sendme;
111     json received = receiveMessage(client_socket);
112
113     // Reading the command
```

```
114     string rd_command = received["command"];
115
116     #ifdef DEBUG_MODE
117     logfile << "Received command was " << rd_command << endl;
118     #endif
119
120     // Check which command was received
121
122     if(rd_command == "GET_INIT_DATA") {
123         sendme = getInitData(robot, gps);
124
125         #ifdef DEBUG_MODE
126         logfile << "Sending initial data..." << endl;
127         #endif
128     }
129     if(rd_command == "GET_IMAGE") {
130         sendme = getCameraImage(width, height, camera);
131
132         #ifdef DEBUG_MODE
133         logfile << "Sending an image..." << endl;
134         #endif
135     }
136     if(rd_command == "GET_SENSORS") {
137         sendme = getSensors(robot, imu, gps, gyro);
138
139         #ifdef DEBUG_MODE
140         logfile << "Sending sensor data..." << endl;
141         #endif
142     }
143     if(rd_command == "SET_VELOCITY") {
144         motor_power_t mp;
145         mp.m1 = received["m1"];
146         mp.m2 = received["m2"];
147         mp.m3 = received["m3"];
148         mp.m4 = received["m4"];
149         setMotors(m1, m2, m3, m4, mp, robot);
150         return;
151     }
152     if(rd_command == "GET_DATA_2") {
153         sendme = send_data_model_2(robot, gps);
154
155         #ifdef DEBUG_MODE
156         logfile << "Sending data for model 2" << endl;
157         #endif
158     }
159     if(rd_command == "GET_DATA_3") {
160         sendme = send_data_model_3(robot, gps, imu);
161
162         #ifdef DEBUG_MODE
```

```
163             logfile << "Sending data for model 3" << endl;
164             #endif
165         }
166         if(rd_command == "STOP") {
167             sendme = stop_simul(robot, gps);
168             stop = 1;
169 
170             #ifdef DEBUG_MODE
171             logfile << "Detected the necessity of stopping the simulation
        !" << endl;
172             #endif
173         }
174         if(rd_command == "ADVANCE_TIME") {
175             int time = received["time"];
176             robot->step(time);
177 
178             #ifdef DEBUG_MODE
179             logfile << "Simulation advanced by " << time << " ms" << endl
        ;
180             #endif
181             return;
182         }
183 
184         #ifdef DEBUG_MODE
185         // Saving the JSON in a file for debugging purposes
186         std::ofstream file("output.json");
187         file << setw(4) << sendme << endl;
188         #endif
189 
190         // Sending the produced result
191         sendMessage(client_socket, sendme);
192         return;
193     }
```

Bibliography

[1] Daniel Caballero-Martin, Jose Manuel Lopez-Guede, Julian Estevez, and Manuel GraĆ±a. «Artificial Intelligence Applied to Drone Control: A State of the Art». In: *Drones* 8.7 (2024). DOI: 10.3390/drones8070296. URL: https://www.mdpi.com/2504-446X/8/7/296 (cit. on p. 1).

[2] Simulink Documentation. *Simulation and Model-Based Design*. 2020. URL: https://www.mathworks.com/products/simulink.html (cit. on pp. 2, 26).

[3] Fabrice Bellard. «QEMU, a fast and portable dynamic translator». In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41 (cit. on pp. 2, 25).

[4] Antmicro. *Renode: a development framework for embedded systems*. https://renode.io/. Accessed: 2024-09-10. 2024 (cit. on pp. 2, 25).

[5] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Tech. rep. UCB/EECS-2011-62. May 2011. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html (cit. on pp. 2, 5).

[6] Olivier Michel. «WebotsTM: Professional Mobile Robot Simulation». In: *International Journal of Advanced Robotic Systems* 1 (Mar. 2004). DOI: 10.5772/5618 (cit. on pp. 2, 5, 31, 32).

[7] «IEEE Standard for Standard SystemC® Language Reference Manual». In: *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)* (2023), pp. 1–618. DOI: 10.1109/IEEESTD.2023.10246125 (cit. on pp. 3, 27).

[8] «IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual». In: *IEEE Std 1666.1-2016* (2016), pp. 1–236. DOI: 10.1109/IEEESTD.2016.7448795 (cit. on pp. 3, 28).

[9] Mohamed Amine Hamdi, Giovanni Pollo, Matteo Risso, Germain Haugou, Alessio Burrello, Enrico Macii, Massimo Poncino, Sara Vinco, and Daniele Jahier Pagliari. *Integrating SystemC-AMS Power Modeling with a RISC-V ISS for Virtual Prototyping of Battery-operated Embedded Devices*. Tech. rep. arXiv:2404.01861. 2024. URL: https://arxiv.org/abs/2404.01861 (cit. on pp. 3, 5, 29).

[10] Nazareno Bruschi, Germain Haugou, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. «GVSoC: A Highly Configurable, Fast and Accurate Full-Platform Simulator for RISC-V based IoT Processors». In: *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, Oct. 2021. DOI: 10.1109/iccd53106.2021.00071. URL: http://dx.doi.org/10.1109/ICCD53106.2021.00071 (cit. on pp. 3, 26).

[11] Bitcraze AB. *Crazyflie 2.1 Nano Quadcopter*. Accessed: 2024-08-30. 2017. URL: https://www.bitcraze.io/products/old-products/crazyflie-2-1/ (cit. on pp. 3, 5).

[12] Stuart Mitchell, Michael O'Sullivan, and Iain Dunning. «PuLP : A Linear Programming Toolkit for Python». In: 2011. URL: https://api.semanticscholar.org/CorpusID:14277904 (cit. on p. 7).

[13] PULP Platform. *PULP Platform GitHub Repository*. https://github.com/pulp-platform. Accessed: 2024-08-30. 2024 (cit. on p. 7).

[14] PULP Platform. *PULPino: A Small Single-core RISC-V SoC*. https://github.com/pulp-platform/pulpino. Accessed: 2024-08-30. 2024 (cit. on p. 7).

[15] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. «Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX». In: *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 2018, pp. 1–3. DOI: 10.1109/S3S.2018.8640145 (cit. on p. 7).

[16] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank Gurkaynak, and Luca Benini. *Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices*. Feb. 2017. DOI: 10.1109/TVLSI.2017.2654506. URL: https://ieeexplore.ieee.org/document/7864441 (cit. on p. 7).

[17] GreenWaves Technologies. *GAP8 MCU AI: Ultra-Low Power RISC-V IoT Application Processor*. https://greenwaves-technologies.com/gap8_mcu_ai/. Accessed: 2024-08-30. 2024 (cit. on p. 7).

[18] GreenWaves Technologies. *GAP8 Documentation*. https://github.com/GreenWaves-Technologies/gap8_docs/blob/master/home/header.md. Accessed: 2024-08-30. 2024 (cit. on p. 7).

[19] Daniel Garcia, Mehdi Ghommem, Nathan Collier, BON Varga, and Victor Calo. «PyFly: A fast, portable aerodynamics simulator». In: *Journal of Computational and Applied Mathematics* 344 (Mar. 2018). DOI: 10.1016/j.cam.2018.03.003 (cit. on p. 8).

[20] Aki Lehtinen and Jaakko Kuorikoski. «Computer Simulations in Economics». In: Dec. 2021, pp. 355–369. ISBN: 9781138824201 (cit. on p. 8).

[21] Rhys Goldstein and Azam Khan. «Simulation-Based Architectural Design». In: *Guide to Simulation-Based Disciplines: Advancing Our Computational Future*. Ed. by Saurabh Mittal, Umut Durak, and Tuncer Ören. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-61264-5_8. URL: https://doi.org/10.1007/978-3-319-61264-5_8 (cit. on p. 8).

[22] Leili Javidpour. «Computer Simulations of Protein Folding». In: *Computing in Science and Engineering* (Mar. 2012). DOI: 10.1109/MCSE.2012.21 (cit. on p. 8).

[23] R.A. Shafik, A. Das, S. Yang, G. Merrett, and B.M. Al-Hashimi. «9 - Design considerations for reliable embedded systems». In: *Reliability Characterisation of Electrical and Electronic Systems*. Ed. by Jonathan Swingler. Oxford: Woodhead Publishing, 2015, pp. 169–194. ISBN: 978-1-78242-221-1. DOI: https://doi.org/10.1016/B978-1-78242-221-1.00009-5. URL: https://www.sciencedirect.com/science/article/pii/B9781782422211000095 (cit. on p. 10).

[24] Majdi Richa, Jean-Christophe Prévotet, Mickaël Dardaillon, Mohamad Mroué, and Abed Ellatif Samhat. «High-level power estimation techniques in embedded systems hardware: an overview». In: *The Journal of Supercomputing* 79.4 (2023), pp. 3771–3790. ISSN: 1573-0484. DOI: 10.1007/s11227-022-04798-5. URL: https://doi.org/10.1007/s11227-022-04798-5 (cit. on p. 10).

[25] Michael Kerrisk. *UNIX(7) - Linux Programmer's Manual*. https://man7.org/linux/man-pages/man7/unix.7.html. Accessed: 2024-08-30. 2024 (cit. on p. 12).

[26] Asif Ali Laghari, Awais Khan Jumani, Rashid Ali Laghari, and Haque Nawaz. «Unmanned aerial vehicles: A review». In: *Cognitive Robotics* 3 (2023), pp. 8–22. ISSN: 2667-2413. DOI: https://doi.org/10.1016/j.cogr.2022.12.004. URL: https://www.sciencedirect.com/science/article/pii/S2667241322000258 (cit. on p. 15).

[27] Faiyaz Ahmed, J. C. Mohanta, Anupam Keshari, and Pankaj Singh Yadav. «Recent Advances in Unmanned Aerial Vehicles: A Review». In: *Arabian Journal for Science and Engineering* 47.7 (2022), pp. 7963–7984. ISSN: 2191-4281. DOI: 10.1007/s13369-022-06738-0. URL: https://doi.org/10.1007/s13369-022-06738-0 (cit. on p. 15).

[28] Khaled Telli, Okba Kraa, Yassine Himeur, Abdelmalik Ouamane, Mohamed Boumehraz, Shadi Atalla, and Wathiq Mansoor. «A Comprehensive Review of Recent Research Trends on Unmanned Aerial Vehicles (UAVs)». In: *Systems* 11.8 (2023). ISSN: 2079-8954. DOI: 10.3390/systems11080400. URL: https://www.mdpi.com/2079-8954/11/8/400 (cit. on p. 15).

[29] Adam Zourari, My abdelkader Youssefi, Youssef Youssef, Rachid Dakir, and Mohamed BAKIR. «Enhancing Autonomous Drone Navigation in Unfamiliar Environments with Predictive PID Control and Neural Network Integration». In: Sept. 2024, pp. 64–70. ISBN: 978-3-031-70991-3. DOI: 10.1007/978-3-031-70992-0_6 (cit. on p. 16).

[30] Gopi Gugan and Anwar Haque. «Path Planning for Autonomous Drones: Challenges and Future Directions». In: *Drones* 7.3 (2023). URL: https://www.mdpi.com/2504-446X/7/3/169 (cit. on p. 16).

[31] Isabel Pinheiro, André Aguiar, André Figueiredo, Tatiana Pinho, António Valente, and Filipe Santos. «Nano Aerial Vehicles for Tree Pollination». In: *Applied Sciences* 13.7 (2023). URL: https://www.mdpi.com/2076-3417/13/7/4265 (cit. on p. 16).

[32] Hoa Nguyen, Toan Quyen, Van-Cuong Nguyen, Anh Le, Hoa Tran, and Minh Nguyen. «Control Algorithms for UAVs: A Comprehensive Survey». In: *EAI Endorsed Transactions on Industrial Networks and Intelligent Systems* 7 (May 2020), p. 164586. DOI: 10.4108/eai.18-5-2020.164586 (cit. on pp. 17–19).

[33] Bashra Oleiwi and Mohamed Mohamed. «Optimal design of linear and non-linear PID controllers for speed control of an electric vehicle». In: *Journal of Intelligent Systems* 33 (Sept. 2024). DOI: 10.1515/jisys-2024-0028 (cit. on p. 18).

[34] Abdelhakim Idir, A. Zemmit, H. Akroum, Mokhtar Nesri, Guedida Sifelislam, and Laurent Canale. *Enhancing Temperature Control of Electric Furnaces Using a Modified Pid Controller Design Strategy*. Aug. 2024. DOI: 10.21203/rs.3.rs-4967918/v1 (cit. on p. 18).

[35] Tri Kuntoro Priyambodo, Oktaf Agni Dhewa, and Try Susanto. «Model of Linear Quadratic Regulator (LQR) Control System in Waypoint Flight Mission of Flying Wing UAV». In: *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)* 12.4 (Dec. 2020), pp. 43–49. URL: https://jtec.utem.edu.my/jtec/article/view/5696 (cit. on p. 18).

[36] RISC-V Software Source. *Spike RISC-V ISA Simulator*. https://github.com/riscv-software-src/riscv-isa-sim. Accessed: 2024-09-12. 2024 (cit. on p. 26).

[37] N. Koenig and A. Howard. «Design and use paradigms for Gazebo, an open-source multi-robot simulator». In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727 (cit. on p. 31).

[38] RoboDK. *RoboDK: Simulation and Offline Programming Software for Industrial Robots*. Accessed: 2024-09-12. 2024. URL: https://robodk.com/ (cit. on p. 31).

[39] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. «AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles». In: *Field and Service Robotics*. 2017. eprint: arXiv:1705.05065. URL: https://arxiv.org/abs/1705.05065 (cit. on p. 31).

[40] FlightGear. *FlightGear: Open Source Flight Simulator*. Accessed: 2024-09-12. 2024. URL: https://www.flightgear.org/ (cit. on p. 31).

[41] Aicha Hentati, Lobna Krichen, Fourati Mohamed, and Lamia Fourati. «Simulation Tools, Environments and Frameworks for UAV Systems Performance Analysis». In: June 2018, pp. 1495–1500. DOI: 10.1109/IWCMC.2018.8450505 (cit. on p. 31).

[42] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. «ROS: an open-source Robot Operating System». In: *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*. Kobe, Japan, May 2009 (cit. on p. 31).

[43] Juan Zhang, James Campbell, Donald Sweeney II, and Andrea Hupman. «Energy Consumption Models for Delivery Drones: A Comparison and Assessment». In: (May 2020) (cit. on p. 33).

[44] Pedram Beigi, Mohammad Sadra Rajabi, and Sina Aghakhani. «An Overview of Drone Energy Consumption Factors and Models». In: Aug. 2022, pp. 1–20. ISBN: 978-3-030-72322-4. DOI: 10.1007/978-3-030-72322-4_200-1 (cit. on p. 33).

[45] Raffaello D'Andrea. «Guest Editorial Can Drones Deliver?» In: *Automation Science and Engineering, IEEE Transactions on* 11 (July 2014), pp. 647–648. DOI: 10.1109/TASE.2014.2326952 (cit. on p. 34).

[46] Kevin Dorling, Jordan Heinrichs, Geoffrey Messier, and Sebastian Magierowski. «Vehicle Routing Problems for Drone Delivery». In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 47 (Aug. 2016), pp. 1–16. DOI: 10.1109/TSMC.2016.2582745 (cit. on p. 34).

[47] Yukai Chen, Donkyu Baek, Alberto Bocca, Alberto Macii, Enrico Macii, and Massimo Poncino. «A Case for a Battery-Aware Model of Drone Energy Consumption». In: Oct. 2018, pp. 1–8. DOI: 10.1109/INTLEC.2018.8612333 (cit. on p. 35).

[48] Hasini Abeywickrama, Beeshanga Jayawickrama, Ying He, and Eryk Dutkiewicz. «Comprehensive Energy Consumption Model for Unmanned Aerial Vehicles, Based on Empirical Studies of Battery Performance». In: *IEEE Access* PP (Oct. 2018), pp. 1–1. DOI: 10.1109/ACCESS.2018.2875040 (cit. on p. 35).

[49] Joshuah Stolaroff, Constantine Samaras, Emma O'Neill, Alia Lubers, Alexandra Mitchell, and Daniel Ceperley. «Energy use and life cycle greenhouse gas emissions of drones for commercial package delivery». In: *Nature Communications* 9 (Feb. 2018). DOI: 10.1038/s41467-017-02411-5 (cit. on p. 35).

[50] T. Kirschstein. «Comparison of energy demands of drone-based and ground-based parcel delivery services». In: *Transportation Research Part D: Transport and Environment* 78 (Jan. 2020), p. 102209. DOI: 10.1016/j.trd.2019.102209 (cit. on pp. 36, 101).

[51] David Meyer. «Crazyflie 2.0 Quadrocopter: System Identification and Controller Design». PhD thesis. ETH Zurich, 2015. DOI: 10.3929/ethz-b-000214143. URL: https://doi.org/10.3929/ethz-b-000214143 (cit. on p. 36).

[52] Mariusz Jacewicz, Marcin Żugaj, Robert Głębocki, and Przemysław Bibik. «Quadrotor Model for Energy Consumption Analysis». In: *Energies* 15.19 (2022). DOI: 10.3390/en15197136. URL: https://www.mdpi.com/1996-1073/15/19/7136 (cit. on pp. 36, 102).

[53] Nicolas Michel, Peng Wei, Zhaodan Kong, Anish Kumar Sinha, and Xinfan Lin. «Modeling and validation of electric multirotor unmanned aerial vehicle system energy dynamics». In: *eTransportation* 12 (2022), p. 100173. ISSN: 2590-1168. DOI: https://doi.org/10.1016/j.etran.2022.100173. URL: https://www.sciencedirect.com/science/article/pii/S2590116822000194 (cit. on pp. 36, 101, 102).

[54] Bitcraze. *PWM to Thrust Conversion for Crazyflie Firmware*. https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/pwm-to-thrust/. Accessed: 2024-08-28. 2024 (cit. on pp. 37, 86).

[55] Niels Lohmann. *JSON for Modern C++*. Version 3.11.3. Nov. 2023. URL: https://github.com/nlohmann (cit. on p. 40).

[56] Bitcraze. *Crazyflie Simulation - C-based Controllers*. https://github.com/bitcraze/crazyflie-simulation/tree/main/controllers_shared/c_based. Accessed: 2024-09-25. 2023 (cit. on p. 53).

[57] Bitcraze. *Crazyflie 2.x Platform Architecture*. https://www.bitcraze.io/documentation/system/platform/cf2-architecture/. Accessed: 2024-09-27. 2024. URL: https://www.bitcraze.io/documentation/system/platform/cf2-architecture/ (cit. on p. 73).

[58] Bitcraze. *Hardware documentation and design - Crazyflie 2.1 schematics*. https://github.com/bitcraze/hardware/blob/master/src/products/crazyflie-2_1/electronics/crazyflie_2.1_schematics_rev.b.pdf. Accessed: 2024-09-27. 2024. URL: https://github.com/bitcraze/hardware/blob/master/src/products/crazyflie-2_1/electronics/crazyflie_2.1_schematics_rev.b.pdf (cit. on p. 73).

[59] Bitcraze Forum User. *Power consumption and energy efficiency in Crazyflie*. Accessed: 2024-09-28. 2021. URL: https://forum.bitcraze.io/viewtopic.php?p=11134#p11134 (cit. on p. 79).

[60] Bitcraze Forum User. *Crazyflie Power Consumption Details*. Accessed: 2024-09-28. 2022. URL: https://forum.bitcraze.io/viewtopic.php?t=5146 (cit. on p. 79).

[61] LipoBattery.us. *High Rate Lithium-ion Battery 15C*. Accessed: 2024-09-28. n.d. URL: https://www.lipobattery.us/high-rate-lithium-ion-battery-15c-2/ (cit. on p. 81).

[62] PlotDigitizer. *PlotDigitizer*. Accessed: 2024-09-28. n.d. URL: https://plotdigitizer.com/app (cit. on p. 81).

[63] Giovanni Pollo. *Digipyze*. Accessed: 2024-09-30. 2024. URL: https://github.com/eml-eda/digipyze (cit. on pp. 81, 93).

[64] Ltd. Guangdong UFine New Energy Co. *UFX 402525 250mAh 3.7V Smallest Rechargeable Battery*. https://www.gdufinebattery.com/products/ufx-402525-250mah-37v-smallest-rechargeable-battery. Accessed: 2024-09-30. 2024 (cit. on p. 94).

[65] Atomic Workshop. *300mAh HD LiPo 60C Battery (High Power, Ultra-Compact)*. https://www.atomicworkshop.co.uk/300LiPoHDPUP.htm. Accessed: 2024-09-30. 2024 (cit. on p. 96).

[66] Bitcraze. *Crazyflie Simulation - Firmware Python Bindings*. https://www.bitcraze.io/documentation/repository/crazyflie-simulation/main/functional_areas/controllers/#firmware-python-bindings. Accessed: 2024-09-30. 2024 (cit. on p. 100).

[67] Hao Gong, Baoqi Huang, Bing Jia, and Hansu Dai. *Modelling Power Consumptions for Multi-rotor UAVs*. 2022. arXiv: 2209.04128 [cs.RO]. URL: https://arxiv.org/abs/2209.04128 (cit. on p. 102).

[68] C. Chan and Tai Yan Kam. «A procedure for power consumption estimation of multi-rotor unmanned aerial vehicle». In: *Journal of Physics: Conference Series* 1509 (Apr. 2020), p. 012015. DOI: 10.1088/1742-6596/1509/1/012015 (cit. on p. 102).

[69] Duc-Kien Phung and Pascal Morin. «Modeling and Energy Evaluation of Small Convertible UAVs*». In: *IFAC Proceedings Volumes* 46.30 (2013). 2nd IFAC Workshop on Research, Education and Development of Unmanned Aerial Systems, pp. 212–219. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20131120-3-FR-4045.00004. URL: https://www.sciencedirect.com/science/article/pii/S1474667015402964 (cit. on p. 102).