POLITECNICO DI TORINO

MASTER's Degree in COMPUTER ENGINEERING

Master's Degree Thesis

Development of Simulation Technologies for Assessing Multi-Rotor Unmanned Aerial Vehicle Performance in Precision Agriculture Operations

Supervisors

Prof. FABRIZIO DABBENE

Prof. GIORGIO GUGLIERI

Prof.ssa MARTINA MAMMARELLA

Prof. LORENZO COMBA

Candidate

RICCARDO LAZZARI

DECEMBER 2020

Abstract

The adoption of Unmanned Aerial Vehicles (UAVs) in agricultural scenarios can aspire to become a reality if the validation of their effectiveness will be sustained by the contemporary and shared improvement of all those technological gaps identified by current research projects. In particular, a commercial quadrotor-based spraying system that will have to fly automatically between vineyard rows would need to implement sophisticated flight control algorithms that could imply a great amount of computational workload just to keep attitude and position under control while following the flight path.

This kind of trajectories are often characterized by steep slopes, limited room for manoeuvring and sharp turns. At the same time, the control algorithm should be flexible enough to deal with rapid changes in the reference signal due to collision avoidance alerts, wind disturbances and spraying reaction effects. Moreover, the field of agriculture needs robust machines that can operate at a wide air temperature range, into various conditions of sunshine and with the necessity of easy use and maintenance. In addition, the cost of the development and the production of the system have to be kept as low as possible to comply with the market rules.

This Master's degree thesis aims at exploring the potentialities of the selected commercial off-the-shelf (COTS) autopilot board, the Pixhwak 4, for the implementation and testing of advanced control algorithms, in particular with Processor-In-the-Loop (PIL) trials with the automatic generated code running on the microcontroller while the modeling environment, developed in MATLAB/Simulink, is running on a generic computer.

In addition, a second objective of this work is to derive virtual 3D scenarios from laser, detection & ranging (LIDAR) point cloud maps realized by survey campaigns on the actual location selected for the flight testing operations: these scenarios have the twofold objective to be a useful tool for generating feasible trajectories and for visualizing the flight simulation of the UAV.

Acknowledgements

Vorrei porgere i miei più sinceri ringraziamenti al Prof. Fabrizio Dabbene per avermi concesso la sua fiducia, nonostante qualche mio tentennamento ed incoprensione iniziale, e per avermi dato la possibilità di svolgere questo lavoro così interessante e di respiro così ampiamente interdisciplinare; questa esperienza ha contribuito, ne sono certo, a sviluppare le mie capacità di adattamento e di "problem solving". Lo ringrazio, inoltre, per la flessibilità che mi ha garantito durante le varie fasi della tesi, che mi ha permesso di conciliarla agevolemente con la mia attività lavorativa.

Ringrazio il Prof. Giorgio Guglieri, con il quale condivido l'innata e profonda passione per le macchine volanti, per i suoi puntuali consigli, le sue rassicuranti e rapide risposte e per la sempre preziosa disponibilità, anche in questi tempi caratterizzati dalla pandemia mondiale.

Non posso, inoltre, non ringraziare il Prof. Lorenzo Comba per i suoi sempre precisi consigli ed idee sul da farsi e per aver condiviso con me dati e considerazioni, parte delle sue importanti ricerche.

Ringrazio, infine, la Prof.ssa Martina Mammarella, persona dotata di incredibile forza di volontà e spirito di sacrificio, per avermi fatto lavorare sempre al limite della mia "comfort zone", ma, al tempo stesso, per esser stata sempre disponibile per qualsiasi mia necessità e per chiarire ogni mio più piccolo dubbio o perplessità. Lavorare con Martina mi ha permesso di mettermi alla prova e di continuare a crescere, capendo di giorno in giorno che c'è sempre qualcosa da imparare o qualcosa per cui vale la pena essere curiosi.

Rivolgo, in conclusione, un pensiero commosso a tutti gli affetti della mia vita, vicini e lontani, che mi hanno accompagnato durante questi anni di lavoro e di studio e che hanno condiviso con me ogni sacrificio: a loro devo infinita gratitudine per la persona che sono oggi.

Riccardo

Table of Contents

List of Figures

Introduction

Technologies for UAV operations in precision agriculture

Farming is undergoing the so-called fourth agricultural revolution, thanks to the introduction of emerging technologies as robotics and artificial intelligence [1], aiming at improving the output and sustainability of plantations, quality of products and working conditions. This revolution came after the first agricultural revolution representing the transition from hunting and gathering to settled agriculture, the second with the British agricultural revolution in the 18th century and the third relating to post-war productivity increment associated with mechanization and Green Revolution in the developing world.

The progressive automation of agricultural processes has significantly improved the productivity of agriculture labour, shifting masses of workers into other productive industrial areas. Since then, scientific advances in chemistry, genetics, robotics and many other applied sciences have boosted an accelerated development of agricultural technology. Actually, in recent years, agricultural production has increased substantially [2].

In this context, the Food and Agriculture Organization of the United Nations (FAO) and the International Telecommunication Union (ITU) worked in cooperation with partners to address some of the challenges faced in agriculture, through the use of sustainable Information and Communication Technologies (ICTs) [3].

One of the latest development is represented by the increased use of Unmanned Aerial Vehicles (UAVs) in agriculture. These systems have a great potential to support and address some of the most pressing challenges in farming.

Nowadays, it is clear that aerial robotics will have its impact throughout the crop cycle: UAVs are already widely used for remote sensing missions and, in the

near future, more complex applications, such as pesticide spraying, will become standard fields of UAVs operations.

More specifically, UAVs are employed or will probably have an utilization in the following frameworks [4]:

- Soil and field analysis: at the beginning of the crop cycle, UAVs could provide precise 3-D maps via remote sensing for early soil analysis, useful in planning seed planting patterns. After planting, this kind of soil analysis would be able to provide data for irrigation and nitrogen-level management [5];

- Crop spraying: UAVs can scan the ground and spray the correct amount of liquid, modulating distance from the ground and the target, thanks to distance-measuring equipment, ultrasonic echoing and lasers such as those used in the laser, detection and ranging (LIDAR), methods that enable to adjust the UAV altitude as the topography and geography vary. In this way, an increased efficiency with a reduction of the amount of chemicals penetrating into groundwater could be achieved [6] [7];

- Crop monitoring: vast fields and low efficiency in crop monitoring are composing together the largest obstacle in farming. Monitoring challenges are aggravated by increasingly unpredictable weather conditions, which drive risk and field maintenance costs. Previously, satellite imagery offered the most advanced form of monitoring but images had to be ordered in advance, could be taken only with low frequency and could be not so precise. Further, services were extremely costly and and the quality could be poor in some days due to the weather conditions. Today, time-series animations taken by UAVs with specific on-board equipment can show the precise development of a crop and reveal production inefficiencies [8] [9];

- Irrigation: On-board hyper-spectral, multi-spectral or thermal sensors can identify which parts of a field are too dry and where the water supply needs to be improved. Additionally, once the crop is growing, these technologies allow the calculation of various irrigation indexes, which describes the relative density of the crop, and show the heat signature, the amount of energy the crop emits [10];

- Health assessment: it is essential to assess crop health and spot bacterial or fungal infections on plantations. By scanning a crop using both visible and near-infrared light, UAV-carried devices can identify which plants reflect different amounts of green light and near infra-red (NIR) light. This information can produce multi-spectral images that track changes in plants and indicate their health [11].

Particularly interesting is the fact that many advanced solutions for spraying and monitoring applications (see e.g., the work presented in [6]) have been proposed but, most of them, are designed to operate in specific and limited scenarios, such as flat terrains covered by crops with homogeneous canopies or where operations are performed from the top of plantations. An example of this kind of operations is shown in Figure 1.

Figure 1: Quadrotor flying above the vine rows for monitoring operations.

These scenarios represent ideal situations: for sure, existing UAV technology can be deepened for bringing advantages also in more challenging and specific situations because, in general, at the state of the art, the level of engagement of the available solutions is still far from being completely representative of the real potential of aerial robotics.

From this point of view, vineyards represent intriguing scenarios where all the potentialities of unmanned aerial platforms could be exploited: in fact, grapes plantations are often placed on sloped and in perched areas where UAVs could guarantee clear advantages with respect to traditional methods, in terms of higher efficiency in operations, reduced environmental impact and enhanced human health and safety [12].

Specifically, in these scenarios, UAVs could exploit their manoeuvrability and flexibility: in fact, they do not suffer problems connected with low traction soil like ground vehicles and they do need a dedicated way to access the field. In this way, UAVs could reach the most perched places and so all the space could be adequately and uniformly exploited to the fullest.

This thesis is inserted in an academic research project called “New technical and operative solutions for the use of drones in Agriculture 4.0” (Progetto di Ricerca

di Interesse Nazionale (PRIN) 2017, Prot. 2017S559BB) where a multi-phase approach is proposed with various types of robotic platforms that are involved into the implementation of innovative solutions for automated navigation and in-field operations within a complex irregular and unstructured scenario, such as vineyards on sloped terrains [13].

For its flying characteristics, a multi-rotor UAV is the ideal choice for flying between vineyard rows and the reason why this platform has been chosen for the project. In fact, due to its light-weight configuration and inherent instability, a multi-rotor has a good flight maneuverability and the capability to perform vertical take-off and landing (VTOL) and hovering in mid-flight: these characteristics could be very useful for bio-pesticide distribution or pruning operations.

Within the PRIN research project, the aim is to exploit 3D point cloud maps of vineyards, collected by survey campaigns carried out with a fixed-wing UAV, for the aims explored in [14], in [15] and [16]. In particular, these maps will be processed to obtain their simplified version to be uploaded on board of the drones for real-time navigation within the vineyard rows, without losing canopy geometry. In fact, the limited computation resources on vehicles and the complexity of the scenario discourages on-line simultaneous localization and mapping procedure.

In this thesis, the same 3D point cloud maps have been used for generating 3D virtual scenarios. In their turn, these scenarios have been used to generate trajectories. The point cloud maps have been processed with a series of software tools to obtain a virtual product to be loaded into the selected simulator. The generation of these 3D scenarios is described in Chapter 3.

On the other hand, the main objective of this thesis has been the exploration of the potentialities of the selected commercial off-the-shelf (COTS) autopilot board, the Pixhwak 4, for the implementation and testing of advanced control algorithms for controlling multi-rotors in precision agriculture operations, exploiting a Model-Based Design approach, and test them on a trajectory generated from 3D virtual scenario.

During these kinds of operations, to ensure a safe flight between vineyard rows, it is mandatory to provide optimal and efficient guidance, navigation and control (GNC) capabilities to the UAVs. To this end, flyable GNC schemes shall be implemented on board of the UAV autopilot to guarantee high efficiency and manoeuvrability.

The literature on control design for multi-rotors is vast and is mainly based on proportional-integrative-derivative (PID) controllers, thanks to their reliability and ease of implementation. Examples that exploit this technology are available in [17], [18], [19] and many others.

Even if these kinds of controllers are versatile and have a low-cost development process, the trajectory between vineyard row is often characterized by limited room for manoeuvring and sharp turns. Even if these features are manageable by

well-tuned PID controllers (an example of PID tuning for this purpose is shown in Chapter 4), they are not flexible enough to deal with rapid changes in the reference signal due to collision avoidance alerts, wind disturbances and spraying reaction effects. In other words, the design of the control strategy has to guarantee the fulfillment of mission, system and safety requirements despite the presence of external and internal disturbances.

For this reason, advanced controllers are gaining attraction for their employment on board of UAVs in precision agriculture operations: Model Predictive Control (MPC) [20], Linear Quadratic Regulator (LQR) [21], Sliding Mode Control (SMC) [22] are only few examples of advanced control algorithms that, nowadays, are a subject of study in academic research for the aforesaid field of application.

These techniques imply a great amount of computational workload just to keep attitude and position under control while following the flight path: understanding if the Pixhawk 4 autopilot board could handle these complex control algorithms, exploiting a familiar development environment, such as MATLAB/Simulink, has been the main driving motivation of this work. In fact, assessing this possibility on a so widespread autopilot could have a big impact, in terms of cost and time spent for the design, on the final platform development.

During the course of the work, a non-negligible focus has been put on the automatic code generation: in fact, this feature of the Model-Based Design paradigm is crucial to pull down the development time, as stated in [23]. Because of this, the LQR controller, chosen for being implemented on the selected board, has been cross-compiled and run using the Mathworks Embedded Coder Support Package for PX4 autopilots. Then, Processor-In-the-Loop (PIL) trials have been executed tracking the trajectory generated thanks to a simple scenario, derived from actual 3D point cloud maps of the location chosen for flight testing.

Details about the selected hardware and the Model-Based Design process used in this project can be found in Chapter 1 and Chapter 2; the results achieved in trajectory generation, visualization and simulation of the quad-rotor flight between grapevine rows are reported in Chapter 4. Chapter 5, on the other hand, contains some reflections on the work carried out and future developments.

Chapter 1

System Selection and Description

1.1 Requirements and selection

The initial requirements for this project depended essentially on the final application: in fact, a precision agriculture multi-rotor that has to fly automatically between vineyard rows. This needs to implement sophisticated flight control algorithms that imply a great amount of computational workload just to keep attitude and position under control while following the flight path.

This trajectory is often characterized by steep slopes, limited room for manoeuvring and sharp turns. At the same time, the control algorithm has to be flexible enough to deal with rapid changes in the reference signal due to collision avoidance alerts, wind disturbances and eventual reaction effects of spraying or other activities. Moreover, the agricultural environment needs robust machines that can operate at a wide air temperature range, into various conditions of sunshine and with the necessity of simple maintenance. In addition, the cost of the development and the production of the system have to be kept as low as possible to comply with the market rules.

Taking into account all of these considerations, the system and the development environment selection was essentially lead by the following prerequisites:

- use of a commercial off-the-shelf (COTS) autopilot hardware with a reasonable computational capability;

- software stack capability to adapt to diverse airframe (in fact, a great advantage would be using the same autopilot not only for a multi-rotor vehicle but for all the robotic platforms employed in the same scenario);

- use of Model-Based Design with automatic code generation for the development process to expedite the prototyping phase.

After an evaluation of various autopilot models (a good summary of the available options can be found in [24]), the Pixhawk 4 was chosen for the project; in addition, to comply with the third prerequisite, the Mathworks Embedded Coder Support Package for PX4 autopilots (now integrated in the UAV toolbox from the 2020b MATLAB release) was chosen, to exploit also the background already held with use of the MATLAB/Simulink suite, against other possible development suites and software stacks, such as ArduPilot [25].

The decision was made also analyzing the interesting results achieved in [17], [22] and [26]. In particular, [17] uses the Pixhawk Pilot Support Package [27] for implementing customized flight control algorithms over a quadrotor platform, giving a good starting point for the purposes of this thesis. More than that [22] uses the same approach for implementing a Sliding Mode Control (SMC) algorithm, giving sufficient assurance that the Pixhawk autopilot board could handle computational complexity of advanced control algorithms.

In conclusion, [26] is an example of a Pixhawk autopilot board interfaced with a spraying system for plant protection, one of the final field of application of the entire project.

1.2 Autopilot hardware and software

1.2.1 Pixhawk 4 autopilot board

Pixhawk is an independent open-hardware project providing low-cost autopilot hardware designs to the academic, hobby and industrial communities.

Its history began in 2008 in ETH Zurich as master thesis project and, nowadays, it is a widespread open-source full-scale solution, reusable and standardized, together with the PX4 flight stack. Pixhawk has become a functional alternative for professional and economic implementation of flight control algorithms [28].

This autopilot board has reached a level in which coding an advanced task does not require to know in depth how to design an autopilot itself, but, for implementing automatic control solutions, an average knowledge of control theory and high-level programming (such as, for example, C++, Java, Python programming languages) is sufficient [25].

The Pixhawk project creates open hardware designs in the form of schematics, which define a set of components (CPU, sensors, etc.) and their connections/pin mappings. Latest standard can be found in [30]. These schematics and reference designs are licensed under CC BY-SA 3: this allows to use, sell, share, modify and

Figure 1.1: Pixhawk 4 autopilot board [29].

build on the files in almost any way providing credit/attribution and sharing any changes made under the same open source license.

Each design is named using the designation FMUvX (Flight Management Unit Version X): higher FMU numbers indicate that the board is more recent, but may not indicate increased capabilities. The FMUv5 is the latest version produced and the one used in the Pixhawk 4: the main feature, already introduced with the v4-PRO version, is the integration of the I/O processor on the same board of the main CPU; it has also more RAM compared to the previous versions.

Boards based on the same design are "binary compatible": it means that they can run the same firmware, the software tied to the board contained in non-volatile memory [31].

The Pixhawk 4 is the latest update to the family of Pixhawk flight controllers. It is optimized to run the full Dronecode stack and the latest PX4 firmware is pre-installed. It comprises advanced processor technology from STMicroelectronics, sensor technology from Bosch and InvenSense and a NuttX real-time operating system, delivering good performance, flexibility and reliability for controlling autonomous vehicles.

Figure 1.1 represents the Pixhawk 4 board. All the ports needed for connecting the board to the actual drone, batteries and sensors are visible on the front face:

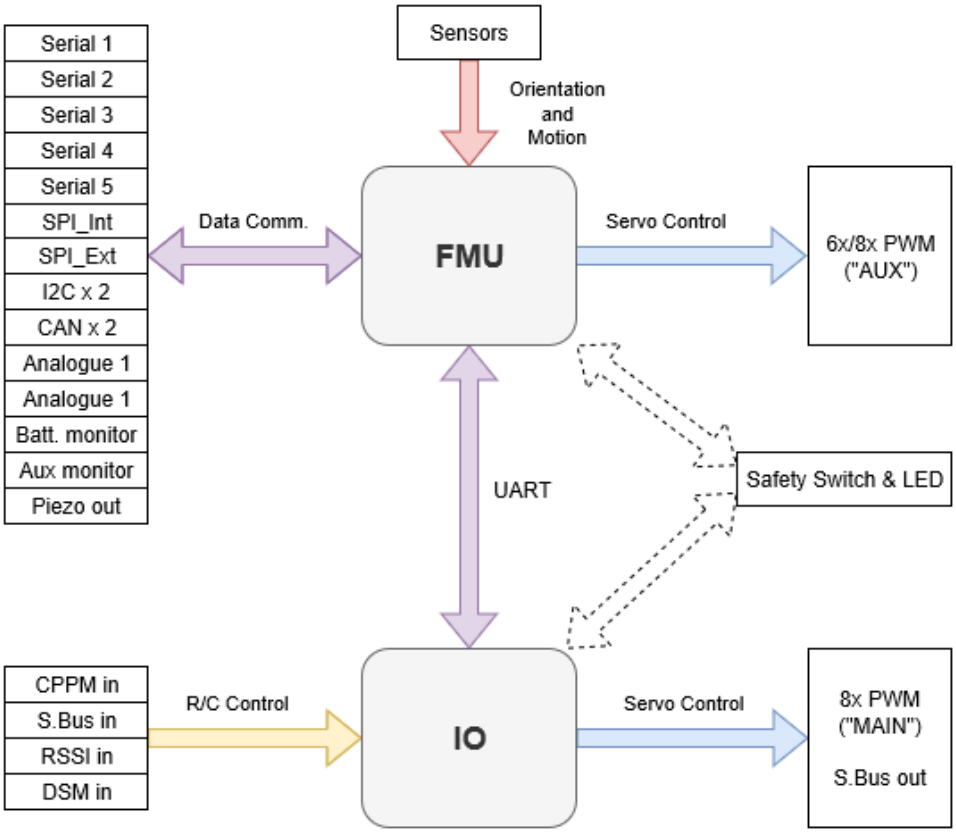

Figure 1.2: Pixhawk FMU and I/O responsibilities [29].

in particular, there are two power supply ports for redundancy, ports used for telemetry, Controller Area Network (CAN) bus port and Inter-Integrated Circuit (I2C) bus ports for connecting additional sensors. In particular, two I2C bus ports are grouped with serial ports for GPS/compass modules. At the bottom of the board, an SD card slot can be used to store startup scripts and log flight data. On the side of the board, the micro Universal Serial Bus (USB) 2.0 port is present: that is the interface for connecting the board to the development computer, deeply involved in Processor-in-the loop (PIL) simulation, described in Section 2.5.

The USB 2.0 ("High-Speed" USB) has a 125 microseconds time base called a "microframe": a "microframe" can contain several transactions and their maximum number may vary with specific system implementation details. In addition, also transaction data payloads can be modified on the basis of application requirements. More information about USB 2.0 can be found in [32], while specific details about NuttX implementation for Pixhawk 4 autopilot board can be found in the next Section.

As already mentioned, Pixhawk 4 autopilot is equipped with two CPU: the main FMU processor (a STM32F765 32 Bit Arm Cortex-M7, 216MHz, 2MB flash memory, 512KB RAM) is the one that runs most of the applications and collect sensors data while the IO processor (a STM32F100 32 Bit 32 Bit Arm Cortex-M3, 24MHz, 8KB SRAM) is mainly delegated to take over the main FMU in case of failure via a dedicated safety switch that passes control to the manual radio-command. The same switch can be used for taking manual control of the vehicle intentionally.

This can be easily seen in Figure 1.2, where the upstream data connections to both processors and the downstream servo control connections to Pulse Width Modulation (PWM) signals, directed to the motors, are represented [33].

Specifically, main FMU Processor has a clock period of approximately 4.62 ns (derived from 216 MHz), 2 MB of flash memory and 512KB of RAM: taking just these specifications into consideration, it is possible to affirm in broad terms that this autopilot board has enough computational resources to support customized algorithms and models with a moderate level of complexity.

1.2.2 NuttX operating system

NuttX is a real-time operating system (RTOS) with an emphasis on UNIX standards compliance and small memory footprint. Its goal is to provide most standard operating system interfaces to support a rich, multi-threaded development environment for deeply embedded processors. Figure 1.3 shows how the NuttX operating system interfaces with other components of the autopilot board taken into consideration in this work: the architecture is very similar to a personal computer where loaded and running applications interact with the hardware (processors and sensors) via standard system calls, provided by the RTOS. The approach selected by NuttX is intended to support greater scalability from the very tiny to moderate embedded platform.

NuttX documentation extensively underlines the importance of the compliance to the standards: users can think about NuttX like a tiny work-alike Linux OS with a much reduced feature set but supporting also, for example, the Executable and Linkable Format (ELF), the standard Linux binary format for compiling customized applications.

NuttX offers a well-developed multithread capability (tasks and threads try to emulate standard Unix processes and threads) and it is "fully preemptible": a task or a thread can be interrupted by the operating system at any time to achieve a strict priority scheduling.

Regarding the I/O, NuttX has its own implementation for managing ports: specifically, as of interest for this work, the implementation regarding the USB serial port is contained in `stm32_otgfshost.c` source code file.

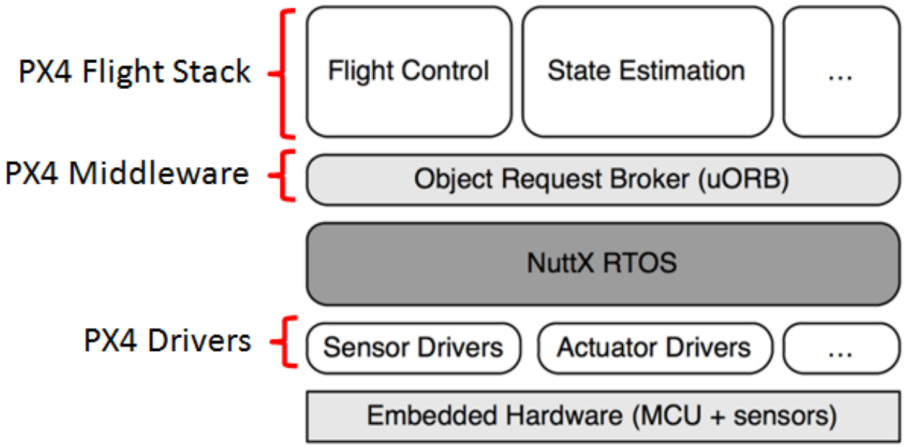

Figure 1.3: Pixhawk software layers [17].

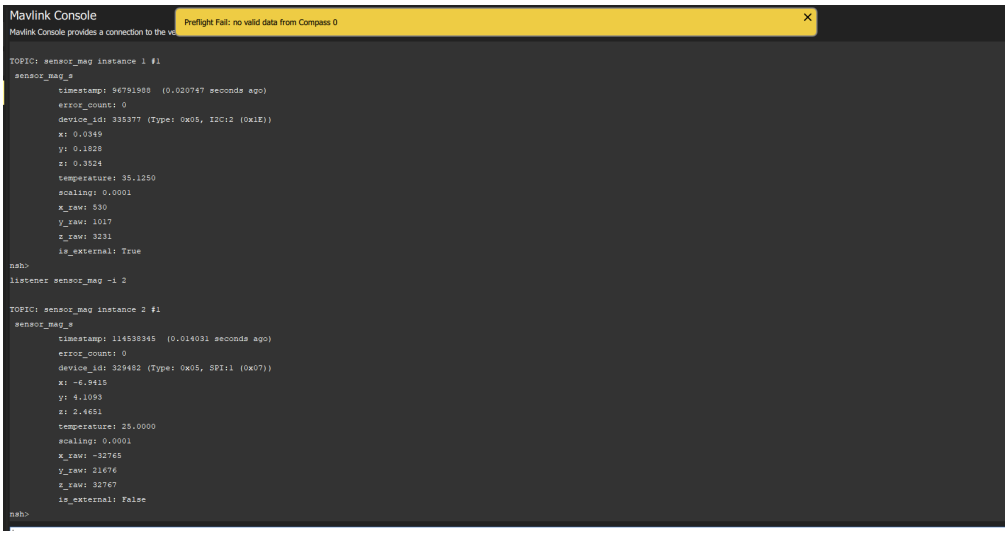

Figure 1.4: NuttShell console view opened from QGroundStation and connected to Pixhawk 4 autopilot board.

It is interesting to observe that the `STM32_MAX_PACKET_SIZE` variable, indicating the maximum size of a USB transaction payload, has a default setting of 64 bytes: referring to tables presented in the USB documentation [32], it means that every USB micro-frame (that can be composed on its turn by many transactions, see Section 1.2.1) can contain a maximum of 4672 bytes of useful data (approximately equivalent to a one-way transfer data rate of 37 MB/s).

NuttX is also provided by a lightweight, bash-like shell with a rich feature

set for basic user interaction called NuttShell (NSH). It supports a rich set of included commands, scripting and the ability to run applications as "built-in". NSH is implemented as an application, part of a library called `nshlib`. Like other components, NSH is completely optional and can be disabled: in this case, at startup, NuttX directly loads a given task instead of the main NSH application (this is the case of `px4_simulink_app` module, see Section 1.3 for details).

When NuttX is installed on a Pixhawk board, NuttShell can be accessed via serial connection setting the correct baud rate (57600) and using a terminal emulator (for example PuTTY, TerraTerm or the NSH console provided with many ground station software such as QGroundControl, as depicted in Figure 1.4). Some useful commands, relating to the PX4 environment, can be found in [27].

All these features make NuttX an useful interface and knowing how it works is a good starting point to understand the environment where an autopilot software stack functions.

For more information regarding NuttX, refer to [34].

1.2.3 uORB middleware

An autopilot system is a multi-task/multi-thread environment where the applications are divided in modules that have to coordinate many operations with each other. For this reason, in this kind of environment, it is necessary an inter-task/inter-thread communication mean to achieve the required synchronization.

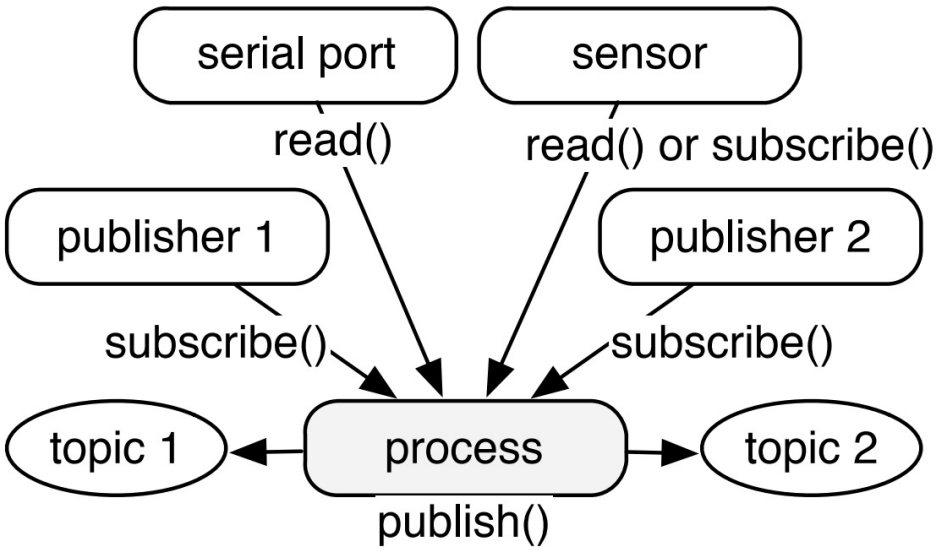

Figure 1.5: uORB publish/subscribe asynchronous messaging API [35].

In 2015, the micro Object Request Broker (uORB) has been developed by the same team who developed Pixhawk design (the original reference is [35]), exploiting the multithreading capabability of the NuttX operating System, described in the previous Section.

```
update: 1s, num topics: 77
TOPIC NAME                          INST #SUB #MSG #LOST #QSIZE
actuator_armed                         0    6    4     0 1
actuator_controls_0                    0    7  242  1044 1
battery_status                         0    6  500  2694 1
commander_state                        0    1   98    89 1
control_state                          0    4  242   433 1
ekf2_innovations                       0    1  242   223 1
ekf2_timestamps                        0    1  242    23 1
estimator_status                       0    3  242   488 1
mc_att_ctrl_status                     0    0  242     0 1
sensor_accel                           0    1  242     0 1
sensor_accel                           1    1  249    43 1
sensor_baro                            0    1   42     0 1
sensor_combined                        0    6  242   636 1
```

Figure 1.6: Published topics on a PX4 autopilot. From left to right the columns represent topic name, multi-instance index, number of subscribers, publishing frequency in Hz, number of lost messages per second (for all subscribers combined), and queue size [29].

The uORB is an asynchronous messaging Application Programming Interface (API) based on shared memory: the whole middleware runs in a single address space; in this way, memory is shared between all modules.

uORB follows the one-to-many publish-subscribe design pattern: all the partecipants to a communication are called "nodes" and are divided into "publishers" (senders) and "subscribers" (receivers). A publisher willing to share information advertises a communication channel called "topic" where it updates the data at its own frequency.

A subscriber can subscribe to a topic and, after the subscription is established,

it can ask for new data at its own pace (the "polling" action) or be woken from the thread sleep state when the new data is available.

A process can be both publisher and subscriber at the same time and it can subscribe and publish to multiple topics. This process and the relative functions to connect sensors and ports to applications are represented in Figure 1.5.

In Figure 1.6, it is possible to see the output of the command `uorb top` given to NuttShell, running on a PX4 autopilot: it can be noticed that most of the sensor topics have a frequency of 242 Hz, that means that a data is updated every 4.13 ms. If this frequency is too high, there is the possibility to limit the rate with which subscribers receive updates.

The uORB framework, combined with the task priority setup of the operating system, gives the possibility to achieve a synchronization between nodes and low-priority and high-priority tasks can be mixed.

In addition, uORB provides a mechanism to publish multiple independent instances of the same topic: this is useful, for example, if the system has several sensors of the same type.

It can be noted that using the uORB middleware, senders do not know any information about the receivers and vice versa: in this way, the system topology is unknown from the point of view of each module.

In conclusion, it is interesting to underline that data publication and subscriber copy are atomic operations (achieved by a read-write lock) to guarantee the consistency of the data. In addition, when a data is transmitted, the previous value is replaced and all the subscribers can only receive the last written value in the topic [29].

1.2.4 PX4 flight control software stack

PX4 is an advanced autopilot software stack, developed by a great variety of contributors since 2011 [28]. At the time of this work, it reached the 1.11 stable release.

PX4 is designed for UAVs, with a great focus on multirotors, but it became enough versatile and modular to be used with many kinds of robotic platforms (the codebase is the same for any type of vehicle). In [36], for example, PX4 is used to control the on-the-ground path of a rover, while, a very detailed study about the design of Autonomous Surface Vehicle (ASV) controlled over water by a PX4 autopilot, is presented in [37].

PX4 flight stack is an estimation and flight control system and it leverages on the uORB middleware that provides internal communications between modules and hardware integration via dedicated drivers. For external communication, PX4 uses a lightweight messaging protocol called MAVLink, specifically designed for the drone ecosystem. This protocol gives the possibility to communicate with ground

stations and to integrate the flight controller board with other components, such as companion computers, enabled cameras, proximity sensors, spraying devices etc.. When running over NuttX operating system, MAVLink can be used to connect the board with a terminal emulator for using NuttShell.

The complete system design is "reactive": it means that all functionality is divided into exchangeable and reusable components and communication is done by asynchronous message passing between self-contained modules/programs (uORB nodes that use topics to share data; for details, refer to Section 1.2.3).

The PX4 architecture allows every module to be rapidly and easily replaced, even at runtime: this feature is very important because it gives the possibility to modify the flight stack with the MathWorks Embedded Coder Support Package for PX4 autopilots, the development environment described in Section 1.3 and used in Chapter 2.

The flight stack is a collection of guidance, navigation and control algorithms. It includes estimators for attitude and position, controllers for any kind of airframe and mixers to traslate outputs into individual motor commands. All of them are included in the the Estimation and Control Library (ECL).

For estimation, the Extended Kalman Filter (EKF) algorithm takes one or more sensor inputs, combines them, and computes a vehicle state (for example, the attitude from Inertial Measurement Unit (IMU) sensor data). The EKF has different modes of operation for different combinations of sensor measurements. On start-up, the filter checks for a minimum viable combination of sensors and after the tilt, yaw and height alignment is completed, it enters a mode that provides rotation, vertical velocity, vertical position, IMU delta angle bias and IMU delta velocity bias estimates. This mode requires IMU data, a source of yaw (magnetometer or external vision) and a source of height data. This minimum data set is required for all EKF modes of operation. Then, other sensor data can then be used to estimate additional states.

A controller is a component that takes a setpoint and a measurement or estimated state (task/thread variable) as input. Its goal is to adjust the value of the process variable such that it matches the setpoint. The output is a correction to eventually reach that setpoint. For example, the position controller takes position setpoints as inputs, the process variable is the currently estimated position and the output is an attitude and thrust setpoint that move the UAV towards the desired position. Default controllers implemented into the flight stack are a mix of Proportional (P), Proportional-Integrative (PI) and Proportional-Integrative-Derivative (PID) controllers. These kinds of controllers are common control feedback mechanisms broadly used: they are simple and can be adapted to various systems but, for position control of extremely dynamic system, robust solutions cannot be obtained with PID controllers (for more details, refer to Section 2.3.1). In this direction, one of the practical objective of this thesis has been replacing the default PX4 attitude

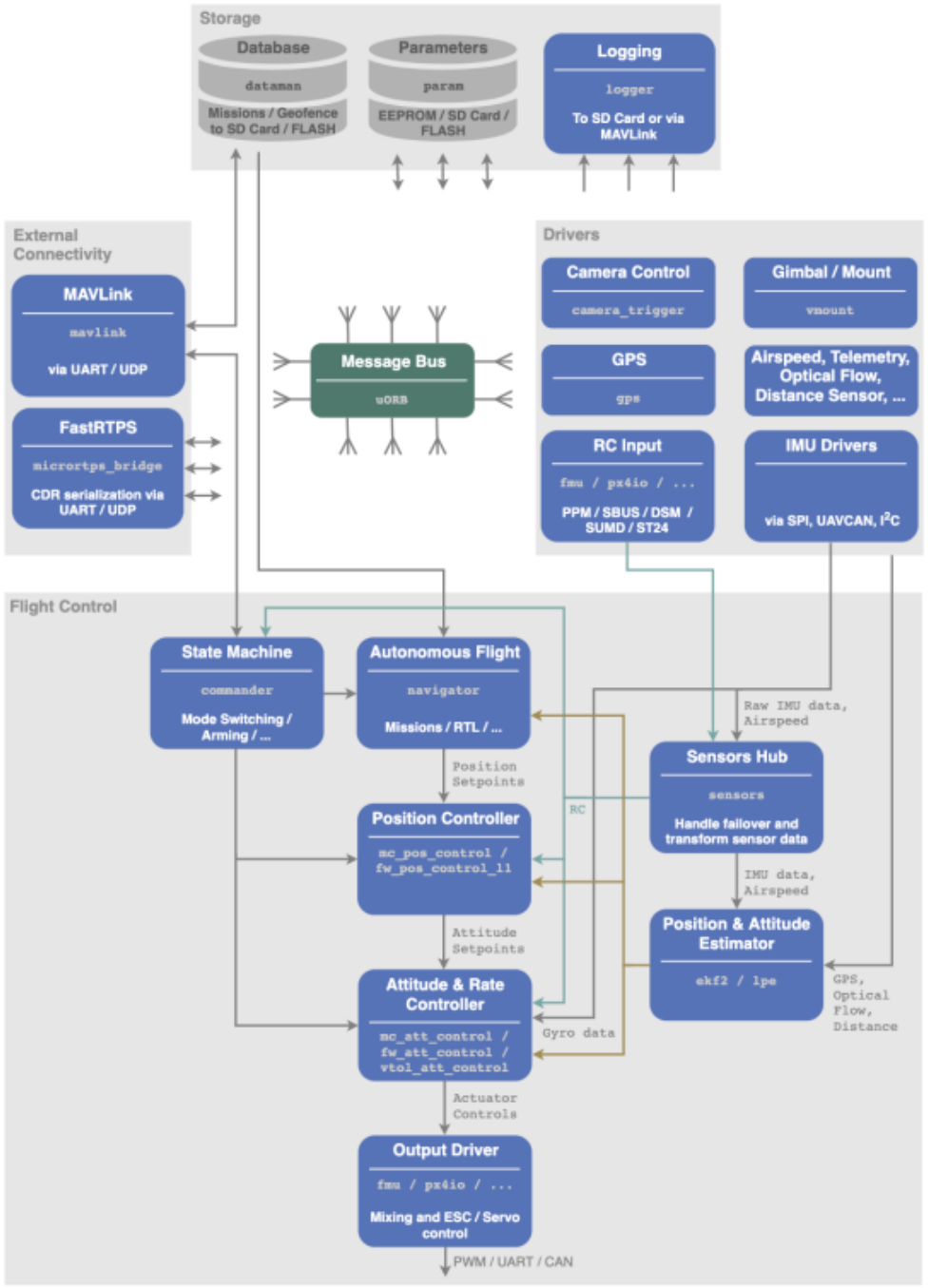

Figure 1.7: PX4 flight stack architecture [29].

and position controllers with a Linear Quadratic Regulator (LQR) controller.

A mixer takes force commands and translates them into individual motor commands, while ensuring that some limits are not exceeded. This translation is specific for a vehicle type and depends on various factors, such as the motor arrangements with respect to the center of gravity or the rotational inertia of the UAV. For example, pitching forward, for a multi-rotor, requires changing the speed of motors, while, for a plane, it implies to move the elevators or the elevons. Separating the mixer logic from the actual attitude controller greatly improves the software stack capability to be employed on every kind of robotic platforms.

It is important to remark that each sensor driver, estimator, controller or mixer is implemented as a module that, when running as a task, sends or receives data publishing or subscribing a topic, exploiting the functionalities of the uORB middleware, described in Section 1.2.3. Figure 1.8 shows an overview of the blocks of the flight stack pipeline: it contains the full outline, from sensors, manual input (RC) and autonomous flight control (Navigator-Position Controller-Attitude & Rate Controller), down to the motor control (Actuators).

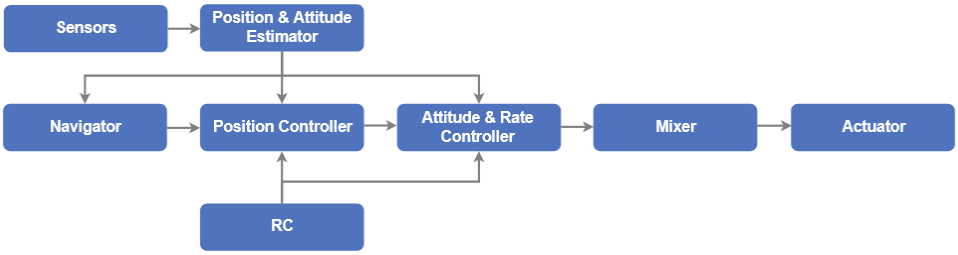

Figure 1.8: PX4 flight stack pipeline [29].

1.3 Development environment

1.3.1 MathWorks Embedded Coder Support Package for PX4 Autopilots

In academic research, MATLAB/Simulink environment is frequently used as a tool for system modeling and control design. In particular, MATLAB/Simulink is the standard tool for exploiting model-based design approach that consists in the development of embedded software, starting from block models. This approach applied to UAVs development cycle is deeply described in Section 2.1.

The most interesting steps in UAV development cycle for the purposes of this thesis are the Software-in-the-loop (SIL) and Processor-in-the-loop (PIL) simulations: during these phases, the production code dedicated to the aerial

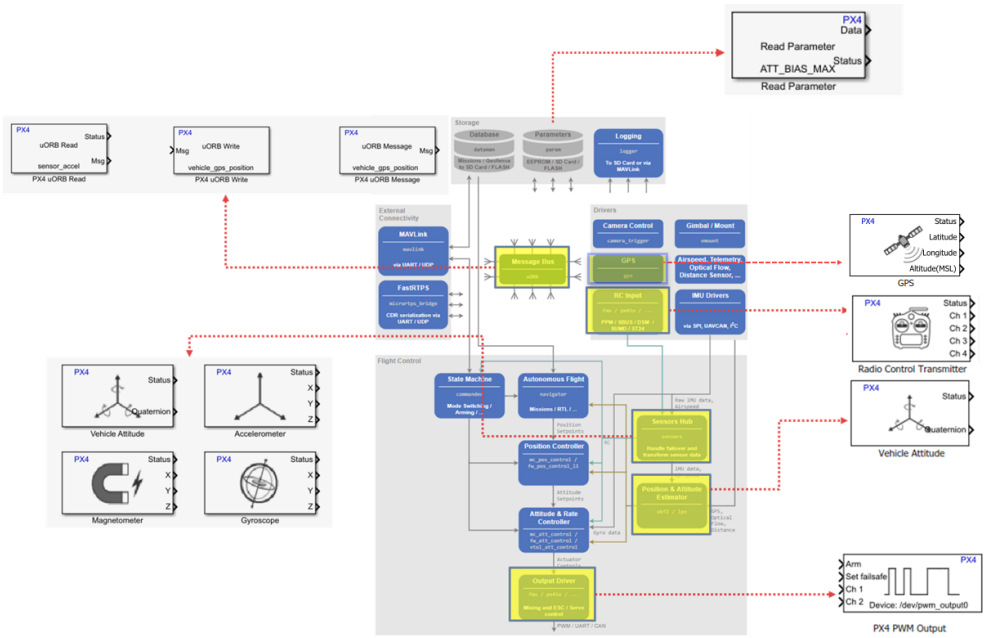

Figure 1.9: Simulink blocks that interface with PX4 modules [38].

system control and derived from the model is tested on an emulated environment (SIL) or on the actual autopilot board (PIL), to check its robustness and to evaluate performances and potential optimizations, before proceeding to real flight tests (more details and applications are given in Sections 2.4 and 2.5).

To exploit this framework, in the past, there has been a great effort to give the possibility to automatically translate algorithms developed in MATLAB/Simulink on the Pixhawk autopilot series: the original approach can be found in [39].

Nowadays, due to the advances in automated embedded coding achieved by MathWorks, PX4 development is able to support system models and control algorithms, designed with a Model-Based Design approach, without the need for the developers to be proficient in low-level programming. In concrete terms, one of the practical objective of this thesis has been exploring the potentiality of the means made available by the Embedded Coder Support Package for PX4 autopilots for implementing the quad-rotor model and controller design directly on the Pixhawk 4 with automatic code generation.

The Embedded Coder Support Package for PX4 autopilots has been available since 2018b MATLAB/Simlulink release. The package is directly derived from the Simulink Pilot Support Package [27], used for the studies [17] and [22] taken as a reference for this work, that in its turn, was derived from [39].

This development environment enables to access autopilot peripherals from MATLAB/Simulink environment and generate C++ code using the PX4 software

stack, building and deploying algorithms while incorporating on-board sensor data. Interfaces for the PX4 architecture components are provided by Simulink blocks that work as inputs and outputs for the model [38].

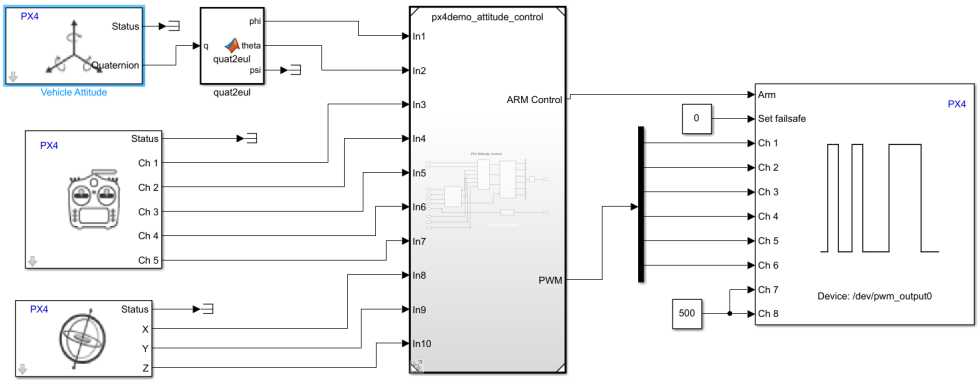

Figure 1.10: Example of a simple attitude control built with PX4 Simulink blocks.

Using these capabilities, position controller and attitude rate controller modules of the general PX4 architecture are replaced with user-defined algorithms: this is possible thanks to a custom startup script, which needs to be copied on the micro-SD card mounted on the Pixhawk (refer to Section 1.2.1 for slot details). This script, launched just after NuttX bootstrap, disables the default Navigator and Commander PX4 modules, substituting them with a module, called `px4_simulink_app`, that acts as a "wrapper" for the generated code.

The Embedded Coder [40], leveraging on CMake builder [41], generate and cross-compile the code from the models developed in Simulink using blocks (more details about automatic code generation can be found in Section 2.5). This code is then run by the module `px4_simulink_app` inside the PX4 software stack.

In Figure 1.9, some of the PX4 Simulink blocks are represented: in general, they give the possibility to subscribe or publish uORB topics to retrieve sensors read or to impose a control output (for more details about uORB middleware, refer to 1.2.3). This allows to build a model referencing directly to peripherals, sensors, commands of the autopilot board.

For example, the Vehicle Attitude block reads the `vehicle_odometry` uORB topic and outputs the attitude measurements from the Pixhawk hardware. With its own frequency, the block representing the software module checks if a new message is available on the `vehicle_odometry` topic. The block outputs the vehicle attitude in roll, pitch and yaw angles (refer to Section 2.2, for reference systems and model details).

These information are computed into an attitude control system that, for following a reference signal, emits control outputs that, throught a mixer matrix,

are delivered to a Pulse Width Modulation (PWM) block (for more details about PWM, refer to [33]). Attitude control system and mixer matrix need to be selected, designed and tuned according to the particular airframe in use (refer to Section 2.3 for more details about position and attitude controllers for quad-copters).

The PWM Block configures the PWM outputs for servo motors: the block accepts the signals from controller as input and writes those values to the selected channels, that are topics on their turn, subscribed by motor drivers modules.

In Figure 1.10, it is depicted the interconnection between PX4 blocks and attitude controller: this model is ready for building process and deployment on the selected Pixhawk Series flight controller, that has to be installed on the actual UAV for flight test. Since that actual flight testing is outside of the purposes of this thesis, in the next chapter, the simulation potentialities of the Support Package is explored, while the controller implementation exploiting these blocks for flight tests is left for future works.

Chapter 2

Implementation and Simulation of Customized Quadrotor Control Algorithms using Model-Based Design

2.1 Model-Based Design for Unmanned Aerial Systems

Embedded software is often the differentiating factor in a product success. Dealing with the need to create more complex software with better quality in less time while staying innovative and competitive, organizations seeking to manage complexity have increasingly turned to Model-Based Design.

Model-Based Design is a model-centric approach for developing control, signal processing, communications and other dynamic systems. Rather than relying on physical prototypes and textual specifications, Model-Based Design uses a model throughout development [42]. The model includes every component relevant to system behavior algorithms, control logic, physical components and intellectual property.

In Figure 2.1, the central role of the model is represented and the focus on it can be found in all the phases shown. In particular:

- Research & Requirements: it is the model that encapsulates all design information, selected system components features and application scenarios. The

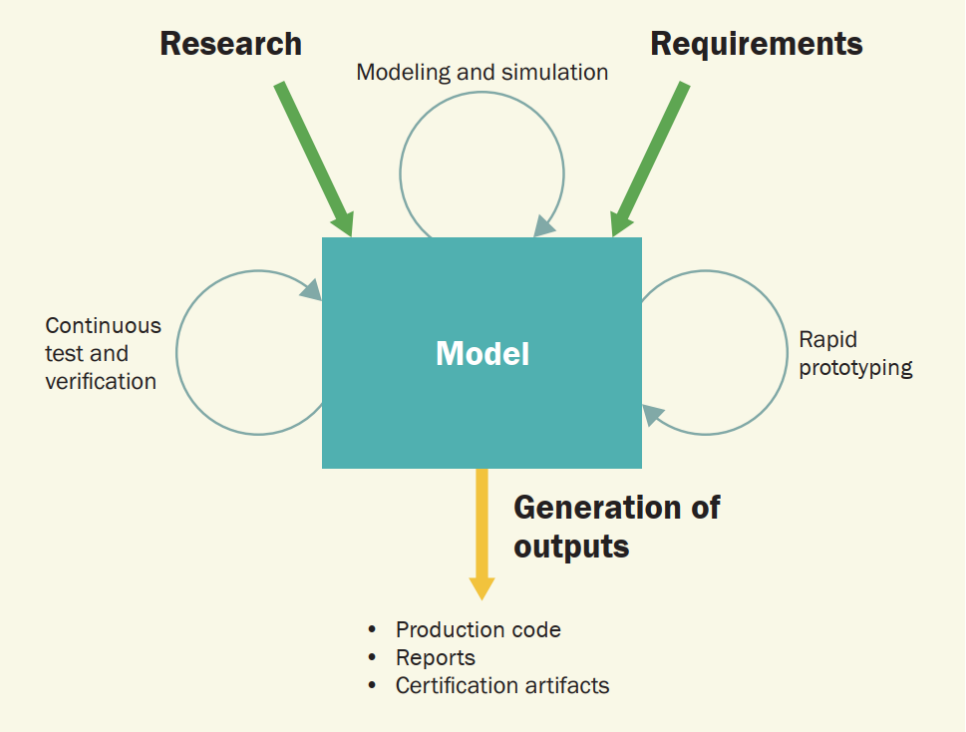

Figure 2.1: Model-Based Design workflow [42].

requirements are modeled to ensure their consistency and accuracy and usually the model contains more information than a text document. This leads to decreased risk for errors of interpretation. At the same time, all the knowledge required for the project coming from research is encapsulated into the model.

- Modeling & Simulation: model elaboration is an iterative process that uses simulation to turn a low-fidelity system model into a high-fidelity implementation. Knowledge derived from the continuous development and improvement of the model includes not only design specifications and details about the system, but also product knowledge, expertise and design best practices. The model of the entire system is simulated to investigate system performance and component interactions, validate requirements, check the feasibility of a project and conduct early test and verification. In simulations, design problems and uncertainties can be investigated early, preventing problems that could emerge only after the construction of expensive hardware prototypes.

- Rapid Prototyping: modules adopted for describing the embedded software can be used to generate code for rapid prototyping when they reach an adequate level of detail. Prototyping is a technique that uses simulation to validate

a design before the hardware is available. Sometimes, to fully understand the system, it may be necessary to use a hardware prototype for experiments and from which building the model. The knowledge acquired from these experiments is then stored in the model and from there the virtual prototype can be modified and updated.

- Continuous Test & Verification: test and verification are the practices of simulating a design at every stage of development. They can be carried out in various forms such as software/processor/hardware-in-the-loop testing (refer to next Sections for details).

- Generation of Outputs: common outputs of the process are production code, reports and certifications. In particular, the model can be used for production code generation using specific tools. The automatic code generation guarantees that software systems deployed in safety-critical applications, such as aerospace or automotive field, could satisfy rigorous development and verification standards and achieve the required certifications (an example of this kind of certification standards is the DO-178 for aerospace software employment [43]). Incidentally, automatic code generation have an impact into the roles of control and software engineers, freeing them from coding algorithms by hand, moving their focus from manual implementation to the software integration.

One of the most common methodologies used for implementing Model-Based for development of complex systems, such as aerial systems, is the V-Model [44]. It uses essentially the same steps as the "waterfall" model [45] that is generally acknowledged as the traditional software development flow, progressing through requirements, design, coding, testing, and release. The difference between V-Model and "waterfall" is that, instead of proceeding through the steps in a linear fashion, V-Model bends upwards after the implementation (coding) phase, with the purpose of matching each development step with a corresponding test phase.

The Figure 2.2 shows how the V-Model splits the development process into two main phases. The left side of the V is the part of requirement analysis, function design and change management. The right side of the V concentrates on the main verification and validation (V&V) activities showing how they are connected between the various activities. The difference between verification and validation is that verification is an objective set of tests to confirm that the product meets the metrics of the requirements, while validation seeks to demonstrate that the product meets the original intent [46].

A key piece for V&V in the V-Model is constituted by in-the-loop simulations:

- A Model-in-the-loop simulation (MIL or standard simulation) helps to evaluate the algorithms in a simulation environment at the beginning of the development [48].

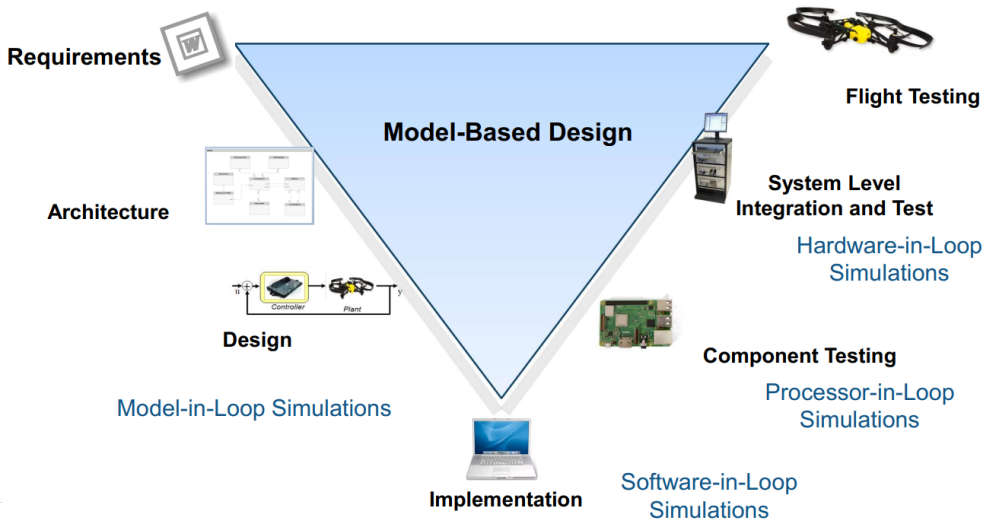

Figure 2.2: V-Model applied to a multi-rotor development [47].

- A Software-in-the-Loop (SIL) simulation compiles algorithms source code and executes that code as a separate process on a generic computer. By comparing SIL and standard simulation results, it is possible to test the numerical equivalence of model and code [49].

- A Processor-in-the-loop (PIL) simulation is a test technique that allows designers to evaluate a designed controller and its coded implementation while running on the selected micro-controller, with the objective of measuring both hardware and software performance. Model still runs in the development environment while the controller only runs on the board [50].

- A Hardware-in-the-loop (HIL) is a real-time simulation in presence of hardware and other control systems in which a dynamic simulator is replaced by the real system. In an ideal HIL test, the system is substituted with its simulator and other hardware and software are exactly implemented [51].

Not every step in this process is strictly necessary before moving to the actual testing: depending on the certification requirements and the complexity of the designed system, developers can adopt one or more of these simulation steps for V&V.

In this thesis, the focus is mostly put on the phase of a quadrotor develompent process regarding the implementation of already existing algorithms on the Pixhawk 4 autopilot board and on their test with in-the-loop simulations.

The suite MATLAB/Simulink has become a standard in the field of the development of models and simulation: this suite enables to design the controllers and

algorithms that are going to be boarded in the UAV, and also model the UAV platform for simulation purposes [47].

To embrace automatic code generation, this suite has been equipped with other tools, such as Embedded Coder, that gives the possibility to rapidly transfer the controller portion of the models to embedded target processors. In addition, libraries help to customize generated code to meet compliance standards, optimize it or integrate the new code with existing application code [40].

For the purposes of this work, the package described in Section 1.3 exploit the Embedded Coder for automatic coding of the modules for the PX4 flight stack (Section 1.2.4).

The same toolbox gives the possibility to test the generated code in two different simulation frameworks: executing the code on the same host platform that is used for the modeling environment (SIL) or flashing and running the code into the autopilot board (PIL).

The next Sections proceed through the V-Model methodology starting from the description of the classical quadrotor model and the control algorithms adopted, and proceeding to the in-the-loop simulation phases with their interface with visualization tools.

2.2 Quadrotor UAV model description

Once the mission and the system requirements have been defined, the first practical step into the V-Model development process is to build a model as accurate as possible of the flying machine that has to be automatically controlled.

The multi-rotor is an UAV lifted and propelled by two or more motors with propellers, usually electrically operated. This vehicle is characterized by a simple design where lift and torque control is delivered by varying the rotational speed of the fixed-pitch rotors, measured in revolutions per minute (RPM) [52].

A quadrotor is a multi-rotor with four rotors with two sets of identical propellers that are rotating in opposite direction (two propellers rotate clockwise, two counter-clockwise). There are two possible configurations for the quadrotor design: "Plus" configuration (“+”) and "Cross" configuration (“×”). The main difference between the two configurations consists in the rotors position compared with the direction of motion of the vehicle. Figure 2.3 shows the possible rotor configurations.

For the aim of this thesis, it has been chosen to model a conventional "+" configuration quadrotor, taking into consideration the different type of design stated in [53] and [54].

Such other 6 degrees of freedom (DOF) rigid systems, kinematics (the branch of the mechanics that studies the motion of a body) offers the basis to identify two reference systems and the transformations between them to understand the

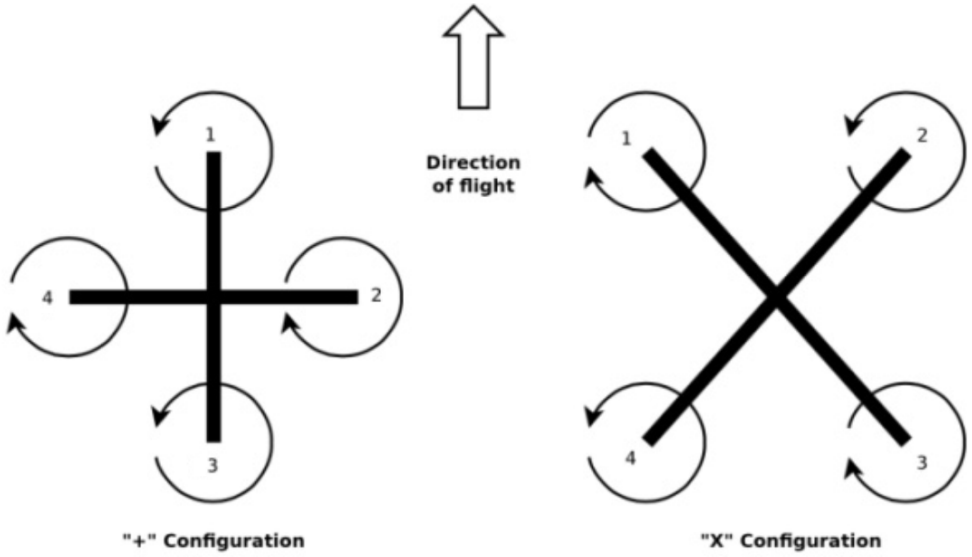

Figure 2.3: Quadrotor configurations.

quadrotor motion.

They are shown in Figure 2.4 and described below:

- The Earth reference system, or NED-frame (o_E, e_N, e_E, e_D) is chosen as the inertial right-hand reference. e_N points toward the North, e_E points toward the West, e_D points Downwards respect to the Earth and o_E is the axis origin. This frame is used to define the linear position and the angular position of the quadrotor. This is not the only way to describe the Earth reference system but it is the most useful for this project;

- The quadrotor reference system, or B-frame (o_B, x_B, y_B, z_B) is attached to the body. x_B points toward the quandrotor front-hand, y_B points toward the vehicle right-hand, z_E points downwards and oB is the axis origin. oB is chosen to coincide with the center of the quadrotor configuration.

As already stated, quadrotor UAV has 6 DOF but, since it is equipped with just four propellers, it is not possible to reach a desired set-point for all the DOF at the same time. However, thanks to its structure, it is quite easy to choose the four best controllable variables and to achieve the four basic movements which allow the vehicle to reach a certain height and attitude [53]. The control of each motion is achieved by altering the rotation rate of two or more rotors, thereby changing the torque load and thrust/lift characteristics.

The four basic movements shown in Figure 2.4 are:

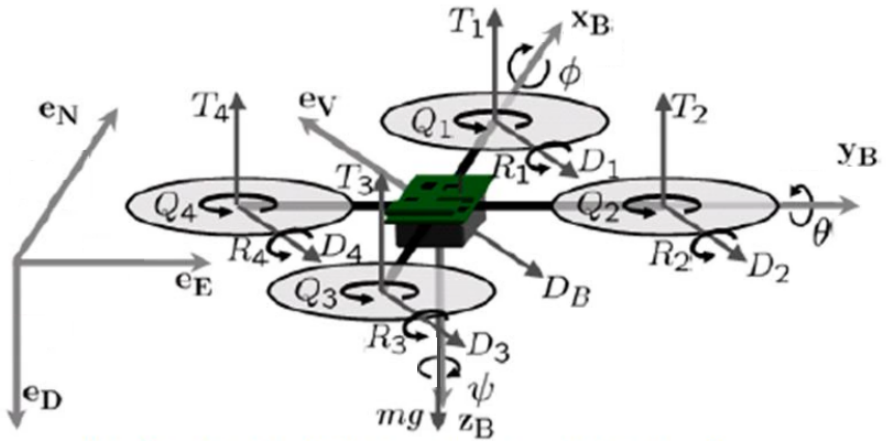

Figure 2.4: Conventional quad-rotor reference systems and basic movements.

- Throttle: increasing or decreasing the power of all motors equally causes the aircraft move along the z_B axis.
- Roll Φ: this movement is caused by increasing the power of one lateral motor while decreasing the power of the motor on the opposite side (e.g. increasing motor 2 power and decreasing motor 4 power or vice versa). The rotation generated is around the x_B and causes a change in speed perpendicular respect to the direction of the flight. The sum of the power of all motor remains equal.
- Pitch Θ: very similar to the preceding movement, it regards front and rear motor. If the motor 1 (front) decreases its power while the motor 3 increases, the vehicle pitch forward; vice versa, it pitches backward. The rotation generated is around the y_B axis and causes a change in speed parallel respect to the direction of the flight. The sum of the power of the all motor remains equal.
- Yaw Ψ: this command is provided by increasing/decreasing the power of the motor couple 1-3 (front-rear) motor power together while decreasing/increasing the power of couple 2-4 (left-right). Since the couples rotates in opposite direction, this unbalance makes the quadrotor turning towards one direction or towards the other, rotating around z_B axis. A torque respect to the z_B axis is generated which makes the quadrotor to turn.

The angular speeds are the derivatives of Φ, Θ, and Ψ conventionally called p, q and r respectively.

Reference systems, basic movements, angular speeds, forces and torques are needed for writing of the quadrotor mathematical model (the equations of motion):

it represents a dynamical system with a 12-dimensional state and 4-dimensional control inputs. Deriving these equations is outside the scope of this thesis but these details can be found in [52], [53] and [54].

Incidentally, the model needs to reflect the quadrotor behaviour in different flight phases (climb, descent, forward flight, manoeuvres, and so on). It incorporates body motion dynamics and propulsion system aerodynamics. The propulsion system aerodynamics modelling tasks include momentum theory, blade element theory, ground effect, vortex ring state and windmill break state which have not been investigated in this work.

2.3 Control algorithms

Once the model has been designed, a controller has to be engineered to proceed to Model-in the-loop simulations, following the V-Model steps.

The control of the quadrotor position and attitude is accomplished by a controller that can implement a wide range of control algorithms.

The quadrotor is an under-actuated system: this means that, for accomplishing one basic movement, a quadrotor needs to vary more than one control inputs. For example, to move forward, gaining speed along x_B direction, the quadrotor must first change its attitude by pitching downwards generating a horizontal force, while maintaining its altitude increasing total thrust. Similarly, in order to move laterally in the y_B direction, the quad-rotor must change its attitude by rolling to the right or left while, again, maintaining its altitude varying the total thrust [55].

For achieving this goal, the control algorithm has to find the value of the motors voltage, which maintains the UAV in a certain position or moves it in a certain direction required for following a reference signal. The control algorithm receives the data from the sensors as inputs and provides the Pulse Width Modulation (PWM) signal of the four motors as main output to vary the rotational speed of the propellers [33].

In this project, two different control algorithm have been investigated and implemented:

- a Proportional-Integral-Derivative (PID) control mechanism is the default technique implemented in the PX4 flight stack described in Section 1.2.4 and the same control algorithm has been used in Section 4.2 to fly a customized trajectory into vineyard scenario.

- a Linear Quadratic Regulator (LQR) control is used to explore the feasibility of the customization of the default control algorithm into the Pixhawk 4 flight controller and to exploit the possibilities of the development environment, described in 1.3. It has been used also for the Processor-in-the loop simulations.

2.3.1 PID controller

Proportional-Integral-Derivative (PID) controller is a common control feedback mechanism broadly used in industrial control systems. The reasons of its success are the simple structure, the good performance for several processes and the possibility of tuning even without a specific model of the controlled system [56].

A PID controller estimates an "error" value as the difference between a measured process variable and a desired set point. The controller attempts to minimize the error by altering the process control inputs. In an attitude control system, for example, roll, pitch and yaw angles are generally used as process variables to acquire the desired orientation.

In the frequency domain, the transfer function of a PID controller can be represented by equation

$$G(s) = K_P + \frac{K_I}{s} + K_D s$$

Each gain in the PID controller can be tuned to modify a particular transient response parameter of the feedback system: in particular K_P is the proportional coefficient, K_I is the integral coefficient and K_D is the derivative coefficient. The blocks $1/s$ and s represents the integration and derivation operations [21].

The K_P value is increased to reduce the time required for the output signal to reach the desired signal. By increasing only the K_P value in the PID controller, a steady-state error can be reduced and expected to be between the desired signal and the output signal. In addition, setting an overly high K_P value will also propagate any inherent disturbance signal within the system and cause the system to be affected by unstable oscillations.

The K_I value is increased to eliminate the steady-state error of the feedback system. However, the system might become increasingly oscillatory in the steady-state when the K_I value is too much increased.

The K_D value is increased to reduce the overshoot and the settling time. Although derivative control does not affect the steady-state error directly, it introduces damping to the feedback system. This would allow the system to use a larger K_P value, which would result in an overall improvement of the steady-state performance.

The quad-rotor model is highly nonlinear and so parameters that work well at some flight conditions, could not work well, for example, during the take-off phase. For this reason, the PID tuning is a complex problem: usually an adequate and careful tuning, such as the one applied in Section 4.2 for tracking a specified trajectory with a quadrotor UAV, is necessary to reach the desired performance [17].

2.3.2 LQR controller

Linear Quadratic Regulator (LQR) is an optimal control that produces a steady-state minimum error minimizing a cost function for providing the best control signal. Intuitively, in general, it does not exits a "universal" best control signal but it depends on the weights given to the performance or to the actuation effort in given conditions (and the control signal is the "best" only in these conditions).

Given a continous-time linear time invariant system state space representation

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

where all states are measurable and the feedback gain is the matrix K

$$u(t) = -K(x_r - x)$$

where x_r is the vector of desired states.

In order to obtain the optimal K that gives the best control signal,

$$J = \int_{t^0}^{\infty} (x^T Q x + u^T R u) dt$$

represents the cost function that has to be minimized solving the LQR problem. If the unknown elements of the matrix K are determined minimizing the cost function J, then $u(t)$ is optimal for any initial state x_0.

The matrices Q and R determine the relative importance of the error (performance) and the expenditure of this energy (actuation effort) [57].

Matrix Q weights the size of state responses: it is diagonal positive definite and have the same number rows and columns as the states. Increasing values in a Q diagonal element reflects into a lower error in the corresponding state. Vice versa low values means that the error in the corresponding states are not so significant to achieve the best performance.

Matrix R is similar to Q and weights the control: if an element of this matrix is large, then the control action of the corresponding input will be penalized, reducing its energy expenditure.

In general, it is possible to state that a trade-off thumb rule could be putting more energy (lower value of the corresponding R element) where a lower error is needed (higher value of the corresponding Q element).

After tuning R and Q and solving the Riccati equation of the LQR problem, (for more mathematical details refer to [57]), it is possible to derive the optimal matrix K that controls each state of the system individually.

In MATLAB, the function `lqr(A,B,Q,R)` solves the continuous-time, linear, quadratic regulator problem and the associated Riccati equation. This command calculates the optimal feedback gain matrix K given the matrices A, B, Q and R.

In conclusion, it is worth to underline that in order to use a LQR controller for a non-linear system such as the quadrotor, the vehicle model must be linearized about a certain operating point. For a very clear example of quadrotor system linearization, refer to [58].

2.4 Software-in-the-loop simulation

As stated in Section 2.1, simulation tools are required because of increasing complexity in algorithms and their software implementation for embedded systems ; "in-the-loop" simulations are known as prominent tools before realistic tests of the system and are used for verification and validation of automation and control software [51].

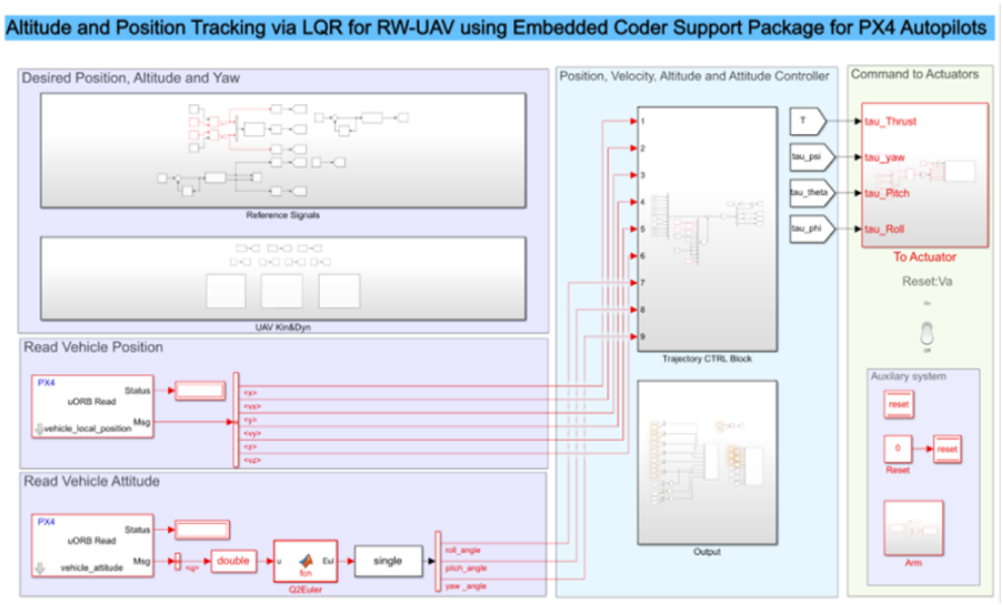

Figure 2.5: SIL Simulator implemented using the Embedded Coder Support Package for PX4 autopilots.

A SIL simulation compiles source code and executes the code as a separate process on a generic computer. In this way, developed embedded software is tested and then rapidly evaluated and debugged.

SIL does not strictly require automatic code generation. After developing code from the model, it can be compiled with traditional compilers and then tested: manual coding of complex control algorithms, even in high level programming language such as C++, is a time-consuming and an error-prone activity.

The MATLAB code generation tools are capable of generating embedded C/C++ code optimized for specific hardware directly from MATLAB codes and Simulink models [59]. Specifically, Embedded Coder provides SIL automation which converts MATLAB projects to executables and also generates the SIL infrastructure for interfacing with MATLAB. Then, it verifies if the generated C/C++ code is correct and it profiles run-time performance [49]. The process for code generation is similar for both Software and Processor-in-the-loop simulations, so for more details about it refer to Section 2.5.

The Embedded Coder Support Package for PX4 Autopilots, described in Section 1.3, provides the option to simulate developed PX4 autopilot algorithms with SIL. The Support Package supports simulation of algorithms by generating an executable referred as PX4 Host Target on the host and jMAVSim simulator. PX4 Host Target is an emulated board that supports code generation and deployment, like other supported hardware boards: it has to be selected in the hardware setup phase of the configuration (specifically it has to be selected `posix_sitl_default.cmake` for the CMake configuration, refer to Appendix A for details). During the simulation, it is possible to perform signal monitoring and parameter tuning of the model.

In [29], it is stated that jMAVSim is a simple quadrotor simulator that allows to fly this type of vehicles, running PX4 in a simulated world. During the hardware setup of the development environment, this simulator is downloaded and installed.

For the purposes of this work, model and controller described in Section 2.2 and Section 2.3.2 have been included in a SIL simulator derived from a template available in the development environment. The complete model derived in this way is represented in Figure 2.5.

Exploring the functionalities of the template and of jMAVsim simulator, it results that they do not guarantee enough flexibility to test the performance of a UAV in a complex scenario, such as operations for precision agriculture.

In Figure 2.6 is shown the jMavSim 3D environment: in particular, in this simulator, customized 3D scenario cannot be loaded and, in addition, there is no easy way to integrate extra sensors or obstacles in the simulation, so there is no possibility of simulating a visual inspection or other types of complex operations [61]. Moreover, loading customized quadrotor CAD models in jMAVsim is not straightforward and also the the customization of the dynamycs used is problematic. In addition, the documentation, that can be retrieved in [60], is pretty poor.

Once assessed the potentialities of the SIL simulation in this development environment and decided that they were not suitable for the final scope of this thesis, the work moved towards PIL simulations as described in the following Section.

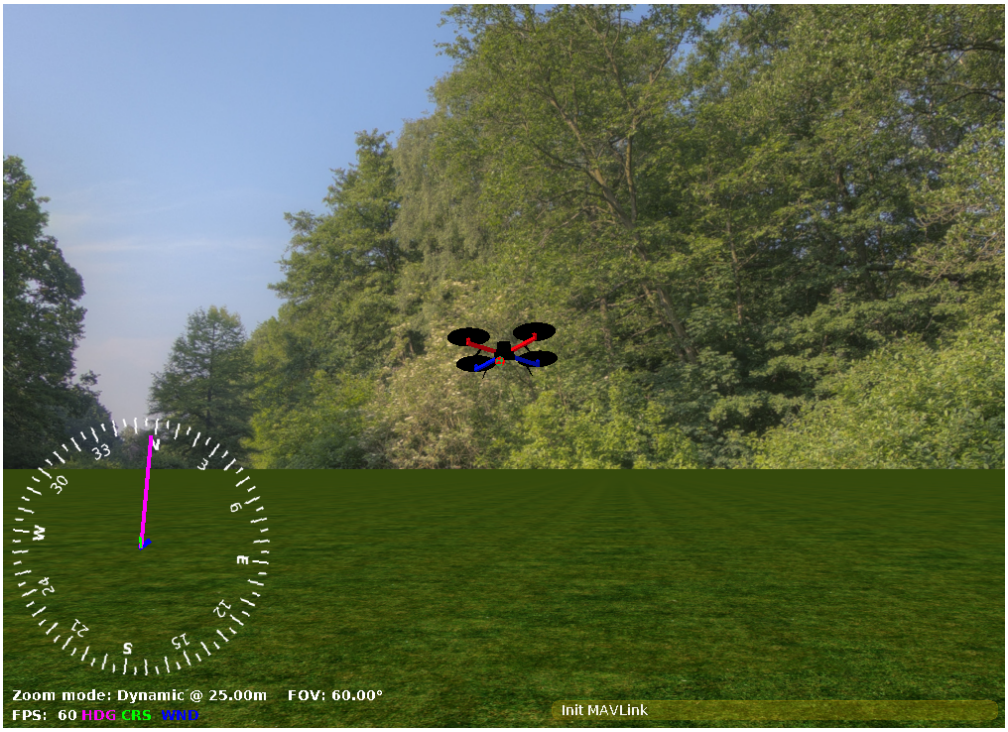

Figure 2.6: jMavSim PX4 Simulator [60].

2.5 Processor-in-the-loop simulation

PIL test environment is an intermediate step between the software simulation and the flight experiments: this step is very important in critical software development such as GNC systems that are going to be boarded in an autopilot, to prevent errors and delays, derived from the execution of the production code, before the flight tests.

In fact, once the GNC algorithms are designed and tested in simulation and the integration framework is developed, it is important to validate the software in the target hardware in order to verify the implemented algorithm behavior.

In particular, during PIL testing, it is possible to detect failures that have not been detected in the standard simulation or in the SIL simulation, such as synchronization and timing issues.

In this thesis, taken into the consideration the results obtained with PIL simulations in [62], it has been explored the framework described in Section 1.3.

Specifically, the objective was to cross-compile the model of the LQR controller described in Section 2.3.2 and make it run on the Pixhawk 4 to control the quadrotor, while tracking a trajectory designed between vineyard rows (see Section 4.1 for trajectory generation details). Hence, PIL simulations have been performed to

validate the effectiveness of the proposed controller scheme and the computational capability of the board.

For performing these kinds of PIL simulations, the development environment makes available a "wrapper model" that allows to integrate in it both the dynamic model and the controller. In Figure 2.5, it can be noticed how the blocks are interconnected: after placing the developed model and controller blocks, it is possible to proceed for generating the code of the LQR. A scope is present in order to test numerical equivalence by comparing standard simulation results against PIL simulation results.

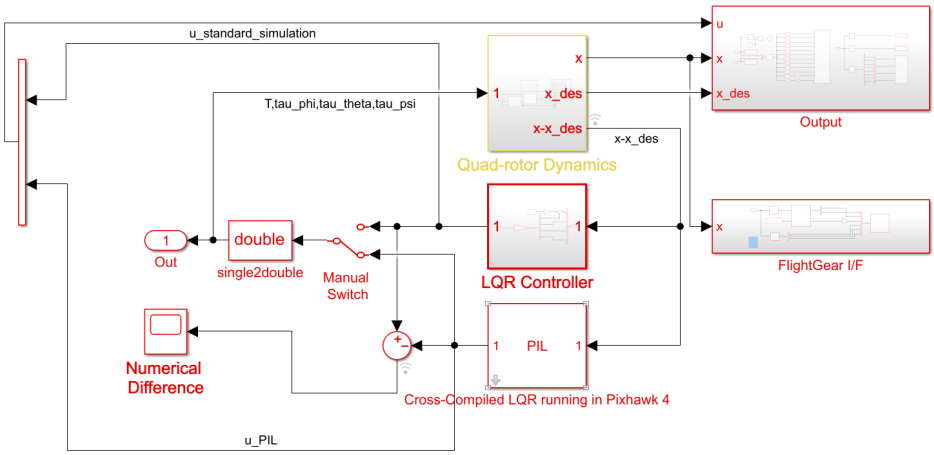

Figure 2.7: PIL "wrapper" used to cross-compile the controller and to assess its performance by comparing with standard simulation. Flight Gear interface is described in Section 2.6.

As preparation for code generation, the quadrotor dynamic model and the controller have been adjusted to satisfy the requirements of the Embedded Coder and the design specification of the PX4 autopilot software: these requirements comprise discretizing the models according to the sample time, configuring the hardware implementation and setting the code style customization options.

When the models are ready for code generation, other parameters have to be set to determine how the code has to be built. Configuration parameters, in fact, determine the method that the code generator uses to produce the code and its format. These parameters can be chosen manually or automatically, in order to maximize the selected pre-defined code generation performance objectives, such as traceability [64], execution efficiency [65] or safety precaution [66]. For pursuing one objective or the other, execution speed, CPU throughput and memory usage need to be traded off as required [40].

All the configuration parameters are stored in the `ert.tlc` file, the Embedded Real-Time system target file. A system target file is a collection of scripts, written

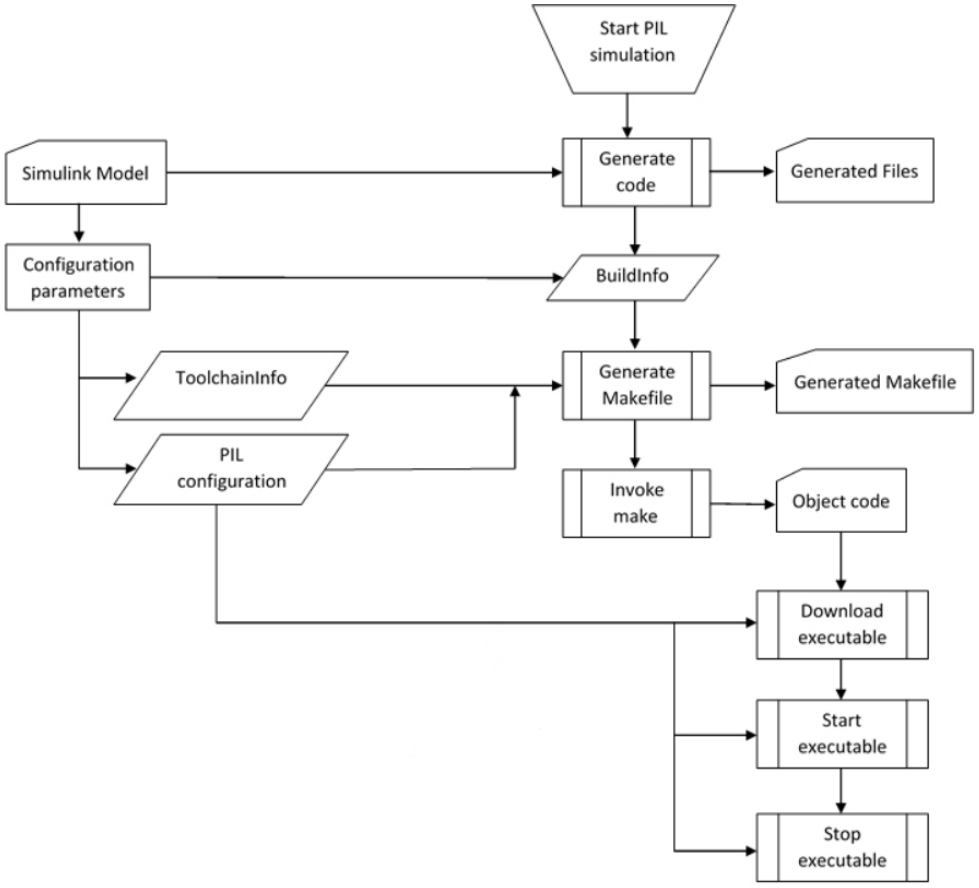

Figure 2.8: PIL simulation process overview [63].

in an interpreted language, that explains how the model has to be converted into source code. This file has to be interpreted by the Target Language Compiler to transform the representation of the Simulink model file into target-specific code. For more information about TLC, refer to [67].

In Figure 2.8 is illustrated the steps executed when a PIL simulation starts in Simulink: the model is translated and the Embedded Coder creates a build folder within the working folder, where the generated source code files are stored. In this case, the build folder name is `Controller_ert_rtw`, derived from the chosen subsystem block name. Here, the target file that calls (with the `include` directive) other target files used for the compiling application is called the "entry point". Another folder, called `slprj` is created for the code that could be shared among multiple models [40].

Directions to `ert.tlc` and to all the source code files are stored in the `buildInfo.mat`. Starting from this file, it is now possible to generate a `.mk`

makefile, using the expected toolchain, in this case the Windows Cygwin toolchain, containing also the GNU tools for PX4 autopilots, installed during the support package setup (refer to Appendix A for details). A makefile consists in one or more commands for making up the project with CMake, that is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner [41].

Following the instructions contained in the makefile and the list of source code files provided in `CMakeLists.txt`, CMake is used to invoke the ARM-GCC compiler for building the application that will have to run on the autopilot board. When the command `make` is performed, CMake generates a native build environment that compiles source code, creates libraries and builds executables. The excutable object code is then downloaded to the Pixhawk autopilot board to be run.

The module that runs the newly generated code is `px4_simulink_app`, the application started during Pixhawk booting process that substitutes the default flight control modules of the PX4 firmware (for more details, refer to Section 1.3).

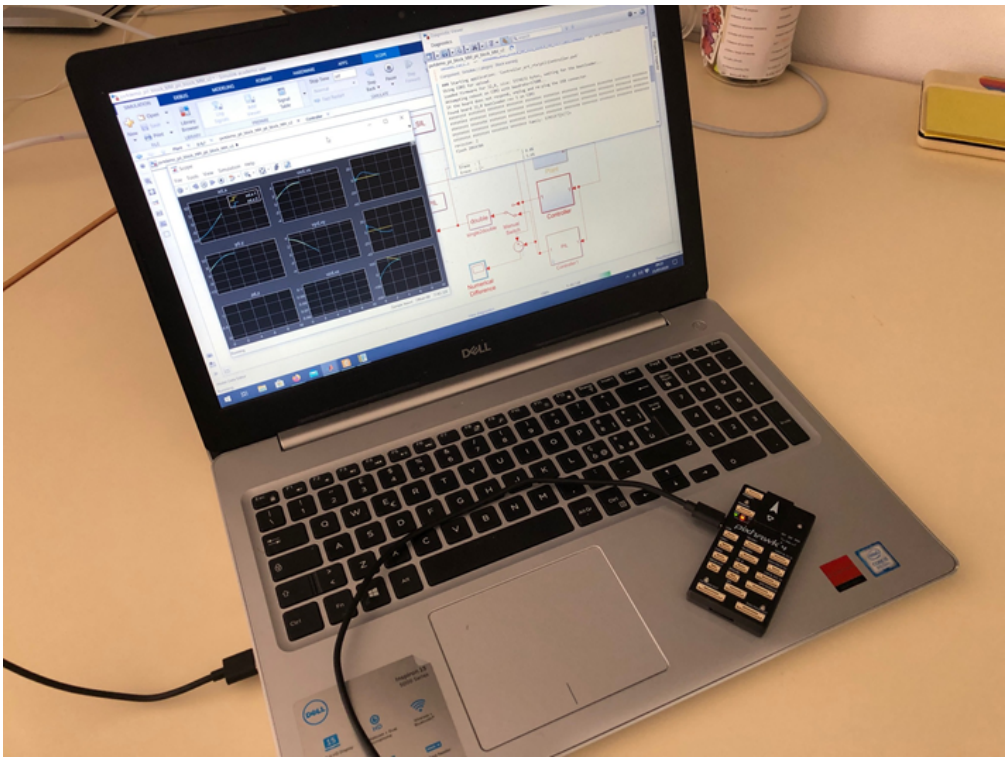

Figure 2.9: PIL simulation: the quadrotor dynamic model is running on the personal computer while the controller is running on the Pixhawk 4 autopilot board.

The "wrapper" model running in the generic computer communicates with the Pixhawk board through a micro USB 2.0 serial port: in Figure 2.9, it can be seen how the autopilot is connected and how a running PIL simulation looks like. The interface program interprets and forwards the USB microframes between the model simulator and the autopilot hardware to accomplish the transmission of the flight state data and flight control actuator commands for regulating the thrust of each simulated rotor and acquiring the indicated reference signal (see details about USB communication in Sections 1.2.1 and 1.2.2).

After downloading the controller code into the autopilot board for PIL simulation,`px4_simulink_app_task_main` function contained in the `ert_main.cpp` source code file is the first being executed by `px4_simulink_app` module: this function initializes the NuttX OS running instance with a "tick-rate" of 1 ms, passing this parameter to the function `nuttxRTOSInit`. The so-called "tick" can be considered as the time given to hardware to update the state of the controlled vehicle. Incidentally, since most of sensor topics send updates every approximately 4 ms (see Section 1.2.3 for details about "topics"), choosing a smaller "tick rate" for simulation purposes would not add realism to the simulation, but it just could introduce delays caused by the USB serial communication. The complete code listing of `ert_main.cpp` source code file is reported in Appendix A.

While performing PIL simulations, in fact, Simulink prevents the user to reduce the sample time under the threshold of 1 ms: in this way, communication delays between the PC and the board do not affect the simulation process. In Figure 2.10, it is shown the simulation exception, caused by a wrong choice in sample time.

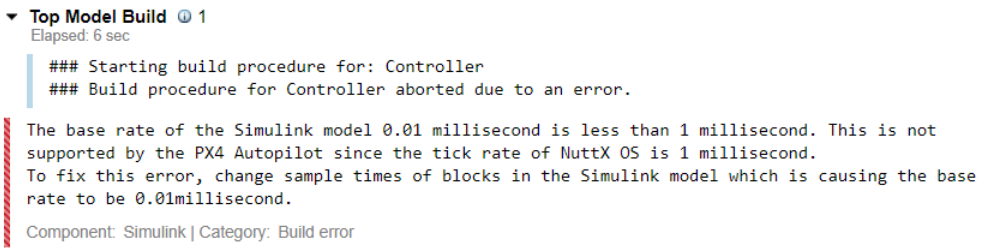

Figure 2.10: Default NuttX "tick rate" threshold for PX4 PIL simulations.

If simulations provide satisfactory results, a similar procedure can be adopted to generate and flash the stand-alone controller into the autopilot board for flight tests, using the Simulink blocks described in Section 1.3.

Results of the simulations performed while tracking a designed trajectory using a cross-compiled controller on the Pixhawk 4 autopilot board are reported in Section 4.3.

In conclusion, a step-by-step guide for realizing a PIL simulator as described in this section is reported in Appendix A.

2.6 Flight Gear simulator interface

Commercial flight simulator software, such as those described in [61], are adopted in academic research as flight dynamic modeler as they are equipped with accurate flight models and detailed 3D visual effects.

As stated in the introductory part of this thesis, basic visualization of the quadrotor performance has not been the only objective pursued in the project: in fact, a lot of effort has been put to use the virtual scenarios, derived from the actual location selected for the flight testing operations and loaded in flight simulators, for generating feasible trajectories and assess the behaviour of a multi-rotor UAV in these contexts.

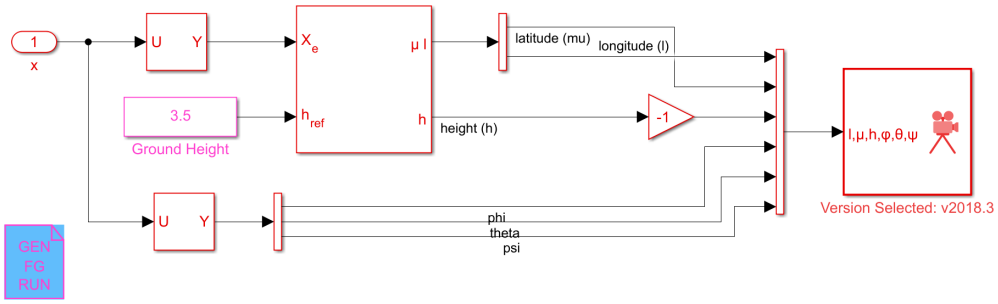

Figure 2.11: FlightGear interface implemented in Simulink and connected to the PIL simulator represented in Figure 2.7.

To carry out this objective, the first attempt has been made with the FlightGear MATLAB/Simulink interface. This choice has been made on the base of the results obtained in [68].

FlightGear is an open-source flight simulator available through a GNU General Public License (GPL). The goal of the FlightGear developers has been creating a sophisticated and open flight simulation framework dedicated to research or academic environments, pilot training, as an industry engineering tool but, at the same time, to amateurs as a challenging desktop flight simulator [69].

A FlightGear interface is included within the Aerospace Blockset for MATLAB/Simulink [70]. The interface consists in a unidirectional transmission link that exploits the FlightGear `net_fdm` binary data exchange protocol: in particular, data packets are transmitted via User Datagram Protocol (UDP) from the simulated model to a running instance of FlightGear. Figure 2.12 shows the implementation of the FlightGear interface in Simulink, connected to the PIL simulator described in the preceding Section. It is possible to notice the input data needed to the simulator block to visualize the flight dynamics.

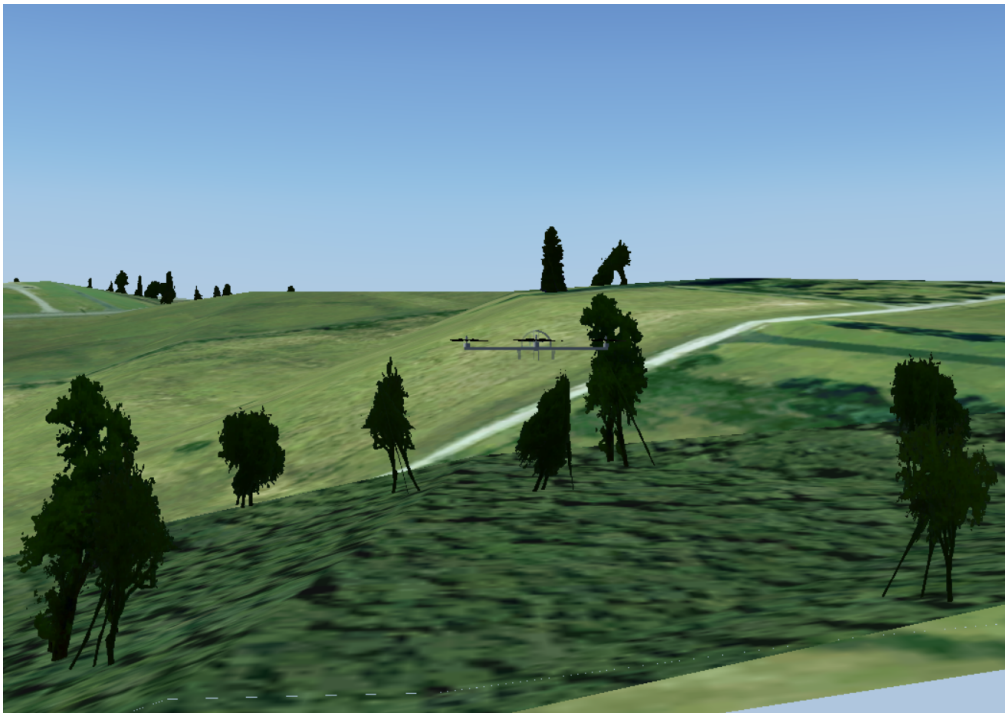

Figure 2.12: Quadrotor PIL simulation in FightGear.

For performing simulation in FlightGear, a quadrotor CAD model available in [29], called `FlightGear-TF-Mx1`, has been loaded and used to visualize the flight dynamics already implemented. In Figure 2.12 it can be seen the quadrotor UAV in stationary flight conditions over the area selected for the flight tests (described in Section 3.3).

It is possible to notice that, even if the flight dynamics is well represented, environment scenarios are not detailed: in particular, the vegetation is almost absent and the grape plantations are not present. Moreover, it is not possible to understand precisely the slope of the terrain. In conclusion, this visualization tool is not suitable to design precise trajectories into a well-defined scenario, as requested for the project.

For this reason, it has been decided to move to another simulation environment for realizing a virtual environment where it could be possible to test different trajectory tracking control technologies: in this way, efforts have been made to bring Simulink 3D Animation [71] in the development loop. This simulation environment is deeply described in Section 3.6.

Chapter 3

Development of 3D Simulation Scenarios from LIDAR Point Cloud Maps

3.1 Virtual scenarios for trajectory tracking design and visualization

After the studies carried out on the simulation framework for implementing advanced control techniques on a commercial autopilot board, the focus of this thesis has been moved on how to exploit the 3D point cloud maps, collected by Laser Imaging, Detection And Ranging (LIDAR) survey campaigns, for generating 3D virtual scenarios.

The point cloud map exploited for this scope has been originally processed for the works illustrated in [15] and [72]. In particular, in [72], 3D point clouds of vineyards are used to generate low-complexity 3D mesh models, reducing the the amount of data contained in the cloud, without losing relevant crop shape information. These models will have their final use as an accurate space description to be loaded in robotic vehicles and machines for their in-field precision agriculture operations [13].

In this project, the point cloud has been processed to become a virtual scenario for simulation of trajectory tracking for a quadrotor UAV flight: a series of software tools (ConverGo [73], CloudCompare [74], Meshlab [75], MATLAB), well described in the following Sections, have been used to obtain a virtual product to be loaded into the selected simulator, Simulink 3D Animation [71].

Figure B.1 shows the overview of the process steps followed for realizing the virtual scenario. In general, after the conversion from World Geodetic System-1984

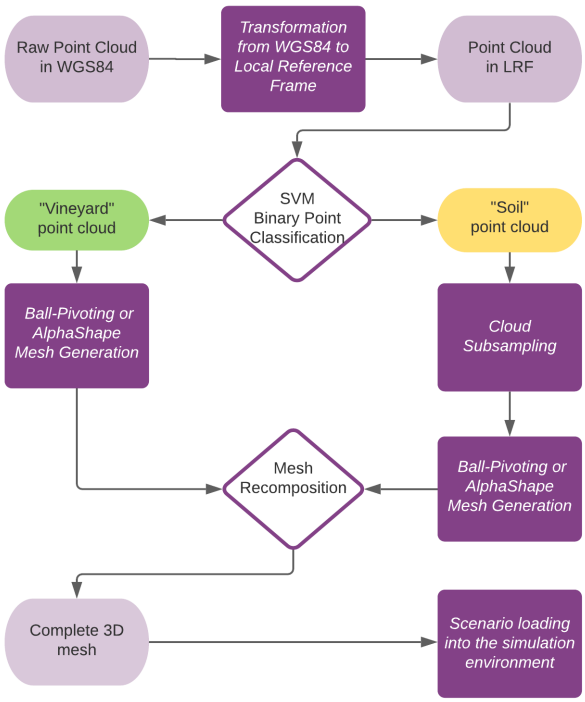

Figure 3.1: 3D scenario development overview.

(WGS84) to a local reference frame, the point cloud has been split with a binary classification algorithm in two other clouds, identified as "vineyard" and "soil". This operation has two objectives: reducing the points density of "soil" cloud for pulling down the data size to handle and building a continuous mesh without the holes caused by the shaded areas under the grape canopies. The following Sections are walking through this process explaining taken decisions, exploited algorithms and used software tools for pursuing the purpose. All the practical details for repeating the achieved results are reported in Appendix B.

3.2 LIDAR technology

Since the invention of the laser in the 1960s, scientists and engineers have been using this technology to image, detect and find ranges to objects. In fact, already at the time of the Apollo 15 lunar mission in 1971, astronauts used a laser system to map the surface of the Moon. This type of implementation is known as Laser Imaging, Detection And Ranging (LIDAR).

LIDAR is implemented by using pulsed or modulated laser beams and laser detectors to determine precise distances to objects. The laser pulses are split outside the laser output: one beam is directed to the receiving components (typically a

telescope with photon counting detectors) while the other portion is directed onward to a distant object. Using the split beam as a time reference, the reflected beam return time is compared and the difference between them will be twice the time required for the light to travel from the origin to the object. Accordingly, it is trivial to calculate the distance [76]. A graphical example of this process can be seen in Figure 3.2.

The LIDAR receiver will see returned light from particles in the air, multiple objects along the way and background. The various return times are used to map the various distances to objects and therefore a 3D map can be generated of the scanned area. The laser beam travels in a straight line and therefore it can be used to place points accurately on a single plane [76].

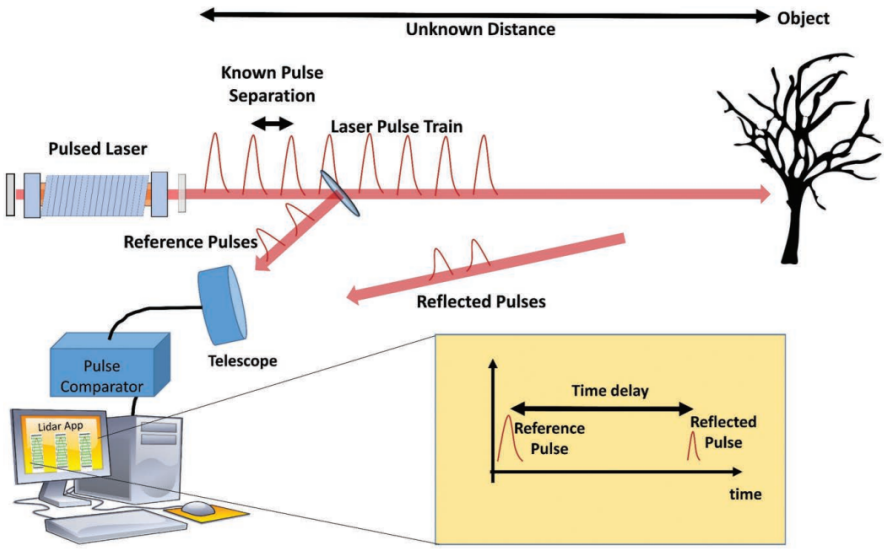

Figure 3.2: LIDAR used to detect the unknown distance of an object [76].

LIDAR systems are often installed on aerial, terrestrial or underwater vehicles for collecting accurate data: distance measurements are then associated to precise GPS positions to give them a standard reference system (normally WGS-84); often, these data are associated also to IMU system to combine also the orientation information and scan angles. Results are dense groups of elevation points, called "point cloud maps" that can be used to generate other geospatial products [77].

LIDARs usually operate at a monochromatic wavelength but, nowadays, multi-spectral LIDAR sensors have been developed. These sensors have the possibility to acquire data at different wavelengths and so diversity of spectral reflectance from objects can be recorded to show the color information.

Airborne LIDARs (installed on a manned or unmanned aircraft) are divided

into topological LIDAR and bathymetric LIDAR: their main difference is about the wavelength of the light beam used. Interesting applications exist in oceanography for knowing the exact depth of the surface of the sea or for locating objects, in the case of a maritime accident or research activities [78].

In conclusion, LIDAR systems allow scientists and mapping professionals to examine both natural and manmade environments with accuracy, precision, and flexibility.

3.3 Point cloud map description

As already stated, data used for this project comes from LIDAR survey campaigns realized for the studies [15] and [72]. In particular, the selected vineyard is located in Serralunga d'Alba (Piedmont, Northwest of Italy). This piece of land includes three contiguous parcels and several partial ones, cultivated with grapevine and covering a total surface of about 2.5 ha.

The area is located at longitude positions range [44.62334 44.62539] and latitude position range [7.99855 8.00250] (WGS84); the elevation ranges from 330 to 420 m above sea level. A loamy soil and a steep slope (ranging from 8 to 30%) characterise the vineyard, which is exposed towards the south-east direction (ranging from 120° to 160°) with the vine rows perpendicular to the maximum slope gradient. Due to the irregularity of the vineyard terrain morphology in terms of altitude, soil features and inclination, the plantation vigour usually varies within and between parcels. A partial picture of the described area can be seen in Figure 3.3.

Figure 3.3: Picture of detail of Cerretta Vineyard in Serralunga d'Alba (Piedmont) [courtesy Az. Agr. Germano Ettore].

The point-cloud map was generated with the Agisoft PhotoScan software, by processing a set of more than a thousand aerial images acquired with an airborne Parrot Sequoia multispectral camera. Agisoft Photoscan is a stand-alone software product that performs photogrammetric processing of 2D raw digital images and generates 3D point cloud and other spatial data to be used in many applications such as cultural heritage documentation, visual effects production, precision agriculture as well as indirect measurements of objects of various scales [79].

The data collection took place at the end of June, with presence of about 1 cm diameter green grapes.

3.4 Point cloud map processing and classification

A LIDAR point cloud map is a set of points, each one represented by an array of WGS84 latitude, longitude and elevation coordinates: most of the open-source processing software tools can only recognize local metric Cartesian reference (LRF) systems and cannot interpret the point clouds in WGS84 reference system.

Moreover, point clouds are saved in `.las` format files: usually, the size of these files is approximately 500MB per ha of scanned land and they are not readable with a common CAD software and are not directly loadable into a flight simulation environment.

For the conversion between the reference systems, the tool used in this work is the ConveRgo software, an open-source platform able to perform coordinate transformations between the various frames in which the geographical data are expressed, also considering the respective cartographic systems. The altimetric component is also considered for the conversions between ellipsoidal and geoidic heights [73]. A very useful guide for using this software can be found in [80].

The ConveRgo conversion is very accurate but during the process the color information cannot be preserved. In Figure 3.4, the all-white Cerretta vineyard point cloud map can be seen after the conversion from WGS84 to LRF: this point cloud is composed by almost 31 million points and has a size of 1.15 GB, once converted in `.txt` format for the processing.

After the conversion step, the main tool exploited for visualizing and classifying the point cloud map is the CloudCompare software.

CloudCompare is a 3D point cloud editing and processing software. Originally, it has been designed to perform direct comparison between dense 3D point clouds. It relies on a specific octree structure that have good performances when performing this kind of task.

CloudCompare can deal with point clouds of great dimensions on a standard laptop, typically more than 10 million points. It also includes various point

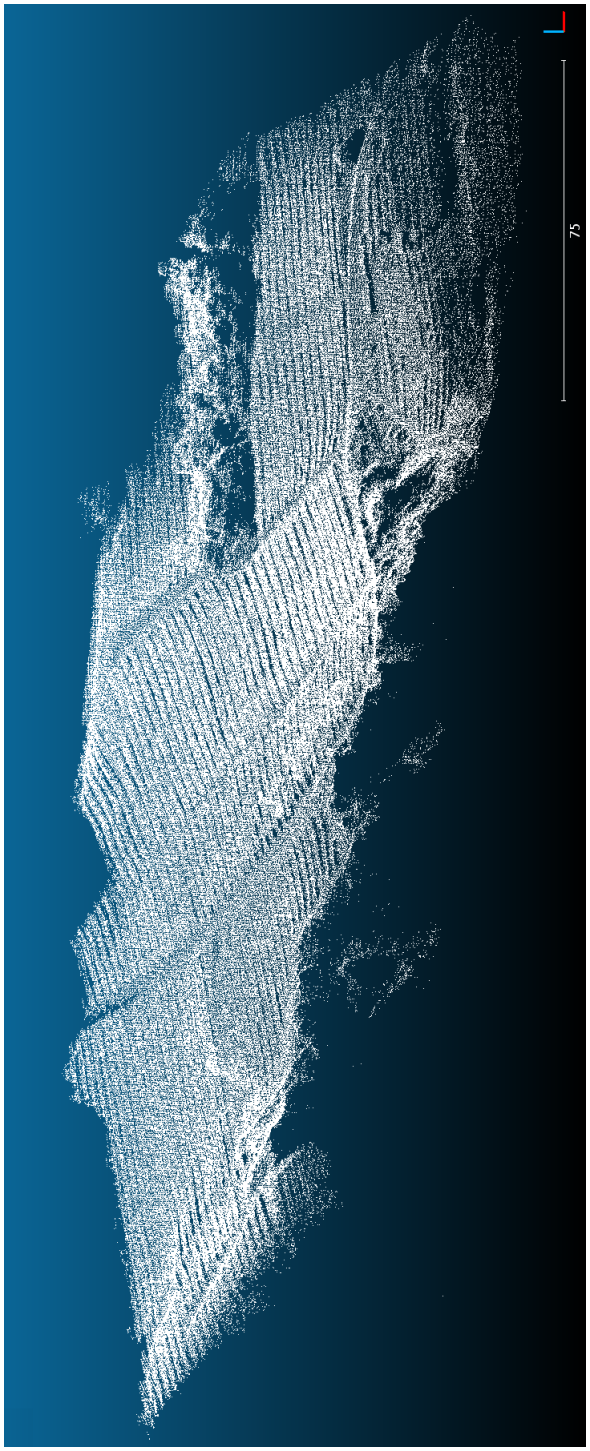

Figure 3.4: Cerretta vineyard point cloud map visualized in CloudCompare.

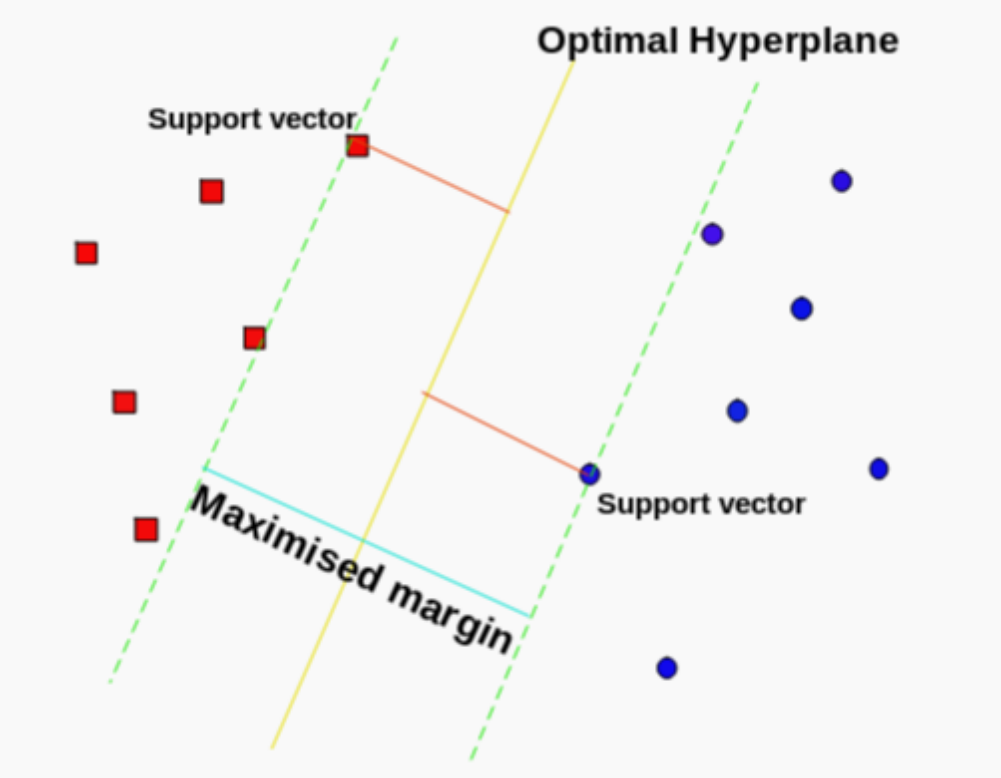

Figure 3.5: Optimal hyperplane separating two linearly separable classes.

cloud processing algorithms like resampling (SPATIAL [81]), color/normal vectors/scalar fields management (smoothing, gradient evaluation, statistics, etc.), statistics computation (χ^2-squared Test [82]), interactive or automatic segmentation (Connected-component labeling [83]) as well as display enhancement tools. In academic research, CloudCompare has already been used in the analysis of vineyard point clouds captured with terrestrial sensors in [84]. An interesting overview of the potentialities of this open-source software can be found in [85] while all the functions are deeply explained in its user manual [74].

The first processing step on the LRF point cloud consisted in identifying if points were part of canopies or soil and then separating the two sets of points. Identifying and separating these points are not simple operations, taking into consideration only the LRF point position information.

For this reason, a binary classifier, called Caractérisation de NUages di POints (CANUPO), has been exploited as a plugin software of CloudCompare to perform the classification.

This tool has been explicitly designed for 3D point clouds classification of complex natural environments: it works directly on point clouds and it is largely

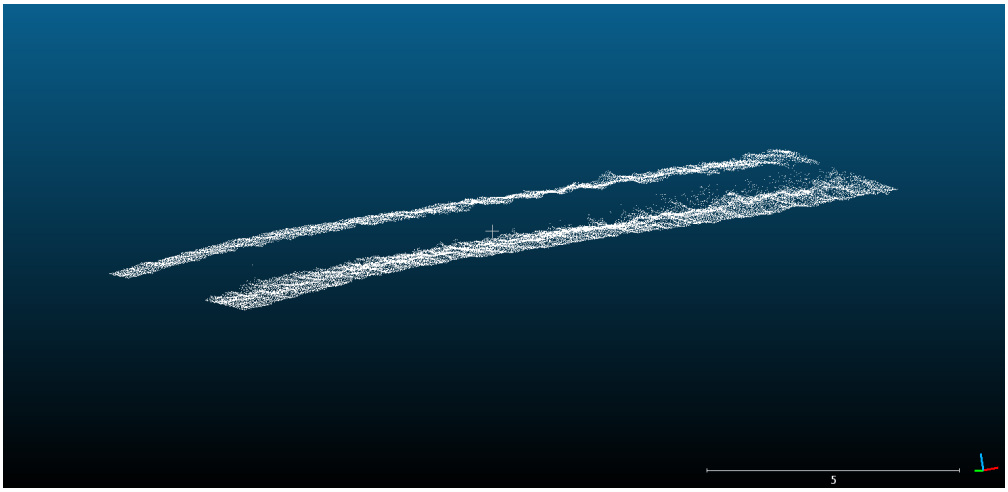

Figure 3.6: "Soil" class example.

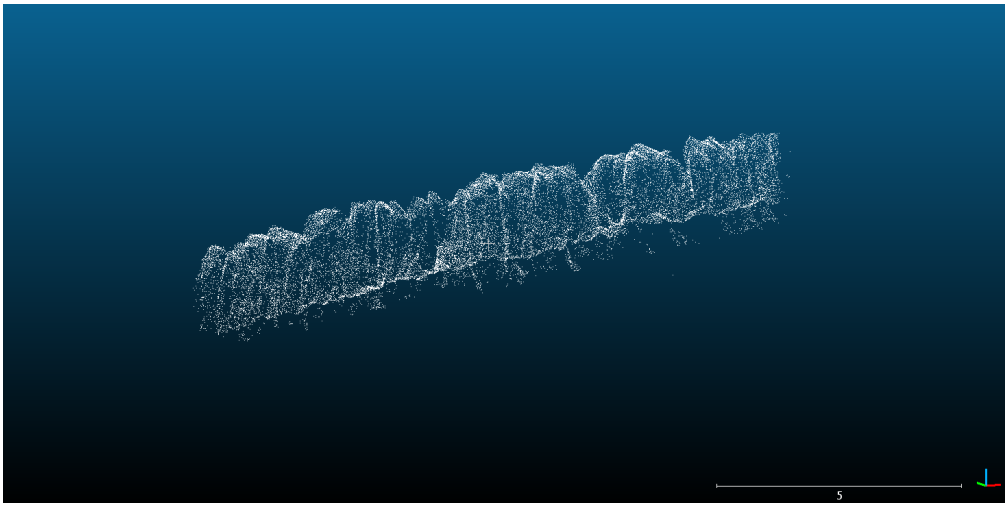

Figure 3.7: "Vineyard" class example.

insensitive to shadow effects or changes in point density. Moreover, it allows some degree of variability and heterogeneity in the selected class and it is coded to handle large point cloud datasets [86].

The approach used by CANUPO strongly rely on Support Vector Machines (SVM) : a support-vector machine is a linear model for classification that constructs a hyperplane for separating classes, maximizing the distance (or the "margin") between each other. The hyperplane for which the margin is maximum is the optimal hyperplane [87]. In the Figure 3.5, an optimal hyperplane is shown while separating two linearly separable classes.

If two classes are not linearly separable, the idea is to add one or more dimensions to find a dimensional space where the classes are easily separable and then project the decision boundary found in the original space [88].

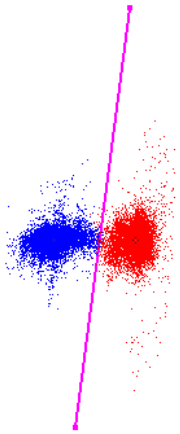

Figure 3.8: Visual representation of the high dimensional space where points are clearly separated in two classes.

In CANUPO, multi-scale dimensionality feature is used to describe the local geometry of a point in the scene and how it can characterize simple elementary environment features (ground and vegetation). The general idea is to define the best combination of scales that allows the maximum separability of classes: the strength of this method is that a reliable classification is based uniquely on the 3D geometrical properties of the elements on multiple scales, allowing for example, recognition of the vegetation on complex scenes with very high accuracy [86].

In practice, the user could have an intuitive sense of the range of scales at which the categories will be the most geometrically different, but in many cases, because of natural variability in shape and size of objects, this is not a trivial exercise. CANUPO solves this issue giving the possibility to automatically construct a classifier that finds the best combination of scales (e.g. all scales contribute to the final classification but with different weights); this combination maximizes the separability of two categories that the user has previously manually defined (e.g. samples of vegetation and samples of ground segmented from the point cloud).

In the case of Cerretta Vineyard point cloud (Figure 3.4), the best examples of classes were defined by manual segmentation of "vineyard" and "soil" samples: this selection allows to have good results in classifying the canopies, accepting the fact

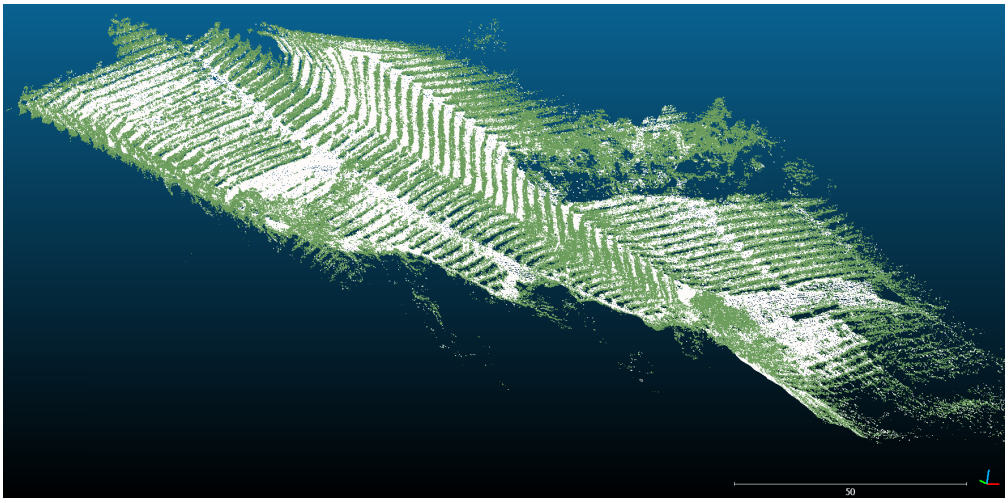

Figure 3.9: CANUPO point classification applied to the data set.

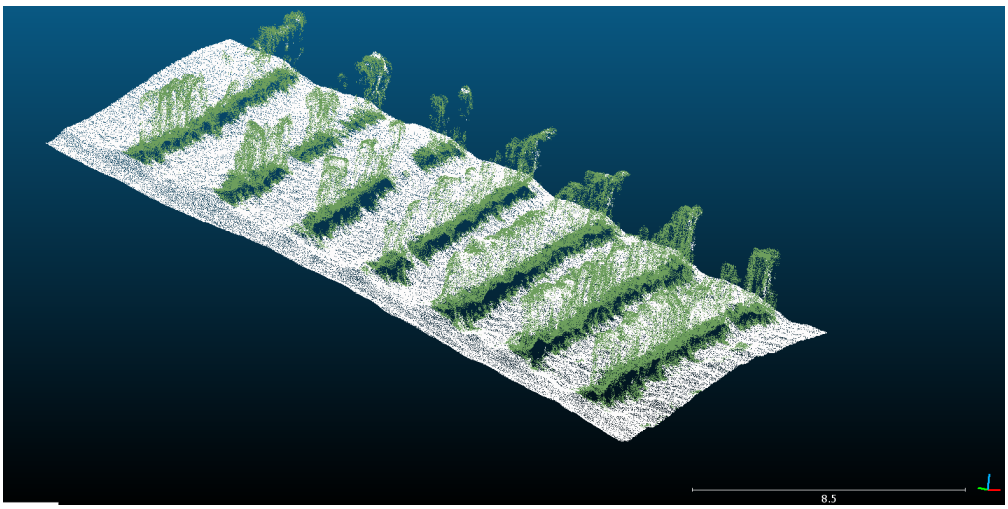

Figure 3.10: Classified point cloud portion selected for building the virtual scenario.

that other type of vegetation could not have been recognized by the classifier (for example, the wood portion of the scenario). In Figure 3.6 and in Figure 3.7, the two classes examples selected for the classification of the entire point cloud map are shown.

The "soil" class sample is composed by two strips of ground (approximately 20 meters long) located in between the vineyard rows and with a small difference in height among each other. The "vineyard" class sample is a vineyard row approximately with a lenght the same as the previous example and approximately

70 cm wide. This last sample has been expressly chosen because the density of the points and the absence of shade effects in the lower part of the canopies.

Once the samples have been loaded in CANUPO, a "trial & error" approach have been used to find a good combination of the parameters requested by the classifier training: it has been found that a minimum scale of 0.1 to a maximum scale of 3.0 with a number of 10000 core points were enough to guarantee a good separability.

Figure 3.8 shows the cloud points relating to the Cerretta vineyard in the dimensional space where a linear hyperplane can guarantee a clear separability between classes: the "soil" class points (in blue color) are separated from the "vineyard" class points (in red color). An optimal hyperplane (in magenta color), automatically calculated by CANUPO, divides the two classes.

In Figure 3.9, the effects of the CANUPO SVM classification algorithm applied to the entire vineyard are represented: in the figure, "soil" classified points are colored in white while "vineyard" classified points are colored in green. It can be noticed that the algorithm does not work really well in the part of cloud where undefined vegetation (like wood) appears: depending on the need of the project, this part could be cut or another classifier could be built for separating the cloud portion in more different classes iteratively.

For building the initial virtual scenario, it has been decided to ignore the portion of the cloud not well classified after the first step of the CANUPO classification process and to select a small portion of the vineyard: this portion is shown in Figure 3.10 and it is composed by seven vineyard rows approximately 29 meters long, interspersed by a free soil gap approximately 1.2 meters wide on each side.

The entire portion has a width of approximately 19 meters covering an area of approximately 580 square meters with an average topographic slope of approximately 30% in the direction perpendicular to the rows, that means an altitude change of 7 meters from lowest to the highest point.

3.5 3D mesh generation

The reconstruction of precise surfaces from point clouds is a fundamental step for building a virtual 3D scene. In fact, generation of polygonal meshes, that can satisfy high modeling and visualization demands, is required in different applications like video-games, movies, virtual reality applications, flight simulation, etc..

An overview of methods and techniques for modeling and visualization of 3D scenes is provided in [89]. Historically, the first algorithm introduced in this field has been the computation of the "convex hull" to generate a shape from a finite point set [90].

Starting from the portion of the vineyard point cloud shown in Figure 3.10, two

approaches for generating a shape have been tried and compared: the alpha-shape generation algorithm [91], implemented in the MATLAB suite, and the ball-pivoting algorithm [92], used in Meshlab software.

The `alphashape` function in MATLAB is based on the alpha-shape generation algorithm: it is a generalization of the "convex hull" approach where, given a finite set of points, a family of shapes can be derived from the Delaunay triangulation of the point set. The Delaunay triangulation maximizes the minimum angle of all the angles of the triangles in the triangulation process: it tends to avoid the construction of slivers, triangles with extremely acute angles. The real parameter α controls the desired level of detail. The construction of the shape graph proceeds in the following way: for each point in our point set, a vertex is created; then an edge is created between two vertices whenever there exists a generalized disk of radius $1/\alpha$ containing the entire point set and which has the property that the two vertices lie on its boundary [91].

Indeed, the ball-pivoting algorithm is based on the fact that three points form a triangle, if a ball of a user-specified radius touches them without containing any other point. Starting with a seed triangle, the ball pivots around an edge until it touches another point, forming another triangle. The process continues until all reachable edges have been tried, and then starts from another seed triangle, until all points have been considered [92]. This algorithm has been implemented in Meshlab, an open source, extensible, mesh processing system developed at the Visual Computing Lab of the Institute for Information Science and Technologies (ISTI-CNR) of Pisa (Italy). MeshLab is a tool focused on mesh processing, instead of mesh editing and mesh design, and it is described as a mesh viewer application, where a 3D object, stored in a variety of formats can be loaded and interactively inspected, dragging and clicking on the mesh itself [75].

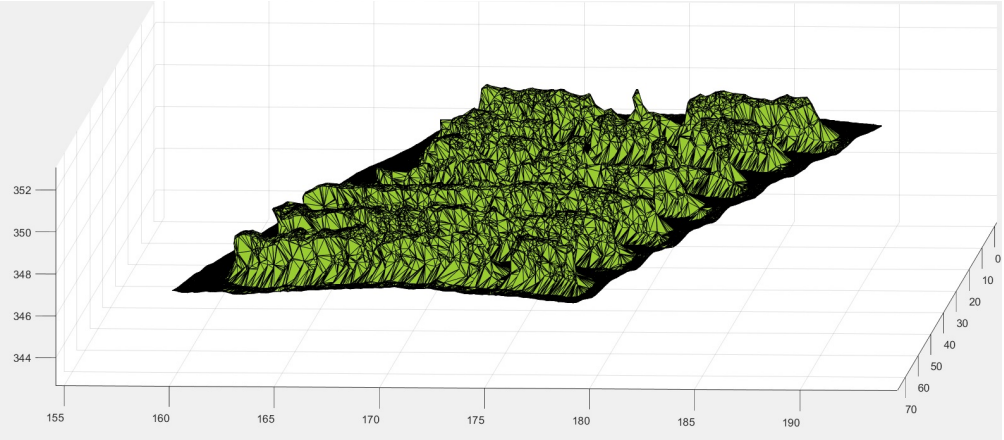

Figure 3.11: AlphaShape mesh construction intermediate result over a selected vineyard portion with $\alpha = 0.7$ (processed and visualized in MATLAB).

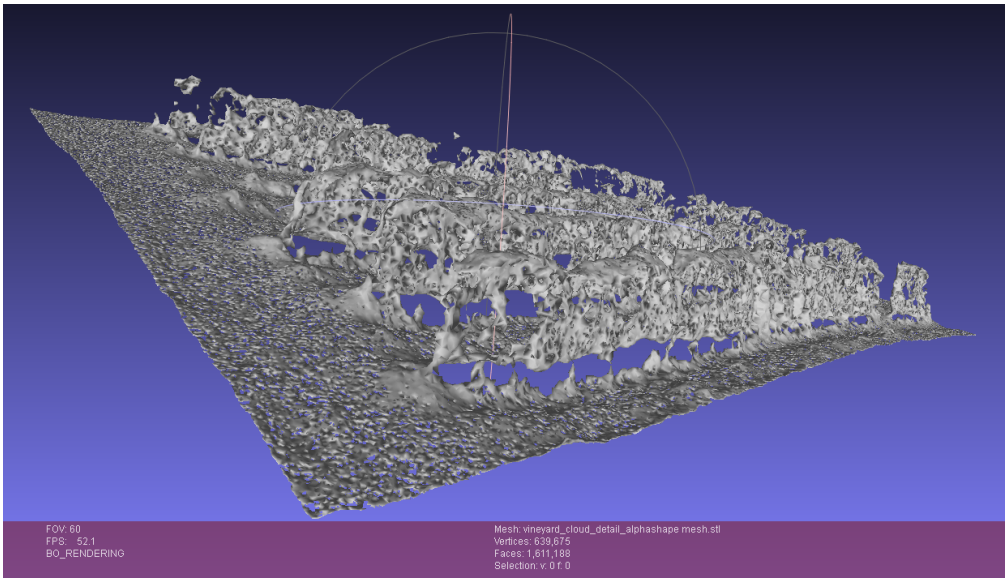

Figure 3.12: AlphaShape mesh construction over a selected vineyard portion with $\alpha = 0.1$ (processed in MATLAB, exported and visualized in Meshlab).

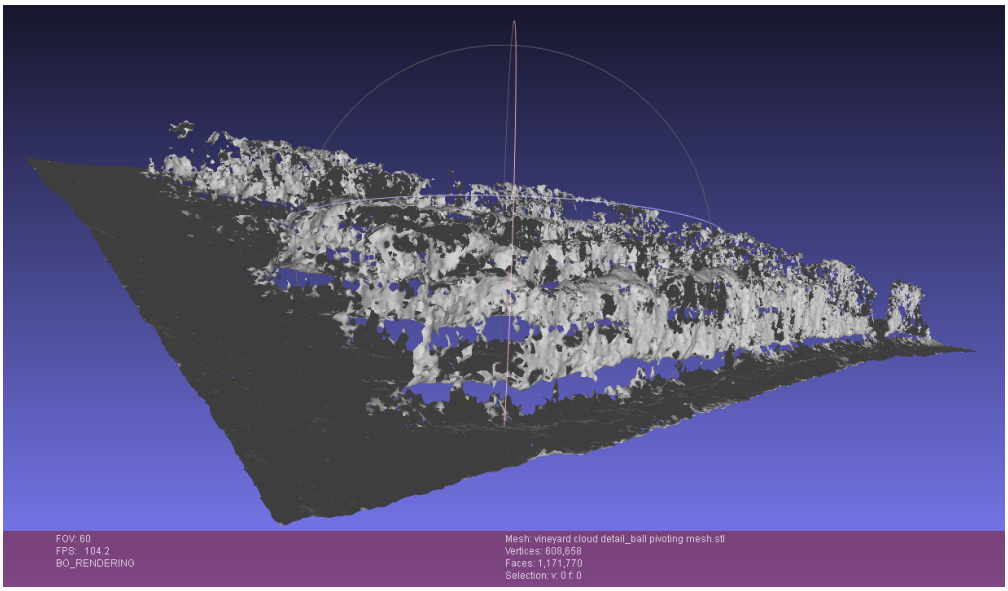

Figure 3.13: Ball-pivoting mesh construction over selected vineyard portion (processed and visualized in Meshlab).

In Figure 3.12, it is shown the application of the `alphashape` function to the portion of the vineyard point cloud to obtain a triangular mesh: it can be noticed that varying the α parameter, the level of detail increases, but if the α becomes too small, holes formed in the portion of soil in the shadow below the vineyard row, where the algorithm is unable to conclude the triangulation with the precision applied to the canopies or to the free soil. In Figure 3.11, the visualization of an intermediate result with α=0.7 is shown: this is a case where the AlphaShape algorithm is able to close the holes but the level of detail is poor.

Similar results in mesh generation were found using the ball-pivoting algorithm in Meshlab, that are shown in Figure 3.13: in particular, it has been found that Meshlab ball-pivoting algorithm gives results comparable to AlphaShape algorithm with $\alpha = 0.1$ (vertices and faces produced are almost the same number in both cases).

To solve the issue about the holes in the soil portion, keeping an appropriate level of detail for the canopies, it has been exploited the CANUPO classification described in Section 3.4: in particular, mesh generation algorithms have been separately applied to the two point clouds, coming from the binary classification. In this way, it has been possible to take advantage of a sub-sampling process of the "soil" point cloud in order to build a continous mesh of the ground, closing the holes.

In Figure 3.14, it can be noticed the successful results of the triangulation process over a sub-sampled "soil" point cloud. Then, after building the "vineyard" mesh with the original level of detail, it has been possible to recompose the scenario using a graphic editor (in this case Blender [93]).

In conclusion, the recomposed mesh in `.stl` format has been loaded into Simulink 3D Animation as explained in the next Section.

3.6 Scenario loading into Simulink 3D Animation simulation environment

Simulink 3D Animation is a MATLAB/Simulink tool for visualizing dynamic system behavior in a virtual reality environment. This simulation tool links developed models and algorithms to 3D graphics objects in virtual reality scenes. It gives the possibility to animate a virtual world during simulation. Collisions and other events can also be simulated in the virtual world and forwarded back into the running models. This tool gives also the possibility to stream video from virtual cameras for processing, debugging or demonstration purposes [71]. An example on how to proceed for simulating dynamic systems with Simulink 3D Animation can be found in [94].

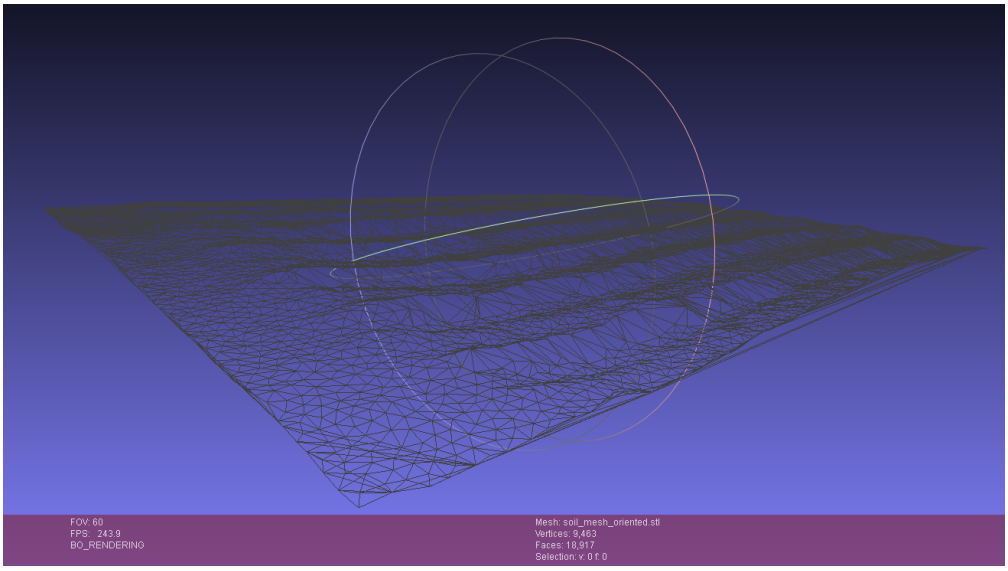

Figure 3.14: Ball-pivoting mesh construction over a subsampled soil portion.

This simulation environment includes editors and viewers for rendering and interacting with virtual scenes: 3D World Editor can, for example, import CAD and 3D mesh file formats (such as `.stl` or `.ply`) and the 3D worlds can be viewed immersively using stereoscopic vision.

For virtual scenarios, Simulink 3D Animation supports Virtual Reality Model Language (VRML) file format. VRML `.wrl` format is one of the most common 3D interactive navigation language that allows to create 3D scenes. It is an ISO standard format for representing 3D models. VRML gives the possibility to store different viewpoints that allow to navigate through the 3D model.

As already stated, the mesh recomposition has been carried out into Blender graphic environment, a tool that gives the possibility to easily interact with the 3D meshes [93].

Blender and Simulink 3D Animation use different reference systems: in particular, attention must be paid about the reference frame while exporting the `.stl` file regarding the scenario.

After exporting the mesh in the correct reference frame, it is possible to load it into a `.wrl` scene: the separation of the in two different meshes, "vineyard" and "soil", if kept unaltered, helps to choose different colors to identify the portions of the scenario.

Figure 3.15 shows the results of the virtual scenario construction process, derived from a real piece of land of Cerretta Vineyard, loaded into the simulation environment.

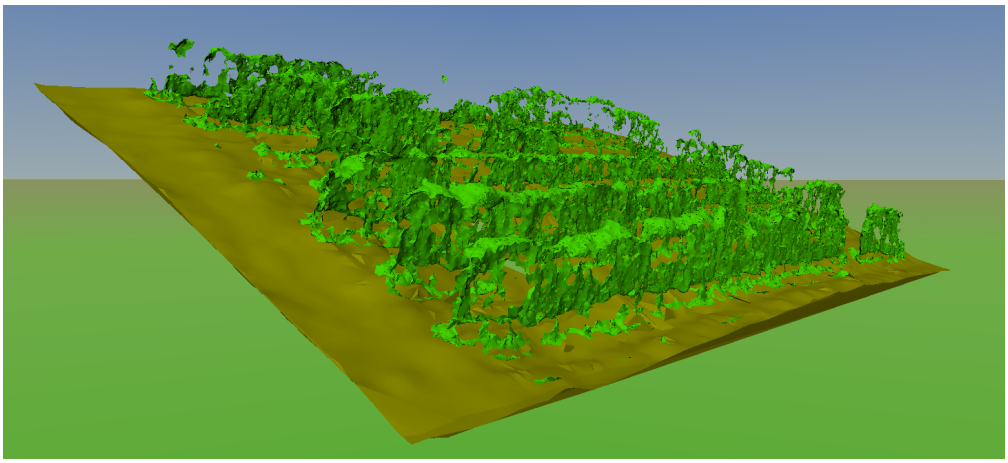

Figure 3.15: Re-assembled "vineyard" and "soil" meshes building the 3D vineyard scenario in Simulink 3D Animation.

Chapter 4

Results

4.1 Trajectory and reference signals generation

Vineyards on steep slope hills represent a high-complexity environment for automation development. Here, the aim has been to discover if the 3D scenarios, developed as described in Chapter 3, could be a useful tool for designing trajectories and visualize their tracking performed by UAVs: the designed trajectory has been used to asses the performance of a PID controller and as a testbed for the Processor-in-the-loop simulations of the LQR controller cross-compiled on Pixhawk 4 autopilot board.

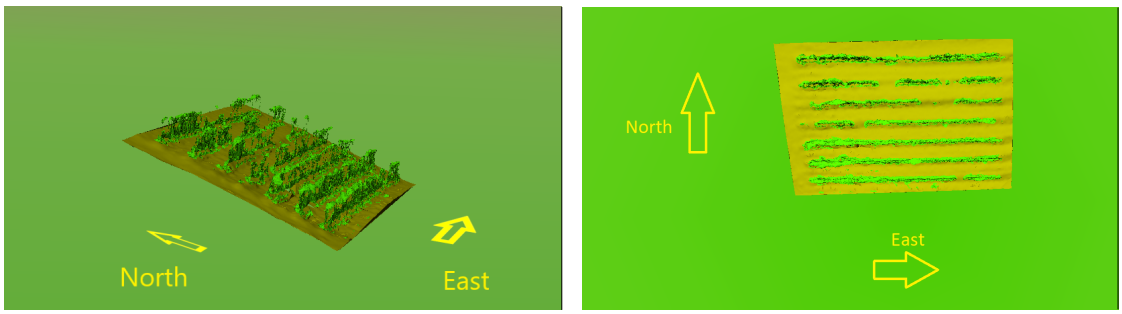

Figure 4.1: Scenario loaded in Simulink 3D Animation aligned to NED reference system

The 3D vineyard scenario has been loaded into an already existing quadrotor project template, developed by Mathworks [95] and already interfaced with Simulink 3d Animation: this choice was made to speed up the learning process about the simulation environment, concentrating the efforts on the control tuning and the evaluation of Simulink 3D Animation as visualization tool.

To ease trajectory design and reference signals generation, a simplification has been made: the vineyard rows have been aligned with the North-East-Down (NED) reference system of the simulator, neglecting the real orientation of the portion of soil in WGS84 (for details about reference systems, refer to Section 2.2 and Section 3.4). In particular, the rows have been aligned with e_E axis (East direction), leaving in this way the e_N axis (North direction) perpendicular to them. In this way, it has been also exploited the fact that the altitude change in the selected soil portion only occurs along the e_N axis. Figure 4.1 shows the generated scenario aligned with NED reference system.

Taken into consideration the measures of the selected portion of land, reported at the end of Section 3.4, and NED coordinates (the lower left-hand corner of the loaded scenario is placed at North=30.6 East=32.7), it became possible to estimate the way-points and the total distance that the quadrotor needed to cover during the flight.

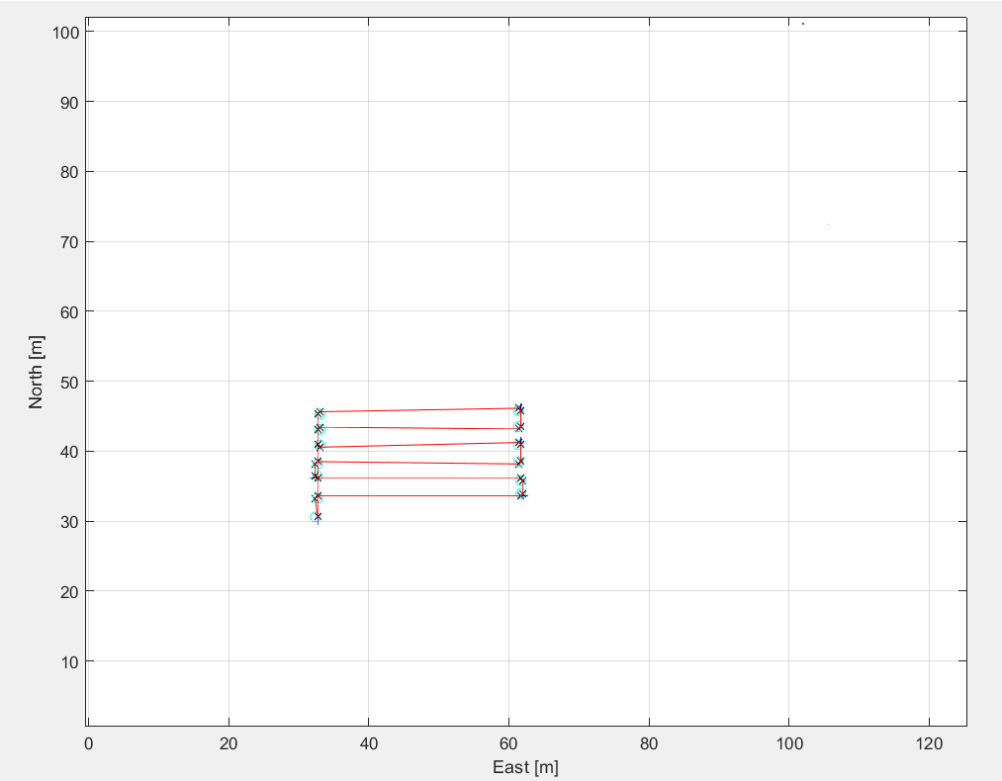

Figure 4.2: Way-points placing and trajectory building on a modified version of the `asbTrajectoryTool`, provided in the Mathworks Quadcopter Project,

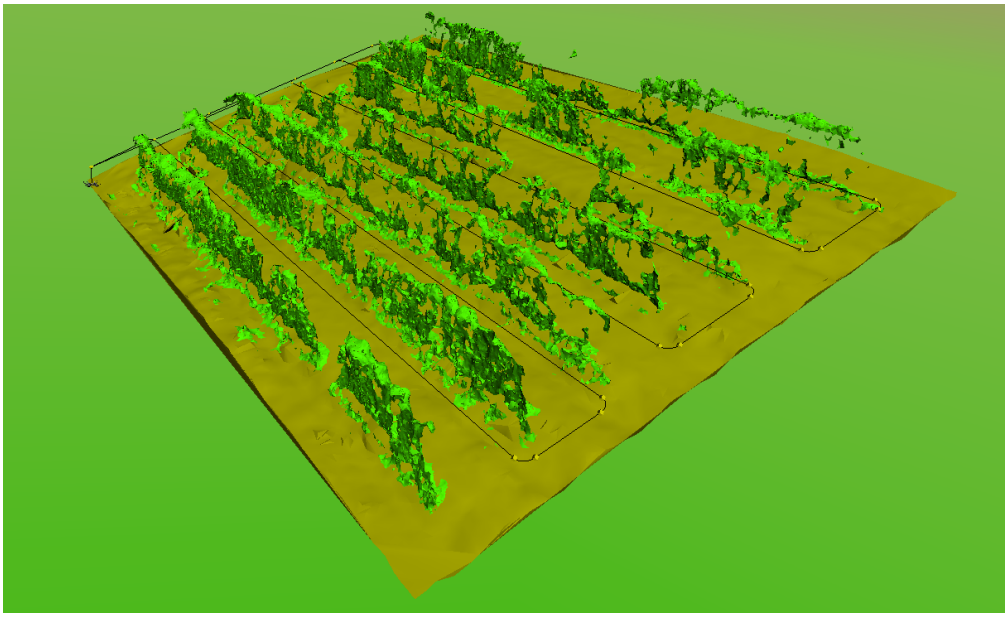

Figure 4.3: Trajectory depicted on the scenario

The lower left-hand corner has been also the starting position from where a square pattern between the vineyard rows been designed: the resultant trajectory keeps a constant altitude in sections parallel to the e_E axis, while the necessary climbs are made only in sections parallel to e_N axis. A final continuous descent has been designed from the higher left-hand corner back to the starting point for landing.

To visualize the pattern on the scenario, the `asbTrajectoryTool`, provided with the Quadcopter Project has been used, customizing the code relating to the altitude change. In Figure 4.2, the built trajectory is shown on the tool user interface while Figure 4.3 shows the pattern visualized between the vineyard rows of the scenario.

Reference signal inputs for tracking the trajectory has been sent to the quadrotor using a `cmdData.mat` file, selecting one of the options provided in the original project: this file is a time series containing a sequence of NED positions, starting from the initial location of the quadrotor, coupled with "yaw" angles and time increments of 50 ms.

This time series has been generated with a dedicated MATLAB script: to simplify its construction, it has been useful to choose the quadrotor starting orientation with the x_B axis of the UAV aligned with the e_N axis. In this way, the derivation of the movements of the quadrotor has been simplified because the position variations never happened on e_N axis and e_E axis at the same time.

NED positions and "yaw" angles time sequences are the only reference signals required in this implementation: other Euler angles and total thrust for achieving the desired positions along the pattern are controlled by the flight controller described and tuned in the next Section.

4.2 PID tuning for quadrotor operation in 3D vineyard scenario

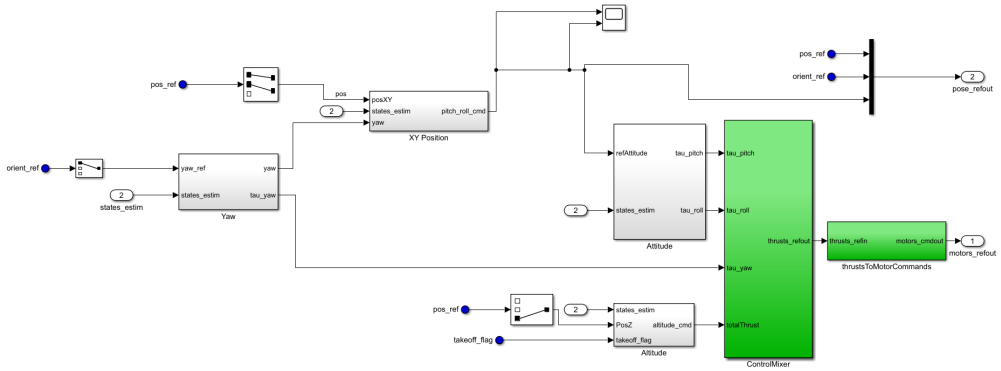

Figure 4.4: Quadcopter project flight controller.

The Quadcopter project uses a quadrotor model based on the Parrot series mini-drones, that implements various combination of proportional-integrtive-derivative (PID) modules for the flight controller (for details about PID controllers refer to Section 2.3.1).

In Figure 4.4, the block scheme of the Quadcopter project flight controller is shown and the control blocks are described below:

- Yaw: the "yaw" control block takes as input the "yaw" reference signal and "yaw" estimated state and gives as output the "yaw" command as a result of PD control;
- XY Position: the "XY position" control block takes as input the NED position reference signal and the "yaw" estimated state from the "yaw" control block, and gives as outputs the pitch and roll reference signals as a result of a P control for position and a D control for velocity;
- Attitude: the "attitude" control block takes as inputs the pitch and roll reference signals, outputs of the "XY position" control block and their estimated states, and gives as outputs "pitch" and "roll" commands as a results of a PID control for "pitch" and one for "roll";

- Altitude: the "altitude" control block takes as inputs the height reference signal and the estimated state and gives as output the total thrust command as a result of a P control for position and a D control for velocity.

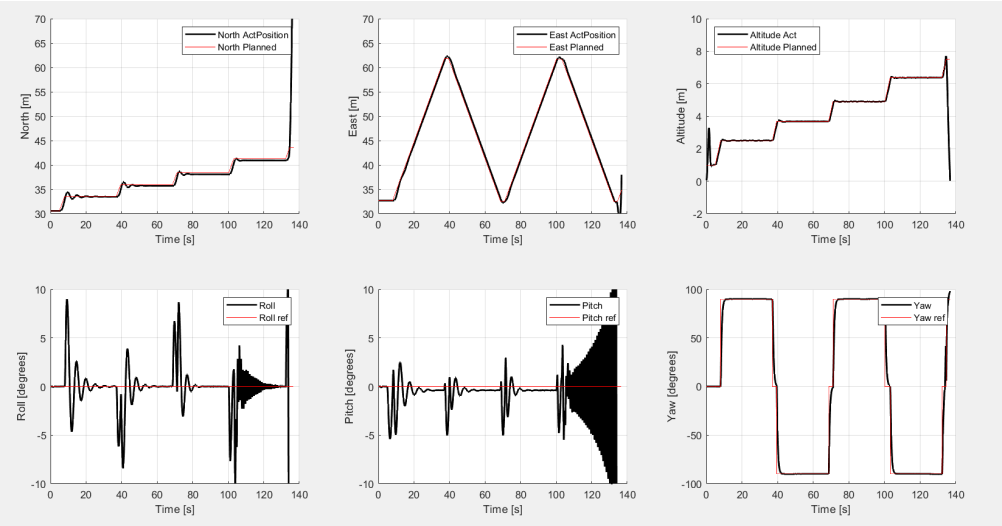

Figure 4.5: Intermediate tuning results obtained modifying the Attitude Controller parameters only.

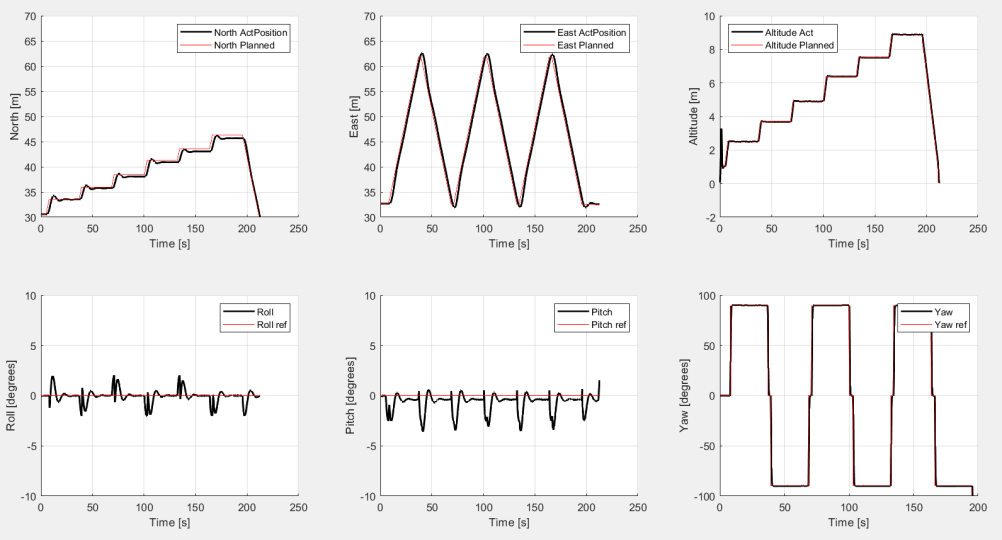

Figure 4.6: Tuning results obtained modifying both the Attitude Controller and XY Position Controller parameters.

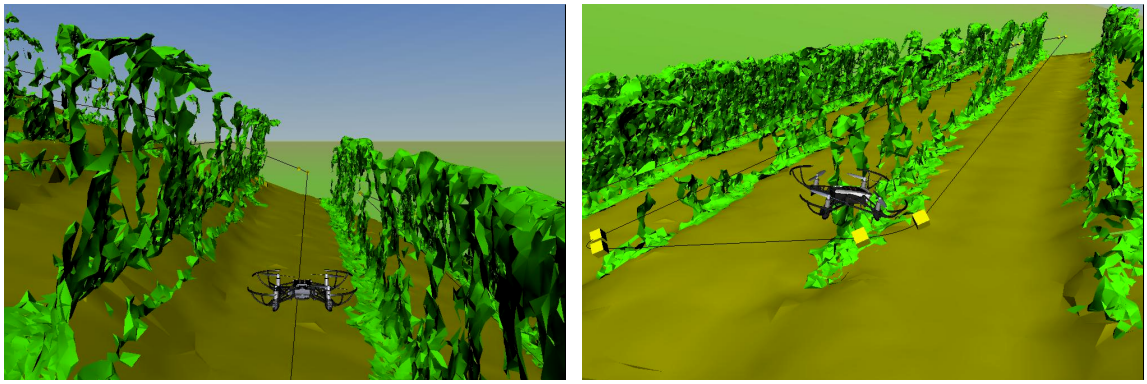

Figure 4.7: Visualization of the quadrotor flying the designed trajectory.

The tuning process started from the default values and fixing an horizontal speed of 1 meter per second along the all trajectory: with these settings, after the first 90° turn to the right, the quadrotor becomes unstable on the "pitch" and "roll" axis. The choice made for correcting this issue was to focus initially on the Attitude PID control, trying to find the best combination of its parameters leaving unaltered the others. In Figure 4.5, the best results achieved with the following parameters for the Attitude Controller are shown:

- Pitch: P=0.04293 I=0.0101 D=0.009761
- Roll: P=0.04133 I=0.0101 D=0.009529

With these settings, even if at the end of the fourth leg (West direction) the quadrotor departed from the planned trajectory, the attitude oscillations were limited and some turns were gained, before loosing control.

Considering these as the best values found and keeping them fixed, the tuning process moved to the XY Position controller. It is necessary to decrease both values, regarding respectively position and velocity, to make the quadrotor flying the planned trajectory with a good approximation:

- X: P=-0.03 dx: D= 0.035
- Y: P= 0.03 dy: D=-0.035

In general, decreasing the values in the XY Position controller allows more time for the attitude controller to absorb the perturbations and achieve the reference without any loss of control of the vehicle.

In Figure 4.6, The final trajectory tracking results are shown:

- "North" coordinate: it can be noticed that the steady state error in this coordinate slightly increases as the flight proceed: in this case, this error can be accepted but a correction it would become necessary if more rows are added to the path;

- "East" coordinate: after the turns, a small delay in the re-acquisition of the reference has to be accepted in order to not put under too much pressure the attitude controllers;

- Altitude: reference signal is well tracked even during the uphill turns, where many changes of motor powers are carried out simultaneously;

- Roll: it can be noticed that the perturbations on are essentially caused by the sharp right-angles bends, made in conjunction with the total thrust increment necessary for the climbs. The PID controller reacts well on re-acquiring the reference signals with a maximum runaway value of the command equal to approximately 2 degrees;

- Pitch: also here the perturbations happen during the turns and the behaviour is similar to the previous axis with the oscillations put under control, with a maximum runaway value of the command equal to approximately 4 degrees. The slight difference between the reference signal and the actual pitch command is due to the pitch-down attitude necessary to keep the forward speed of 1 m/s;

- Yaw: following the yaw signal during sharp turns is the action that mostly causes the perturbations on "roll" and "pitch" axis, together with total thrust increment for altitude change. It could be possible to relax the constraint relative to the two broken turns around the end of the row and make a long continuous turn as a simplification, but that was not adopted in order to look for the finest tuning values of the "roll" and "pitch" controllers.

In Figure 4.7, it is possible to see the visualization of the quadrotor flying the designed trajectory: in particular, it is shown a stretch of straight-in level flight between the vineyard rows and a right-angled left turn performed while climbing. On the other hand, in conclusion, Figure 4.8 shows the entire trajectory and the path flown by the quadrotor.

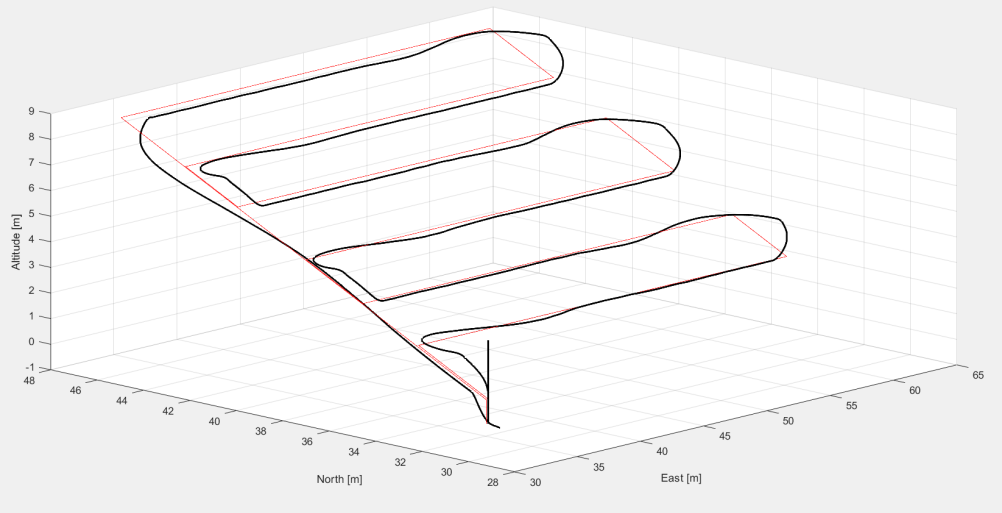

Figure 4.8: Path flown by the quadrotor controlled by PID controllers, tracking the trajectory designed starting from the 3D vineyard scenario.

4.3 LQR PIL simulations results tracking vineyard scenario-generated trajectory

As the very last step of this project, the simulation framework developed in Chapter 2 has been tested on the trajectory generated thanks to the scenario developed in Chapter 3. The main goal is not only to analyze the effectiveness of the proposed control scheme tracking the designed trajectory, but also verifying if the cross-compilation generates delays or compatibility problems on the Pixhawk 4 autpilot, described in detail in Chapter 1.

After tuning the LQR controller and fixing a horizontal speed of 2 meters per second along the all pattern, two different simulations have been performed, one as PIL simulation and one as standard simulation, exploiting the same setting parameters and comparing the results by numerical difference (for details about PIL simulations refer to Section 2.5).

Figure 4.9 and figure 4.10 represent the main outputs obtained respectively during PIL testing and standard simulation. It is possible to observe the complete adherence among PIL and standard simulation results, thus highlighting the reliability of the simulation environment and the effectiveness of the control scheme. The same result can be visualized also in Figure 4.11 where the numerical difference between PIL simulation and standard simulation is shown always equal to zero along all control channels.

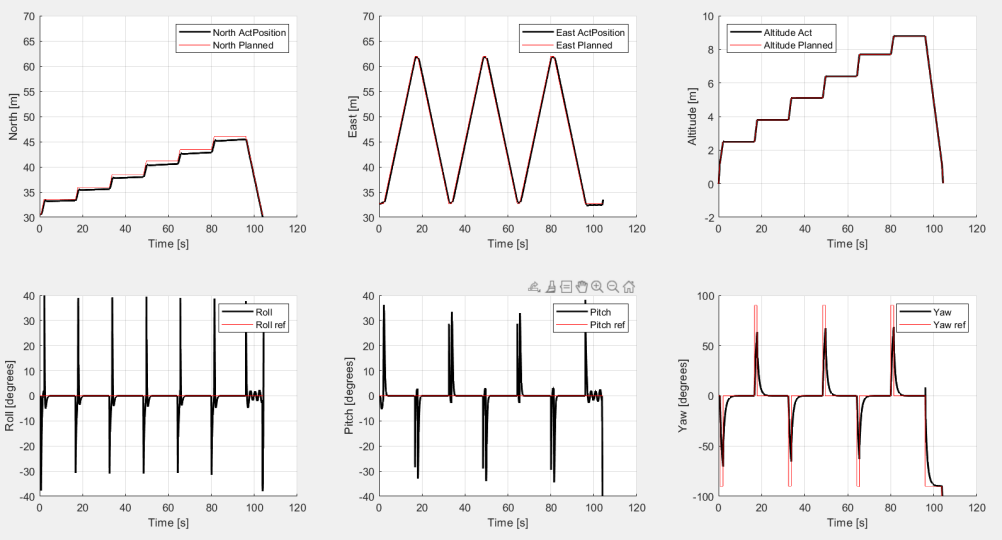

Figure 4.9: LQR PIL simulation results.

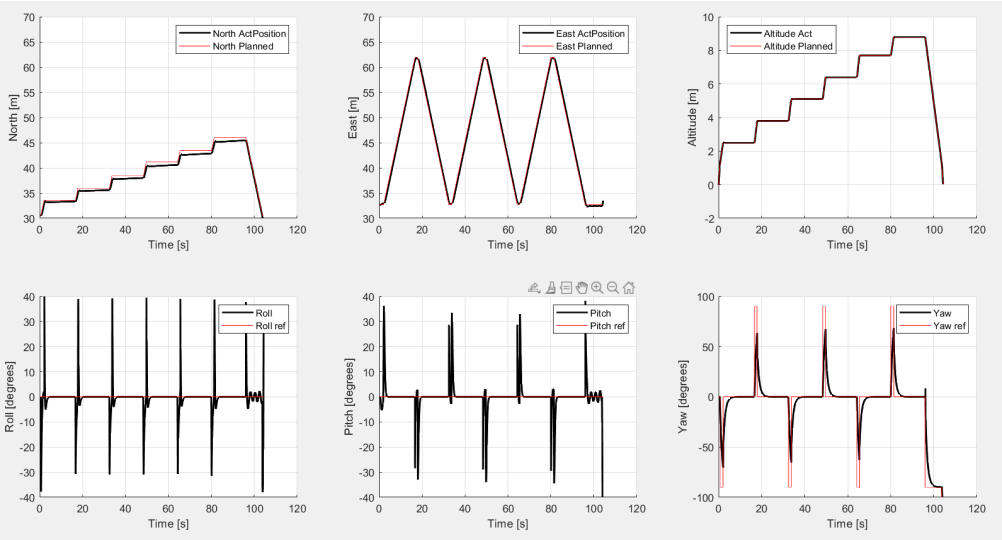

Figure 4.10: LQR standard simulation results.

Moreover, it is observable that the disturbances caused by the sharp turns are not implying any loss of control on the "pitch" and "roll" axis. This behaviour is demonstrated by the fact that the UAV always remains within reasonable limits from the reference of the designed pattern. This is also widely demonstrated in Figure 4.12, where the 3D PIL trajectory tracking is shown.

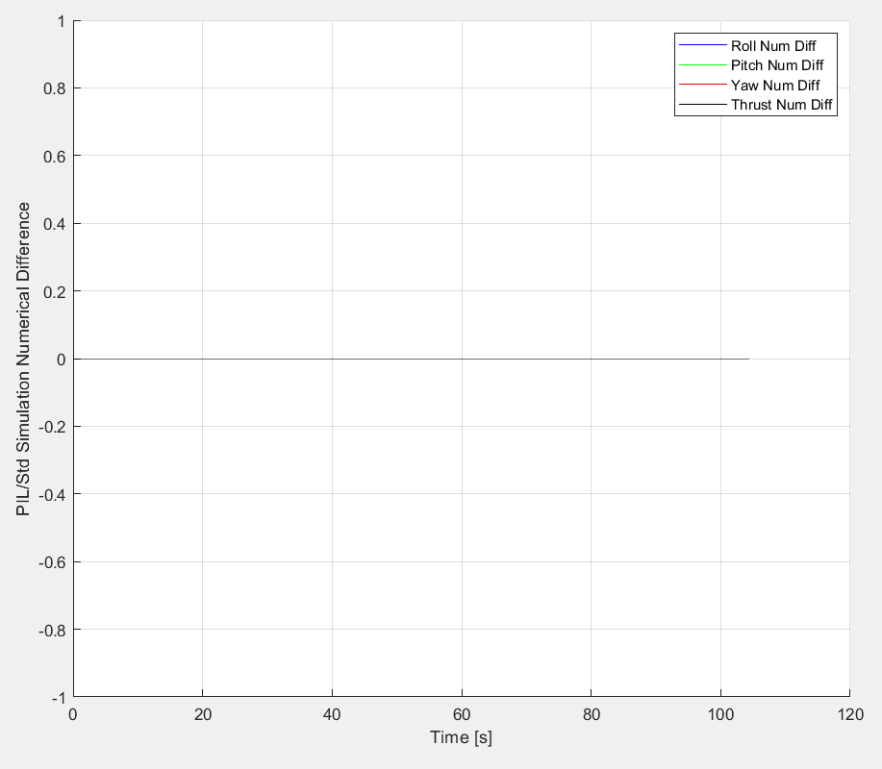

Figure 4.11: Pitch - Roll - Yaw - Total thrust numerical difference between LQR PIL simulation and LQR standard simulation.

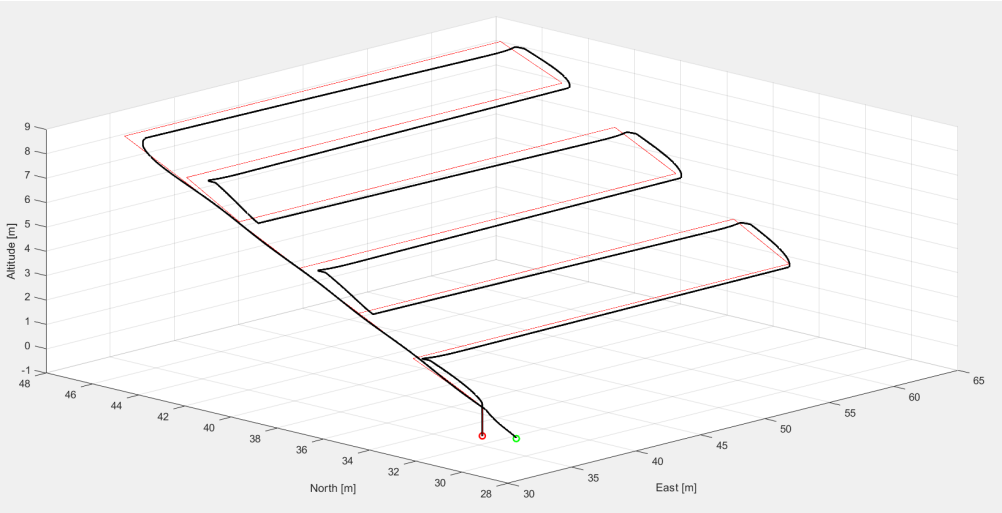

Figure 4.12: LQR PIL trajectory tracking.

Since the matrix K is calculated offline using a MATLAB function for resolving the Riccati Equation with the selected Q and R matrices (see Section 2.3.2 for details), the main task delegated to the hardware remains to compute the matrix multiplication between K and the matrix resulting from the difference between the actual state and the desired state.

The code generation of a matrix multiplication like this case is well optimized in the process and when the autopilot board executes the code, it does not generate any delay.

As already stated in Section 2.5, a "tick-rate" of 1 millisecond is a lot of time to perform operations for a 216MHz CPU (considering also a round-trip time for USB data transfer of 250 microseconds) and to stress it out we would need a lot more complex control algorithm respect to the LQR, taken into consideration in this work. This is just a general consideration: to deeply understand the actual performance of the generated code, it would be necessary to perform a rigorous code verification and profiling, as described in [96], but this is left for future works.

Chapter 5

Conclusions and Future Works

5.1 Conclusions

Aim of this project was to assess the potentialities of the Embedded Coder Support Package for PX4 autopilots for implementing customized control algorithms over the Pixhawk 4 autopilot board. It is possible to say that this purpose has been achieved: it was demonstrated that the development environment gives the possibility to cross-compile an LQR controller with automatic code generation for controlling a conventional model of quadrotor along a generated trajectory, obtaining results comparable with a standard simulation.

Unfortunately, this is not a "free meal": in fact, even if the effort on code writing and testing is highly reduced, understanding how an automatic code generator (such as the Embedded Coder) works and setting all its requested parameters requires a non-negligible learning curve and expenditure of time and resources. Moreover, replacing the standard modules in the PX4 stack with the `px4_simulink_app`, necessary step for using the development environment still causes incompatibility issues with mission planning software (such as QGroundControl), making difficult the use of them for Hardware-in-the-loop simulations, inside the current potentialities of the support package.

Second objective of the project was realizing useful virtual scenarios from LIDAR point cloud map: even if Simulink 3D Animation is not a simple tool to understand and customize, the process found for processing the point cloud is general and can be applied also to other fields where a 3D mesh is necessary.

5.2 Future works

As future works, a lot has to be done exploiting the selected hardware and many ideas were born during the work:

- implementing a more complex and advanced controller, such as Model Predictive Control (MPC) techniques presented in [20] and [97], to understand if this kind of algorithms can be used for tracking trajectories between vineyard rows, also modeling flight disturbances and performing a rigorous code profiling as described in [96];
- interfacing Simulink 3D Animation with the PIL framework, as done with FlightGear simulator, to exploit the illustrated visualization tool also for this kind of simulations, introducing also collision detection;
- removing the simplifications introduced loading the scenario into the simulation environment while considering bigger land portions and generating more complicated trajectories (also using automatic approaches, such as the one that has to be presented in [98]);
- finding a solution for performing HIL simulations (that are still not directly supported in the Support Package) and/or moving to flight tests, cross-compiling the selected controller using the PX4 Simulink blocks described in Section 1.3.

Appendix A

Guide for Implementing a Pixhawk 4 Processor-in-the-loop Simulator

This Guide has the aim to list and describe all the steps required to use the Embedded Coder Support Package for PX 4 autopilots in order to perform Processor-in-the-loop simulations, exploiting a Pixhawk 4 autopilot board. As described in Section 2.5, this type of simulations wants to exploit the real hardware in order to test advanced control algorithms on the Pixhawk 4 autopilot board.

The main reference of this guide is the MathWorks documentation, available in [38] and [40]. Internal references are Chapter 1 and Sections 2.1, 2.4 and 2.5.

This guide is based on MATLAB/Simulink 2020a release running on Windows 10 operating system: it is possible to adapt the listed steps also for Linux distributions but it not recommended to use any kind of OS virtualization tools (e.g. VirtualBox, VMware, etc.) because it causes too much overhead with the hardware communication and it can invalidate the obtained results.

A.1 Package installation and hardware setup

1. During the MATLAB setup process or, if already installed, using the Add-on Manager, download and install the following required packages:
   - Simulink;
   - Embedded Coder;

- MATLAB Coder;
- Simulink Coder;
- Aerospace Blockset,

2. At the end of the installation process, it is requested to download and install the Mingw-w64, the GCC compiler support for Windows [99]. For reference, the compiler can be retrieved at `https://www.mathworks.com/matlabcentral/fileexchange/52848-matlab-support-for-mingw-w64-c-c-compiler`.

3. Open MATLAB/Simulink and begin the installation process of the Embedded Coder Support Package for PX4 autopilots from the Add-on Manager.

4. Follow the installation instructions. Be sure to proceed with the setup of the required Windows Cygwin Toolchain version 0.5. If during the setup, the step is skipped, the toolchain can be retrieved at `https://www.mathworks.com/help/supportpkg/px4/ug/setup-cygwin-toolchain.html`. For more information about supported toolchains, refer to `https://dev.px4.io/v1.8.0/en/setup/dev_env_windows.html#other-windows-toolchains`.

5. At the end of the installation process, clone the PX4 Firmware version 1.8.0 in the same location where the toolchain has been installed: this operation can be done selecting the correct option at the end of the process or directly from `https://github.com/PX4/Firmware/tree/v1.8.0` . It is recommended to install the toolchain and clone the firmware at the default path `C:\px4_cygwin` .

6. During board selection, choose "Pixhawk 4" and the `nuttx_px5fmu-v4_default.cmake` CMake configuration. If it is needed to perform SIL tests without using any board, select PX4 Host Target and `posix_sitl_default.cmake` as CMake configuration. PX4 Host Target is, in fact, the emulated board for performing SIL simulations (refer to Section 2.4 for more details). For advanced users, there is the possibility to select a customized Cmake configuration. Once the choice is made, build the firmware by clicking on the button "Build".

7. Connect the Pixhwak 4 autopilot board to the development computer via the USB serial port.

8. Begin the hardware setup process. The process can be started from the setup option, looking for the support package from the Matlab Add-on Manager, or from the Simulink "Hardware" tab->"Hardware Board"->"Setup Hardware".

9. Select the installed Cygwin Toolchain and verify its setup. After that, veryfy the cloned PX4 firmware by clicking the appropriate button (make sure to indicate the correct location of the firmware, the default is `C:\px4_cygwin`).

10. Select the option "Design Flight Controller Algorithm in Simulink" in the "Select Application" screen.

11. For the hardware setup of the Pixhawk 4 autopilot board, it is requested to copy the `rc.txt` startup script into the `/etc` SD-card directory of the Pixhawk 4 hardware board. To perform this action, it is necessary to extract the SD card from the Pixhawk and plug it into the development computer via card reader (embedded or external), because it not directly accessible from the board. Specifically, deploying a customized flight control systems on the board, requires to suppress the execution of some default processes (in particular the Navigator and the Commander modules, refer to Section 1.3 for details). This is achieved with this start-up script: changing its contents, it is possible to choose which flight software modules have to be run or not. After the selection, the script starts the `px4_simulink_app`, "wrapper" module for executing applications generated by Simulink. Addtitional details about the system startup can be found at `https://dev.px4.io/master/en/concept/system_startup.html`.

    The script can be found at the path `C:\ProgramData\MATLAB\SupportPackages\R2020a\toolbox\target\supportpackages\px4\lib\etc`. The code of the script is also reported below for reference and example:

```
#Copyright 2020 The MathWorks, Inc.

# This is the custom rc.txt which loads px4_simulink_app on
#start-up
usleep 1000
uorb start
usleep 1000
tone_alarm start
usleep 1000
px4io start
#gps start -f   #Starts GPS driver and Fake a GPS signal
#(useful for testing)
usleep 1000
sh /etc/init.d/rc.sensors
usleep 1000
#Uncomment the below 2 lines to use LPE estimator
#attitude_estimator_q start
#local_position_estimator start
```

```
#Using EKF2 estimator by default as PX4 does build LPE on
#px4fmu-v2 due to a limited flash.
ekf2 start
usleep 1000
mtd start
set PARAM_FILE /fs/mtd_params
param select $PARAM_FILE
usleep 1000
param load
usleep 1000
rgbled start
usleep 1000
fmu mode_pwm  #This is required for AUX PWM channels
usleep 1000
px4_simulink_app start
#exit #30-Jan-2020
```

12. As the last step of the hardware setup, choose the USB COM port of the development computer for firmware upload and flash it into the board (the default selection should be the port where the board is already connected). During the process, the message shown in Figure A.1 could appear: to solve the issue unplug and plug the USB connector on the Pixhawk side of the cable to re-establish the two-way communication. The operation has success if the FMU LEDs light up: if this does not happen, repeat operation, taking care to unplug the USB connector from the Pixhawk side. In case of success, at the end of the process, it will be possible to visualize Pixhawk accelerometer data.

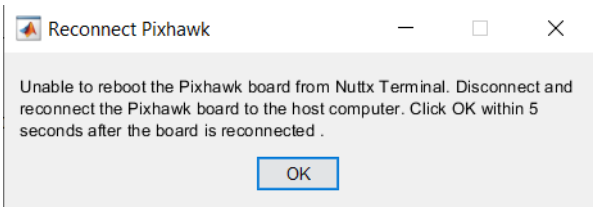

Figure A.1: USB communication error between computer and Pixhawk.

A.2 Processor-in-the loop simulation of a deployed controller subsystem

To design and simulate the flight control, the first step is to realize a Model-in-the-loop simulator as a test-bench model. It can be realized with any preferred methodology or following the template shown in Figure A.2, provided by the Support Package.

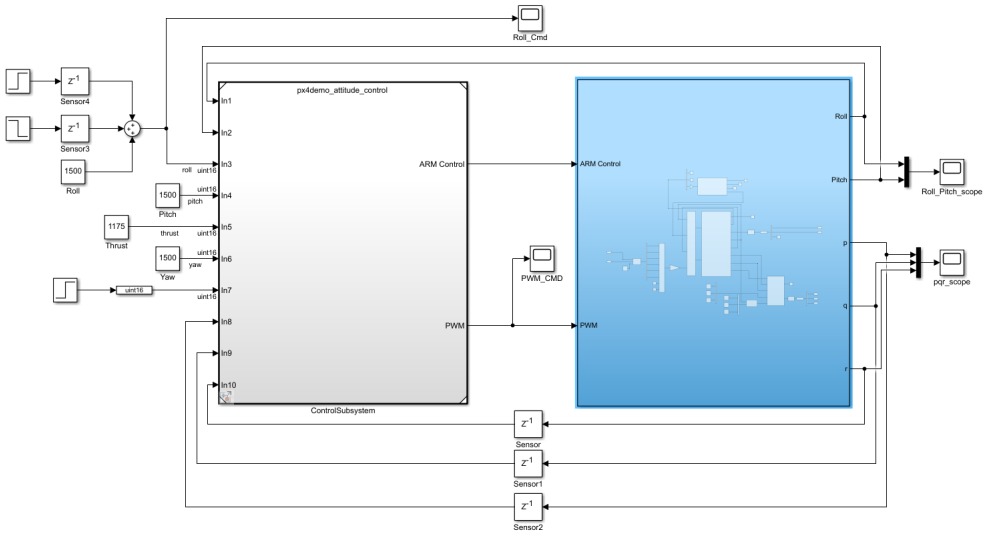

Figure A.2: PX4 Attitude Control scheme template.

After the flight control system has been successfully simulated (with the aid of a MATLAB script for the initialization of variables in the the workspace), simulator subsystems can be moved in the Processor-in-the loop model to generate code for the Pixhawk hardware. the initial Model-in-the-loop realization is out of the scope of this guide but, for reference and example, in Figure A.3 are reported the realizations of both quadrotor plant and controller, used in this project.

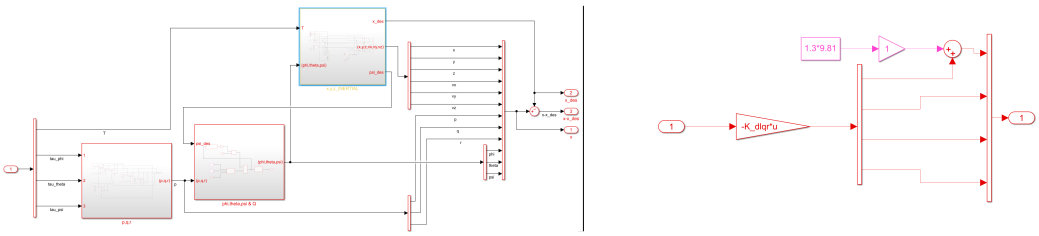

Figure A.3: Examples of quadrotor plant and controller.

Since the controller has to be cross-compiled, it is recommended to design it with blocks supported for code generation. More information about block compatibility and S-functions can be retrieved at `https://it.mathworks.com/help/releases/R2020b/ecoder/ug/supported-products-and-block-usage.html` and `https://it.mathworks.com/help/rtw/ug/s-functions-and-code-generation.html` .

After having implemented the Model-the-loop simulator, follow the following steps to realize a Processor-in-the-loop simulator:

1. Open the PIL Block model `px4demo_pil_block.slx`, available in the Support Package, that can be retrieved at the path `C:\ProgramData\MATLAB\SupportPackages\R2020a\toolbox\target\supportpackages\px4`. The template is shown in Figure A.4.

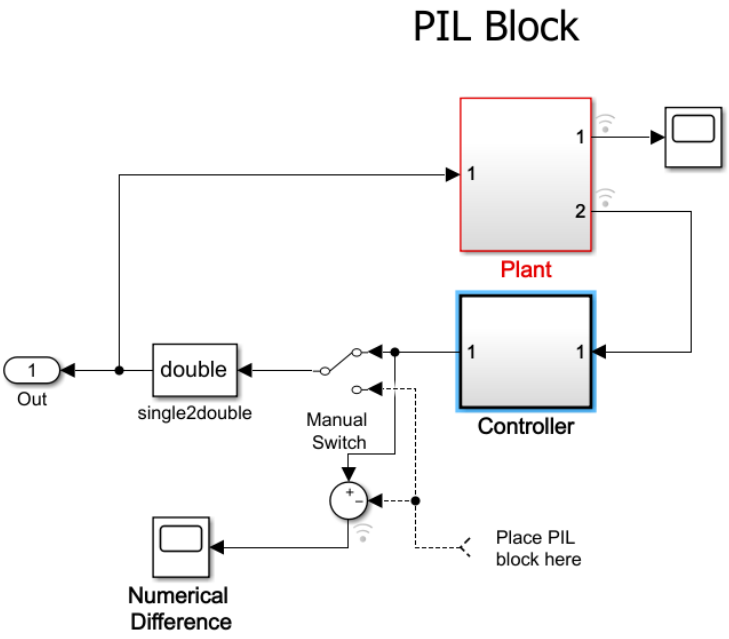

Figure A.4: PX4 PIL Block template.

2. Modify "Plant" and "Controller" blocks with own subsystems, already part of the Model-in-the-loop simulator. It is possible to customize the necessary outputs for the project. After moving the subsystem blocks, make a first simulation trial to check if everything works in the same way of the Model-in-the-loop simulator.

3. Open the "Hardware" tab and click on the "Hardware settings" button to open "Configuration Parameters" window. This window is essential for configuring all the parameters for code generation: here it is indicated the minimum configuration but it is possible to customize more parameters depending on the project needs. Select the following settings:

   - Select "Hardware Implementation" -> "Target Hardware Resources" -> "PIL". Select the hardware board serial port and enter the value of the

host serial port. The default value should be the USB COM port already used for firmware upload, during the hardware setup.

- Select "Code Generation" -> "Verification" -> "Advanced parameters" and choose PIL.

4. Go back to the model and right-click on the "Controller" subsystem and select "Deploy this Subsystem to Hardware". Creating a PIL block out of the Controller subsystem is the fundamental step for cross-compiling the controller subsystem for the execution on the hardware board. This selection is shown in Figure A.5.

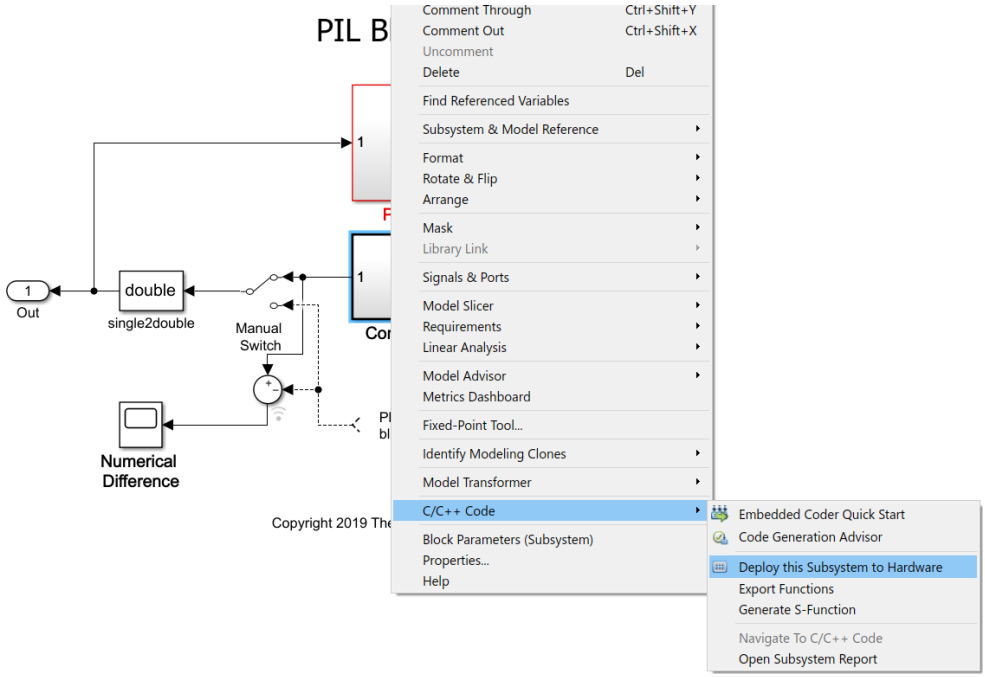

Figure A.5: Deploy subsystem to hardware selection.

5. In the next window (shown in Figure A.6), it is possible to choose tunable parameters: in this case, since the matrix K for the LQR controller was calculated offline, it has been chosen to make tunable the sample time. Before beginning the build process, be sure to connect the autopilot board to the computer and to choose a determined folder as MATLAB workspace (do not use Windows Desktop). For more details about managing and understanding the complex hierarchy of build folders, refer to `https://it.mathworks.com/help/rtw/ug/build-process-folders-.html`.

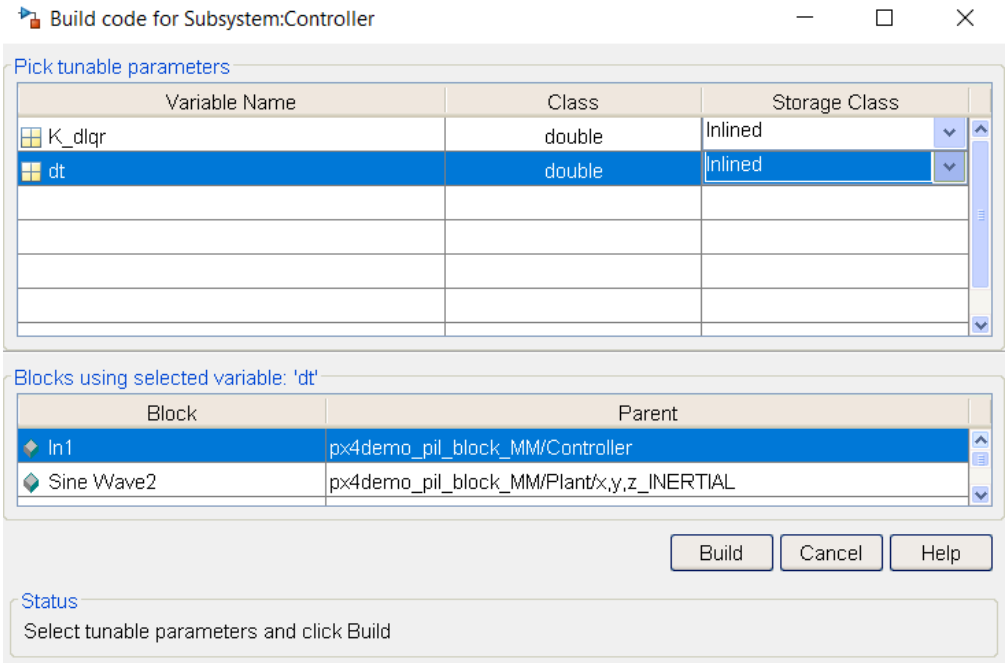

Figure A.6: LQR controller build window.

6. After the selection, click on the "Build" button to begin the code generation of the controller. If the build process is successful, the diagnostic messages in the listing below and another window with the generated PIL block are automatically shown up.

```
### Starting build procedure for: Controller
Removing old px4_simulink_app directory: C:\px4_cywin\home\
    Firmware\src\modules\px4_simulink_app.
Build path: C:\Users\lazza\Desktop\Controller_ert_rtw
### Successful completion of build procedure for: Controller
### Creating PIL block ...
### Connectivity configuration for "C:\Users\lazza\Desktop\
    Controller_ert_rtw": PX4 Autopilot ###
### COM port: COM3
### Baud rate: 3000000
Building with 'MinGW64 Compiler (C)'.
MEX completed successfully.
Build process completed successfully
```

7. The code of the generated `ert_main.cpp` is reported below for reference and example:

```
// File: ert_main.cpp
//
// Code generated for Simulink model 'Controller'.
//
// Model version                  : 1.263
// Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
// C/C++ source code generated on : Tue Nov 24 18:04:22 2020
//
// Target selection: ert.tlc
// Embedded hardware selection: ARM Compatible->ARM Cortex
// Code generation objectives: Unspecified
// Validation result: Not run
//
#include <stdio.h>
#include <stdlib.h>
#include "Controller.h"
#include "Controller_private.h"
#include "rtwtypes.h"
#include "limits.h"
#include "MW_PX4_TaskControl.h"
#include "nuttxinitialize.h"
#define UNUSED(x)                      x = x
#define NAMELEN                        16

// Function prototype declaration
void exitFcn(int sig);
void *terminateTask(void *arg);
void *baseRateTask(void *arg);
void *subrateTask(void *arg);
volatile boolean_T stopRequested = false;
volatile boolean_T runModel = true;
sem_t stopSem;
sem_t baserateTaskSem;
pthread_t schedulerThread;
pthread_t baseRateThread;
void *threadJoinStatus;
int terminatingmodel = 0;
void *baseRateTask(void *arg)
{
  runModel = (rtmGetErrorStatus(rtM) == (NULL));
  while (runModel) {
    sem_wait(&baserateTaskSem);
    step();

    // Get model outputs here
```

```
46      stopRequested = !((rtmGetErrorStatus(rtM) == (NULL)));
47      runModel = !stopRequested;
48    }
49
50    runModel = 0;
51    terminateTask(arg);
52    pthread_exit((void *)0);
53    return NULL;
54  }
55
56  void exitFcn(int sig)
57  {
58    UNUSED(sig);
59    rtmSetErrorStatus(rtM, "stopping the model");
60  }
61
62  void *terminateTask(void *arg)
63  {
64    UNUSED(arg);
65    terminatingmodel = 1;
66
67    {
68      runModel = 0;
69    }
70
71    MW_PX4_Terminate();
72
73    // Disable rt_OneStep() here
74    sem_post(&stopSem);
75    return NULL;
76  }
77
78  int px4_simulink_app_task_main (int argc, char *argv[])
79  {
80    px4_simulink_app_control_MAVLink();
81    rtmSetErrorStatus(rtM, 0);
82
83    // Initialize model
84    initialize();
85
86    // Call RTOS Initialization function, passing the "tick rate"
       as a parameter
87    nuttxRTOSInit(0.001, 0);
88
89    // Wait for stop semaphore
90    sem_wait(&stopSem);
91
92  #if (MW_NUMBER_TIMER_DRIVEN_TASKS > 0)
93
```

```
94    {
95      int i;
96      for (i=0; i < MW_NUMBER_TIMER_DRIVEN_TASKS; i++) {
97        CHECK_STATUS(sem_destroy(&timerTaskSem[i]), 0, "sem_destroy
      ");
98      }
99    }
100
101 #endif
102
103   return 0;
104 }
105
106 //
107 // File trailer for generated code.
108 //
109 // [EOF]
110 //
```

8. Place the generated PIL block where indicated in the Figure A.4 and run the simulation from Simulation tab, making sure to have initialized the workspace variables requested by "Plant" and "Controller" subsystems. The process starts with CMAke commands (that, in this case, can be found in `Controller.mk` file, while the list of the source code file use is in the `CMakelist.txt` file) for building the code to pass to `px4_simulink_app` module for the execution.

   The object code is flashed to the board and the application started for performing the simulation. The results of the PIL simulation are transferred to Simulink to verify if a numerical difference exists between the standard simulation and the PIL simulation results. Double click on the "Numerical Difference" block to see the difference between the simulated Controller subsystem and the PIL block running on the board. It is possible to switch between the original block and "PIL" block by double clicking on the "Manual Switch" block.

   The PIL verification process is a crucial part of the development cycle to ensure that the behavior of the deployment code matches the design.

   Incidentally, to run the Simulink model using by using PIL and having MAVLink communication protocol also enabled (for starting NSH console or other debugging purposes), it is necessary to choose another USB COM port for this additional connection (for example, choose `/dev/ttyS6`, under "Target hardware resources" -> "External mode"). In this case, to connect another USB cable to board, a serial-to-USB converter is needed.

   Below is reported the diagnostic messages listing of the PIL simulation execution.

```
### Connectivity configuration for "C:\Users\lazza\Desktop\Controller_ert_rtw": PX4 Autopilot ###
### COM port: COM3
### Baud rate: 3000000
### Connectivity configuration for "C:\Users\lazza\Desktop\Controller_ert_rtw": PX4 Autopilot ###
### Preparing to start PIL block simulation: px4demo_PIL_final_2020a/Controller1 ...
ninja: Entering directory `/cygdrive/c/px4_cywin/home/Firmware/build/nuttx_px4fmu-v5_default'
[0/1] Re-running CMake...
-- PX4 VERSION: v1.8.0
-- CONFIG: nuttx_px4fmu-v5_default
-- Build Type: MinSizeRel
CMake Deprecation Warning at /usr/share/cmake-3.6.2/Modules/CMakeForceCompiler.cmake:79 (message):
  The CMAKE_FORCE_C_COMPILER macro is deprecated.  Instead just set
  CMAKE_C_COMPILER and allow CMake to identify the compiler.
Call Stack (most recent call first):
  cmake/toolchains/Toolchain-arm-none-eabi.cmake:31 (cmake_force_c_compiler)
  build/nuttx_px4fmu-v5_default/CMakeFiles/3.6.2/CMakeSystem.cmake:6 (include)
  CMakeLists.txt:176 (project)
CMake Deprecation Warning at /usr/share/cmake-3.6.2/Modules/CMakeForceCompiler.cmake:93 (message):
  The CMAKE_FORCE_CXX_COMPILER macro is deprecated.  Instead just set
  CMAKE_CXX_COMPILER and allow CMake to identify the compiler.
Call Stack (most recent call first):
  cmake/toolchains/Toolchain-arm-none-eabi.cmake:37 (cmake_force_cxx_compiler)
  build/nuttx_px4fmu-v5_default/CMakeFiles/3.6.2/CMakeSystem.cmake:6 (include)
  CMakeLists.txt:176 (project)
-- C compiler: arm-none-eabi-gcc.exe (GNU Tools for Arm Embedded Processors 7-2017-q4-major) 7.2.1 20170904 (release) [ARM/embedded-7-branch revision 255204]
-- C++ compiler: arm-none-eabi-g++.exe (GNU Tools for Arm Embedded Processors 7-2017-q4-major) 7.2.1 20170904 (release) [ARM/embedded-7-branch revision 255204]
-- PX4 ECL: Very lightweight Estimation & Control Library v0.9.0-553-g1a11068
-- Building and including px4io-v2
-- Using C++03
-- Release build type: MinSizeRel
-- Adding UAVCAN STM32 platform driver
```

```
-- NuttX: px4fmu-v5 nsh cortex-m7
-- ROMFS: px4fmu_common
-- Configuring done
-- Generating done
-- Build files have been written to: /cygdrive/c/px4_cywin/home/Firmware/build/nuttx_px4fmu-v5_default
[1/17] Building CXX object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/xil_interface.cpp.obj
[2/17] Building CXX object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/pil_main_px4.cpp.obj
../../src/modules/px4_simulink_app/pil_main_px4.cpp: In function 'void* baseRateTask(void*)':
../../src/modules/px4_simulink_app/pil_main_px4.cpp:24:34: warning: invalid conversion from 'void*' to 'void**' [-fpermissive]
     errorCode = xilInit(argc, argv);
                                  ^

In file included from ../../src/modules/px4_simulink_app/pil_main_px4.cpp:4:0:
../../src/modules/px4_simulink_app/xil_interface_lib.h:22:37: note:   initializing argument 2 of 'XIL_INTERFACE_LIB_ERROR_CODE xilInit(int, void**)'
 extern XIL_INTERFACE_LIB_ERROR_CODE xilInit(const int argc,
                                     ^~~~~~
../../src/modules/px4_simulink_app/pil_main_px4.cpp:44:1: warning: no return statement in function returning non-void [-Wreturn-type]
 }
 ^
[3/17] Building CXX object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/MW_PX4_TaskControl.cpp.obj
[4/17] Building CXX object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/MW_PX4_rtiostream_serial.cpp.obj
../../src/modules/px4_simulink_app/MW_PX4_rtiostream_serial.cpp: In function 'int rtIOStreamOpen(int, void**)':
../../src/modules/px4_simulink_app/MW_PX4_rtiostream_serial.cpp:528:5: warning: 'result' may be used uninitialized in this function [-Wmaybe-uninitialized]
   if (result == RTIOSTREAM_ERROR) {
   ^~
../../src/modules/px4_simulink_app/MW_PX4_rtiostream_serial.cpp:455:9: note: 'result' was declared here
   int result;
       ^~~~~~
[5/17] Building CXX object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/Controller.cpp.obj
```

```
[6/17] Building CXX object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/nuttxinitialize.cpp.obj
[7/17] Building CXX object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/Controller_data.cpp.obj
[8/17] Building C object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/xil_services.c.obj
[9/17] Building C object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/xilcomms_rtiostream.c.obj
[10/17] Building C object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/xil_rtiostream.c.obj
[11/17] Building C object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/rtiostream_utils.c.obj
[12/17] Building C object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/xil_data_stream.c.obj
[13/17] Building C object src/modules/px4_simulink_app/CMakeFiles/modules__px4_simulink_app.dir/xil_interface_lib.c.obj
../../src/modules/px4_simulink_app/xil_interface_lib.c:371:6: warning: no previous prototype for 'xilProcessMsg' [-Wmissing-prototypes]
 void xilProcessMsg(void) {
      ^~~~~~~~~~~~
[14/17] Linking CXX static library src/modules/px4_simulink_app/libmodules__px4_simulink_app.a
[15/17] Linking CXX executable nuttx_px4fmu-v5_default.elf
[16/17] Generating ../../px4fmu-v5.bin
[17/17] Creating /cygdrive/c/px4_cywin/home/Firmware/build/nuttx_px4fmu-v5_default/px4fmu-v5_default.px4
### Starting application: 'Controller_ert_rtw\pil\Controller.px4'
Using COM3 for upload.
Loaded firmware for 32,0, size: 1574632 bytes, waiting for the bootloader...
Attempting reboot on COM3 with baudrate=57600...
If the board does not respond, unplug and re-plug the USB connector.
Found board 32,0 bootloader rev 5 on COM3
ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
    ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
     ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
    ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
     ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
    ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
     ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
    ffffffff ffffffff family: STM32F7[6|7]x
revision: Z
flash 2064384


```

```
Erase   : [                         ] 0.0%
Erase   : [=                        ] 5.6%
Erase   : [==                       ] 11.3%
Erase   : [===                      ] 16.9%
Erase   : [====                     ] 22.6%
Erase   : [=====                    ] 28.3%
Erase   : [======                   ] 33.9%
Erase   : [=======                  ] 39.6%
Erase   : [========                 ] 45.3%
Erase   : [=========                ] 50.9%
Erase   : [==========               ] 56.6%
Erase   : [===========              ] 62.2%
Erase   : [============             ] 67.9%
Erase   : [=============            ] 73.5%
Erase   : [==============           ] 79.1%
Erase   : [===============          ] 84.7%
Erase   : [=========================] 100.0%

Program : [                         ] 4.1%
Program : [=                        ] 8.2%
Program : [==                       ] 12.3%
Program : [===                      ] 16.4%
Program : [====                     ] 20.5%
Program : [====                     ] 24.6%
Program : [=====                    ] 28.7%
Program : [======                   ] 32.8%
Program : [=======                  ] 36.9%
Program : [========                 ] 41.0%
Program : [=========                ] 45.1%
Program : [=========                ] 49.2%
Program : [==========               ] 53.3%
Program : [===========              ] 57.4%
Program : [============             ] 61.4%
Program : [=============            ] 65.5%
Program : [============             ] 69.6%
Program : [=============            ] 73.7%
Program : [==============           ] 77.8%
Program : [===============          ] 81.9%
Program : [================         ] 86.0%
Program : [=================        ] 90.1%
Program : [================         ] 94.2%
Program : [==================       ] 98.3%
Program : [=========================] 100.0%

Verify  : [                         ] 1.0%
Verify  : [=========================] 100.0%
Rebooting.
}
```

Appendix B

Guide for Creating a 3D Virtual Scenario from a LIDAR Point Cloud Map

This Guide aims at generating 3D virtual scenarios, in particular for simulating UAVs precision agriculture operations, exploiting the 3D point cloud maps, collected by LIDAR survey campaigns.

After the creation, it is also briefly described in this Guide how to load the scenarios into the selected simulator.

The steps for realizing the virtual scenario are summarized in the process shown in Figure B.1. The software tools used for the process are:

- ConveRgo [73]
- CloudCompare [74]
- MATLAB
- Meshlab [75]
- Blender [93]
- Simulink 3D Animation [71]

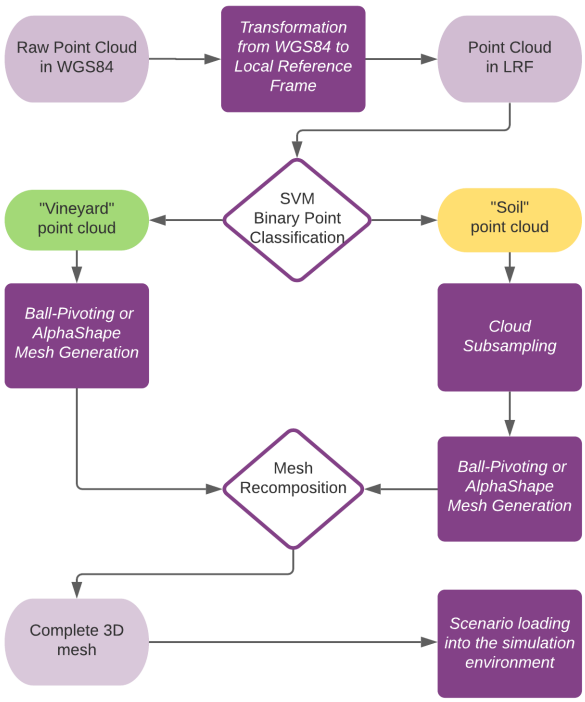

Figure B.1: 3D scenario development overview.

B.1 Reference system conversion

Since the data taken into consideration in this project (see Section 3.3 for details) are generated with the Agisoft PhotoScan software from a set of aerial images and saved in `.las` file in WGS84 reference system, it is necessary to convert them in a Local Reference Frame for the processing with the listed software tools.

1. Open the point cloud `.las` file with CloudCompare software tool.
2. Opening the file, a prompt window requests to choose a translation to apply: keep the default settings and choose "Apply to all".
3. The point cloud will be visualized as a long strip without showing any kind of scenario. Select "File"->"Save File" and save the point cloud in `.txt` format file. When the prompt window appears for choosing saving settings, select the precision requested, tabulation as separator and "ASC" for order selection.
4. After the conversion is finished, open the resultant `.txt` file with ConveRgo software tool: in this Guide, all the ConferGo potentialities will not be explained but, if necessary, refer to [80] for more details about it. In Figure B.2 is represented the user interface of the tool. On the left side, the features

of the input file are indicated: they depend on the nature of the file in input and should be acquired from who has done the survey; in this case, ETRF89 is selected to indicate that the data are saved in WGS84 reference system.

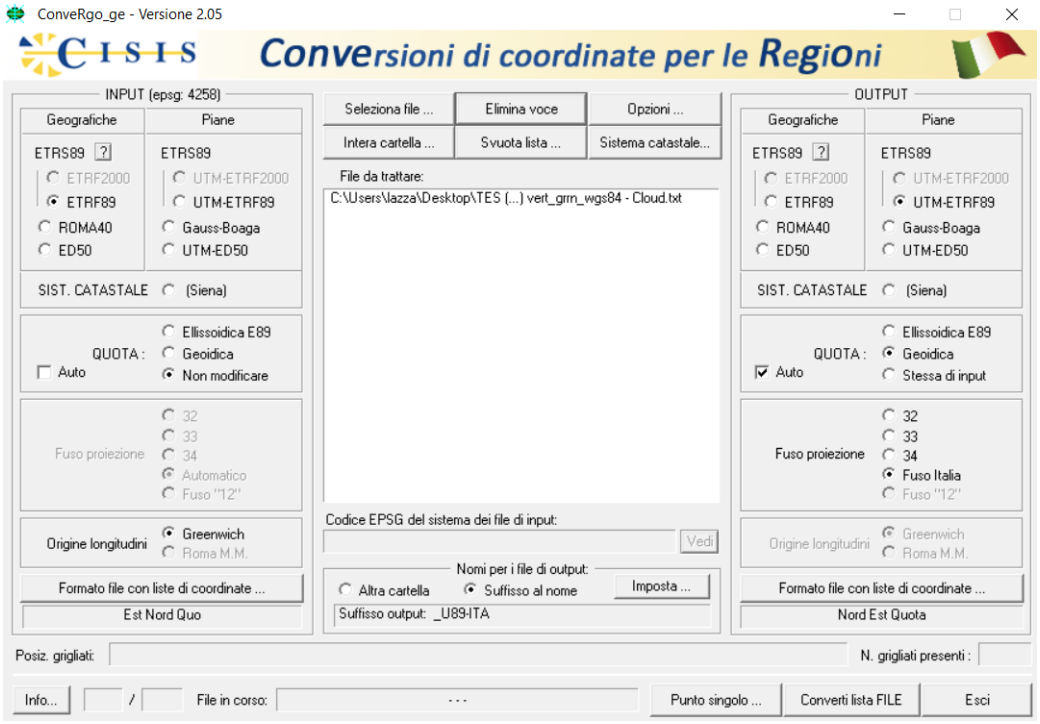

Figure B.2: ConveRgo reference system conversion tool.

On the right side, there are the characteristics of the file in output: choose UTM-ETRF2000 for "Piane" coordinates and "Stessa di Input" for altitude. Click on the "Formato file con liste di coordinate" button: in the new window select the options indicated in Figure B.3.

5. Click "Converti Lista dei file" button. Longitude and latitude will be converted in North and East coordinates in Local Reference Frame. Unfortunately this type of conversion does not preserve the color information and other feature that could be associated to every point in the original `.las` point cloud file.

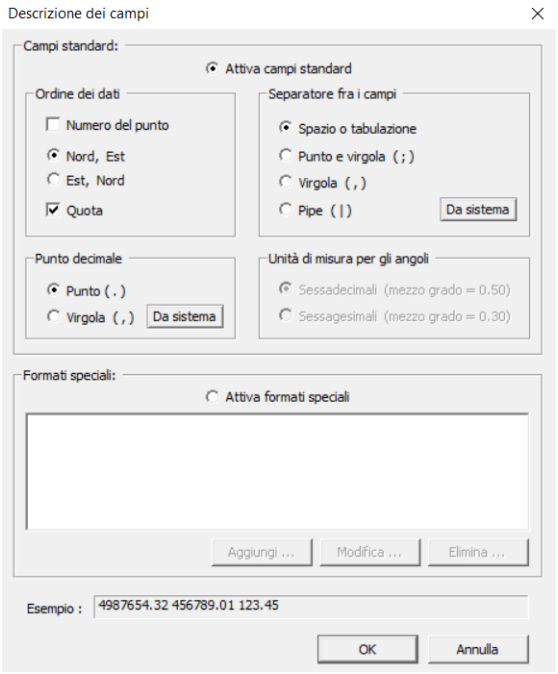

Figure B.3: ConveRgo settings.

B.2 Point cloud classification and separation

The `.txt` point cloud now obtained contains three columns with North East Altitude coordinates. Now it is necessary to classify the points to discriminate if they belong to the vegetation or to the soil. This is done mostly to generate a correct and lightweight 3D mesh. For point classification, a plugin for CloudCompare called CANUPO is exploited: for more details about it, refer to Section 3.4.

1. Open the point cloud `.txt` file with CloudCompare software tool. Opening the file, a prompt window request to choose a translation: now it is possible to keep the default settings or choose a customized translation, that can be useful for the project. In Figure **??** is shown the point cloud after reference system conversion.

2. For using CANUPO for point classification, it is necessary to select two class samples: this can be done using the "Segmentation" tool available in CloudCompare toolbar or via "Edit"->"Segment". Once the selection is made, save the two samples in two separate point cloud `.txt` files. In Figure B.4 are shown the samples used for this project.

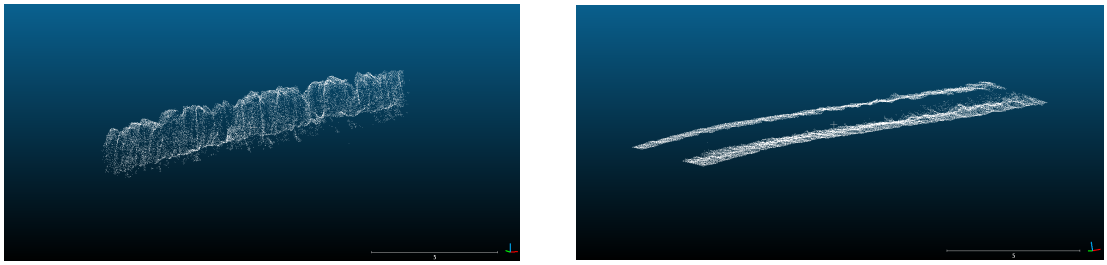

Figure B.4: "Vineyard" and "Soil" samples.

3. Open CANUPO from "Plugin"->"CANUPO"->"Train classifier": select the two class samples and the parameters for classification as shown in Figure B.5 (see Section 3.4 for classification details). After the selection, click "Ok" for building the classifier.

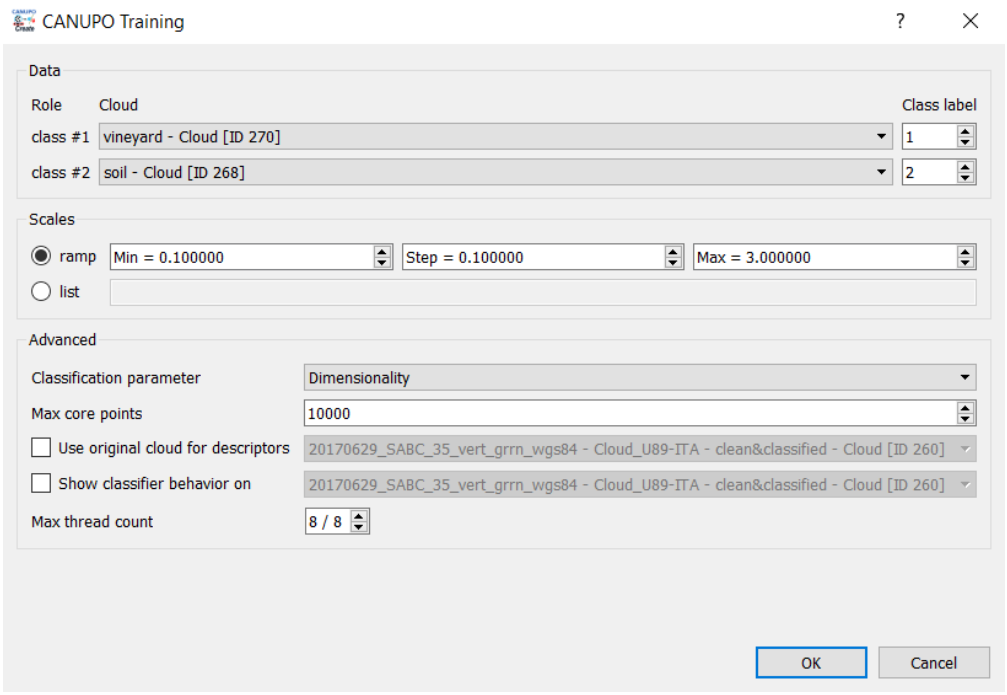

Figure B.5: CANUPO Training settings.

4. Start the classification process by selecting "Plugin"->"CANUPO"->"Classify": select the created `.prm` classifier file and the cloud to classify, then click "Ok". The prompt window indicates the clusters, separated by the hyerplane: it is possible to adjust the position of the hyperplane to acquire a better separation of the clouds. Click "Ok" to apply the classification. The result of

the classification is shown in Figure B.6.

5. If necessary, use the "Segmentation" tool to clean the cloud and/or to select a portion of it. In Figure B.7 is shown a portion of the point cloud, selected with the "Segmentation" tool.

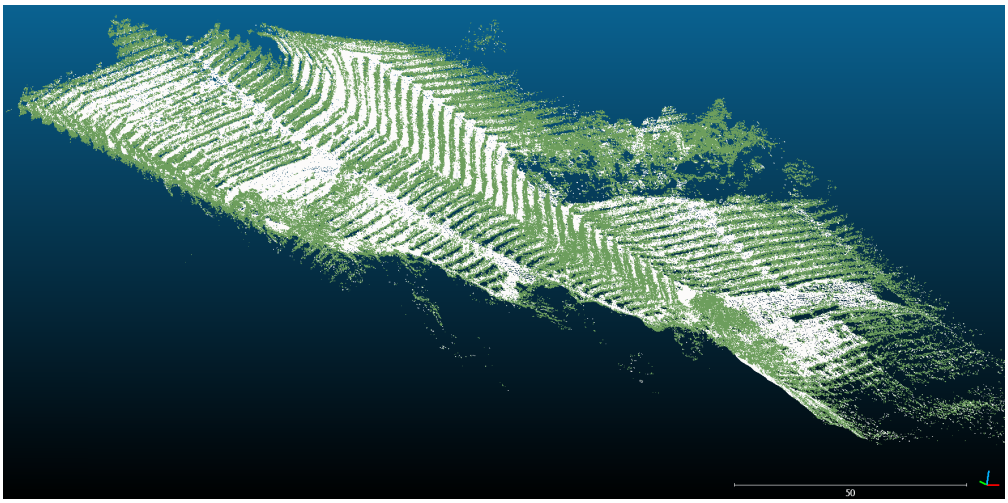

Figure B.6: Classified point cloud map.

6. Once the cloud is classified, it is also possible to separate it per class: in doing this separation, it is possible to exploit the color assigned to each classified point. This color can be customized from the "Properties" window -> "Color scale" section. For splitting, Select "Edit"->"Scalar Fields"->"Filter by value" and, in the next window, 1.0 and 1.5 values. Click "Split" button to create two different clouds based on the CANUPO classification.

7. If necessary, apply a different subsampling for each cloud: in the case of this project a subsampling has been applied to the "soil" point cloud the ease the correct mesh generation of that portion. Subsampling can be achieved selecting "Edit"->"Subsampling" and then choosing the minimum distance between each point.

8. Save the point clouds in two different files: if it is intended to use Meshlab software tool for mesh generation, it is recommended to perform the next step and then to choose `.ply` format for saving the files in order to guarantee compatibility.

9. Go to "Edit"-> "Edit global shift/scale". In this window, it is possible to see the translation applied to the point cloud during the import into CloudCompare (the settings chosen at Step 1 of this Section). Take note of these numbers,

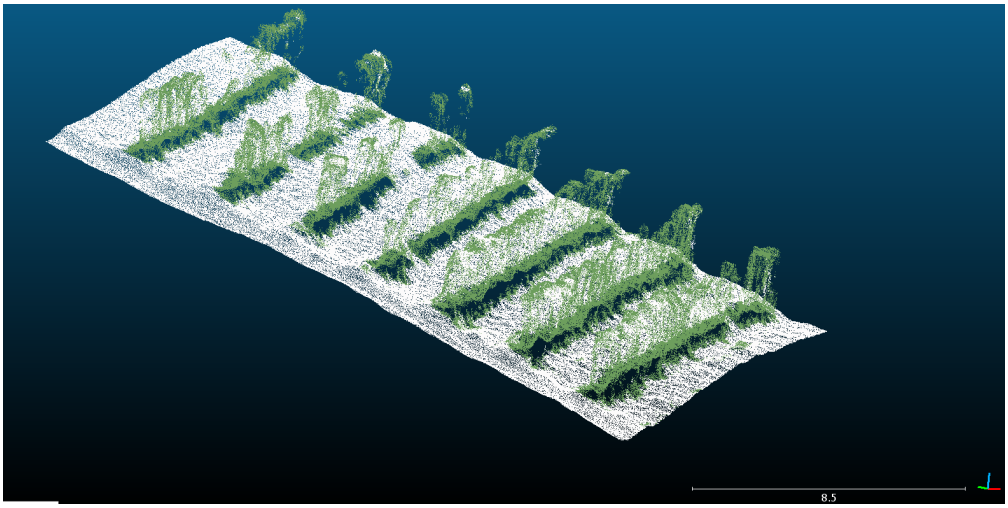

Figure B.7: Classified point cloud map portion.

go to "Edit"->"Apply transformation" and insert them in the "Translation" section. Click "Ok" and then proceed in saving the two files. This action has the purpose of reducing coordinates number sizes that Meshlab has to handle for doing calculations, reducing the processing time for mesh generation, explained in the next Section.

B.3 Mesh generation

For mesh generation, two approaches with two different software tools are explored in this Guide: the first is the use of the AlphaShape algorithm, used in the MATLAB commercial software, the second is the processing with Ball-Pivoting algorithm, implemented in the Meshlab open-source software (for more information about the algorithms, refer to Section 3.5).

- MATLAB

  1. Import in the MATLAB workspace the first three columns, representing North-East-Altitude coordinates, from the point cloud `.txt` file using the preferred technique (for example, refer to `https://it.mathworks.com/help/matlab/import_export/ways-to-import-text-files.html` and save the the into a matrix variable.

2. Apply the following MATLAB script for generating the 3D mash with Alphashape algorithm and the selected α parameter:

```
%V is the matrix containing the coordinates of the points
shp=alphaShape(V);
%Choose the necessary Alpha parameter for the detail
    granularity
shp.Alpha=0.35;
plot(shp);
[A,B]=boundaryFacets(shp);
%Export the mesh in .stl format file
stlwrite(triangulation(A,B), 'mesh.stl');
```

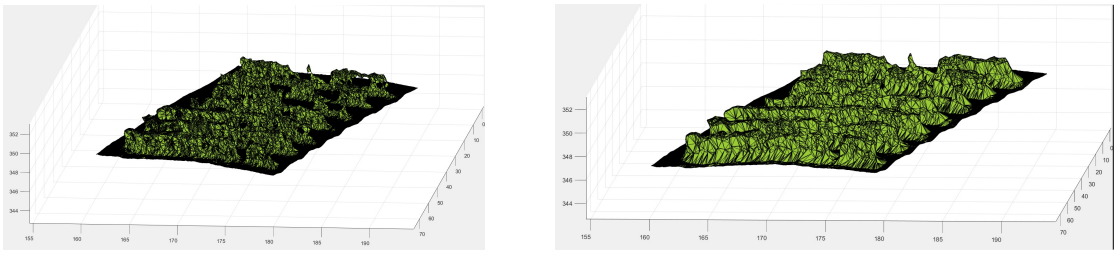

Figure B.8: AlphaShape mesh generation with α=0.35 and α=0.7 respectively (point cloud map portion without splitting).

- Meshlab

1. Open the `.ply` format file (or another compatible format) containing the point cloud.
2. Select "Filters"->"Remeshing, Simplification and Reconstruction" -> "Surface Reconstruction: Ball Pivoting". Then, In the prompt window, select the parameters for mesh generation: it is possible to leave the default settings or try to find the best combination, remembering the there is no "undo" command in Meshlab, so every combination originates a permanent result.
3. If, after the generation, the mesh has "holes" the function "Close holes" can be applied. For selecting it, go to "Filters"->"Remeshing, Simplification and Reconstruction" -> "Close holes".
4. Save the obtained result in `.stl` format file.

B.4 Mesh recomposition and loading into the simulation environment

For this project, it has been necessary to recompose the complete mesh from the two portions obtained from cloud classification: for this scope, Blender graphic editor has been used, even if Meshlab also has the same potentialities.

1. Open Blender and select "File->"Import"->"Stl (.stl)" to import the `.stl` meshes taken into consideration in the scenes hierarchy.
2. To apply the simplifications seen in this project, move the mesh in the position shown in Figure B.9. This can be done highlighting the meshes and move them with the short-cut commands ctrl+R (Rotation) and ctrl+G (Translation) or using the "Transform" window.

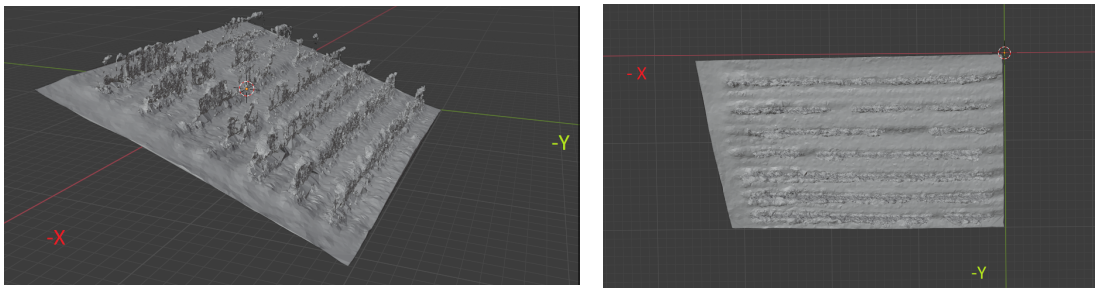

Figure B.9: 3D mesh position in Blender graphic editor to obtain a correct loading in Simulink 3D animation.

3. Once the meshes are in the desired position, export them as a single `.stl` file by selecting "File"->"Export"->"Stl (.stl)". This file format has been chosen to be loaded in a `.wrl` template scenario for Simulink 3D Animation. It is possible to choose other formats for other simulation environment (e.g. Collada for Gazebo). When exporting the `.stl` for Simulink 3D animation, make sure to choose Y-up/Z-ahead options: Simulink 3D animation, in fact, works with Y-up worlds (this is a graphic convention and does not affect reference systems for simulated dynamics).
4. Create a new `.wrl` scenario or open an already existing one with 3D World Editor, the Simulink 3D Animation editor (for this project it has been used a template scenario available in the Quadcopter Project [95]).

5. Select "Nodes"->"Import from.."-> "STL file..." and choose the `.stl` file exported from Blender. When imported, meshes composing the scenario remain separated: in this way it is possible to select each one of them for customization (refer to 3.6 for other details). In Figure B.10, the `.stl` meshes imported in 3D World Editor are shown and it is possible to observe the selection to change the color of the single mesh.

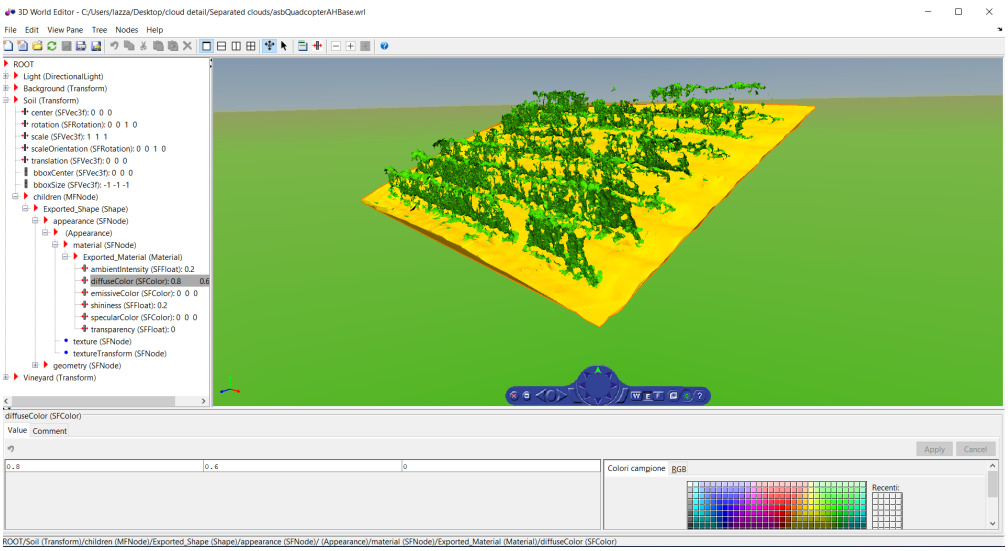

Figure B.10: 3D scenario loaded into 3D World Editor.

Bibliography

[1] V. Mazzia L. Comba A. Khaliq M. Chiaberge P. Gay. «UAV and Machine Learning Based Refinement of a Satellite-Driven Vegetation Index for Precision Agriculture». In: *Sensors* 20 (2020) (cit. on p. 1).

[2] D. Rose and J. Chilvers. «Agriculture 4.0: responsible innovation in an era of smart farming». In: *Frontiers in Sustainable Food Systems* 2 (2018), p. 87 (cit. on p. 1).

[3] G. Sylvester. *E-agriculture in action: Drones for agriculture*. Food, Agriculture Organization of the United Nations, and International Telecommunication Union, 2018 (cit. on p. 1).

[4] *Six Ways Drones Are Revolutionizing Agriculture*. MIT Technology Review. URL: https://www.technologyreview.com/2016/07/20/158748/six-ways-dron%20es-are-revolutionizing-agriculture/ (cit. on p. 2).

[5] G. Sona et al. «UAV Multispectral Survey to Map Soil and Crop for Precision Farming Applications». In: *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences (ISPRS)* XLI-B1 (2016), pp. 1023–1029 (cit. on p. 2).

[6] B. Faiçal et al. «An adaptive approach for UAV-based pesticide spraying in dynamic environments». In: *Computers and Electronics in Agriculture* 138 (2017), pp. 210–223 (cit. on pp. 2, 3).

[7] X. Xue et al. «Develop an unmanned aerial vehicle based automatic aerial spraying system». In: *Computers and Electronics in Agriculture* 128 (2016), pp. 58–66 (cit. on p. 2).

[8] J. Primicerio G. Caruso L. Comba A. Crisci P. Gay S. Guidoni L. Genesio D. Ricauda Aimonino F. Vaccari. «Individual plant definition and missing plant characterization in vineyards from high-resolution UAV imagery». In: *European Journal of Remote Sensing* 50 (2017) (cit. on p. 2).

[9] R. R. Shamshiri et al. «Fundamental Research on Unmanned Aerial Vehicles to Support Precision Agriculture in Oil Palm Plantations». In: 2018 (cit. on p. 2).

[10] N. Waskitho. «Unmanned aerial vehicle technology in irrigation monitoring». In: *Advances in Environmental Biology* 9 (2015), pp. 7–10 (cit. on p. 2).

[11] E. Puig Garcia et al. «Assessment of crop insect damage using unmanned aerial systems: A machine learning approach». In: *21st International Congress on Modelling and Simulation*. 2015, pp. 1420–1426 (cit. on p. 2).

[12] B. Vroegindeweij et al. «Autonomous Unmanned Aerial Vehicles for Agricultural Applications». In: *EurAgEng Zurich*. 2014 (cit. on p. 3).

[13] M. Mammarella L. Comba A. Biglia F. Dabbene P. Gay. «Cooperative Agricultural Operations of Aerial and Ground Unmanned Vehicles». In: *IEEE International Workshop On Metrology for Agriculture and Forestry*. 2020 (cit. on pp. 4, 43).

[14] P. Gay L. Comba J. Primicerio D. Ricauda Aimonino. «Vineyard detection from unmanned aerial systems images». In: *Computers and Electronics in Agriculture* 114 (2015), pp. 78–87 (cit. on p. 4).

[15] L. Comba A. Biglia D. Ricauda Aimoino P. Gay. «Unsupervised detection of vineyards by 3D point-cloud UAV photogrammetry for precision agriculture». In: *Computers and Electroincs in Agricolture* 155 (2018), pp. 84–95 (cit. on pp. 4, 43, 46).

[16] L. Comba A. Biglia D. Ricauda Aimoino C. Tortia E. Mania S. Guidoni P. Gay. «Leaf Area Index evaluation in vineyards using 3D point clouds from UAV imagery». In: *Precision Agricolture* 21 (2020), pp. 881–896 (cit. on p. 4).

[17] W.Z. Fum. *Implementation of Simulink controller design on Iris+ quadrotor*. Naval Post Graduate School, 2015. URL: `https://calhoun.nps.edu/handle/10945/47258` (cit. on pp. 4, 8, 12, 19, 31).

[18] A. Koszewnik. «The parrot UAV controlled by PID controllers». In: *Acta Mechanica et Automatica* 8 (2014) (cit. on p. 4).

[19] A. Salih et al. «Flight PID Controller Design for a UAV Quadrotor». In: *Scientific research and essays* 5 (2010), pp. 3660–3667 (cit. on p. 4).

[20] M. Mammarella G. Ristorto E. Capello N. Bloise G. Guglieri F. Dabbene. «Waypoint Tracking via Tube-based Robust Model Predictive Control for Crop Monitoring with Fixed-Wing UAVs». In: *IEEE International Workshop On Metrology for Agriculture and Forestry*. 2019 (cit. on pp. 5, 72).

[21] S. Khatoon et al. «PID & LQR Control for a Quadrotor: Modeling and Simulation». In: *International Conference on Advances in Computing,Communications and Informatics (ICACCI)*. 2014, pp. 796–802 (cit. on pp. 5, 31).

[22] A. Khattab et al. «Implementation of Sliding Mode Fault Tolerant Control on the IRIS+ Quadrotor». In: *2018 IEEE Conference on Control Technology and Applications (CCTA)*. 2018, pp. 1724–1729 (cit. on pp. 5, 8, 19).

[23] *NASA Interns Develop Guidance, Navigation, and Control Software for Quadcopter with Model-Based Design*. MathWorks. URL: https://it.mathworks.com/company/user_stories/nasa-marshall-space-flight-center-internship-program.html (cit. on p. 5).

[24] E.Ebeid et al. «A Survey of Open-Source UAV Flight Controllers and Flight Simulators». In: *Microprocessors and Microsystems* 61 (2018) (cit. on p. 8).

[25] J. A. Mendoza-Mendoza et al. *Advanced Robotic Vehicles Programming: An Ardupilot and Pixhawk Approach*. Apress, 2020 (cit. on p. 8).

[26] K. Yang et al. «Research of Control System for Plant Protection UAV Based on Pixhawk». In: *Procedia Computer Science* 166 (2020), pp. 371–375 (cit. on p. 8).

[27] *Pixhawk Pilot Support Package (PSP) User Guide Version 3.04*. MathWorks Pilot Engineering Group. 2018 (cit. on pp. 8, 13, 19).

[28] *The history of Pixhawk*. Auterion. URL: https://auterion.com/company/the-history-of-pixhawk/ (cit. on pp. 8, 15).

[29] *PX4 Development Guide*. DroneCode. URL: https://dev.px4.io/ (cit. on pp. 9, 10, 14, 15, 17, 18, 34, 41).

[30] *DS-011 Pixhawk Autopilot v5X Standard Revision 0.3.0*. Pixhawk. 2020 (cit. on p. 8).

[31] S. Ahn S. Malik. «Modeling Firmware as Service Functions and Its Application to Test Generation». In: *Hardware and Software: Verification and Testing*. Springer International Publishing, 2013, pp. 61–77 (cit. on p. 9).

[32] *Universal Serial Bus Specification Revision 2.0*. USB Implementers Forum, Inc. 2000 (cit. on pp. 10, 12).

[33] *Pulse Width Modulation*. AspenCore - ElectronicsTutorials. URL: https://www.electronics-tutorials.ws/blog/pulse-width-modulation.html (cit. on pp. 11, 21, 30).

[34] *NuttX Real-Time Operating System*. Apache Software Foundation. URL: https://cwiki.apache.org/confluence/display/NUTTX/NuttX (cit. on p. 13).

[35] L. Meier et al. «PX4: A Node-Based Multithreaded Open Source Robotics Framework for Deeply Embedded Platforms». In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 6235–6240 (cit. on pp. 13, 14).

[36] Q. Jiang L. Wang. «Research on Obstacle Avoidance System and Path Planning of Unmanned Ground Vehicle Based on PX4». In: *Dynamical Systems and Control* 08 (2019), pp. 167–180 (cit. on p. 15).

[37] J. Moulton et al. «An Autonomous Surface Vehicle for Long Term Operations». In: *OCEANS 2018 MTS/IEEE Charleston*. 2018, pp. 1–10 (cit. on p. 15).

[38] *Embedded Coder Support Package for PX4 Autopilots Documentation*. MathWorks. 2020. URL: https://it.mathworks.com/help/releases/R2020a/supportpkg/px4/ (cit. on pp. 19, 20, 73).

[39] A. Polak. *PX4 development kit for Simulink*. Polakium Engineering, 2014 (cit. on p. 19).

[40] *Embedded Coder*. MathWorks. 2020. URL: https://it.mathworks.com/help/ecoder/ (cit. on pp. 20, 27, 36, 37, 73).

[41] *CMake*. Kitware. URL: https://cmake.org/overview/ (cit. on pp. 20, 38).

[42] R. Aarenstrup. *Managing Model-Based Design*. MathWorks, 2015 (cit. on pp. 23, 24).

[43] B. Potter. *Model-Based Design for DO-178B*. MathWorks. URL: https://it.mathworks.com/company/newsletters/articles/model-based-design-for-do-178b.html (cit. on p. 25).

[44] *Essential aspects of the V-cycle software development process*. X-Engineer. URL: https://x-engineer.org/graduate-engineering/modeling-simulation/model-based-design/essential-aspects-of-the-v-cycle-software-development-process/ (cit. on p. 25).

[45] H. D. Benington. «Production of Large Computer Programs». In: *Annals of the History of Computing* 5.4 (1983), pp. 350–361 (cit. on p. 25).

[46] *Difference between Verification and Validation*. Software Testing Class. URL: https://www.softwaretestingclass.com/difference-between-verification-and-validation/ (cit. on p. 25).

[47] N. C. Pemmaraju. «Model-Based Design for Autonomous Aerial Systems». In: *Matlab Expo*. 2019 (cit. on pp. 26, 27).

[48] M. Kale N. Ghatwai S. Repud. *Processor-In-Loop Simulation: Embedded Software Verification & Validation In Model Based Development*. eInfochips. URL: https://www.design-reuse.com/articles/42548/embedded-software-verification-validation-in-model-based-development.html (cit. on p. 25).

[49] K. Schultz. *All About SIL in MATLAB*. MATLAB Central File Exchange. URL: https://it.mathworks.com/matlabcentral/fileexchange/60245-all-ab%20out-software-in-the-loop-in-matlab (cit. on pp. 26, 34).

[50] J. Mina et al. «Processor-in-the-loop and hardware-in-the-loop simulation of electric systems based in FPGA». In: *13th International Conference on Power Electronics (CIEP)*. 2016, pp. 172–177 (cit. on p. 26).

[51] P. Sarhadi S. Yousefpour. «State of the art: hardware in the loop modeling and simulation with its applications in design, development and implementation of system and control software». In: *International Journal of Dynamics and Control* 3 (2014) (cit. on pp. 26, 33).

[52] P. Marqués A. Da Ronchor. *Advanced UAV Aerodynamics,Flight Stability and Control -Novel Concepts, Theory and Applications*. Wiley, 2017 (cit. on pp. 27, 30).

[53] T. Bresciani. *Modelling, Identification and Control of a Quadrotor Helicopter*. Department of Automatic Control - Lund University, 2008 (cit. on pp. 27, 28, 30).

[54] R. Beard. «Quadrotor Dynamics and Control». In: (2008) (cit. on pp. 27, 30).

[55] B. J. Emran H. Najjaran. «A review of quadrotor: An underactuated mechanical system». In: *Annual Reviews in Control* 46 (2018), pp. 165–180 (cit. on p. 30).

[56] D. Greenfield. *When is PID Not the Answer?* Control Engineering. URL: https://www.controleng.com/articles/when-is-pid-not-the-answer/ (cit. on p. 31).

[57] K. Ogata. *Modern Control Engineering*. Prentice Hall, 2010 (cit. on p. 32).

[58] F. Sabatino. *Quadrotor control: modeling, nonlinearcontrol design, and simulation*. School of Electrical Engineering (EES) - KTH Royal Institute of Technology, 2015 (cit. on p. 33).

[59] M. Zuo et al. «Model-Based Design of UAV Autopilot Software». In: *Advanced Materials Research* (2014), pp. 756–759 (cit. on p. 34).

[60] *jMAVsim ReadMe*. DrTon. URL: https://github.com/DrTon/jMAVSim (cit. on pp. 34, 35).

[61] A. Driss et al. «Simulation Tools, Environments and Frameworks for UAV Systems Performance Analysis». In: *14th International Wireless Communications & Mobile Computing Conference (IWCMC)*. 2018, pp. 1495–1500 (cit. on pp. 34, 40).

[62] M. Mammarella E. Capello. «Tube-Based Robust MPC Processor-in-the-Loop Validation for Fixed-Wing UAVs». In: *Journal of Intelligent & Robotic Systems* 100 (2020) (cit. on p. 35).

[63] L. Rosqvist R. Aarenstrup K. Lindqvist. *Processor-In-the-Loop Simulation on Embedded Linux Boards*. MathWorks. URL: https://it.mathworks.com/company/newsletters/articles/processor-in-the-loop-simulation-on-embedded-linux-boards.html (cit. on p. 37).

[64] R. Parizi et al. «Towards Gamification in Software Traceability: Between Test and Code Artifacts». In: 2015, pp. 393–400 (cit. on p. 36).

[65] *Code Efficiency*. Techopedia. URL: https://www.techopedia.com/definition/27151/code-efficiency (cit. on p. 36).

[66] I. Fey I. Stürmer. «Code Generation for Safety-Critical Systems-Open Questions and Possible Solutions». In: *SAE International Journal of Passenger Cars - Electronic and Electrical Systems* 1 (2009) (cit. on p. 36).

[67] *Target Language Compiler Basics*. MathWorks. 2020. URL: https://it.mathworks.com/help/rtw/tlc/what-is-the-target-language-compiler.html (cit. on p. 37).

[68] E. Sorton S. Hammaker. «Simulated Flight Testing of an Autonomous Unmanned Aerial Vehicle Using FlightGear». In: 2005 (cit. on p. 40).

[69] *FlightGear Flight Simulator*. FlightGear Project. URL: https://www.flightgear.org/about/ (cit. on p. 40).

[70] *Aerospace Blockset*. MathWorks. 2020. URL: https://it.mathworks.com/help/aeroblks/ (cit. on p. 40).

[71] *Simulink 3D Aninmation*. MathWorks. 2020. URL: https://it.mathworks.com/help/sl3d/ (cit. on pp. 41, 43, 56, 89).

[72] L. Comba S. Zaman A. Biglia A. D. Ricauda F. Dabbene P. Gay. «Semantic interpretation and complexity reduction of 3D point clouds of vineyards». In: *Biosystems Engineering* 197 (2020), pp. 216–230 (cit. on pp. 43, 46).

[73] *ConveRgo*. GeoPiemonte - Regione Piemonte. URL: https://www.geoportale.piemonte.it/cms/servizi/servizi-di-conversione/25-il-programma-convergo (cit. on pp. 43, 47, 89).

[74] *CloudCompare Version 2.11.1*. CloudCompare. 2019 (cit. on pp. 43, 49, 89).

[75] P. Cignoni et al. «MeshLab: an Open-Source Mesh Processing Tool». In: *Computing*. Vol. 11. 2008, pp. 129–136 (cit. on pp. 43, 54, 89).

[76] T. S. Taylor. *Introduction to Laser Science and Engineering*. CRC Press, 2020 (cit. on p. 45).

[77] *Introduction to Lidar Point Cloud Data - Active Remote Sensing*. Earth Lab. URL: https://www.earthdatascience.org/courses/earth-analytics/lidar-raster-data-r/explore-lidar-point-clouds-plasio/ (cit. on p. 45).

[78] *What is lidar?* NOAA (National Oceanic and Atmospheric Administration). URL: https://oceanservice.noaa.gov/facts/lidar.html (cit. on p. 46).

[79] *Agisoft PhotoScan*. Agisoft. URL: https://www.agisoft.com/ (cit. on p. 47).

[80] P. Corradeghini. *Trasforma coordinate con il software gratuito CONVERGO*. 3DMetrica - Youtube. 2019. URL: https://www.youtube.com/watch?v=vrk4VYmCyak (cit. on pp. 47, 90).

[81] *Spatial Re-Sampling*. ESA - European Space Agency. URL: https://sentinel.esa.int/web/sentinel/technical-guides/sentinel-3-olci/level-1/spatial-re-sampling (cit. on p. 49).

[82] William G. Cochran. «The χ^2 Test of Goodness of Fit». In: *Annals of Mathematical Statistics* 23.3 (1952), pp. 315–345 (cit. on p. 49).

[83] H. Samet M. Tamminen. «Efficient component labeling of images of arbitrary dimension represented by linear bintrees». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 10.4 (1988), pp. 579–586 (cit. on p. 49).

[84] Hugo Moreno et al. «On-Ground Vineyard Reconstruction Using a LiDAR-Based Automated System». In: *Sensors* 20 (2020) (cit. on p. 49).

[85] D. Girardeau. «CloudCompare - Point Cloud Processing Toool». In: *CloudCompare Workshop*. 2019 (cit. on p. 49).

[86] N. Brodu D. Lague. «3D terrestrial lidar data classification of complex natural scenes using a multi-scale dimensionality criterion: Applications in geomorphology». In: *ISPRS Journal of Photogrammetry and Remote Sensing* 68 (2012), pp. 121–134 (cit. on pp. 50, 51).

[87] C. Cortes V. Vapnik. «Support-vector networks». In: *Machine Learning* 20 (1995), pp. 273–297 (cit. on p. 50).

[88] W. Yu et al. «Application of support vector machine modeling for prediction of common diseases: The case of diabetes and pre-diabetes». In: *BMC medical informatics and decision making* 10 (2010), p. 16 (cit. on p. 51).

[89] F. Remondino. «From point cloud to surface: The modeling and visualization problem». In: *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XXXIV-5/W10 (2004) (cit. on p. 53).

[90] R.A. Jarvis. «On the identification of the convex hull of a finite set of points in the plane». In: *Information Processing Letters* 2 (1973), pp. 18–21 (cit. on p. 53).

[91] H. Edelsbrunner E. Mucke. «Three-Dimensional Alpha Shapes». In: *ACM Transactions on Graphics* 13 (1994) (cit. on p. 54).

[92] F. Bernardini et al. «The Ball-Pivoting Algorithm for Surface Reconstruction». In: *IEEE Transactions on Visualization and Computer Graphics* 5 (1999), pp. 349–359 (cit. on p. 54).

[93] B. Kent. *3D Scientific Visualization with Blender*. Morgan & Claypool Publishers, 2015 (cit. on pp. 56, 57, 89).

[94] L. Tran et al. «Simulation and Visualization of Dynamic Systems Using MATLAB, Simulink, Simulink 3D Animation, and SolidWorks». In: *International Mechanical Engineering Congress and Exposition (ASME)*. Vol. 7. 2011 (cit. on p. 56).

[95] *Quadcopter Project*. MathWorks. URL: `https://it.mathworks.com/help/aeroblks/quadcopter-project.html` (cit. on pp. 59, 97).

[96] *Code Execution Profiling with SIL and PIL*. MathWorks. 2020. URL: `https://it.mathworks.com/help/ecoder/ug/configuring-code-execution-profiling.html` (cit. on pp. 69, 72).

[97] M. Mammarella T. Alamo F. Dabbene M. Lorenzen. *Computationally efficient stochastic MPC: a probabilistic scaling approach*. 2020 (cit. on p. 72).

[98] M. Beul S. Behnke. *Trajectory Generation with Fast Lidar-based 3D Collision Avoidance for Agile MAVs*. 2020 (cit. on p. 72).

[99] *Mingw-w64*. 2020. URL: `http://mingw-w64.org/doku.php/documentation` (cit. on p. 74).