Journal of Intelligent & Robotic Systems (2019) 96:223–243
https://doi.org/10.1007/s10846-018-0969-3

Integrated Framework for Fast Prototyping and Testing of Autonomous Systems

Luigi Pannocchi[1] · Carmelo Di Franco[1] · Mauro Marinoni[1] · Giorgio Buttazzo[1]

Received: 9 January 2018 / Accepted: 3 December 2018 / Published online: 18 December 2018

Abstract
Validating the behavior of a complex system is a fundamental step in the development process to avoid costly damages and dangerous circumstances. Such a phase requires a realistic simulation of the system and the reproduction of the full operative scenario, including the environment with all the possible events and situations in which the system could get into. Although several tools exist to design, simulate and validate specific functions, checking the overall system behavior in an operative scenario usually requires the development of custom simulation frameworks. These are often tailored to the specific system under study, with the consequence that they are either incomplete or not fully reusable for other projects. This paper presents a modular hardware-in-the-loop development simulation framework that allows realistic simulation, supporting multi-vehicle scenario and comprehending tools for reproducing realistic testing environments with advanced sensors. A case of study is presented to show the employment of the proposed framework for testing the behavior of unmanned vehicles, focusing on the timing properties of the system.
Category (2).

Keywords Simulation · Hardware-in-the-loop · Multi-robot

Mathematics Subject Classification (2010) 68-04

1 Introduction

Nowadays, the presence of autonomous vehicles has become common in disparate application domains. The spreading of their employment is due to the clear advantages they offer in terms of flexibility, safety, and performance. These results have been obtained thanks to the continuous improvement of computational platforms, materials, and algorithms, paving the way towards ”real autonomous” vehicles capable of interacting with dynamic environments in a wide range of circumstances, like fire detection [14], surveillance, inspection of industrial plants, or monitoring of the fauna [20] and flora in large areas [3].

Accomplishing a task with autonomous vehicles requires taking into account different aspects like control, navigation, guidance, multi-agent coordination, and computer vision.

The complexity is often tackled structuring the project on different abstraction levels, which can be managed separately and in parallel, as Lum et al. pointed out in their work [12]. Figure 1 represents an example of such structure for the design of software involving autonomous vehicles. At the lowest level, controlling the dynamics of the vehicle requires the knowledge of the vehicle detailed model, whilst at the Strategic Level, the algorithms often abstract the dynamics of the vehicles and the control laws.

As for many embedded systems, at the end of the development phase, all the individual components are going to be implemented on a physical board, with limited computational power and memory. Starting from the theoretical aspects, engineers go through the software implementation

✉ Luigi Pannocchi
l.pannocchi@santannapisa.it

Carmelo Di Franco
c.difranco@santannapisa.it

Mauro Marinoni
m.marinoni@santannapisa.it

Giorgio Buttazzo
g.buttazzo@santannapisa.it

[1] Scuola Superiore Sant'Anna, Pisa, Italy

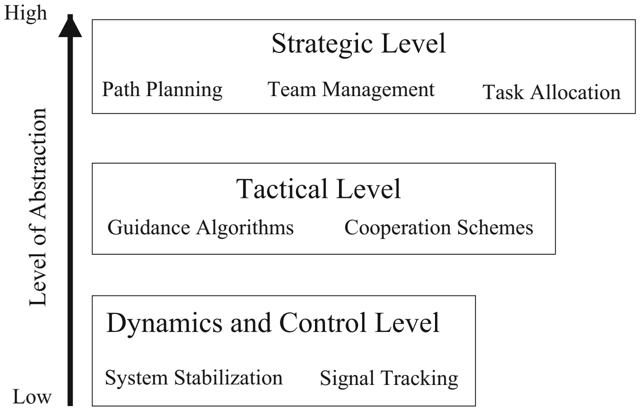


Fig. 1 Abstraction levels in the development of applications involving autonomous vehicles

phase and then they end up with the deployment on the target hardware, as shown in Fig. 2.

This aspect should be taken into account when coming up with feasible solutions, employing tools able to validate the system to avoid unwanted problems during/after the deployment phase.

This multi-disciplinary approach, together with the complexity of the design process and the embedded nature of autonomous vehicles, entails a complex validation step. This complexity is also determined by the interest in the topic, both from the academical and industrial point of view, which is pushing forward the technology and, in turn, the capabilities of those systems. In order to answer the nowadays demands it is therefore necessary to continuously chase new possibilities with the relative mushrooming of different operating conditions that need to be simulated.

The possibility of validating and testing the full system behavior on a realistic simulation framework would allow

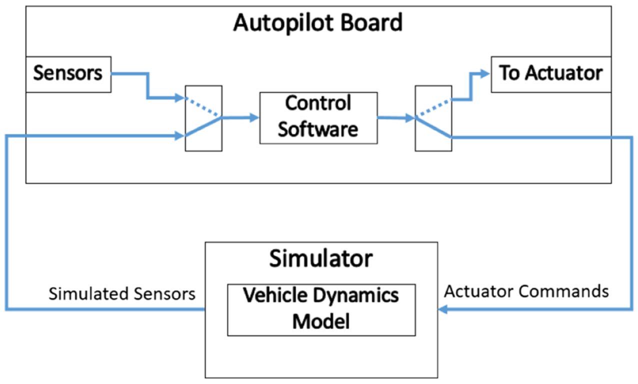
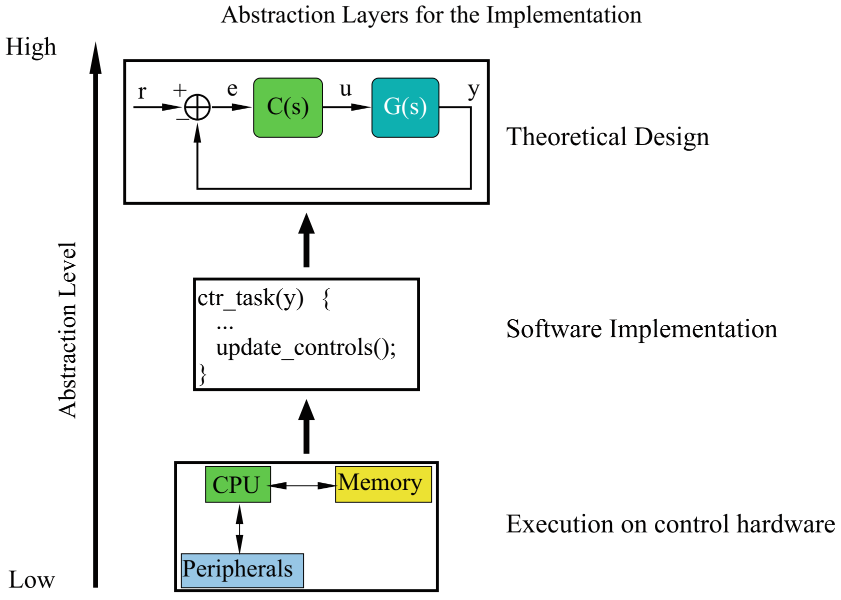


Fig. 3 Hardware-in-the-loop scheme

significant benefits, as reducing the development time, avoiding crashing the vehicles due to software failures, and evaluating the reaction to peculiar situations that could be too rare or not reproducible in practice. The possibility of evaluating control software in a simulation framework is particularly important in a research setting, where novel approaches, which are not supported by a background of extensive usage and thus can exhibit unknown shortcomings, are typically adopted to develop innovative solutions. For the sake of clarity, since the concept of simulation itself is very general and exhibits several facets, talking about simulation framework it would be useful to distinguish two main aspects: *Plant Simulation* and *Environment Simulation*.

Plant Simulation Reproducing realistic simulation, also taking into consideration the implementation details mentioned above (Fig. 2), is a crucial objective in the development process and this can be achieved through the

Fig. 2 Abstraction levels in the implementation of a control application on an embedded system



hardware-in-the-loop simulation approach. This method, schematically illustrated in Fig. 3, consists in running the control software directly on the embedded control board of the vehicle, in this case an autopilot board, which interacts with a simulated physical plant. In this setting, the physical sensors and actuators of the vehicle are disabled and replaced with virtual counterparts running in the simulator, while the control hardware carries out the control tasks. For the sake of clarity, the origin of the term "hardware-in-the-loop" refers to the fact that the "control hardware", in this case the autopilot board, is included in the simulation loop. This terminology is generally accepted by the research community [9, 11, 15, 16, 18, 19, 21].

Environment Simulation To test high-level functions (e.g., obstacle avoidance, target recognition and tracking, integration of video information for navigation), it is necessary to recreate a synthetic environment and model the interactions between this simulated world and the vehicle. The details of such interactions should be sufficient to cover a wide set of scenarios typical of modern applications.

Inline with the aim of identifying the support necessary for the development of solution involving autonomous systems, these simulation facilities should be provided as an outcome of a integrated tool to make them really profitable. This entails that, in addition to what/how simulate, other properties come into play. Given the frenetic pace of the research development, the success of a tool also depends on its maintainability and interoperability with other software. It should be possible to use it with a large variety of systems, and any possible upgrade should be easy to do, providing a good maintainability. The support tools should be able to deal with complex scenarios, allowing to validate advanced functionalities of the developed systems. Moreover, the development of models and testing scenarios should be made as simple as possible, to reduce the total effort in designing the solution of the actual control problem. Without such properties, the tool is likely to become obsolete in a short time.

While this consideration seems to be reasonable, it often happens that research communities, working on this topics, tend to develop their own custom simulation frameworks. Such tools are often too specific and tailored to the system under development, and it is likely to see them used for a limited number of applications.

Contributions This work presents a hardware-in-the-loop simulation framework for multi-vehicle autonomous systems[1], with the aim of supporting the complex development of tasks involving them. The main contribution of the work is the designed structure of the Simulation Framework, which allows providing the required support, integrating different and possibly heterogeneous components. Indeed, the proposed solution allows managing in a single framework the mathematical modeling of the plant and the environment, the design of the testing scenario, the simulation of the whole system in a hardware-in-the-loop configuration to spot hardware problems before the deployment, and the reproduction of the operating scenario embedding the application for vehicle management in the simulation loop. As a second contribution, the data flow in the Simulation Framework has been managed paying particular attention to timing issues, which allows achieving higher timing precision and accuracy in data exchange with respect to the existing related works. To summarize, the Simulation Framework has been designed to fit out the following characteristics:

1. Support for multi-agent scenario;
2. Modular design to improve maintainability;
3. Support for integrating generic autopilot boards;
4. Possibility to interoperate with a wide range of user interfaces for autonomous vehicles;
5. Support for testing high-level functions of the vehicles;
6. Precise time management to increase the realism of the simulation;
7. Rapid prototyping of complex environment for testing and validation:
   - Easily model environment sensors like distance sensors, sound sensors, laser sensors, etc.;
   - Easily model visual effects like lights, shadows, clouds and different weather conditions;
   - Easily include moving objects, vehicles and other entities, also with the capability to drive them directly online by user inputs.

The rest of the paper is organized as follows. Section 2 presents an overview of existent solutions, identifying interesting approaches and drawbacks, to set up the starting point of this work. Section 3 gives a general overview of the proposed approach, while Sections 4 and 5 provide the implementation details of the Framework Core, including the Simulation Engine, and the Synthetic Environment, respectively. Section 6 reports a case of study to show the utilization of the framework. Section 7 is dedicated to the evaluation of the performance and limits of the proposed implementation. Section 8 concludes the paper and identifies the future research directions.

2 Related Works

The need for realistic simulated environments is something the research groups working with robotics and automation

[1]The project is available online on the ReTiS laboratory website under: http://retis.sssup.it/?q=content/simulation-framework-autonomous-vehicles

came across in their activities. Indeed, simulation is not only limited to the dynamics of the systems, but also includes the environment reproduction and relative interaction with that comes when planning complex missions. It is not by chance that, dealing with tools for robotics, it is possible to talk about “situated robots”. The adjective “situated” is used to stress the fact that the agent is embedded in an environment and can interact with it.

There are a lot of solutions already available, trying to answer this need of realism for different aspects of the problem. The panorama is ample and a good evidence of this is the fact that, in the research community, there are attempts to compare and classify them like the works of Cook et al. [6], Castillo et al. [5] and Jouvencel et al. [18]. Given the large set of possible applications, as explained in Section 1, making surveys on this topic is not an easy deal: the examinations cover only a partial list of possible framework characteristics. In this section some relevant related works are proposed, looking at the features necessary to cover the validation needs as much as possible:

– realism of a synthetic environment;
– realism of the simulation; and
– possibility to integrate the user interface software used in the real operating condition.

Considering the aspect of the environment simulation, the work of Carpin et al. [4] was focused on developing a realistic simulation framework (*USARSim*) for multi-robot scenario in complex environments. Applications of such simulator are found in robotics challenges like the *RoboCup Rescue Simulation League* and in the *IEEE Virtual Manufacturing Automation Competition*. In the examples reported in their work the task was to reproduce a cluttered environment for the robots and simulate the interaction with it. The authors identified the *Unreal Engine*[2] game design suite as a an useful tool for engineering and research activities. Given the ability to easily model the environment, the *USARSim* simulator has been successfully used by Birk et al. [2] in their work for the simulation of the Mars soil in the context of terrain classification with a rover. Sehgal et al. [22] used the same simulator to model an underwater scenario. *Unreal Engine* was used both for designing the environment and for simulating the dynamics of the systems.

Ganoni and Mukundan [8] proposed a framework for multi-robot simulation providing a software-in-the-loop simulation based on the Ardupilot and PX4 autopilot software interacting with the Unreal Engine synthetic environment, similarly to the approach proposed in this paper. Their approach allows simulating multiple vehicles, but the main focus is the development of complex vision algorithms, leaving the support for Hardware-in-the-loop simulation is left as a future work [8]. Furthermore, there is not yet interaction with the synthetic world (Unreal Engine), and the vehicle dynamic is not affected by wind gusts or impacts with objects. Despite the authors claim that their approach is Realtime (since it runs at 30 Hz), they do not provide studies on the latency. Also, they are not taking into account the impact of executing the vision algorithms directly on the autopilot board. Finally, the Ardupilot and PX4 firmware must be modified for being used in the framework.

Another simulator that allows the employment of multiple heterogeneous robots with a high degree of realism is *MORSE*, which was developed by Echeverria et al. [7]. In their work they made use of a 3D modeling and rendering application (*Blender*) to recreate realistic scenario, useful also for testing image processing. *MORSE* uses the *Bullet physics library* for the simulation of the physics.

Considering the features more related to the control part, it is worth noticing that both *USARSim* and *MORSE* do not allow a hardware-in-the-loop simulation. While the first deals only with pure simulations, the latter has been conceived for supporting software-in-the-loop and to be independent from specific communication middlewares. Moreover, the modeling of the system is accomplished using the libraries included in the tools. This could lead to limits in the modeling freedom and it does not take advantages of other classical modeling/simulation/design tools which are widespread in the academical and industrial field, like *Matlab*. In this way, the user is forced to get accustomed to the adopted modeling method, making the initial approach more difficult. Another point that deserves consideration is that even if the presence of such frameworks constitutes an answer to some of the needs mentioned in the introduction, they focus on the robot/environment interaction for testing specific behavior, with a limited interaction with the user. On the contrary, applications involving autonomous systems usually comprehend also complex user interfaces and other software for the management/control of the vehicles/robots. Since the final aim of the validation is to test the overall system, reproducing the real operating conditions, proper simulation should include these user interfaces. This is particularly important for unmanned aerial vehicles, which are managed through ground stations.

Looking at solutions that allow hardware-in-the-loop simulation, it is usual to find approaches that consider only a single vehicle, like the ones proposed in [9, 11, 15, 19, 21]. Mainly this is due to a trade-off between realism and system scalability. Such tools were more oriented towards the realism of the simulation. As an example, the work of Kamali and Shikha [9] presented a simulation model developed in *Matlab/Simulink* and running on an FPGA to support real-time communication on I/O data buses (I2C

[2]https://www.unrealengine.com/ (*Unreal Engine* 4)

and SPI) with the control board. If on the one hand this approach allows achieving a good level of realism, on the other hand it does not provide good scalability. A highly realistic hardware-in-the-loop simulation tool for multi-agent systems has been proposed by Kamal and Shumaker [1], but it is not scalable to many vehicles given the complexity of the simulation system. Santos et al. [21] made use of the flight simulator software *X Plane* for simulating the vehicle dynamics. With this approach it was not possible to simulate different kinds of vehicles and the simulation rate was coupled with the refresh rate of the video. This coupling turns out to be a problem for simulation purposes. Indeed the timing of the video output frames is dependent on the workload necessary to process the images, which is not constant and it is influenced by several factors, such as image type, quality, etc. Moreover the maximum achievable simulation frequency is in the order of hundred of Hz, given the high computational load required to process images, and this could be not always sufficient.

Takaya et al. [24] proposed a simulation environment for mobile robots based on the well known simulation platform for robotic systems *ROS* and the associated physical simulator *Gazebo*. The interest in *ROS* consists in the fact that the simulation results can be directly deployed to the *ROS*-enabled robot hardware and allow a straithforward transition to hardware-in-the-loop simulation as made by Odelga et al. [16]. *In their work they show that ROS/Gazebo is quite flexible and it makes possible to achieve good performance in simulation, for example running with a simulation frequency of 1KHz*. The drawbacks in adopting *ROS* and *Gazebo* are that the installation, configuration and utilization are not straightforward and the presence of compatibility problems forces to use only the supported OS platforms. The modeling of the vehicle in *Gazebo* is made using *xml* description files and then the simulation is automatically carried out. Regarding this modeling approach, the same consideration about the ease usage made for the *USARSim* and *MORSE* can be made.

As far as the user interface is concerned, some of these research groups developed their ground station application and embeeded them in the simulation environment like in [11, 19, 21]. Others included commercial ground stations in their frameworks as in the work of Kamali et al. [9] and Pannocchi et al. [17]. Going back to the considerations made about the requirements of a useful simulation environment, most of these hardware-in-the-loop frameworks, with he exception of Jouvencel [18] and Pannocchi [17], were not made with the aim of providing realism of the environment and the interaction with it. This is justified by the fact that they are mainly developed to support flying vehicles, which are moving in a 3D free space, with the focus on the control part. More precisely, Jouvencel et al. [18] proposed a well-structured simulation framework for underwater vehicles, able to manage multiple heterogeneous agents. The simulator models the environment, including obstacles, water properties and constraints related to communication in water, but the framework does not include a 3D visualization. Pannocchi et al. [17] proposed a modular simulation framework for multi-vehicle scenario, with a high degree of realism, both in the dynamics modeling and visualization (*Unreal Engine*), together with the inclusion of user interface software. In their work the interaction with the synthetic environment was still limited to a bare visual feedback and some basic information about the identified collisions. A similar work was made by Shah et al. [23], where they proposed a realistic environment for testing advanced functionality using the *Unreal Engine*. They described the vehicle model, the architecture and the simulation outputs, however, they did not investigate the timing properties of their platform and they were focused on a single vehicle.

Having considered all these related works, in the following, a novel framework will be proposed, with the aim of gathering together the conceived positive aspects and make up for the identified lacks.

3 System Description

The proposed simulation framework consists of four main intercommunicating components, as illustrated in Fig. 4.

The structure has been designed to allow distribution of the components on separated machines in order to provide scalability, modularity, and separate the high computational burden, which could be required by the framework. For this purpose, the communication with the ground station and the synthetic environment has been performed using the UDP protocol. A brief description of the individual components is given in the following and then the details are provided in the subsequent sections.

3.1 Framework Core

The core of the Simulation Framework consists of the Simulation Engine and the Data Router.

The Simulation Engine component simulates the dynamic behavior of the vehicles using mathematical models for their structure and the equipped sensors. Providing different models, it is possible to simulate multiple heterogeneous vehicles. Using the information contained in each model, the simulation step performs the vehicle state integration, considering the actuation commands computed by the Control Software and the simulation of the sensory data. The actuation commands are provided by the Data Router, which is listening for data coming from the Control Software. Differently to common flight simulators, like

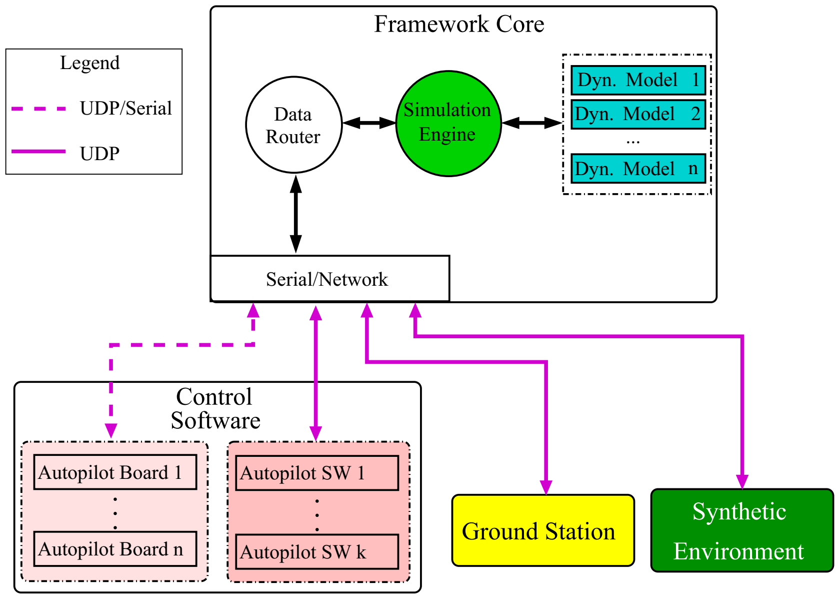


Fig. 4 Overall structure of the Simulation Framework

FlightGear or *X-Plane*, and the *Unreal Engine* itself, the simulation frequency can be changed by the user and it is not bounded by the video refresh rate. The sensory data output frequency can be decoupled from the simulation frequency. A straightforward solution to achieve this is by simulation sub-stepping, that is, by sending the sensory data every N simulation cycles, where N is the number of sub-steps.

The Data Router constitutes the manager of the information that flows among the Simulation Engine and all the other components. Its presence guarantees a correct interoperability among components and ensures that data exchange is carried out avoiding unnecessary delays, also considering different priorities of the communication participants and the worst-case blocking times on the access to shared resources.

3.2 Control Software

As introduced in Section 1, the control software implements the functionality of the autonomous vehicles, starting from the processing of bare sensory data up to high level functions, like the mission management. The aim of the simulation tool is to validate this software, also allowing its direct testing on the target control board. The software expects sensory data from the simulator, together with other possible commands from the ground station, as during real operating conditions. In turn, computed control signals and status messages are sent back from the control device to the simulator and ground station, respectively. As far as the hardware-in-the-loop configuration is concerned, the system supports boards that can communicate via serial port or network interface using the *MAVLink* protocol. The rationale behind this choice is that *MAVLink* has become a de facto standard communication protocol for open-source autopilot boards and in this way, the proposed Simulation Framework natively supports all of them without requiring ad-hoc modification.

3.3 Synthetic Environment

This component provides the capability of simulating “situated” vehicles while providing a synthetic realistic environment with three main functions:

1. 3D visual feedback Visualizing the vehicles in a virtual environment is a valuable feature that simplifies the verification of the developed functions.
2. Simulating the output of Cameras The image generated from the synthetic environment can be used as an output from a virtual camera located on the vehicles. This facilitates the testing of advanced functions, as vision-based maneuvers.
3. Advanced Sensory Data It is possible to retrieve information about the interaction of a vehicle with the synthetic environment, such as collisions with virtual obstacles, distance from the obstacles, intensity of sound sources, etc.

The synthetic environment has been designed with *Unreal Engine*, which is a free suite for game development. It makes possible to design realistic scenarios without directly concerning about complex visual effects, which are managed automatically by the graphical engine of the application. This is necessary to support the design of computer vision solutions, where common phenomena like light reflections, shadows, and mist constitute a problem that must be taken into account. The suite includes useful

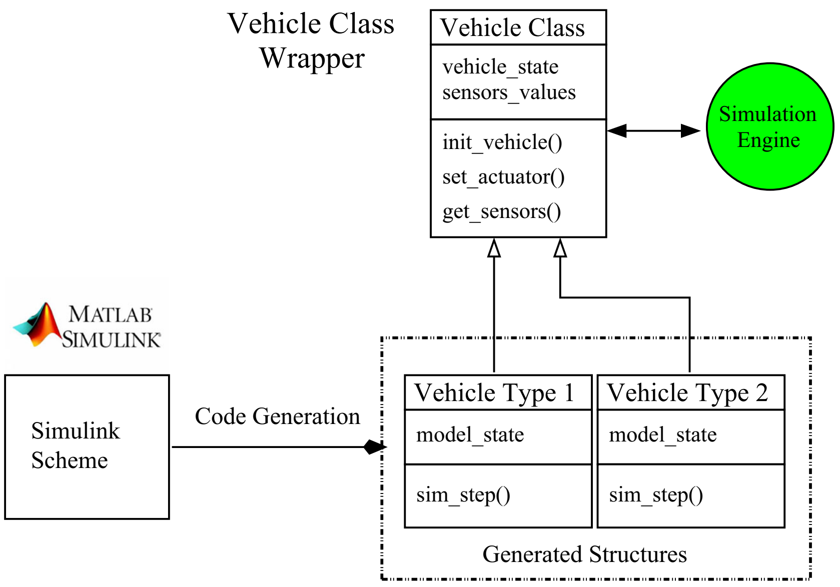


Fig. 5 Implementation of the vehicle models with a Vehicle Class that wraps different dynamical model generated by *Matlab/Simulink*

libraries to create realistic effects, move objects, measure the distance between points, manage collisions, detects the presence in a given area, etc.

3.4 Ground station

The ground station is the application that allows interacting with the autopilot board through a user-friendly interface. It allows the user to change the values of autopilot firmware parameters and perform mission planning and flight control operations. It also provides an interface to visualize the telemetry data and the status of the connected vehicles. Instead of proposing an ad-hoc ground station, in line with the goal of modularity and interoperability, the proposed framework can operate with any ground station implementing the *MAVLink* protocol over UDP. This is the case with most common ground station applications, which can be tested, together with the autopilot board, in this simulation environment.

The ground station used in our implementation is *QGroundControl*[3], which is a free open-source application allowing full modification and code analysis. This application supports the *MAVLink* protocol for communicating with the connected vehicles. It supports different communication interfaces for connecting to the vehicles, also allowing the required UDP protocol. Through it the user can visualize the vehicle on a map (*Google Maps, Bing Map*), plan a mission by specifying waypoints directly on the map, plot telemetry data and simplify several operations on the vehicles, such as flashing firmware, setting up the controller parameters and the remote controller.

[3]http://qgroundcontrol.com/ (QGroundControl - Drone Control)

4 Framework Core

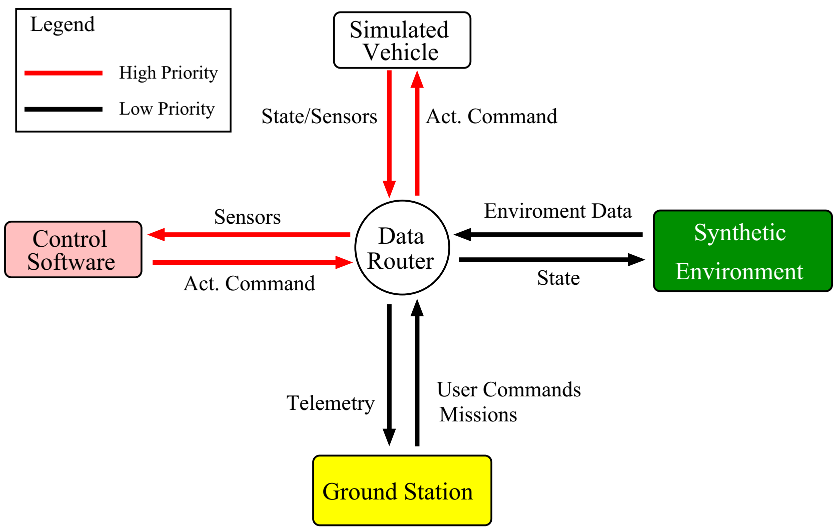
In this section, the Framework Core, which constitutes the main component of the overall system, is described in detail. As far as the Simulation Engine is concerned, the modeling of the vehicles and their sensors is carried out using the C language. This choice offers the possibility to have a full control on modeling of the system and does not exclude the possibility to use *Matlab/Simulink*, taking advantage of C code generation. In this way, it is possible to guarantee scalability and fast execution, which is required in case of multi-vehicle systems. In order to ease the inclusion of different vehicle/sensors models, the integration with the Simulation Engine is made through a vehicle class that constitutes a wrapper for the structures generated by *Matlab*. As represented in Fig. 5, the idea is to have a single class, which can be associated with different types of vehicle.

In this way, a trade-off between maintainability, extensibility, and scalability is achieved.

The interaction between the simulated vehicles and the rest of the framework is carried out through the Data Router, as introduced in Section 3. The information flowing around the Simulation Framework is not limited to sensory data and actuation commands. In Fig. 6 an outline of the data exchange between the participants is reported, stressing the complexity and also the presence of different priorities. In this regards, a correct implementation of the Data Router is particularly important when supporting hardware-in-the-loop-simulations, which demand high timing precision and accuracy.

Indeed the realism of the simulation depends not only on the correctness of the computed values, but also on the time at which they are provided.

Fig. 6 Data routing between different components of the framework and relative priorities



Another aspect that should be taken into account when supporting this kind of simulation is that it consists of two interacting elements: the software under test, running on the autopilot board, and the Simulation Engine executing on a separate machine, respectively. Since the two elements run on separate platforms, it is crucial to have a correct synchronization between them. One possible solution to this issue could be to run the simulation directly on the control board, together with the control software, as done by Mueller [15]. Clearly, this has its drawbacks, since the simulation running on-board constitutes an additional workload, which risks to invalidate the realism of the hardware-in-the-loop simulation. Avoiding this latter approach, there are two possible solutions, depending on how the autopilot software was designed. The first one, which can be defined as *board driven synchronization*, is to let the autopilot board trigger the simulation step, as done by Pollini et al. [19]. The second viable solution, adopted by the *PX4 Flight stack,px4*, is to make the simulator responsible for triggering the execution of the control software. This approach, which can be called *simulator driven synchronization*, is based on a publisher/subscriber middleware and a chained execution pattern optimized to reduce the control latency. During the normal operation of the autopilot board, the execution pace of this chain is given by the publication of new sensory data, achieved by means of high precision timers. Figure 7 visually illustrates the two synchronization methods, stressing the fact that one of the

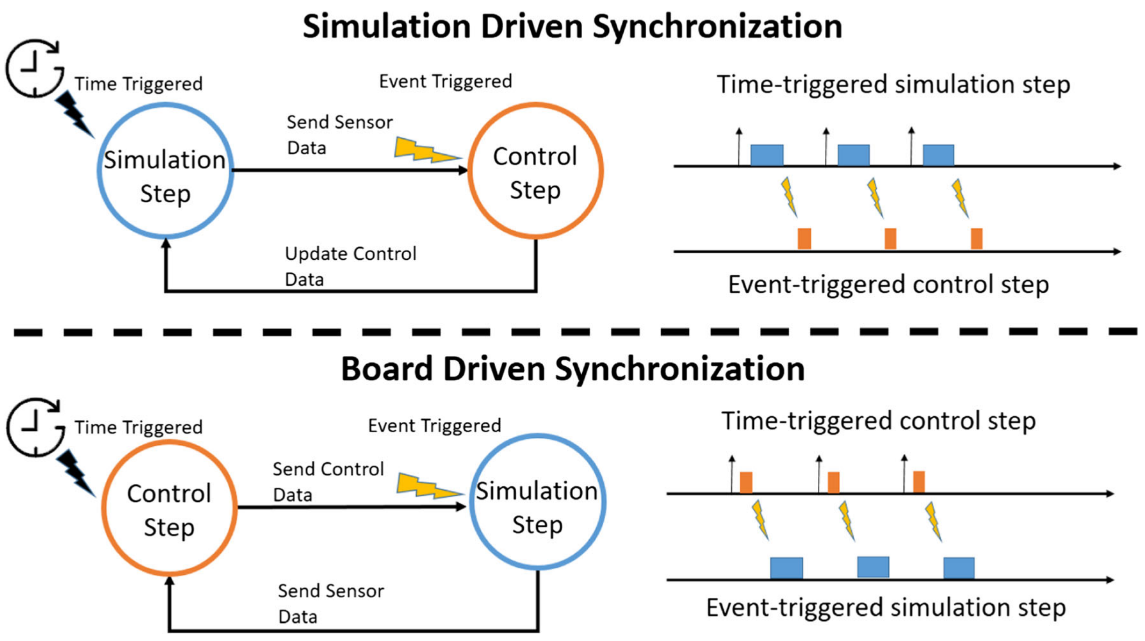


Fig. 7 Synchronization approaches between Simulation Engine and Control Software

two platforms gives the pace to the other. Both approaches are viable and supporting them in a simulation framework allows being as more general as possible. Limiting the support capability to just a specific kind of firmware leads to a tailored solution, as done in some works available in the state of the art [13, 19].

Depending on the kind of synchronization used, the simulator should guarantee different properties. In case of a board driven synchronization, it is important to guarantee low-latency response to avoid introducing extra delay, which could reduce the fidelity of the dynamics model. In case of a simulator driven synchronization, it is still important to provide low latency, but it is also necessary to ensure a precise and accurate period of the simulation cycle to reproduce the triggering of high precision timers.

4.1 Software Implementation

In order to guarantee the previous features, the Framework Core has been designed with a multi-thread architecture. Figure 8 shows an example of the implemented structure for the case of two connected vehicles.

The architecture includes four different kinds of threads:

– *Inflow thread* This thread reads the data produced by the autopilot software via the selected communication channel and then routes each message to the corresponding recipient (e.g., Simulation Engine, Ground Station). This thread is aperiodic and executes when new data from the autopilot software is available.
– *Simulator thread* This thread, which constitutes the Simulation Engine, simulates the dynamics of the vehicle and its sensors. It reads the actuation commands provided by the Inflow thread and the external reactions of the environment updated by the Synthetic Environment; then, it computes the new vehicle state and sensors values, using the vehicle model. The sensor data is sent directly to the autopilot software, while the vehicle state will be used by the Synthetic Environment. Depending on the synchronization method used, the thread can perform a busy wait for the new data from the Inflow thread, which triggers the simulation to ensure low latency, or can freely run and send sensory data to the autopilot software with high precision and accuracy.
– *Ground Station thread* This thread manages the messages exchange between the Ground Station and the vehicles connected to it. This thread is activated periodically.
– *Synthetic Environment thread* This thread manages the data exchange between the Simulator thread and the Synthetic Environment. It sends the state of the vehicle (e.g., position and orientation) to the Synthetic Environment and reads back the information about the

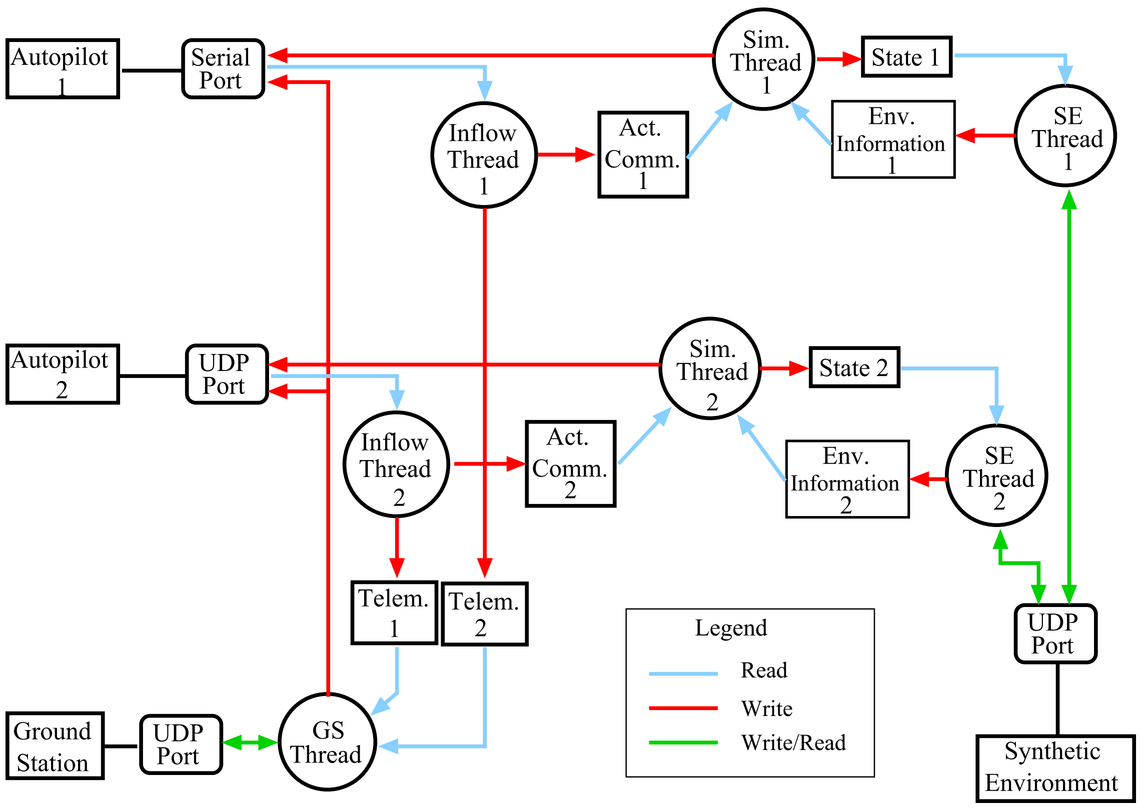


Fig. 8 Scheme of the software implementation of the Framework Core in case of two vehicles

interaction with the virtual environment. This thread is synchronized with the Synthetic Environment: it runs when new data from the latter is available.

Summarizing, for each connected vehicle three threads are created: Inflow thread, Simulation thread, and Synthetic Environment thread. The design choice of separating threads for different vehicles has been undertaken to improve scalability and reduce blocking times due to shared resources.

5 Synthetic Environment

Unreal Engine is a suite of integrated tools for designing and building games, simulations, and visualizations. It has been chosen to be included in this framework for the possibility of reproducing a world with high fidelity using the C++ language and its extensive use in the games community. It has been used for providing two main functions: a) visualization of a realistic synthetic environment and b) interaction with the simulated world. In fact, it is possible to create worlds composed of all the possible terrains, containing rivers, lakes, forests, cities, etc. The objects can be dynamic and follow the laws of physics. For examples, the wind moves the leaves of a tree, and the weather may change over time. Such features allow achieving simulations that are realistic enough to test complex vision algorithms where, for example, the light of the environment may occasionally change (e.g., due to different weather conditions), or moving objects can obstruct the camera or the target. Applications should include advanced guidance algorithms such as target following, human/object recognition.

It is also worth noting that it is possible to use the environment as an augmented reality tool for real flights. Consider a drone that is performing a real mission but gets the images from the synthetic environment instead of receiving them from the onboard camera. In this way, a real drone can see imaginary objects interacting with them without any risk. For instance, specific sensors can be simulated to detect metallic objects for testing de-mining applications.

These kinds of applications can be implemented in the proposed framework thanks to the possibility of interacting with *Unreal Engine*.

5.1 Interaction with *Unreal Engine*

At a high level, the Framework Core interacts with the Synthetic Environment generated by *Unreal Engine* as shown in Fig. 9. The vehicle state is sent to *Unreal Engine* and used to draw the vehicle at the given position and orientation. *Unreal Engine* will send back to the framework core the information of possible collisions with objects in the synthetic world.

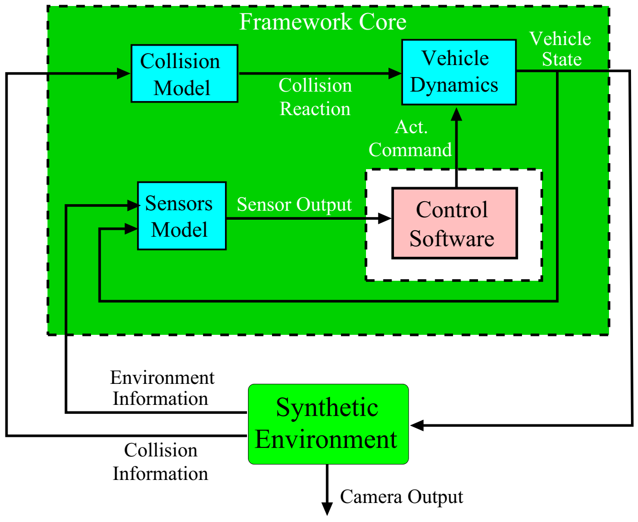


Fig. 9 Scheme of the interaction with the Synthetic Environment

If there is the need of simulating virtual sensors (e.g., ultrasound, camera, laser sensors, etc.), the required information can be extracted from the virtual environment and sent back to the framework core. The collision information consists of the Hit normal and the normal impulse, computed by *Unreal Engine* physics and sent back to the Framework core, where they will be used as input for a damper spring model. To implement a ultrasound sensor, the distance from the virtual vehicle and a virtual objects can be requested to be sent by *Unreal Engine*. Moreover, the physical engine has a wide variety of implemented functions, as sound propagation.

Now, we describe in detail how the binding with the framework core is realized. *Unreal Engine* is programmed in C++ and exploits the Object-oriented programming to ease the design of the desired application. As an example, the class *Actor* represents every object that can be placed into the world. The Class *Pawn*, which extends *Actor*, represents those actors that can be controlled by players or AI. The base of the game framework is the *GameMode* class that sets the rules for the game and many other fundamental features. It also handles spawning the players. All the classes may implement a *Tick()* function that is executed periodically (at each frame or at a minimum time interval). Any periodic action is performed inside this function.

The communication between *Unreal Engine* and the simulator is performed using the UDP protocol. In particular, it is possible to interact with the GameState class to create and delete new objects/vehicles. Then, a one-to-one UDP connection between the SE Thread and the vehicle instance in *Unreal Engine* is created. In this way it is possible to set the pose of the vehicle, start/stop the reception of sensory data, and periodically require the collision state of the vehicle.

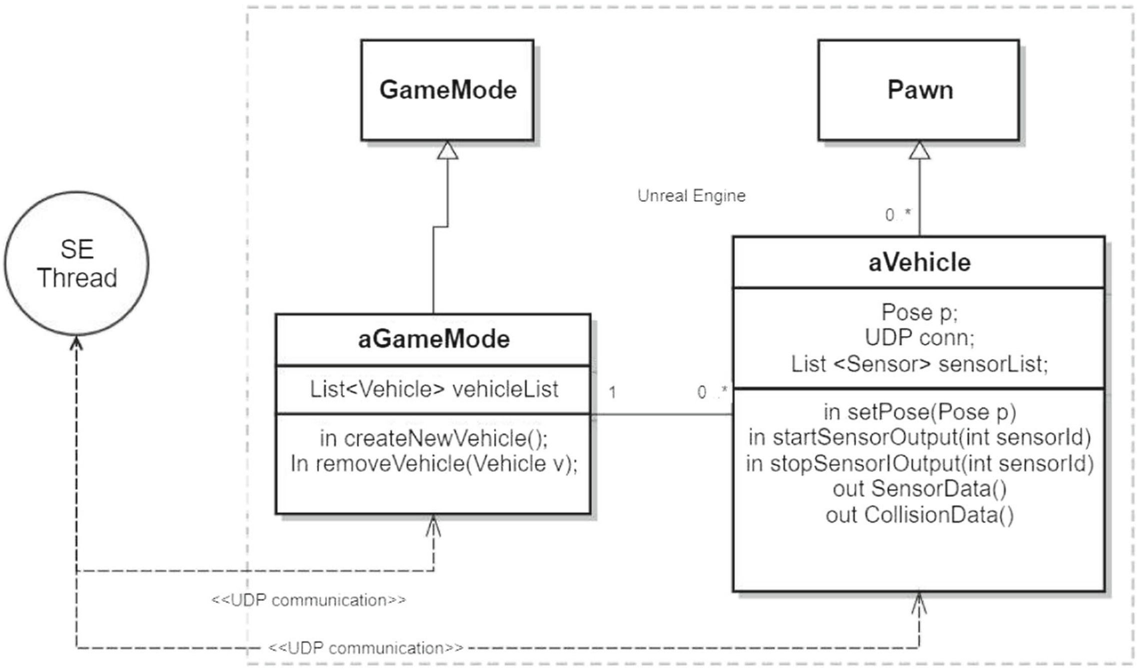


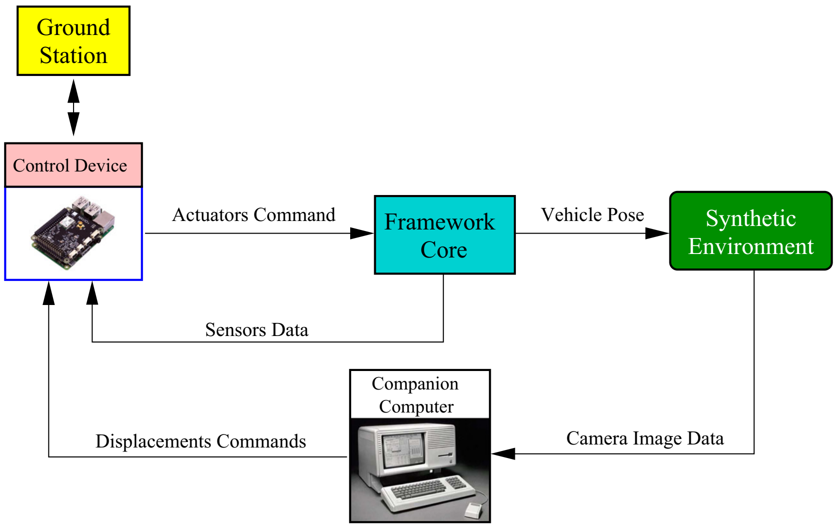
Fig. 10 UML implementation of the *Unreal Engine* classes interacting with the Framework core

Figure 10 provides a UML implementation of the add-ons to the *Unreal Engine* framework. The GameState class is redefined to listen to a UDP port for any new request. When a new request arrives, the GameState create/delete a new vehicle in the synthetic world. Each vehicle (extended from pawn) contains the pose of the vehicle, the UDP information for communicating with the corresponding SE thread and sensors implementations. In the following case of study, an RGB camera has been implemented as a sensor for the vehicle.

6 Case of Study

This section describes a case of study to show how the proposed framework can be used to develop an application involving autonomous visual-based landing. Autonomously landing is an interesting feature for an autonomous aerial vehicle, which is much more complex than simply lowering the altitude of the controlled vehicle. In fact, obstacles could be on the landing area, the GPS position could be not precise enough or not available, the

Fig. 11 Case of study setup



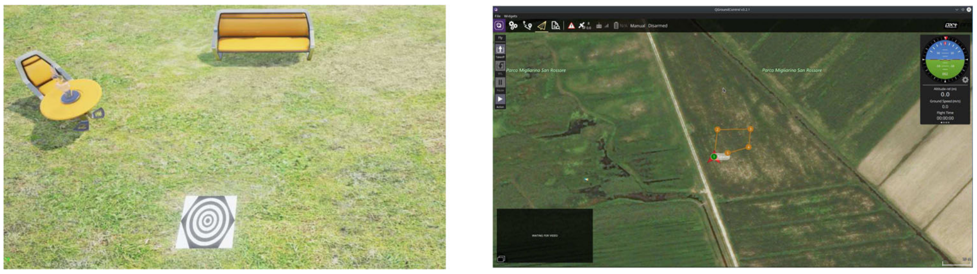


Fig. 12 Screenshots of the case of study: the simulated environment and the ground station used for controlling the vehicle

(a) Synthetic environment for the automatic landing (b) Planned mission with the ground station application

landing pad could be moving, etc. These cases are just a few examples of situations in which the vehicle needs an advanced sensing of the environment and an open-loop maneuver would not suffice to safely accomplish the task. Being able to interact with a dynamic and uncertain environment is a key feature for implementing unmanned systems. Nowadays, such a capability is achieved through the use of computer vision, which involves tasks like feature extraction, filtering, outliers identification, removal of visual artifacts, pattern matching, etc. Testing vision algorithms in real environments is time consuming, hence having the possibility to test the algorithms in a virtual environment would speed up the verification process. For this reason, the following case of study has been specifically selected to show how the proposed framework can be used to design a complex and realistic scenario for testing such advanced visual functions.

6.1 Problem Description

The proposed scenario consists of a quadcopter, equipped with a gimballed camera, that has to accomplish an autonomous mission and then automatically land in a given point of the map, where a landing pad is located. The final landing maneuver is guided by vision to land with a sufficient accuracy over the prescribed landing platform.

The landing pad has been the same shape as the one used by Lange et al. in their work on autonomous landing [10]. The idea was to use a simple pattern, as a series of concentric black and white rings, which can be recognized with simple vision algorithms. The system setup used for the experiment is shown in Fig. 11 and it consists of the components described below.

6.2 System Components

– Navio+ board The board runs the *FlightStack* autopilot firmware and is connected to the Simulation Framework.
– Framework Core The Framework Core simulates the dynamics of the quadrotor using the received control commands evaluated by the autopilot board and providing back sensor data to it.
– Synthetic Environment The Synthetic Environment has been used to represent a realistic landing scenario, with a landing pad and other objects to test the vision algorithm. A screenshot of the simulated scenario is shown in Fig. 12a. This component also simulates the video of the gimballed camera mounted on the quadcopter used for autonomous landing.
– Ground Station The *QGroundControl* Ground Station application is used to set up the waypoints, program the autonomous mission and monitor the state of the system, as during normal operating conditions. Figure 12b shows a screen-shot of the Ground Station interface during the set of the autonomous mission.
– Companion Computer This computer takes the video stream of the simulated camera and carries out the image processing, sending the results, in the form of positioning command, to the autopilot board.

Originally, the Companion Computer was not intended to be used in the case of study, and all the computation was supposed to be done on the Navio+ board. Unfortunately, this solution was not adopted because, during tests, it was discovered that the autopilot board was not able to process the image with a sufficient rate. Indeed, being unable to keep the pace, of $15Hz$ for 640×480 resolution, at which the images were produced, the adopted control strategy did not work and the vehicle did not converge over the landing pad. This problem could be solved by changing the control software or the image processing algorithm by a different implementation, updating the hardware with a more powerful computing platform. In our case of study, the employment of a Companion Computer turned ou This fact stresses the importance of having a simulation system able to run with a hardware-in-the-loop approach. Indeed, with the proposed Simulation Framework, it was possible to discover this problem before testing it on the real vehicle.

6.3 Automatic Landing Control Flow

The control architecture to carry out the automatic landing is illustrated in Fig. 13, whereas the landing problem is schematically represented in Fig. 14, where P_v is the

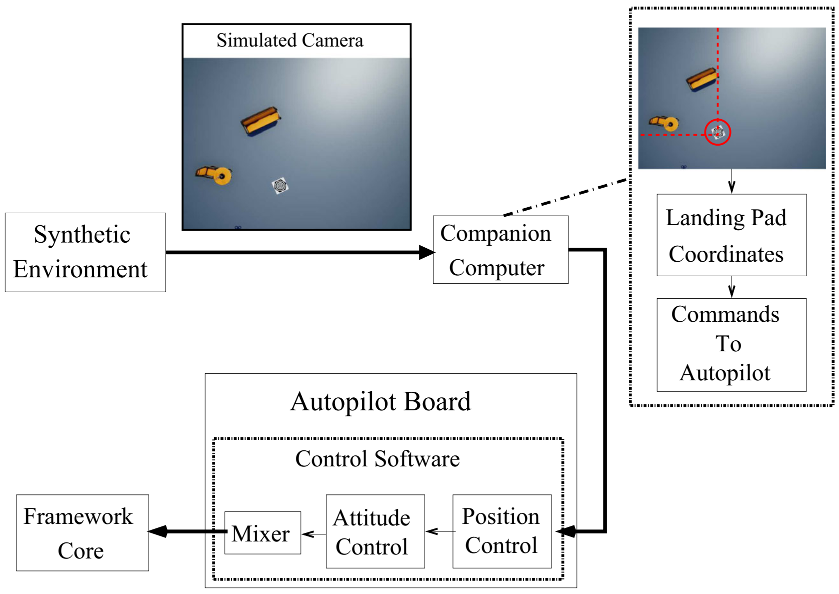


Fig. 13 Diagram of the control flow for the automatic landing maneuver

position of the vehicle with respect to the Navigation Frame, estimated by the onboard navigation algorithms; P_t is the position of the target with respect to the Navigation Frame, and T is the position of the target with respect to the Body Frame.

The image from the camera simulated by the Synthetic Environment is processed by the Companion Computer to identify the landing pad center and calculate its relative position with respect to the vehicle Body Frame (T). The computer vision algorithm has been developed with the *Image Processing Toolbox* of *Matlab/Simulink* and then the C code has been autogenerated to be executed in a custom application. In this experiment, the Companion Computer runs the *Linux* operating system and the application has been written with the support of the *POSIX PThread* library. Once the position of the landing pad with respect to the vehicle Body Frame is known, it is possible to define the reference point (P_t) in the Navigation Frame with respect to the $Home$ starting point of the mission, for the position control loop running on the autopilot board. The communication between the Companion Computer and the autopilot board is accomplished with the *MAVLINK* protocol, using the *MAVLINK* positioning commands. Once the Autopilot receives those commands, the onboard control loops makes the vehicle to descend on the center of the landing pad. The control software used in this case of study is the native one, implemented in the *PX4 Flightstack*.

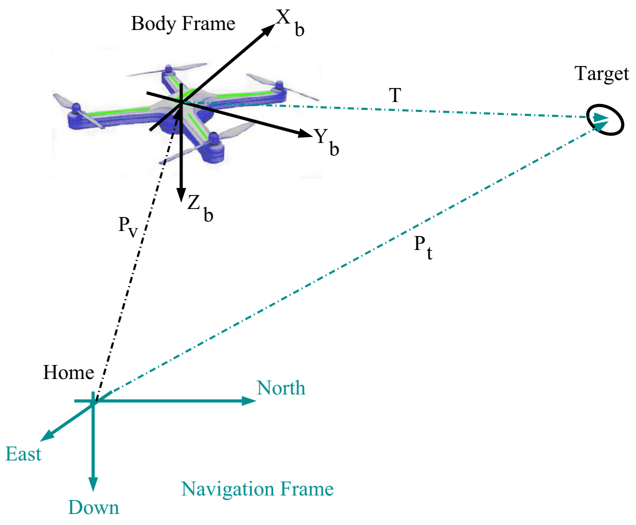


Fig. 14 Schematic representation of the automatic landing problem with the involved reference frames

6.4 Simulation Results

To test the landing algorithm, the vehicle mission was to reach a landing point located near the landing pad. Precisely, the initial displacement of the pad with respect to the landing point in the mission was $[2m,\ 6m,\ 12m]$ along the North, East, Down axes of the navigation frame. From this initial position, the algorithm was able to identify the pad and lands on its center. Figure 15 plots the position error as a function of time during the descent phase. Note that the small discrepancy that can be noted in the initial positioning is due to the drone oscillations while hovering. The landing maneuver took about 25 s to complete and the final landing position had an error of about 5 cm and 7 cm along the North and East axes, respectively.

Summarizing, the aim of this example was both to show a possible employment of the Simulation Framework for a case of study involving autonomous vehicles and highlight the insight it could give over design choices. In this case, the initial idea to run the image processing directly on the

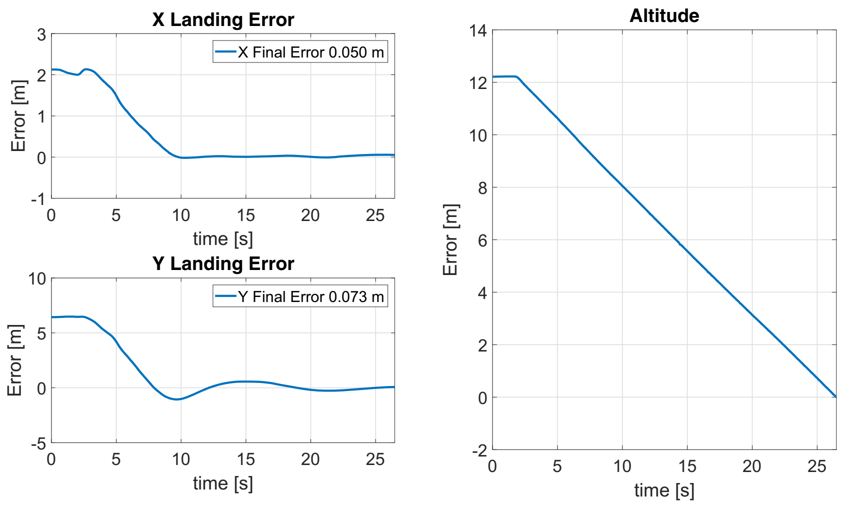


Fig. 15 Position errors as a function of time during the automatic landing procedure

autopilot board was not feasible, and the control architecture was changed during the validation phase.

7 Experimental Results

This section presents the results of the measurements made during simulations to quantify the achievable realism of the framework. Two aspects of the simulation have been considered: the vehicle simulation part, carried out in the Framework Core, and the integration of the Synthetic Environment.

In the experiments, two different setups have been used. In the first one, the Simulation Framework was connected with multiple instances of the control software (*PX4 Flight Stack* firmware), both allocated on the physical autopilot boards (two *Raspberry 2* extended with the *Navio+*[4] modules and a *Pixhawk*[5]) and on a Linux machine running them as processes. This setup was used to extract information about the performance of the Framework Core. In the second setup, consisting in the case of study presented in Section 6, there was only one physical autopilot board connected to the system. This setup was used to make the measurements involving the Synthetic Environment. In both setups, the measures were performed by connecting a logic analyzer to the free I/O pins of one autopilot board (*Raspberry Pi 2*) and recording their toggling, associated with specific events. This allowed having a minimally invasive measure with a sufficient time resolution. The Framework Core executed on a dedicated Linux PC equipped with an $Intel\ Core\ 2\ Duo$ - $E8500$ @ $3.16Ghz$ CPU and $3GB$ of RAM, whereas the Ground Station and the Synthetic Environment were running on another PC. This latter was equipped with an $Intel\ Core\ i7$ - $4790K$ @ $4\ Ghz$ CPU, $16GB$ of RAM, $NVIDIA\ GTX\ 750\ Ti$ GPU and was connected to the Framework Core via Ethernet on a dedicated LAN.

[4]https://emlid.com/ (Navio Linux autopilot on Raspberry 2)
[5]http://px4.io/ (PX4 Pro Open Source Autopilot)

7.1 Timing Properties of the Framework Core

The timing performance of the Framework Core has been assessed by measuring the interaction times in the simulation loop, varying the number of connected vehicles. Figure 16 illustrates an example of the execution pattern of the involved tasks, under the *Simulator Driven Synchronization approach*, where τ_{sim} is the periodic simulation thread, with period T_{sim}, while τ_{ctr} and τ_{snd} are the control and the communication tasks running on the autopilot board, respectively. The measured events are the reception time t_{sns} of a new sensor data, the actuation time t_c, and the instant t_{cs} at which the control data is sent to the simulator. With these measures it was possible to calculate the latency and timing precision properties of the simulation loop.

The simulation latency has been computed as the difference between the time at which the actuation message is sent to the simulator and the time at which the corresponding sensor data is received back, that is $L_{sim} = t_{sns}(k+1) - t_{cs}(k)$. This quantity is important for the realism of the simulation, since it represents the delay introduced by the Simulation Engine, that is, the time taken by the actuation signal to be actuated and make its effects on the vehicle. The timing precision of the simulator has been measured by recording the inter-arrival time of the sensory data ($T_{sns} = t_{sns}(k+1) - t_{sns}(k)$) that triggers the execution

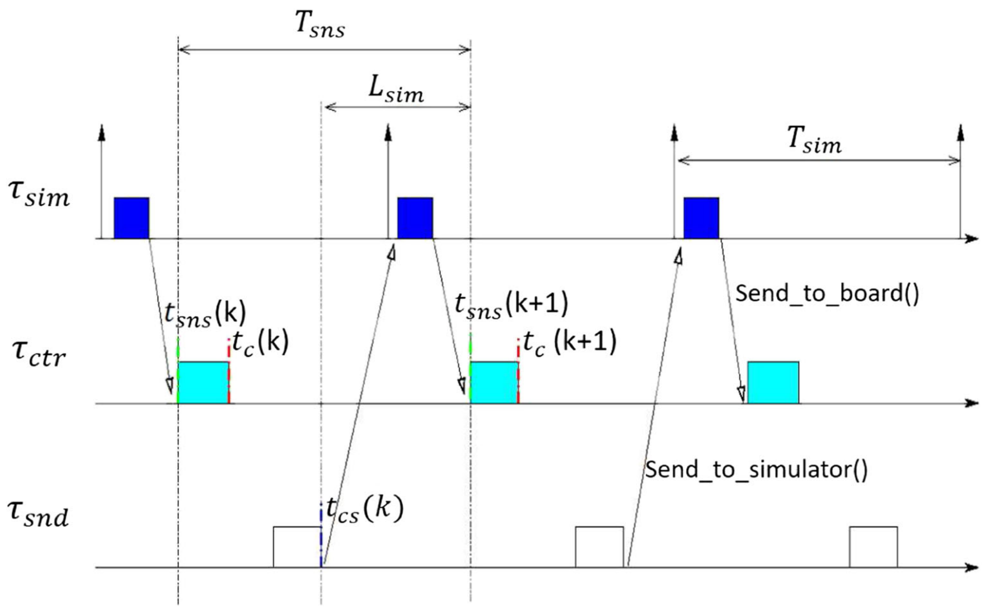


Fig. 16 Example of execution behavior

Table 1 Latency as a function of the number of connected vehicles

Num. Vehicles	3	10	15
Latency mean value	0.445 ms	0.448 ms	0.450 ms
Latency Std	0.089 ms	0.091 ms	0.101 ms

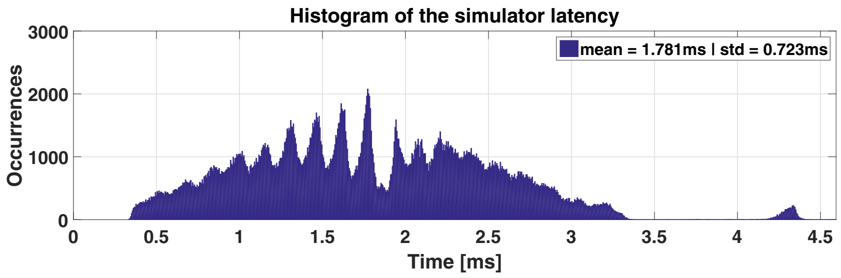


Fig. 17 Simulator latency measured when using the original *px4* firmware

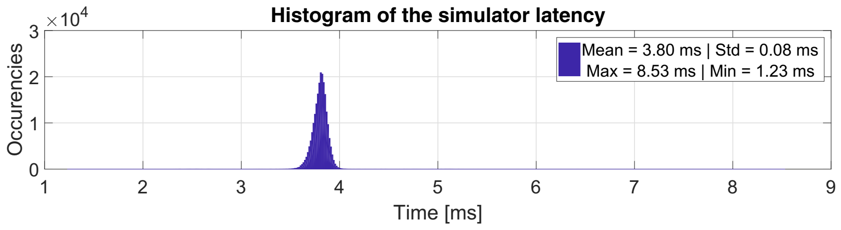


Fig. 18 Simulator latency measured when using the customized *px4* firmware

of control tasks in the *PX4 Flight Stack*. The actuation time ($t_c(k)$) has also been recorded to check the behavior of the autopilot board while interacting with the simulator.

7.1.1 Timing Properties of the Framework Core: Results

The simulation latency, measured during 30 minutes of simulation, with 3, 10, and 15 vehicles, exhibited a linear trend with respect to the number of vehicles. The results are summarized in Table 1. However, the differences in performance between the three cases resulted to be negligible, with a simulation latency variation in the order of 5 μs, confirming a good scalability of the Simulation Framework. Note that, having a latency under 0.5 ms allows running periodic controls with a frequency up to 1 KHz.

The Framework Core resulted capable of triggering the autopilot board with sufficient precision. Measuring the the sensor data inter-arrival times, when requiring a period of 4 ms, it was 4.001 ms with a standard deviation of 0.022 ms. Note that such results were obtained on a machine running a standard Linux kernel; hence they can be improved by using real-time extensions or executing the autopilot software on a real-time operating system.

It is worth noting that the characterization of the latency has been carried out with the *board driven synchronization approach*.With this method, the simulator waits for the trigger of the autopilot board, and the measured time is the actual response time of the Framework Core. The latency measured when running the *simulation driven synchronization approach* is less representative because it is influenced by the implementation of the autopilot software. The original *PX4* firmware, using the *simulation driven synchronization approach*, is characterized by poor results concerning latency. This is due to the not optimized implementation of the communication routine. Precisely, the measured results were characterized by a high variance (about 1 ms), as can be seen in Fig. 17, which shows the latency measured with the original firmware.

A modification of the autopilot firmware was introduced[6] to improve the precision of the data exchange, obtained isolating the communication with the simulator in a dedicated task. Note that, this approach has been selected by the developers of the mainline *PX4* firmware, but its implementation is still is progress. The latency obtained with the modified firmware is shown in Fig. 18 and is characterized by a smaller standard deviation. These results should be interpreted considering the software behavior, which is shown in Fig. 16. Indeed, the fact that the mean value of the latency with the original firmware was smaller is a sign that the communication of the control value was delayed with respect to its computation.

7.2 Timing Properties of the Synthetic Environment

Another set of experiments have been done to assess the timing properties of the communication with the Synthetic Environment. The interaction with this module involves signaling of events generated by sensors, collisions with surfaces, and video streaming from virtual cameras. The time taken to provide such data to the Simulation Engine, the control board or a companion computer directly affects the realism of the simulation. For this reason, some tests have been done to measure the time taken to receive a collision event feedback or an image from the simulated camera. Figure 19 illustrates an example of schedule of the events involved in the interaction with the Synthetic Environment.

Referring to the software implementation represented in Fig. 8 of Section 4, the *Sim Thread* the *Sim Thread* periodically updates the state of the simulated vehicle, while the *SE Thread* waits for the data from the Synthetic Environment (Impact Msg). The execution period of the Synthetic Environment is computed as $T_{se}(k+1) = t_i(k+1) - t_i(k)$, setting the achievable update rate. The figure also shows the reception of a new image from the Synthetic Environment. Note that the latency introduced in processing the information about the impact (Impact Latency) is given by $L_{impact} = t_s(k+1) - t_s(k)$. Indeed, the impact is signaled by the Synthetic Environment with the message at $t_i(k)$, when the state of the impacting vehicle was computed by the *Sim Thread* at the time instant $t_s(k)$; the impact information is received at $t_{snd}(k+1)$ and the reaction is considered only at the next execution of the *Sim Thread*, $t_s(k+1)$. This reasoning is also valid for other kind of interactions, different from the impact. As an example, when retrieving distance measures to simulate a laser range sensor, that time value would represent the latency of the simulated sensor. It follows that, the results relative to the Impact Latency shown in the following should be interpreted as a general timing behavior for any kind of sensors/interaction.

As far as the Image Latency is considered, its value can be calculated as $L_{image} = t_p(k) - t_s(k)$, where $t_p(k)$ is the time at which the image message, evaluated considering the state computed at time $t_s(k)$, is received. Measuring this quantity it is important to quantify the achievable timing precision and accuracy for the simulated sensors. The maximum latency that can be experienced during impacts and image acquisitions can be estimates as

$$L_{impact}^{max} = \max(T_{se}) + 2T_{sim}, \text{ and} \quad (1)$$
$$L_{image}^{max} = \max(T_{se}) + T_{sim} + \max(L_{se}), \quad (2)$$

[6]The modified version of the *PX4* firmware is available online on GitHub: https://github.com/rt-2pm2/Firmware

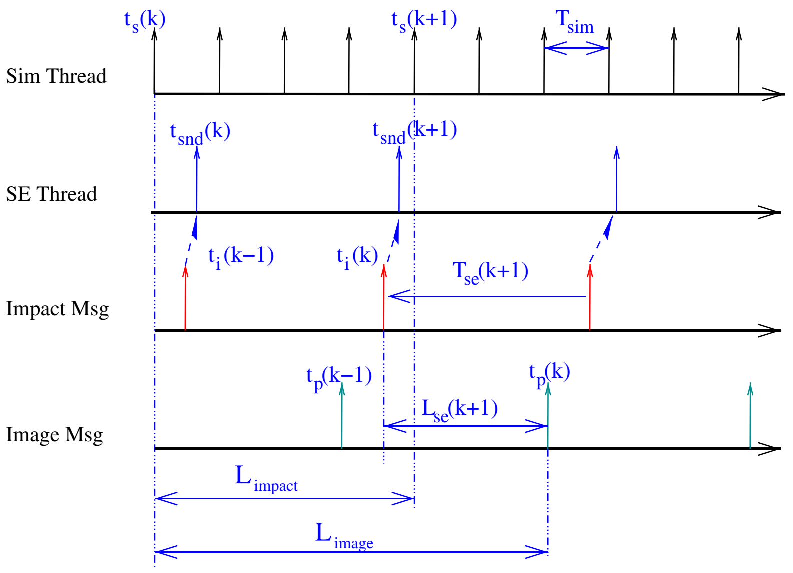


Fig. 19 Example of execution behavior of the interaction with Synthetic Environment

In the experiment, the Impact Latency and Image Latency have been evaluated also considering their dependency on some case specific parameters. In particular, the following relations have been considered:

1. Image Latency as a function of Image Dimension (L_{image})
2. Frame rate as a function of Image Dimension ($\frac{1}{T_{se}}$)
3. Impact Latency as a function of Image Dimension (L_{impact})

The simulation lasted 15 minutes, which is in the timing range of a typical autonomous mission with a quadrotor holding a camera gimbal. The resolution of the images has been chosen among the ones commonly used in applications involving autonomous vehicles, that is, 320x240, 400x300 and 640x480 pixels, with a dimension of about 2.5 KB, 3.2 KB and 6.5 KB, respectively.

7.2.1 Timing Properties of the Synthetic Environment: Results

Figure 20 shows the Image Latency distribution obtained for different image resolutions.

As it can be expected, the image latency increases with respect to the image resolution, reaching a mean value of 126.7681ms for the case of 640x480 pixels. This values are coherent with the images inter-sample times, which are reported in Fig. 21.

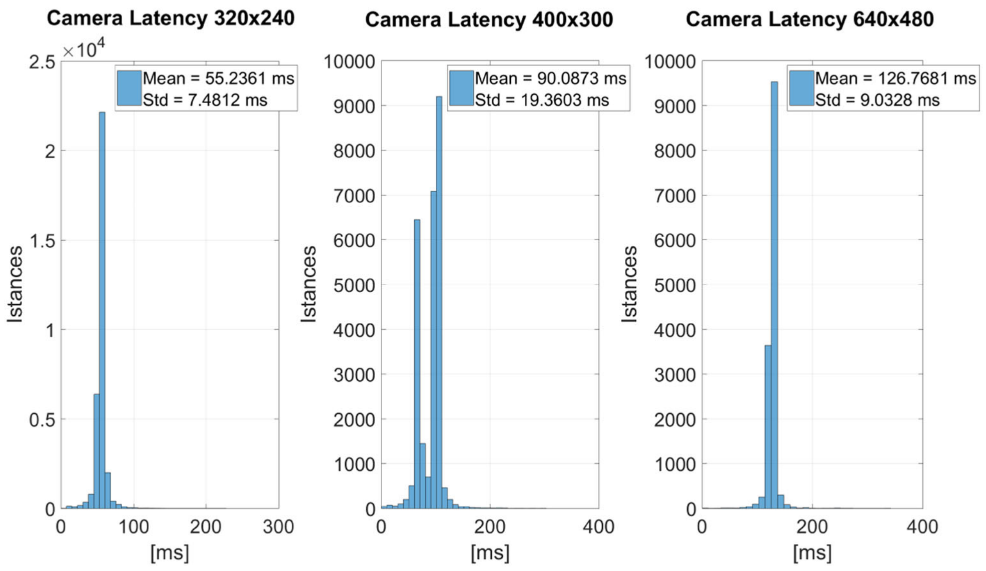


Fig. 20 Time latency (L_{image}) between the generation of image frames and the reception of the frame on the target board

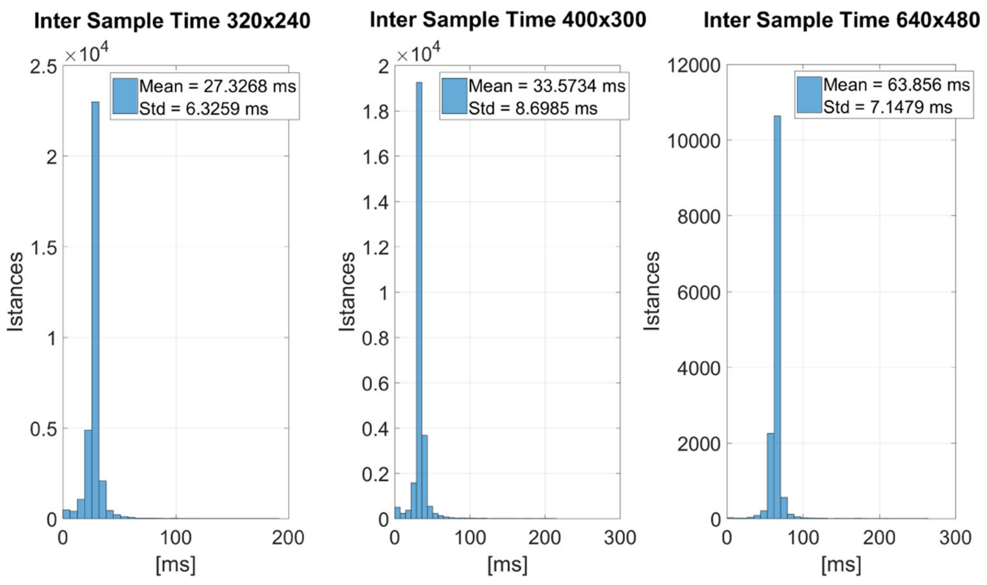


Fig. 21 Inter sample time (T_{se}) for different image dimensions

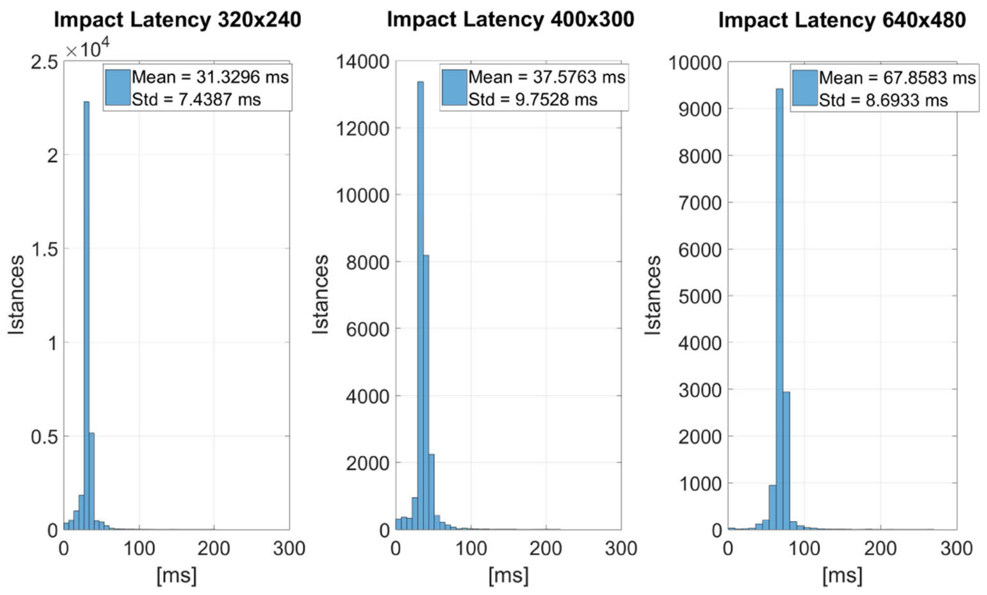


Fig. 22 Time latency (L_{impact}) between the generation impact over a surface and the reception of the event on the target board when streaming images

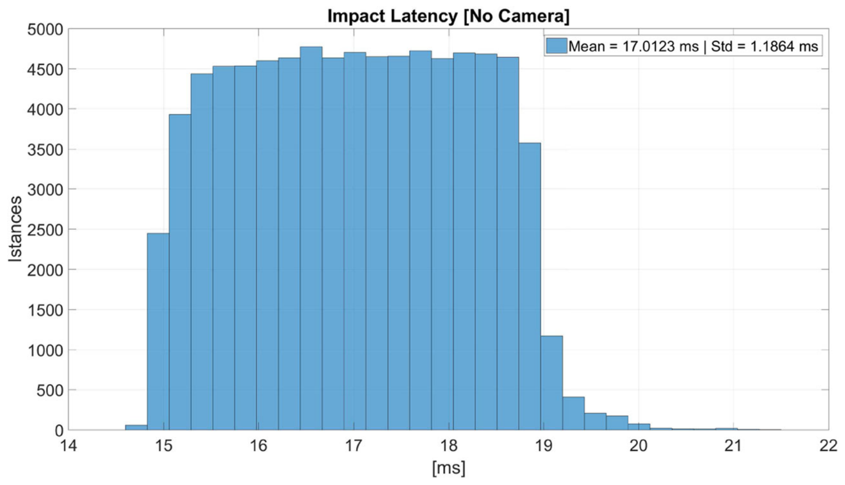


Fig. 23 Time latency (L_{impact}) between the generation impact over a surface and the reception of the event on the target board when not streaming images

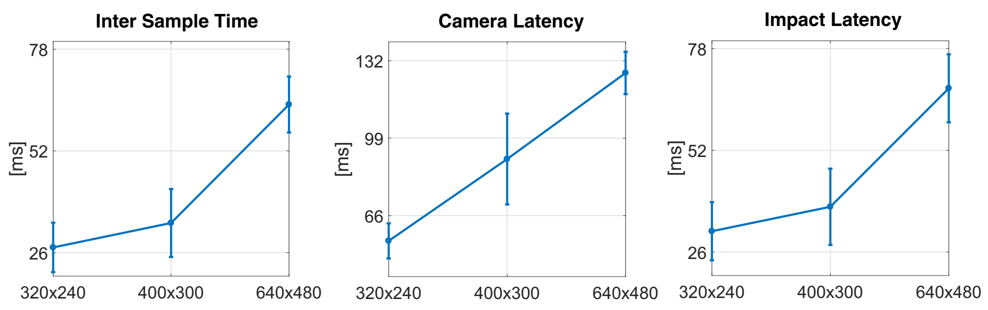


Fig. 24 Trend of the T_{se}, T_{image} and T_{impact} as a function of the image dimension

The values of these times correspond to an update rate of approximately $36Hz$, $30Hz$ and $15Hz$ for the case with 320x240, 400x300 and 640x480 pixels, respectively. The values of the impact latency, shown in Fig. 22, match with the inter-sample time of the images.

This is rational, since the *Unreal Engine* timing is ruled by the rendering time of the frames. It is thus expected to get something in the order of the inter-sample time of the images. The variations from this latter time are due to the effective execution pattern of the threads and, as noticed when estimating the worst-case latency, are in the order of twice the simulation period.

Since the presence of the simulated camera heavily influenced the impact latency, this latter quantity was measured also in the case without camera simulation. The results in Fig. 23 show that it is possible to get better performance when the camera is not needed.

The recorded latency was consistent with the max *Unreal Engine* framerate, which was set to $100Hz$. The mean value recorded is about 17 ms. This results can be considered not sufficient to simulate fast response sensors. As an example, it means that, simulating the vehicle dynamics at $250Hz$, after an impact, it will take five simulation cycles before getting a feedback from the Synthetic Environment. Given the limitations of the *Unreal Engine* update rate, this results can be improved using some mechanism to predict possible impacts and anticipate the reaction. An overall representation of the obtained results is reported in Fig. 24, where the increasing trend of the latency and sample times with respect to the resolution of the image is noticeable.

8 Conclusion

This work presented a Simulation Framework for testing and validating multi-vehicle complex control algorithms. The Framework supports software and hardware-in-the-loop simulations with multiple vehicles and allows modeling complex testing scenario, considering also the graphical realism. In particular, it is possible to test and validate high-level control algorithms that require the interaction with a synthetic world. For this reason the physical world is simulated using *Unreal Engine*, a tool used for design computer games that provides high-realism in terms of visual output and the possibility of interacting, both with the static environment and with dynamic objects such as people and cars.

The structure of the Simulation Framework has been designed to be modular, assuring maintainability, scalability, and interoperability with different user interfaces and control hardware. Leveraging on the capabilities of Matlab/Simulink to define models and generate the related C/C++ code, it was possible to achieve full control on the physical modeling, both in terms of integration techniques and sampling time of the simulation. The communication among different components has been implemented paying particular attention to timing properties and avoiding unnecessary blocking times in the data exchange.

A case of study was used to show the effectiveness of the proposed framework in tackling design purposes, which included high-level functions of the vehicles. An interesting result obtained with such a case of study was to understand that the designed algorithm, expected to be run directly on the control board, was computationally too costly for the embedded hardware. Thanks to the proposed framework, it was possible to recognize this issue before performing real flights, re-designing the system architecture to comply with the appropriate timing requirements.

Several experiments have been run to assess the capability of the Simulation Framework to support multi-agent scenarios while guaranteeing precise timing properties to achieve a more realistic behavior. The obtained results showed that increasing the number of vehicles does not impact on the simulator performance, making it a useful tool for managing large fleets of robots. The experiments were also intended to provide a characterization of the timing performance of the Simulation Framework, both in terms of the Simulation Core and the Synthetic Environment.

As a future work, the results of the simulation will be compared with the ones obtained via real experiments, to validate the realism of the Simulation Framework. To accomplish this task, an identification procedure is necessary to ensure that the parameters used in the model match the ones of the real vehicle, together with the hardware setting used for the experiments.

References

1. Ali, K.S., Shumaker, J.L.: Hardware in the loop simulator for multi agent unmanned aerial vehicles environment. Amer. J. Eng. Appl. Sci. **6**, 172–177 (2013)
2. Birk, A., Poppinga, J., Stoyanov, T., Nevatia, Y.: Planetary Exploration in USARsim: A Case Study Including Real World Data from Mars, pp. 463–472. Springer, Berlin (2009)
3. Bryson, M., Reid, A., Ramos, F., Sukkarieh, S.: Airborne vision-based mapping and classification of large farmland environments. J. Field Robot. **27**(5), 632–655 (2010)
4. Carpin, S., Lewis, M., Wang, J., Balakirsky, S., Scrapper, C.: Usarsim: a robot simulator for research and education. In: Proceedings 2007 IEEE International Conference on Robotics and Automation, pp. 1400–1405 (2007)
5. Castillo-Pizarro, P., Arredondo, T.V., Torres-Torriti, M.: Introductory Survey to Open-Source Mobile Robot Simulation Software. In: 2010 Latin American Robotics Symposium and Intelligent Robotics Meeting, pp. 150–155. https://doi.org/10.1109/LARS.2010.19 (2010)
6. Cook, D., Vardy, A., Lewis, R.: A Survey of Auv and Robot Simulators for Multi-Vehicle Operations. In: 2014 IEEE/OES Autonomous Underwater Vehicles (AUV), pp. 1–8. https://doi.org/10.1109/AUV.2014.7054411 (2014)
7. Echeverria, G., Lassabe, N., Degroote, A., Lemaignan, S.: Modular Open Robots Simulation Engine: Morse. In: 2011 IEEE International Conference on Robotics and Automation, pp. 46–51. https://doi.org/10.1109/ICRA.2011.5980252 (2011)
8. Ganoni, O., Mukundan, R.: A framework for visually realistic multi-robot simulation in natural environment. CoRR arXiv:1708.01938 (2017)
9. Kamali, C., Jain, S.: Hardware in the Loop Simulation for a Mini Uav. In: 4Th IFAC Conference on Advances in Control and Optimization of Dynamical Systems ACODS 2016, Vol 49, pp. 700–705. Tiruchirappalli, India (2016)
10. Lange, S., Sunderhauf, N., Protzel, P.: A Vision Based Onboard Approach for Landing and Position Control of an Autonomous Multirotor Uav in Gps-Denied Environments. In: 2009 International Conference on Advanced Robotics, pp. 1–6 (2009)
11. Lugo-Cardenas, I., Salazar, S., Lozano, R.: The Mav3dsim Hardware in the Loop Simulation Platform for Research and Validation of Uav Controllers. In: 2016 International Conference on Unmanned Aircraft Systems (ICUAS), pp. 1335–1341 (2016)
12. Lum, C., Rowland, M., Rysdyk, R.: chap. Human-in-the-Loop Distributed Simulation and Validation of Strategic Autonomous Algorithms. Fluid Dynamics and Co-located Conferences. American Institute of Aeronautics and Astronautics. https://doi.org/10.2514/6.2008-4366.0 (2008)
13. Meier, L., Honegger, D., Pollefeys, M.: Px4: a node-based multithreaded open source robotics framework for deeply embedded platforms. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), pp. 6235–6240 (2015)
14. Merino, L., Caballero, F., Martinez-de Dios, J., Maza, I., Ollero, A.: An unmanned aircraft system for automatic forest fire monitoring and measurement. J. Intell. Robot. Syst. **65**, 533–548 (2012)
15. Mueller, E.R.: Hardware-in-the-loop simulation design for evaluation of unmanned aerial vehicle control systems. In: Proceedings of the AIAA Modeling and Simulation Technologies Conference and Exhibit (2007)
16. Odelga, M., Stegagno, P., Bülthoff, H.H., Ahmad, A.: A Setup for Multi-Uav Hardware-In-The-Loop Simulations. In: 2015 Workshop on Research, Education and Development of Unmanned Aerial Systems (RED-UAS), pp. 204–210. https://doi.org/10.1109/RED-UAS.2015.7441008 (2015)
17. Pannocchi, L., Marinoni, M., Buttazzo, G.: Hardware-In-The-Loop Development Framework for Multi-Vehicle Autonomous Systems. In: 2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC), pp. 17–22. https://doi.org/10.1109/ICARSC.2017.7964046 (2017)
18. Parodi, O., Lapierre, L., Jouvencel, B.: Hardware-in-the-loop simulators for multi-vehicles scenarios: survey on existing solutions and proposal of a new architecture. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 225–230 (2009)
19. Pollini, L., Parnenzini, V., Innocenti, M.: Distributed Real-Time Hardware- and Man-In-The-Loop Simulation for the Icaro Ii Unmanned Systems Autopilot. In: Latest Trends in Information Technology/Recent Advances in Computer Engineering Series 7, pp. 420–427 (2012)
20. Posch, A., Sukkarieh, S.: Uav Based Search for a Radio Tagged Animal Using Particle Filters. In: Australasian Conference on Robotics and Automation (ACRA). Sydney, Australia (2009)
21. Barros dos Santos, S.R., Givigi, S., Nascimento, C.L.J., Oliveira, N.: Modeling of a hardware-in-the-loop simulator for uav autopilot controllers. In: Proceedings of the 21th Brazilian Congress of Mechanical Engineering (COBEM 2011), Natal, Brazil (2011)
22. Sehgal, A., Cernea, D.: A Multi-Auv Missions Simulation Framework for the Usarsim Robotics Simulator. In: 2010 18Th Mediterranean Conference On Control Automation (MED), pp. 1188–1193. https://doi.org/10.1109/MED.2010.5547632 (2010)
23. Shah, S., Dey, D., Lovett, C., Kapoor, A.: Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In: Field and Service Robotics. arXiv:1705.05065 (2017)
24. Takaya, K., Asai, T., Kroumov, V., Smarandache, F.: Simulation Environment for Mobile Robots Testing Using Ros and Gazebo. In: 2016 20Th International Conference on System Theory, Control and Computing (ICSTCC), pp. 96–101. https://doi.org/10.1109/ICSTCC.2016.7790647 (2016)

Luigi Pannocchi is a Ph.D. Student of the ReTiS Lab at the Scuola Superiore Sant'Anna, Pisa. He received his MSc cum Laude in Robotics and Automation Engineering at the University of Pisa in 2015. Since then, he joined the Ph.D. in Emerging Digital Technologies, curriculum Embedded System at the ReTiS Lab of the Scuola Superiore Sant'Anna, Pisa, under the supervision of Professor Giorgio Buttazzo. From January 2018 he was a visiting Ph.D. student for 6 months at the Cyber-Physical Systems Laboratory (CyPhyLab) of University of California Los Angeles under the supervision of Paulo Tabuada. His research topics are Cyber Physical Systems, in particular Autonomous Aerial Vehicles, and Simulation Environments.

Carmelo Di Franco is a Post Doctoral Researcher of the ReTiS Lab at the Scuola Superiore Sant'Anna, Pisa. In March 2017, he concluded cum Laude is Ph.D. in Emerging Digital Technologies, curriculum Embedded System at the ReTiS Lab of the Scuola Superiore Sant'Anna, Pisa, under the supervision of Professor Giorgio Buttazzo. From September 2015 he was a visiting Ph.D. student for 6 months at the G.R.A.S.P. Laboratory of University of Pennsylvania under the supervision of George J. Pappas. His research topics are Indoor Localization and energy-aware path planning of mobile robots.

Mauro Marinoni is Assistant Professor at the Scuola Superiore Sant'Anna in Pisa. He received his MSc in Computer Engineering at the University of Pavia (Italy) in 2003 where he also obtained his Ph.D. in Computer Engineering in 2007. He has been working since 2007 at the Institute of Communication, Information and Perception Technologies, where he is now area leader of Resource Management at the Real-Time Systems Laboratory (ReTiS). His leading research topics cover Scheduling Theory, Operating Systems and Energy management for Real-Time systems, focusing on the integration of Real-Time enhancements in different application fields, from e-Health devices to autonomous systems and distributed systems. He has been local coordinator of the FP7 JUNIPER project as well as several industrial projects exploiting the ReTiS Lab research outcomes.

Giorgio Buttazzo is full professor of computer engineering at the Scuola Superiore Sant'Anna of Pisa. He graduated in electronic engineering at the University of Pisa in 1985, received a M.S. degree in computer science at the University of Pennsylvania in 1987, and a Ph.D. in computer engineering at the Scuola Superiore Sant'Anna of Pisa in 1991. From 1987 to 1988, he worked on active perception and real-time control at the G.R.A.S.P. Laboratory of the University of Pennsylvania, Philadelphia. He has been Program Chair and General Chair of the major international conferences on real-time systems and Chair of the IEEE Technical Committee on Real-Time Systems. He is Editor-in-Chief of Real-Time Systems, Associate Editor of the ACM Transactions on Cyber-Physical Systems, and IEEE Fellow since 2012. He has authored 7 books on real-time systems and over 200 papers in the field of real-time systems, robotics, and neural networks.