

# Chapter 4

## Simulation

This chapter describes, in detail, the devised simulation environment. Section 4.1 recalls the overall design adopted for the simulation framework. Section 4.2 introduces Gazebo, the robotics simulator selected for the system. Section 4.3 presents the user interface developed to spawn and simulate vehicles in Gazebo and to run PX4 SITL instances. Section 4.4 describes the software emulation of the physical motion capture system. Section 4.5 discusses solutions to reduce simulation overhead. Finally, Section 4.6 provides a complete overview of the simulation setup, summarizing all the information presented over the chapter.

### 4.1 Architecture of the simulation environment

The overall architecture of the simulation environment of the Flying Arena was presented and discussed in Chapter 2. A slightly different representation of that architecture is depicted in Fig. 4.1.

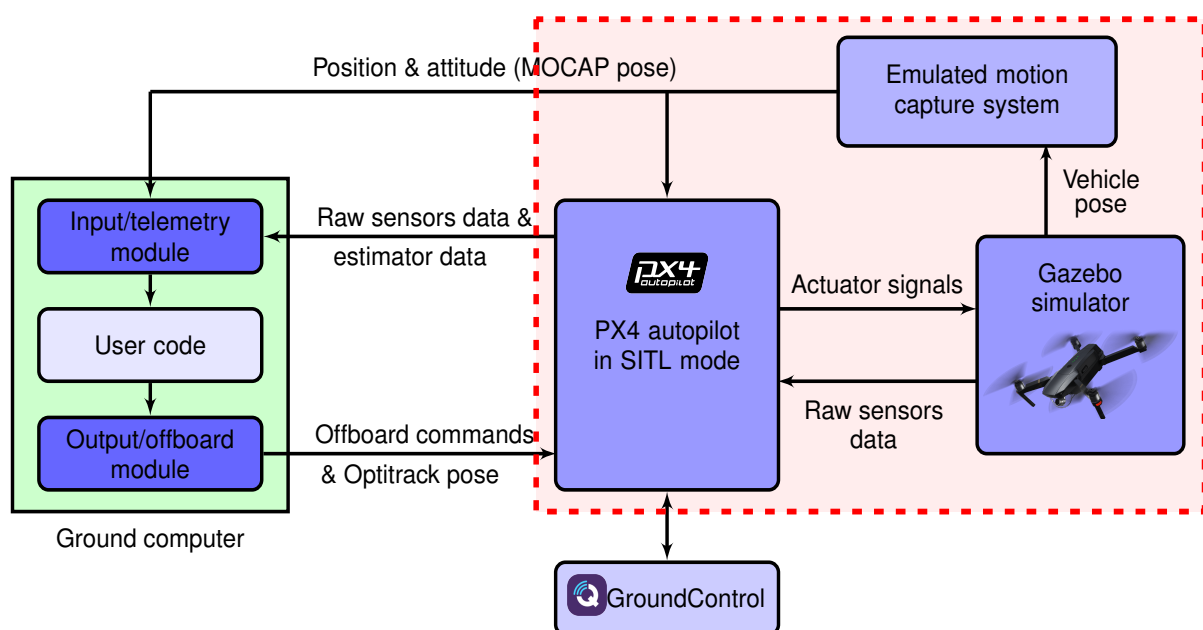


Figure 4.1: Architecture of the simulation environment.

The blocks inside the red area correspond to the ones that change when experiments are conducted in the simulation environment, instead of the real environment (the actual Flying Arena, with real vehicles and real hardware):

- In the Gazebo simulator block, vehicles are represented as physical entities with configurable dynamic and kinematic properties. The behavior of sensors and actuators is modeled by plugins. Based on the actuator signals received and the sensor noise characteristics, the simulator computes the motion of the vehicles and the new sensor readings.
- In the emulated motion capture system block, a software module retrieves the pose of the vehicles from the Gazebo simulator and delivers it to the user and the PX4. In addition, this module also adds adjustable Gaussian noise to the retrieved quantities, to model the uncertainty of the measurements provided by a real motion capture system.
- In the autopilot block, the PX4 communicates with the simulator to receive sensor data and send actuator signals. The PX4 behaves and has the same capabilities in simulation as when connected to actual vehicles.

The detailed implementation and integration of these three blocks is presented over the following sections. The remaining blocks, common to both the real and the simulation environments, will be discussed in Chapter 5.

## 4.2 Gazebo simulator

Simulators allow testing of navigation and control solutions in a quick and safe way. Users can interact with a simulated vehicle just as they might with a real one, by using the QGroundControl software or by running offboard programs on the ground computers to send commands and control references to the PX4 autopilot. Before attempting to fly real vehicles in the ISR Flying Arena, it is recommended to use the simulator to ensure that the estimation and control algorithms work properly and the vehicles behave as expected. This is a key measure to guarantee the safety of users and equipment.

The selected simulator for the devised setup is Gazebo [18], a powerful 3D robotics engine suitable for testing autonomous vehicles. Gazebo was the chosen simulator because: i) it is compatible with the software-in-the-loop capabilities of the PX4 autopilot; ii) it supports all kinds of aerial vehicles, such as multicopters, fixed wing aircrafts, and VTOL drones; iii) it accepts custom models of those vehicles, so that the user can simulate UAVs with dynamic and kinematic properties close to those of the real drones used in the Flying Arena; iv) it offers plugins to simulate the behavior of sensors and actuators; v) it supports multi-vehicle simulation; vi) it is compatible with the ROS middleware, a set of robotics libraries and tools used in this thesis; and finally vii) it is suitable to test object-avoidance and computer vision algorithms, two important research topics at the Institute for Systems and Robotics. The major drawback of the Gazebo simulator is that it is computationally heavy. In multi-vehicle experiments, depending on the number of UAVs simulated, it is necessary to run the simulation in a dedicated computer or even across multiple dedicated computers, for close to real time performance.

Fig. 4.2 shows the message flow between Gazebo and a PX4 autopilot. Gazebo communicates with the PX4 using the MAVLink API [19]. This API defines a set of MAVLink messages that provide raw sensor data from the simulated world to PX4 and receive the actuator signals that will be applied to the motors and controlling devices of the simulated vehicle. A software-in-the-loop build of PX4 uses the `simulator_mavlink.cpp` [20] module to exchange MAVLink messages with Gazebo. It is important to note that the PX4 autopilot and the Gazebo simulator run in lockstep, which means that they wait on each other for sensor and actuator messages, rather than running at their own speeds. Due to the lockstep feature, it is possible to run the simulation faster or slower than real time without losing messages. If a computer is not powerful enough to run a given experiment at the rate requested by the user, the simulation will run at a slower pace. This is only problematic when performing experiments with simultaneously real and simulated UAVs that interact with each other. In such cases, the simulator must run close to real time. To help achieve a real time simulation rate in experiments with numerous vehicles, solutions were developed, as presented in Section 4.5, to reduce simulation overhead. Moreover, the simulation environment was designed to allow simulation of UAVs across multiple Gazebo instances, dividing the required computing power for the experiment over multiple computers.

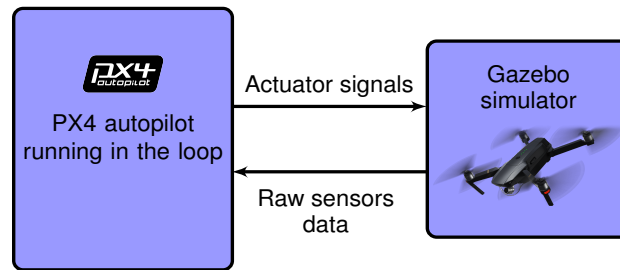


Figure 4.2: Message flow between Gazebo and the PX4.

The installation steps of the Gazebo software are described in detail in the digital repository that complements this thesis. The next section describes how the user can simulate vehicles in Gazebo and run PX4 instances in the loop.

### 4.3 Simulating vehicles and running the PX4 in the loop

Gazebo can operate as a standalone software or within a ROS environment. Launching Gazebo simulations within ROS is advantageous because this middleware provides a set of services and topics to modify and retrieve information about the state of the simulated world and the simulated vehicles [21]. Since these tools allow software programs to retrieve the position and attitude of the drones from the simulator, which are required for the emulation of the motion capture system, it was decided to launch Gazebo within ROS. The services provided by ROS also allow to use Gazebo as a visualization tool, as presented in Chapter 5.

One of the goals of this thesis is the development of an user interface enabling the quick and effortless use of the simulation framework. Fig. 4.3 presents the user interface developed to launch Gazebo within ROS, spawn vehicles in the simulator, and run PX4 SITL instances. It consists in a simple configuration

file, named *gazebo.launch*, that hides the complexity of the simulation launch process.

The *gazebo.launch* file is divided in two different code blocks. The first one, from lines 5 to 8, is responsible for launching Gazebo with ROS. The second one, from lines 15 to 31, is responsible for spawning a vehicle and starting a PX4 SITL instance. For multi-vehicle simulations, multiple blocks of the latter must be created. To launch simulations with this interface, the user just has to fill the arguments of each block with the desired configurations for Gazebo, for the simulated vehicles, and for the PX4 SITL instances. Then, a sequence of software programs operating in the background will automatically launch the simulation, according to the arguments requested. This interface enables the user to benefit from the full capabilities of the simulation environment without prior experience with Gazebo, the ROS tools, or the PX4 firmware. Without the compact interface presented in Fig. 4.3, users would have to extensively edit multiple different files of the PX4 firmware, in order to fully configure the simulation. The next two sections will describe the arguments and the background programs responsible for starting the simulation.

```
1 <?xml version="1.0"?>
2 <launch>
3   <!-- LAUNCH GAZEBO: CHOOSE WHETHER OR NOT TO LAUNCH THE GRAPHICAL INTERFACE.
4   SELECT THE WORLD. -->
5   <include file="$(find gazebo_ros)/launch/empty_world.launch">
6     <arg name="gui" value="true"/>
7     <arg name="world_name" value="$(find mavlink_sitl_gazebo)/worlds/empty.world"/>
8   </include>
9
10  <!-- LAUNCH AN UAV: ASSIGN A UNIQUE NAMESPACE. CHOOSE THE VEHICLE INITIAL
11  POSITION AND INITIAL ATTITUDE. ASSIGN A UNIQUE ID FOR THE DRONE AND FOR THE
12  SYSTEM. ASSIGN A UNIQUE PORT. SELECT THE VEHICLE MODEL. STATE THE FILE WITH
13  THE DESIRED INTERNAL CONFIGURATIONS FOR THE PX4. IF YOU WANT TO CONNECT TO
14  THE DRONE FROM EXTERNAL COMPUTERS, INSERT THE IP ADDRESS OF THOSE COMPUTERS.-->
15  <group ns="uav1">
16    <include file="$(find px4)/launch/drone.launch">
17      <arg name="N" value="0"/>
18      <arg name="E" value="0"/>
19      <arg name="D" value="0"/>
20      <arg name="r" value="0"/>
21      <arg name="p" value="0"/>
22      <arg name="y" value="0"/>
23      <arg name="ID" value="1"/>
24      <arg name="port" value="16001"/>
25      <arg name="vehicle_model" value="iris"/>
26      <arg name="px4_config" value="px4_config"/>
27      <arg name="gcs_1_ip" value="192.168.1.21"/>
28      <arg name="gcs_2_ip" value="192.168.1.22"/>
29    </include>
30  </group>
31 </launch>
```

Figure 4.3: Code of the *gazebo.launch* file: the configuration file used to launch Gazebo, spawn vehicles in the simulator, and run PX4 SITL instances in the loop.

### 4.3.1 Launching Gazebo

The developed user interface, that consists in the *gazebo.launch* file presented in Fig. 4.3, launches Gazebo through an auxiliary program called *empty.world.launch* [22], the standard simulation launcher provided by the ROS API for Gazebo. The *empty.world.launch* program accepts multiple arguments that allow the user to start the simulator with different configurations. In order to present an user interface that is as clean and simple as possible, only the most relevant arguments, those that can be advantageous for the user to change depending on the experiment or on the hardware of the computer used for simulation, were selected for the *gazebo.launch* file:

- The **gui** argument determines whether Gazebo will run with or without the graphical user interface. Performing simulations with the graphical user interface enables 3D visualization of the motion and behavior of the vehicles during the experiments. By launching Gazebo without GUI, the user is not able to visualize the drones during the experiments, but the simulation runs faster and uses less system resources. This is a way to run the simulation with less overhead and is the solution to run Gazebo in computers or servers without a dedicated Graphics Processing Unit (GPU). After the simulation, through the generated log files, the user can recreate and visualize the experiment in 3D in more lightweight tools, which will be presented in Chapter 5.
- The **world\_name** argument specifies the path to the world file that will be launched by Gazebo. In the world file, it is possible to customize basic features of the simulation, such as physics engine parameters, ground plane textures, and lighting. Since the visualization and physics engine of Gazebo are computationally heavy, the default world adopted for this thesis includes the most simple ground plane and global light source plugins, so as to not generate unnecessary overhead.

In the background, the *empty.world.launch* program runs two different executables. The first is called *gzserver* and the second *gzclient*. The *gzserver* runs the physics engine of Gazebo with the configurations of the world selected by the user, and is responsible for computing the motion of the vehicles and generating sensor data. The *gzclient* runs the graphical interface. The scheme of Fig. 4.4 summarizes the launching process of Gazebo. It is important to note that the user only needs to define the values of the **gui** and **world\_name** arguments, the remaining steps of the launching process are automatic.

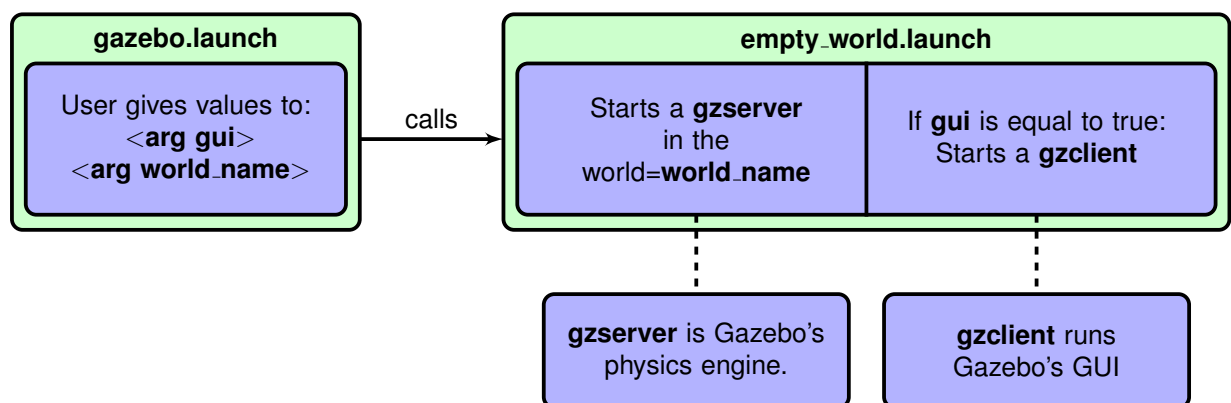


Figure 4.4: Launching process of the Gazebo simulator.

### 4.3.2 Launching vehicles and PX4 SITL instances

The second block of the *gazebo.launch* file (Fig. 4.3) is responsible for spawning a vehicle in the Gazebo simulation and starting a PX4 SITL instance for that specific vehicle. To perform these two tasks, a roslaunch XML program named *drone.launch* is called. The *drone.launch* program is a deeply customized version of the recommended roslaunch XML files available in the PX4 firmware [23]. In addition to launching vehicles and PX4 instances, it was also programmed to automatically define the communication settings between the PX4 SITL and the Gazebo simulator and to enable the connection of the simulated drone with multiple offboard computers and offboard QGroundControl stations. The *drone.launch* program uses the following arguments, defined by the user in the *gazebo.launch* file:

- The **N** (North), **E** (East), and **D** (Down) arguments consist in the initial position for the simulated drone, in meters, in the NED coordinate system adopted for the ISR Flying Arena, previously described in Section 3.2. Similarly, the **r** (roll), **p** (pitch), and **y** (yaw) parameters define the initial attitude of the simulated drone, in radians, in the same coordinate frame.
- The **vehicle\_model** argument specifies the name of the ROS model used to generate the simulated vehicle. A vehicle model is composed by a XACRO file and a BASE.XACRO file:
  - The XACRO file defines the physical properties of the simulated vehicle, such as its mass, dimensions and moments of inertia. Additionally, this file also contains the parameters that model the behavior of motors and actuators. By editing the XACRO file, the user can generate a simulated drone with dynamic properties similar to those of the real drones used in the Flying Arena. For instance, in the tests described in Chapter 6, the drag coefficient of the simulated drones was reduced from its default value in order to produce identical results to those obtained in the tests performed with the Intel Aero quadcopters, in the actual Flying Arena.
  - The BASE.XACRO file contains the plugins that model the sensors of the simulated vehicle. By editing the BASE.XACRO file, the user can add, modify, and remove plugins. When modifying plugins, the user can tune the uncertainty of the measurements produced by each sensor, by adjusting the bias and the noise density parameters.
- The **ID** argument consists in a positive integer that identifies the PX4 of each drone in a multi-vehicle situation. The **port** argument defines the network port to which user programs have to connect to, for communication with the PX4 of the simulated vehicle. Note that each vehicle must have a unique **ID** and **port**. The **gcs\_1.ip** argument defines the local IP address of an external computer, running an user program and/or an instance of the QGroundControl software, that is configured to communicate with the PX4 of the simulated vehicle. The user can add multiple instances of this argument: **gcs\_2.ip**, **gcs\_3.ip**, **gcs\_4.ip**, etc.
- Finally, the **px4\_config** argument specifies the file that stores the desired internal configurations for the PX4 SITL instance of the simulated drone. In this file, the user can set the frequency

with which the PX4 publishes each message, define the active failsafe modes, tune the internal controllers of the autopilot, and configure the extended Kalman filter of the PX4. The user can also perform these configuration procedures with the QGroundControl, once the PX4 SITL instance is running. However, it is more convenient to use the **px4.config** file because it only has to be set once and the user can employ it to start a PX4 instance of as many simulated drones as desired. As an example, in the **px4.config** file produced for this thesis, the extended Kalman filter of the PX4 is already set to use the position and yaw measurements given by the motion capture system, as described in Section 3.3.2. This means that the user can perform simulations using this **px4.config** file knowing that the PX4 of each simulated drone will start with the extended Kalman filter already configured.

After receiving these arguments from the *gazebo.launch* file, the *drone.launch* program performs the following steps, represented in Fig. 4.5:

1. First, it generates a URDF model of the simulated drone, through the XACRO and BASE.XACRO files associated with the **vehicle.model** argument selected by the user. This is necessary because Gazebo does not accept vehicles represented by XACRO models. In this thesis, it was decided to work with XACRO files instead of working directly with URDF files because the first result in shorter and more readable models, that are also more easily modified.
2. Then, it launches a ROS node that, using the URDF vehicle model, spawns the drone in the Gazebo simulation. The vehicle is spawned in the position defined by the user in the **N**, **E**, and **D** arguments and with the initial attitude selected through **r**, **p**, and **y** parameters.
3. Finally, it launches a ROS node that starts a PX4 SITL instance for the new simulated vehicle, with the internal configurations defined in the file selected in the **px4.config** argument. This PX4 SITL instance will communicate with user programs through the **port** of the simulation computer selected by the user. The traffic of this **port** is then continuously routed, using the MAVLink-Router tool, to the same **port** of the external computers with the IP address defined in the **gcs\_1\_ip** and **gcs\_2\_ip** arguments, so user programs running in those computers can also communicate with the PX4 of the simulated vehicle.

The ROS nodes mentioned in the steps 2 and 3 are created inside the ROS namespace defined in the **group.ns** argument, present in line 15 of the *gazebo.launch* file (see Fig. 4.3). Namespaces allow the coexistence of ROS topics with the same name but that relate to different drones. This is important because every PX4 expects to receive the position and attitude data from the motion capture system over the */mavros/vision\_pose/pose* and */mavros/vision/twist* topics, respectively. By running the PX4 node inside a namespace, the PX4 starts to expect receiving the motion capture data over the **namespace/mavros/vision\_pose/pose** and **namespace/mavros/vision/twist** topics, instead. This means that, by defining a unique namespace for each drone in the **group.ns** argument, it is possible to simulate multiple vehicles without their topics names conflicting.

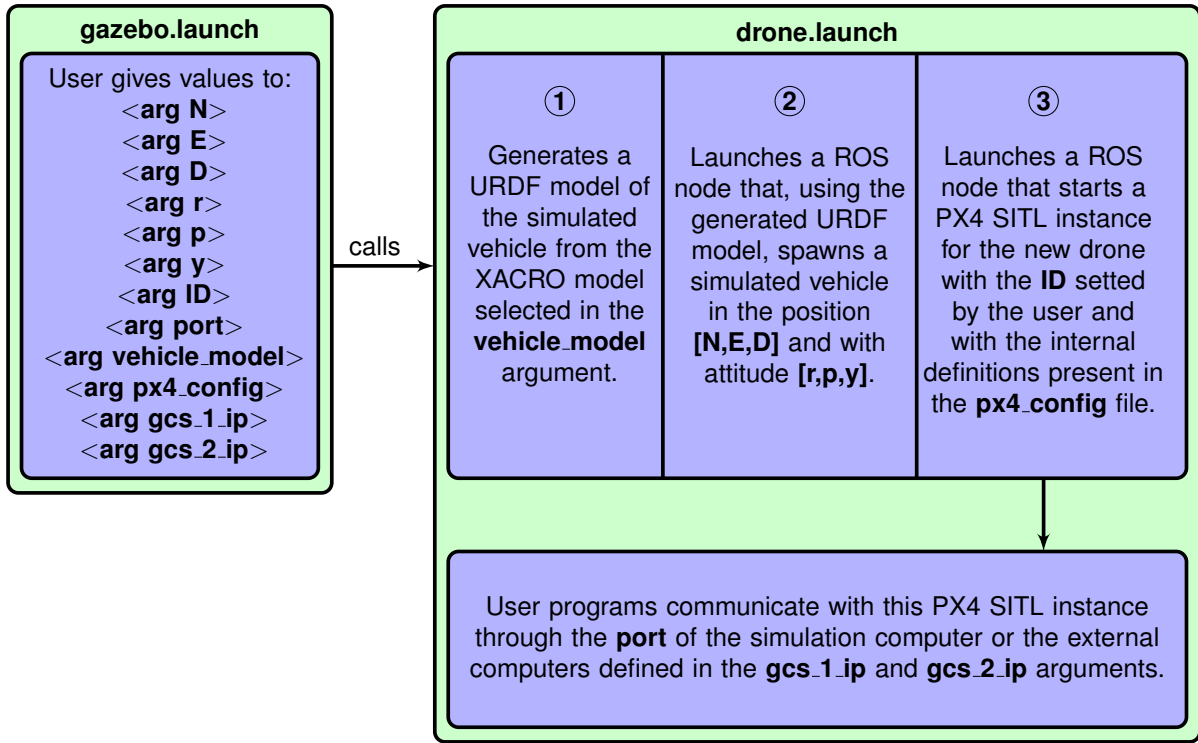


Figure 4.5: Description of the process of spawning a vehicle and launching a PX4 SITL instance.

As stated in Section 4.2, the Gazebo software communicates with the PX4 SITL instances using the MAVLink protocol. The ports used for the communication are automatically defined in the *drone.launch* program, based on the **ID** argument selected by the user for each simulated drone.

In summary, an user can easily launch simulations through the *gazebo.launch* file. This file is divided into two code blocks. The first one calls the *empty.world.launch* program to start the Gazebo software in the **world** selected by the user. The **gui** argument defines if the graphical user interface of Gazebo will be launched or not. The second code block calls the *drone.launch* program to spawn a drone in the initiated Gazebo simulation and to start a PX4 SITL instance for that simulated drone. For multi-vehicle simulations, multiple blocks of this type must be created. The vehicle spawns in the position selected through the **N**, **E**, and **D** arguments and with the initial attitude selected through the **r**, **p**, and **y** parameters. The PX4 SITL instance starts with the internal configurations defined in the file selected over the **px4\_config** argument. User programs can communicate with this PX4 instance through the **port** of the simulation computer or through the **port** of external computers defined in the **gcs\_1\_ip** and **gcs\_2\_ip** arguments. Finally, the motion of the vehicle is computed by the physics engine of Gazebo based on the physical properties of the drone and on the characteristics of sensors and actuators defined in the XACRO and BASE.XACRO files associated with the **vehicle\_model**. Before spawning the drone, the XACRO files are converted in a URDF model because the first ones are not compatible with Gazebo.

To conclude this section, the graphical user interface that results from executing the *gazebo.launch* file with the exact code shown in Fig. 4.3, is presented in Fig. 4.6. As expected, it features the Iris quadcopter [24], the standard multicopter available for Gazebo, in the origin of an empty world with the most basic light and ground models.



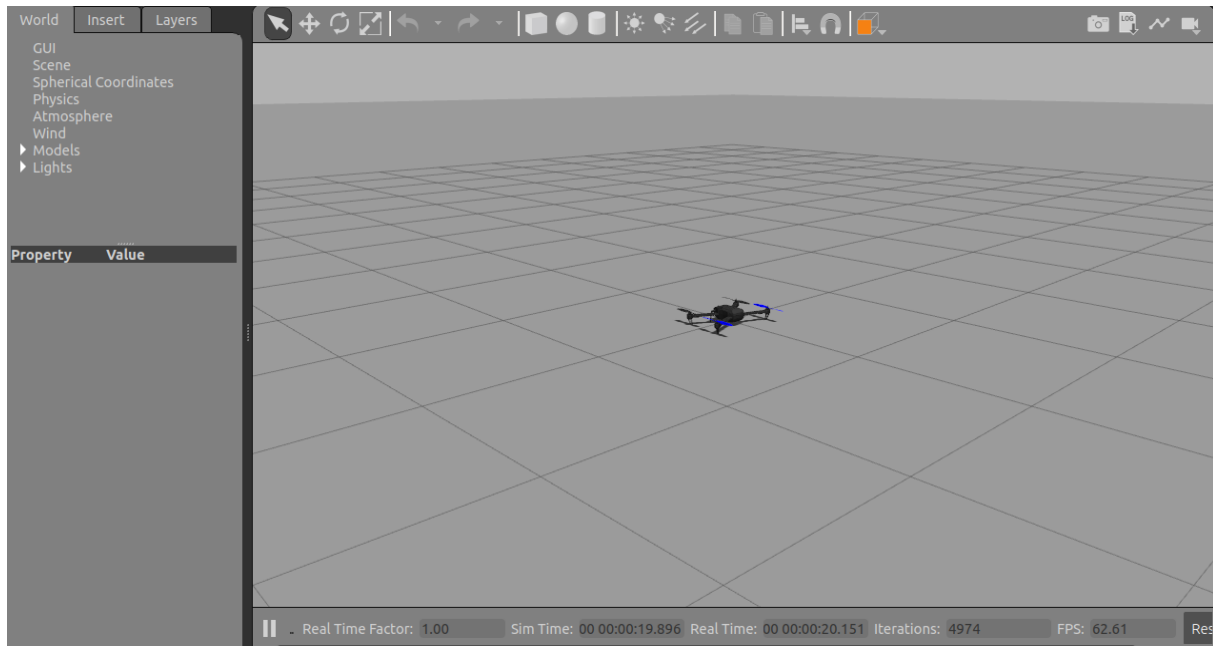


Figure 4.6: Resulting Gazebo's graphical user interface from executing the *gazebo.launch* file presented in this chapter.

## 4.4 Emulation of the motion capture system

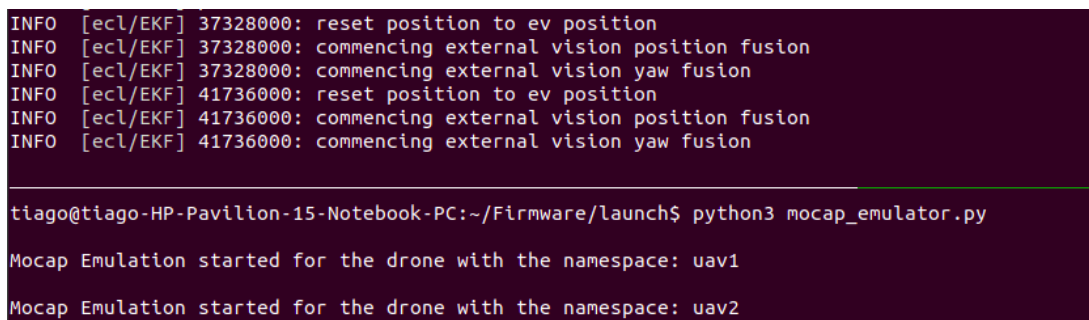
In order to have a simulation environment with the same architecture as the real environment of the ISR Flying Arena, it is necessary to implement a motion capture system capable of providing, to the user programs and to the running PX4 SITL instances, high-frequency and low-latency measurements of the position and attitude of the simulated UAVs. Since there are no official Gazebo plugins or reliable third party tools to emulate a MOCAP system, contrary to what happens for most sensors, it was created a software program using the capabilities of the ROS middleware. More specifically, the developed program was built using *rospy* [25], the Python client library for ROS.

Given the design of the Flying Arena and according to Fig. 4.1, the MOCAP emulator must, as soon as the user launches the simulation, start running a process in the background that continuously retrieves the current position and attitude data of the simulated drones from Gazebo and sends it in the appropriate format to the correct PX4 SITL instances, while also making the data available to the user programs. In light of this, the developed program, *mocap.emulator.py*, performs the following steps:

1. First, it creates a ROS subscriber that listens to the *gazebo/model\_states* topic, where the true position and attitude data of every object in the simulation is continuously being published at a frequency of 250 Hz. Every time a new message is published in this topic, a callback routine stores the new pose data of the vehicles, along with their ID, in a global variable. Consequently, this global variable will contain, for each simulated vehicle, a list with the format [ID, position, attitude] and each one of these lists will be constantly updated with the most recent pose of the vehicles. In conclusion, in this step, the true position and attitude of the drones are continuously retrieved from the Gazebo simulator and stored in a global variable.

2. Then, it reads the *gazebo.launch* file used to start the simulation and extracts the **ID** and **namespace** assigned by the user to each simulated drone. The **namespace** corresponds to the value of the `group_uav` argument. This step is indispensable because we need to know which **namespace** is associated with the **ID** of each drone, in order to be able to send the position and attitude obtained in the previous point to the PX4 topics of the correct vehicles. Recall that each PX4 expects to receive the position and attitude data from the motion capture system over, respectively, the **namespace/mavros/vision/pose/pose** and **namespace/mavros/vision/twist** topics.
3. In the last step and for each simulated vehicle, it creates a thread that runs a set of four ROS publishers. Each thread receives, as arguments, the **ID** and **namespace** of the specific drone it represents. Since the **ID** of the vehicle is known inside the thread, the true position and attitude are easily taken from the global variable that stores the lists in the format [**ID**, position, attitude]. Then, Gaussian white noise is added to the retrieved true position and attitude values, to represent the uncertainty of the measurements provided by a real motion capture system. Finally, based on the **namespace** of the vehicle, a pair of ROS publishers communicates the emulated position and attitude measurements to the PX4 SITL instance, through the topics mentioned in the last step, at a frequency of 60 Hz. The other pair of ROS publishers communicates the emulated position and attitude data to the user programs through two custom topics at a frequency of 180 Hz. These frequency values are the same ones used in the actual ISR Flying Arena. As stated in Section 3.2, user programs receive the pose updates at the work frequency of the Optitrack motion capture system to enable the development of offboard estimation algorithms. The PX4 autopilot receives the position and attitude measurements at a lower rate to avoid exhausting the bandwidth of the wireless communication channel, which could cause delays and loss of packets. Since only the MOCAP position and yaw measurements are used by the extended Kalman filter of the PX4 - to estimate the linear dynamics and correct the IMU drift, which are tasks performed at a low frequency - this downsampling procedure does not affect the performance of the EKF.

Before starting experiments in the simulation environment, the user should wait for the messages indicating that the MOCAP emulator is working and that the EKF of each PX4 is successfully using the external position and yaw measurements. Figure 4.7 shows an example of a terminal with these messages, for a simulation with two drones.



```
INFO [ec1/EKF] 37328000: reset position to ev position
INFO [ec1/EKF] 37328000: commencing external vision position fusion
INFO [ec1/EKF] 37328000: commencing external vision yaw fusion
INFO [ec1/EKF] 41736000: reset position to ev position
INFO [ec1/EKF] 41736000: commencing external vision position fusion
INFO [ec1/EKF] 41736000: commencing external vision yaw fusion

tiago@tiago-HP-Pavilion-15-Notebook-PC:~/Firmware/launch$ python3 mocap_emulator.py

Mocap Emulation started for the drone with the namespace: uav1
Mocap Emulation started for the drone with the namespace: uav2
```

Figure 4.7: Terminal with two panes, showing the messages confirming that the EKF of each PX4 (in the top pane) and the *mocap\_emulator.py* program (in the bottom pane) are successfully working.

To conclude this section, it is highlighted the fact that the emulation of the motion capture system is an automatic process, independent from the user. This meets the goal of designing an intuitive and straightforward simulation environment. The only reason that could lead the user to modify the MOCAP emulator would be to adjust the standard deviation of the Gaussian white noise added to the true position and attitude of the drones. To ensure that this is easily achievable, the standard deviation value is defined in a YAML configuration file.

## 4.5 Reducing simulation overhead

The major drawback of the Gazebo simulator and, therefore, the developed simulation environment, is that it is computationally heavy. A researcher using their personal computer to run the Gazebo software will discover that, for multi-vehicle experiments, the simulation evolves slowly, at a rate much lower than real time. This can be a problem when consistently performing long tests because it can significantly delay research. Moreover, executing simulations in real time is important when, for example, users are testing control algorithms for UAV formations, in which part of the drones of the formation are real and flying in the actual ISR Flying Arena, and the other part is being simulated in the Gazebo software. Since the control signal computed for the real drones will depend on the position or velocity of the simulated ones, and vice-versa, the tests will only be successful if the simulation is running in real time and not at a slower rate.

In order to allow users to run Gazebo at the most convenient rate, a set of solutions that make simulations computationally lighter are presented below. Some of these solutions have already been briefly mentioned throughout this chapter.

- The first recommended solution is to run the simulations without the graphical user interface of Gazebo. This makes Gazebo start faster and use less system resources. The disadvantage of this solution is that the user cannot visualize the motion of the vehicles during the simulation. To overcome this downside, lightweight visualization tools were created in Python, in Matlab, and using the graphical user interface of Gazebo but without employing its physical engine. These tools will be presented in detail in Chapter 5. The Python visualization tool is the computationally lightest one and allows the user to monitor the 3D position of the vehicles during the simulation. However, the 3D representation of the drones is too simplistic and it does not provide information about the attitude of the vehicles. The Matlab and the Gazebo-based visualization tools produce a better 3D representation of the vehicles and include attitude information. However, these can only be used once the simulation is finished, since they reproduce the data saved in the log files.
- The second recommended solution is to remove from the vehicle models all sensor plugins that are not being used during the experiments. The plugins that model the behavior of the sensors are found in the BASE.XACRO file. In the vehicle models used in this thesis for validation of control solutions, the GPS, the magnetometer, and the barometer plugins were removed, because the extended Kalman filter of the PX4 autopilot uses the data from the motion capture system as external

position and yaw sources. If these plugins were still present during the experiments, the physics engine of Gazebo would have to keep computing new measurements for these sensors, creating unnecessary overhead in the simulation. When a sensor plugin is removed, it is usually necessary to change some internal settings of the PX4. For example, when the magnetometer and barometer plugins were removed, it was necessary to change the SIS.HAS.BARO and SIS.HAS.MAG flags because, otherwise, the PX4 pre-flight tests would not pass, and it would not be possible to arm the vehicle.

- The third recommended solution consists of using a dedicated computer for the simulation or even dividing the simulated drones over multiple different computers. This solution distributes the computational power required for the simulation over several machines. It also takes advantage of the design adopted for the ISR Flying Arena, which allows user programs to communicate with any drone regardless of whether it is real or simulated on any of the computers of the local network. When conducting experiments with drones simulated across multiple computers, it is advisable to analyze the logs and reproduce them in the visualization tools developed in this thesis to make sure that no collisions have occurred.
- The final alternative is only recommended for experienced users and consists of reducing the performance of the physics engine solver of Gazebo. There is a natural trade-off between the performance of the solver and the amount of system resources used. Thus, if the user recognizes that the physics engine is over-performing for the particular experiment being carried out, it can decrease, for example, the number of iterations performed by the solver in order to increase the time rate of the simulation.

## 4.6 General overview

The simulation environment allows testing of navigation, guidance, and control solutions in a safe and non-compromising way. The development of this environment required the emulation of a motion capture system, the integration of Gazebo (a software capable of simulating the dynamics of vehicles represented by complex customizable models), and the integration of the PX4 firmware, as shown in Fig. 4.8.

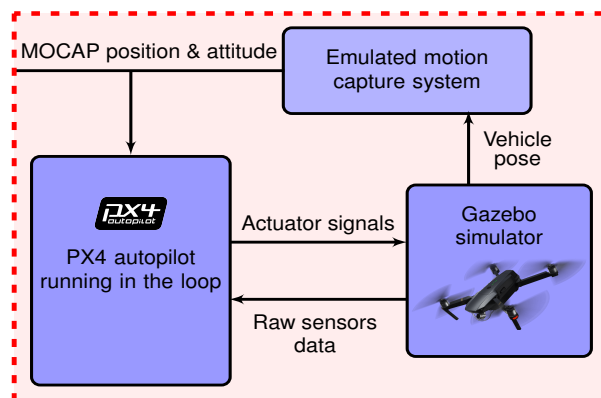


Figure 4.8: Blocks implemented and integrated in the simulation environment.

Throughout this chapter, the set of programs developed to start simulations automatically, according to the settings desired by the user for Gazebo, for the vehicles, and for the PX4 SITL instances were presented. A new and final representation of these programs is depicted in Fig. 4.9. To launch a simulation, the user only needs to perform two steps: assign values to the arguments of the *gazebo.launch* file and run the *simulator.sh* executable. The *simulator.sh* is a bash program that automatically initiates the *gazebo.launch* and the *mocap\_emulator.py* programs, which can be a complicated procedure for users that are less experienced with ROS.

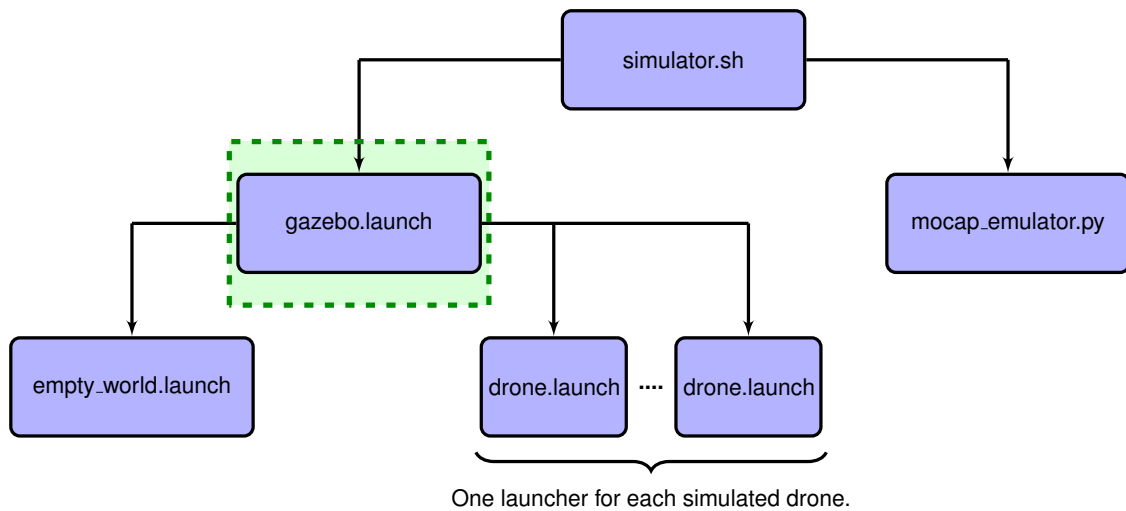


Figure 4.9: Launching process of the simulation environment.

In a brief and final summary of the remaining simulation launching process, the *empty\_world.launch* program starts Gazebo in the requested world, the *drone.launch* program spawns a vehicle in the simulation and starts a PX4 SITL instance for that vehicle, and the *mocap\_emulator.py* is the Python solution developed to emulate the motion capture system.

The next chapter presents the modules developed to allow user programs to easily perform experiments in both the real and the simulation environments.