CrazyS: a software-in-the-loop simulation platform for the Crazyflie 2.0 nano-quadcopter

Giuseppe Silano** and Luigi Iannelli

Department of Engineering
University of Sannio in Benevento
Piazza Roma, 21 - 82100 Benevento, Italy
{giuseppe.silano,luigi.iannelli}@unisannio.it

Abstract. This chapter proposes a typical use case dealing with the physical simulation of autonomous robots (specifically, quadrotors) and their interfacing through ROS (Robot Operating System). In particular, we propose CrazyS, an extension of the ROS package RotorS, aimed to modeling, developing and integrating the Crazyflie 2.0 nano-quadcopter in the physics based simulation environment Gazebo. Such simulation platform allows to understand quickly the behavior of the flight control system by comparing and evaluating different indoor and outdoor scenarios, with a details level quite close to reality. The proposed extension, running on Kinetic Kame ROS version but fully compatible with the Indigo Igloo one, expands the RotorS capabilities by considering the Crazyflie 2.0 physical model, its flight control system and the Crazyflie's on-board IMU, as well. A simple case study has been considered in order to show how the package works and how the dynamical model interacts with the control architecture of the quadcopter.
The contribution can be also considered as a reference guide for expanding the RotorS functionalities in the UAVs field, by facilitating the integration of new aircrafts. We released the software as open-source code, thus making it available for scientific and educational activities.

Keywords: software-in-the-loop simulation, virtual reality, UAV, Crazyflie 2.0, ROS, Gazebo, RotorS, Robotics System Toolbox, Continuous Integration

1 Introduction

Unmanned Aerial Vehicles (UAVs), although originally designed and developed for defense and military purposes (e.g., aerial attacks or military air covering), during recent years gained an increasing interest and attention related to the civilian use. Nowadays, UAVs are employed for several tasks and services like surveying and mapping [1], for rescue operations in disasters [2,3], for spatial information acquisition, buildings inspection [4,5], data collection from inaccessible areas, geophysics exploration [6,7], traffic monitoring [8], animal protection [9], agricultural crops and monitoring [10], manipulation and transportation or navigation purposes [11,12].

** Corresponding author.

Fig. 1. The Crazyflie 2.0 nano-quadcopter. Retrieved from [26]. Copyright 2018 by Bitcraze AB.

Many existing algorithms for the autonomous control [13,14] and navigation [15,16] are provided in the literature, but it is particularly difficult to make the UAVs able to work autonomously in constrained and unknown environments or also indoor. Thus, it follows the need for tools that allow to understand what it happens when some new applications are going to be developed in unknown or critical situations. Simulation is one of such helpful tools, widely used in robotics [17,18,19], and whose main benefits are costs and time savings, enabling not only to create various scenarios, but also to study and to carry out complex missions that might be time consuming and risky in the real world. Finally, bugs and mistakes in simulation cost nothing: it is possible to crash a vehicle virtually several times and thereby getting a better understanding of implemented methods under various conditions. To this aim, simulation environments are very important for fast prototyping and educational purposes. Indeed, they are able to manage the complexity and heterogeneity of the hardware and the applications, to promote the integration of new technologies, to simplify the software design, to hide the complexity of low-level communication [20].

Different solutions, typically based on external robotic simulators such as Gazebo [21], V-REP [22], Webots [23], AirSim [24], MORSE [25], are available to this purpose. They employ recent advances in computation and graphics (e.g., the AirSim photorealistic environment [15]) in order to simulate physical phenomena (gravity, magnetism, atmospheric conditions) and perception (e.g., providing sensor models) in such a way that the environment realistically reflects the actual world. Definitely, it comes out that complete software platforms able to test control algorithms for UAVs in a simulated 3D environment are becoming more and more important.

In this tutorial chapter, the Micro Aerial Vehicles (MAVs) simulation framework RotorS[1] [28] has been employed as a base for proposing CrazyS, a software package for modeling, developing and integrating the dynamics and the control architecture of the nano-quadcopter Crazyflie 2.0 [26] (Fig. 1) in the Gazebo simulator.

Our work may be considered the answer to impelling needs of many researchers working on Crazyflie that ask for a simulator specific for such nano-quadrotor platform, as clearly stated in [29, Sect. 9.5]. At the same time, the chapter can be seen as a

[1] Together with Hector Quadrotor [27], RotorS is among the most used platforms for simulating a multi-rotor in Gazebo through ROS middleware.

reference guide for expanding functionalities of RotorS and facilitating the integration of new vehicles equipped with both on-board sensors and control systems. In addition, the contribution aims to highlight how the development of control strategies may be facilitated allowing performance evaluation in a scenario quite close to reality, thanks to software-in-the-loop (SITL) simulation methodologies (see [30] for a general overview and [31,32] for mechatronics and UAV applications).

The chosen aircraft, the Crazyflie 2.0, is available on the market at a price of less than $200 and it is ideal for many research areas (e.g., large swarm [33], tethered flight [34], path planning [35], mixed reality [36], education [37], disturbances rejection [38], etc.). The source code and the hardware are open, making it possible to go through any part of the system for complete control and full flexibility. New hardware and sensors can be linked through the versatile expansion ports, enabling the addition of the latest sensors. The small size and light weight reduce the need for safety equipment and increase the productivity. For all such reasons, it appears valuable to have a detailed flexible simulator of the Crazyflie dynamic behavior, with the possibility of validating in an easy way the effects of modifying the control architecture for achieving complex missions. We published the software as open-source [39] and at the same time we opened a pull request [40] on RotorS repository with the aim to share our result with other researchers who already use such tools and would like to use the platform. However this chapter may help also those researchers, as control engineers, that are familiar with UAV applications and software-in-the-loop simulation concepts but have no experience with Gazebo and ROS.

The chapter is organized as follows. First, we briefly describe the quadcopter dynamical model and its flight control system, i.e., the architecture of the Crazyflie's low level control system. Then we model the on-board sensors, i.e., the Inertial Measurement Unit (IMU MPU$-$9250 [41]) in order to develop a simulation platform as close as possible to the real system. The entire procedure followed to bring datasheet values to the simulation environment will be explained in detail by describing the mathematical models and the related specifications. A further part will deal with the complementary filter, i.e., the default Crazyflie state estimator, that has been implemented in CrazyS according to the 2018.01.1 firmware release of the aircraft. After that we demonstrate how to download and to use the CrazyS ROS package by providing step by step instructions on how to proceed, by taking into account all software pre-requisites and dependencies (see Sec. 3.1).

At this point we give a complete overview of the simulation environment. Starting from a RotorS example, it is here described how CrazyS is structured for taking, as command signals, the yaw rate $\dot{\psi}_c$, the pitch angle θ_c, the roll angle ϕ_c and the thrust (actually, it denotes the desired rotors speed Ω_c). Such commands correspond to the inputs (references) of the on-board low level control in the Crazyflie 2.0 architecture (see Sec. 3.2).

We show how to use the MathWorks® Robotics System Toolbox (RST) to build-up a simulation platform in which control strategies are implemented through Simulink schemes, i.e., the usual tools that control engineers are familiar with. The RST allows to run Simulink schemes and to interface them to Gazebo that is in charge of simulating the detailed aircraft physical model (Sec. 3.4.1). Then, control strategies will be

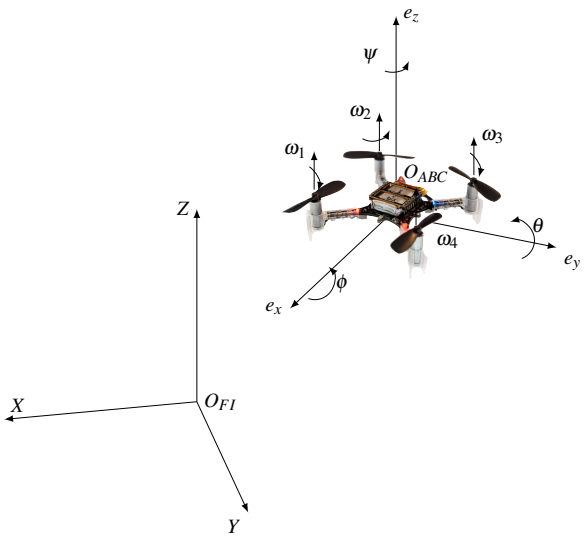

Fig. 2. Crazyflie in the body-frame (O_{ABC}) and the fixed-frame (O_{FI}) reference system. Forces, spin directions and the propellers angular velocity ω_i of each rotor are depicted.

implemented in C++ code thus achieving a complete software-in-the-loop simulation platform based on ROS and Gazebo (see Sec. 3.4.2). Finally, it will be described how to configure a Continuous Integration (CI) infrastructure, by proposing a solution to link the open-source platform TravisCI with the CrazyS repository. Advantages related to the use of CI system when developing ROS packages are described in Sec. 3.5. Conclusions close the chapter.

2 Crazyflie 2.0 nano-quadcopter

In this section, we describe the quadcopter physical model and how the simulator works. Moreover the flight control system architecture is presented together with the on-board sensors model. Contents of this section are inspired by our previous work [42] but are here reviewed and explored in detail.

2.1 Dynamical model

The design of a suitable position controller for the quadcopter exploits an accurate dynamical model. As the usual approach in the literature, we introduce two orthonormal frames: the fixed-frame O_{FI} (where FI stands for Fixed Inertial), also called inertial (or reference) frame, and the body-frame O_{ABC} (where ABC stands for Aircraft Body Center) that is fixed in the aircraft center of gravity and oriented along the aircraft main directions (so defining its attitude), see Fig 2.

According to [43], the forces (eqs. (1) and (2)) and the momentum (eqs. (3) and (4)) equations can be derived. Such model consists of twelve differential equations for the system dynamics and four algebraic equations describing the relations between inputs (forces and momenta) to the system and rotor velocities (eqs. (5) and (6)).

$$\begin{bmatrix} \dot{x}_d \\ \dot{y}_d \\ \dot{z}_d \end{bmatrix} = \mathbf{R}^T(\phi_d, \theta_d, \psi_d) \begin{bmatrix} u_d \\ v_d \\ w_d \end{bmatrix}, \tag{1}$$

$$\begin{bmatrix} \dot{u}_d \\ \dot{v}_d \\ \dot{w}_d \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ F_z/m \end{bmatrix} - \mathbf{R}(\phi_d, \theta_d, \psi_d) \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} - \begin{bmatrix} p_d \\ q_d \\ r_d \end{bmatrix} \times \begin{bmatrix} u_d \\ v_d \\ w_d \end{bmatrix}, \tag{2}$$

$$\begin{bmatrix} \dot{p}_d \\ \dot{q}_d \\ \dot{r}_d \end{bmatrix} = \mathbf{J}^{-1} \left(\begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} - \begin{bmatrix} p_d \\ q_d \\ r_d \end{bmatrix} \times \mathbf{J} \begin{bmatrix} p_d \\ q_d \\ r_d \end{bmatrix} \right), \tag{3}$$

$$\begin{bmatrix} \dot{\phi}_d \\ \dot{\theta}_d \\ \dot{\psi}_d \end{bmatrix} = \begin{bmatrix} 1 & s_{\phi_d} t_{\theta_d} & c_{\phi_d} t_{\theta_d} \\ 0 & c_{\phi_d} & -s_{\phi_d} \\ 0 & s_{\phi_d}/c_{\theta_d} & c_{\phi_d}/c_{\theta_d} \end{bmatrix} \begin{bmatrix} p_d \\ q_d \\ r_d \end{bmatrix}, \quad \theta_d \neq \frac{\pi}{2}. \tag{4}$$

The body-frame orientation is described through the Euler angles ϕ_d, θ_d and ψ_d, defined according to the *ZYX* convention [44] and it can be computed by considering that the rotation matrix $\mathbf{R}(\phi_d, \theta_d, \psi_d)$ allows to convert a vector expressed in the fixed-frame to a vector expressed in the O_{ABC} body-frame. Thus, equation (1) relates the linear velocities of the aircraft in the O_{ABC} frame, i.e., $\begin{pmatrix} u_d & v_d & w_d \end{pmatrix}^\top$, to the linear velocities of the aircraft in the fixed frame, denoted by $\begin{pmatrix} \dot{x}_d & \dot{y}_d & \dot{z}_d \end{pmatrix}^\top$, through the inverse matrix $\mathbf{R}(\phi_d, \theta_d, \psi_d)^{-1} = \mathbf{R}(\phi_d, \theta_d, \psi_d)^\top$. Whereas, by considering the time derivative of $\mathbf{R}(\phi_d, \theta_d, \psi_d)$, the angular velocities of the aircraft in the O_{FI} frame are related to the corresponding velocities expressed in the body frame through eq. (4), where $c_\bullet$, $s_\bullet$ and $t_\bullet$ denote $\cos(\bullet)$, $\sin(\bullet)$ and $\tan(\bullet)$ functions, respectively.

Conversely, the remaining six equations (eqs. (2) and (3)) describe the UAV linear and angular accelerations in the O_{ABC} frame. The diagonal matrix $\mathbf{J}$ has the inertia of the body about the x, y and z-axis, respectively, while m is the total mass of the quadcopter and g the gravitational constant.

The system inputs are reported in eqs. (5) and (6), where ω_1, ω_2, ω_3 and ω_4 represent the rotors angular velocities expressed in $\text{rad}\,\text{s}^{-1}$:

$$F_z = C_T \left(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2 \right), \tag{5}$$

$$M = \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} C_T\, d \left(-\omega_1^2 - \omega_2^2 + \omega_3^2 + \omega_4^2 \right) \\ C_T\, d \left(-\omega_1^2 + \omega_2^2 + \omega_3^2 - \omega_4^2 \right) \\ \sqrt{2} C_M \left(-\omega_1^2 + \omega_2^2 - \omega_3^2 + \omega_4^2 \right) \end{bmatrix}. \tag{6}$$

Finally, d is the distance of the propellers from the center of gravity while C_T and C_M are the rotor thrust and rotor moment constants, respectively [28]. By increasing or decreasing uniformly the propellers speed it causes an altitude change, while by varying

Entries	Sym.	Value	Unit
Motor_constant	C_T	$1.28192 \cdot 10^{-8}$	$\mathrm{kg\,m\,rad^{-2}}$
Moment_constant	C_M	$5.964552 \cdot 10^{-3}$	$\mathrm{kg\,m^2\,rad^{-2}}$
Rotor_drag_coefficient	C_D	$8.06428 \cdot 10^{-5}$	$\mathrm{kg\,rad^{-1}}$
Rolling_moment_coefficient	C_R	$1 \cdot 10^{-6}$	$\mathrm{kg\,m\,rad^{-1}}$

Table 1. Crazyflie 2.0 parameter values according to the MAV model employed in RotorS.

the speed ω_1 and ω_4 (or the pair ω_2 and ω_3 with the opposite effect) it causes the aircraft to tilt about the y-axis, i.e., the pitch angle θ_d. Similarly, by varying the speeds ω_1 and ω_2 (or the pair ω_3 and ω_4) it causes the aircraft to tilt about the x-axis, i.e., the roll angle ϕ_d. Finally, the vector sum of the reaction moment produced by the speed of the pair ω_1 and ω_3 and the reaction moment produced by the speed of ω_2 and ω_4 will cause the quadcopter to spin about its z-axis, i.e., modifying the yaw angle ψ_d. Further details are given in [43,45] while the parameter values of the Crazyflie have been taken from the repository [46] by the same research group of [29].

According to the MAV model employed in RotorS [28], Table 1 summarizes the drone parameter values reported in the `crazyflie2.xacro` file and used to describe the aircraft dynamics with the corresponding entries.

2.2 Flight control system

In order to illustrate how to apply SITL testing methodologies to UAV design, we consider a common architecture of a flight control system for controlling the position of a quadrotor, so as illustrated in [43]. We have a "reference generator" that takes the position to reach (x_r, y_r and z_r) and the desired yaw angle ψ_r and generates the command signals (θ_c, ϕ_c, Ω_c and $\dot{\psi}_c$) that are inputs for the on-board control architecture of the Crazyflie. Figure 3 describes the overall system while Figs. 4 and 5 describe the reference generator and the on-board control architecture, respectively. In the event that the desired position is not available (it should be published on the ROS topic *command/trajectory*), the drone maintains its previous pose until the next waypoint.

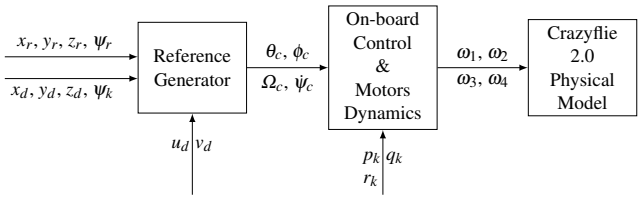

Fig. 3. The control scheme. Subscript c refers to commands, r to references, d indicates the actual drone variables and k indicates the sensors and data fusion outputs when the Crazyflie's state estimator is in the loop (when it is not, they are replaced by the values coming from the odometry).

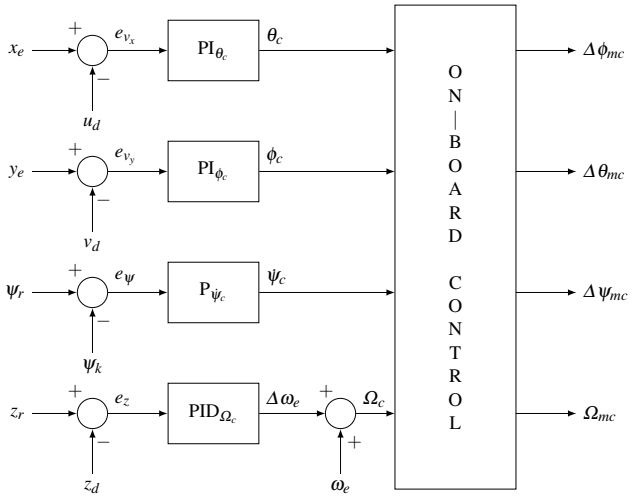

Fig. 4. The reference generator scheme. The obtained heuristic PID gains are: $K_{P_{\dot{\psi}_c}} = 0.0914$, $K_{P_{\Omega_c}} = 70$, $K_{I_{\Omega_c}} = 3.15$, $K_{D_{\Omega_c}} = 373$, $K_{P_{\theta_c}} = 3.59$, $K_{I_{\theta_c}} = 5.73$, $K_{P_{\phi_c}} = -3.59$ and $K_{I_{\phi_c}} = -5.73$.

2.2.1 Reference generator

The reference generator uses drone measurements, in particular the drone position (x_d, y_d and z_d) and its body-frame velocity (u_d, v_d), and the estimated orientation along z-axis (i.e., the yaw ψ_k) to compute the command signals (θ_c, ϕ_c, Ω_c and $\dot{\psi}_c$). In real indoor applications the drone position and velocity come from a motion capture system (MoCap), such as Vicon [47], Optitrack [48] or Qualisys [49]. Here, for simplicity we modeled such data coming from an ideal (no bias and no noise) virtual odometry sensor in the simulation environment. However the platform allows to model also typical measurement data coming from such systems, without much difficulty.

As described by the overall scheme in Fig. 4, the reference generator computes the desired attitude (θ_c and ϕ_c), the yaw rate ($\dot{\psi}_c$) and thrust (Ω_c) commands for the Crazyflie, later used as references for the on-board control system. Such command signals are limited as summarized in Table 2.

	Sym.	Unit	Output limit
Roll command	ϕ_c	rad	$[-\pi/6,\ \pi/6]$
Pitch command	θ_c	rad	$[-\pi/6,\ \pi/6]$
Yaw rate command	$\dot{\psi}_c$	$\mathrm{rad\,s^{-1}}$	$[-1.11\pi,\ 1.11\pi]$
Thrust command	Ω_c	UINT16	[5156, 8163]

Table 2. Physical constraints of the Crazyflie 2.0 nano-quadcopter.

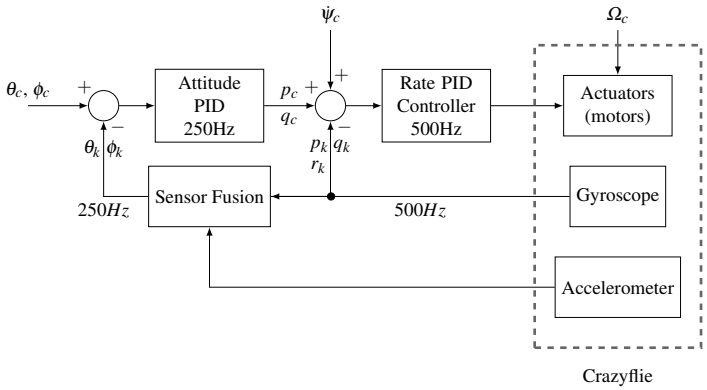

Fig. 5. On-board control architecture of the Crazyflie 2.0, release 2018.01.1. The obtained heuristic gains are: $K_{P_{p_c}} = 0.0611$, $K_{I_{p_c}} = 0.0349$, $K_{P_{q_c}} = 0.0611$, $K_{I_{q_c}} = 0.0349$, $K_{P_{\Delta\phi_{mc}}} = 1000$, $K_{P_{\Delta\theta_{mc}}} = 1000$, $K_{P_{\Delta\psi_{mc}}} = 1000$ and $K_{I_{\Delta\psi_{mc}}} = 95.683$.

The thrust is expressed directly as a pulse with modulation (PWM) signal (a 16 bit unsigned integer), and obtained by the sum of two terms: the feedforward term $\omega_e = 6874$ corresponding to the hovering condition (perfect horizontal attitude and propellers velocities that counteracts the gravity force) and the feedback term $\Delta\omega_e$. The reference signals x_e and y_e (see Fig. 4) are computed as:

$$x_e = (x_r - x_d)\cos(\psi_k) + (y_r - y_d)\sin(\psi_k) \tag{7a}$$
$$y_e = (y_r - y_d)\cos(\psi_k) - (x_r - x_d)\sin(\psi_k). \tag{7b}$$

Such signals are employed as setpoints for the velocities body-frame u_d and v_d, respectively. As explained in [43], the logic behind such choice consists in the fact that bigger is the error faster the quadcopter should move in order to arrive at the desired point. Instead, when the error is small, the drone is close to the desired point and the setpoint for the velocity should be also small.

2.2.2 On-board control system

The on-board control is decomposed into two parts: the attitude and the rate controller, both illustrated in Fig. 5. They work together in a cascaded control structure. As commonly implemented in such structures, the inner loop needs to regulate at a rate faster than the outer. In this case, the attitude controller runs at 250 Hz while the rate controller runs at 500 Hz.

We considered the on-board control architecture existing in the firmware release[2], the 2018.01.1. The same software architecture has been followed also to integrate the complementary filter (see Sec. 2.3.1), the default Crazyflie state estimator. Starting from

[2] Of course that has been possible thanks to the fact that Crazyflie firmware is open-source.

the accelerometer and gyroscope data, the filter allows to estimate the attitude (ϕ_k, θ_k and ψ_k) and the angular velocities (p_k, q_k and r_k) used by the on-board control loop also managing the sensors' bias and noise terms. All controller parameter values are provided in the file `controller_crazyflie2.yaml` and they can be easily modified as explained in Sec. 3.4.2.

Finally, we modeled the actuators dynamics (see Fig. 5) by considering the relationship between the PWM signals sent to the motors and the actual propellers speed, so as explained in [50],

$$\omega_i = \frac{\pi}{30}\left(\alpha \cdot PWM_i + q\right), \tag{8}$$

where $\alpha = 0.2685$ and $q = 4070.3$. The PWM signals are computed according to the rate controller outputs $\Delta\phi_{mc}$, $\Delta\theta_{mc}$ and $\Delta\psi_{mc}$, i.e., the total variations from the equilibrium, and the thrust command Ω_{mc} (that, in particular, corresponds to Ω_c):

$$\begin{cases} PWM_1 = \Omega_{mc} - {}^{\Delta\theta_{mc}}/_2 - {}^{\Delta\phi_{mc}}/_2 - \Delta\psi_{mc} \\ PWM_2 = \Omega_{mc} + {}^{\Delta\theta_{mc}}/_2 - {}^{\Delta\phi_{mc}}/_2 + \Delta\psi_{mc} \\ PWM_3 = \Omega_{mc} + {}^{\Delta\theta_{mc}}/_2 + {}^{\Delta\phi_{mc}}/_2 - \Delta\psi_{mc} \\ PWM_4 = \Omega_{mc} - {}^{\Delta\theta_{mc}}/_2 + {}^{\Delta\phi_{mc}}/_2 + \Delta\psi_{mc}. \end{cases} \tag{9}$$

Usually, a DC motor can be characterized as a first order transfer function, but in our application a well approximated behavior assumes that the transient is fast enough and that it will not cause much delay in the system.

2.3 State estimation

One of the key elements enabling stable and robust UAV flights is an accurate knowledge of the state of the aircraft. In CrazyS, as well as in RotorS, such information can be directly provided by an (ideal) odometry sensor. This means that position, orientation, linear and angular velocities of the Crazyflie come from Gazebo plugins[3].

As mentioned before, the odometry sensor has been used only to know the position and the linear velocity of the vehicle. Conversely, the drone orientation and angular velocity have been obtained by using the default Crazyflie state estimator: the complementary filter. Nevertheless, with the aim of highlighting the flexibility of the simulation platform (it is quite easy to move from a simulation scenario to another), we compared the outputs of the complementary filter with the ideal case (see Sec. 3.4.2) where position, orientation, angular and linear velocities come from the odometry sensor (without noise and bias). Therefore, in this section we will give an overview of how the filter works (further details can be found in [51]) and how to model and integrate the IMU measurements in Gazebo starting from the sensor datasheet values.

2.3.1 Complementary filter

The Kalman filter is a well known and established solution for combining sensors data into navigation-ready data, although its nonlinear version is difficult to apply with low-cost and high-noise sensors [52]. Moreover also Extended Kalman filter (EKF) techniques might give unsatisfactory results [53] and accurate calibration for gyroscope

[3] Such data can be easily processed by adding noise and bias terms when required, as explained in [28].

offsets, noise and other constants, might be needed to properly implement Kalman filtering. On the other hand, complementary filters, that are not model based techniques, are not well-suited for high-risk applications like space or unmanned missions. However, compared to Kalman filtering, the complementary filter is less computationally intensive, requires less calibration and more readily performs on small, low-power processing hardware. In practice that technique is ideal for small, low-cost aircrafts as the Crazyflie 2.0. Thus, a complementary filter has been implemented in CrazyS. Nevertheless, a Kalman filter solution can be investigated and implemented in order to evaluate the trade off between precision and computational burden. The modular structure of CrazyS allows to replace the complementary filter with another estimator (e.g., Luenberger observer, EKF, particle filter, etc.), in easy way.

The key idea behind the complementary filter is to use the information coming from the gyroscope (that is precise and not susceptible to external forces), and data from the accelerometers (they have no drift). In particular, the on-board complementary filter of the Crazyflie follows the implementation of Madgwick's IMU and AHRS (Attitude and Heading Reference Systems) algorithms [51].

Among its advantages, the filter allows a significant reduction in the computational load, guarantees good performances and eliminates the need for a predefinition of the magnetic field direction.

2.3.2 IMU sensor model

As explained in [54], we modeled the on-board Crazyflie's IMU (MPU−9250, [41]) by integrating it in the `component_snippets.xacro`. The Xacro file [55], that is a particular eXtensibile Markup Language (XML) file used to generate a more readable and often shorter XML code, contains all the macros employed to model the sensors behavior in the Gazebo simulator. The XML tag structure allows to set properties that are related to the physical features of the IMU, like the measurement delay, the divisor (it allows to set up the sensor frequency response, see [56]), the mass or other physical parameters. The macros in such file can be used by any aircraft in the simulation environment, each one described by using an own xacro file (`crazyflie2_base.xacro`, in our case). Such file contains the full list of the on-board integrated components (IMU, barometer, camera, odometry sensor, etc.). More details on how they are related to the *launch* files[4] are reported in Sec. 3.3.1.

```
<xacro:macro name="crazyflie2_imu" params="namespace
   parent_link">
<xacro:imu_plugin_macro
namespace="${namespace}"
imu_suffix=""
parent_link="${parent_link}"
imu_topic="imu"
measurement_delay="0"
```

[4] Launch files are very common in ROS to both users and developers. They provide a convenient way to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters.

```
measurement_divisor="1"
mass_imu_sensor="0.00001"
gyroscope_noise_density="0.000175"
gyroscope_random_walk="0.0105"
gyroscope_bias_correlation_time="1000.0"
gyroscope_turn_on_bias_sigma= "0.09"
accelerometer_noise_density="0.003"
accelerometer_random_walk="0.18"
accelerometer_bias_correlation_time="300.0"
accelerometer_turn_on_bias_sigma="0.588">
<inertia ixx="0.00001" ixy="0.0" ixz="0.0" iyy="0.00001"
   iyz="0.0" izz="0.00001" />
<origin xyz="0 0 0" rpy="0 0 0" />
</xacro:imu_plugin_macro>
</xacro:macro>
```

Listing 1.1. Crazyflie IMU tag structure.

In RotorS (and, thus, in CrazyS), measurements are modeled by two types of sensor errors affecting both the angular rate measurement $\tilde{\omega}$ and the linear acceleration $\tilde{a}$, expressed as

$$\tilde{\omega}(t) = \omega(t) + b_\omega(t) + n_\omega(t) \tag{10a}$$
$$\tilde{a}(t) = a(t) + b_a(t) + n_a(t), \tag{10b}$$

where $n_\bullet(t)$ is an additive noise term that fluctuates very rapidly (the white noise) and $b_\bullet(t)$ is a slowly varying sensor bias. All gyroscope and accelerometer axis measurements are modeled, independently. Table 3 summarizes all the model parameters that are reported as entries in the Xacro file (see Listing 1.1).

Since the aim of this chapter is to illustrate a SITL simulation platform and its use rather than simulating a specific hardware component, it is not important for our work to identify accurately the model of all hardware components and sensors. Instead, we are

	Sym.	Unit	Value
Gyroscope			
White noise density	n_ω	$\mathrm{rad/s/\sqrt{Hz}}$	0.000175
Random walk	b_ω	$\mathrm{rad/s^2/\sqrt{Hz}}$	0.0105
Bias correlation time	b_{t_ω}	s	1000
Turn on bias sigma	b_{ω_0}	$\mathrm{rad\,s^{-1}}$	0.09
Accelerometers			
White noise density	n_a	$\mathrm{m/s^2/\sqrt{Hz}}$	0.003
Random walk	b_a	$\mathrm{m/s^3/\sqrt{Hz}}$	0.18
Bias correlation time	b_{t_a}	s	300
Turn on bias sigma	b_{a_0}	$\mathrm{m\,s^{-2}}$	0.588

Table 3. Summary of the IMU model parameters.

interested in getting realistic values for the parameters of the simulated models. To this aim datasheets are enough for getting values of interest. In particular, the accelerometer and gyroscope noise densities (the *white noise density*) have been easily obtained from the MPU−9250 datasheet, requiring just a scaling due to the fact that Gazebo uses SI units measurements [57]. On the other hand, the bias part of the model (the *random walk*) is rarely specified into datasheets. However it can be characterized [54,58,59] as

$$b_\bullet = n_\bullet \sqrt{T}, \tag{11}$$

where the parameter $n_\bullet$ is the noise density (aka spectral noise density), and T is the time period over which the idealized white noise process is integrated (one hour, in our case). Finally, the *turn on bias* and the *bias correlation time* refer to the bias value, originated when the inertial sensor turns on, and its time constant, respectively [60].

As said previously, the above mentioned procedure is independently of a specific sensor. Thus, it can be employed to model any sensor in the virtual scenario expanding the functionalities of the simulation framework.

3 Tutorials

This section explains how to use the CrazyS simulation framework with its main components. The setting-up in Ubuntu, both Trusty (14.04) and Xenial (16.04) distros, is shown in Sec. 3.1.2. Although the platform is fully compatible with Indigo Igloo version of ROS and Ubuntu 14.04, such configuration is not recommended since the ROS support will close in April 2019.

Section 3.2 demonstrates how to put the nano-quadcopter into hovering mode, with and without the aircraft on-board sensors, and how to attach such sensors to it (see Sec. 3.3.1). Section 3.3 describes the simulator through the hovering example, while Sec. 3.4 illustrates how to employ the Robotics System Toolbox for testing the controller strategy before implementing the corresponding ROS code. The aim is to show how the controlled system can change its behavior with respect to the Matlab/Simulink version when tested in an environment closer to the reality like Gazebo, and how to verify it before writing many lines of `C++` or Python code.

3.1 Simulator setup

Before installing and using CrazyS, it is necessary to install and configure ROS over a suitable Linux distribution. Although it could be possible to install ROS also on other platforms (like MacOS), Ubuntu is the recommended operating system (OS) and its package manager should be used to install all necessary dependencies. All suggested operations are discussed on the official wiki-pages: see `http://wiki.ros.org/indigo/Installation/Ubuntu` or `http://wiki.ros.org/kinetic/Installation/Ubuntu` for Indigo Igloo and Kinetic Kame distros, respectively.

3.1.1 Ubuntu with ROS

In this subsection, for the sake of completeness and practicality, we report the commands for installing ROS Indigo Igloo (see Listing 1.2) and Kinetic Kame (see Listing 1.3). Before running such commands it is suggested to give a look at the OS compatibility in the official ROS wiki-pages mentioned above.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/
   ubuntu $(lsb_release -sc) main" > /etc/apt/sources.
   list.d/ros-latest.list'
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-
   keyservers.net:80 --recv-key 421
   C365BD9FF1F717815A3895523BAEEB01FA116
$ sudo apt-get update
$ sudo apt-get install ros-indigo-desktop-full
$ sudo rosdep init
$ rosdep update
$ echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
$ sudo apt-get install python-rosinstall
```

Listing 1.2. Installation instructions - Ubuntu 14.04 with ROS Indigo Igloo.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/
   ubuntu $(lsb_release -sc) main" > /etc/apt/sources.
   list.d/ros-latest.list'
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-
   keyservers.net:80 --recv-key 421
   C365BD9FF1F717815A3895523BAEEB01FA116
$ sudo apt-get update
$ sudo apt-get install ros-kinetic-desktop-full
$ sudo rosdep init
$ rosdep update
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
$ sudo apt-get install python-rosinstall python-
   rosinstall-generator python-wstool build-essential
```

Listing 1.3. Installation instructions - Ubuntu 16.04 with ROS Kinetic Kame.

3.1.2 Installing CrazyS from source

After having configured both the OS and ROS, the platform can be installed from source. Although CrazyS is completely independent of the chosen OS or ROS distribution, the package dependencies have to be satisfied according to the chosen OS and ROS distro. Therefore, in Listing 1.4 we report the installing procedure for the Kinetic Kame version of ROS. Whereas, in Listing 1.5 the package dependencies for the Indigo Igloo distro are reported (the procedure is exactly the same as for Kinetic Kame).

```
$ sudo apt-get ros-kinetic-joy ros-kinetic-octomap-ros
   ros-kinetic-mavlink python-catkin-tools protobuf-
   compiler libgoogle-glog-dev ros-kinetic-control-
   toolbox
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ catkin init
$ git clone https://github.com/gsilano/CrazyS.git
$ git clone https://github.com/gsilano/mav_comm.git
$ cd ~/catkin_ws/src/mav_comm & git checkout crazys
$ rosdep update
$ cd ~/catkin_ws
$ rosdep install --from-paths src -i
$ catkin build
$ echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Listing 1.4. Installation instructions from source with ROS Kinetic Kame.

```
$ sudo apt-get ros-indigo-octomap-ros python-wstool
   python-catkin-tools protobuf-compiler
$ sudo apt-get ros-indigo-joy libgoogle-glog-dev
```

Listing 1.5. Package dependencies for installing CrazyS from source with ROS Indigo Igloo.

Such procedure allows to create a workspace folder (`catkin_ws`) that will contain (in the `src` directory) the code that simulates the Crazyflie dynamics and behavior (determined by sensors model and control algorithms). Further details about the workspace and its meaning can be found in [61], while in [62] there are reported more details regarding messages and services used during the simulation.

3.2 Hovering example

Launching the simulation is quite simple, so as customizing it: it is enough to run in a terminal the command

```
$ roslaunch rotors_gazebo crazyflie2_hovering_example.
   launch
```

By default the state estimator is disabled since on-board Crazyflie's sensors are replaced by the odometry one. For running the simulation by taking into account the Crazyflie's IMU and the complementary filter, it is enough to give a command that turns on the flag `enable_state_estimator`:

```
$ roslaunch rotors_gazebo crazyflie2_hovering_example.
   launch enable_state_estimator:=true
```

The visual outcome will see the nano-quadcopter taking off after 5s (time after which the *hovering_example* node publishes the trajectory to follow) and flying one

meter above the ground, at the same time keeping near to zero the position components along x and y-axis.

For understanding how the controllers work (the reference generator and the Crazyflie's on-board controller, see Sec. 2.2), two plots of the drone position and orientation have been added in the *launch* file. At each time step, data coming from the Gazebo plugins are reported on the plots avoiding to go through the *rosbag* files[5]. The flexible and fully controllable structure of the *launch* file allows to plot any information coming from the simulator. Among such data we can consider the drone state (u_d, v_d, w_d, etc.), the command signals (θ_c, ϕ_c, Ω_c and $\dot{\psi}_c$) or the trajectory references (x_r, y_r, z_r and ψ_r).

3.3 Simulator description

This section is focused on describing how RotorS (and thus CrazyS), works together with ROS and Gazebo, by considering as illustrative application the *hovering example*. An overview of the main components is reported in Fig. 6 while further details can be found in [28].

To facilitate the development of different control strategies, we recommend to provide a simple interface, like the modular architecture developed for CrazyS and appropriately adapted from RotorS. In the illustrative example we developed a linear position control strategy (see Sec. 2.2), but other control laws can be considered, even nonlinear [63,64,65]. Indeed, the simulator has to be meant as a starting point to implement more advanced control strategies for the Crazyflie and, more generally, for any quadrotor that can be modeled in Gazebo through RotorS.

All the components of the nano-quadcopter are simulated by Gazebo plugins and the Gazebo physics engine. The body of the aircraft consists of four rotors, which can be placed in any location allowing configuration changes (e.g., from "+" to "×", see Sec. 3.3.1), and some sensors attached to the body (e.g., gyroscope, accelerometer, camera, barometer, laser scanner, etc.). Each rotor has properly dynamics and accounts for the most dominant aerodynamic effects. Also external influences can be taken into account, such as a wind gust, but they are neglected in this tutorial chapter.

A further block is the state estimator, used to obtain information about the state of the drone (see Sec. 2.3). While it is crucial on a real quadcopter, in simulation it can be replaced by a generic (ideal) odometry sensor (with or without noise and bias) in order to understand the effects of the state estimation. In Section 3.4.2 some graphics show how the vehicle behavior changes when the drone state is not completely available and it is partially replaced by the on-board complementary filter outputs.

In order to easily test different scenarios, ROS allows to use a suitable *launch* file. As we said before, such file allows to enable or disable the state estimator. That means that the drone orientation and angular velocities are provided by the odometry sensor when the state estimator is turned off, and by the complementary filter (that uses gyroscope and accelerometer data coming from the on-board IMU) when it is switched on (as depicted in Fig. 6). For simplicity, the proposed application considers that in both cases

[5] Bags are typically created by a tool like *rosbag*, which subscribes to one or more ROS topic, and stores the serialized message data in a file as it is received.

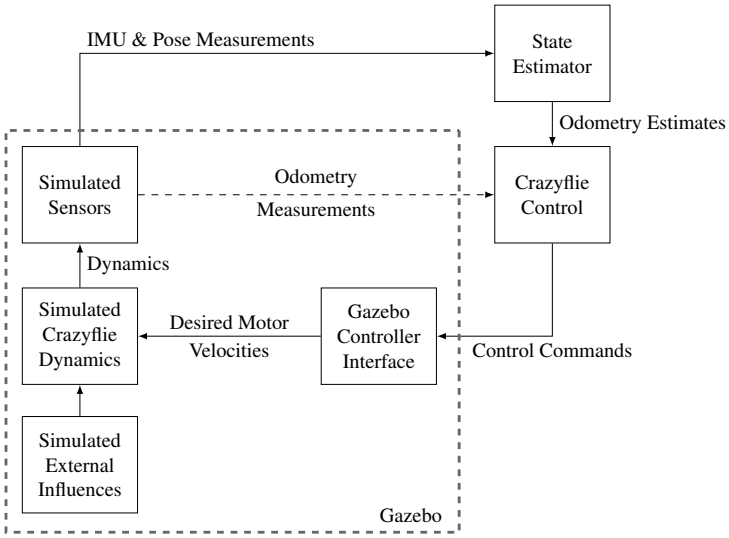

Fig. 6. Crazyflie 2.0 components in CrazyS, inspired by the RotorS structure.

the drone position and linear velocities are provided by the odometry sensor, as described in Sec. 2.2. A different possibility might arise, when drones fly indoors [66,67], when a MoCap system is used to provide such information. However, in place of the complementary filter, a more complicated state estimator has to be used in that case.

It is important to highlight how all such features make the tool potentialities endless. Once the Crazyflie is flying, higher level tasks can be carried out and tested in the simulation environment, such as simultaneous localization and mapping (SLAM) [68], planning [69], learning [70], collision avoidance [71], etc. Moreover, it is possible to evaluate easily different scenarios (e.g., how a different sensor time response affects the aircraft stability).

3.3.1 Model description and simulation

One of main objectives of using the proposed methodology is to simulate a scenario quite closely to the real world, so that it comes easy the reuse of the software architecture when porting it on the real Crazyflie vehicle (e.g., through the ROS packages Crazyswarm [33] or Crazyros [36,29]). With this aim we started from one of the available examples in RotorS (specifically the `mav_hovering_example.launch`) having a quite detailed model of drone dynamics.

Thus, we cast that model and control parts to corresponding parts of the Crazyflie nano-quadcopter by considering the specific components (see Fig. 6), the Crazyflie physical dynamics and parameters, and the perception sensors. The overall ROS architecture is depicted in Fig. 7 where the topics and nodes are represented. The whole process is the following: the desired position coordinates (x_r, y_r, z_r, ψ_r) are published

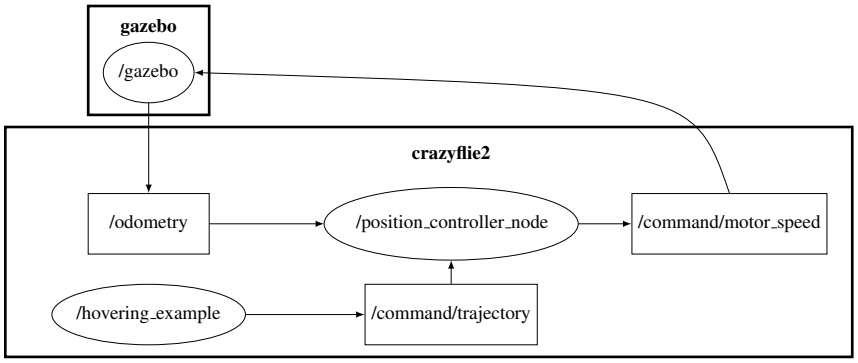

Fig. 7. Graph of ROS nodes (ellipses) and topics (squares) of the hovering example with the Crazyflie 2.0. The continuous line arrows are topic subscriptions, with directions going from the subscriber node to the publisher one.

by the *hovering_example* node on the topic *command/trajectory*, to which the *position_controller* node (i.e., the Crazyflie controller) is subscribed. The drone state (*odometry* topic) and the references are used to run the control strategy designed for the position tracking. The outputs of the control algorithm consist into the actuation commands (ω_1, ω_2, ω_3 and ω_4) sent to Gazebo (*command/motor_speed*) for the physical simulation and the corresponding graphical rendering, so to visually update the aircraft position and orientation. When the state estimator is turned off, the drone orientation (ϕ_k, θ_k and ψ_k) and angular velocities (p_k, q_k and r_k) published on the topic *odometry* are replaced by the ideal values coming from the odometry sensor. Thus, the on-board control architecture of the Crazyflie changes as depicted in Fig. 8.

RotorS uses Xacro files for describing vehicles, the same structure employed also for the sensors. Thus, for defining the Crazyflie aircraft, the XML tag structure is employed to set properties that are related to the physical features of the drone, like the

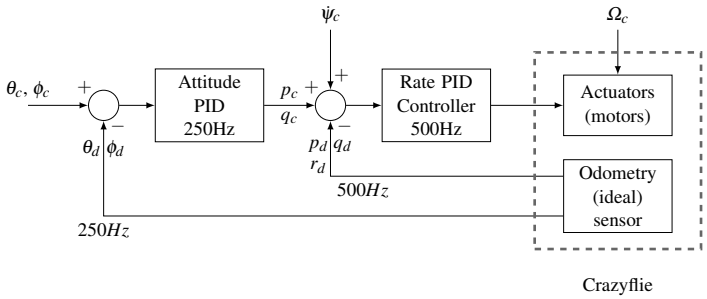

Fig. 8. On-board control architecture of the Crazyflie 2.0 when the state estimator is not considered in the simulation: the estimated data are replaced by the ideal values.

quadrotor aerodynamic coefficients [50] or other physical parameters [45]. In particular, the `crazyflie2.xacro` file (see Listing 1.6) allows to describe components and properties such as the motors constant, the rolling moment coefficient, the mass of the vehicle, the moments of inertia along the axes, the arm length, the propellers direction, and so on, in according to the aircraft model (see Sec. 2.1). Such file is executed at runtime when the simulation is going to start.

```
<robot name="crazyflie2" xmlns:xacro="http://ros.org/wiki
    /xacro">
<xacro:property name="namespace" value="$(arg mav_name)"
    />
<xacro:property name="rotor_velocity_slowdown_sim" value=
    "50" />
<xacro:property name="use_mesh_file" value="true" />
<xacro:property name="mesh_file" value="package://
    rotors_description/meshes/crazyflie2.dae" />
<xacro:property name="mass" value="0.025" />
<xacro:property name="body_width" value="0.045" />
<xacro:property name="body_height" value="0.03" />
<xacro:property name="mass_rotor" value="0.0005" />
<xacro:property name="arm_length" value="0.046" />
<xacro:property name="rotor_offset_top" value="0.024" />
<xacro:property name="radius_rotor" value="0.0225" />
<xacro:property name="sin45" value="0.707106781186" />
<xacro:property name="cos45" value="0.707106781186" />

<xacro:property name="motor_constant" value="1.28192e-08"
     />
<xacro:property name="moment_constant" value="5.964552e
    -03" />
<xacro:property name="time_constant_up" value="0.0125" />
<xacro:property name="time_constant_down" value="0.025" /
    >
<xacro:property name="max_rot_velocity" value="2618" />
<xacro:property name="rotor_drag_coefficient" value="
    8.06428e-05" />
<xacro:property name="rolling_moment_coefficient" value="
    0.000001" />

...

<xacro:vertical_rotor
robot_namespace="${namespace}"
suffix="front-right"
direction="ccw"
motor_constant="${motor_constant}"
```

```
moment_constant="${moment_constant}"
parent="${namespace}/base_link"
mass_rotor="${mass_rotor}"
radius_rotor="${radius_rotor}"
time_constant_up="${time_constant_up}"
time_constant_down="${time_constant_down}"
max_rot_velocity="${max_rot_velocity}"
motor_number="0"
rotor_drag_coefficient="${rotor_drag_coefficient}"
rolling_moment_coefficient="${rolling_moment_coefficient}
    "
color="Red"
use_own_mesh="false"
mesh="">
<origin xyz="${cos45*arm_length} -${sin45*arm_length} ${
    rotor_offset_top}" rpy="0 0 0" />
<xacro:insert_block name="rotor_inertia" />
</xacro:vertical_rotor>

...
```

Listing 1.6. Crazyflie 2.0 parameters and geometry file.

The mentioned files, i.e., `crazyflie2.xacro`, `crazyflie_base.xacro`, `component_snippets.xacro` (see Sec. 2.3.2), are related to each other making the aircraft model like a chain, where each link has a proper aim and without them the simulation cannot start. Thus, in order to facilitate the understanding and making clear how to develop an own platform, Fig. 9 illustrates the overall architecture of the simulation that is instantiated by invoking the *launch* file.

Conversely, the robot geometry has been modeled by using the open-source software Blender (see Fig. 10) and the *vertical_rotor* macro defined in the `multirotor_base.xacro` file. Starting from the mesh file available on [46], the digital representation of the propellers has been changed from a "+" configuration (Crazyflie 1.0) to a "×" configuration (Crazyflie 2.0) providing textures and materials with the `crazyflie2.dae` file (it employs the COLLADA [72] format). That illustrates how it is possible to start from a CAD file, i.e., the 3D model of the vehicle, to the simulation environment in few steps, taking care to convert the file to a format readable by Gazebo. In particular, it is possible to note how the position of the propellers was set up by varying the parameters of the tag `<origin xyz="X Y Z" rpy="ROLL PITCH YAW">` (see Listing 1.6), where `X`, `Y` and `Z` represent the x, y and z propeller coordinates in the fixed inertial frame, respectively, and `ROLL`, `PITCH` and `YAW` its attitude.

3.4 Developing a custom controller

This section (in particular Sec. 3.4.1) explains how to use the MathWorks Robotics System Toolbox [73] to build-up a SITL simulation architecture in which Simulink schemes

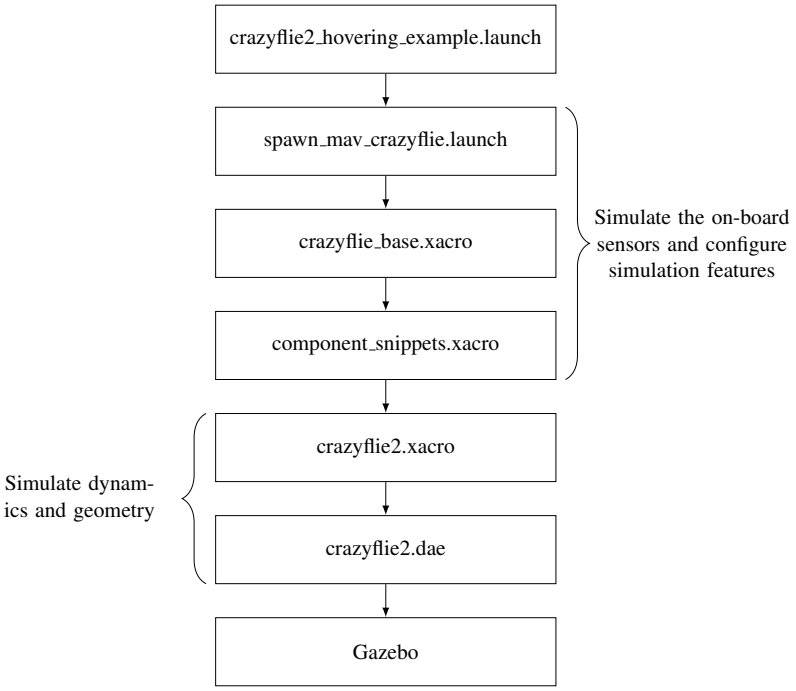

Fig. 9. The software flow diagram in CrazyS. The rectangles represent the file while the arrows the flow of calls from the *launch* file to the Gazebo 3D virtual environment.

of control loops[6] are reused and interfaced to Gazebo in order to simulate the detailed aircraft physical model. The `C++` code implementation of the Simulink schemes and their ROS integration will be discussed in the following Sec. 3.4.2 by closing the process and achieving the final and complete SITL simulation architecture. The overall procedure will be described in details, however we illustrate here the motivations of the proposed approach.

The first phase based on MathWorks RST allows in few steps to compare the results obtained from the interaction between Simulink schemes (controller) and Gazebo (physics) with the outcomes of the system completely implemented in Matlab/Simulink (both physical model and controller). In this way, implementation details like controller discretization, concurrency, timing issues, can be isolated when looking at the Matlab/Simulink platform only, while their effects can be investigated by considering the Simulink and Gazebo simulations.

[6] Matlab/Simulink is widely spread among control engineers that use those tools for designing their control strategies. Control design is not the aim of the chapter and thus we assume Simulink schemes have already been defined in an earlier phase and are available for the SITL simulation.

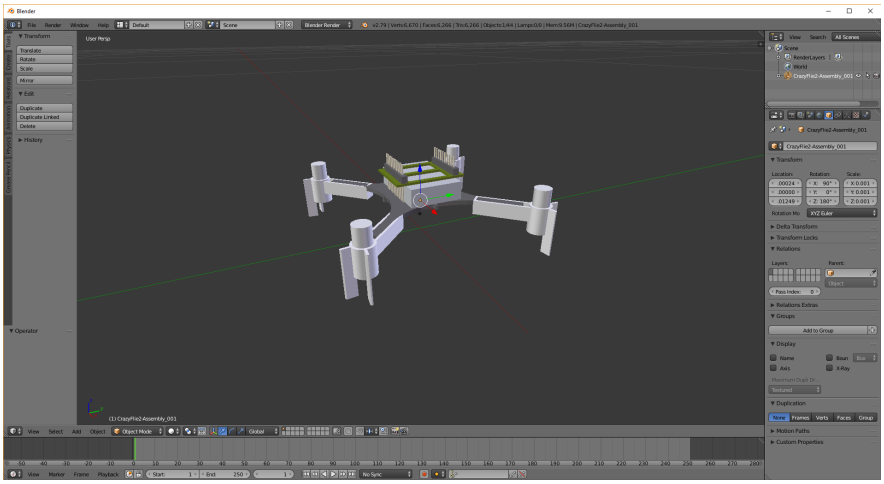

Fig. 10. Crazyflie digital representation by using the open-source software Blender.

In few words, the RST allows in an easy way to test and verify the behavior of the flight control system (see Sec. 2.2), by comparing and evaluating different control strategies, making possible to come back easily to the control design phase (whose outputs are usually the Simulink schemes) before implementing the ROS code. Such approach saves time in the development of possible problematic code and fulfills requirements of modern embedded systems development based on the well-known V-Model [31].

The entire process has been tested with the 2017b release of Matlab, but it is compatible with any Matlab release successive to 2015a. The code is specific for the use case study, but it can be easily and quickly customized to work with any quadrotor in the simulation framework.

3.4.1 Robotics System Toolbox

The MathWorks Robotics System Toolbox provides an interface [74] between Matlab/Simulink and ROS, allowing to establish a connection with the ROS master (Gazebo in our case) directly controlling the Crazyflie dynamics.

Starting from that scheme, the feedback loops are replaced by RST blocks implementing the *publish/subscribe* paradigm dealing with ROS topics, as depicted in Fig. 11. The Gazebo plugins will provide the sensors data, while the controller outputs (actuators commands) will be sent to the detailed physical model in the virtual scenario. Therefore, the Crazyflie model, that was present in the Simulink scheme when simulating the controlled drone dynamics in Matlab, can be removed. Although now the simulation is based on the phsyical engine of Gazebo and runs through the ROS middleware, any change or modification of the control law is limited to standard Simulink blocks at a higher abstraction level.

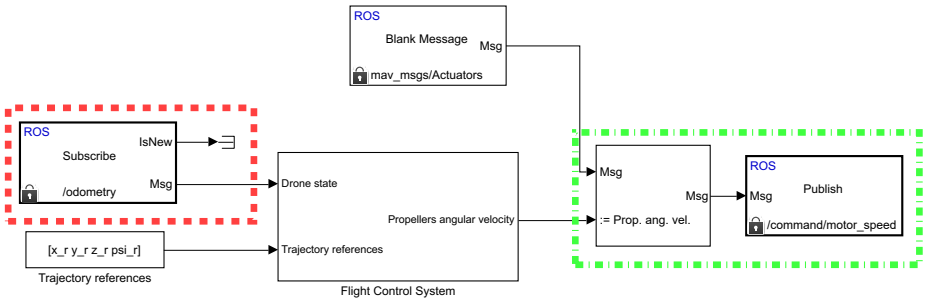

Fig. 11. Simulink control scheme by using RST blocks. The red box highlights the block implementing the ROS topic subscription to the sensors values, while the green box indicates the block in charge to publish the propellers angular velocity.

RST is available from the 2015a release of Matlab but not all types of ROS messages are supported. In particular, RST does not support `mav_msgs/Actuators` messages employed in RotorS to send the propellers angular velocities to the Gazebo physics engine, at least till Matlab release 2017b. The issue can be partially overcome by installing a suitable add-ons `roboticsAddons`, hosted on the MathWorks add-ons explore site [75], and by creating the custom messages starting from the properly ROS package [76]. Indeed, the toolbox supports the forwarding of custom messages only via Matlab scripts. Therefore, the Simulink schemes have to be adapted and integrated with Matlab scripts for exchanging data and commands with ROS and Gazebo. Due to space constraints, the whole procedure as well as the employed schemes and scripts will not be described here but all information are available in [77].

As shown in [77,78], the communication between Simulink and Gazebo needs to be synchronized via Gazebo services (*unpause* and *pause_physics*) that run and stop the simulation so to avoid data losses and system instabilities. When the scheme runs in synchronization mode, the client (Matlab) is in charge of deciding when the next step should be triggered by making the server (Gazebo) advance the simulation. In this way it is avoided any possible synchronization/communication issue arising from a real implementation of a cyberphysical system. When the control strategy is sufficiently investigated and verified, all implementation issues can be modeled and/or taken into account thus removing the artificial synchronization and proceeding with the coding for implementing the control strategy on middleware like ROS or even on a real time OS.

In Listing 1.7 the *launch* code (specifically the `crazyflie_without_controller.launch`) employed to link Matlab/Simulink with ROS and Gazebo, is reported. Such code starts the server (Gazebo) that simulates the Crazyflie dynamics and sensors. Then, Gazebo goes in stand-by waiting for the Simulink scheme implementing the controller. It will be in charge to run and pause the physical engine computations in order to simulate the controlled scenario.

```
<launch>
```

```
<arg name="mav_name" default="crazyflie2"/>
<arg name="world_name" default="basic"/>
<arg name="enable_logging" default="false" />
<arg name="enable_ground_truth" default="true" />
<arg name="enable_state_estimator" default="false" />
<arg name="log_file" default="$(arg mav_name)" />
<arg name="paused" value="true"/>
<arg name="debug" default="false"/>
<arg name="gui" default="true"/>
<arg name="verbose" default="false"/>

<env name="GAZEBO_MODEL_PATH" value="${GAZEBO_MODEL_PATH
    }:$(find rotors_gazebo)/models"/>
<env name="GAZEBO_RESOURCE_PATH" value="${
    GAZEBO_RESOURCE_PATH}:$(find rotors_gazebo)/models"/>
<include file="$(find gazebo_ros)/launch/empty_world.
    launch">
  <arg name="world_name" value="$(find rotors_gazebo)/
    worlds/$(arg world_name)_crazyflie.world" />
  <arg name="debug" value="$(arg debug)" />
  <arg name="paused" value="$(arg paused)"/>
  <arg name="gui" value="$(arg gui)" />
  <arg name="verbose" value="$(arg verbose)"/>
</include>

</launch>
```

Listing 1.7. Launch file employed to simulate the Crazyflie dynamics and sensors.

Note that although the RST supports C++ code generation [79] and it is able to generate automatically a ROS node from a Simulink scheme and deploying it into a ROS network, it is not immediate to integrate everything within RotorS obtaining, at the same time, a readable code. Thus, we followed the approach of developing manually the code paying attention to the software reuse and to modular design.

3.4.2 ROS integration

In this section it is described and analyzed the code structure that implements the controller of the vehicle. As illustrated in Fig. 7 and already referred in Sec. 3.3.1, the *nav_msgs/Odometry* messages published on the topic *odometry* by Gazebo, are handled by the *position_controller* node that has the aim of computing the propellers speed. Such speeds are later published on the *command/motor_speed* topic through *mav_msgs/Actuators* messages.

The controller implementation is divided into two main parts: the first part handles the parameters and the messages passing, and it is implemented as a ROS node (i.e., the *position_controller* node); while the second part is a library of functions,

called by the ROS node and get linked to it at compilation time by using the `CMakeList.txt` file[7], that implements all required computations (the *crazyflie_onboard_controller*, the *crazyflie_complementary_filter*, etc.). Parameters (both controller and vehicle ones) are set in YAML files[8] (e.g., `controller_crazyflie2.yaml`, `crazyflie2.yaml`, etc.) and passed to the ROS parameter server by using the *launch* file in which the following line between the `<node>` tags is added.

```
<rosparam command="load" file= "$(find rotors_gazebo)/
   resource/controller_crazyflie2.yaml"/>
```

The ROS parameter server makes those values available to the ROS network avoiding to build-up all executables every time a slight modification occurs (a very time consuming step). In this way it is possible to modify the controller gains described in Sec. 2.2 or the vehicle parameters (like the Crazyflie mass, its inertia or the rotor configuration) in a very simple way, evaluating more quickly how system performance changes at each different simulation.

In order to show the potentialities and the flexibility of the platform, a ROS node has been developed to simulate the scenario with and without the Crazyflie on-board state estimator. The node is able to catch the data coming from Gazebo, or other nodes in the ROS network (e.g., the *hovering_example* that is in charge to publish the trajectory to follow), and to send the actuation commands (ω_1, ω_2, ω_3 and ω_4) to the Gazebo physics engine. To that aim, a suitable *launch* file, i.e., the `crazyflie2_hovering_example.launch`, was made to handle the simulation starting. That file allows to switch from a scenario to another one by varying the boolean value of the variable `enable_state_estimator` as illustrated in Sec. 3.2.

When the state estimator is disabled (i.e., the odometry sensor is used), the callback methods work only with the *Odometry* (for reading sensors data) and *MultiDOFJointTrajectory* (for reading trajectory references) messages. Instead, when the complementary filter works a callback method considers also the IMU messages. ROS timers have been introduced for dealing with the update rate of the Crazyflie on-board control and the sampling time of the position control loop (chosen to be 1 ms as defined in `basic_crazyflie.world` file). In both cases, at each time step, the method `CalculateRotorVelocities` computes the rotor velocities ω_i from the controller's input and drone current (or estimated) state.

In order to facilitate the reuse of the software modules developed in CrazyS, the inner loop (the attitude and rate controllers, i.e., the Crazyflie on-board controller, see Fig. 5) and the complementary filter have been implemented as libraries. In such a way, state estimators and controllers can be easily employed in any node of the ROS network or replaced by improving Crazyflie's performance. In Figure 12 the CrazyS packages structures and the main files included into the CrazyS ROS repository are depicted.

The overall system has been simulated through Gazebo/ROS and the results illustrate in a direct way how the system works (the corresponding video is available [80]):

[7] It manages the build process of the software. It supports directory hierarchies and applications that depend on multiple libraries.

[8] YAML (YAML Ain't Markup Language) is a human-readable data serialization language and is commonly used for configuration files. YAML targets many of the same communications applications as XML but has a minimal syntax which intentionally breaks.

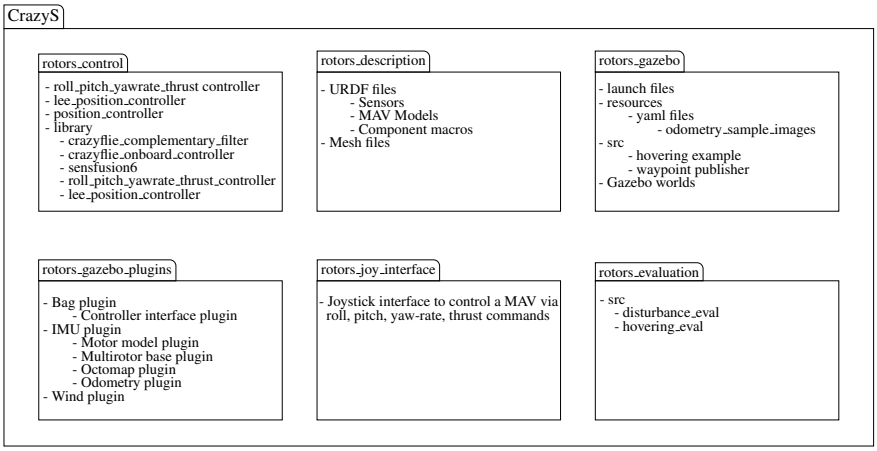

Fig. 12. Structures of the packages contained in the CrazyS repository.

the Crazyflie 2.0 keeps the hovering position until the simulation stops. Moreover, from the video [81] it appears evident how the control system is able to compensate attitude and position disturbances coming back to the hovering position. Finally, a further scenario (video [82]) considers the "real" sensors (see Fig. 5) by taking into account the IMU and the complementary filter. All the experiments have been carried out by using Kinetic Kame version of ROS (as we said before, it is also compatible with the Indigo Igloo version) for visualization and scripting, and they were run on a workstation based on Xeon E3-1245 3.50 GHz, Gallium 0.4 on NV117 and 32GB of RAM with Ubuntu 16.04.

Figure 13 reports numerical results obtained in Matlab/Simulink (both the physical model and control are simulated there) by considering the perfect state information ("n" subscript signals, solid lines). Simulation results obtained in Gazebo/ROS ("s" subscript signals) are depicted, as well. In particular, the subscript "imu" has been used to discriminate the data when the state estimator is in the loop. The controller works quite well in all considered scenarios. Nevertheless, designing a high performance hovering controller is not the aim of this work but we considered such task to show the advantages of the SITL simulation implemented through the CrazyS platform. From a control point of view, better results might be obtained by using a Kalman filter [83] (already developed in the Crazyflie firmware but not used as the default state estimator, probably due to the increase of computational burden) or the new on-board control [84] released with the 2018.10 version of the firmware.

3.5 Continuous integration system

In this section we illustrate our proposed solution to link the continuous integration (CI) open-source platform TravisCI [85] with the CrazyS repository. Moreover we de-

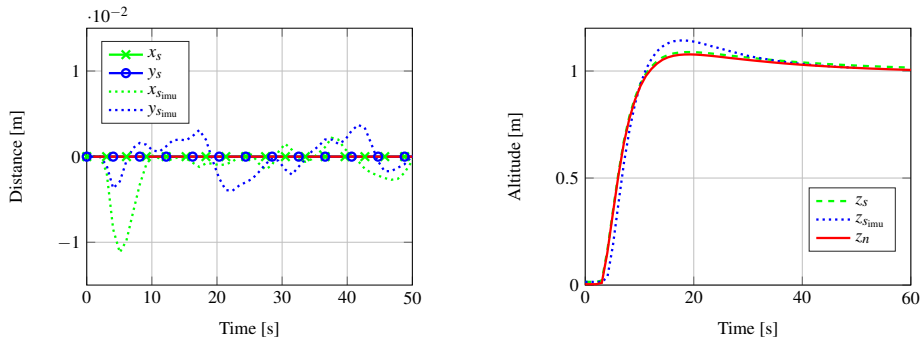

Fig. 13. Drone position during the hovering example. In red the numerical results (Matlab/Simulink) and in blue and green the simulation results (Gazebo/ROS) with and without the real sensors.

scribe the corresponding advantages that a CI system may give when developing a ROS component like CrazyS.

Listing 1.8 reports the script used to configure the CrazyS repository with TravisCI. The code is based on an existing open-source project [86] and has been customized to make it compatible with the Kinetic Kame distro of ROS. Also, a pull request [87] has been opened to share our code with other researchers and developers.

```
matrix:
  include:
    - os: linux
      dist: trusty
      sudo: required
      env: ROS_DISTRO=indigo
    - os: linux
      dist: xenial
      sudo: required
      env: ROS_DISTRO=kinetic

language:
  - generic
cache:
  - apt

env:
  global:
    - ROS_CI_DESKTOP="`lsb_release -cs`"
    - CI_SOURCE_PATH=$(pwd)
    - ROSINSTALL_FILE=$CI_SOURCE_PATH/dependencies.
   rosinstall
```

```
    - CATKIN_OPTIONS=$CI_SOURCE_PATH/catkin.options
    - ROS_PARALLEL_JOBS='-j8 -l6'
    - PYTHONPATH=$PYTHONPATH:/usr/lib/python2.7/dist-
   packages:/usr/local/lib/python2.7/dist-packages

before_install:
    - sudo sh -c 'echo "deb http://packages.ros.org/ros/
   ubuntu $ROS_CI_DESKTOP main" > /etc/apt/sources.list.d
   /ros-latest.list'
    - wget http://packages.ros.org/ros.key -O - | sudo
   apt-key add -
    - if [[ "$ROS_DISTRO" == "indigo" ]]; then sudo apt-
   get update && sudo apt-get install dpkg; fi
    - if [[ "$ROS_DISTRO" == "kinetic" ]]; then sudo rm /
   var/lib/dpkg/lock; fi
    - if [[ "$ROS_DISTRO" == "kinetic" ]]; then sudo dpkg
    --configure -a; fi
    - sudo apt-get update
    - sudo apt-get install ros-$ROS_DISTRO-desktop-full
   ros-$ROS_DISTRO-joy ros-$ROS_DISTRO-octomap-ros python
   -wstool python-catkin-tools
    - sudo apt-get install protobuf-compiler libgoogle-
   glog-dev
    - sudo rosdep init
    - rosdep update
    - source /opt/ros/$ROS_DISTRO/setup.bash

install:
    - mkdir -p ~/catkin_ws/src
    - cd ~/catkin_ws/src
    - catkin_init_workspace
    - catkin init
    - git clone https://github.com/gsilano/CrazyS.git
    - git clone https://github.com/gsilano/mav_comm.git
    - rosdep update
    - cd ~/catkin_ws
    - rosdep install --from-paths src -i
    - catkin build
    - echo "source ~/catkin_ws/devel/setup.bash" >> ~/.
   bashrc
    - source ~/.bashrc
```

Listing 1.8. TravisCI script for Ubuntu 14.04 and 16.04 with ROS Indigo Igloo and Kinetic Kame, respectively.

In order to use TravisCI, a GitHub account and the TravisCI script are all the necessary components. The script, i.e., the `.travis.yml` file, has to be put in the root of the active repository[9].

Looking at the listing, the file is split into five main parts: `include`, `language` and `cache`, `env`, `before_install` and `install`. In the first part, the `matrix` command tells TravisCI that two machines should be created sequentially. That allows to build and to test the code with different ROS distros (Indigo Igloo and Kinetic Kame, in our case) and OS (Thrusty and Xenial, 14.04 and 16.04 versions of Ubuntu, respectively) through the `include` command.

The second part, `language` and `cache`, enables the installing of the required ROS packages (see Sec. 3.1). It allows to customize the environment running in a virtual machine. Finally, the parts `env` and `before_install` configure all variables (they are used to trigger a build matrix) and system dependencies.

When the process starts, the catkin workspace is build with all the packages under integration (the commands listed in the `install` section). TravisCI clones the GitHub repository(-ies) into a new virtual environment, and carries out a series of tasks to build and test the code. If one or more of those tasks fails, the build is considered *broken*. If none of the tasks fails, the build is considered *passed*, and TravisCI can deploy the code to a web server, or an application host. In particular, the build is considered *broken* when one or more of its jobs complete with a state that is not passed:

- *errored*: a command in the `before_install` or `install` phase returned a non-zero exit code. The job stops immediately;
- *failed*: a command in the script phase returned a non-zero exit code. The job continues to run until it completes;
- *canceled*: a user cancels the job before it completes.

At the end of the process, email notifications are sent to all the listed contributors members of the repository. The notifications can be forwarded: on success, on failure, always or never. Finally, the CI system can be also employed to automatically generate documentation starting from the source code and to link it to an online hosting. It is very useful when the project is going to increase or a lot of people are working on it or, more generally, when it is difficult to have an overview of the developed code. Further information on how to use the CI system and how to configure it can be found in [88].

Such procedure allows to easily verify the code quality, underlying errors and warnings through automated software build (including tests), that may not appear when building on own machine. It also ensures that modules working individually (e.g., SLAM, vision or sensors fusion algorithms) do not fail when they are put together due to causes that were difficult to predict during the development phase. For all such reasons, having a software tool able to catch what happened and why it happened, and able to suggest possible solutions, is extremely desirable when working with complex platforms as Gazebo and ROS.

[9] For students or academics, GitHub gives the possibility to build infinite private builds.

4 Conclusion and future work

In this tutorial chapter we illustrated how to expand the functionalities of the ROS package RotorS for modeling and integrating the nano-quadcopter Crazyflie 2.0 in a detailed simulation framework achieving a complete SITL simulation solution. The overall approach aimed at developing the system in a modular way by facilitating the reuse of software modules. The proposed methodology allows to combine different state estimators and control algorithms, evaluating the performances before deploying them on a real device.

The chapter discussed the CrazyS platform from the installation to the development of a custom controller and the presentation has been thought not only for researchers but also for educational purposes, so that interested students might work in a complete and powerful environment developing their own algorithms.

Future directions for this works can include several aspects. Firstly, controller's code and all proposed algorithms should be tested in real-world experiments on the real Crazyflie platform in different scenarios, thus allowing to understand in a quantitative way how the CrazyS platform reflects the real drone behavior. Secondly, the latest firmware release, the 2018.10, may be included in the repository, aligning CrazyS with the current version of the quadcopter. Finally, it may be possible to look for some improvements of the inner loop (on-board controller) that, after having been tested on CrazyS, can be thought to replace the on-board controller of the Crazyflie.

References

1. D. Scaramuzza, M. C. Achtelik, L. Doitsidis, F. Friedrich, E. Kosmatopoulos, A. Martinelli, M. W. Achtelik, M. Chli, S. Chatzichristofis, L. Kneip, D. Gurdan, L. Heng, G. H. Lee, S. Lynen, M. Pollefeys, A. Renzaglia, R. Siegwart, J. C. Stumpf, P. Tanskanen, C. Troiani, S. Weiss, and L. Meier, "Vision-Controlled Micro Flying Robots: From System Design to Autonomous Navigation and Mapping in GPS-Denied Environments," *IEEE Robotics Automation Magazine*, vol. 21, no. 3, pp. 26–40, 2014.
2. M. A. Stuart, L. L. Marc, and J. C. Friedland, "High Resolution Imagery Collection for Post-Disaster Studies Utilizing Unmanned Aircraft Systems," *Photogrammetric Engineering and Remote Sensing*, vol. 80, no. 12, pp. 1161–1168, 2014.
3. D. Erdos, A. Erdos, and S. E. Watkins, "An experimental UAV system for search and rescue challenge," *IEEE Aerospace and Electronic Systems Magazine*, vol. 28, no. 5, pp. 32–37, 2013.
4. S. Choi and E. Kim, "Image acquisition system for construction inspection based on small unmanned aerial vehicle," *Lecture Notes in Electrical Engineering*, vol. 352, pp. 273–280, 2015.
5. C. Eschmann, C. M. Kuo, C. H. Kuo, and C. Boller, "Unmanned aircraft systems for remote building inspection and monitoring," in *6th European Workshop - Structural Health Monitoring*, vol. 2, 2012, pp. 1179–1186.
6. F. Fraundorfer, L. Heng, D. Honegger, G. H. Lee, L. Meier, P. Tanskanen, and M. Pollefeys, "Vision-based autonomous mapping and exploration using a quadrotor MAV," in *IEEE International Conference on Intelligent Robots and Systems*, 2012, pp. 4557–4564.

7. B. Kamel, M. C. S. Santana, and T. C. De Almeida, "Position estimation of autonomous aerial navigation based on hough transform and harris corners detection," in *International conference on Circuits, Systems, Electronics, Control and Signal Processing*, 2010, pp. 148–153.
8. K. Kanistras, G. Martins, M. J. Rutherford, and K. P. Valavanis, "A survey of unmanned aerial vehicles (UAVs) for traffic monitoring," in *International Conference on Unmanned Aircraft Systems*, 2013, pp. 221–234.
9. J. Xu, G. Solmaz, R. Rahmatizadeh, D. Turgut, and L. Boloni, "Animal monitoring with unmanned aerial vehicle-aided wireless sensor networks," in *IEEE 40th conference on local computer networks*, 2015, pp. 125–132.
10. D. Anthony, S. Elbaum, A. Lorenz, and C. Detweiler, "On crop height estimation with UAVs," in *IEEE International conference on Intelligent Robots and Systems*, 2014, pp. 4805–4812.
11. C. Bills, J. Chen, and A. Saxena, "Autonomous MAV flight in indoor environments using single image perspective cues," in *IEEE international conference on robotics and automation*, 2011, pp. 5776–5783.
12. M. Blosch, S. Weiss, D. Scaramuzza, and R. Siegwart, "Vision based MAV navigation in unknown and unstructured environments," in *IEEE international conference on robotics and automation*, 2010, pp. 21–28.
13. B. Landry, "Planning and control for quadrotor flight through cluttered environments," Master's thesis, MIT, 2015.
14. D. Ferreira de Castro and D. A. dos Santos, "A Software-in-the-Loop Simulation Scheme for Position Formation Flight of Multicopters," *Journal of Aerospace Technology and Management*, vol. 8, no. 4, pp. 431–440, 2016.
15. M. Mancini, G. Costante, P. Valigi, T. A. Ciarfuglia, J. Delmerico, and D. Scaramuzza, "Toward Domain Independence for Learning-Based Monocular Depth Estimation," *IEEE Robotics and Automation Letters*, vol. 2, no. 3, pp. 1778–1785, 2017.
16. T. Hinzmann, J. L. Schönberger, M. Pollefeys, and R. Siegwart, "Mapping on the Fly: Real-Time 3D Dense Reconstruction, Digital Surface Map and Incremental Orthomosaic Generation for Unmanned Aerial Vehicles," in *Field and Service Robotics*. Springer International Publishing, 2018, pp. 383–396.
17. I. A. Sucan and S. Chitta, "Moveit!" 2013. [Online]. Available: http://moveit.ros.org/
18. A. Tallavajhula and A. Kelly, "Construction and validation of a high fidelity simulator for a planar range sensor," in *IEEE Conference on Robotics and Automation*, 2015, pp. 6261–6266.
19. R. Diankov and J. Kuffner, "Openrave: A planning architecture for autonomous robotics," Robotics Institute, Pittsburgh, PA, Tech. Rep. 79, 2008.
20. A. Elkady and T. Sobh, "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography," *Journal of Robotics*, 2012, article ID 959013.
21. N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *IEEE International Conference on Intelligent Robots and Systems*, 2004, pp. 2149–2154.
22. E. Rohmer, S. P. N. Singh, and M. Freese, "V-REP: a Versatile and Scalable Robot Simulation Framework," in *IEEE International Conference on Intelligent Robots and Systems*, 2013, pp. 1321–1326.
23. O. Michel, "Webots professional mobile robot simulation," *International Journal of Advanced Robotics Systems*, vol. 1, pp. 39–42, 2004.
24. S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," in *Field and Service Robotics*, 2017.

25. G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan, "Modular open robots simulation engine: MORSE," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 46–51.
26. Bitcraze AB, "Crazyflie official website." [Online]. Available: https://www.bitcraze.io/
27. J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. von Stryk, "Comprehensive Simulation of Quadrotor UAVs Using ROS and Gazebo," in *Simulation, Modeling, and Programming for Autonomous Robots*, I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, Eds. Springer Berlin Heidelberg, 2012, pp. 400–411.
28. F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, "RotorS – A Modular Gazebo MAV Simulator Framework," in *Robot Operating System (ROS): The Complete Reference (Volume 1)*, K. Anis, Ed. Springer International Publishing, 2016, pp. 595–625.
29. W. Hönig and N. Ayanian, "Flying Multiple UAVs using ROS," in *Robot Operating System (ROS): The Complete Reference (Volume 2)*, A. Koubaa, Ed. Springer International Publishing, 2017, pp. 83–118.
30. H. Shokry and M. Hinchey, "Model-Based Verification of Embedded Software," *Computer*, vol. 42, no. 4, pp. 53–59, 4 2009.
31. H. Van der Auweraer, J. Anthonis, S. De Bruyne, and J. Leuridan, "Virtual engineering at work: the challenges for designing mechatronic products," *Engineering with Computers*, vol. 29, no. 3, pp. 389–408, 2013.
32. A. Aminzadeh, M. Atashgah, and A. Roudbari, "Software in the loop framework for the performance assessment of a navigation and control system of an unmanned aerial vehicle," *IEEE Aerospace and Electronic Systems Magazine*, vol. 33, no. 1, pp. 50–57, 1 2018.
33. J. A. Preiss, W. Honig, G. S. Sukhatme, and N. Ayanian, "Crazyswarm: A large nano-quadcopter swarm," in *IEEE International Conference on Robotics and Automation*, 2017, pp. 3299–3304.
34. B. Galea and P. G. Kry, "Tethered flight control of a small quadrotor robot for stippling," in *IEEE International Conference on Intelligent Robots and Systems*, 2017, pp. 1713–1718.
35. B. Araki, J. Strang, S. Pohorecky, C. Qiu, T. Naegeli, and D. Rus, "Multi-robot path planning for a swarm of robots that can both fly and drive," in *IEEE International Conference on Robotics and Automation*, 2017, pp. 5575–5582.
36. W. Hönig, C. Milanes, L. Scaria, T. Phan, M. Bolas, and N. Ayanian, "Mixed reality for robotics," in *IEEE International Conference on Intelligent Robots and Systems*, 2015, pp. 5382–5387.
37. W. Giernacki, M. Skwierczyski, W. Witwicki, P. Wroski, and P. Kozierski, "Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering," in *22nd International Conference on Methods and Models in Automation and Robotics*, 2017, pp. 37–42.
38. N. Bucki and M. W. Mueller, "Improved Quadcopter Disturbance Rejection Using Added Angular Momentum," in *2018 IEEE International Conference on Intelligent Robots and Systems*, 2018.
39. G. Silano, "CrazyS GitHub Repository," 2018. [Online]. Available: https://github.com/gsilano/CrazyS
40. ——, "CrazyS GitHub pull request on RotorS," 2018. [Online]. Available: https://github.com/ethz-asl/rotors_simulator/pull/465
41. Bitcraze AB, "Crazyflie 2.0 hardware specification," Bitcraze Wiki, 2018. [Online]. Available: https://goo.gl/1kLDqc
42. G. Silano, E. Aucone, and L. Iannelli, "CrazyS: a software-in-the-loop platform for the Crazyflie 2.0 nano-quadcopter," in *2018 26th Mediterranean Conference on Control and Automation*, June 2018, pp. 352–357.

43. C. Luis and J. Le Ny, "Design of a Trajectory Tracking Controller for a Nanoquadcopter," École Polytechnique de Montréal, Tech. Rep., 2016. [Online]. Available: https://arxiv.org/pdf/1608.05786.pdf
44. B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics - Modelling, Planning and Control*, 2nd ed., ser. Advanced Textbooks in Control and Signal Processing. Springer, 2008.
45. J. Förster, "System identification of the Crazyflie 2.0 nano quadrocopter," Bachelor's Thesis, 2015, ETH Zurich. [Online]. Available: https://www.research-collection.ethz.ch/handle/20.500.11850/214143
46. The Automatic Coordination of Teams Lab, "GitHub Repository, RotorS fork, crazyflie-dev branch." [Online]. Available: https://goo.gl/tBbS9G
47. Vicon Motion Systems, "Vicon official website," 2018. [Online]. Available: https://www.vicon.com
48. NaturalPoint, Inc., "Optitrack official website," 2018. [Online]. Available: http://optitrack.com/
49. Qualisys AB, "Qualisys official website," 2018. [Online]. Available: https://www.qualisys.com/
50. G. Subramanian, "Nonlinear control strategies for quadrotors and CubeSats," Master's thesis, University of Illinois, 2015.
51. S. O. H. Madgwick, A. J. L. Harrison, and R. Vaidyanathan, "Estimation of IMU and MARG orientation using a gradient descent algorithm," in *2011 IEEE International Conference on Rehabilitation Robotics*, June 2011, pp. 1–7.
52. J. Myungsoo, S. I. Roumeliotis, and G. S. Sukhatme, "State estimation of an autonomous helicopter using Kalman filtering," in *1999 IEEE International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients*, vol. 3, 1999, pp. 1346–1353.
53. J. M. Roberts, P. I. Corke, and G. Buskey, "Low-cost flight control system for a small autonomous helicopter," in *2003 IEEE International Conference on Robotics and Automation*, vol. 1, Sept 2003, pp. 546–551.
54. J. Rehder, J. Nikolic, T. Schneider, T. Hinzmann, and R. Siegwart, "Extending kalibr: Calibrating the Extrinsics of Multiple IMUs and of Individual Axes," in *IEEE International Conference on Robotics and Automation*, 2016, pp. 4304–4311.
55. S. Glaser and W. Woodall, "Xacro (2015)." [Online]. Available: https://wiki.ros.org/xacro
56. RotorS GitHub issues tracker, "Increase Odometry sensor rate." [Online]. Available: https://github.com/ethz-asl/rotors_simulator/issues/423
57. Kalibr issue tracker, "Kalibr Calibration." [Online]. Available: https://github.com/imrasp/LearnVI_Drone/issues/1#issuecomment-350726256
58. ——, "Obtaining IMU parameterss from datasheet," 2018. [Online]. Available: https://github.com/ethz-asl/kalibr/issues/63
59. Kalibr GitHub wiki, "IMU Noise Model, Kalibr Wiki." [Online]. Available: https://github.com/ethz-asl/kalibr/wiki/IMU-Noise-Model
60. J. Zheng, M. Qi, K. Xiang, and M. Pang, "IMU Performance Analysis for a Pedestrian Tracker," in *Intelligent Robotics and Applications*, Y. Huang, H. Wu, H. Liu, and Z. Yin, Eds. Springer International Publishing, 2017, pp. 494–504.
61. W. Woodall, "Creating a workspace for catkin," 2018. [Online]. Available: http://wiki.ros.org/catkin/Tutorials/create_a_workspace
62. Autonomous System Laboratory, "mav_comm repository," 2018. [Online]. Available: https://github.com/ethz-asl/mav_comm
63. Z. Liu and K. Hedrick, "Dynamic surface control techniques applied to horizontal position control of a quadrotor," in *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*, 2016, pp. 138–144.

64. S. Islam, M. Faraz, R. K. Ashour, G. Cai, J. Dias, and L. Seneviratne, "Adaptive sliding mode control design for quadrotor unmanned aerial vehicle," in *2015 International Conference on Unmanned Aircraft Systems*, 2015, pp. 34–39.
65. Z. T. Dydek, A. M. Annaswamy, and E. Lavretsky, "Adaptive Control of Quadrotor UAVs: A Design Trade Study With Flight Evaluations," *IEEE Transactions on Control Systems Technology*, vol. 21, no. 4, pp. 1400–1406, 2013.
66. A. Weinstein, A. Cho, G. Loianno, and V. Kumar, "Visual Inertial Odometry Swarm: An Autonomous Swarm of Vision-Based Quadrotors," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 1801–1807, 2018.
67. A. S. Vempati, M. Kamel, N. Stilinovic, Q. Zhang, D. Reusser, I. Sa, J. Nieto, R. Siegwart, and P. Beardsley, "PaintCopter: An Autonomous UAV for Spray Painting on Three-Dimensional Surfaces," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 2862–2869, 2018.
68. O. Dunkley, J. Engel, J. Sturm, and D. Cremers, "Visual-Inertial Navigation for a Camera-Equipped 25g Nano-Quadrotor," in *IEEE International Conference on Intelligent Robots and Systems*, 2014, pp. 1–2.
69. L. Campos-Macías, D. Gómez-Gutiérrez, R. Aldana-López, R. de la Guardia, and J. I. Parra-Vilchis, "A Hybrid Method for Online Trajectory Planning of Mobile Robots in Cluttered Environments," *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 935–942, 2017.
70. S. Bansal, A. K. Akametalu, F. J. Jiang, F. Laine, and C. J. Tomlin, "Learning quadrotor dynamics using neural network for flight control," in *IEEE Conference on Decision and Control*, 2016, pp. 4653–4660.
71. L. Matthies, R. Brockers, Y. Kuwata, and S. Weiss, "Stereo vision-based obstacle avoidance for micro air vehicles using disparity space," in *2014 IEEE International Conference on Robotics and Automation*, May 2014, pp. 3242–3249.
72. T. Field, R. Diankov, I. Sucan, and J. Kay, "COLLADA URDF," ROS Wiki, 2018. [Online]. Available: http://wiki.ros.org/collada_urdf
73. MathWorks, "Robotics System Toolbox," MathWorks official website. [Online]. Available: https://www.mathworks.com/products/robotics.html
74. ——, "Connect to a ROS-enabled Robot from Simulink," MathWorks official website. [Online]. Available: https://it.mathworks.com/help/robotics/examples/connect-to-a-ros-enabled-robot-from-simulink.html
75. ——, "Install Robotics System Toolbox Add-ons," MathWorks official website. [Online]. Available: https://it.mathworks.com/help/robotics/ug/install-robotics-system-toolbox-support-packages.html
76. ——, "Create Custom Messages from ROS Package," MathWorks official website. [Online]. Available: https://it.mathworks.com/help/robotics/ug/create-custom-messages-from-ros-package.html
77. G. Silano, "CrazyS wiki," GitHub. [Online]. Available: https://github.com/gsilano/CrazyS/wiki/Interfacing-CrazyS-through-MATLAB
78. ——, "Crazyflie hovering example by using the Robotics System Toolbox." YouTube, 2018. [Online]. Available: https://youtu.be/ZPyMnu7A11s
79. MathWorks, "Generate a Standalone ROS Node from Simulink," MathWorks official website. [Online]. Available: https://it.mathworks.com/help/robotics/examples/generate-a-standalone-ros-node-in-simulink.html
80. G. Silano, "Crazyflie 2.0 hovering example when only the ideal odometry sensor is in the loop," YouTube, 2018. [Online]. Available: https://youtu.be/pda-tuULewM
81. ——, "Crazyflie 2.0 hovering example when disturbances act on the drone," YouTube, 2018. [Online]. Available: https://youtu.be/sobBFbgkiEA
82. ——, "Crazyflie 2.0 hovering example when the state estimator and the on-board sensors are taking into account," YouTube, 2018. [Online]. Available: https://youtu.be/qsrYCUSQ-S4

83. M. W. Mueller, M. Hamer, and R. D'Andrea, "Fusing ultra-wideband range measurements with accelerometers and rate gyroscopes for quadrocopter state estimation," in *IEEE International Conference on Robotics and Automation*, 2015, pp. 1730–1736.
84. D. Mellinger and V. Kumar, "Minimum snap trajectory generation and control for quadrotors," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 2520–2525.
85. Travis CI, GMBH, "TravisCI official website," 2018. [Online]. Available: https://travis-ci.org/
86. F. Duvallet, "ROS package continuous integration using Travis-CI repository," 2018. [Online]. Available: https://github.com/felixduvallet/ros-travis-integration
87. G. Silano, "ROS TravisCI Integration pull request," 2018. [Online]. Available: https://github.com/felixduvallet/ros-travis-integration/pull/12
88. Travis CI, GMBH, "Getting started with TravisCI," 2018. [Online]. Available: https://docs.travis-ci.com/user/getting-started/

Authors' Biographies

Giuseppe Silano received the bachelor's degree in computer engineering (2012) and the master's degree in electronic engineering for automation and telecommunication (2016) from the University of Sannio, Benevento, Italy. In 2016, he was the recipient of a scholarship entitled "Advanced control systems for the coordination among terrestrial autonomous vehicles and UAVs". Actually, he is a Ph.D. student at the University of Sannio since 2016. His research interests are in simulation and control, objects detection and tracking from images, state estimation, and planning for micro aerial vehicles. He was among the finalists of the "Aerial robotics control and perception challenge", the Industrial Challenge of the 26th Mediterranean Conference on Control and Automation (MED'18). Mr. Silano is a member of the IEEE Control System Society and IEEE Robotics and Automation Society.

Luigi Iannelli received the master's degree (Laurea) in computer engineering from the University of Sannio, Benevento, Italy, in 1999, and the Ph.D. degree in information engineering from the University of Napoli Federico II, Naples, Italy, in 2003. He was a guest researcher and professor at the Department of Signals, Sensors, and Systems, Royal Institute of Technology, Stockholm, Sweden, and then a research assistant at the Department of Computer and Systems Engineering, University of Napoli Federico II. In 2015, he was a guest researcher at the Johann Bernoulli Institute of Mathematics and Computer Science, University of Groningen, Groningen, The Netherlands. In 2004, he joined the University of Sannio as an assistant professor, and he has been an associate professor of automatic control since 2016. His current research interest include analysis and control of switched and nonsmooth systems, stability analysis of piecewise-linear systems, smart grid control and applications of control theory to power electronics and unmanned aerial vehicles. Dr. Iannelli is a member of the IEEE Control System Society, the IEEE Circuits and Systems Society, and the Society for Industrial and Applied Mathematics.