POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Mechatronic Engineering

Tesi di Laurea Magistrale

ROS/Gazebo based simulation environment for UAV controller performance evaluation

Supervisors
prof. Alessandro Rizzo
prof. Kimon Valavanis
prof. Matt Rutherford
Simone Martini

Candidato
Edoardo Todde

Anno Accademico 2022-2023

To my Sister, Mom and Dad, thanks to you for all.

Abstract

This project aims to create a ROS/Gazebo based environment for UAV controller performance evaluation through simulated experiments. A general purpose interface is also developed to integrate MATLAB/Simulink with ROS/Gazebo. The combined setting allows for incorporating realistic environment constraints and uncertainty, system model requirements, and other exogenous factors into the testing framework.

This work takes as case of study a quadrotor, which is widely used nowadays in military, commercial and private purposes; these systems have a highly nonlinear behaviour and different challenges to deal with like sensor noise, external disturbances (i.e. wind gust), hard and precise maneuvers and payload release. The model of the quadrotor is implemented on Gazebo taking into account a detailed aerodynamics model and feedback sensor noise. All these effects are implemented on the URDF model with plugins that are built on the testbench and can be activated by the user depending on the simulation purpose.

The use of ROS/Gazebo and the implementation with MATLAB isn't a straightforward first approach application, so this thesis try to create and explain a better and clear workflow to achieve the controller simulation. The work presents the experimental data from simulation of different types of control techiniques to show if the interface between the software tools is reliable. The simulated experiment on ROS/Gazebo and its performance evaluation lead to a better testing phase and an optimization in the sofftware-in-the-loop step in the design flow.

Contents

1 Introduction 1
- 1.1 Introduction and overview . . . 1
- 1.2 Thesis organization . . . 3

2 Literature review 5
- 2.1 Mathematical Model . . . 5
- 2.2 Control Problem . . . 11

3 Problem Statement 15
- 3.1 Software Environment . . . 15
  - 3.1.1 ROS . . . 15
  - 3.1.2 Gazebo . . . 19
  - 3.1.3 Matlab/Simulink . . . 19

4 ROS package 21
- 4.1 URDF . . . 21
  - 4.1.1 Code Explanation . . . 22
- 4.2 Launch and world files . . . 25
  - 4.2.1 Launch . . . 25
  - 4.2.2 World . . . 26
- 4.3 Dynamic and Feedback plugin . . . 27

5 Bridge Matlab/Simulink-ROS/Gazebo 31
- 5.1 ROS input/output . . . 32
- 5.2 Model validation set up . . . 32

6 Simulation 35
- 6.1 Model validation results . . . 35
- 6.2 PID deployement . . . 54

7 Conclusion and future work 57

A Code 59
- A.1 quadrotor.urdf.xacro . . . 59
- A.2 quadrotor.launch . . . 61

A.3 default.world 62
A.4 motorpid_plugin.cpp 63
A.5 aerodynamic_plugin.cpp 67
A.6 odometry_plugin.cpp 72

Chapter 1

Introduction

1.1 Introduction and overview

Unmanned Aerial Systems (UAS), which are multi-component systems that incorporate a ground-based controller, an unmanned aerial vehicle (UAV), and a communications system between the two, have gained enormous popularity in recent decades. These systems, often known as drones, are unmanned aircraft that may be remotely controlled or fly autonomously without the need for a human pilot on board. These systems are gaining popularity because they enable a wide range of missions and tasks that typically require manned flight with aircraft like planes and helicopters to be completed at a fraction of the cost. They also have the potential to be used to increase human productivity or perform novel tasks that are either too risky or impossible for unaided humans.

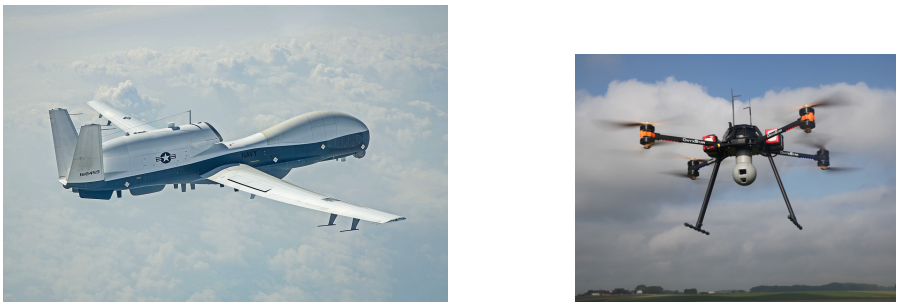
Early UAV generations were mostly utilized for military purposes like surveillance and reconnaissance missions. UAS and UAV utilization, however, has changed due to technological developments, and they are now used for commercial, industrial, and civilian applications. There are numerous uses for these systems, such as for surveying, mapping, monitoring, search and rescue, agriculture, protecting wildlife, and providing delivery services.

UAS and UAVs are being used in a variety of applications across numerous sectors, despite the difficulties. Drones are employed in the agricultural sector for pesticide application, crop monitoring, and mapping. Also, they are employed in wildlife conservation to keep track of animal populations and to spot and stop poaching. Drones are being considered for usage in the delivery sector to speed up deliveries and lower prices. Moreover, building sites and structures are being inspected by UAS and UAVs in the construction sector, eliminating the need for human inspection and improving safety. They are also employed in the oil and gas sector for pipeline surveillance and inspection. Drones are employed in the film and entertainment industries for aerial photography and cinematography.

The future of UAS and UAV technology is exciting and offers limitless possibilities. With advancements in technology and continued development, these systems will be increasingly deployed in a wide range of applications, making them an integral part of our

daily lives. However, the growth of this technology must be balanced with adequate regulations and guidelines to ensure the safe and responsible use of these systems. Overall, UAS and UAV technology offer significant benefits, and the potential for their continued growth and impact on various industries is vast.

Some UAV types may be more suited for a given application than others: fixed-wing vehicles are typically used when large areas need to be covered quickly and for a longer period of time; multirotor vehicles are best when dealing with indoor environments or needing hovering capability and more flexibility. (Figure 1.1).



(a) (b)

Figure 1.1: The Northrop Grumman MQ-4C Triton is an American high-altitude long endurance unmanned aerial vehicle (UAV) under development for the United States Navy as a surveillance aircraft[1], Onyxstar quadrotor from ESA(European Space Agency)(b)[2].

Multirotors are widely used in last years and the institutions have regularized the flight, like FAA for USA and EASA (European Union Aviation Safety Agency) for Europe; the aim is to set rules to control and differentiate the amateur flights with the professionals ones and to avoid incidents and improper use. EASA provides a classification of UAVs explained inf Figure 1.2.

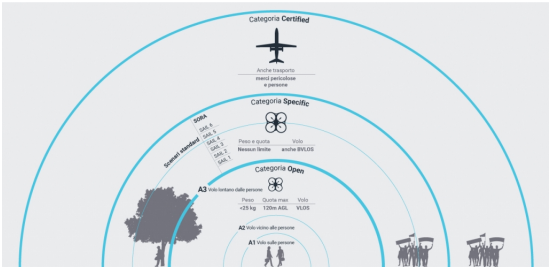


Figure 1.2: EASA drone classification: open, specific and certified.[3]

The U.S. Department of Defence (D.o.D.) also made a classification of UAVs in different groups shown in Figure 1.3.

Due to the easily access to drones for everybody, the operations of them should be regulated proportional to the risk of the operation. Because of the large different types of operations where a drone can be used, the EASA propose 3 categories: Open, Specified

Category	Size	Maximum Gross Takeoff Weight (MGTW) (lbs)	Normal Operating Altitude (ft)	Airspeed (knots)
Group 1	Small	0-20	<1,200 AGL*	<100
Group 2	Medium	21-55	<3,500	<250
Group 3	Large	<1320	<18,000 MSL**	<250
Group 4	Larger	>1320	<18,000 MSL	Any airspeed
Group 5	Largest	>1320	>18,000	Any airspeed

*AGL = Above Ground Level
**MSL = Mean Sea Level
Note: If the UAS has even one characteristic of the next level, it is classified in that level.

Figure 1.3: U.S. DoD Unmanned Aerial System group classification.[4]

and Certified.

The 'open' category addresses the lower-risk civil drone operations in , where safety is ensured provided the civil drone operator complies with the relevant requirements for its intended operation. This category is subdivided into three subcategories, namely A1, A2 and A3. Operational risks in the 'open' category are considered low and, therefore, no operational authorisation is required before starting a flight.

The 'specific' category covers riskier civil drone operations, where safety is ensured by the drone operator by obtaining an operational authorisation from the national competent authority before starting the operation. To obtain the operational authorisation, the drone operator is required to conduct a risk assessment, which will determine the requirements necessary for the safe operation of the civil drones.

In the 'certified' category, the safety risk is considerably high; therefore, the certification of the drone operator and its drone, as well as the licensing of the remote pilot(s), is always required to ensure safety.

The management of drone traffic will be ensured through the U-space: a set of services that will be deployed in airspace where heavier traffic is expected, such as in urban areas. The U-space Regulation establishes and harmonises the necessary requirements for manned and unmanned aircraft to operate safely in the U-space airspace, so as to prevent collisions between aircraft and to mitigate air and ground risks. The U-space regulatory framework will provide for safe aircraft operations in all areas and for all types of unmanned aircraft operations. The U-space Regulation was adopted in April 2021.[3]

1.2 Thesis organization

The thesis is organized as following. Chapter 2 covers the literature review about the mathematical modelling of a quadrotor describing all the dynamics involved on the model; it also presents the literature review about the different types of controllers used in the work. Chapter 3 presents the difference between Matlab/Simulink with respect to ROS/Gazebo and why it is needed a workflow between them. It exploits the characteristic of ROS and how it works. Chapter 4 describes the ROS repository created for the thesis work and explain the workflow to connect the Matlab design with ROS tools to have

the simulations. In the last two chapters simulations results are shown with conclusions about them and explanation, and at the end will be the future improvements to the work to increase the performance and the feasibility of the test bench environment.

Chapter 2

Literature review

2.1 Mathematical Model

This section decribes the Lagrange modelling of the quadrotor that includes aerodynamic and gyroscopic effects. The quadrotor under study is shown in Fig. 1. The quadrotor body is considered rigid and symmetric, with the arms aligned to x_b and y_b axes [5].

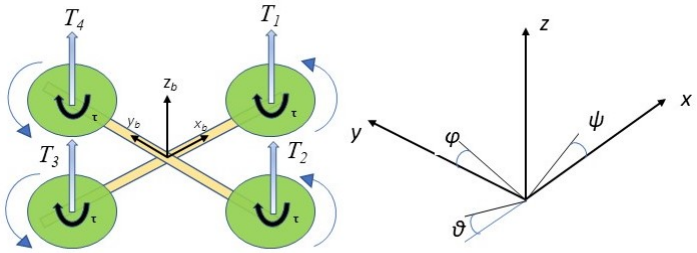


Figure 2.1: Quadrotor inertial and body farme.

The inertia matrix IB in the body frame is diagonal because of the symmetric structure and $I_x = I_y$.

$$\begin{bmatrix} Ix & 0 & 0 \\ 0 & Iy & 0 \\ 0 & 0 & Iz \end{bmatrix} \tag{2.1}$$

For the purpose of balancing the reaction torque the rotor induces and controlling the yaw angle, the rotor's angular velocities on the xb and yb axes have opposite signs ($\omega_{1,3} > 0$, $\omega_{2,4} < 0$). the center of mass (COM) is located at (0, 0, H_{COM})

$$h_{\mathrm{COM}} = \frac{(\frac{h_{\mathrm{b}}}{2} * m_{\mathrm{b}} + (\frac{h_{\mathrm{p}}}{2} + h_{\mathrm{b}}) * 4 * m_{\mathrm{p}}}{m} . \tag{2.2}$$

where h_b, m_b, h_p, m_p are the height and mass of the body and propellers respectively, and m the total mass of the quadrotor. The position in the inertial frame is expressed by the

vector ζ while the attitude, defined as Euler angles coordinates in the inertial frame, is represented by the vector η. Position and attitude are then grouped in the vector $q \in R^6$.

$$\xi = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \eta = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}, q = \begin{bmatrix} \xi \\ \eta \end{bmatrix} \tag{2.3}$$

In the body frame, the linear and the angular velocities are expressed by the vectors V_b and ν respectively

$$V_{\mathrm{B}} = \begin{bmatrix} v_{\mathrm{x,B}} \\ v_{\mathrm{y,B}} \\ v_{\mathrm{z,B}} \end{bmatrix}, \nu = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{2.4}$$

To compute the rotation matrix from the body frame and the inertial frame it is used the 3-2-1 Euler rotation matrix, where $s(x) = sin(x)$, $c(x) = cos(x)$ and $t(x) = tan(x)$. The rotation matrix is obtained from the composition of Roll-Pitch-Yaw rotations; the sequence is:

- z-rotation for roll angle:

$$R_z(\psi) = \begin{bmatrix} c(\phi) & -s(\phi) & 0 \\ s(\phi) & c(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

- y-rotation for pitch angle:

$$R_y(\theta) = \begin{bmatrix} c(\theta) & 0 & s(\theta) \\ 0 & 1 & 0 \\ -s(\theta) & 0 & c(\theta) \end{bmatrix} \tag{2.6}$$

- x-rotation for yaw angle:

$$R_z(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c(\psi) & -s(\psi) \\ 0 & s(\psi) & c(\psi) \end{bmatrix} \tag{2.7}$$

Combining the three rotations as $R_{321} = R_z(\phi)R_y(\theta)R_x(\psi)$

$$R_{321} = \begin{bmatrix} c(\theta)c(\psi) & s(\phi)s(\theta)c(\psi) - c(\phi)s(\psi) & c(\phi)s(\theta)c(\psi) + s(\phi)s(\psi) \\ c(\theta)c(\psi) & s(\phi)s(\theta)s(\psi) + c(\phi)c(\psi) & c(\phi)s(\theta)c(\psi) - s(\phi)c(\psi) \\ -s(\theta) & s(\phi)c(\theta) & c(\phi)c(\theta) \end{bmatrix} \tag{2.8}$$

Because rotation matrices have the orthogonality property, the rotation from the inertial

frame to the body frame is supplied by the transpose of R_{321}.

$$\begin{bmatrix}\dot{\phi}\\ \dot{\theta}\\ \dot{\psi}\end{bmatrix} = \underbrace{\begin{bmatrix}1 & s(\phi)t(\theta) & c(\phi)t(\theta)\\ 0 & c(\phi) & -s(\phi)\\ 0 & s(\phi)/c(\theta) & c(\phi)/c(\theta)\end{bmatrix}}_{W^{-1}} \begin{bmatrix}p\\ q\\ r\end{bmatrix},$$
$$\begin{bmatrix}p\\ q\\ r\end{bmatrix} = \underbrace{\begin{bmatrix}1 & 0 & -s(\theta)\\ 0 & c(\phi) & c(\theta)s(\phi)\\ 0 & -s(\phi) & c(\phi)c(\theta)\end{bmatrix}}_{W} \begin{bmatrix}\dot{\phi}\\ \dot{\theta}\\ \dot{\psi}\end{bmatrix} \tag{2.9}$$

W is invertible if $\theta \neq (2k-1), (k \in Z)$. (i.e. W is not singular). Each rotor produces a force T_i in the direction of the rotor z-axis and proportional to the square of the rotor velocity w_i and to the lift constant k [6].

$$T_{\mathrm{i}} = k\omega_{\mathrm{i}}^2 \tag{2.10}$$

The rotor angular velocities and acceleration also produce the torque τ_{Ri} around the rotor axis with b and I_R being the drag constant and the inertia moment of the rotor respectively.

$$\tau_{\mathrm{Ri}} = b\omega_{\mathrm{i}}^2 + I_{\mathrm{R}}\dot{\omega}_{\mathrm{i}} \tag{2.11}$$

For low speed maneuvers, it is common practice to consider k and b as constants despite their dependence on air density, propeller shape and angle of attack [7]. Combining all the rotor's vertical force contributions, it is obtained the total thrust along the z-axis

$$T_{\mathrm{B}} = \begin{bmatrix}0\\ 0\\ T\end{bmatrix}, T = \sum_{i=1}^{4} T_{\mathrm{i}} \tag{2.12}$$

The torque vector τ_B represent the torques in the direction of the respective body frame angles

$$\tau_{\mathrm{B}} = \begin{bmatrix}\tau_\phi\\ \tau_\theta\\ \tau_\psi\end{bmatrix} = \begin{bmatrix} lk(\omega_4^2 - \omega_2^2)\\ lk(\omega_3^2 - \omega_1^2)\\ \tau_{R1} - \tau_{R2} + \tau_{R3} - \tau_{R4}\end{bmatrix}, \tag{2.13}$$

where l is the lenght of the quadrotor arm. As a first approximation, the effect of $\dot{\omega}_i$ in τ_ψ is considered to be null.

Euler-Lagrange equations

The quadrotor dynamics may be described by the equation of motion expressed in the following form [8]:

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + g = \zeta \tag{2.14}$$

where:

- $B(q) \in \mathbb{R}^{6x6}$ is the inertia matrix

- $C(q,\dot{q}) \in \mathbb{R}^{6x6}$ is the Christoffel matrix accounting for centrifugal and Coriolis effect
- $g \in \mathbb{R}^6$ is the vector accounting for the gravitational contribution
- $\zeta \in \mathbb{R}^6$ is the vector of forces in the inertial reference frame and torques acting on the UAV.

Equation (14) can be systematically determined using the Lagrangian formulation. The total of the translational and rotational kinetic energies $\mathcal{T}$ minus the potential energy $\mathcal{U}$ is the mechanical system's Lagrangian.

$$\mathcal{L} = \mathcal{T} - \mathcal{U}, \tag{2.15}$$

which leads to

$$\mathcal{L}(q,\dot{q}) = \frac{1}{2} m \dot{\xi}^T \dot{\xi} + \frac{1}{2} \dot{\eta}^T J \dot{\eta} - m g_z z, \tag{2.16}$$

where $J = W^T I_B W$ is the Jacobian matrix expressing the moment of inertia in the inertial reference frame, and g_z is the gravitational acceleration. The Lagrange equation is given by

$$\frac{d}{dt}\left(\frac{\vartheta \mathcal{L}}{\vartheta \dot{q}}\right)^T - \left(\frac{\vartheta \mathcal{L}}{\vartheta q}\right)^T = \zeta, \tag{2.17}$$

and sets the relationship between the generalized forces applied to the quadrotor ζ and the position q, velocities $\dot{q}$ and accelerations $\ddot{q}$[8], which is rewritten as:

$$\begin{bmatrix} f \\ \tau \end{bmatrix} = \frac{d}{dt}\left(\frac{\vartheta \mathcal{L}}{\vartheta \dot{q}_i}\right)^T - \left(\frac{\vartheta \mathcal{L}}{\vartheta q_i}\right)^T, \quad i = 1, ...,6 \tag{2.18}$$

where the external forces and torques acting on the UAV are:

$$f = R T_B, \quad \tau = \tau_B. \tag{2.19}$$

The equation of motion is derived solving (18) and it contains the following matrices:

$$B = \begin{bmatrix} M & 0 \\ 0 & J \end{bmatrix}, \quad C = \begin{bmatrix} 0_{3x3} & 0 \\ 0 & C_\tau \end{bmatrix}, \quad g = \begin{bmatrix} 0 & 0 & m g_z & 0 & 0 & 0 \end{bmatrix}^T \tag{2.20}$$

where $B, C \in \mathbb{R}^{6x6}$, $M = m * I_{3x3}$ is the mass matrix, and $C_t(\eta,\dot{\eta}) = \dot{J} - \frac{1}{2}\frac{\vartheta}{\vartheta \eta}(\dot{\eta}^T J)$ is the matrix that take into account the centrifugal and Coriolis effect produced by the angular velocity.

$$C(\eta,\dot{\eta}) = \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} \tag{2.21}$$

$C_{11} = 0$
$C_{12} = (I_y - I_z)(\dot{\theta} C_\phi S_\phi + \dot{\psi} S_\phi^2 C_\theta) + (I_z - I_y)\dot{\psi} C_\phi^2 C_\theta - I_x \dot{\psi} C_\theta$
$C_{13} = (I_z - I_y)\dot{\psi} C_\phi S_\phi C_\theta^2$
$C_{21} = (I_z - I_y)(\dot{\theta} C_\phi S_\phi + \dot{\psi} S_\phi^2 C_\theta) + (I_y - I_z)\dot{\psi} C_\phi^2 C_\theta - I_x \dot{\psi} C_\theta$

$$
\begin{aligned}
C_{22} &= (I_z - I_y)\dot{\phi}C_\phi S_\phi \\
C_{23} &= -I_x\dot{\psi}S_\theta C_\theta + I_y\dot{\psi}S_\phi^2 C_\theta S_\theta + I_z\dot{\psi}C_\phi^2 C_\theta S_\theta \\
C_{31} &= (I_y - I_z)\dot{\psi}C_\theta^2 C_\phi S_\phi - I_x\dot{\theta}C_\theta \\
C_{32} &= (I_z - I_y)(\dot{\theta}C_\phi S_\phi S_\theta) + (I_y - I_z)\dot{\phi}C_\phi^2 C_\theta + I_x\dot{\psi}S_\theta C_\theta - I_y\dot{\psi}S_\phi^2 C_\theta S_\theta - I_z\dot{\psi}C_\phi^2 C_\theta S_\theta \\
C_{33} &= (I_y - I_z)\dot{\phi}C_\phi S_\phi C_\theta^2 - I_y\dot{\theta}S_\phi^2 C_\theta S_\theta - I_z\dot{\theta}C_\phi^2 C_\theta S_\theta + I_x\dot{\theta}S_\theta C_\theta
\end{aligned}
$$

To further improve model accuracy, the gyroscopic effect and the aerodynamic drag are considered and included in the model[5].

Gyroscopic Effect

The Gyroscopic effect is modeled as in [7]

$$
\begin{aligned}
G_{gyro} &= \sum_{i=1}^{4} S(\nu)_x I_r \omega_i \mathbf{e_3}, \\
S(\nu) &= \begin{bmatrix} 0 & -r & q \\ r & 0 & -p \\ -q & p & 0 \end{bmatrix},
\end{aligned}
\tag{2.22}
$$

where $S(\nu)_x$ is the skew symmetric matrix with ν components. Factoring out the vector position and Euler angle derivatives as

$$
G_{gyro} = G_a(q, \dot{q}, \omega)\dot{q} \tag{2.23}
$$

, it can be rewritten equation (22), where

$$
G_a^T = \begin{bmatrix} 0_{3x3} & 0 & 0 & 0 \\ 0 & 0 & -I_{rz}(\sum_{i=1}^{4}\omega_i) & 0 \\ 0 & I_{rz}c_\phi(\sum_{i=1}^{4}\omega_i) & 0 & 0 \\ 0 & I_{rz}c_\theta s_\phi(\sum_{i=1}^{4}\omega_i) & I_{rz}s_\theta(\sum_{i=1}^{4}\omega_i) & 0 \end{bmatrix}_{6x6}. \tag{2.24}
$$

Aerodynamic drag

The aerodynamic forces acting on the quadrotor are modelled as [9], and expressed in the matrix form to adapt to the Lagrange formulation. The drag forces responsible for the multiple aerodynamic effects are:

- Blade flapping is a phenomenon that occurs when rotor blades are into translational motion. A drag force is applied at the propeller COM as a result of this effect opposing the UAV's motion. Considering a small advance ratio $\mu = \frac{|v_p|}{\omega r}$, where V_p and ωr are the horizontal velocity respectively, the blade flapping drag is described as

$$
D_{flap,i} = T_i(A_{flap}\frac{V_{pi}}{\omega_i} + B_{flap}\frac{\nu}{\omega_i}), \tag{2.25}
$$

where

$$A_{flap} = \frac{1}{r}\begin{bmatrix} -A_{1c} & A_{1s} & 0 \\ -A_{1s} & -A_{1c} & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{2.26}$$

and

$$B_{flap} = \begin{bmatrix} -B_2 & B_1 & 0 \\ B_1 & -B_2 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{2.27}$$

are the parameter matrices that must be identified from flight testing.

- Because of the rotor's thrust imbalance, which causes the advancing blade to produce more lift than the retreating one, induced drag is generated. This effect opposes the motion in the direction of the apparent wind and is most common in tiny quadrotor UAVs because of the relatively rigid rotor blades. The induced drag is therefore modelled as a drag force applied a the propeller COM

$$D_{I,i} = K_I V_{pi}. \tag{2.28}$$

- Similar to how airfoils experience induced drag associated with an increase in lift, translational drag is produced when the rotor moves in forward flight. The force model varies depending on the translational speed of the rotor

$$D_{T,i} = \begin{cases} K_{T_1} V_{pi} & V_{pi} < \bar{V}_{pi} \\ K_{T_2}(V_{z,i} - v_i)^4 V_{pi} & V_{pi} > \bar{V}_{pi} \end{cases}, \tag{2.29}$$

where $V_{z,i}$ is the rotor z-axis linear velocity expressed in the body frame, v_i is the vertical velocity induced through the rotor, and $\bar{V}_{pi}$ is the threshold velocity which depends on the propeller. Only the first functional relationship is taken into account because this force is applied to the rotor COM and the threshold velocity is not exceeded for simplicity's sake.

- Profile Drag is due to the transverse velocity on the propeller moving in to the air. This effect is absent when hovering, but it increases drag when moving along the body's x, y-plane. The following linear model is a simplified variation of the complete nonlinear one derived from Blade Element Theory[9] and is applied at the propeller COM

$$D_{P,i} = K_P V_{pi}. \tag{2.30}$$

- The resistance created by the UAV's flying envelope is called parasitic drag. For small scale quadrotor travelling at less than 10 m/s, it is typically ignored. The parasitic drag is described as

$$\begin{aligned} D_{par} &= K_{par}|V_B|V_B, \\ K_{par} &= \frac{1}{2}\rho S C_{D_{par}} \end{aligned} \tag{2.31}$$

where ρ is the air density, S is the quadrotor lateral area and C_{Dpar} is drag coefficient. The resulting force is applied to the quadrotor COM.

The drag coefficients K_I, K_{T_1}, K_{T_2}, K_P depend on the shape of propellers and need to be empirically estimated. The sum of the aerodynamic forces acting on the propeller COM are grouped in the vector

$$D_i = D_{flap,i} + D_{I,i} + D_{T_1,i} + D_{P,i} \tag{2.32}$$

and the torques induced on the quadrotor COM are described as

$$\begin{aligned} \tau_{D_1} &= S(D_1)_x(d\mathbf{e_1} + h\mathbf{e_3}), \\ \tau_{D_2} &= S(D_2)_x(-d\mathbf{e_2} + h\mathbf{e_3}), \\ \tau_{D_3} &= S(D_3)_x(-d\mathbf{e_1} + h\mathbf{e_3}), \\ \tau_{D_4} &= S(D_4)_x(d\mathbf{e_2} + h\mathbf{e_3}). \end{aligned} \tag{2.33}$$

The total aerodynamic forces and torques are modelled as

$$D_b = D_{par} + \sum_{i=1}^{4} D_{flap,i} + D_{I,i} + D_{T_1,i} + D_{P,i}, \tag{2.34}$$

$$\tau_D = \sum_{i=1}^{4} \tau_{D_1} \tag{2.35}$$

The following matrix form represents the drag forces and torques after factoring out the position and attitude derivative vector $\dot{q}$

$$\begin{bmatrix} D_b \\ \tau_D \end{bmatrix} = F_{flap}(q,\dot{q},\omega)\dot{q} + F_I(q,\dot{q})\dot{q} + F_T(q,\dot{q})\dot{q} + F_p(q,\dot{q})\dot{q} + F_{par}(q,\dot{q},|\dot{q}|)\dot{q}, \tag{2.36}$$

$F_{flap}, F_I, F_T, F_P, F_{par} \in \mathbb{R}^{6x6}$.

Dynamic Equations of Motion

The gyroscopic effect and the aerodynamic forces are added to (11). Therefore, the complete model becomes

$$B(q)\ddot{q}+C(q,\dot{q})\dot{q}+g+G_a(q,\dot{q},\omega)\dot{q}+F_{flap}(q,\dot{q},\omega)\dot{q}+F_I(q,\dot{q})\dot{q}+F_T(q,\dot{q})\dot{q}+F_p(q,\dot{q})\dot{q}+F_{par}(q,\dot{q},|\dot{q}|)\dot{q} = \zeta. \tag{2.37}$$

Grouping the aerodynamic effect matrices into a single F, the model can be simplified as

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + g + G_a(q,\dot{q},\omega)\dot{q} + F(q,\dot{q},|\dot{q}|,\omega)\dot{q} = \zeta(\omega). \tag{2.38}$$

2.2 Control Problem

Similar to the majority of highly nonlinear systems, the dynamics of unmanned aerial vehicles (UAV) have a number of features that make it challenging to design trajectory tracking or stabilization.

Unknown nonlinearities, underactuation, a tight coupling of subsystems, parametric and nonparametric model uncertainties, measurement noise, output disruptions, and system failure are some of these traits. The six degrees of freedom that quadrotors have, which is more than the number of independent control inputs, is an example of an underactuated system. This underactuation consequently reduces the number of system configurations that may be directly controlled. Considering $\Omega \in \mathbb{R}^4$ as the input, which correspond to the square of the rotor angular velocity

$$\Omega = \begin{bmatrix} \omega_1^2 & \frac{2}{2} & \omega_3^2 & \omega_4^2 \end{bmatrix}^T, \tag{2.39}$$

the equation of motion (34) cabe rewritten as

$$B\ddot{q} + C\dot{q} + g + G_a\dot{q} + F\dot{q} = \zeta = K_\zeta \Omega, \tag{2.40}$$

where $K_\zeta \in \mathbb{R}^{6x4}$ is the matrix which establish the relation between forces, torques and the rotor velocities as

$$\begin{bmatrix} T_B \\ \tau_B \end{bmatrix} = K'_\zeta \Omega, \tag{2.41}$$

and since $f = R_{321}T_B, \tau = \tau_B$, it is obtained

$$K_\zeta = \begin{bmatrix} R_{321} & 0 \\ 0 & I \end{bmatrix} K'_\zeta. \tag{2.42}$$

Since $rank(K_\zeta) = 4$, the quadrotor is an underactuated system.[10] A solution for this type of control problem can be found using a hierarchical control approach which consist in two different loop control, one for the position control (outer loop) and one for the attitude tracking (inner loop); both of them are controlled using PID technique. The outer loop control action is designed following [7], where the position closed loop equation for the position is

$$\ddot{\xi} = \ddot{\xi}_d + K_{P_p}(\xi_d - \xi) + k_{D_p}(\dot{\xi}_d - \dot{\xi}). \tag{2.43}$$

Taking the control vector $U = \ddot{\xi} = (U_1, U_2, U_3)^T$, through (14), it follows

$$U = -g_z\mathbf{e_3} + \frac{T}{m}R\mathbf{e_3}, \tag{2.44}$$

resulting to

$$R^T(U + g_z\mathbf{e_3}) = \frac{T}{m}\mathbf{e_3}, \tag{2.45}$$

which, as stated in [7] by using some mathematical manipulation, it allows to compute the desired pitch and roll attitude trajectory

$$\theta_{c_1} = arctan\left(\frac{U_1 cos\psi + U_2 sin\psi}{U_3 + g_z}\right), \tag{2.46}$$

$$\phi_{c_1} = arcsin\left(\frac{U_1 sin\psi + U_2 cos\psi}{\sqrt{U_1^2 + U_2^2 + (U_3 + g_z)^2}}\right), \tag{2.47}$$

which are the attitude-commanded inputs for the inner loop. From (45) it can be also derived the total thrust that is generated from the fours rotors as a function of the attitude and the control

$$T_{c_1} = m[U_1(sin\theta cos\psi cos\phi + sin\psi sin\phi) + U_2(sin\theta sin\psi cos\phi - cos\psi sin\phi) + (U_3 + g_z)cos\theta cos\phi]. \tag{2.48}$$

With the use of a PD controller strengthened by feedback linearization, the inner loop regulates the torques τ. Following [8] the control action is formulated as

$$\tau_{c_1} = J(\eta)y + C_\tau(\eta, \dot{\eta})\dot{\eta}, \tag{2.49}$$

where the virtual control vector $y = \ddot{\eta}$ is

$$y = K_{P_a}(\eta_d - \eta) + K_{D_a}(\dot{\eta_d} - \dot{\eta}). \tag{2.50}$$

The controlled rotor squared velocities are determined after computing the appropriate thrust and torques by

$$\begin{bmatrix} T \\ \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} k(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \\ lk(\omega_4^2 - \omega_2^2) \\ lk(\omega_3^2 - \omega_1^2) \\ b(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) \end{bmatrix} = \begin{bmatrix} k & k & k & k \\ 0 & -kl & 0 & kl \\ -kl & 0 & kl & 0 \\ b & -b & b & -b \end{bmatrix} \begin{bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{bmatrix}. \tag{2.51}$$

To have a more realistic mathematical model of the quadrotor, it is added also the gyroscopic and aerodynamics effects. Equation (45) becomes

$$R^T(U + g_z\mathbf{e_3} + \frac{1}{m}F(q, \dot{q}, |\dot{q}|, \omega)\dot{q}) = \frac{T}{m}\mathbf{e_3}, \tag{2.52}$$

where the controlled roll and pitch angles result

$$\theta_{c_2} = arctan\Big(\frac{U_1^* cos\psi + U_2^* sin\psi}{U_3^* + g_z}\Big), \tag{2.53}$$

$$\phi_{c_2} = arcsin\Big(\frac{U_1^* sin\psi + U_2^* cos\psi}{\sqrt{U_1^{*^2} + U_2^{*^2} + (U_3^* + g_z)^2}}\Big), \tag{2.54}$$

and the controlled thrust

$$T_{c_2} = m[U_1^*(sin\theta cos\psi cos\phi + sin\psi sin\phi) + U_2^*(sin\theta sin\psi cos\phi - cos\psi sin\phi) + (U_3^* + g_z)cos\theta cos\phi], \tag{2.55}$$

where

$$\begin{aligned} U_i^* &= U_i + F_i^*, \\ F_i^* &= \frac{1}{m}\mathbf{e_i}^T F(q, \dot{q}, |\ddot{q}|, \omega)\ddot{q}. \end{aligned} \tag{2.56}$$

The feedback linearization accounting for the torque caused by the aerodynamic drag is

$$\tau_{c_1} = J(\eta)y + C_\tau(\eta, \dot{\eta})\dot{\eta} + G_{a_\tau}(\eta, \dot{\eta}, \omega)\dot{\eta} + F_\tau(\eta, \dot{\eta}, |\dot{\eta}|, \omega)\dot{\eta}. \tag{2.57}$$

As a result, a cascade PD controller with feedback linearization is used as the control strategy.

Chapter 3

Problem Statement

Advanced robotic simulation tools like ROS and Gazebo are required to provide a real-time and improved modeling of the controller design. These tools enable the creation of a real-time 3D experiment environment where many scenarios and missions may be evaluated; however, because of their not straightforward use, in a preliminary testing phase, it may be challenging and consuming time to implement controller designs using C++ or Python. For this reason the workflow between Matlab/Simulink and ROS/Gazebo can be very powerful; Simulink is a tool that allow to design faster and easier a controller and with the ROS toolbox provided by Mathworks it can be done the connection with ROS. This chapter describes the functionality of each environment to have an overview on them, focusing on ROS/Gazebo ones.

3.1 Software Environment

3.1.1 ROS

Robotic Operating System (ROS) is an open-source, meta-operating system that allows robot simulations with hardware abstraction, low-level control, messages between nodes and repository management[11].

ROS can only run on Unix-based platforms like Ubuntu and it is released regularly with ROS distribution; distribution are collections of versioned stacks that can you install. The one used in this work is ROS Noetic.

ROS provides tools and libraries needed to create, build and run codes (c++ or Python) that can run on more devices and can communicate with them. In this operating system there are different types of communication such as ROS services and ROS topics; the execution processes are called nodes. Nodes have the capability of communication between them and they are managed by a master; the master is a node that manage and give all the information needed to the nodes to work.[12]

Figure 3.1: ROS logo.[13]

ROS topic, publisher and subscriber

The publish-subscribe method ensures complete autonomy between the sources who generate the information, in this case, known as publishers, and those who receive it, known as subscribers. The only term that unites the two parties is topic. The functionality calls for one or more providers to distribute information on a specific subject via messages, and recipients who are interested in receiving it to subscribe to that same topic. All of this occurs without any logical interaction between the two parties. This method was made possible by the nodes, which will be described in the sections that follow.

Package

ROS's internal software is organized into packages. A package may contain the code for one or more nodes, ROS message definitions, support classes for communication and transport, test classes, etc.; however, in addition to these roughly related components of the ROS architecture, it is also possible to insert configuration files, third-party software products, and anything else that could possibly constitute a logical module on its own. For this reason, the package is regarded as the smallest (atomic) item capable of being assembled independently within the ROS environment. Its goal is to provide its own functionalities in the most straightforward manner possible so that the software it contains can be easily reused. A package should typically include enough internal functionality to be considered useful, but not so much that it becomes burdensome or difficult for other applications to use. It is possible to construct a package using either manual labor to create the necessary files or appropriate tools, such as the catkin create pkg command.[14]

Node

The system's internal processes are identified by the nodes. ROS was designed to be as modular as possible in all respects, including the execution of the most basic operations. In fact, a robotic control system only consists of various units that can be distinguished not only from a physical level but also from a logical one. Every node must be capable of functioning independently in order to guarantee a certain level of robustness in terms of tolerance for potential updates, bugs, or attacks from other elements. Also, the introduction of higher levels of modularity has the benefit of reducing the complexity of the code in comparison to analogous monolithic systems because the implementation details

of one module are not necessary for the other elements. A node during communication may take on one or more roles beyond those anticipated, hence it is possible for it to be both publisher and subscriber at once depending on the task assigned to each role. The most popular way to launch a ROS node is by using the command *rosrun*, which is followed by the name of the package from which the node is derived and then by the name of the node's executable. Additionally, it is possible to specify any command-line arguments that might be used during the configuration phase. As is frequently assumed, within a single machine, more nodes may be present. These nodes may be of different types and open various roles, but it is also possible to have many nodes of the same kind operating simultaneously.

ROS messages

Each node within the ROS architecture communicates with the others by distributing information on related topics and employing data structures with various fields that vary depending on the situation and take the form of messages. Every message can be defined based on the conventional types: int, float, boolean, string, etc.; however, it is also possible to define arrays. Finally, it is possible to create a more complex or simpler structure by defining new message types while still using existing message fields. The structure of each messages is defined within files with the extension ".msg" within the subdirectory "msg" that will be created within the package. As expected, a message may occasionally contain inside itself additional ROS messages in the form of fields. In particular, the field:header, which gets its name from the same type and ought to be present in all ROS messages. Certain common information is contained within it, such as the sequence number, timestamp, and frame-ID; each of these can be set manually by the programmer or automatically by the client libraries during the publishing phase.

Master

Its purpose is to enable the other system nodes to locate one another and then carry out effective communication among themselves. As previously mentioned, the ROS architecture implements a system based on publish-subscribe, which provides that the parties involved in the communication are not aware of who would be sending the data or receiving it in advance. In actual practice, what happens is that each node of the system, when it is being activated, has to know one other address in addition to the one for its own machine, i.e. the Master's. When a node wants to publish data in a message-based format on a particular topic, the only action that needs to be taken is to inform the master. The master will need to know, among other things, the address and port of the node that initiated the request, the topic's name, and the type of message. Later, if another node is interested in receiving the data published on that topic, the only information that needs to be provided is the topic's name.

Catkin

The official build system for ROS is called Catkin, and it is required for the compilation and linking of the many pieces of the code that make up the system. The decision

to use a personalized build system in the context of ROS was made due to the need to manage a system that is significantly larger and more complex than those used in traditional environments. This system includes multiple programming languages and systems, all of which have its own rules for compilation and unique dependencies that would be challenging to manage using other tools. Catkin uses the Python scripts and the CMake software's fixed-size macros to provide several features that go beyond this last one's typical operation. The goal was to create a tool that was more compatible than its predecessor, rosbuild, both in terms of structure and use, enabling better project organization in addition to better support for cross-compiling and portable code. As previously stated, Catkin's functionality is quite similar to CMake's, to which support for automatic package identification inside the workspace and the simultaneous compilation of multiple projects with reciprocal dependencies has been added. In general, a build system is in charge of producing "target" objects that are created from source code. After that, any final user will be able to use these last ones directly. Targets can be scripts, header files, libraries, executables, or anything else that isn't static code. As previously stated, in the terminology of ROS, the source code is organized under "packages," and each of these often consists of one or more targets that must be created during the build process. The group of packages that Catkin is now considering during this procedure are connected to the same workspace. Hence, Catkin enables simultaneous generation of more targets belonging to several Packages while respecting their mutual dependencies. To be able to do this, just like any other build system, it is necessary to know a number of details, such as where the compilation tools are located (for example, the location of the C++ compiler), where the code source's file system is located, what the various dependencies are called and where they are stored, and where the executables will be generated and installed. These details are often contained in a configuration file that must be accessible by the build system; in the context of CMake and consequently also of Catkin, these details are specified inside a file called "CMakeList.txt." In addition to a file of this type designated at the workspace level, each package has a unique configuration file bearing the same name that is located inside its own root directory.

Creation of new package

Use the catkin create pkg command to create a new package, followed by the name you want to assign and any required dependencies. All requested files and order forms, including CMakelist.txt, will be generated automatically. This final one will have a default configuration that will then be customized based on user input. In addition to this file, a manifest called "package.xml" is generated during the package creation phase. It is always located inside the root directory and contains descriptions of all the package's properties, including its name, version, author, and dependencies on other packages; if these were to be incorrect or incomplete, the program could still be compiled on its own computer, provided it has all the necessary software. The code for creating nodes, generating messages, or both may be contained in the various packages. Typically, the node code will be placed in the src folder (generated automatically), while the message and service files (.msg and.srv) will be placed inside the msg folder, which must be manually created. In addition to "msg" it is also possible to manually include other folders, such

as "launch" (which will contain the files with the previously described extensions) and "config," which may contain one or more files in the ".yaml" format for configuring the ingress parameters related to the various nodes.

3.1.2 Gazebo

Gazebo is an open-source robot 3D simulation tool that provides a realistic simulation environment for robotics and automation applications. It is widely used in the robotics industry to test and validate robot designs and control algorithms before deploying them in the real world.[15]

One of the key components of Gazebo is its physics engine(ODE), which is responsible for simulating the physical behavior of objects in the virtual world. The physics engine calculates the motion and interaction of objects based on their physical properties, such as mass, inertia, and friction. This allows Gazebo to provide a realistic simulation environment where robots can interact with objects and their surroundings just like they would in the real world.

Plugins are another important component of Gazebo. They are used to extend the functionality of Gazebo and provide additional features and capabilities. Plugins can be used to add new sensors or actuators to a robot, to modify the behavior of the physics engine, or to implement custom controllers for the robot.

Gazebo works by using a combination of models, sensors, controllers, and plugins to create a virtual world where robots can operate. The user can create a robot model using a 3D modeling tool and import it into Gazebo. The robot model is then equipped with sensors and actuators, such as cameras, lidars, and motors, which are controlled by custom controllers implemented using plugins.

The simulation can be visualized using the Gazebo GUI, which allows the user to interact with the simulation in real-time, controlling the robot and observing its behavior.

In summary, Gazebo is a powerful simulation tool that provides a realistic virtual environment for testing and validating robot designs and control algorithms. Its physics engine, plugins, and models work together to create a realistic simulation environment where robots can interact with objects and their surroundings just like they would in the real world.

3.1.3 Matlab/Simulink

Matlab and Simulink are two software tools developed by MathWorks for numerical computing and simulation, respectively.

Matlab is a high-level programming language and numerical computing environment that is used by engineers, scientists, and mathematicians to solve complex problems in a variety of domains, such as signal processing, control systems, image processing, and machine learning. It provides a rich set of functions and tools for data analysis, visualization, and modeling, making it a popular choice for scientific computing.[17]

Simulink, on the other hand, is a graphical simulation tool that is built on top of Matlab. It allows users to model and simulate complex dynamic systems, such as control systems, power systems, and communication systems, using a block diagram environment.

Figure 3.2: Gazebo logo.[16]

Simulink provides a library of pre-built blocks that can be used to construct models of systems, and also allows users to create custom blocks to extend the functionality of the tool. It provides a comprehensive set of tools for analyzing and optimizing system performance, as well as tools for generating code for embedded systems.[17]

One of the key benefits of using Matlab and Simulink is their integration with each other. Simulink models can be easily integrated with Matlab scripts, allowing users to leverage the power of Matlab for data analysis and visualization. Matlab code can also be used in Simulink models, making it easier to incorporate custom algorithms and functionality.

Figure 3.3: MathWorks logo.[18]

These features are very useful for this project purpose, allowing to design and test the controller in an user friendly environment and, only after, deploy the code in the ROS environment as C++ code.

Chapter 4

ROS package

This chapter presents the ROS package created by the author. In the package are codes to describe the quadrotor model, files to launch and manage the simulation world in Gazebo, and plugins that go into describing the dynamics of the model. The tutorials [19], [20], [21], [22], and the simulator presented in 2016, RotorS [23], were taken as reference for creating the package. All the complete codes are in Appendix A.

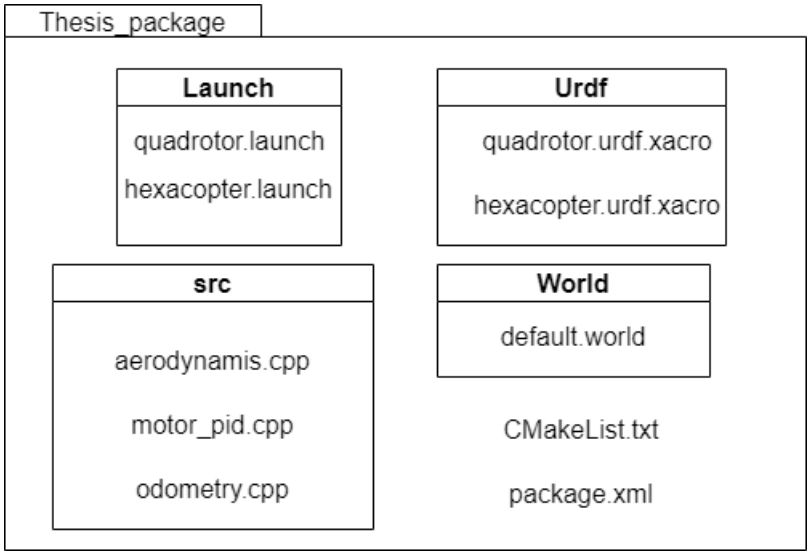


Figure 4.1: Thesis package with sub-folders and files.

4.1 URDF

The robot's model needs to be able to be understood by ROS and Gazebo to run the simulation. The URDF is an XML format which is used to describe the robot model;

ROS can parse the URDF through a C++ parser contained in the "URDF" package [24]. The format used is Xacro which is a XML macro language that support the use of macros to create XML elements that are shorter and more understandable[20]. The parameters for the robot creation are the ones used in the mathematical model of the Chapter 2 and by doing so the generate model in Gazebo will work as the mathematical one.

4.1.1 Code Explanation

It is necessary to specify a namespace for the URDF file in order for it to be properly parsed.[24]

```
<?xml version="1.4"?>
<robot name="quadrotor"
      xmlns:xacro="http://www.ros.org/wiki/xacro">
```

The quadrotor model is created connecting different links with the use of joints. URDF form allows to describe the main robot objects, links and joints, using tag. The "joint" and "link" tags enable the specification of robot attributes like mass, inertia and spawn position and pose for each link and joint. The base link of the quadrotor is a cylinder shape link with a mass and inertia and describe the base frame of the model; the other links are the propellers which are attached to the base link using joints. a joint can be of four different types:

- fixed: don't allow movements in any direction.
- revolute: allow rotation along a specified axis (x, y, z) with upper and lower limits.
- continuous: allow rotation along axis without upper and lower limits.
- prismatic: allow sliding movements along axis.
- floating: allow movements for all 6 degrees of freedom.
- planar: allow movements in a plane perpendicular to the axis.

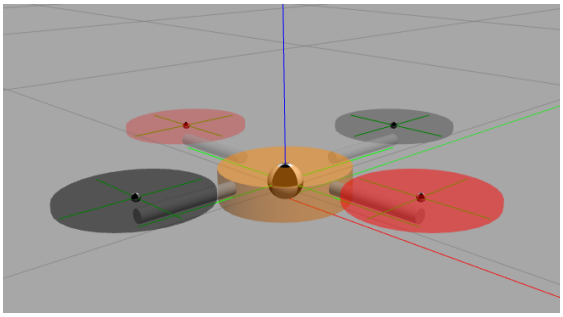
In this case the joints are of type continuous and the axis specified is the z-axis. Below the resulting lines of code for the base link, one of the propeller link and one of the joint.

```
<link name="frame">
 <visual>
   <origin xyz="0 0 0" rpy="0 0 0"/>
   <geometry>
     <cylinder radius="${frame_radius}" length="${frame_height}"/>
   </geometry>
   <material name="frame_material">
     <color rgba="0.8 0.8 0.8 1.0"/>
```

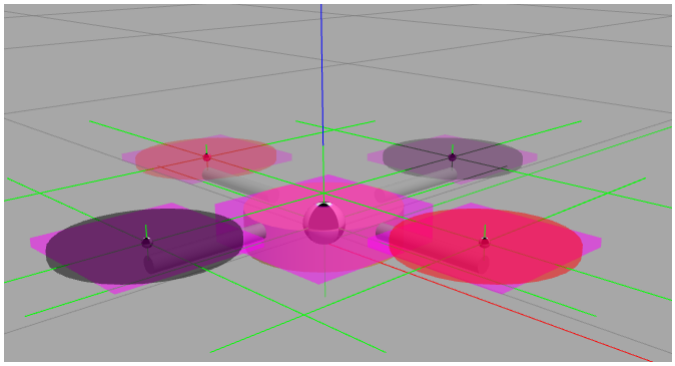
```
    </material>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="${frame_radius}" length="${frame_height}"/>
    </geometry>
  </collision>
  <xacro:cylinder_inertial radius="${frame_radius}" height="${frame_height}"
      mass="${frame_mass}">
    <origin xyz="0 0 0" rpy="0 0 0" />
  </xacro:cylinder_inertial>
</link>

<link name="propeller">
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <cylinder radius="${propeller_radius}" length="${propeller_height}"/>
      </geometry>
      <material name="propeller_material"/>
    </visual>
    <collision>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
        <cylinder radius="${propeller_radius}" length="${propeller_height}"/>
      </geometry>
    </collision>
    <xacro:cylinder_inertial radius="${propeller_radius}"
        height="${propeller_height}" mass="${propeller_mass}">
      <origin xyz="0 0 0" rpy="0 0 0"/>
    </xacro:cylinder_inertial>
  </link>

  <joint name="arm${i}_propeller${i}" type="continuous">
    <parent link="arm${i}"/>
    <child link="propeller${i}"/>
    <origin xyz="${cos((i-1)*pi/2)*(frame_radius+arm_length)}
        ${sin((i-1)*pi/2)*(frame_radius+arm_length)}
        ${frame_height/2-arm_radius+propeller_height_offset}" rpy="0 0 0"/>
    <axis xyz="0 0 1"/>
  </joint>
```

Key part in the URDF file are the tags to insert the model plugins that will be attached to the model during the spawn phase. As shown below, gazebo plugin tag allow to insert different parameter which will be automatically passed to the plugin;this makes it more efficient to change and tune key parameters without having to re-build the code.

```
  <gazebo>
  <plugin name="drone_plugin" filename="libaerodynamic_plugin.so">
    <updateRate>1000</updateRate>
    <publishTf>true</publishTf>
    <rotorThrustCoeff>0.000011487</rotorThrustCoeff>
    <rotorTorqueCoeff>0.0000000269</rotorTorqueCoeff>
    <sim_slow>20.0</sim_slow>
  </plugin>
</gazebo>

<gazebo>
      <plugin name="motorpid_plugin" filename="libmotorpid_plugin.so">
      <kp>0.0005</kp>
      <ki>0.0</ki>
      <kd>0.0</kd>
      <sim_slow>20.0</sim_slow>
      </plugin>
</gazebo>
```



(a)



(b)

Figure 4.2: quadrotor spawned in the gazebo world showing the links mass(a), quadrotor spawned showing the inertia box created by gazebo(b).

4.2 Launch and world files

4.2.1 Launch

Both users and developers in ROS frequently work with launch files. They offer an easy approach to initialize numerous nodes, a master, and other startup requirements including specifying parameters. Launch files adopt a particular XML format and have the extension *.launch*. Despite the fact that they can be put anywhere in a package directory, it is typical to create a directory called "Launch" inside the workspace directory to keep track of all your launch files. A pair of launch tags must separate the contents of a launch file. In order to launch nodes some arguments must be provided using the node tag; these parameters are:

- *pkg, type, name*: the input "type" denotes the name of the executable file for the node, whereas "pkg" corresponds to the package associated with the node that is to be started. The name argument can be used to replace the node's name, which will take precedence over the name specified in the code.
- *Respawn/required*: however optional, it's common to either have a respawn argument or a required argument, but not both. If respawn=true, then this particular node will be restarted if for some reason it closed. Required=true will do the opposite, that is, it will shut down all the nodes associated with a launch file if this particular node comes down.
- *ns*: launching a node inside a namespace is another frequent application for a launch file. When employing many instances of the same node, this is helpful. The "ns" option can be used to specify a namespace.
- *arg*: a local variable must often be used in launch files; with the *arg* tag it is possible to declare variables to make the file more understandable and organized.

The *arg* tag is really important because it allows to load the world file into the launch file and allows to set important properties for the simulation in gazebo as []

- *paused*: start Gazebo in a paused state (default false).
- *use_sim_time*: tells ROS nodes asking for time to get the Gazebo-published simulation time, published over the ROS topic /clock (default true).
- *gui*: launch the user interface window of Gazebo (default true).
- *verbose*: run gzserver and gzclient with –verbose, printing errors and warnings to the terminal (default false).

```
<launch>

  <!-- these are the arguments you can pass this launch file, for example
      paused:=true -->
```

```
  <arg name="paused" default="true"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

  <!-- We resume the logic in empty_world.launch, changing only the name of
      the world to be launched -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find du_drone)/world/default.world"/>
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="headless" value="$(arg headless)"/>
  </include>

  <!-- Load the URDF into the ROS Parameter Server -->
  <param name="robot_description"
    command="$(find xacro)/xacro --inorder '$(find
        du_drone)/URDF/drone.URDF.xacro'" />

  <!-- Run a python script to the send a service call to gazebo_ros to spawn a
      URDF robot -->
  <node name="URDF_spawner" pkg="gazebo_ros" type="spawn_model"
      respawn="false" output="screen"
    args="-URDF -model quadrotor -param robot_description -z 0.1"/>


</launch>
```

4.2.2 World

An URDF file and a .world file have the same format. The URDF files, however, specify the types of models that can be made with Gazebo. These models can be used in different worlds if they are in their own file. A.world file describes a whole scene, including all of the objects and models. To incorporate a model into your world, you can copy the contents of a URDF file into your world file. Inside the world file several features can be set to create a custom scenario such as gravity, atmosphere or magnetic field. In addition, the physics engine to be used during the gazebo simulation can be chosen and the parameters of it can be set. In this work, ODE was chosen as the physics engine, which is the one used by default.

```
    <?xml version="1.0" ?>
<sdf version="1.4">
       <world name="default">
```

```
            <include>
                    <uri>model://ground_plane</uri>
            </include>
            <include>
                    <uri>model://sun</uri>
            </include>
            <gravity>0 0 -9.8</gravity>
        <physics:ode>
                <max_step_size>0.001</max_step_size>
                <updateRate>1000</updateRate>
        </physics:ode>
        </world>
</sdf>
```

<max_step_size> and *<updateRate>* tags are used to force the simulation to run as fast as it ca be done in order to reach a real time factor as close as possible to 1; in fact, the real time factor parameter is a key variable in simulation as it goes to greatly affect in the performance and quality of simulation.

4.3 Dynamic and Feedback plugin

Thanks to the URDF file, it is possible to create a simplified model of quadrotor; to make it as realistic as possible for the purpose of having a proper simulation of its behavior, two different plugins were created for the dynamic.

Aerodynamic plugin

The aerodynamic plugin, inspired by the work of [rotors], is responsible for the aerodynamic forces that the rotors experience. The forces examined are the same as those described in Chapter 2, thrust force, torque and drag forces, while the gyroscopic effect and Coriolis effect are simulated directly by Gazebo. The main functions in this plugin are:

- *Load()* is responsible for defining variables, security checks and storing parameters that are passed by the URDF.
- *onUpdate()* is the function that is automatically called at each gazebo clock; within it are all the aerodynamic functions and actions of the model.
- *updateForces()* is contained in onupdate and within it all forces are calculated and applied to the model.

Shown below are some lines of code that are used to store the link and joint pointers and to apply forces ni certain frames accordingly.

```
gazebo::physics::LinkPtr link_frame _model-GetLink("frame")
gazebo::physics::LinkPtr _link = _model->GetLink(link_v[i]);
gazebo::physics::JointPtr _joint = _model->GetJoint(joint_v[i]);
 if (_link != NULL) {
    _link->AddRelativeForce(ignition::math::Vector3<double>(0, 0, thrust));
    link_frame->AddRelativeTorque(ignition::math::Vector3<double>(0, 0,
        torque));
    _link->AddRelativeForce(drag_forces);
```

Propeller rotation reaction torque is the only force that is applied to the quadrotor body and not on the rotors; this is because, if put in the rotor reference system it would go to disturb the next plugin, motor plugin, And the rotors would not be able to reach the target speed.

Motor plugin

The role of the motor plugin is to take as input the controlled speeds of the rotors from the controller and apply them to the joints. The velocities are subscribed through an *sdt::msgs* of ROS, *Float64MultiArray*. In order to set up the subscriber node the following lines of codes are used.

```
// Create our ROS node.
    this->rosNode.reset(new ros::NodeHandle("rotor"));

// Create a named topic, and subscribe to it.
    ros::SubscribeOptions so =
        ros::SubscribeOptions::create<std_msgs::Float64MultiArray>(
        "/" + this->model->GetName() + "/vel_cmd",
        1000,
        boost::bind(&MotorpidPlugin::OnRosMsg, this, _1),
        ros::VoidPtr(), &this->rosQueue);
    this->rosSub = this->rosNode1->subscribe(so);
```

As you can see, a subscriber node and a subscriber is created where the message type, topic name and queue length is defined. The main function in the motor plugin are:

- *Load()* is responsible for defining variables, security checks and storing parameters that are passed by the URDF.
- *OnRosMsg()* is called whenever there is a new incoming message; it stores the values and sends them to the joint controller.
- *QueueThread()* is a function that help ROS to process the messages.

In order to set a velocity on a joint in Gazebo and allow the propeller to spin, Gazebo need a controller. From the Gazebo API, a pid controller is provided that is attached to the joint and brings it to the target speed. The parameters of the pid have been tuned to simulate a first-order motor with a τ-motor constant equal to 20.

K_P	K_I	K_D
0.005	0.002	0.0

Table 4.1: Reference parameters.

Below are the lines of code to set the PID controller and apply it.

```
// Setup a PID-controller.
    this->pid = gazebo::common::PID(kp, ki, kd);

// Apply the PID-controller to the joint.
    this->model->GetJointController()->SetVelocityPID(
        this->joint->GetScopedName(), this->pid);

// Set the velocity target
    this->model->GetJointController()->SetVelocityTarget(
    this->joint->GetScopedName(), vel_cmd);
```

Odometry plugin

The odometry plugin the simulates a sensor attached on the model in gazebo so it can take all the feedback data needed by the controller and for performance evaluation purposes. The message published by the plugin is a $nav_msgs/Odometry.msg$ provided by ROS and it is composed as

```
Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

where the pose and twist messages are defined by

```
// pose msg
  geometry_msgs/Point position // x, y, z
```

```
   geometry_msgs/Quaternion orientation // x, y, z, q

// twist msg
   geometry_msgs/Vector3 linear // linear velocity
   geometry_msgs/Vector3 angular // angular velocity
```

. The main functions in the plugin are:

- *Load()*, where the node and the publisher are initialized and also where the function *rosThread()* is called.
- *onUpdate()*, where for each clock iteration of Gazebo, the pose and velocity of the model is stored.
- *rosThread()*, where there is the function *publish()* and it help ROS to process the publisher setting a rate and allowing ROS to spin.
- *publish()* is the function responsible for the filling and publishing of the odometry message and also for the application of noise errors.

To have more realism in the simulation of a sensor, an error model was adopted to be added to the odometry data. The error model chosen is a white gaussian noise, which is a highly used model in process and model simulation. The model is described as

$$\begin{aligned} Y_i &= X_i + E_i, \\ E_i &\sim \mathbb{N}(0, N). \end{aligned} \tag{4.1}$$

$\mathbb{N}$ identifies a normal distribution with a zero-mean value and with a variance N (the noise boundary value). To achieve this error model in the plugin the following lines of code are used

```
    unsigned seed =
        std::chrono::system_clock::now().time_since_epoch().count();
    std::default_random_engine generator(seed);
    std::normal_distribution<double> distribution_pos(0.0,
        double_sigma_position);
    std::normal_distribution<double> distribution_rpy(0.0,
        double_sigma_quaternion);
    /* white_gaussian_noise_error = distribution_pos(generator);*/
```

The first two lines are used to initialize a clock and a random value generator; the generator will be applied on the normal distribution function to obtain the desired error model. The value used for the variance is doubled to achieve a 95% of coverage of the normal distribution.

Chapter 5

Bridge Matlab/Simulink-ROS/Gazebo

Matlab's Rostoolbox allows its environment to be linked with that of ROS. This can be done in two different ways:

- Enable external mode: this method allows you to connect the two environments by running them simultaneously; from the Matlab/Simulink interface you launch the deployment, which when finished will be loaded and started on ROS.
- Generate a standalone ROS node: this method allows a ros node to be generated that can be built automatically or by the user in a defined workspace, and then run at a later time.

The second method is more effective than the first one since the simultaneous use of the two environments strongly impacts the computational performance by affecting the simulation. In order to set up the model validation and the experiment some features of Simulink and ROStoolbox are used. The Simulink code created for the model validation, fig(5.1), takes the rotor speed signals as input and publishes them in the *geometry_msgs/Float64MultiArray* topic and receives via a Subscriber block the odometry data published by the odometry plugin in the simulation in a .mat file. The Simulink code is auto generated by creating 3 different files needed for the subsequent build (.sh file, rtw file and .tgz file). These three file are moved and built to the workspace and built using the following two command line terminal

```
$ ./build_ros_model.sh <filename>.tgz .
$ /workspace rosrun <filename> <filename>
```

Once the simulation is finished, the output .mat file will be generated in the same folder where the node was launched.

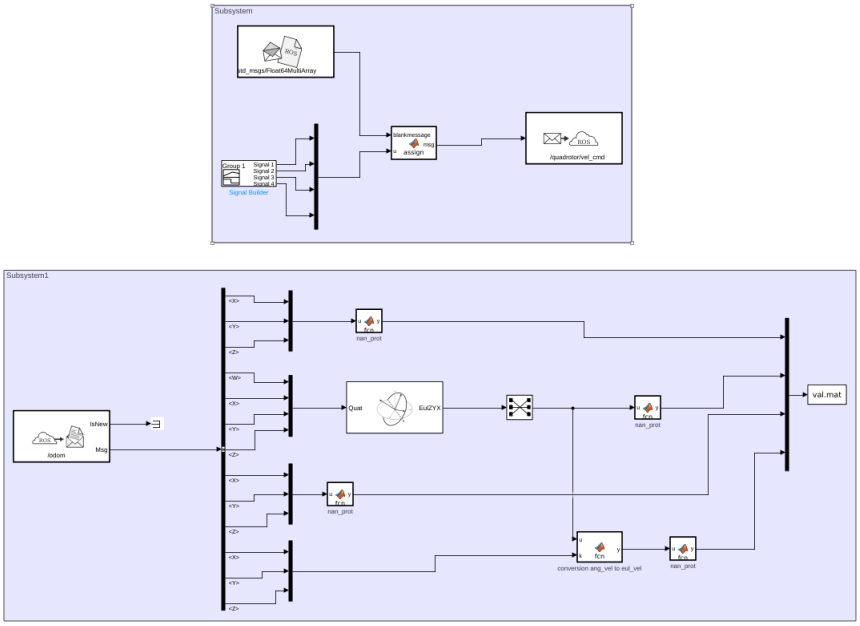


Figure 5.1: Simulink model for input/output model validation simulation.

5.1 ROS input/output

The ROS input subsystem is composed as

- *Blank message*: this block allow to select the type of ROS message to use, selecting it from the topics available, and to set the length of the queue.
- *Publish block*: this block connect and send the message from the input to the topic publisher in the simulation.
- *Bus function*: take as input the signal and fill the Blank message to match the publisher type topic.

The ROS output subsystem is composed with a Subscriber block which takes information from the selected topic whenever it is called, in this case the odometry topic, and send the information to a .mat file which stores all the feedback data.

5.2 Model validation set up

The model validation consists of comparing two different models; the first, that of Matlab mathematician, and already validated in [5], and the second the model presented in this work on ROS. The two simulations will be compared using the same signal input. The signal input, was created through trial and error to obtain an upward trajectory without having angles reaching 90 degrees, in which case the mathematical model would encounter

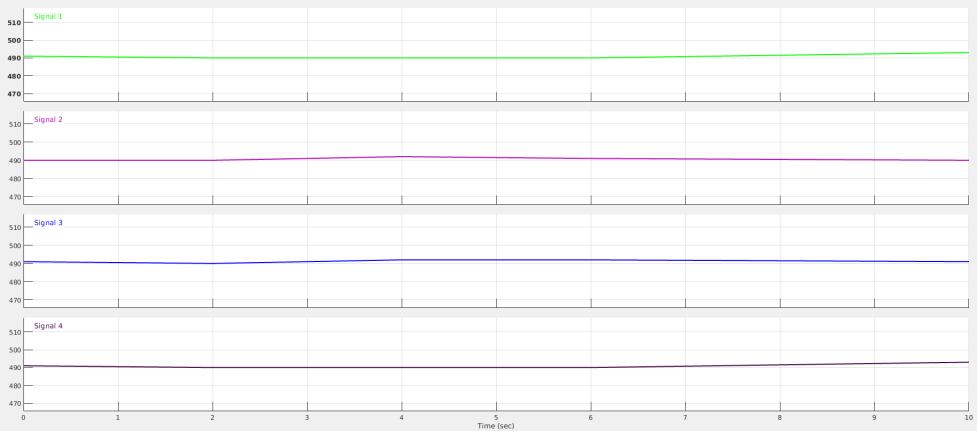


Figure 5.2: Speed input signal.

singularities and lose validity. The input signal is added as it is shown in fig(5.1) and in the figure below, which represent the simply set up for the mathematical model.

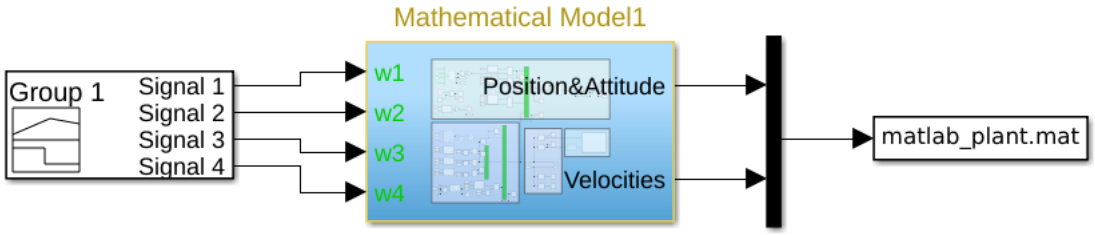


Figure 5.3: Mathematical model set up.

Chapter 6

Simulation

This chapter shows the results of model validation and deployment of the PID controller. During the testing phase, it was noticed how Gazebo's performance is highly variable depending on several conditions; the first of all is the CPU power of the computer or virtual machine on which the simulation is launched. Also by observing behaviors and results that are sometimes very far from the results of the mathematical model, it was found that another big cause of error is the ODE physics engine. Ode physics engine uses numerical integration methods to calculate the motion of the objects and when fasting rotating objects are simulated, the number of calculation required to accurately track their motion increase significantly, which can cause errors. In fact, when there are objects in the scenario that rotate at high speeds, as in this case there are the four rotors, Gazebo cannot keep up with the simulation, thus leading to strange behaviors of the model that exhibits slipping and bouncing effects as will be seen in several graphs. Furthermore, the numerical integration can affect the value and property of the moment of inertia of the model, leading to behavior different from that expected. To try to overcome this, a velocity slowdown parameter was included in the plugins and for the model validation different values of it have been tested; it must be remembered, however, that a slowdown coefficient on the rotors goes to change the gyroscopic effect acting on them, so a trade of between these problems must be pursued.

6.1 Model validation results

The results of the model validation are shown in the figures below in this section. The graphs show the evolution of the state variables ξ, η, $\dot{\xi}$ and $\dot{\eta}$, given the defined input signal in the previous chapter. Each drag force was simulated by turning off the others to evaluate their behavior and error, and then went to test the simulation with all drag forces present for different slowdown parameter values. The slowdown values chosen are 1, 8 and 18.

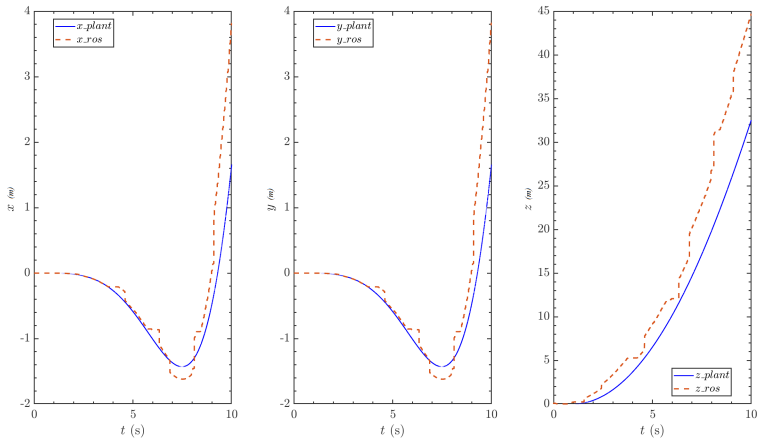


Figure 6.1: x, y, z profile response

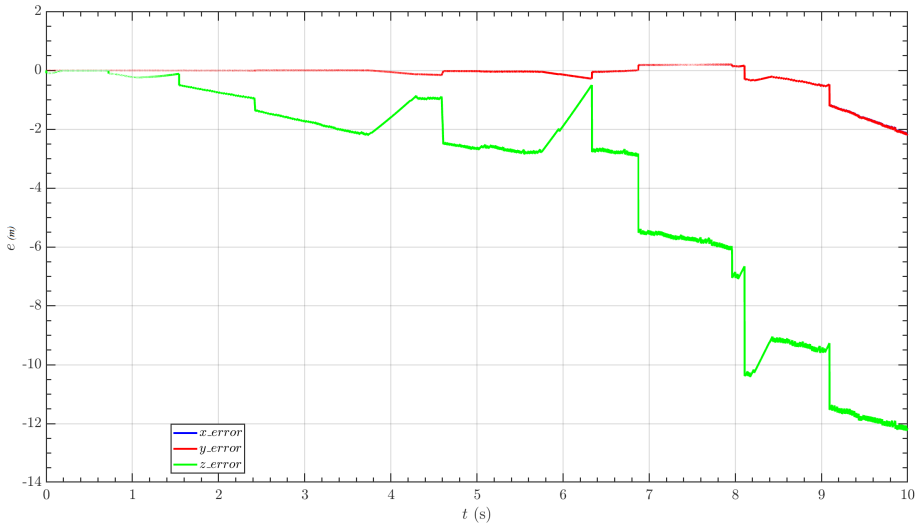


Figure 6.2: $e_{x,y,z}$ profile response

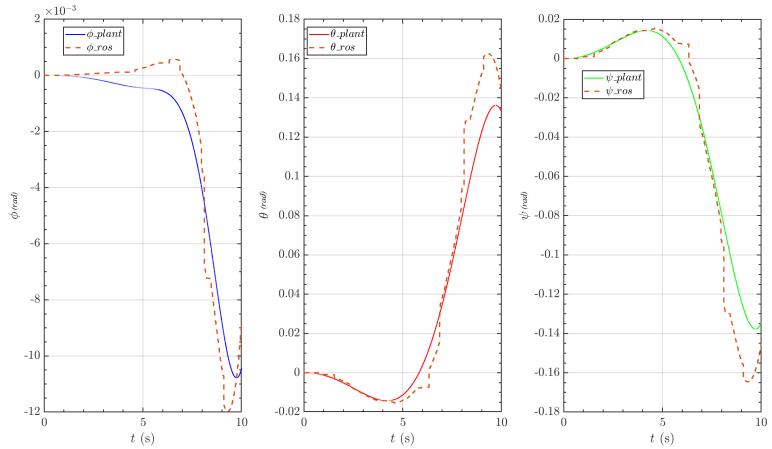


Figure 6.3: ϕ, θ, ψ profile response

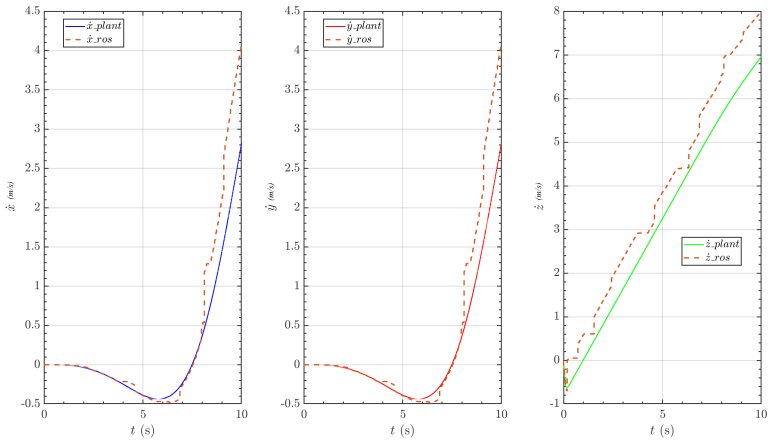


Figure 6.4: $\dot{x}, \dot{y}, \dot{z}$ profile response

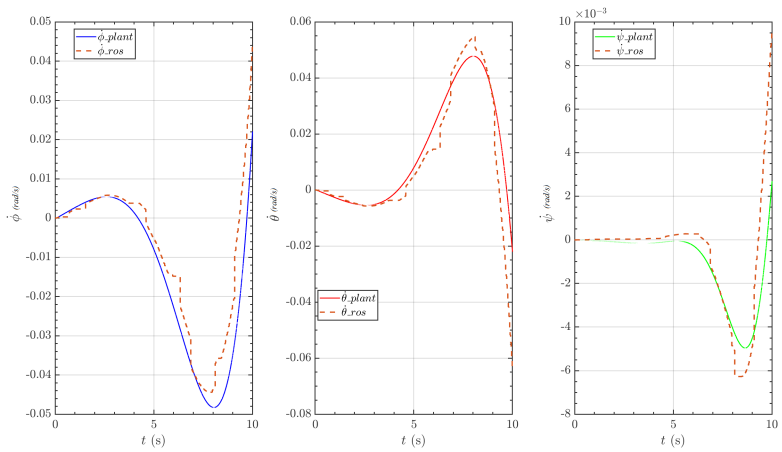


Figure 6.5: $\dot{\phi}, \dot{\theta}, \dot{\psi}$ profile response

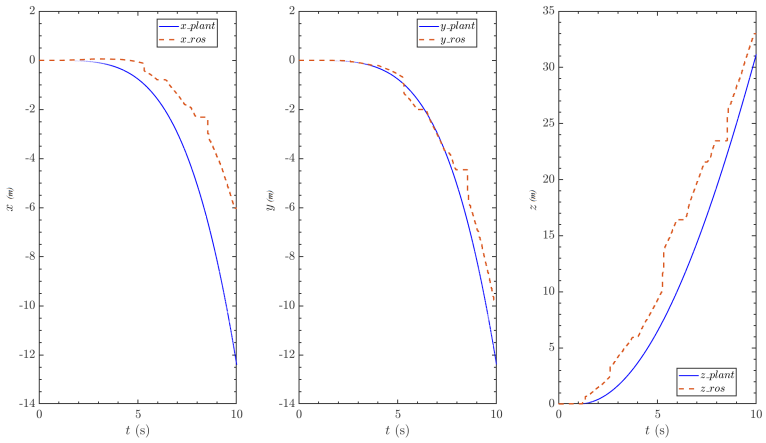


Figure 6.6: x, y, z parasitic response

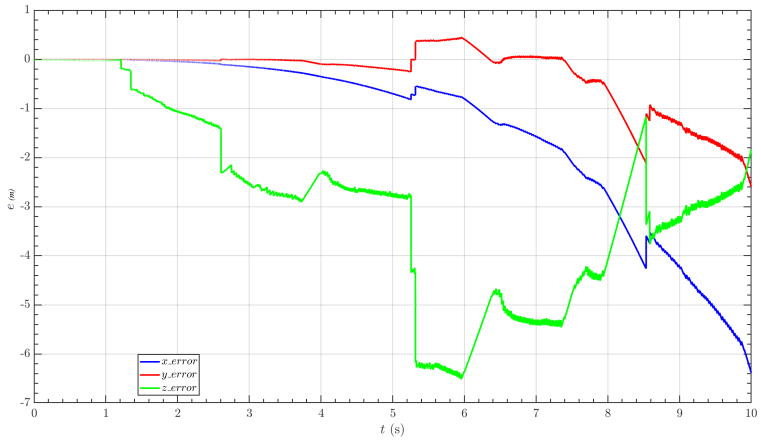


Figure 6.7: $e_{x.y.z}$ parasitic response

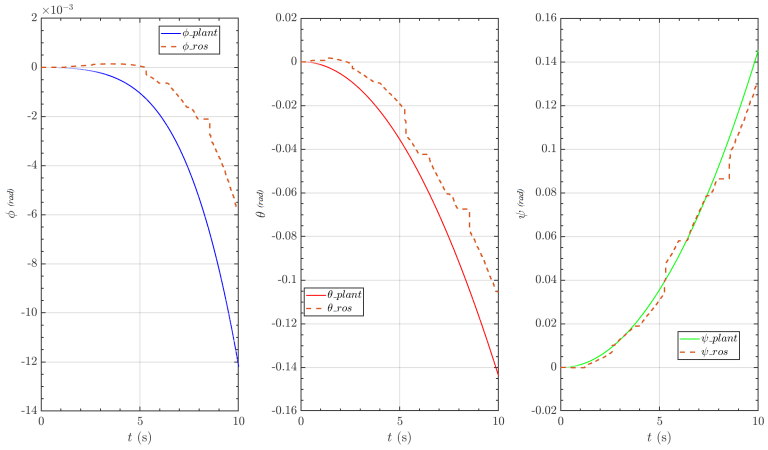


Figure 6.8: ϕ, θ, ψ parasitic response

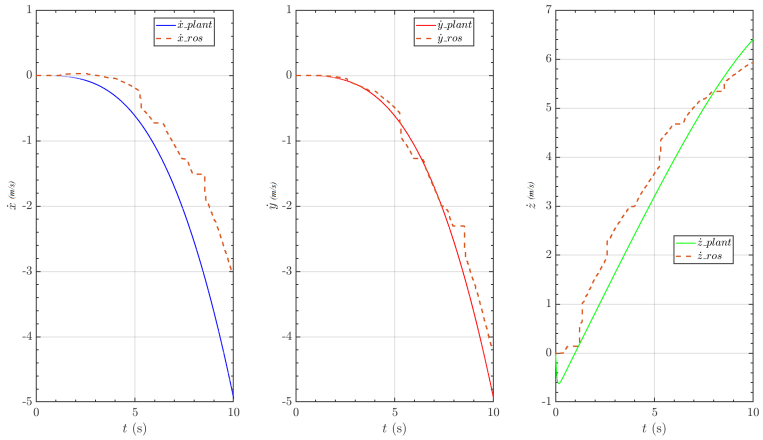


Figure 6.9: $\dot{x}, \dot{y}, \dot{z}$ parasitic response

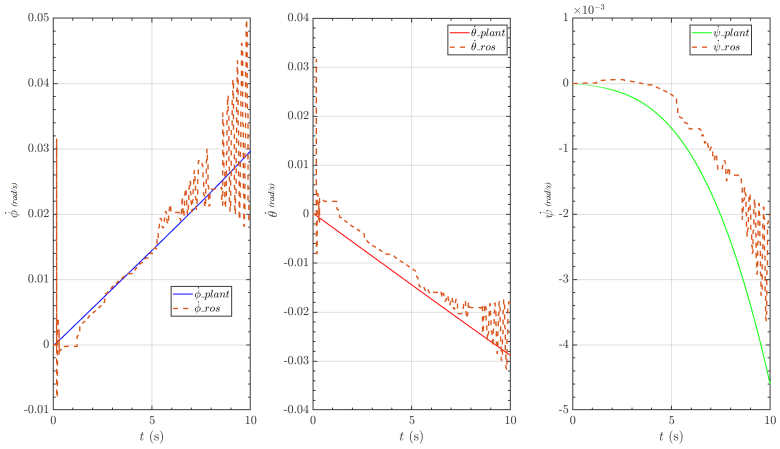


Figure 6.10: $\dot{\phi}, \dot{\theta}, \dot{\psi}$ parasitic response

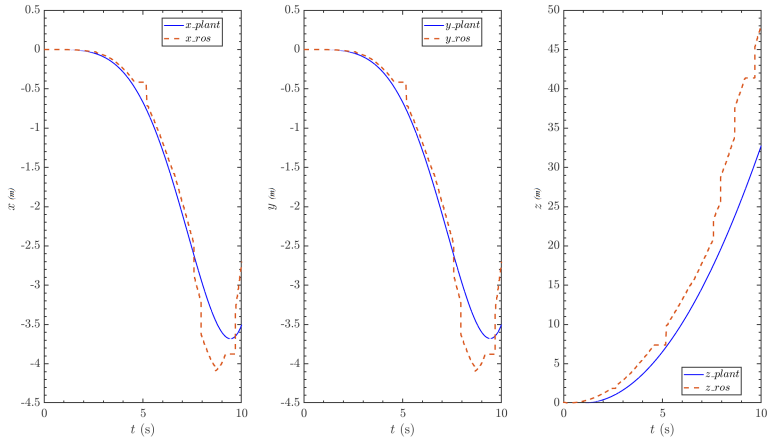


Figure 6.11: x, y, z translational response

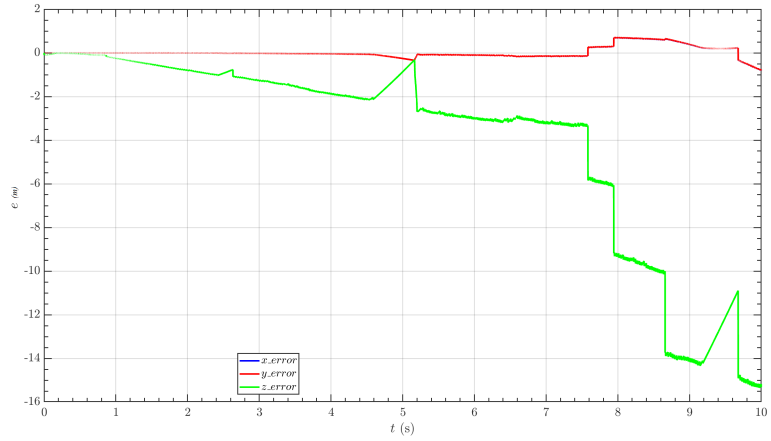


Figure 6.12: $e_{x,y,z}$ translational response

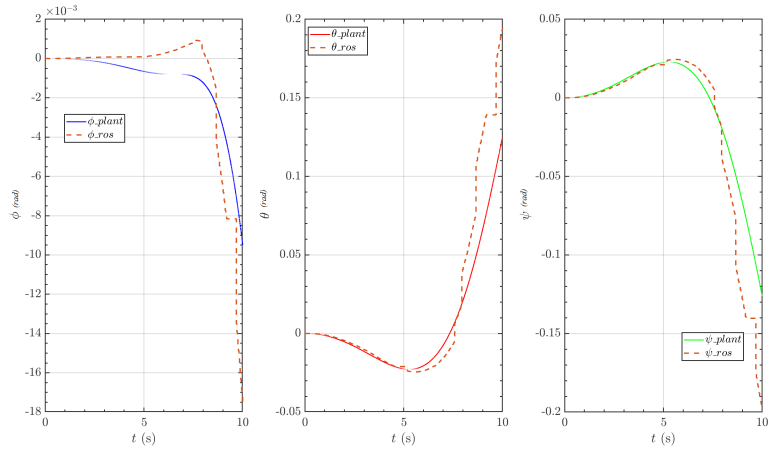


Figure 6.13: ϕ, θ, ψ translational response

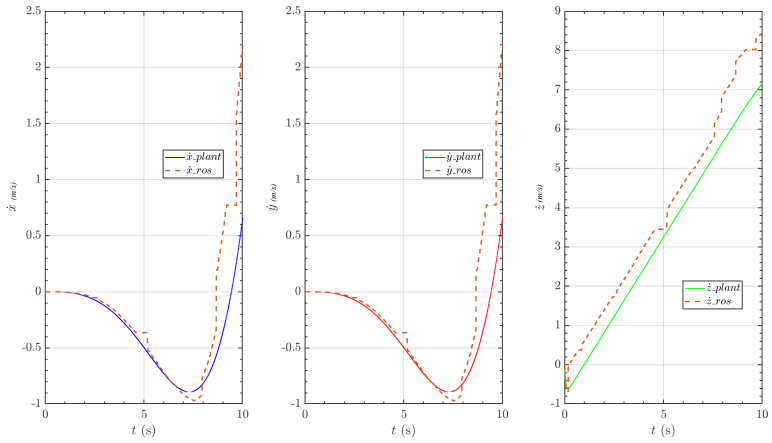


Figure 6.14: $\dot{x}, \dot{y}, \dot{z}$ translational response

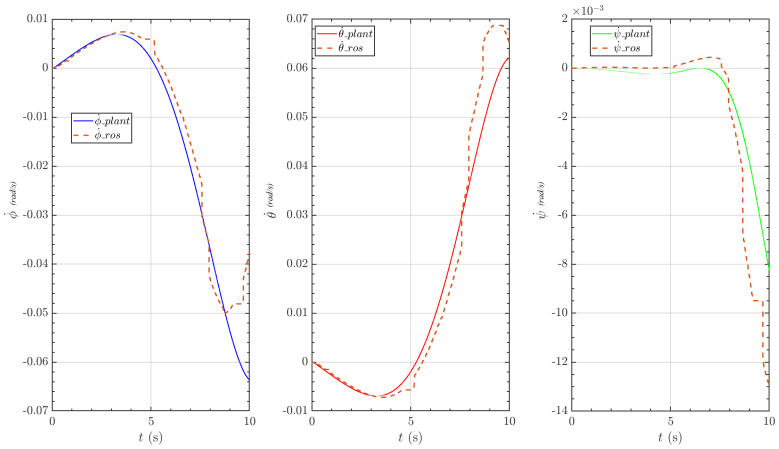


Figure 6.15: $\dot{\phi}, \dot{\theta}, \dot{\psi}$ translational response

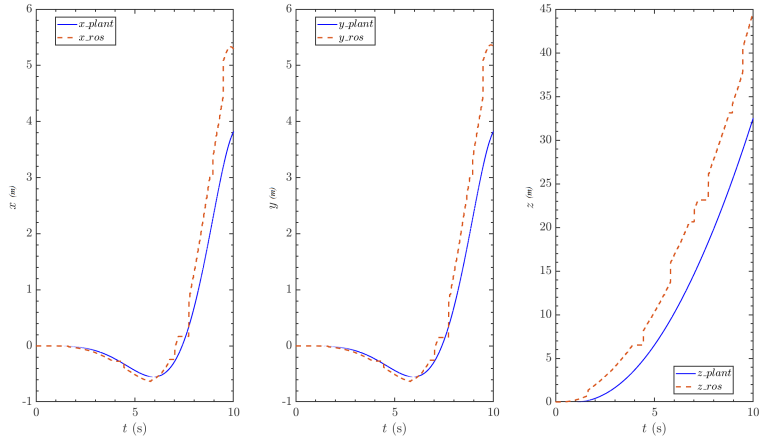


Figure 6.16: x, y, z induced response

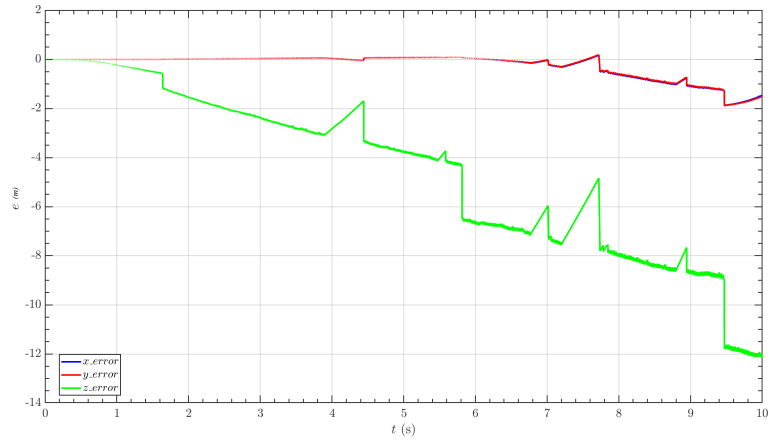


Figure 6.17: $e_{x,y,z}$ induced response

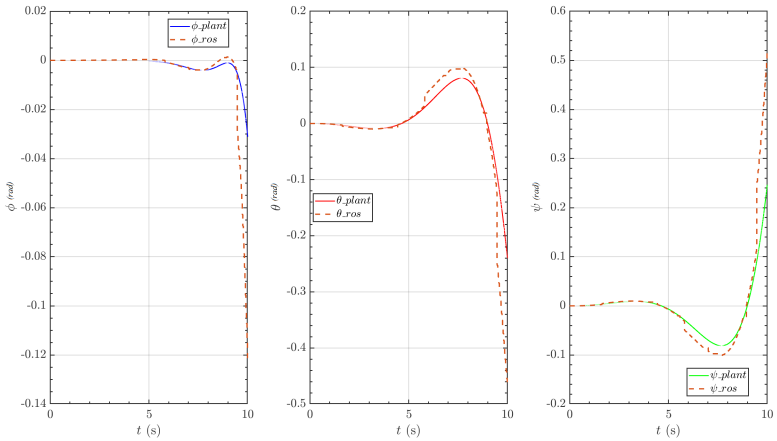


Figure 6.18: ϕ, θ, ψ induced response

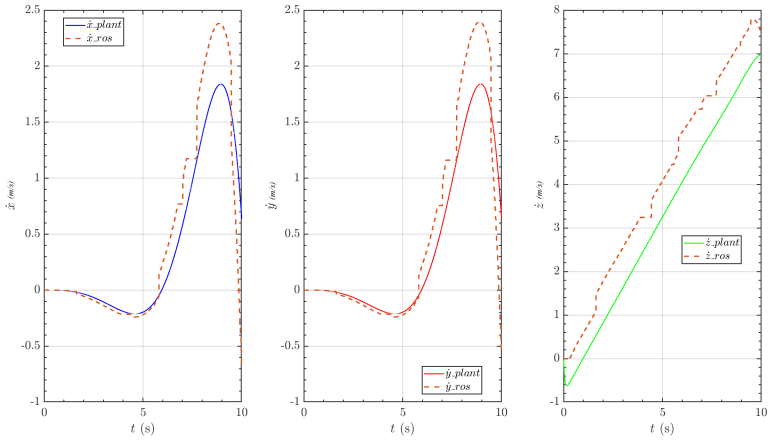


Figure 6.19: $\dot{x}, \dot{y}, \dot{z}$ induced response

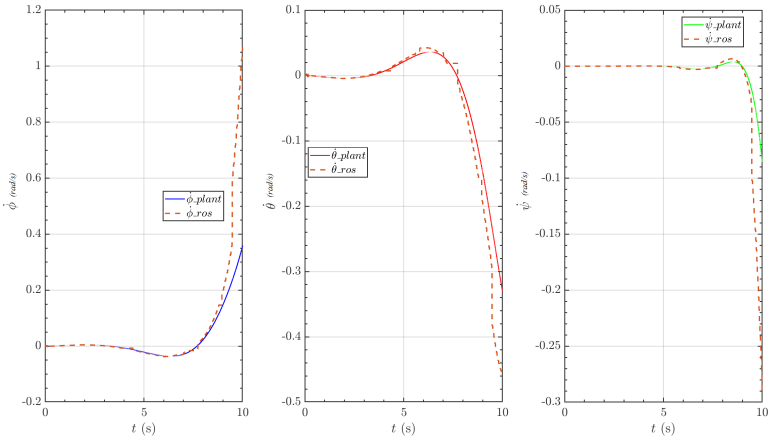


Figure 6.20: $\dot{,\theta}, \dot{\psi}$ induced response

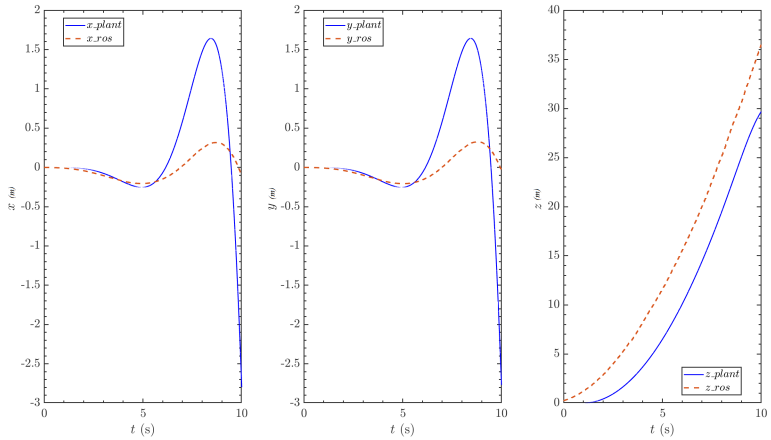


Figure 6.21: x, y, z all drag activated response with slowdown 1

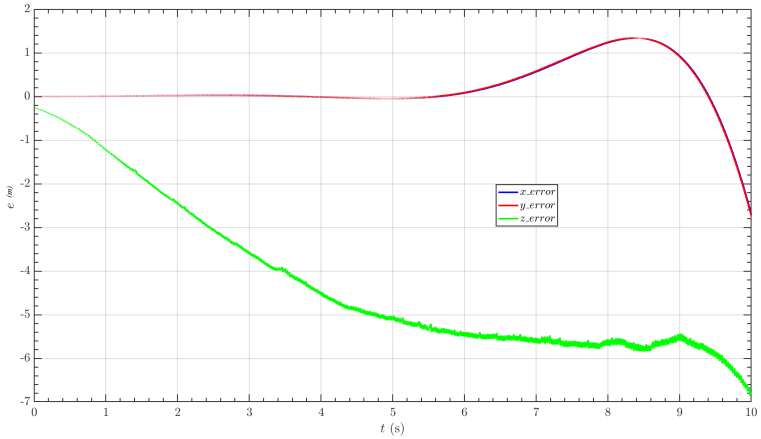


Figure 6.22: $e_{x,y,z}$ all drag activated response with slowdown 1

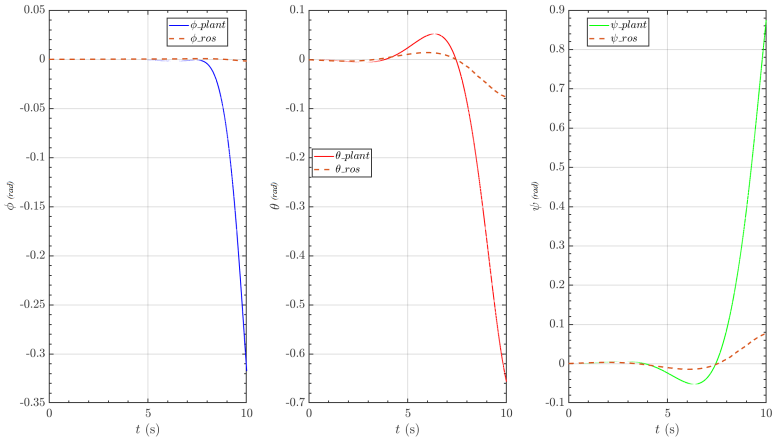


Figure 6.23: ϕ, θ, ψ all drag activated response with slowdown 1

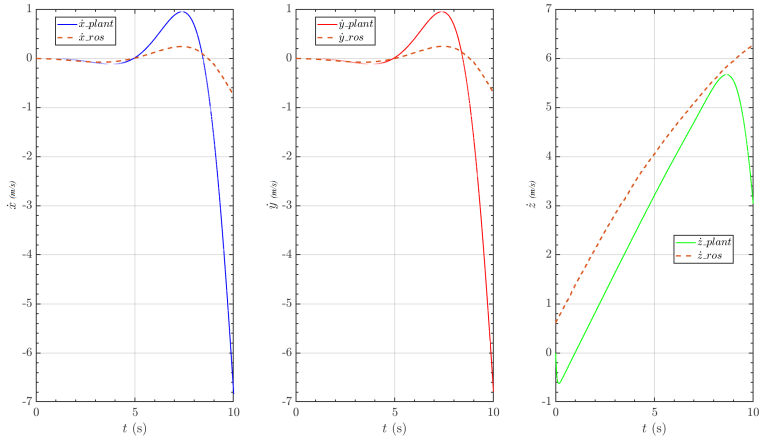


Figure 6.24: $\dot{x}, \dot{y}, \dot{z}$ all drag activated response with slowdown 1

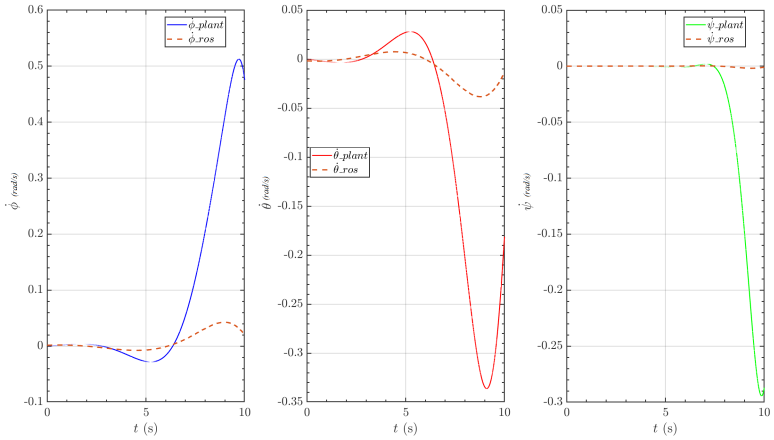


Figure 6.25: $\dot{\phi}, \dot{\theta}, \dot{\psi}$ all drag activated response with slowdown 1

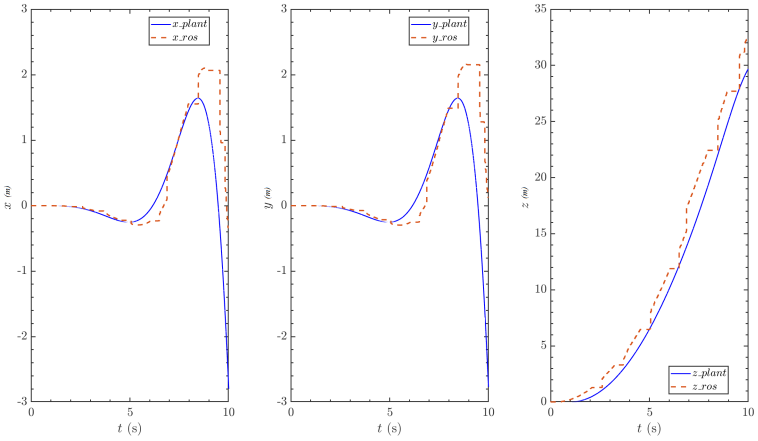


Figure 6.26: x, y, z all drag activated response with slowdown 8

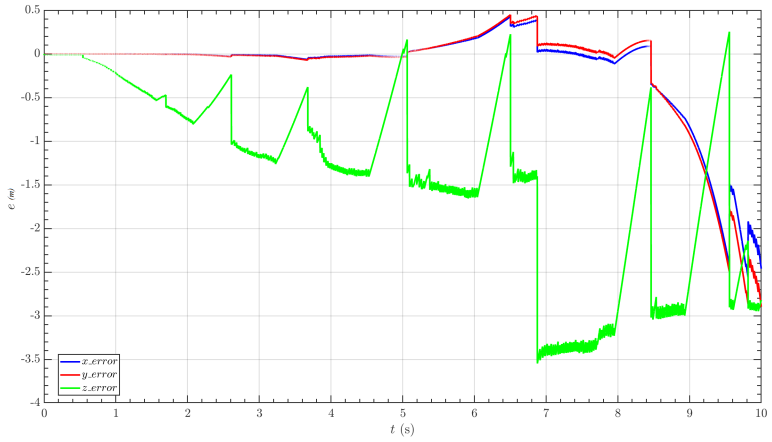


Figure 6.27: $e_{x,y,z}$ all drag activated response with slowdown 8

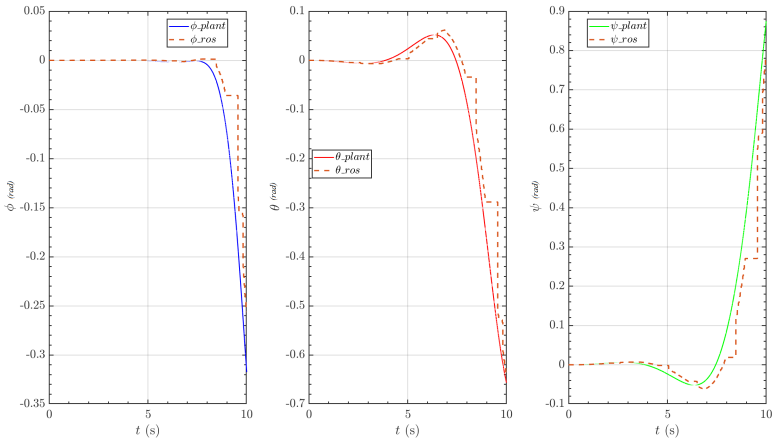


Figure 6.28: ϕ, θ, ψ all drag activated response with slowdown 8

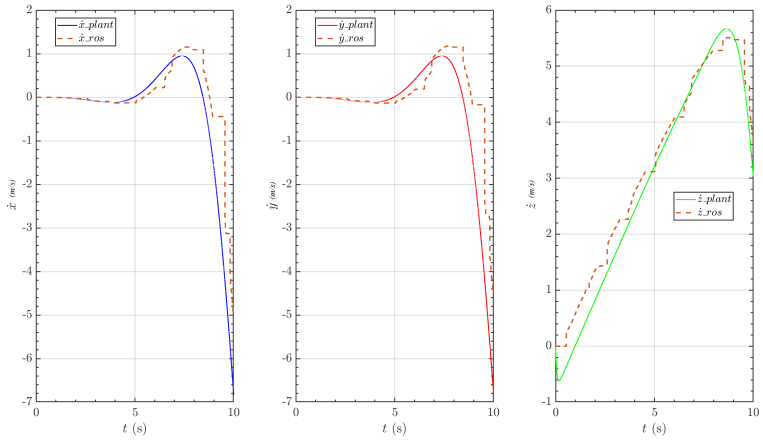


Figure 6.29: $\dot{x}, \dot{y}, \dot{z}$ all drag activated response with slowdown 8

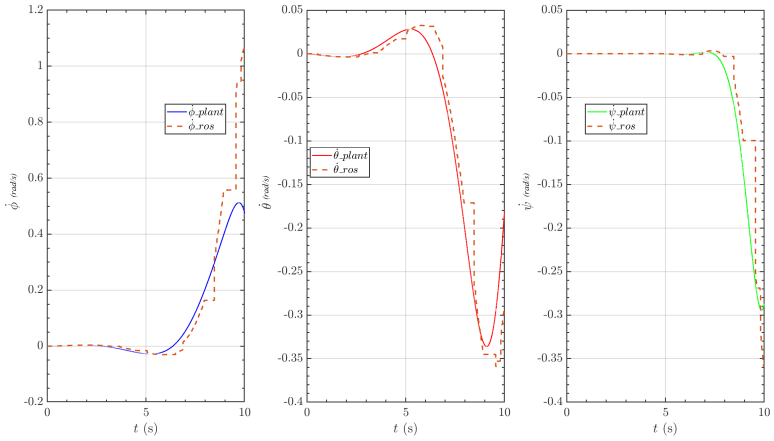


Figure 6.30: $\dot{\phi}, \dot{\theta}, \dot{\psi}$ all drag activated response with slowdown 8

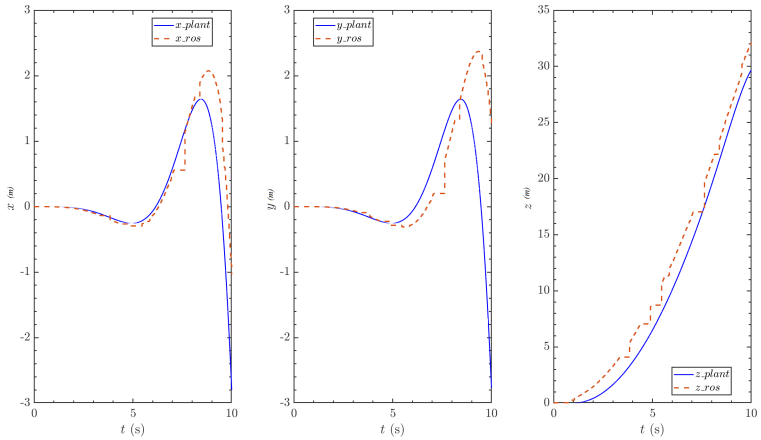


Figure 6.31: x, y, z all drag activated response with slowdown 10

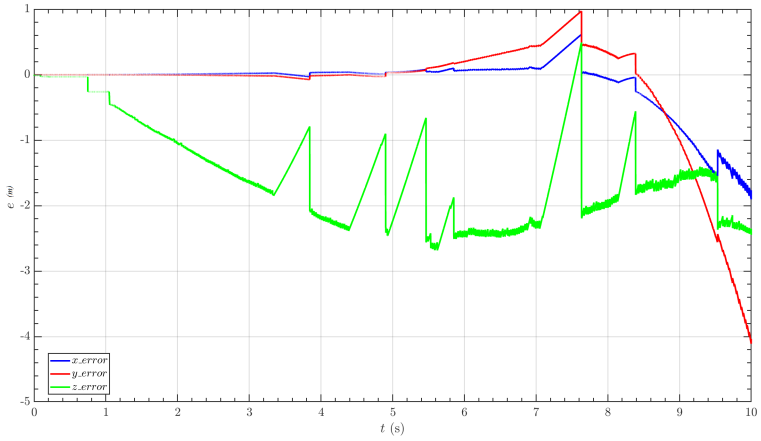


Figure 6.32: $e_{x,y,z}$ all drag activated response with slowdown 10

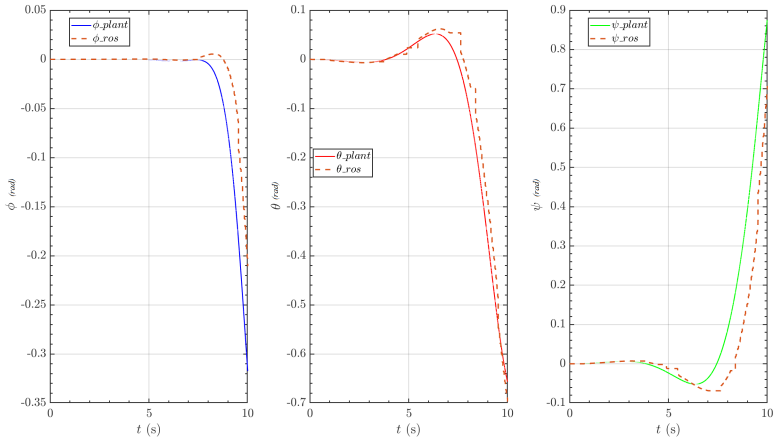


Figure 6.33: ϕ, θ, ψ all drag activated response with slowdown 10

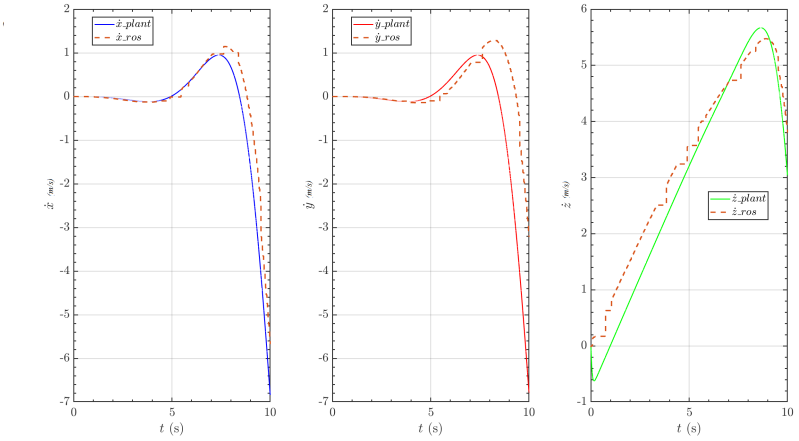


Figure 6.34: $\dot{x}, \dot{y}, \dot{z}$ all drag activated response with slowdown 10

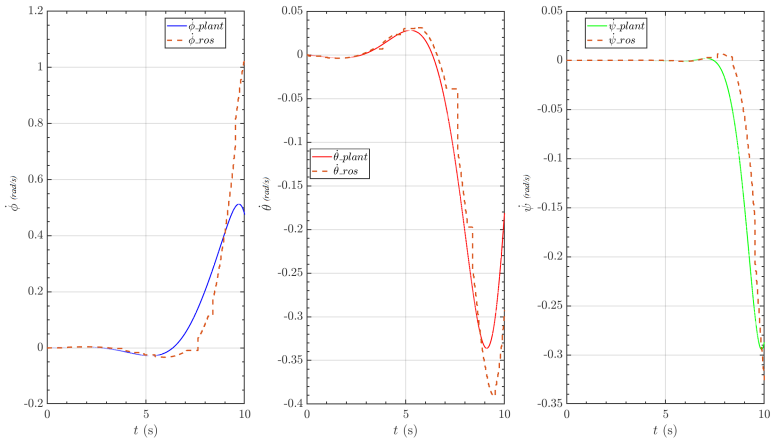


Figure 6.35: $\dot{\phi}, \dot{\theta}, \dot{\psi}$ all drag activated response with slowdown 10

6.2 PID deployement

The reference signal used for the simulation of the controller is a spiral ascendant trajectory, which is defined by

$$\begin{aligned} x &= k_{x,1} cos(k_{x,2} * t) - x_0 \\ y &= k_{y,1} sin(k_{y,2} * t) - y_0 \\ z &= k_z * t - z_0 \\ \psi &= 0 \end{aligned} \tag{6.1}$$

where all the constants $k_{x,y}$ describe the period and the amplitude of the trajectory, while k_z defines how fast the model go up.

$k_{x,1}, k_{y,1}$	2m
$k_{x,2}, k_{y,2}$	0.5Hz
k_z	1m/s
x_0, y_0, z_0	0

Table 6.1: Reference parameters.

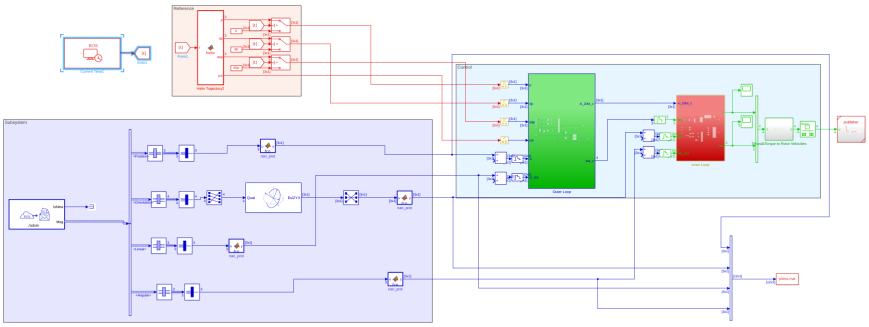


Figure 6.36: Simulink set up for the PID node deployement.

At first, the parameters of the Matlab simulation scaled by a factor of 0.8 were used for tuning the PID but this values lead to an unstable simulation and a crash of the controller after 10s from the launch because of the differences between the mathematical model and the ROS model explained above.

Even by re-tuning the controller, the results are not reliable for performance evaluation; the causes could be found in the error sources mentioned at the beginning of the chapter, on all of them the CPU power in numerical integration of the physics engine. The controller

PID	Outer loop	Inner loop
K_P	2.4[1 1 1]	96[1 1 1]
K_I	0.8[1 1 1]	96[1 1 1]
K_D	0.125[1 1 90]	16[1 1 1]

Table 6.2: PID parameters.

is deployed, built and ran correctly showing the functionality of the bridge between the two simulation environments. Future tests may investigate deeper into the sources of errors, using a higher performance hardware or machine to run the simulations.

Chapter 7

Conclusion and future work

From the results obtained from model validation and PID deployment, it can be concluded that the model and ROS package follow the behavior of the proposed mathematical model; however, the errors present in both model validation and deployment are too high for more accurate performance evaluation of complex control systems. The causes of these errors are many, as a to high complexity in the model, slipping and bouncing effects in Gazebo, but the most prominent are the the power of the CPU and physics engine that must support a complex mathematical model during simulation. The bridge between the two simulation environments, Matlab/Simulink and ROS/Gazebo, is a very strong tool that allows a more direct and intuitive design in the first and a more realistic testing phase in the second. In order to improve and decrease the discrepancies between the two models, several actions can be implemented for the future, such as the use of performing and specific hardware that can withstand the numerical integration of the physics engine and the simplification of the mathematical model, which in this case has a high number of drag forces on each rotor. Also regarding the odometry plugin, in future work the modeling of noise in the feedback data from white gaussian noise to a more realistic noise model can be improved; the package can also be improved by expanding the plugins with scenarios and weather conditions (i.e. wind gust) and by adding new models such as multirotors since the aerodynamics plugin can be easily made interchangeable for different uses by editing only the URDF file. The versatility and modularity of ROS/Gazebo allows you to be able to create different test scenarios and goals, using a structure that can be easily adapted even to different models.

Appendix A

Code

A.1 quadrotor.urdf.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://wiki.ros.org/xacro" name="quadrotor">

  <!-- properties -->
  <xacro:property name="frame_radius" value="0.1"/>
  <xacro:property name="frame_height" value="0.025"/>
  <xacro:property name="frame_mass" value="1.0"/>
  <xacro:property name="arm_radius" value="0.0125"/>
  <xacro:property name="arm_length" value="0.1"/>
  <xacro:property name="arm_mass" value="0.0"/>
  <xacro:property name="propeller_radius" value="0.075"/>
  <xacro:property name="propeller_height" value="0.01"/>
  <xacro:property name="propeller_height_offset" value="0.0168"/>
  <xacro:property name="propeller_mass" value="0.01"/>
  <xacro:property name="z_com" value="0.0132"/>


  <xacro:macro name="arm" params="i">
  </xacro:macro>
  <xacro:macro name="propeller" params="i mat">
    <link name="propeller${i}">
      <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
          <cylinder radius="${propeller_radius}" length="${propeller_height}"/>
        </geometry>
        <material name="propeller_material"/>
      </visual>
      <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
```

```
      <geometry>
        <cylinder radius="${propeller_radius}" length="${propeller_height}"/>
      </geometry>
    </collision>
      <inertial>
      <mass value="0.01"/>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <inertia ixx="0.0000140625" ixy="0.0" ixz="0.0" iyy="0.0000140625"
          iyz="0.0" izz="0.000028125"/>
    </inertial>
  </link>

  <joint name="rotor${i}" type="continuous">
    <parent link="frame"/>
    <child link="propeller${i}"/>
    <origin xyz="${cos((-i+1)*pi/2)*(frame_radius+arm_length)}
        ${sin((-i+1)*pi/2)*(frame_radius+arm_length)} 0.016826923076923"
        rpy="0 0 0"/>
    <axis xyz="0 0 1"/>
  </joint>

  <gazebo reference="propeller${i}">
    <material>${mat}</material>
  </gazebo>
</xacro:macro>


<!-- links -->

<link name="frame">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="${frame_radius}" length="${frame_height}"/>
    </geometry>
    <material name="frame_material">
      <color rgba="0.8 0.8 0.8 1.0"/>
    </material>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="${frame_radius}" length="${frame_height}"/>
    </geometry>
    </collision>
  <inertial>
      <mass value="1.0"/>
      <origin xyz="0 0 0" rpy="0 0 0"/>
```

```
      <inertia ixx="0.006822916666667" ixy="0.0" ixz="0.0"
          iyy="0.006822916666667" iyz="0.0" izz="0.013541666666667"/>
    </inertial>
  </link>

  <xacro:propeller i="1" />
  <xacro:propeller i="2" />
  <xacro:propeller i="3" />
  <xacro:propeller i="4" />

  <!-- gazebo -->
  <gazebo reference="frame">
    <material>Gazebo/Red</material>
  </gazebo>



  <gazebo>
    <plugin name="drone_plugin" filename="libaerodynamic_plugin.so">
      <updateRate>1000</updateRate>
      <publishTf>true</publishTf>
      <rotorThrustCoeff>0.000011487</rotorThrustCoeff>
      <rotorTorqueCoeff>0.0000000269</rotorTorqueCoeff>
      <sim_slow>10.0</sim_slow>
    </plugin>
  </gazebo>

  <gazebo>
        <plugin name="motorpid_plugin" filename="libmotorpid_plugin.so">
        <kp>0.0005</kp>
        <ki>0.0</ki>
        <kd>0.0</kd>
        <sim_slow>10.0</sim_slow>
        </plugin>
  </gazebo>

    <plugin name="odometry" filename="libodometry_plugin.so">
        <rate>1000.0</rate>
    </plugin>
  </gazebo>
</robot>
```

A.2 quadrotor.launch

```
    <launch>
```

```
  <!-- these are the arguments you can pass this launch file, for example
      paused:=true -->
  <arg name="paused" default="true"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

  <!-- We resume the logic in empty_world.launch, changing only the name of
      the world to be launched -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find du_drone)/world/default.world"/>
    <arg name="debug" value="$(arg debug)" />
    <arg name="gui" value="$(arg gui)" />
    <arg name="paused" value="$(arg paused)"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="headless" value="$(arg headless)"/>
  </include>

  <!-- Load the URDF into the ROS Parameter Server -->
  <param name="robot_description"
    command="$(find xacro)/xacro --inorder '$(find
        du_drone)/urdf/drone.urdf.xacro'" />

  <!-- Run a python script to the send a service call to gazebo_ros to spawn a
      URDF robot -->
  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
      respawn="false" output="screen"
    args="-urdf -model quadrotor -param robot_description -z 0.1"/>


</launch>
```

A.3 default.world

```
    <?xml version="1.0" ?>
<sdf version="1.4">
       <world name="default">
              <include>
                     <uri>model://ground_plane</uri>
              </include>
              <include>
                     <uri>model://sun</uri>
              </include>
              <gravity>0 0 -9.8</gravity>
       <physics:ode>
```

```
            <max_step_size>0.001</max_step_size>
            <updateRate>1000</updateRate>
        </physics:ode>
        </world>
</sdf>
```

A.4 motorpid_plugin.cpp

```
#ifndef _MOTORPID_PLUGIN_HH_
#define _MOTORPID_PLUGIN_HH_

//  Gazebo api
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/transport/transport.hh>
#include <gazebo/msgs/msgs.hh>

// ROS
#include <thread>
#include "ros/ros.h"
#include "ros/callback_queue.h"
#include "ros/subscribe_options.h"
#include "std_msgs/Float64MultiArray.h"
#include <mutex>
#include <du_drone/MotorCommand.h>



/// \brief A plugin to control a Velodyne sensor.
class MotorpidPlugin : public gazebo::ModelPlugin
{
public:
    /// \brief Constructor
    MotorpidPlugin(){ }

    virtual void Load(gazebo::physics::ModelPtr _model, sdf::ElementPtr sdf)
    {
        // Safety check
        if (_model->GetJointCount() == 0)
        {
            std::cerr << "Invalid joint count, Motor plugin not loaded\n";
            return;
        }

        if (sdf->HasElement("kp")) {
            kp = sdf->GetElement("kp")->Get<double>();
```

```
} else {
    kp = 0.0;
}

if (sdf->HasElement("ki")) {
    ki = sdf->GetElement("ki")->Get<double>();
} else {
    ki = 0.0;
}

if (sdf->HasElement("kd")) {
    kd = sdf->GetElement("kd")->Get<double>();
} else {
    kd = 0.0;
}
if (sdf->HasElement("sim_slow")) {
    rotor_vel_sim_slowdown = sdf->GetElement("sim_slow")->Get<double>();
} else {
    rotor_vel_sim_slowdown = 10;
}

// Store the model pointer for convenience.
this->model = _model;

// Get the first joint. We are making an assumption about the model
// having one joint that is the rotational joint.
this->joint1 = _model->GetJoint("arm1propeller1");
this->joint2 = _model->GetJoint("arm2propeller2");
this->joint3 = _model->GetJoint("arm3propeller3");
this->joint4 = _model->GetJoint("arm4propeller4");

// Setup a P-controller, with a gain of 0.1.
this->pid = gazebo::common::PID(kp, ki, kd);

// Apply the P-controller to the joint.
this->model->GetJointController()->SetVelocityPID(
    this->joint1->GetScopedName(), this->pid);

this->model->GetJointController()->SetVelocityPID(
    this->joint2->GetScopedName(), this->pid);

 this->model->GetJointController()->SetVelocityPID(
    this->joint3->GetScopedName(), this->pid);

 this->model->GetJointController()->SetVelocityPID(
    this->joint4->GetScopedName(), this->pid);


// Create the node
```

```
    this->node = gazebo::transport::NodePtr(new gazebo::transport::Node());
    #if GAZEBO_MAJOR_VERSION < 8
    this->node->Init(this->model->GetWorld()->GetName());
    #else
    this->node->Init(this->model->GetWorld()->Name());
    #endif

    // Initialize ros, if it has not already bee initialized.
    if (!ros::isInitialized())
    {
        int argc   = 0;
        char** argv = NULL;
        ros::init(argc, argv, "gazebo_client",
          ros::init_options::NoSigintHandler);
    }


    // Create our ROS node. This acts in a similar manner to
    // the Gazebo node
    this->rosNode1.reset(new ros::NodeHandle("rotor"));


    // Create a named topic, and subscribe to it.
    ros::SubscribeOptions so1 =
      ros::SubscribeOptions::create<std_msgs::Float64MultiArray>(
        "/" + this->model->GetName() + "/vel_cmd",
        1000,
        boost::bind(&MotorpidPlugin::OnRosMsg1, this, _1),
        ros::VoidPtr(), &this->rosQueue);
    this->rosSub1 = this->rosNode1->subscribe(so1);

    // Spin up the queue helper thread.
    this->rosQueueThread =
      std::thread(std::bind(&MotorpidPlugin::QueueThread, this));
} // Load

void OnRosMsg1(const std_msgs::Float64MultiArray::ConstPtr& array)
{
    int i = 0;
    for(std::vector<double>::const_iterator it = array->data.begin(); it !=
        array->data.end(); ++it)
    {
           //Arr[i] = *it;
    if(i == 0){
        vel_cmd = *it * (1/rotor_vel_sim_slowdown);
        this->model->GetJointController()->SetVelocityTarget(
    this->joint1->GetScopedName(), vel_cmd);
    }else if(i == 1){
       vel_cmd = *it * (1/rotor_vel_sim_slowdown);
```

```
            this->model->GetJointController()->SetVelocityTarget(
        this->joint2->GetScopedName(), vel_cmd);
        }else if(i == 2){
            vel_cmd = *it * (1/rotor_vel_sim_slowdown);
            this->model->GetJointController()->SetVelocityTarget(
        this->joint3->GetScopedName(), vel_cmd);
        }else if(i == 3){
            vel_cmd = *it * (1/rotor_vel_sim_slowdown);
            this->model->GetJointController()->SetVelocityTarget(
        this->joint4->GetScopedName(), vel_cmd);
        }
                i++;
        }


    }




    /// \brief ROS helper function that processes messages
    void QueueThread()
    {
        static const double timeout = 0.01;

        while (this->rosNode1->ok())
        {
            this->rosQueue.callAvailable(ros::WallDuration(timeout));
        }


    }

private:

    double kp, ki, kd;
    double vel_cmd;
    double rotor_vel_sim_slowdown;
    /// \brief Pointer to the model.
    gazebo::physics::ModelPtr model;

    /// \brief Pointer to the joint.
    gazebo::physics::JointPtr joint1;
    gazebo::physics::JointPtr joint2;
    gazebo::physics::JointPtr joint3;
    gazebo::physics::JointPtr joint4;

    /// \brief A PID controller for the joint.
```

```
    gazebo::common::PID pid;

    /// \brief A node used for transport
    gazebo::transport::NodePtr node;

    /// \brief A subscriber to a named topic.
    gazebo::transport::SubscriberPtr sub;

    /// \brief A node use for ROS transport
    std::unique_ptr<ros::NodeHandle> rosNode1;

    std::mutex _cmd_mtx;

    /// \brief A ROS subscriber
    ros::Subscriber rosSub1;

    /// \brief A ROS callbackqueue that helps process messages
    ros::CallbackQueue rosQueue;

    /// \brief A thread the keeps running the rosQueue
    std::thread rosQueueThread;

};

// Tell Gazebo about this plugin, so that Gazebo can call Load on this plugin.
GZ_REGISTER_MODEL_PLUGIN(MotorpidPlugin)

#endif // ifndef _MOTORPID_PLUGIN_HH_
```

A.5 aerodynamic_plugin.cpp

```
#include <iostream>
#include <array>
#include <cmath>
#include <Eigen/Dense>
#include <random>
#include <chrono>
#include <functional>
#include <thread>
#include <mutex>
#include <math.h>
#include <ros/ros.h>
#include <boost/bind.hpp>
#include <std_msgs/Float64MultiArray.h>
#include <ignition/math.hh>
#include <ignition/math/Vector3.hh>
```

```
#include <ignition/math/Pose3.hh>
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/common/common.hh>
#include <gazebo/common/Plugin.hh>



class AerodynamicPlugin : public gazebo::ModelPlugin {
public:
  AerodynamicPlugin() : gazebo::ModelPlugin() {
    std::cout << "Starting aerodynamic_plugin" << std::endl;
  }

  virtual ~AerodynamicPlugin() {
    std::cout << "Closing aerodynamic_plugin" << std::endl;
    delete _nh;
  }

  void Load(gazebo::physics::ModelPtr parent, sdf::ElementPtr sdf) {
    _model = parent;
    link_frame = _model->GetLink("frame");

    link_v = {"propeller1", "propeller2", "propeller3", "propeller4"};
    joint_v = {"arm1propeller1", "arm2propeller2", "arm3propeller3",
        "arm4propeller4"};

    if (sdf->HasElement("updateRate")) {
      _rate = sdf->GetElement("updateRate")->Get<double>();
    } else {
      _rate = 100.0;
    }


    if (sdf->HasElement("rotorThrustCoeff")) {
      _rotor_thrust_coeff = sdf->GetElement("rotorThrustCoeff")->Get<double>();
    } else {
      _rotor_thrust_coeff = 0.0000149;
    }

    if (sdf->HasElement("rotorTorqueCoeff")) {
      _rotor_torque_coeff = sdf->GetElement("rotorTorqueCoeff")->Get<double>();
    } else {
      _rotor_torque_coeff = 0.0000000269;
    }
    if (sdf->HasElement("sim_slow")) {
            rotor_vel_sim_slowdown = sdf->GetElement("sim_slow")->Get<double>();
        } else {
            rotor_vel_sim_slowdown = 10;
```

```
    }



  if (!ros::isInitialized()) {
    int argc = 0;
    char** argv = NULL;
    ros::init(argc, argv, "du_drone", ros::init_options::NoSigintHandler);
  }

  _nh = new ros::NodeHandle("");

  _updateConnection =
      gazebo::event::Events::ConnectWorldUpdateBegin(boost::bind(&AerodynamicPlugin::onUpdate,
      this, _1));
}

void onUpdate(const gazebo::common::UpdateInfo& _info) {
  _pose_mtx.lock();
  _pose = link_frame->WorldCoGPose();
  _pose_mtx.unlock();
  sampling_time = _info.simTime.Double() - prev_sim_time;
  prev_sim_time = _info.simTime.Double();
  updateForces();
}



/*void calculatedragflap(double w, double vx, double vy, double bx, double
    by, double t, double* dflapx, double* dflapy){
  Eigen::Matrix3d Aflap;
  Aflap << -20.0*0.1, 20.0/0.1, 0.0,
          -20.0/0.1, -20.0/0.1, 0.0,
          0.0, 0.0, 0.0;

  Eigen::Matrix3d Bflap;
  Bflap << -20.0, 20.0, 0.0,
          20.0, -20.0, 0.0,
          0.0, 0.0, 0.0;

  Eigen::Vector3d vp(vx,vy,0.0);
  Eigen::Vector3d omega(bx,by,0.0);
  Eigen::Vector3d Flapvector;
  if(w < 0.01){
    *dflapx = 0.0;
    *dflapy = 0.0;
  } else{
  Flapvector = t*(Aflap*(vp/w) + Bflap*(omega/w));
```

```
  *dflapx = Flapvector[0] * (-1);
  *dflapy = Flapvector[1] * (-1);
  }


}*/

double calculateThrust(double w) {
  double thrust = _rotor_thrust_coeff * w * w;
  return thrust;
}

double calculateTorque(double w) {

  double torque = copysign(_rotor_torque_coeff * w * w, w);
  return torque;
}


void updateThrust() {

  int n = 4;
  for (int i = 0; i < n; ++i) {
    gazebo::physics::LinkPtr _link = _model->GetLink(link_v[i]);
    gazebo::physics::JointPtr _joint = _model->GetJoint(joint_v[i]);
    ignition::math::Vector3d joint_axis = _joint->GlobalAxis(0);
    selectdragvector(i);
    ignition::math::Vector3d propeller_linear_vel =
        _link->RelativeLinearVel();
    ignition::math::Vector3d drone_angular_vel =
        _model->RelativeAngularVel();
    ignition::math::Vector3d drone_lin_vel = _model->WorldLinearVel();
    double body_x_ang = drone_angular_vel.X();
    double body_y_ang = drone_angular_vel.Y();
    ignition::math::Vector3d propeller_velocity_xy;
    propeller_velocity_xy.X() = propeller_linear_vel.X();
    propeller_velocity_xy.Y() = propeller_linear_vel.Y();
    propeller_velocity_xy.Z() = 0;
    real_rotor_velocity = _joint->GetVelocity(0) * rotor_vel_sim_slowdown;
    double joint_vel = _joint->GetVelocity(0);

  if (joint_vel / (2 * M_PI) > 1 / (2 * sampling_time)){
    gzerr << "Aliasing";
    return;
  }
    double dfx, dfy; //to use as outputs of function for Dflap
    double thrust = calculateThrust(real_rotor_velocity);
    //ROS_INFO("thrust %d: %f", i,thrust);
    double torque = calculateTorque(real_rotor_velocity);
```

```
      ignition::math::Vector3d drag_torque(0, 0, torque);
      //calculatedragflap(real_rotor_velocity, vp_x, vp_y, body_x_ang,
          body_y_ang, thrust, &dfx, &dfy);
      ignition::math::Vector3d induced_drag = (-0.01 * propeller_velocity_xy);
      ignition::math::Vector3d transl_drag = (-0.0025 * propeller_velocity_xy);
      ignition::math::Vector3d profile_drag = (-0.005 * propeller_velocity_xy);
      ignition::math::Vector3d parasitic_drag = -0.003 * (drone_lin_vel.Abs()
          * drone_lin_vel);
      ignition::math::Vector3d drag_forces = induced_drag + transl_drag +
          profile_drag + parasitic_drag;


      //ROS_INFO("torque: %f", sampling_time );
      if (_link != NULL) {
        _link->AddRelativeForce(ignition::math::Vector3<double>(0, 0, thrust));
        link_frame->AddRelativeTorque(ignition::math::Vector3<double>(0, 0,
            torque));
        _link->AddRelativeForce(drag_forces);

      }
      }
      }

private:
  ros::NodeHandle* _nh;
  ros::Publisher _odom_pub;
  ros::Subscriber sub;
  std::thread _ros_thread;
  std::mutex _pose_mtx;
  std::mutex _cmd_mtx;
  std::array<std::string,4> link_v;
  std::array<std::string,4> joint_v;
  std::array<double,4> vel_vector;
  double rotor_vel_sim_slowdown;
  double real_rotor_velocity;
  double _rate;
  double _rotor_thrust_coeff;
  double _rotor_torque_coeff;
  double sampling_time;
  double prev_sim_time = 0;
  gazebo::physics::ModelPtr _model;
  gazebo::physics::LinkPtr link_frame;
  gazebo::event::ConnectionPtr _updateConnection;
  ignition::math::Pose3<double> _pose;
  ignition::math::Vector3d dragvector;
  tf::TransformBroadcaster _tf;


};
```

```
GZ_REGISTER_MODEL_PLUGIN(AerodynamicPlugin)
```

A.6 odometry_plugin.cpp

```
#include <nav_msgs/Odometry.h>
#include <ros/ros.h>
#include <cmath>
#include <random>
#include <chrono>
#include <functional>
#include <thread>
#include <mutex>
#include <ros/ros.h>
#include "geometry_msgs/Quaternion.h"
#include <nav_msgs/Odometry.h>
#include "tf/transform_datatypes.h"
#include <ignition/math.hh>
#include <ignition/math/Pose3.hh>
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <gazebo/common/common.hh>
#include "gazebo_msgs/GetModelState.h"
#include "gazebo_msgs/ModelStates.h"
#include <tf/transform_broadcaster.h>


class DronePlugin : public gazebo::ModelPlugin {
public:
  DronePlugin() : gazebo::ModelPlugin() {
    std::cout << "Starting drone_plugin" << std::endl;
  }

  virtual ~DronePlugin() {
    std::cout << "Closing drone_plugin" << std::endl;
    delete _nh;
  }

  void Load(gazebo::physics::ModelPtr parent, sdf::ElementPtr sdf) {
    _model = parent;

    if (sdf->HasElement("updateRate")) {
      _rate = sdf->GetElement("updateRate")->Get<double>();
    } else {
      _rate = 1000.0;
    }
```

```
  if (!ros::isInitialized()) {
    int argc = 0;
    char** argv = NULL;
    ros::init(argc, argv, "odometry", ros::init_options::NoSigintHandler);
  }

  _nh = new ros::NodeHandle("");
  _odom_pub = _nh->advertise<nav_msgs::Odometry>("odom", 1000);

  _ros_thread = std::thread(std::bind(&DronePlugin::rosThread, this));

  _updateConnection =
      gazebo::event::Events::ConnectWorldUpdateBegin(std::bind(&DronePlugin::onUpdate,
      this));
}

void onUpdate() {
  _pose_mtx.lock();
  _pose = _model->WorldPose();
  _pose_mtx.unlock();
}

void rosThread() {
  ros::Rate rate(_rate);
  while (ros::ok()) {
    ros::spinOnce();
    publishDronePose();
    rate.sleep();
  }
}
void publishDronePose() {
  _pose_mtx.lock();
  ignition::math::Pose3<double> pose = _pose;
  _pose_mtx.unlock();

  ignition::math::Vector3<double> rpy = pose.Rot().Euler();
  geometry_msgs::Quaternion quat;
  tf::Quaternion q(pose.Rot().X(), pose.Rot().Y(), pose.Rot().Z(),
      pose.Rot().W());
  tf::quaternionTFToMsg(q, quat);
  tf::Matrix3x3 m(q);
  double roll, pitch, yaw;
  m.getRPY(roll, pitch, yaw);
  double double_sigma_position = 2*0.0001053*0;
  double double_sigma_rpy = 2*0.0001*0;
```

```
    unsigned seed =
        std::chrono::system_clock::now().time_since_epoch().count();
    std::default_random_engine generator(seed);
    std::normal_distribution<double> distribution_pos(0.0,
        double_sigma_position);
    std::normal_distribution<double> distribution_rpy(0.0, double_sigma_rpy);

    nav_msgs::Odometry odom_msg;
    odom_msg.pose.pose.position.x = pose.Pos().X();
    odom_msg.pose.pose.position.y = pose.Pos().Y();
    odom_msg.pose.pose.position.z = pose.Pos().Z();
    odom_msg.pose.pose.orientation.x = pose.Rot().X();
    odom_msg.pose.pose.orientation.y = pose.Rot().Y();
    odom_msg.pose.pose.orientation.z = pose.Rot().Z();
    odom_msg.pose.pose.orientation.w = pose.Rot().W();

    ignition::math::Vector3d drone_angular_vel = _model->RelativeAngularVel();
    ignition::math::Vector3d drone_linear_vel = _model->WorldLinearVel();
    odom_msg.twist.twist.linear.x = drone_linear_vel.X();
    odom_msg.twist.twist.linear.y = drone_linear_vel.Y();
    odom_msg.twist.twist.linear.z = drone_linear_vel.Z();
    odom_msg.twist.twist.angular.x = drone_angular_vel.X();
    odom_msg.twist.twist.angular.y = drone_angular_vel.Y();
    odom_msg.twist.twist.angular.z = drone_angular_vel.Z();
    _odom_pub.publish(odom_msg);

    if (_publish_tf) {
      tf::Transform T;
      T.setOrigin(tf::Vector3(pose.Pos().X(), pose.Pos().Y(), pose.Pos().Z()));
      T.setRotation(tf::Quaternion(pose.Rot().X(), pose.Rot().Y(),
          pose.Rot().Z(), pose.Rot().W()));
      _tf.sendTransform(tf::StampedTransform(T, ros::Time::now(), "world",
          "drone"));
    }
  }

private:
  ros::NodeHandle* _nh;
  ros::Publisher _odom_pub;
  tf::TransformBroadcaster _tf;
  std::thread _ros_thread;
  std::mutex _pose_mtx;
  std::mutex _cmd_mtx;
  double rotor_vel_sim_slowdown;
  double _rate;
  bool _publish_tf;
  gazebo::physics::ModelPtr _model;
  gazebo::event::ConnectionPtr _updateConnection;
  ignition::math::Pose3<double> _pose;
```

```
};

GZ_REGISTER_MODEL_PLUGIN(DronePlugin)
```

Bibliography

[1] wikipedia.org. mq-4c triton. https://en.wikipedia.org/wiki/Northrop-Grumman-MQ-4C-Triton/.

[2] ESA. onyxstar-drone. https://www.esa.int/ESA-Multimedia/Images/2016/10/OnyxStar-drone/.

[3] easa.europa.org. civil drones. https://www.easa.europa.eu/en/domains/civil-drones/.

[4] defense.gov. uas. https://dod.defense.gov/UAS/.

[5] Alessandro Rizzo Margareta Stefanovic Matt J Rutherford Kimon P Valavanis Simone Martini, Serhat Sönmez. Euler-lagrange modeling and control of quadrotor uav with aerodynamic compensation. *2022 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 369–377, 2021.

[6] T. Luukkonen. "modelling and control of quadcopter,". *Independent research project in applied mathematics, Espoo*, 22:22, 2011.

[7] Z. Zuo. "trajectory tracking control design with command-filtered compensation for a quadrotor. " *IET control theory applications*, 4(11):2343–2355, 2010.

[8] Luigi Villani Bruno Siciliano, Lorenzo Sciavicco and Giuseppe Oriolo. *Robotics: modelling, planning and control*. Springer Science Business Media, 2010.

[9] R. Mahony et al. M. Bangura. "nonlinear dynamic modeling for high performance control of a quadrotor. " *IET control theory applications*, 2012.

[10] R. B. Anderson A. L'afflitto and K. Mohammadi. "an introduction to nonlinear robust control for unmanned quadrotor aircraft: how to design control algorithms for quadrotors using sliding mode control and adaptive control techniques [focus on education]. *IEEE Control Systems Magazine*, 70(1):117–133, 2013.

[11] ROS.org. about ros. https://www.ros.org/about-ros/.

[12] ROS.org. introduction,2018. http://wiki.ros.org/ROS/Introduction.

[13] ROS.org. ros brand guidelines. https://www.ros.org/press-kit/.

[14] ROS.org. core components. https://www.ros.org/core-components/.

[15] gazebosim.org why gazebo? http://gazebosim.org/.

[16] gazebosim.org logos, 2014. http://gazebosim.org/media#logos.

[17] MathWorks. simulink. https://it.mathworks.com/help/simulink/.

[18] MathWorks. Matlab documentation center.

[19] ROS.org. urdf, 2019. http://wiki.ros.org/urdf.

[20] ROS.org. xacro, ros wiki. http://wiki.ros.org/xacro.

[21] gazebosim.org. tutorial: Using gazebo plugin with ros, 2014. http://gazebosim.org/tutorials?tut=ros_gzplugins.

[22] gazebosim.org. tutorial: Using a urdf in gazebo, 2014. http://gazebosim.org/tutorials?tut=ros_urdf.

[23] Markus Achtelik Fadri Furrer, Michael Burri and Roland Siegwart. Rotors— a modular gazebo mav simulator framework. *Robot operating system (ROS), Springer*, pages 595–625, 2016.

[24] ROS.org. urdf, ros wiki. http://wiki.ros.org/urdf.