POLITECNICO DI TORINO

DIPARTIMENTO DI ELETTRONICA E TELECOMUNICAZIONI
CORSO DI LAUREA MAGISTRALE IN MECHATRONIC ENGINEERING

TESI DI LAUREA

in

INFORMATICA

Development of a simulation environment for precision agriculture applications with Unmanned Aircraft Systems

Relatore:

Chiar.mo Dr.
Stefano Primatesta

Laureando:

Alessandro Cavalli

Anno Accademico 2023-2024

Contents

Introduction

Mankind has always searched ways to predict future. In ancient time transcendent rituals were adopted by figure such as oracles and augures. In modern days, instead, a quantifiable, logical and systematic approach has been studied and mathematical models has been developed. A great example of the effectiveness in future prediction of such models is weather forecast, which has proven its reliability with the massive adoption in nautical and aeronautical sectors. Nowadays, computer and high level abstractions for model description make the creation, validation, maintaining and operation of a model based simulator way cheaper than before. Thus, new sectors, which historically hardly adopted high complexity solutions for their problems, are starting to invest in simulation technology, because the cost reduction achievable from a more precise management of resources can be important and the initial investment is no longer prohibitive. In this perspective simulators are essential because, with the increase in complexity of the adopted solutions, the difficulty in the prediction of their possible issues increases. The agricultural sector is one of the main sectors where the industrial approach used in the past decades, consisting in applying a standard solution for the whole crop, has proven to be highly sub-optimal since the natural organisms are far from being uniform in needs. Precision agriculture studies how to overcome the problem by tuning the treatments depending on the specific circumstances. This new approach requires the use of robots in order to be cost efficient, thus a development and testing environment for these robots is mandatory. There are three main categories of robots adopted in precision agriculture applications: rovers, multi-copters and fixed wing drones. Each of them has its optimal use case. The "Department of Mechanical and Aerospace Engineering" of the "Politecnico di Torino" is studying the development of a precision aerial spraying system to be used along the rows of vineyards. The long term objective of the research consists in developing a system to perform selective spraying based on the data collected in real time by the drone. In this kind of application a multi-copter appears to be the most convenient choice, since it can handle heavy payloads remaining a flexible and versatile vehicle. Therefore, this work aims to provide a multi-copter simulation environment to be used as a starting point for the future research activities of the department.

This document is structured in five chapters, each of them describes a different phase of the work. The first chapter provides an overview of the available technologies which could be used to implement the simulation environment and explains the reasons which brought to the selection of the adopted solutions. In the second chapter all the details about the setup of the simulator has been addressed. The third chapter describes the autopilot architecture, its features and the integration effort spent to achieve a modular environment. In the fourth chapter the ROS2 technology has been presented in detail as well as its use in the context of this work. The fifth and last chapter resumes the achieved performances of the complete simulation environment developed in this work.

1 Solutions available and state of the art

The increased request in low cost automation has brought to the development of many powerful Open Source solutions, which are becoming competitive in terms of features with other proprietary products. In the following paragraphs will be presented an overview of the available options and the reasons that brought these technologies to be adopted in this work.

1.1 Drone Autopilots

In the aviation sector there are many autopilot solutions certified to be compliant with the in force safety regulations. Anyway such solutions are not meant to be applied to service drones, since they are too expensive for a non mission-critical application. In this perspective a new market of budget autopilots appeared. Right now, open source solutions seems to be leader in this new sector.

1. Ardupilot

   This autopilot is the evolution of a project started in the hobby sector. It is the most cost effective solution on the market and it is compatible with the ROS2 environment and Gazebo sim. It is also compatible with many ground station software solutions.

2. PX4

   This other autopilot was created for the hobby sector as well, but the definition of the open standard "Pixhawk" and the highly modular and reliable design allowed it to be adopted by the industry sector. In particular the Pixhawk standard created a specific market of a PX4 compatible electronics where companies have precise information about what is an interface requirement and what can be modified during the design phase for minimizing cost or maximizing performances. In this way the PX4 ecosystem allows to build highly customizable systems by means of a modular and reliable plug and play hardware. As Ardupilot, PX4 is compatible with ROS2, many simulators and ground stations. For all the aforementioned reasons PX4 is the autopilot that has been chosen for this application.

1.2 Robotic middleware

The development of a robot can involve many sensors, actuators and intermediate logical elements. The complexity of handling directly all these components into a single piece of software grows exponentially, because the bare organization of software into libraries can create a modular ecosystem at compile time only, whereas at run-time all this disparate logical elements are grouped together. Furthermore, a monolithic approach impedes the development of a distributed architecture, which is known to be more reliable and easier to maintain. The solution used in industry in the last decades was to organize software into functional elements interconnected by means of a specific communication protocol. In the past, many industries had its own technology to implement this kind of architecture, but nowadays seems that the industry is converging into the use of the platform ROS. Since ROS is an open source project and in its second version, ROS2, solves many reliability issues of the ancestor, there were no doubts that it would be the middleware adopted in this application.

1.3 Robotic Simulators

There are many solutions on the market to provide the user the ability to perform simulations. Anyway, a simulator must be chosen according to the application to be tested on. So a brief overview of the available options in the robotic sector will be proposed.

1. Nvidia Isaac

   This simulator is part of the "Omniverse" platform, which is the proprietary solution provided by Nvidia. Through the plugins Nvidia Isaac ROS and Nvidia Pegasus it is possible to support communications with ROS2 and PX4. Anyway, if there are viable open source solutions, it is not advisable to use a proprietary platforms even if it is available in freemium version. In fact, using an open source product there is no exposure to vendor lock-in and there is more control on the technology in use, without third party influences.

2. Gazebo

   There are two versions of Gazebo, an older one, which is reaching the end of life in 2025 and a newer one, which is taking its place. Both versions are open source simulators compatible with ROS2 and PX4, but clearly it is wise to choose the newer one, even if not all the features are yet ready. Furthermore, ROS2 and PX4 developers, in most cases, already use this newer version as default, therefore the new version of Gazebo appeared to be the most convenient choice.

3. FlightGear

This open source simulator is not directly compatible with ROS2 because has been specifically developed to test flying vehicles, therefore it is possible to simulate sensors measurements only for the standard sensors of an aircraft. Since in this particular application it is required to simulate a depth camera sensor, FlightGear cannot be selected in this case. Anyway this simulator is worth to be mentioned because it is based on the flight dynamics model JSBSim, which has been validated by NASA, and also has an advanced weather model, which is able to precisely predict the interaction between the vehicle and the wether conditions. Therefore, in case of a fixed wing drone mounting a simple camera, this would have been the optimal choice.

2 Simulator configuration

The new version of Gazebo is constituted by an highly modular architecture based on plugins. Therefore, what is call Gazebo is the platform, i.e. a set of libraries, where plugins can share data, plus a number of official plugins to carry out the most common operations. From an architectural point of view, there is no difference between a custom plugin and an official one, this allows the user to build very complex and powerful plugins.

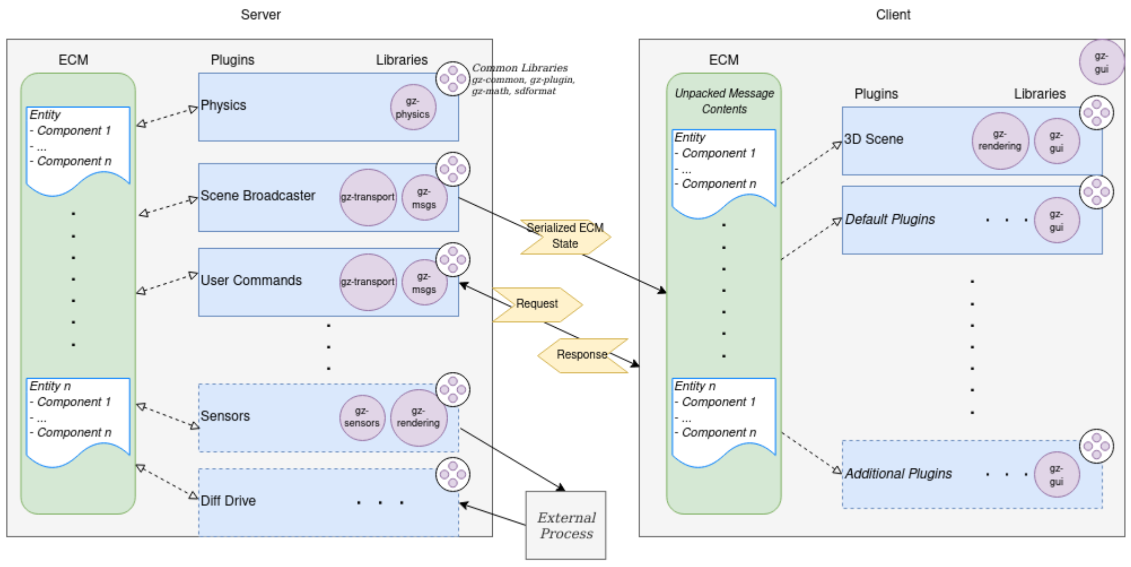


Figure 1: Gazebo architecture

In Figure 1 it is possible to see all the abstraction layers used in the Gazebo architecture [1]. First of all, there is a distinction between front-end and back-end services, allowing to decouple calculations and rendering of the simulation. Then both client and server run a number of plugins which can exchange model data through the Entity Component Manager (ECM). Finally, each plugin is based on a set of libraries which enable the inspection and modification of the elements of the ECM and the creation of data distribution services (DDS) to enable communications with other programs such as ROS2 or PX4. The primary way in which a simulation environment can be configured in Gazebo is by means of an XML description known as Simulation Description Format (SDF). In the old version of Gazebo there was also a GUI for a graphical configuration, but this feature in the new version has not been implemented yet. The SDF allows the configuration of the kinematic and dynamic parameters of robots as well as the parameters of their sensors and actuators. In the SDF it is possible to define also the world informations as weather condition, terrain shape, initial position of the robots and illumination sources. The details of the

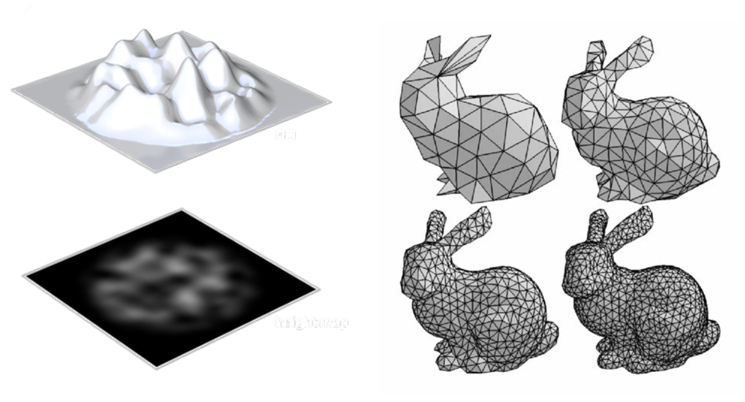
[1] Soruce of image: staging.gazebosim.org/docs/garden/architecture

configurations of the key elements of the simulation will be explained in the following sections.

2.1 Terrain

In order to configure a realistic model of the terrain in Gazebo there are two different approaches: using a heightmap or a mesh. In both cases the starting point is to collect some data of the environment of interest and converting them into a format compatible with Gazebo. In this specific case a GeoTIFF file representing a laser scan of the field was provided. This kind of data are common in the geographic sector and are classified as Digital Surface Models (DSM). In order to evaluate the performances of the two different terrain descriptions the input file has been converted into both heightmap and mesh description.

Table 1: On the right: four meshes of the same 3D object with a different level of detail. On the left an heightmap with its corresponding 3D representation.



1. Heightmaps

   An heightmap is a raster image to be interpreted as a two-dimensional square matrix where row and column indexes represent unitary displacements along x and y, while the pixels in gray scale represent the height along z. An example of such data format is shown in the bottom left figure[2] of Table 1. This kind of description is the simplified version of a DSM. In fact, a DSM file, in addition to the heightmap, contains additional metadata to precisely localize the data in the real world. Anyway Gazebo is able to take in input a square DSM, so the only modification needed to load the GeoTIFF into the simulator is

[2]Source: njb.design/en/about/10/heightmap-generation

to cut it into a square, using an open source program called GDAL. Such program is specific for converting and manipulating geographic rasters, thus it has been used also to perform a down-sampling of the square DSM, in order to reduce noise in the data and computational effort for the simulator, due to an excess of details. To be precise, Gazebo documentation mentions Digital Elevation Models (DEM) instead of DSM when explains how to load an heightmap. The difference between the two models is not in the structure of the data type but in the meaning of the data stored. In fact, a DSM file stores the raw data collected by the sensors containing vegetation, buildings and all the other elements that are on the terrain surface. In the other hand, a DEM file stores the processed data that try to reconstruct the actual terrain height filtering out the disturbance of the surface elements interposed. The implication is that a DEM is generally smoother than a DSM, so its information loss after a down-sampling is lower. This is relevant because in a precision agriculture simulator, vegetation has paramount importance, thus it must not be lost after a large down-sampling. Unfortunately, the gazebo physics engines, are not efficient in handling big and detailed heightmaps, so, in this work, meshes has been adopted to model the terrain.

2. Meshes

   A mesh is a collection of vertices, edges and faces that defines the shape of a 3D object. The faces usually consist of triangles or other simple convex polygons. This kind of data structure is used in computer graphics for rendering objects because it is very versatile and allows to model even complex shapes that show particular features only from specific perspectives, e.g. the ears of the rabbit in the right images[3] of Table 1. In fact, by construction, heightmaps cannot model features like caves into mountains, since they can only store details measurable from a "top view" perspective, achievable for instance from an airplane. Therefore, any other information related to other points of view, in that data structure, would be lost. Furthermore, meshes are computationally efficient, because do not require a fixed step sampling of the space, like heightmaps, but instead they add information only where needed, like a variable step sampling does. This allow to have a detailed model of the terrain, with a clear distinction of the vineyard lines, while keeping the binary representation lightweight. In order to convert the GeoTIFF

[3]Source: www.researchgate.net/figure/3D-mesh-triangles-with-different-resolution-3D-Modelling-for-programmers-Available-at_fig2_322096576

file into a mesh a geographic tool is required. In this work the open source program Qgis has been used. To further optimize the handling of the physics engine it is a good practice to partition the obtained mesh into several smaller meshes, this allows the collision engine to skip the collision checks for meshes far away from the mobile robot position, that clearly cannot collide with it.

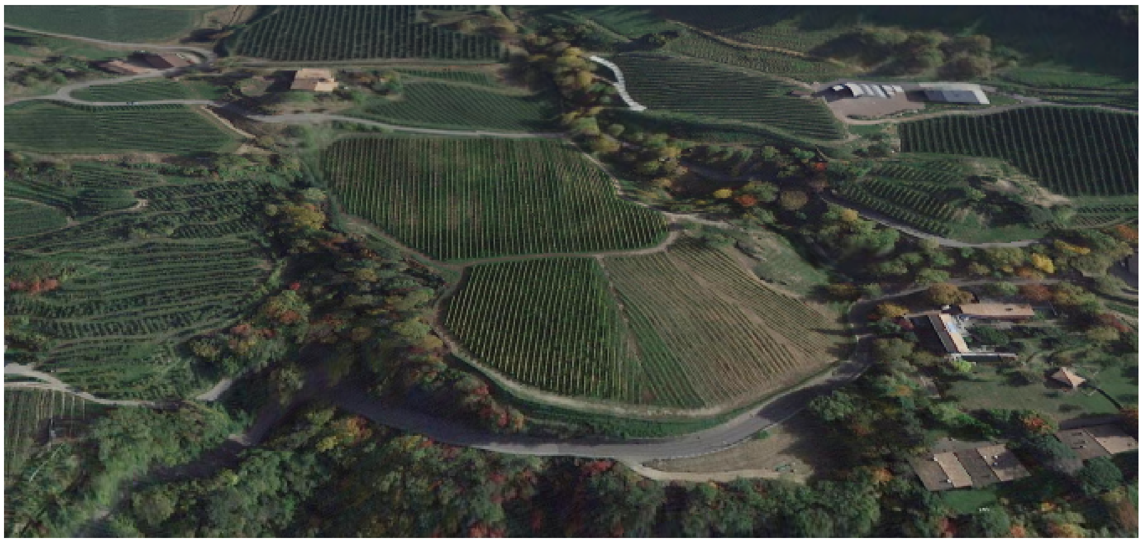


Figure 2: The Google Earth satellite image of the scansioned field

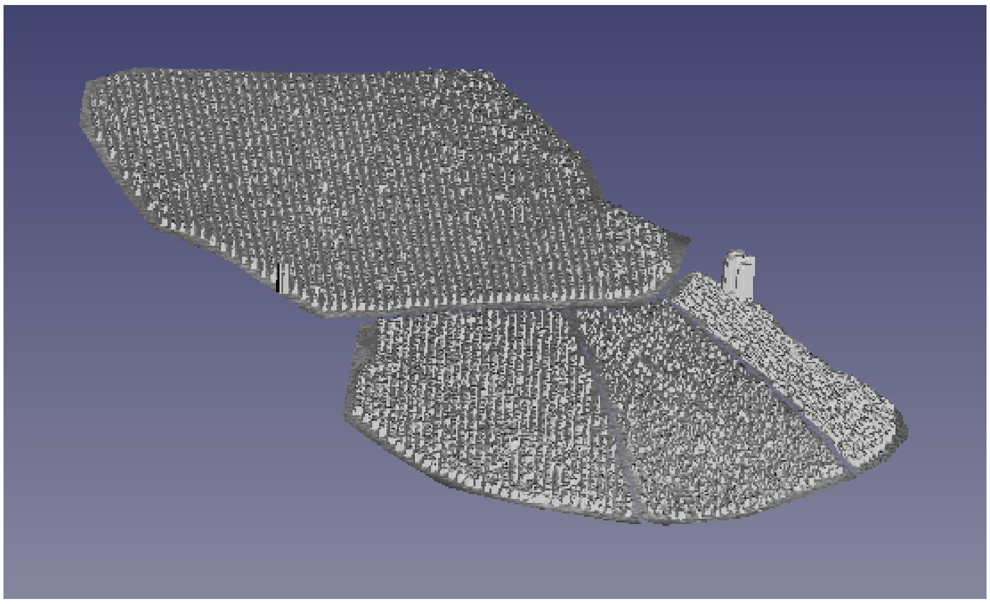


Figure 3: The Mesh representation of the scansioned field

2.2 Dynamical model

Other than the world model, the simulator needs a dynamical model to compute how the robot is interacting with the environment. The official

Gazebo plugins are meant mainly for terrestrial robotics applications, so, for some aspect relevant to this work, they are not enough to correctly describe the system. To add the missing features, two custom plugins has been developed. In the following sections will be described the functionality of each plugin adopted.

2.2.1 Official Gazebo plugins

The official Gazebo plugins used are the following:

1. Physics plugin

   The developers of this plugin designed it on the premise that there is not a single physics engine that is universally best for all simulation contexts, so, instead of implementing directly a single engine, this plugin provides an abstract interface for several physics engines. In this way the user can easily select the best engine according to its needs. In this particular case the engine adopted is Dynamic Animation and Robotic Toolkit (DART), the default physics engine of Gazebo. According to the study presented in [7], DART appears to be less precise than Bullet, the other engine officially supported by the simulator. Anyway, since the differences are in many cases negligible and the default choice is usually better documented, DART was selected to be the physics engine of this work.

2. Multi-copter motor plugin

   In order to make the drone able to fly, it is necessary to model an actuation system, which is the purpose of multi-copter motor plugin. As the name suggests, for sake of simplicity, instead of modeling with two separate generic plugins the motor torque and the rotating propeller fluid dynamic forces, this plugin is specific for the standard configuration of a real actuation system in a multi-copter. In particular it is assumed that the motor is an ideal velocity controlled motor and that the propellers' fluid dynamic forces can be described by means of the standard quadratic relation of aerodynamics. In this way, according to [6], thrust, drag and moment of a propeller are quantified using the following formulas:

$$\begin{aligned} Thrust_z &= K_{motor} \cdot \omega_{motor}^2 \\ Drag_z &= K_{drag} \cdot \omega_{motor} \cdot V_{apparentWind}^{\perp} \\ Moment_x &= C_{roll} \cdot \omega_{motor} \cdot V_{apparentWind}^{\perp} + C_{m_{drag}} \cdot Thrust_z \end{aligned}$$

In the first expression the thrust is function of the rotor angular velocity because the lift has been integrated along the propeller length and all the information is converted into the rotational domain by means of the constant K_{motor}. The second expression represents the force known as H-force or rotor drag in the helicopter literature, where what here is named K_{drag} is generally called "rotor inflow ratio" [4]. The H-force can be seen as the resistance produced by the propeller disk on the normal component of the apparent wind, which is proportional to the angular velocity. The last formula accounts for the rotational momenta described in [6]. Finally, since the drone motors are usually brush-less ones driven by Electronic Speed Controllers (ESC), it is reasonable to consider the rotor angular velocity a known quantity to be used as input variable for the previous formulas. All these formulas are valid in the "near hovering" assumption, where holds that the velocity of the drone is lower than a seventh of the propeller tip speed. An other important assumption which greatly simplifies the calculations requests a null rotor blade flapping. This second assumption is equivalent of stating that the propeller can be modeled as a perfectly rigid body.

3. Sensor plugins

   In order to test the control algorithms some quantities must be extracted from the simulation and published in dedicated topics. The set of plugins considered here enable this extraction as well as the simulation of measurement difficulties due to the noise or the installation of a sensor in a sub-optimal position e.g. a gyroscope mounted far from the center of mass. To better model the behavior of a real sensor it is possible to configure also the sampling rate and the measurement accuracy of the sensor, introducing for instance a quantization. For the depth-camera there are available many advanced configurations including depth of field and resolution.

2.2.2 Custom plugins

The official Gazebo plugins has been initially developed by Open Robotics with the purpose of providing the users a general set of tools mainly focused on classical robotics applications. With the increase in popularity of the simulator, some other official plugins has been developed to enable simulation of physical phenomena related to underwater or aerial activities. Unfortunately, due to the complexity of the fluid-dynamic relations, these plugins could not be too much generic and in this specific case they resulted not applicable. Therefore, the development of some custom plugin become mandatory. Anyway, in Appendix B some plugins, related to fluid handling, has been analyzed in detail and the specific reasons which led to this conclusion has been presented. Before starting describing in detail the actual plugins implemented, it is worth to describe the general structure of a plugin. A plugin is a C++ derived class of the super class System, included into the Gazebo library. The class System provides four abstract methods which can be implemented by a plugin:

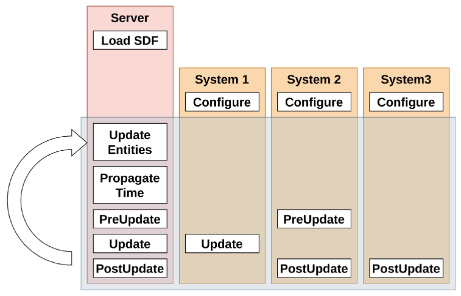


Figure 4: The generic architecture of a Gazebo plugin

- Configure
- PreUpdate
- Update
- PostUpdate

Each method is invoked in a specific point of the simulation execution, enabling different features in the different plugins. In Figure 4 it is represented[4] the timing of each method call. It is possible to see that the Configure method is invoked once, to initialize the plugin class with the parameters passed from SDF. Instead, the PreUpdate, Update and PostUpdate methods are invoked in each simulation step. In particular the PreUpdate has reading and writing privileges on the model parameters, allowing the plugin to apply forces and momenta; while the PostUpdate method has only reading privileges, allowing to extract model data and make them available to third programs. The Update method is mainly

[4]Source of image: download.ros.org/downloads/roscon/2022/ROS%202%20and%20Gazebo%20Integration%20Best%20Practices.pdf

used by physics engines to update the model state, so it is not very interesting in the perspective of developing a standard plugin. The custom plugins developed are the following:

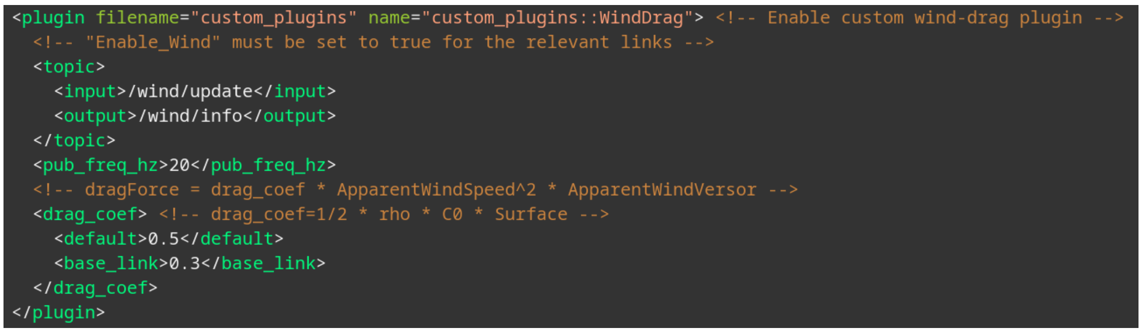
1. Wind-drag

   Using the aforementioned official plugins, the effect of the wind is not yet taken into account. The primary purpose of this plugin is to overcome this issue applying a drag force to the simulation objects, such as the drone frame, according to the well known expression:

$$\begin{aligned} F_{drag} &= \frac{1}{2} \cdot \rho \cdot surface \cdot C_{D_0} \cdot |apparentWind|^2 \cdot dragDirection \\ &= coef_{drag} \cdot |apparentWind|^2 \cdot dragDirection \end{aligned}$$

   It is important to note that the linearized drag coefficient $C_D = C_{D_0} + C_{D_\alpha} \cdot \alpha$, where α is the angle of attack of the drone's body, is approximated with a constant. This simplification greatly reduces the complexity of the plugin and its computational burden, still remaining reasonable in this specific case, since the interest is limited to predict the disturbance attenuation of the autopilot in the worst case scenario, where the drag coefficient is maximum.

   With this new feature active in the simulator, become interesting to vary the wind speed and direction according to a known law. In order to do so, the plugin was made capable of subscribing a topic where it takes in input the wind velocity vector. For debugging purposes, the relevant quantities computed by the plugin has been made available to ROS2 by means of some publishing nodes.



```
<plugin filename="custom_plugins" name="custom_plugins::WindDrag"> <!-- Enable custom wind-drag plugin -->
  <!-- "Enable_Wind" must be set to true for the relevant links -->
  <topic>
    <input>/wind/update</input>
    <output>/wind/info</output>
  </topic>
  <pub_freq_hz>20</pub_freq_hz>
  <!-- dragForce = drag_coef * ApparentWindSpeed^2 * ApparentWindVersor -->
  <drag_coef> <!-- drag_coef=1/2 * rho * C0 * Surface -->
    <default>0.5</default>
    <base_link>0.3</base_link>
  </drag_coef>
</plugin>
```

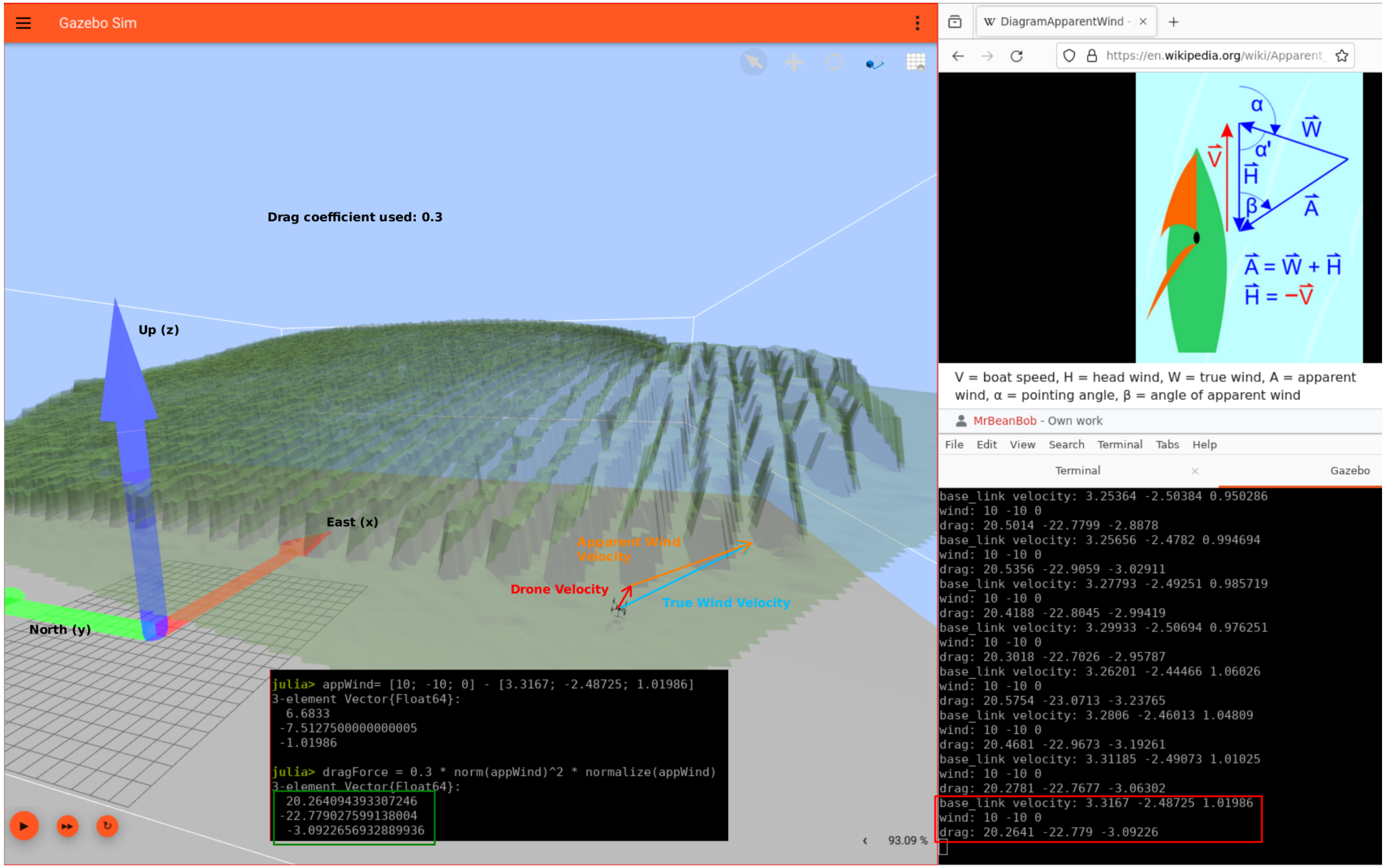
Figure 5: SDF used for the configuration of the Wind-Drag plugin

   Finally, an API for the plugin configuration through SDF has been developed. This interface allows to configure the topic names and a different drag coefficient for each link. The SDF configuration used in this work is shown in Figure 5.

In Figure 6 it is possible to check the correct application of the designed mathematical relations. In particular the vectors of the true wind and drone velocity are explicited so that the apparent wind velocity can be derived. Then, using those data, the drag force has been computed using an external program and compared with the result of the plugin. In the red box are shown the plugin data, while in the green box is shown the result of the computation performed with the external program. Since the two results are very close, it is clear that the differences can be addressed only to different approximations in the data representation of the two used programs. In chapter 5 is performed a more detailed analysis of the plugin behavior.

2. Sloshing-force

   An other characteristic that cannot be directly handled by the official plugins is a variable mass. Furthermore, a spraying drone not only presents a variable mass, but also some kind of sloshing effect. In fact, the fluid inside the tank, that gradually decreases over time as the spraying activity continues, produces forces that cannot be quantified using only the theory applicable for inertial and gravitational forces. To overcome these issues this second plugin was developed. The base requirement for the plugin is to provide a variable mass to the simulator, to allow the correct sizing of the motors and to evaluate the performances of the autopilot in controlling a drone with variable inertia. There are two possible approaches to achieve this requirement: modifying the mass component of the tank link and let the physics engine apply the gravitational and inertial forces; or storing the information related to the tank status in a class variable and from that variable directly compute and apply the forces to the model. This second approach was adopted since it provides more flexibility and paves the way for the modeling of the sloshing effect force, whose description is outside the scope of this work but nevertheless will be relevant to be added to the simulator in future updates. The mechanism which enables the update of the tank status is based on the subscription to a topic containing the flow rate of the spraying system outlet. With this information a backward rectangular integration is performed and the new tank level is computed. Furthermore, a dedicated node publishes the topic to drive a particle emitter plugin attached to the drone to enable a visual feed-


Gazebo Sim
Drag coefficient used: 0.3
Up (z)
East (x)
North (y)
Apparent Wind Velocity
Drone Velocity
True Wind Velocity
93.09 %
DiagramApparentWind -
https://en.wikipedia.org/wiki/Apparent
α
α'
β
$\vec{W}$
$\vec{V}$
$\vec{H}$
$\vec{A}$
$\vec{A} = \vec{W} + \vec{H}$
$\vec{H} = -\vec{V}$
V = boat speed, H = head wind, W = true wind, A = apparent wind, α = pointing angle, β = angle of apparent wind
MrBeanBob - Own work
File Edit View Search Terminal Tabs Help
Terminal
Gazebo

```
julia> appWind= [10; -10; 0] - [3.3167; -2.48725; 1.01986]
3-element Vector{Float64}:
  6.6833
 -7.5127500000000005
 -1.01986

julia> dragForce = 0.3 * norm(appWind)^2 * normalize(appWind)
3-element Vector{Float64}:
  20.264094393307246
 -22.779027599138004
  -3.0922656932889936
```

```
base_link velocity: 3.25364 -2.50384 0.950286
wind: 10 -10 0
drag: 20.5014 -22.7799 -2.8878
base_link velocity: 3.25656 -2.4782 0.994694
wind: 10 -10 0
drag: 20.5356 -22.9059 -3.02911
base_link velocity: 3.27793 -2.49251 0.985719
wind: 10 -10 0
drag: 20.4188 -22.8045 -2.99419
base_link velocity: 3.29933 -2.50694 0.976251
wind: 10 -10 0
drag: 20.3018 -22.7026 -2.95787
base_link velocity: 3.26201 -2.44466 1.06026
wind: 10 -10 0
drag: 20.5754 -23.0713 -3.23765
base_link velocity: 3.2806 -2.46013 1.04809
wind: 10 -10 0
drag: 20.4681 -22.9673 -3.19261
base_link velocity: 3.31185 -2.49073 1.01025
wind: 10 -10 0
drag: 20.2781 -22.7677 -3.06302
base_link velocity: 3.3167 -2.48725 1.01986
wind: 10 -10 0
drag: 20.2641 -22.779 -3.09226
```

Figure 6: Validation of the drag force applied on the drone due to an apparent wind

back of the spraying activity, shown in Figure 7. Also in this plugin, for debugging purposes, the computed relevant quantities has been made available to ROS2 by means of some publishing nodes. Finally, an API has been developed to allow the user to configure the plugin using the SDF description. The configuration adopted in this work is shown in Figure 8.

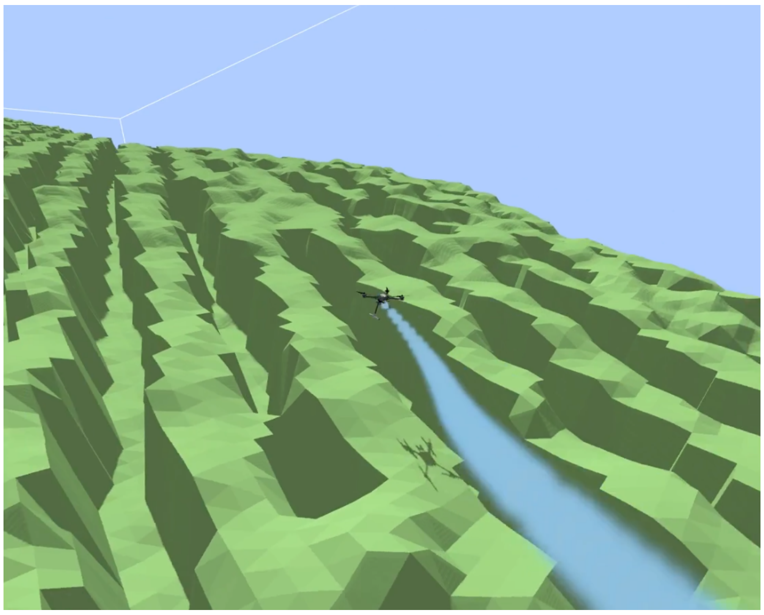
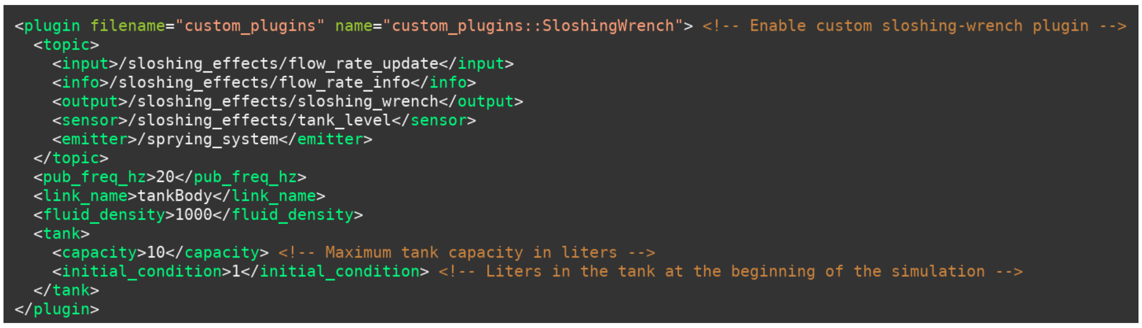


Figure 7: Spraying activity visualization



```xml
<plugin filename="custom_plugins" name="custom_plugins::SloshingWrench"> <!-- Enable custom sloshing-wrench plugin -->
  <topic>
    <input>/sloshing_effects/flow_rate_update</input>
    <info>/sloshing_effects/flow_rate_info</info>
    <output>/sloshing_effects/sloshing_wrench</output>
    <sensor>/sloshing_effects/tank_level</sensor>
    <emitter>/sprying_system</emitter>
  </topic>
  <pub_freq_hz>20</pub_freq_hz>
  <link_name>tankBody</link_name>
  <fluid_density>1000</fluid_density>
  <tank>
    <capacity>10</capacity> <!-- Maximum tank capacity in liters -->
    <initial_condition>1</initial_condition> <!-- Liters in the tank at the beginning of the simulation -->
  </tank>
</plugin>
```

Figure 8: SDF used for the configuration of the Sloshing-Effects plugin

2.3 GUI plugins

The Gazebo front-end is composed by a number of plugins that allows the user to interact with the simulation by visualizing data and sending inputs. A GUI plugin can be visible or hidden, depending on its purpose. The hidden plugins are used to enable new features in the front-end that will be used by other visible plugins to provide their functionalities. The visible plugins can be displayed under the form of windows, toolbars or buttons to provide the actual user interface of the client. There are three main category of plugins used in this work that will be described in detail the in the following sections. The final form for the GUI of the simulator is shown in the left side window in Figure 6. All the GUI configurations can be stored directly into the SDF configuration of the single world or in a separated configuration file which can be shared with many different worlds.

1. Plugins for simulation visualization

   In the Gazebo client the graphical visualization of the world is not mandatory, it must be enabled with the Minimal-Scene plugin. In the plugin configuration it is possible to select the 3D graphics engine to be used to render the scene. The only engine supported is OGRE, which is an open source engine available in two version: OGRE and OGRE Next. OGRE Next, also identified with OGRE2, is the engine adopted in this work, since is much faster in rendering and has an improved ray tracing approximation, which leads to a more natural illumination of the scenes. In order to complete the simulation visualization it is necessary to activate:

   - The hidden plugin GzSceneManager, which enables the updating of the 3D model as the simulation proceeds
   - The WorldControl plugin, which provides play and pause buttons to control the simulation status from the client window
   - The WorldStats plugin, which provides a small dialog to resume the simulation information such as the time elapsed and the real time factor.

2. Plugins for world navigation and model inspection

   With the aforementioned set of plugins is enabled only the visualization of the scene from a fixed perspective. In order to add the navigation functionalities, expected from a modern graphical interface of a 3D environment, some other plugins must be used. In particular: the InteractiveViewControl hidden plugin enables mouse navigation; the SelectEntities hidden plugin enables selection of a model; the EntityContextMenuPlugin visible plugin provides a pop-up menu, callable by right clicking on a model, with some action such as "show [reference] frames" or "follow [with the camera]". Anyway, the EntityContextMenuPlugin requires the hidden plugins VisualizationCapabilities and CameraTracking to enable the "show frames" and "follow" menu entries, respectively. Therefore both the hidden plugins have been added. For debugging purposes the EntityTree visible plugin has been used extensively because it allows to inspect the world entities in a convenient manner. Anyway, it has been removed from the default plugins loaded at the startup of the Gazebo client because its window is too cluttered to be always active. Finally, the Plotting visible plugin enables the visualization of numerical topics in a 2D Cartesian graph, which, for the same reason of the EntityTree, has been disabled by default.

3. Plugins for world and model editing

   In the world creation phase it is handy to have some tool to edit the model in a graphical way. In fact, Gazebo can export a graphically configured world into an SDF file. This is not the recommended procedure to create SDF configurations, but it is helpful to perform some activity that result inconvenient in a text based procedure. For instance, tuning the pose of a model is easier in a graphical environment which feedbacks the current position and orientation, enabling to modify them in an interactive way. In fact, the graphical interface allows to spawn and translate models using drag and drop and rotating them by means of gimbal circles. This is more convenient than directly tuning the coordinates and the quaternion in the SDF file. When the pose is determined by means of the graphical procedure, it can be pasted into the original SDF file configured with the text based procedure. In this way it is possible to take advantage of the benefits of both the procedures. The TransformControl visual plugin enables the aforementioned features by adding a toolbar with the tools for translations and rotations of models. Similarly, the Spawn hidden plugin enables the spawn of new models, while the CopyPaste hidden plugin activates copy and paste capabilities in the menu entry of the EntityContextMenuPlugin.

3 Autopilot configuration

In a physical drone an autopilot is constituted by a dedicated electronics programmed with an autopilot software. In this work, the platform chosen is PX4, which consists in a Pixhawk compliant hardware programmed with the PX4 software. In simulation stages the PX4 software can be run on a standard computer without the dedicated hardware, using the data streams coming from a simulator as sensor readings. Since this work consists in a simulation setup, the software architecture only has been used and therefore analyzed. All the high level drone control logic, such as the ROS2 nodes, is executed in a separated board, which the official documentation calls companion computer.

3.1 PX4 architecture

The PX4 architecture has been designed by its developers according to the reactive manifesto[5], which is a guideline to achieve responsiveness, reliability, scalability and modularity. In Figure 9 it is shown the high level architecture[6] of the PX4. The first important thing to note in the figure is the subdivision in functional elements, which are very similar to the ROS nodes. In fact, the autopilot, to be compliant with the reactive design, has to implement a middleware with publisher-subscriber communication. Here, instead of ROS2, Micro Object Request Broker (uORB) has been used, which is a shared memory based bus, optimized for embedded applications. The advantage of the uORB in this context consists in its narrowed scope to a single board communication, that makes it smaller and more computationally efficient with respect to a more general middleware like ROS2. The uORB bus has been adopted for all the internal communications between sensors, processor, memory and interfaces. For what concerns external communication a different protocol must be used, since the access to the shared memory is no longer available. In this case the PX4 developers have chosen MAVLink and uXRCE-DDS. MAVLink is a lightweight protocol that was designed for efficiently sending messages over unreliable low-bandwidth radio links and it is mainly used by the ground station to send commands to the autopilot. Instead, the uXRCE-DDS is the protocol used to bridge the uORB topics with the ROS2 environment. In order to provide time sensitive services, PX4 assumes to be running on a real time operating system such as the Open Source Apache NuttX, anyway using the PX4 SIL binaries it is possible to test it on general purpose

[5]www.reactivemanifesto.org

[6]Source of image: docs.px4.io/main/en/concept/architecture.html

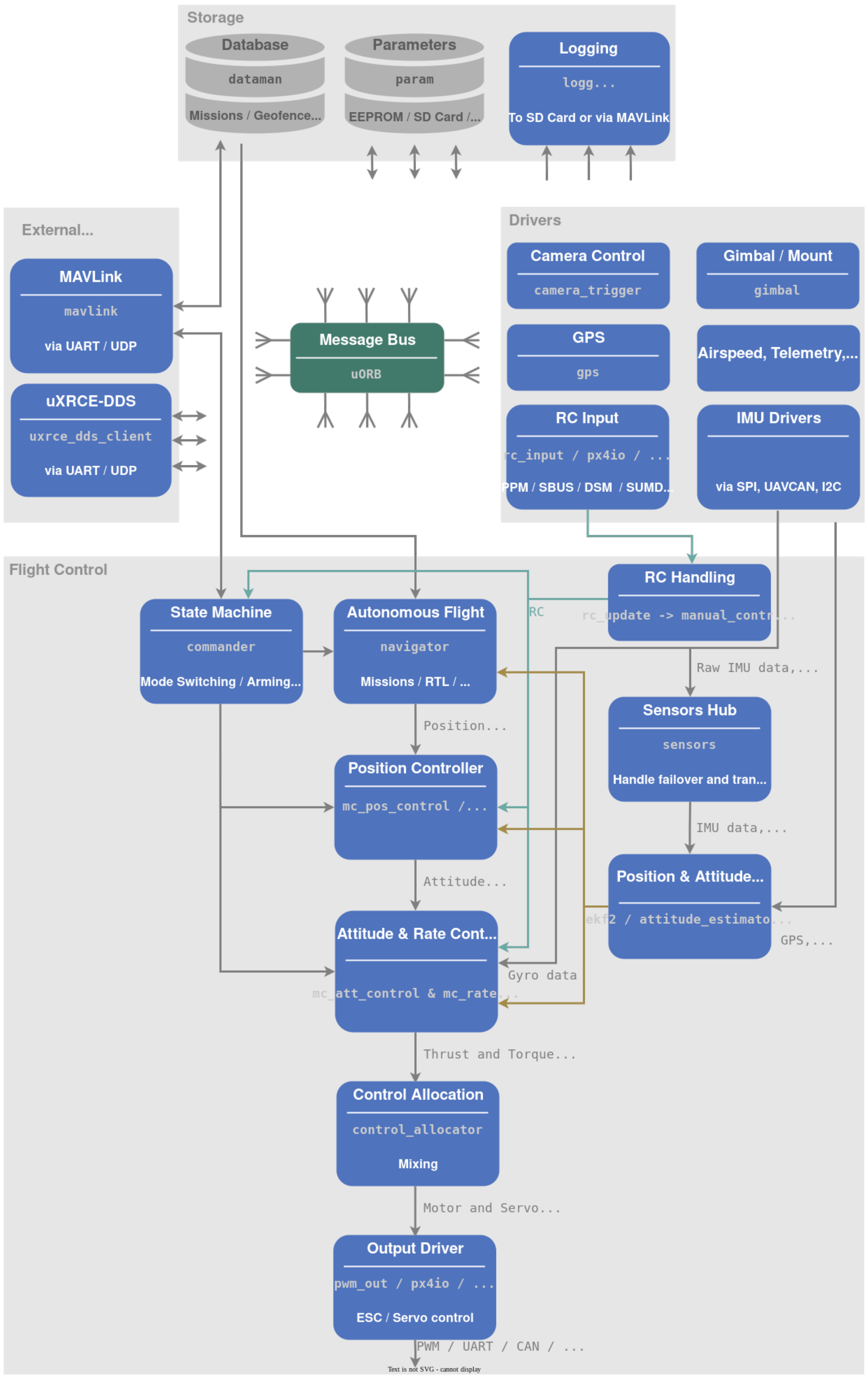


Figure 9: High level architecture of PX4 autopilot

operating systems, such the standard GNU/Linux distributions.

3.2 Safety functionalities

An autopilot runs in a separated hardware from the actual controller due to safety reasons. In fact this separation decouples the safety functionalities from the operational ones, providing the user more freedom in the development and allowing more complex implementations of the control algorithm. In particular, the separation allows the autopilot to handle a failure and transit to a safe state even in case an exception in the controller's code arise or a glitch in the companion computer occurs. For instance, it is possible to program the autopilot to have an operational space limited by a well defined geographic boundary, this is called geofence; in case of an high level controller in the companion board requests a perimeter violation for any reason, the autopilot can autonomously switch to a safe state to handle the exception, such as hovering in the last allowed position, landing in place or returning home.

3.3 PX4 operational functionalities

The main purpose of the PX4 autopilot is to abstract the details on how the actuators should be driven in order to achieve a specific task. Since the autopilot operates primarily as a controller, the achievable tasks are determined by the sensors available on board and are based on the control of the position and/or the velocity of the drone, considered in the generalized sense of both the linear and angular quantities. Furthermore, the high level references for the autopilot can be provided via a radio channel, stored directly on the memory on board or computed in real time by the companion computer. In this work, all the commands has been sent using the Open Source ground station software QGroundControl. In order to provide the best noise rejection capabilities, the PX4 adopt one or more sensor fusions of the available physical sensors. Each of them estimates, by means of the Extended Kalman Filter, not only the global position and orientation of the drone and its generalized velocity, but also the biases on the inertial and magnetic sensors and the wind velocity. The PX4 autopilot can operate on a broad family of vehicles ranging from multi-copters to rovers passing through planes, so a vehicle description is required to provide it the mechanical parameters needed to compute the correct control action and predict its effects. This description, within the PX4 environment, is called airframe. There are many official airframes that can be used as starting point for the customization of the description of the specific drone in use. In this work a simple custom airframe has been configured according to the modeled drone setup.

3.4 Setup of the SIL environment

The Software In the Loop (SIL) testing approach is based on the idea that the code implementing the control algorithm tested in simulation must be the very same code to be deployed on field. This allows to identify logical problems peculiar to the control algorithm as well as implementation problems due to the specific infrastructure used. The PX4 repository provides a build system which allows to compile the hardware's firmware as well as the software in the loop binaries and launch them with an already connected simulator. This is very user friendly for simple projects, where a good handling of the complexity is not needed. In fact, to configure the simulator's world it is just necessary to replace the default world configuration in the PX4 repository and re-run the build system command. Anyway it is not a good architecture the one that divides resources which are logically related. Thus, some effort has been spent to manually connect the two pieces of software by means of the additional configurations available in the PX4 SIL binaries, without relying on the PX4 build system, in order to gain more control on the organization of the resources. In particular, the solution consisted in launching a Gazebo world with the /clock topic running, i.e. with the simulation not in pause, and then launching the PX4 compiled as bare SIL executable configured to connect to a specific model within the running simulation. An other advantage of this manual setup is that the drone model is not spawn by the PX4 script, as in the default case, but instead an existing model is linked to the PX4 SIL instance. This allows to test separately the simulation environment alone or the simulation environment connected with the autopilot. To get the connection between a running Gazebo world and a PX4 SIL instance it is necessary to configure in the PX4, by means of an appropriate environment variable, the name of the model which will represent the drone into the simulator and to set in Gazebo the sensor topics to the specific names expected by the PX4/Gazebo bridge. In this way the PX4/Gazebo bridge embedded in the SIL binaries can translate all the sensor topics from Gazebo to the uORB middleware and all the actuation commands from the middleware back to the simulator. The only sensor which it is not possible to bridge is the Global Navigation Satellite System (GNSS) receiver, which remains simulated within the PX4 environment. This slightly reduces the modularity of overall system since, changing the simulated world, the PX4 cannot localize itself without an explicit configuration in the airframe. Furthermore, loosing the control on the GNSS model also the control on the simulation of its noise is lost, which reduces the possible test activities that the simulator can perform. Finally, all the resources has been stored in a dedicated repository and a hierarchy of ROS2 launch files has been created to startup the environment in the most neat and tidy way. The details of the structure of this repository will be discussed in the 4.1.3 paragraph.

4 ROS2

The Robotic Operating System 2 (ROS2) is a set of libraries that implements an abstract interface which provides the core functionalities of a Data Distribution System (DDS) while simplifies many details in order to keep the configuration effort low as in the ROS1 interface. A DDS is an open international middleware standard issued by the Object Management Group (OMG) addressing publish-subscribe communications for real-time and embedded systems. This standard has been adopted in many sectors among which railway, aerospace, military and financial [10].

The main features that ROS2 inherited from DDS are the node discovery functionalities in a single host, Local Area Network (LAN) or Virtual Private Network (VPN) and the communication layer functionalities. The actual implementation of these features depends on the DDS vendor adopted. ROS2 has been designed to be vendor agnostic, in order to achieve interoperability with existing DDS. The default DDS adopted is eProsima Fast DDS, which is an open source implementation of the standard. The default node discovery functionality in ROS2 has a distributed architecture, i.e. each node periodically advertise its presence and the services that makes available on the network through a multicast IP address and waits the acknowledgment from every other node. This approach is very convenient when the number of nodes is low because it requires zero setup and it is intrinsically fault tolerant. It is worth to point out that the zero setup property holds in all the networks where the multicast traffic is supported[7]. Anyway, the distributed node discovery functionality generates lots of traffic only related to the discovery of new nodes. To overcome this problem some DDS, like Fast DDS, support a centralized architecture which is based on a backbone of discovery servers that regulates the advertisement of the new nodes. Clearly, in this settings additional configuration effort is required. In Figure 10 it is possible to see a comparison[8] between the number of packets produced by the default decentralized ar-

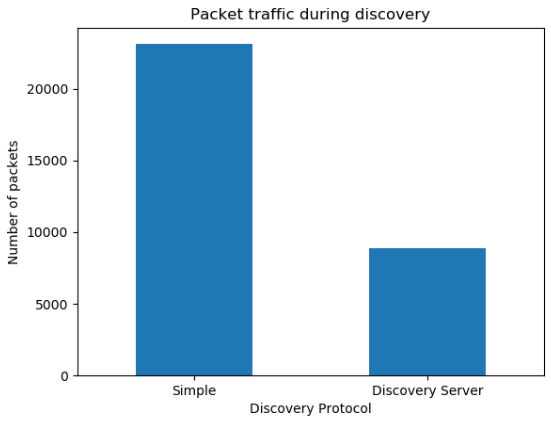


Figure 10: Traffic comparison between centralized and decentralized architecture.

[7]Some VPN do not have this support enabled by default

[8]Source of data: docs.ros.org/en/humble/Tutorials/Advanced/Discovery-Server/Discovery-Server.html#setup-discovery-server

chitecture and the centralized one. The data used in the comparison are related to the traffic of both discovery and data packets of one publisher and fifty listeners, collected for fifteen seconds. It is clear that the traffic reduction is dramatic in the centralized case. Furthermore, it is worth to notice that, after the discovery phase, the communication between nodes remains peer-to-peer also in the centralized architecture, therefore a centralized system with limited resources but with some redundancy at the discovery server level still remains reasonably fault tolerant. The default communication layer of ROS2 is based on a shared memory protocol for the intra-process communications and the UDP/IP stack for the others. Since the UDP protocol can support only a "best effort" quality of service, the DDS standard implemented a custom transport protocol on top of it, enabling the user to configure the desired behavior. Furthermore, in order to ensure secure communication, an official ROS2 package enables the DDS feature of encrypting the whole nodes' traffic. Finally, in case of many ROS2 applications must be run on the same network, to avoid mixing the nodes together, the DDS provide a mechanism of logical separation in domains. Therefore, the nodes will discover each other only in case the identifier of the logical network, called domain id, is the same.

4.1 ROS2 abstraction

Since the naming convention and the mechanisms available in a DDS are similar, but still different from the ones used by ROS2, in this section will be proposed the ROS2 abstraction regardless the actual implementation in the underlying DDS.

4.1.1 Patterns of communication

The available communication patterns in ROS2 are handled by default by asynchronous non blocking calls based on:

1. Topics

   A topic is an unidirectional data stream exiting from a single node, named publisher, and entering in multiple other nodes, named subscribers. It enables the publisher-subscriber communication which is the most flexible pattern available in ROS2, since it allows to handle a generic data stream which can be produced or utilized by both low level nodes, such as sensors and actuators, and high level nodes such as controllers, estimators and filters.

2. Services

A service is a single request-response interaction between nodes. It enables the client-server communication which is usually more suitable for high level task such as requesting quick calculations or for sending acknowledged commands.

3. Actions

An action is the most complex node interaction in ROS2. It is the generalization of the client-server communication and allows to handle the difficulties related to long task executions. In fact, in case of long tasks, the bare request-response pattern is not flexible enough to manage the various needs related to the changes due to the passage of time. Furthermore, long tasks are more likely to result in race conditions so a resolution mechanism becomes necessary.

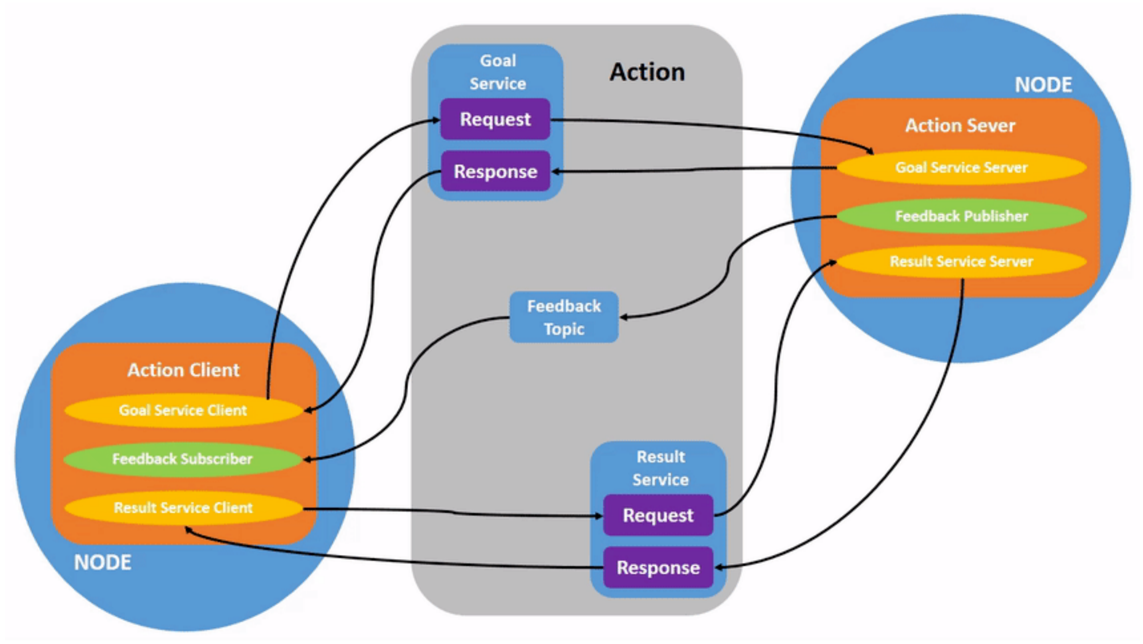


Figure 11: The logical elements composing a ROS2 action

To overcome these problems, actions provides the user an unified interface to: set the task objective, receive updates on the task status and receive the task results. This complex behavior is implemented in ROS2 by means of a combination of services and topics that can be resumed, with some simplifications, in the diagram[9] in Figure 11. During the setting of the task objective the action server applies a user configurable policy to solve the race conditions. Finally, if the task objective become outdated, while waiting the end of the execution, the action client has the possibility to request a cancellation.

[9]Source of image: docs.ros.org/en/foxy/_images/Action-SingleActionClient.gif

4.1.2 Node features

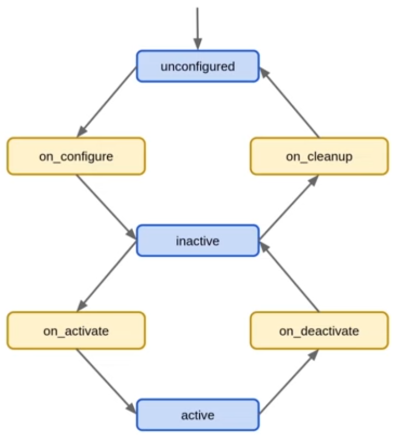


Figure 12: State machine of a managed node

In ROS1 a node is a unit of computation in a ROS graph performing a single logical operation. ROS2 inherits the concept of node of ROS1 and enhances its capabilities in terms of decentralization and reliability. In fact the ROS1 nodes depended on the parameter server on the rosmaster to store their dynamical configurations. In this second version of the library each node embeds a parameter server which hosts the parameters relevant for its configuration. Furthermore, the concept of "managed node" has been introduced to ensure that all components have been instantiated correctly before any component begin executing its behavior. This feature will also allow nodes to be restarted, reconfigured or replaced on-line. The states of a node are shown in the blue boxes of Figure 12. The only state that has been neglected for sake of simplicity from the figure is the "Finalized" state, that can be reached from any state in case of node shutdown or fatal error. The entry point of the state machine[10] after the node creation is "Unconfigured", to trigger the subsequent transitions it is mandatory to have received an explicit signal from an other process, such as the ROS2 command line tool or an other node. The transition success is regulated by the related implementation of the abstract callback in the node class. All the abstract callbacks are shown in the yellow boxes of Figure 12. Finally, in order to optimize the communications, it is possible to make minimal changes to a node definition and add a predisposition to be composed into a single process in a launch file definition[11]. This allows to adopt zero-copy intra-process communications, based on shared pointers.

4.1.3 Package structure

A ROS2 application is structured in packages which follow a set of best practices[12] that standardize their form in order to create a consistent ecosystem of packages to improve readability and modularity. In particular it has a fixed organization in folders typical of a build system such as

[10]Source of image: www.youtube.com/watch?v=_GXHBP5sA70

[11]docs.ros.org/en/rolling/Tutorials/Intermediate/Writing-a-Composable-Node.html

[12]docs.ros.org/en/rolling/The-ROS2-Project/Contributing/Developer-Guide.html#package-layout

CMake. Other than the source code folders, a package contains the launch file declarations as well as the URDF models and Rviz2 configurations.

The root folder of the project, in the ROS ecosystem, is called workspace and it contains an src, a build and an install folder. As usual, the src folder contains all the source code of the packages which compose the application while the build and install folders contains all the compiled resources, organized in such a way that the build system can avoid to re-build them in case their corresponding source code has not been changed. Furthermore, the install folder contains the scripts that allows to make available the final executable of the application to the standard ROS environment. This process is not a standard installation with a copy of the executable in the standard search path of the operating system, rather it is a volatile change in the shell's search path environment variable. The advantage of this approach is based on the improved modularity of the resulting installation. In fact, it is possible to install on the same system, in different times, multiple sets of packages without any conflict. Furthermore, this installation type, based on the modification of the search path variable, enables to port at the package level the concept of variables shadowing, present in high level programming languages. In particular, this concept in the ROS2 abstraction is called overlay and it is particularly useful to replace a subset of packages provided by a ROS2 application without changing its source code or launch files definition.

In order to automate the build of each dependency, ROS2 provides colcon, a build tool which determines the dependency graph and invokes the specific build system for each package in topological order. This build tool can compile both ROS1 and ROS2 packages as well as Gazebo plugins [9]. The main build system adopted in ROS2 packages are ament_cmake and python setuptools.

4.2 Implemented nodes

In this work the ROS2 environment has been adopted to:

1. Start the simulation environment

   The simulation environment depends on a number of element that must be started in the correct order. This requirement is taken into account by the launch file hierarchy shown in Figure 13. Anyway, the same functionality can be provided by a single launch file. The reason why a more complex architecture has been used, is that it enables to have a more granular control on the nodes startup during the debug phase. In fact, for some activities, such as unit testing, it is useful to activate the least possible nodes to check the performances of a module which provides a single logical functionality.

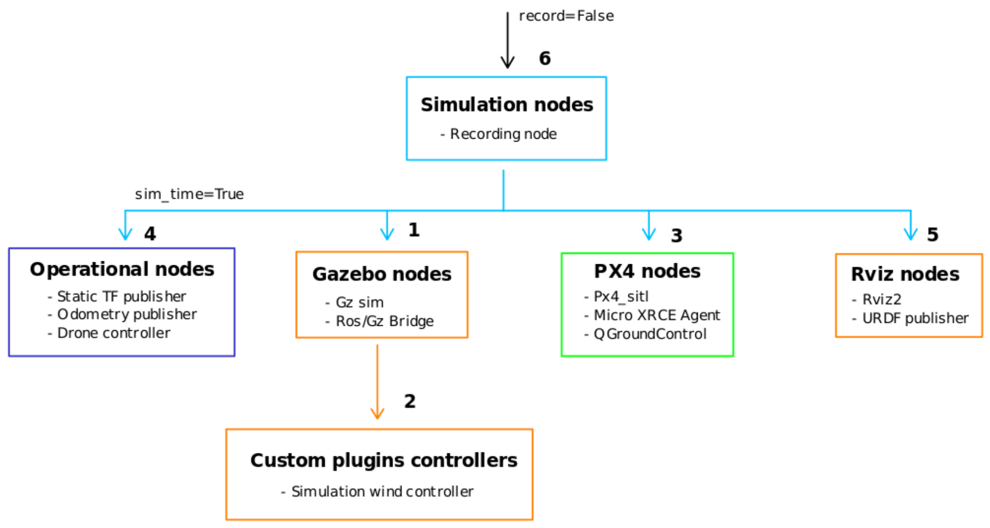


Figure 13: Adopted launch files hierarchy

An other advantage of this modular hierarchy of launch files is that all the operational nodes, which are the actual nodes that will run in the physical implementation of the drone, can be launched by means of the same launch file with different input parameters.

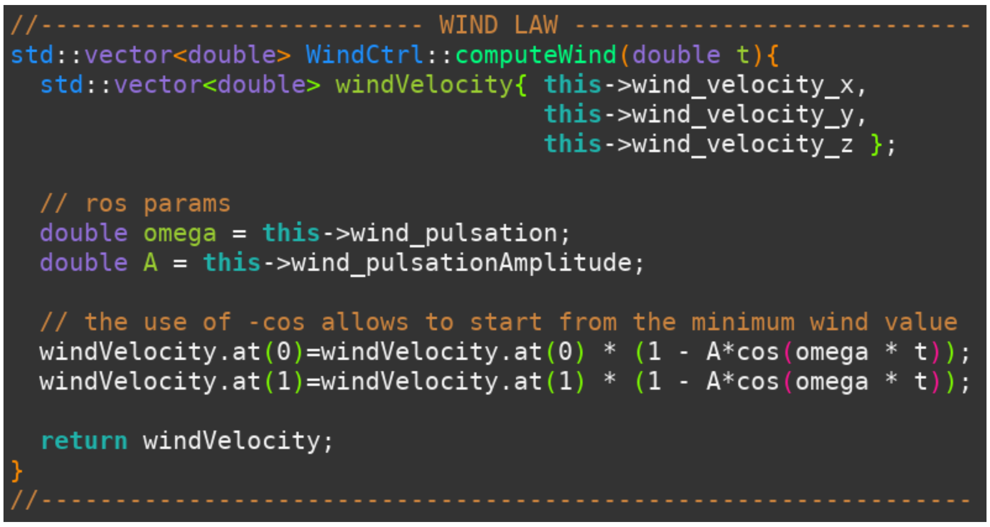
2. Develop the aforementioned Gazebo custom plugins

   In the paragraph 2.2.2 the generic structure of a Gazebo plugin and the mathematical description of the specific custom plugins has been discussed. Here will be discussed the development environment adopted to implement such plugins and its connection with the ROS ecosystem. In fact, as has been discussed in the paragraph 4.1.3, the Gazebo build tool has been unified with the ROS2 one and therefore a Gazebo plugin can be developed alongside the standard ROS2 nodes and built with them. This is what has been done in this work: two separate libraries dependencies which shares the same building environment. The adopted approach achieves the highest modularity, enabling to develop pure Gazebo plugins independent from the ROS2 environment. The main drawback of this approach is the lack of flexibility because, in order to enable a connection with ROS2, it is mandatory the use of the ROS/GZ bridge, which does not support many topic data types, limiting the communication capabilities. In some applications, though, a clear distinction where Gazebo ends and ROS2 begins is not needed, therefore it is possible to adopt a much more flexible approach. In fact, since both the dependencies of

Gazebo and ROS2 are handled by the same software, such software can build programs which use the two libraries at the same time, enabling to create Gazebo plugin with embedded ROS2 nodes. In this way the Gazebo plugin does not require a bridge to communicate with ROS2 and has more options to configure the quality of service of the communications as well as the data types transmitted.

3. Develop the wind controller

   The wind controller is a ROS2 node which takes in input, from the launch file configuration, the parameters of the desired sinusoidal wind law that will describe the wind behavior in the simulation. In Figure 14 it is shown the parameterized wind law adopted for the tests performed in this work. To customize the wind law with a different signal it is necessary to change the source code of the node and rebuild the package. Afterwards, the node periodically publishes the corresponding sampling of the wind law in the update topic of the wind-drag plugin.



```
//-------------------------- WIND LAW --------------------------
std::vector<double> WindCtrl::computeWind(double t){
  std::vector<double> windVelocity{ this->wind_velocity_x,
                                    this->wind_velocity_y,
                                    this->wind_velocity_z };

  // ros params
  double omega = this->wind_pulsation;
  double A = this->wind_pulsationAmplitude;

  // the use of -cos allows to start from the minimum wind value
  windVelocity.at(0)=windVelocity.at(0) * (1 - A*cos(omega * t));
  windVelocity.at(1)=windVelocity.at(1) * (1 - A*cos(omega * t));

  return windVelocity;
}
//--------------------------------------------------------------
```

Figure 14: Wind law configuration provided to the user

4. Configure the robot proprioception's simulator

   A robotic simulation environment is not complete if it simulates only the external dynamics of the robot. In fact, it is relevant also the simulation of the understanding that the robot has of itself. In particular, in this work this understanding consists in retrieving and interpreting the odometry from the autopilot and in correctly locating the acquired point-cloud from the depth-camera. The precise description adopted to achieve these purposes will be discussed in the point number five of this paragraph. Here will be addressed a methodological problem related to complexity management of a robot model in projects where both Gazebo and Rviz2 are used. Rviz2 is the simulation software embedded in ROS2 which allows to visualize the information that the robot has about itself. In order to work, this second simulator needs at least a minimal description of the robot kinematics. Unfortunately this description is not compliant with the SDF specification so, in the early stages of ROS2, a second model of the drone should have been configured and maintained in the URDF format, the native description of Rviz2. Tanks to the sdformat_urdf package this redundant description is no longer needed since this official package allows to translate, during the launch phase, the Gazebo model into the URDF description needed by Rviz2 nodes. Anyway, in this work the drone model has been kept as simple as possible, so only a minimal URDF description has been provided. In the cases where only a URDF model is available, the conversion to SDF is directly supported by Gazebo. In this way the migration to SDF-only descriptions, which very likely will be the predominant format adopted in the years to come, is simplified.

5. Develop the operational nodes

   The operational nodes perform two logical tasks: defining the reference frame hierarchy against which the data are expressed and the actual high level controller of the drone. The hierarchy of reference frames, shown in Figure 15, embeds the information of the drone pose, therefore it is necessary to collect the odometry data from the PX4 and interpret them correctly. This data are published in a topic in the form of position and orientation of a Forward Right Down (FRD) mobile frame with respect to a North East Down (NED) inertial frame. This convention is common among aerospace engineers, but it tends to be less intuitive than the more standard Forward Left Up (FLU) and East North Up (ENU), for this reason in this work the frames configured in both Gazebo and Rviz2 follow these latter conventions. In order to translate the information coming from the odometry of the PX4 to something that Rviz can correctly interpret, some geometrical transformations are required. In particular, the final objective is to translate all the data into the ENU inertial frame, therefore two transformations are needed. The first one converts the NED frame directly to the ENU frame, implemented with a static TF publishing the quaternion $(0, [\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0])$, which represents the left matrix of Table 2. The second one converts the FLU frame into the FRD frame, by means of an other static TF publishing the quaternion (0,[1,0,0]), which represent the right matrix of Table 2. Finally, a node publishes the PX4 odometry as a dynamic TF between the FRD frame and the NED frame, closing the path from the mobile FLU frame to the ENU inertial frame. In order to correctly interpret the point-cloud an additional reference frame for the relative pose of the camera with respect to the FLU frame of the drone is required. This can be achieved with a static TF configured with the mechanical parameters of the specific drone in use.

Table 2: On the left the NED frame expressed with respect to an ENU frame. On the right a FLU frame expressed with respect to FRD frame.

X_{NED}	Y_{NED}	Z_{NED}	X_{FLU}	Y_{FLU}	Z_{FLU}
0	1	0	1	0	0
1	0	0	0	-1	0
0	0	-1	0	0	-1

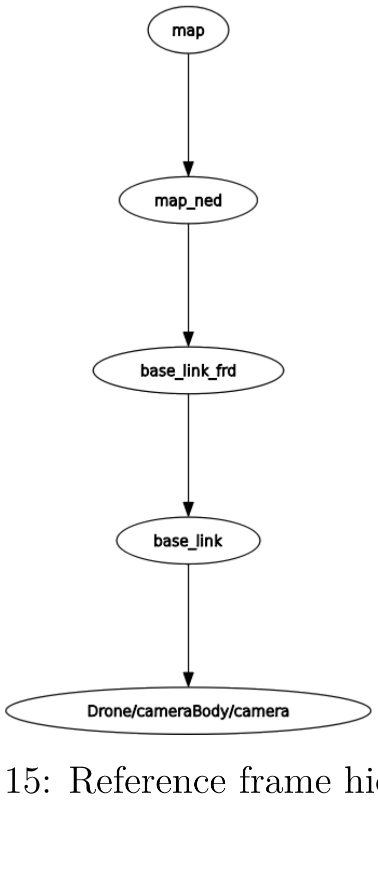


Figure 15: Reference frame hierarchy

5 Simulation results

The simulation environment has been tested and all the elements seems to work as they should, with no noticeable misbehavior. For this reason the plugins that has been analyzed in detail in this work are the custom ones only. The experiment adopted to test them together is a simple hovering. In fact, all the data collected observing a takeoff, the spraying activity and all the countermeasures deployed by the PX4 to contrast a strong wind gust are enough to check the correct behavior of such software. In the following sections each part of the test with the relative results will be analyzed.

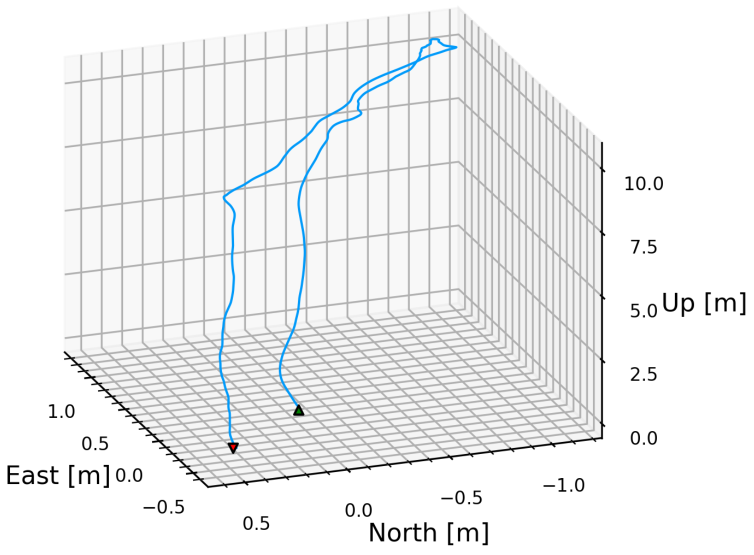


Figure 16: Actual path traveled by the drone, disturbed by the wind

5.1 Takeoff

During the takeoff phase it is possible to check the gravitational and inertial forces produced by the sloshing-effects plugin. In particular, before the takeoff the resultant of the sum of the two forces should be equal to the weight of the tank, since a still body does not have any inertial force. In the other hand, after the takeoff an inertial component appears and increases in magnitude the resultant force computed by the plugin. The tank has been initially filled with one liter of water, which has mass of one kilogram. Instead, the empty tank mass has been unified to the drone frame mass configured in the SDF model.

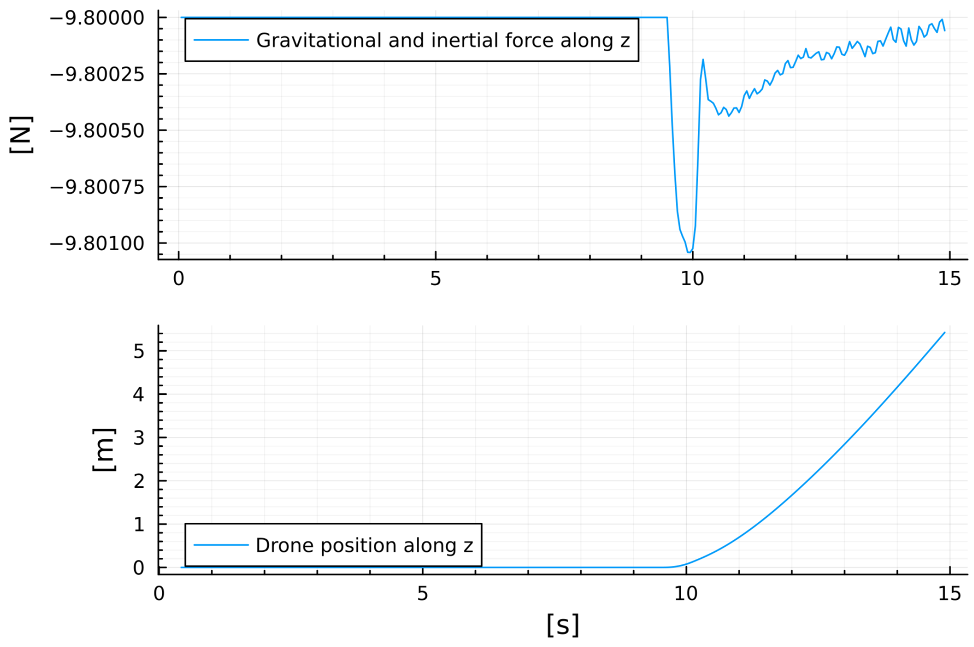


Figure 17: Gravitational and inertial forces along the z axis produced by the tank filled with one liter of water

In Figure 17 it is shown a detail of the takeoff phase. In particular, it is clear that the resultant force cease to be constant and have a jump right before the drone leaves the ground, in other words when the acceleration is maximum. Furthermore, when the climbing rate reaches the steady state, the acceleration returns near zero as well as the inertial force; this behavior is evident during the last seconds of the graph when the resultant force settles near to the tank weight force value.

The takeoff phase also allows to test the quality of the modeling of the drone actuation system by means of the multi-copter motor plugin. In fact a good model should be able to predict the maximum payload that the specific multi-copter can carry. In this work the configurations adopted for the motors were provided by the drone manufacturer so they performed very well without any tuning. This feature has been tested in a simulation configured with a payload exceeding the manufacturer specification and in that case drone was correctly unable to takeoff because the thrust generated by the motors was not enough.

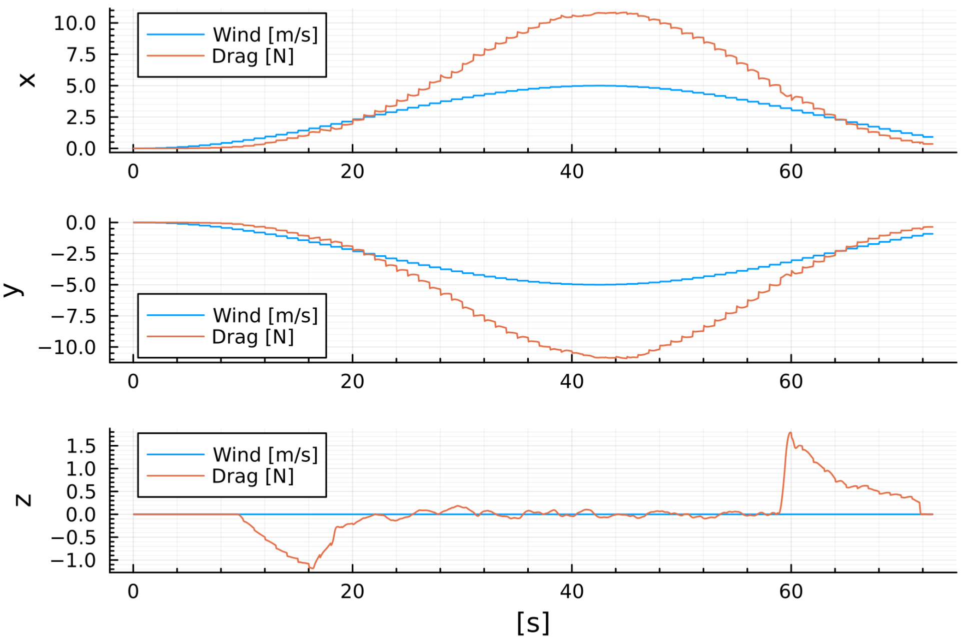


Figure 18: Analysis component by component of the drag force produced by the given wind, on x and y axis, and by the drone velocity on the z axis

An other effect that can be seen during the takeoff and landing phases is the drag force opposing the motion. In fact along the z axis the true wind speed has been imposed to be zero, so cannot interfere with the apparent wind speed generated by the vertical drone motion. This behavior is shown in the third graph of the Figure 18.

5.2 Spraying

The spraying activity reduces the quantity of liquid inside the tank, modifying the inertial and gravitational forces on the whole drone. In Figure 19 it is possible to see how the "sloshing-effects" custom plugin quantifies such behavior given a flow rate control signal. In particular, when the spraying flow rate is imposed different to zero the liquid inside the tank decreases and the corresponding weight with it. This variation in weight can be detected also in Figure 20, where around the twentieth second the motor speed start reducing due to the reduction of the fluid in the tank.

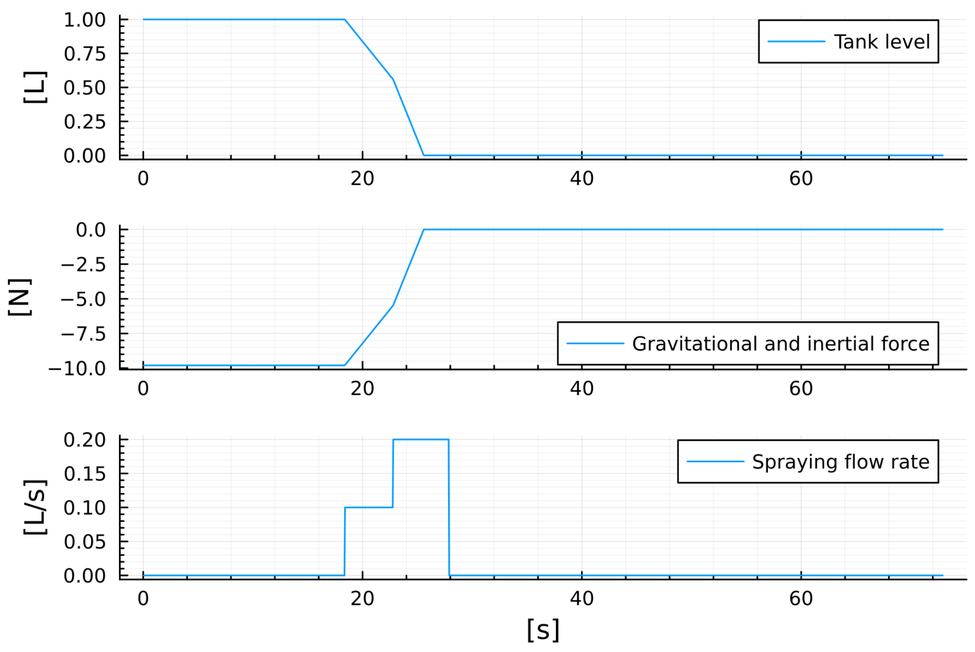


Figure 19: Variation of the fluid in the tank due to the spraying activity

5.3 Wind gust compensation

In hovering condition, other than the influence of the tank on the drone, the wind is a source of disturbance for the PX4 controller. The Figure 20 resumes how the autopilot handles all the disturbances active on the system. After the twenty-fourth second the tank is completely discharged and all the command sent to the motors is due to the wind compensation. Since in this specific simulation the wind has been set with a sinusoidal shape, it is possible to match such shape also in the motor speed.

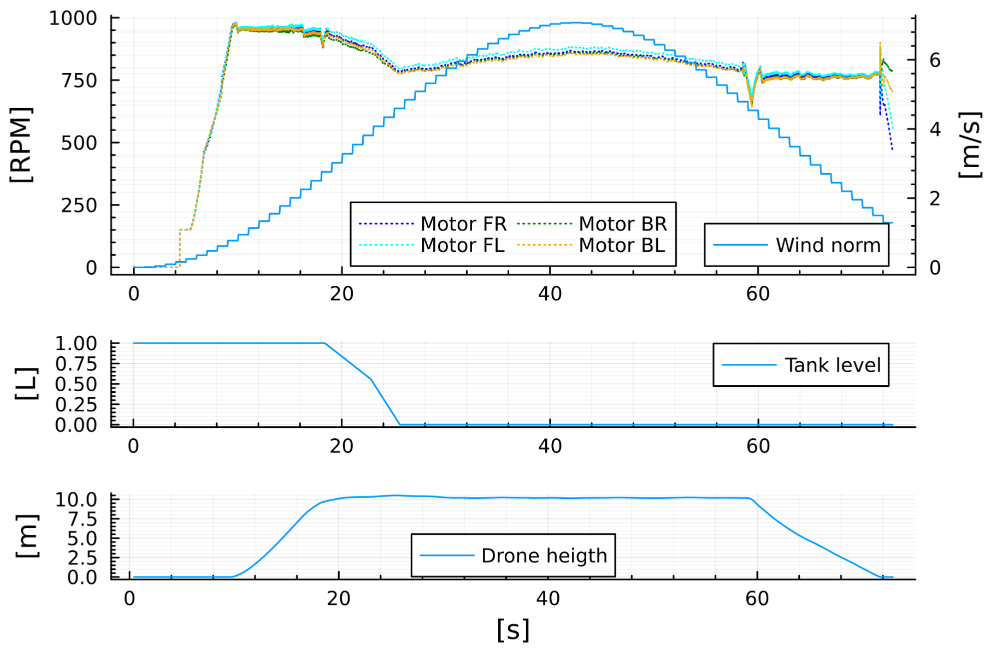


Figure 20: Motor activity compared with the factors which influence its behavior

Anyway, in order to explain why the drone exhibits such small increase in motors' speed in front of a strong wind, it is necessary to observe how the drone's attitude changes over time. This is resumed in Figure 21, where it is clear that the PX4 is pitching and rolling the drone in order to orient the propellers against the wind during the peak of the gust. In this way the thrust direction is optimized to compensate the effect of the wind drag.

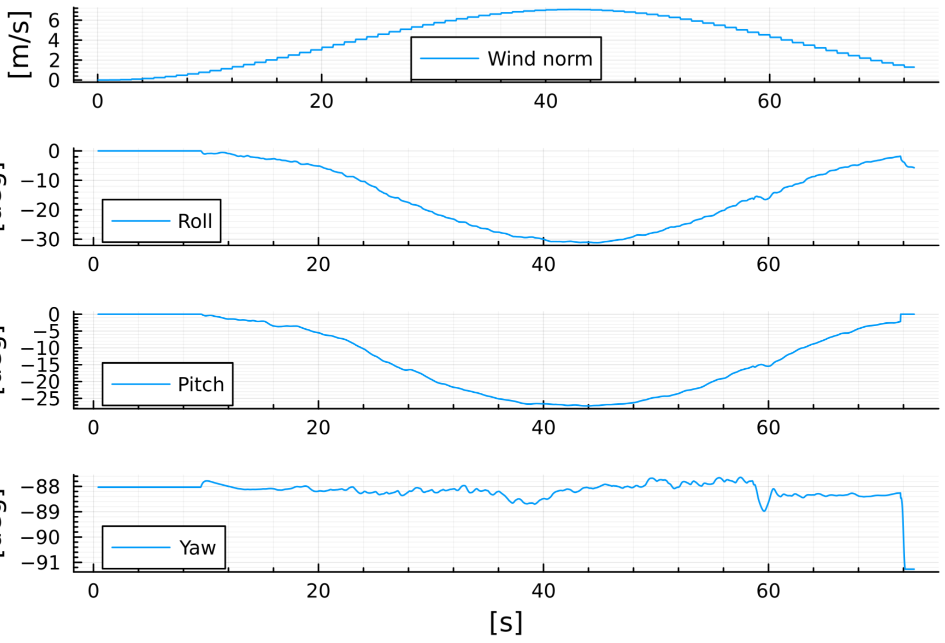


Figure 21: Change of the drone orientation to compensate the wind drag

6 Conclusions

In this work many software has been integrated in order to provide the "Department of Mechanical and Aerospace Engineering" of the "Politecnico di Torino" a simulation environment to test the control algorithms related to their research activity. This environment has been designed to be as modular as possible, allowing the users to startup only the minimal set of tools required to perform their tests and to incentive the re-use of each module. The outputs of the test simulations are coherent with the expected behavior, therefore the environment started to be used by the researchers as planned. Anyway there are still some important improvements that can increase the representativeness of the simulator, such as the introduction of a real sloshing effect formula in the sloshing-effect plugin, rather than using a simple variable mass relation, and the adoption of the geometry of the actual drone that will be used by the research group, instead of relying on a default model.

Appendix A - Point-clouds

A point-cloud is a discrete set of points in space. These datasets can represent 3D objects and usually are collected from the physical world to re-create a solid model. Since this kind of representation is usually expensive in term of disk space and computational resources, there are different formats to store the data, optimized for specific tasks. Furthermore, there are many techniques to filter out the sensor noise and to convert them into solid meshes, much more convenient to be used.

Acquisition

A point-cloud can be collected in many ways, in this section the four more common methodology will be discussed.

Range finder scansion

A range finder is a tool which allows to measure the distance of an object with specific characteristics. Nowadays, the more common range finders are sonar, radar and lidar. All these names are acronyms standing for SOund Navigation And Ranging, RAdio Detection And Ranging and LIght Detection And Ranging. Since they can measure the distances of objects with respect to a specific point in space, they can collect a point-cloud performing a scansion of the space. The sonar technology performs poorly outside water, so it is mainly used in maritime applications such as mapping the seabed. The radar technology is highly effective in long ranges with big objects, so, other than detecting airplanes and ships, it is used in military satellites to map the earth's surface. The lidar technology has shorter ranges, but it has an outstanding precision in measurements, so it is used in many contexts, such as robotics, where precision rather than range is needed. All these range finders can work in real time and outdoor, but they cannot measure colors in their base form, which is a big drawback in some sectors.

Stereoscopic Vision

In nature many animals, among which the humans, have eyes on the same side of the body. This enables the sensing of depths by means of the so called stereoscopic vision. Using two cameras is it possible to implement computer vision systems based on such phenomenon.

In fact, since the relative position of the cameras is known, the parallax phenomenon can be exploited to calculate the distance of some features of the same scene, viewed from two different but close perspectives as in Figure 22. This methodology can be implemented by means of standard cameras with an external elaboration unit or using embedded systems called depth-cameras[13]. Both the implementations have high quality outputs and can work outdoor in real time. Furthermore, the coloring of the resulting point-cloud is maintained.

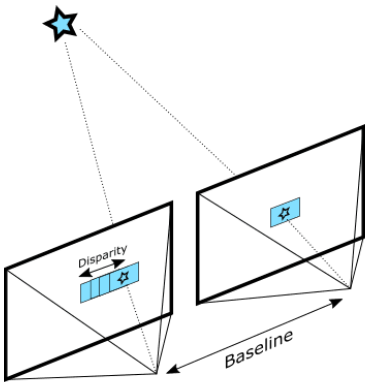


Figure 22: The Depth camera's working principle

Light projection and detection

A cheaper solution than a depth camera is a system composed by a single infrared camera and a infrared projector. This technology was made famous by the Xbox Kinect developed by Microsoft. The idea is to avoid the computational burden due to the feature matching process between the stereoscopic images, enabling the use of the parallax phenomenon by evaluating the distortions on a projected pattern. In particular, if an infrared light beam is filtered through a diffractive element its transversal section shape will be dependent on the traveled distance. By capturing, with an infrared camera, the distorted diffraction pattern it is possible to measure the distance of each point where the pattern is projected from the light source, creating the point-cloud [11]. This method is less precise and it is suitable for indoor environments where the infrared noise remains low. Furthermore, the range of such devices is limited and in their base form they do not support colors. Anyway, an additional RGB camera can added to provide the color layer to the point-cloud. As a matter of fact, even if the performances are relatively poor, the real time acquisition speed and the low prices make the devices based on this technology still a viable

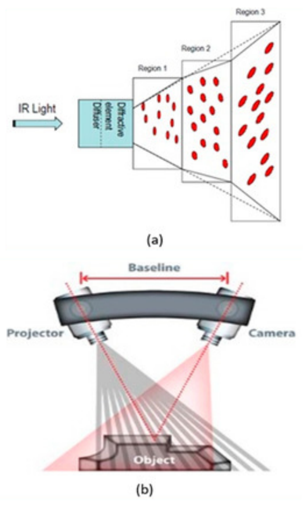


Figure 23: The Depth camera's working principle

[13]Image: www.intelrealsense.com/stereo-depth-vision-basics

alternative [5].

Image post processing

Capturing images of the same object from many perspectives with a constant illumination allows to match the same features, triangulate them and create the point-cloud. This procedure is called photogrammetry and requires at least one resolution camera with a system to change framing or several fixed high resolution cameras. An environment with a controlled illumination is highly recommended, therefore it is not suitable for generic outdoor applications. Finally, a good computational power to perform the post process of the data is required, anyway the real time acquisition speed is out of reach. This method is expensive in terms of elaboration time, but, in the other hand, it produces high quality outputs. Furthermore, the collected point-cloud keeps the information regarding the coloring of the object, making the photogrammetry the principal digitalization method adopted in the art conservation field. On the market many proprietary photogrammetry software as well as open source ones, such as Meshroom and Regard3D, are available.

Data formats, filtering and conversions

In the past, point clouds were invented independently by various parties to fulfill different purposes. Therefore, there are many data formats to accomplish the task of storing a point-cloud [3, 8]. Anyway, each data format has been optimized to perform the specific task for which it was designed, but the scope was usually too narrow to be efficient also in other contexts [8]. The open source project Point Cloud Library (PCL) defined a flexible format for efficient point-cloud general purposes elaborations. The library allows to elaborate and convert the point-clouds in many formats. An other important open source tool for point-cloud elaboration and conversion is Cloud Compare, which integrates many libraries other than PCL to extend the overall processing capability. The last relevant open source software to consider is MeshLab, which enables to editing the point-cloud by means of a convenient graphical interface. A particularly useful capability of this software is to unify the filtering and re-sampling functionalities, useful for the data complexity reduction, with the algorithms needed to convert the point-cloud into a mesh. This passage is non trivial and there are many options to fit a point-cloud with a solid shape. In this work some tests has been carried out to estimate the main difficulties of the creation of a terrain model starting from a point-cloud.

Using some freely available point-clouds[14] a simple but effective procedure has been identified. In particular, the raw point-cloud has been first re-sampled into a uniform 3D grid of points and then the surface has been reconstructed into a mesh by means of the Ball Pivoting algorithm.

[14]geoslam.com/sample-data

Appendix B - Official Gazebo Plugins

In order to write the custom plugins some official ones has been studied since the documentation about the new version of Gazebo is too little to be exhaustive. This study has been useful, other than to learn some best practices and coding techniques, which are not relevant to this document, to have a precise idea of the level of detail introduced in the model by these plugins and what kind of applications can be simulated with the default suite. In the following paragraphs will be described the mathematical representation of each plugin.

Buoyancy

This plugin adds a buoyancy force to all the objects which have a collision domain and has been enabled to be buoyant. In particular, the z axis of the fixed frame is divided into layers and each layer is characterized by a fluid density. This allows to model a lumped density change due to the increase of depth or altitude. Then the collision domain of each object is partitioned according to the layers and for each partition is computed the center of buoyancy and the volume. With this information the Archimedes buoyancy force is computed for each layer and applied to the center of buoyancy according to the formula reported below. It is worth to notice that this plugin is suitable for both maritime and aerial simulations, since the airships dynamics can be modeled in first approximation with the same rule.

$$F_{buoyancy}(layer, pose) = density(layer) * volume(layer, pose) * gravity$$

Hydrodynamics

The scope of this plugin is specifically for the maritime simulations. In air, all the effects modeled with this plugin can be neglected due to a small fluid density. Unlike the buoyancy plugin, here the density has been assumed constant along the z axis. This further reduces the plugin applicability to only displacement ships and submarines in a fixed range of depth. The effects modeled by the plugin will be described in the following paragraphs.

Ocean currents

Underwater, the currents act similarly to the wind, therefore the plugin provides an API to configure the topics where the local current velocity experienced by each link is published. This allows to compute the apparent linear velocities of the link with respect to the water.

Added mass

In fluid mechanics, when a moving body accelerates or decelerates, it needs to deflect some of the fluid around it. This phenomenon is called added mass and it is modeled through a second order tensor which bounds the generalized fluid acceleration vector to the resulting force vector on the body [2]. Anyway, in many cases it is not necessary to keep the six degrees of freedom of the system coupled, so it is convenient to set to zero the off-diagonal terms of the matrix. The formula using the convention adopted in the plugin API is the following:

$$\mathrm{F}_{\mathrm{addedMass}} = \begin{bmatrix} x\dot{U} & x\dot{V} & x\dot{W} & x\dot{P} & x\dot{Q} & x\dot{R} \\ y\dot{U} & y\dot{V} & y\dot{W} & y\dot{P} & y\dot{Q} & y\dot{R} \\ z\dot{U} & z\dot{V} & z\dot{W} & z\dot{P} & z\dot{Q} & z\dot{R} \\ k\dot{U} & k\dot{V} & k\dot{W} & k\dot{P} & k\dot{Q} & k\dot{R} \\ m\dot{U} & m\dot{V} & m\dot{W} & m\dot{P} & m\dot{Q} & m\dot{R} \\ n\dot{U} & n\dot{V} & n\dot{W} & n\dot{P} & n\dot{Q} & n\dot{R} \end{bmatrix} \begin{bmatrix} linAccX_{apparent} \\ linAccY_{apparent} \\ linAccZ_{apparent} \\ angAccRoll \\ angAccPitch \\ angAccYaw \end{bmatrix}$$

Linear and quadratic damping

The linear and quadratic damping are modeled as decoupled relations active on each degree of freedom. In particular in the linear damping the force is proportional to the generalized velocity where in the quadratic damping the force is proportional to the square of the generalized velocity. Letting j a generalized coordinate and V_j the associated generalized velocity, holds for each j:

$$F_{damp_j} = \beta 1_j \cdot V_j$$

$$F_{damp_j^{2nd}} = \beta 2_j \cdot |V_j| \cdot V_j$$

Lift and Drag

The lift and drag plugin is directly related to a specific link, therefore for each airfoil a different plugin must be configured. The scope of the plugin is to quantify on a 2D plane lift, drag and pitching moment of the given aerodynamic profile with a flap to modify its behavior. The mathematical relations bounding the apparent wind with the wrench applied on the airfoil are presented below. In the convention adopted here, the suffix I denotes a vector expressed with respect to the inertial reference frame. The fluid dynamic forces are computed according to:

$$F_{(L/D)}/M_p = \frac{1}{2} \cdot \rho \cdot |velI_{ProjectedOnLiftDragPlane}|^2 \cdot C_{(L/D/M)} \cdot S \cdot direction$$

Where F_L refers to the lift, F_D refers to the drag, M_p refers to the pitching moment [1], ρ represents the fluid density and S the airfoil surface. The direction of pitching moment, lift and drag are spanwiseI, liftI and dragDirectionI respectively and they will be discussed in the next lines together with the apparent wind velocity velI vector. The lift, drag and pitching moment coefficients depend on the airfoil polars plus a contribution due to the flap control action.

$$\begin{aligned} C_M &= Cm_{polar} + (cm_\delta \cdot controlJointPosition) \\ C_L &= Cl_{polar} + (controlJointRadToCl \cdot controlJointPosition) \\ C_D &= Cd_{polar} \end{aligned}$$

The polar coefficients are linearized with respect to the angle of incidence α and during normal operations are computed according to:

$$C(m/l/d)_{polar} = C(m/l/d)_\alpha \cdot \alpha \cdot cos(sweepAngle)$$

While in stall condition the line used is $Cl_0 + Cl_{\alpha_{stall}} \cdot \alpha_{afterStall}$, which has negative angular coefficient.

$$C(m/l/d)_{polar} = C(m/l/d)_\alpha \cdot \alpha_{stall} \cdot cos(sweepAngle) + C(m/l/d)\alpha_{stall} \cdot (\alpha - \alpha_{stall})$$

The apparent wind velocity not only considers the relative linear velocity between the body and the surrounding fluid, but also the contribution of the instantaneous velocity of the airfoil center of pressure due to a non zero angular velocity.

$$\begin{aligned} velI = normalize(&linkLinVelI - windLinearVelI \\ &+ linkAngVelI \times centerOfPressureI) \end{aligned}$$

The versor of the pitching moment i.e. the normal to the lift drag plane is defined as:

$$spanwiseI = normalize(forwardI \times upwardI)$$

The lift versor is defined as:

$$liftI = normalize(spanwiseI \times velI_{ProjectedOnLiftDragPlane})$$

The drag versor is defined as:

$$dragDirectionI = -normalize(velI_{ProjectedOnLiftDragPlane})$$

In the formulas of the polar coefficients the sweepAngle is used to neglect the velocity component normal to the lift-drag plane. In fact, sweepAngle is defined as the angle between velI and the lift-drag plane.

$$sweepAngle = asin(velI \cdot spanwiseI); note : sin(x) = cos(90 - x)$$

The incidence angle is computed starting from the zero lift angle α_0 and summing the angle between the profile normal and the apparent wind normal.

$$alpha = \alpha_0 + acos(liftI \cdot upwardI)$$

Other than the Lift and Drag plugin, Gazebo provides the Advanced Lift and Drag plugin, which provides similar functionalities but taking into account the three-dimensionality of the airfoil and all the effects active on the six degrees of freedom of the body.

Acknowledgments

I would like to express my deepest gratitude to Dr. Stefano Primatesta, who proposed this interesting work of thesis and dedicated a lot of time and attentions in guiding me during the whole development of the project. I am also grateful to the Ph.D. students Riccardo Enrico and Petre Ricioppo who gave me a precious support on technical aspects in several situations during the implementation. Finally, I would like to thank my parents without whom I could not have undertaken this journey. With love and support they gave me the opportunity of studying at Politecnico di Torino and helped me in any possible way to face my challenges.

References

[1] Andrew Wood. *Aerodynamic Lift, Drag and Moment Coefficients.* https://aerotoolbox.com/lift-drag-moment-coefficient. Consulted on 18-01-2024.

[2] Thor I. Fossen. *Guidance and Control of Ocean Vehicles.* Wiley, 1994.

[3] Shengwei Tian Baoli Lu Liping Zhang Xin Ning Huang Zhang Changshuo Wang and Xiao Bai. “Deep learning-based 3D point cloud classification: A systematic survey and outlook”. In: *Displays* 79 (2023), p. 102456. ISSN: 0141-9382. DOI: 10.1016/j.displa.2023.102456. URL: https://www.sciencedirect.com/science/article/pii/S0141938223000896.

[4] W. Johnson. *Helicopter Theory.* Princeton University Press, 1980.

[5] Evan Hemingway Mu-Lin Cheng Kurillo Gregorij and Louis Cheng. “Evaluating the Accuracy of the Azure Kinect and Kinect v2”. In: *MDPI Sensors* 7 (2022), p. 2469. DOI: 10.3390/s22072469.

[6] Philippe Martin and Erwan Salaün. “The true role of accelerometer feedback in quadrotor control”. In: *2010 IEEE International Conference on Robotics and Automation.* 2010, pp. 1623–1629. DOI: 10.1109/ROBOT.2010.5509980.

[7] Robotic Systems Lab - Legged Robotics at ETH Zürich. *SimBenchmark physics engine benchmark for robotics applications: RaiSim vs. Bullet vs. ODE vs. MuJoCo vs. DartSim.* https://leggedrobotics.github.io/SimBenchmark. Consulted on 24-02-2024.

[8] Radu Bogdan Rusu and Steve Cousins. “3D is here: Point Cloud Library (PCL)”. In: *IEEE International Conference on Robotics and Automation (ICRA).* Shanghai, China: IEEE, 2011.

[9] Dirk Thomas. *A universal build tool.* https://design.ros2.org/articles/build_tool.html. Consulted on 23-03-2024.

[10] William Woodall. *ROS on DDS.* https://design.ros2.org/articles/ros_on_dds.html. Consulted on 27-02-2024.

[11] Dingtian Yan and Huosheng Hu. “Application of Augmented Reality and Robotic Technology in Broadcasting: A Survey”. In: *MDPI Robotics* 6 (2017), p. 18. DOI: 10.3390/robotics6030018.