POLITECNICO DI TORINO

Master's degree in Mechatronic Engineering

Master's degree thesis

The development of ROS-based offboard algorithms for autonomous UAVs intended for Mars exploration

Supervisors
Prof. Giorgio GUGLIERI
Dott. Stefano PRIMATESTA

Candidate
Riccardo ENRICO

April 2023

Abstract

Unmanned Aerial Vehicles (UAVs) have received increased interest in the field of mobile robotics. Recently, research in the field of UAVs is also studying their use in space applications and planetary exploration. An example is Ingenuity, the first drone on Mars developed by NASA.

This is because a UAV offers a wider area of operation with respect to an unmanned ground vehicle (UGV), and it provides the mission with a higher resolution regarding images if compared to a satellite or an orbiter.

Moreover, Software in the loop (SITL) simulation of such vehicles is preferred during the development phase since it enables to evaluate the algorithms implemented without making use of a real vehicle and without the risk of damaging the hardware.

This thesis proposes a set of algorithms related to UAV-based planetary exploration. Such software implementations range from the more straightforward functionalities, related to the navigation and off-board control of the drone to more complex ones which also incorporate on-board sensors, such as the available ventral camera, to provide imagery data of an object by encircling it through a circular trajectory. Additionally, an area coverage algorithm is used to map an area of the terrain by following a grid sweep path, while providing images of the underlying terrain.

Each of these algorithms is composed of a path generation functionality and subsequent, through the usage of a PID controller, trajectory tracking capability.

All the proposed algorithms have been developed with Python. The drone off-board control software implementation has been developed through the Robotics Operating System (ROS), more specifically the ROS 2 version, and the PX4 autopilot.

They are designed to work with a Linux operating system along with the firmware available on board the drone, without needing to resort to the UAV computational resources. Each of the proposed navigational features are organized according to a request/reply model available through the interface offered by ROS 2.

Besides, the UAV has been equipped with a precision landing algorithm making it capable of re-entering on top of a dedicated platform, in this thesis case a UGV. Moreover, the drone is supplied with a ROS 2 node implementing a Kalman filter algorithm that produces the relative position and velocity estimates between the drone and the UGV landing platform, from sensor readings. The available sensors are the ones already implemented via the PX4 autopilot flight stack, furthermore ultrawide-band antennas and a top-view camera for AprilTag detection are added to the simulation environment to increase the estimate precision.

Every sensor combination available on the estimation algorithm is tested in the simulation environment, to check their effectiveness and interaction.

Finally, the simulation environment has been developed through ROS 2 and the Gazebo simulator, it provides with the condition simulating the drone itself, a rover, and a reproduction of the Martian ground. The testing phase of the algorithms has been done by following the Software-in-the-Loop (SITL) approach. The drone model available through the PX4 autopilot has been used as the starting point and modified according to the estimation algorithm needs by adding the necessary sensors via Gazebo plugins.

Acknowledgements

I would like to express my gratitude to my advisor, Giorgio Guglieri, for giving me the opportunity to work on a thesis in such an interesting topic.

I would also like to extend my sincere thanks to my supervisor, Stefano Primatesta, for his support throughout the development of this thesis. His invaluable feedback and knowledge have been essential in developing and shaping my approach to the project.

I am deeply grateful to my family for their unconditional support, which has sustained me through the ups and downs of my academic journey. Their encouragement and belief in me have been fundamental in helping me achieve my goals.

Finally, I would like to thank all my friends for their constant support and encouragement, both in my academic pursuits and in my personal life.

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

DDS Data Distribution Service.

EKF Extended Kalman Filter.

ENU East-North-Up.

NED North-East-Down.

PID Propotional-Integral-Derivative.

ROS Robotics Operating System.

RTPS Real Time Publish Subscribe.

SITL Software in the Loop.

UAV Unmanned Aerial Vehicle.

UAVs Unmanned Aerial Vehicles.

UDP User Datagram Protocol.

UGVs Unmanned Ground Vehicles.

uORB micro Object Request Broker.

UWB Ultra-wideband.

VTOL Vertical Takeoff and Landing.

Chapter 1

Introduction

1.1 State of the art

Drones and Unmanned Aerial Vehicles (UAVs) are more and more used in the field of mobile robotics, this is due to the low cost of their hardware components [1] and especially because of their possibility of navigating the outdoors much more freely with respect to ground robots [2].

Moreover, the market offering in regards to UAVs propulsion system is a lot varied, ranging from multi-copters and fixed wing to Vertical Takeoff and Landing (VTOL) drones.

In recent years such surge in utilization has been also reflected in regards to space exploration. An example is the Mars Helicopter or Ingenuity, an UAV developed by NASA, an autonomous flight system which is fitted with onboard control and navigation algorithms, which receives commands from the Perseverance rover [3].

The usage of UAVs has been largely appreciated for such context mainly due the possibility of offering a mapping of a wider area, with respect to UGVs and provide an higher image resolution with respect either to satellites or orbiters.

This increase in both interest and utilization of such vehicles has led to the development of dedicated state-of-the-art frameworks from both a generalized robotics point of view, this has lead to the creation of the Robotics Operating System (ROS) [4]. Concerning the flight stack of UAVs the PX4 autopilot [2] has been introduced. Both of these software tools are publicly available as open source projects. Both projects have been inspected in details in the following chapters (Chapter 2).

1.2 Thesis objective

The objective of this thesis is to provide with an easily employable simulation environment for a UAV, such vehicle is intended for mars exploration.

In order to do so, navigation, control and estimation algorithms are developed through Python. The communication between these algorithms, or nodes, is enabled using the Robotics Operating System (ROS) framework. The lower level section of the flight stack

(i.e. attitude control) is taken care of by the PX4 Autopilot. While Gazebo has been utilized as the simulation environment.

All of the software tools listed previously have been described in their relative section in chapter 2.

The proposed navigation and control algorithms in order to tackle the main tasks required for such a vehicle, range from the most simple (e.g. takeoff and reaching a target) to the most complex (e.g. mapping the underlying area while following a trajectory and performing a precision landing). The algorithms development and their employment is described in chapter 3, instead in chapter 4 the ROS 2 nodes employed for the position estimation are discussed.

1.3 Thesis organization

The structure of the thesis is shown in the following list:

- **Software employed introduction**: Introductory chapter explaining the main software tools used during the implementation of the steps required for the thesis project.
- **Control algorithms**: Analysis of the user interface, the algorithms, and their organization, used to accomplish the goal requested by the user.
- **State estimation**: The state of the UAV is estimated through a Kalman Filter node developed by using the FilterPy library, as with the proposed algorithms the communication is enabled trough ROS.
- **Simulation results**: Finally, the proposed algorithms are tested through a SITL simulation, with the Gazebo simulator.

Chapter 2

Simulation environment

2.1 ROS Introduction

The Robotics Operating System (ROS) is not a full operating system as the name would suggest. It has two main components, a middleware or a communication layer enabling algorithms developed with different programming languages to exchange information with each other, and also offers a set of libraries and tools used for software development for a robotic application [4].

The ROS project has started in 2007 and a newer version (ROS 2) has been released starting from 2017 with the first non-beta release (Ardent Apalone). The distribution used in this thesis is Foxy Fitzroy, released in 2020 [5].

The core concepts of ROS have been described by [4]:

- **Peer-to-peer**: The ROS system is distributed, meaning that it consist of a number of processes connected at runtime through a peer-to-peer topology.
- **Tools-based**: ROS is developed in order to maintain a large number of small tools, so that everything is pushed into separate module.
- **Multi-lingual**: ROS supports a number of languages, according to [4] those are Python, C++, Octave and LISP. In this thesis case Python has been chosen for the algorithms implementation.
- **Thin**: The devolvement of each ROS-based applications is encouraged, by the developers, to use standalone libraries that have no dependencies on ROS. Thus placing all the complexity in the libraries and creating small executables, allowing for easier code extraction and reusability.
- **Free and open source**: To add on the previous point the focus of code developed for ROS is to be as reusable as possible, thus the source code of ROS is publicly available.

2.1.1 Concepts

The fundamental concepts of ROS revolve around its organization through nodes, messages, topics, and services [4].

Nodes: Are processes that perform the computation, a ROS -based system will be comprised of many nodes.

Messages: Nodes communicate through messages. A message is a strictly typed data structure and many data primitive types are supported (integer, float, bool etc.).

Topics: For a node to receive certain messages it must subscribe to the topic on which such messages are sent.
This publisher-subscriber model support one-to-many or many-to-many communication.

Services: Although flexible, the previously shown communication model is not suitable for synchronous communications. In order to support such communications a request-reply model is implemented through the use of services.
This is analogous to web services and it has to be noted that in this case only one-to-one communication is possible.

Actions: Another way to have synchronous, request/reply communications is to use Actions. This also allows for the client which is requesting the action, to receive a steady feedback while the action is being executed.

The ROS conceptual graph is shown in figure 2.1. Among the tools made available by the framework there are `rqt` and the `rqt_graph` [6], to visualize such connections.

2.2 ROS 2

2.2.1 Comparison of ROS 2 and ROS 1 features

While the first version of ROS has been designed with mainly academic projects in mind it has now started to be used in market-oriented products [7].

To satisfy the requirements of such applications a newer version of ROS had to be developed. It addresses the specifications listed below [8]:

- Multiple robots system versus single robot applications [9].
- Usage via a small embedded platform. Instead the first physical application of ROS 1 was PR2 a robotics system which was equipped with workstation level computational capabilities [9][8].
- Real-time communication support [10].
- Quality-of-Service (QoS) support, allowing whoever is writing the code related to the publisher and subscriber to have more control over how the messages are exchanged, especially in the case of a less than ideal network [7].
- Ability of ROS 2 being able to work also in non-ideal networks condition.
- While still maintaining its research lab applications focus, ROS2 now offers more support towards production environments.

In order to address those needs some changes between the first and the second version were made, according to [10] [8] are:

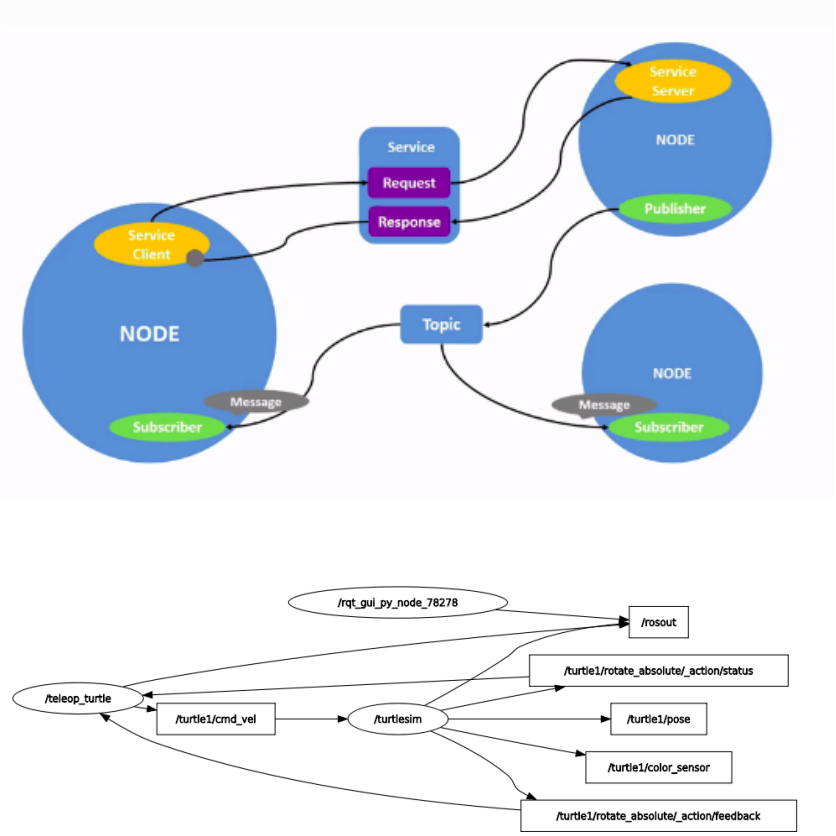


Figure 2.1: ROS graph as shown in the ROS 2 Foxy Fitzroy documentation and the `rqt_graph` [5]

API changes

The user facing APIs have been rewritten, previously the ROS libraries used in order to write nodes in C++ or Python were respectively `roscpp` and `rospy`, these were both independent from each other and so it is possible that some features may be developed for only for one of them [11].

In ROS2 these libraries are built by following a layered approach in which the base library `rcl`, implemented in C, is the same. On top of that the two language-specific libraries (`rclcpp` and `rclpy`) are built which now share all the same core features.

Programming languages versions

Focusing on the C++ library while previously ROS was built targeting C++03 it now supports C++11 and uses some parts of C++14, in the future ROS 2 might start employing C++17.

Instead on the Python side ROS 1 targets Python 2, while in the new version the

minimum requisite is Python 3.5.

Also, in ROS 2 Python is another supported method on how launch files are written, instead of only XML. On top of that according to [7] it has now become best practice to write launch file with Python.

ROS Master removal

With ROS 2 starting a ROS master in order for the nodes to discover each other is not necessary anymore. Each node has the capacity to discover others, thus avoiding the ROS master requisite and allowing to create a fully distributed system.

Building nodes

In ROS 1 in order to build a package CMake has to be used (`catkin_make`) instead ROS 2 supports Ament as a build system on top of which stands the colcon command line tool, so in order to compile the command `colcon_build` is used when in the ROS 2 workspace.

This also influences how packages are created as previously there was no distinction between C++ and Python packages, with a ROS 2 package it's now required to specify either `ament_cmake` in the case of a C++ package or `ament_python` if instead Python is used.

OS Support

While the main target of ROS 1 is Ubuntu, with ROS 2 the main OSes supported are Ubuntu, MacOS and Windows 10.

2.2.2 ROS 2 Topics vs Services vs Actions

As explained previously, in section 2.1.1, the robotics operating system offers three type of interfaces and consequently three types of node elements: topics, services and actions.

The ROS 2 documentation [12] provides a more in depth information regarding how to use the core components already introduced. In this section, those guidelines are discussed and examine how these components have been organized in this thesis.

Topics

Topics are based on a publish/subscribe communication model, then should be used for continuous data streams (such as UAV sensor data or state) to which other nodes have the possibility of subscribing or publishing.

For example, in this thesis, topics are used in the estimation part for the input and output of the Kalman filter (Section 4.2).

Services

Services instead are based on a request/response type of communication, and should be used for procedure calls that are carried out quickly.

The usage of a service then requires the creation of a client/server pairing and the related message definition (`.srv` files), in these types of files the message type of the request and the response must be defined.

An example in the algorithm is the service used to switch the drone from an a completed action to the hover state.

Actions

Instead to carry out operations which require more time with respect to services, actions are employed.

Both actions and services are based on a request and reply protocol and they need a server that must be started before the operation (i.e. by a launch file), actions are called to achieve a goal and send back feedback messages to the action client during the time required to perform it.

Similarly to services an action must be defined using an `.action` file, in which the goal, the result and feedback are defined.

In this thesis work actions have been employed to perform the control algorithms of the drone, in particular the takeoff, reaching a given target, getting images of a given target, performing an area coverage and the precision landing algorithm.

2.3 PX4

PX4 is an open source flight control software for drones and other unmanned vehicles. The project offers a set of tools that allow developers to create solutions for specific applications and to share them. Much like ROS the project has been developed and used in an academic environment, on top of that is supported by a world wide online community [2][13].

The PX4 flight stack consist of an ensemble of control, guidance and navigation algorithms [14]. These may be used on a real UAV or, in order to validate the software code developed via SITL simulation, as it has been done in this thesis.

2.3.1 PX4 and ROS 2 for offboard control

ROS is used by PX4 to provide an offboard control functionality, with a linux companion computer. In this way its possible to control the PX4 flight stack using a software outside of the autopilot.

PX4 supports both versions of the robotic operating system, but while the communication between ROS 2 and the autopilot is done through the ROS2-PX4 bridge (Figure A.2.3) the communication with the first version can be done either via two bridges (the PX4-ROS2 bridge and subsequently `ros_bridge`) or through a MAVROS package over MAVLink protocol[1] [16].

The usage of the second version of ROS is highly recommended, by the development team of PX4, as the PX4-ROS2 bridge is able to take advantage of the communication middleware (DDS/RTPS).

PX4 and ROS 2 communication

In this thesis the second version of ROS has been employed thus requiring only to setup the connection to the autopilot through the PX4-ROS2 bridge.

From the communication standpoint the PX4 autopilot employs an RTPS protocol and a DDS middleware in order to interface itself with an offboard DDS application (such as ROS 2 nodes), in this way it is possible to exchange uORB[2] topics sent by the client (the PX4 autopilot) to RTPS messages to the agent side (offboard computer) and vice versa.

The connection between such devices is enabled trough an UART or UDP link. This translation and communication bridge is called the microRTPS bridge, the main components of such architecture are shown in figure 2.2.

[1] MAVLink is a lightweight messaging protocol used in the drone ecosystem [15].

[2] uORB is an asynchronous publish/subscribe messaging API used between the components of the PX4 autopilot [17].

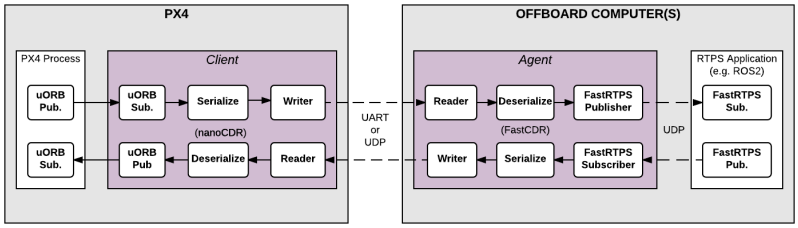


Figure 2.2: Components of the PX4-ROS 2 communication architecture [18].

Specifically, in the event of using PX4 in conjunction with an offboard computer running ROS 2 it is necessary to install and build the following packages in the ROS workspace, in order to enable a translation layer between the uORB topics and the RTPS messages.

The packages are[3]:

- `px4_ros_com`: Which contains the microRTPS agent code templates for both publisher and subscribers. Whenever the build process is initiated the file `urtps_bridge_topics.yaml` is mirrored on the agent side, the `px4_ros_com` folder, by the `uorb_to_ros_urtps_topics.py` file.

- `px4_msgs`: Contains the pre-configured px4 message definitions. At build time the content of the `px4_msgs` folder is updated via the `uorb_to_ros_msgs.py` automatically.

In both cases the packages are populated at build time via the mentioned python scrips contained in the PX4 firmware. Since not all uORB topics are made available to the ROS application as a default setting if specific messages are required to be transmitted from uORB to ROS messages then these scripts must be ran manually [19].

Both packages are available as GitHub repositories for ease of access an installation. In order to perform it, is's necessary to clone the repositories and then run the build command [20][21], as shown in the installation tutorial Appendix-A.

The packages are placed on the agent side of the communication, in the ROS 2 workspace directory.

[3]Such package are part of the ROS 2 workspace as shown in the appendix chapter regarding the installation Appendix-A

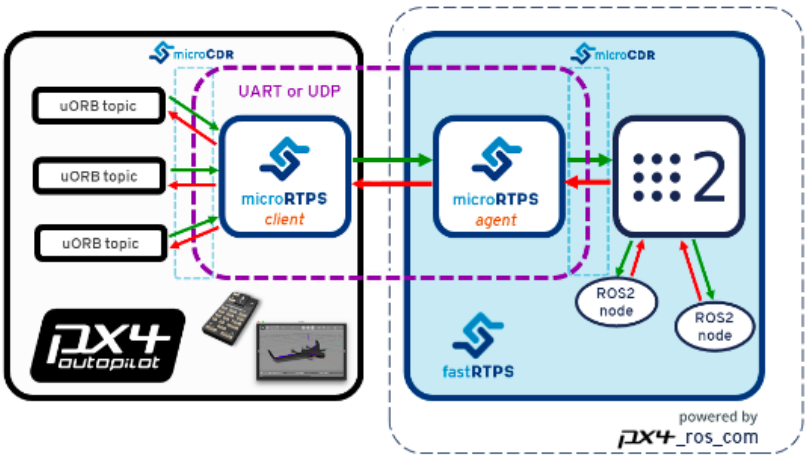


Figure 2.3: microRTPS link between the PX4 autopilot firmware and ROS 2 nodes [19].

The client runs on PX4 and an agent will run on the mission/companion computer, the communication is done through message translation of uORB (PX4 publish-subscribe messaging API) and ROS 2 messages.

In this way it is possible to create ROS 2 subscriber nodes that receive directly PX4 uORB topics, and vice versa with a publisher node (as shown in figure 2.3).

2.3.2 Offboard control

Offboard control is employed to perform each of the tasks shown in this thesis, this means that the vehicle responds to position velocity or attitude setpoints computed by algorithms running on a companion computer, while the PX4 autopilot is charged only with actuating such commands and stabilizing the drone.

PX4 offboard mode has some requirements that must be satisfied in order to be ran, as detailed in [22]:

- At least a stream of $> 2\ Hz$ of setpoints commands that shall be sent to the autopilot from the companion computer.
- At least one of the following pose/attitude information has to be available (GPS, optical flow, visual-inertial odometry, mocap, etc.)
- The vehicle must be previously armed, and it must receive an appropriate stream of commands prior to the offboard mode engagement.
- RC communication must be disabled. If an RC command is issued to the autopilot or the stream of setpoints commands is stopped, then the vehicle will exit the offboard mode and enter a failsafe state, thus landing.
- Offboard mode requires also a continuos connection to a remote MAVLink system or a ground control station software (such as QGroundControl), otherwise if such connection is lost the vehicle will enter a failsafe mode and land.

Figure 2.4 shows the high level flight stack organization of the PX4 autopilot. The flight stack is a collection of guidance, navigation and control algorithms for the drone.

The algorithms developed for the thesis focus on the high-level control and navigation as opposed to the low-level control algorithms which are implemented directly by the autopilot.

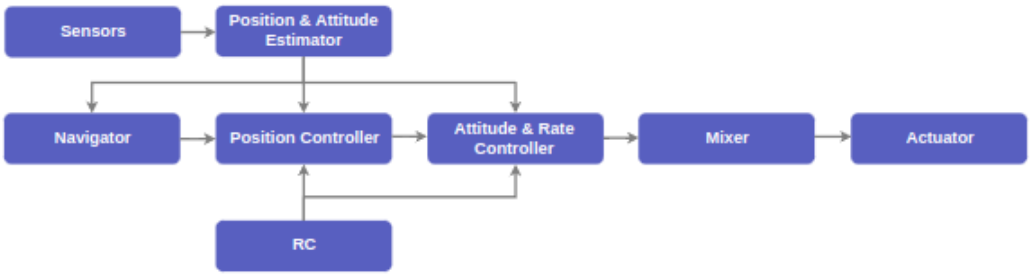


Figure 2.4: High level organization of the PX4 flight stack [23].

The PX4 flight stack allows to get the information regarding the position, velocity and altitude of the drone. This data is computed through the EKF2, an extended kalman filter, in the local NED frame [23].

In the estimation chapter it will be shown how this data will be used to compute the relative position and velocity with respect to the landing platform.

All of the algorithms developed are subscribed to the `VehicleOdometry_PubSubTopic` which gets the data from the EKF2 algorithm in the local NED frame (figure 2.6b).

Low level controllers

As shown in figure 2.5 the controller section of the flight stack is organized through a cascaded architecture, offering a position and velocity controller which are employed to computed the attitude setpoints fed to the attitude controller.

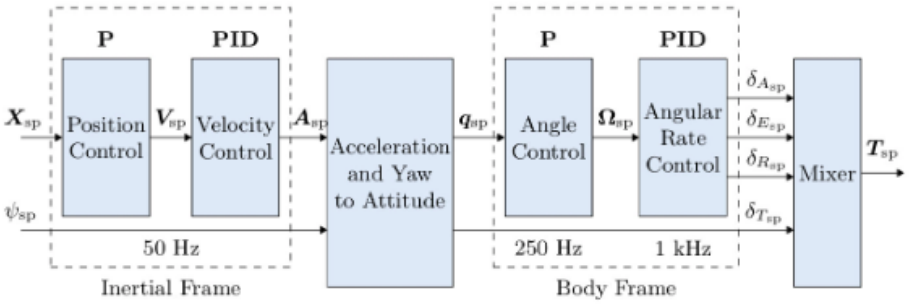


Figure 2.5: Cascaded architecture of the position and attitude controllers [24].

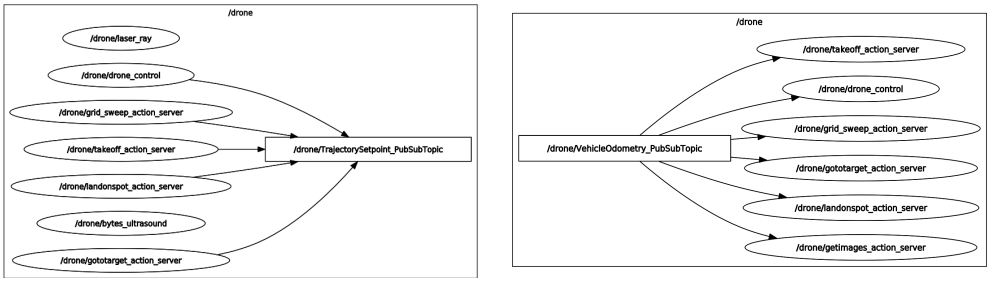
Position and velocity controllers

The figure does not show that it is possible to send directly a velocity vector (v_{SP}) as reference to the PID velocity controller, thus bypassing the first outer loop.

During the development of the algorithms this has been done for the algorithms which demanded a stricter trajectory following capability, the orbiting algorithm, the area coverage and the precision landing algorithm.

During the testing phase bypassing the outer loop and using a customizable PID controller showed better results in term of trajectory tracking.

The topic used by the algorithm to communicate with the controllers is the `TrajectorySetpoint_PubSubTopic` (figure 2.6a).



(a) Nodes communicating of the Trajectory Setpoint topic.

(b) Nodes subscribed to the Vehicle Odometry topic.

Figure 2.6: Nodes publication and subscription to the main topics available through the PX4 flight stack. Visualized through the rqt_graph tool.

ROS 2 Offboard control implementation

ROS 2 nodes can be used to interact with the PX4 flight stack in offboard mode. This is done through the microRTPS bridge explained section 2.3.1 which enables the ROS 2 node to publish the topics directly as uORB topics. In this thesis, the offboard control is implemented through ROS 2 nodes organized as actions.

2.4 Simulation Tools

Simulation tools are used to control a vehicle model via ROS 2/PX4 code. This is done trough Software in the Loop (SITL) in which the flight stack runs on computer as opposed to Hardware in the Loop (HITL) simulation in which the firmware runs on a controller board [25].
The PX4 autopilot supports a variety of simulators:

- Gazebo
- FlightGear
- JSBSim

- JMAVSim
- AirSim

Gazebo is the simulator used in this thesis project as it directly support a ROS integration and it comes pre-installed with the ROS 2 package.

2.4.1 Gazebo

Gazebo is an open source 3D simulator, it can be used to simulate autonomous robots and in particular is suitable to test computer vision related algorithms. Moreover it is necessary to point out that it supports also multi-vehicle simulation [26].

The simulator supports a lot of vehicle models, the one used for the thesis project has been the Iris quadrotor which comes as part of the PX4 autopilot SITL section.

Such models are defined in the `.sdf`[4] format through which is possible to easily modify the starting model and implementing custom plugins such as sensors.

While it is still possible to be used independently, the Gazebo simulator comes pre-installed when installing a ROS distribution via the set of packages called `gazebo_ros_pkgs`. Because of that and due to the fact that extensive documentation is provided by the autopilot docs, the gazebo simulator has been employed for the SITL simulation in this thesis work.

2.4.2 UAV model

The PX4 SITL package provides a UAV model of the 3DR Iris, which is a commercially available multicopter.

Figure 2.7: 3DR Iris

[4] `sdf` stands for Simulation Description Format which is an XML format used to describe, visualize and control models used in robotics simulators.

While it is possible to start with the base model offered via the PX4 Autopilot to simulate the most simple tasks (i.e. takeoff, reach a target) through Gazebo. In order to test the more complex tasks, which require the usage of additional sensors (i.e. camera), it has been necessary to modify the `sdf` model in order to add sensors plugins:

- **Camera**: a ventral camera is added with the `libgazebo_ros_camera` plugin, the camera points downward and is used both for image collection and to detect the position of the apriltag used during the precision landing procedure.
- **Ultra-wide band sensor**: Similarly, the ultra wide band sensor is added via a specific plugin (`libros2_px4_gazebo_uwb`).
- **Laser scan/Laser altimeter sensor**: In order to simulate the operations of a laser altimeter a `libgazebo_ros_ray_sensor` plugin is employed. This plugin is not specific to a laser altimeter and so it is necessary to specify a narrow angle in the `.sdf` file.

  This plugins offers an array of different methods regarding the output message type of the data, this is done by modifying the `<output_type>` entry in the `.sdf` file. In this thesis work two of them has been employed:

  - `sensor_msgs/LaserScan` message type: employed in the control section of this thesis whenever the altitude of the UAV had to be published as feedback, in order to provide a 2D scan with multiple results.
  - `sensor_msgs/Range` message type: employed in the estimation section of the code, this message type returns a single distance value which is the minimum of all ray ranges recorded by the sensors.

Figure 2.8: Modified Iris quadrotor in the simulation environment

2.4.3 World model

The PX4 Autopilot SITL package provides the user with a convenient bash file in which is possible to instruct the world and model files employed during the simulation, the bash file is the `sitl_multiple_run.sh`. With this is possible to simulate multiple vehicles using the Gazebo simulator.

The file has been suitably modified in order to employ the mars terrain world file and placing a platform simulating the UGV on top of which the precision landing is performed.

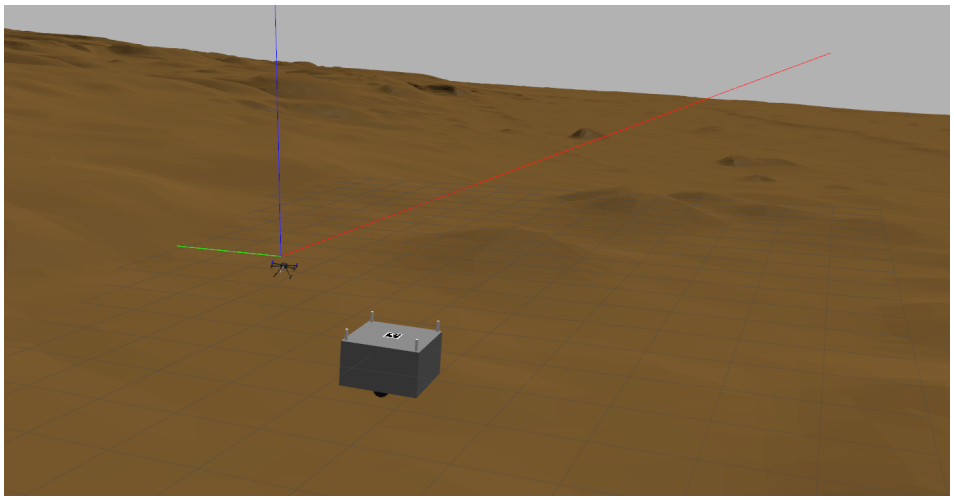


Figure 2.9: Simulation environment in which the UGV and UAV model are placed on startup together with the martian ground.

Moreover, the same file can be used to run directly the UAV model in the Gazebo environment.

2.4.4 Reference frames

It is also necessary to point out that the world reference frame employed by the simulator is the East-North-Up (ENU) reference frame while the PX4 autopilot employs the North-East-Down (NED) frame.

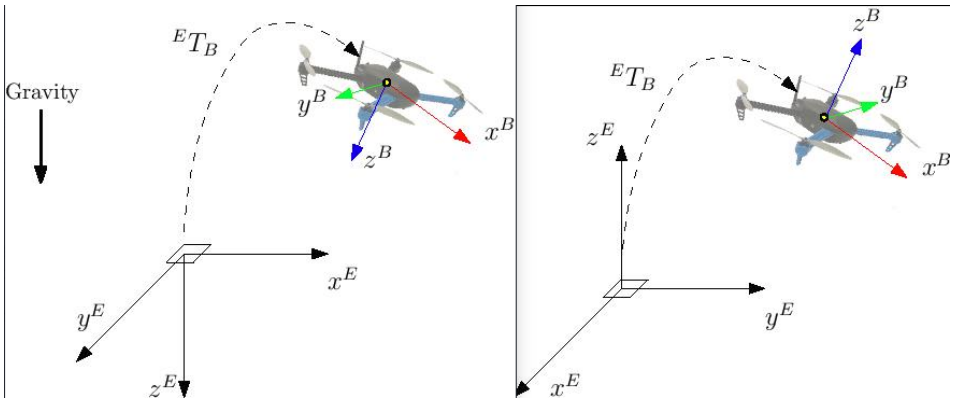


Figure 2.10: Reference frame comparison between PX4 (left) and Gazebo (right) [27].

Due to this, it is necessary to convert any position data coming from the simulation environment towards the algorithms paired with the autopilot, one example is the ground-truth measurement coming from the `libgazebo_ros_p3d` and employed during the estimation algorithm validation and tuning.

$$R_{NED}^{ENU} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{2.1}$$

Chapter 3

Offboard algorithms

3.1 Algorithms organization

In order to provide the UAV with the capabilities necessary to perform the tasks, from the simpler ones i.e. takeoff to the more complex i.e. taking images following a precise pattern and performing a precision landing, a number of algorithms have been developed.

Furthermore by taking advantage of the different types of interfaces made available by the robotic operating system it has been chosen to organize such algorithms mainly through ROS 2 actions.
The main tasks performed by the UAV in the simulation are:

- Takeoff
- Reaching a user-defined target
- Orbiting around a specific target while taking images
- Providing a coverage of a designated area while performing a grid sweep
- Precision landing

To do so, an action server has been developed and associated to each task. Subsequently an action client, has been set up, in order for the user to request the tasks to be performed, thus creating a request/reply communication model between the action clients and server.

Everyone of these server is then started via a single launch file for ease of use, and are paired with a continuously running control node. The organization is displayed in figure 3.1 while the code organization is examined more in depth in the appendix A.
Each of the main launch files starts some nodes and processes, described below:

- Simulation launch file
  - The `gazebo_sitl_multiple_run.sh` is a bash script used to start:
    * Launch the Gazebo simulation environment
    * Generate the world model for the martian ground
    * Position the drone model, which is a modified Iris model, in the Gazebo simulation

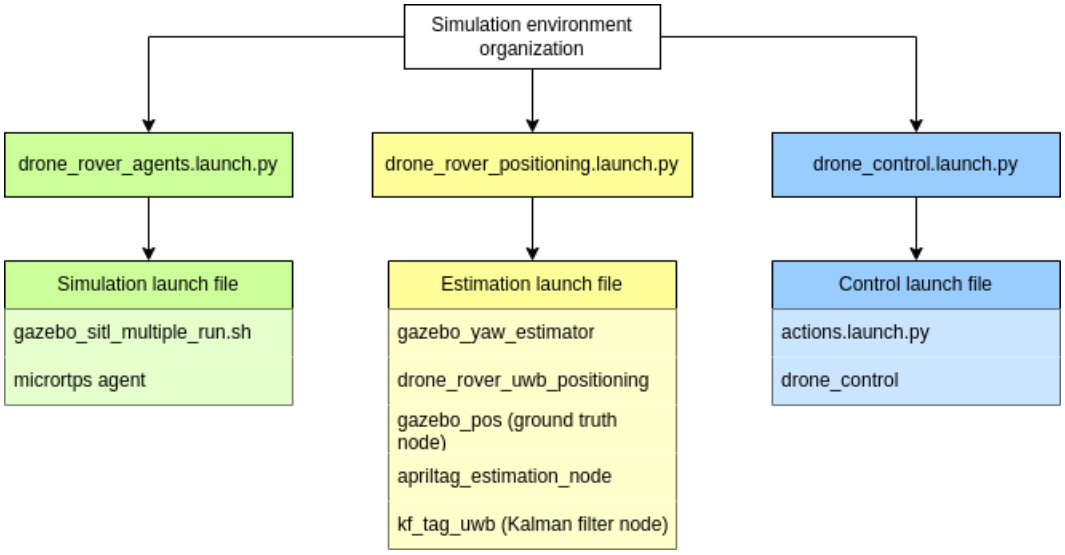


Figure 3.1: General organization of the simulation environment. This figure shows how launch files are used during the startup procedure. In this case also the launch files related to the simulation and the estimation are shown, more in depth detail are offered in the specific chapters respectively 2.4 and 4.

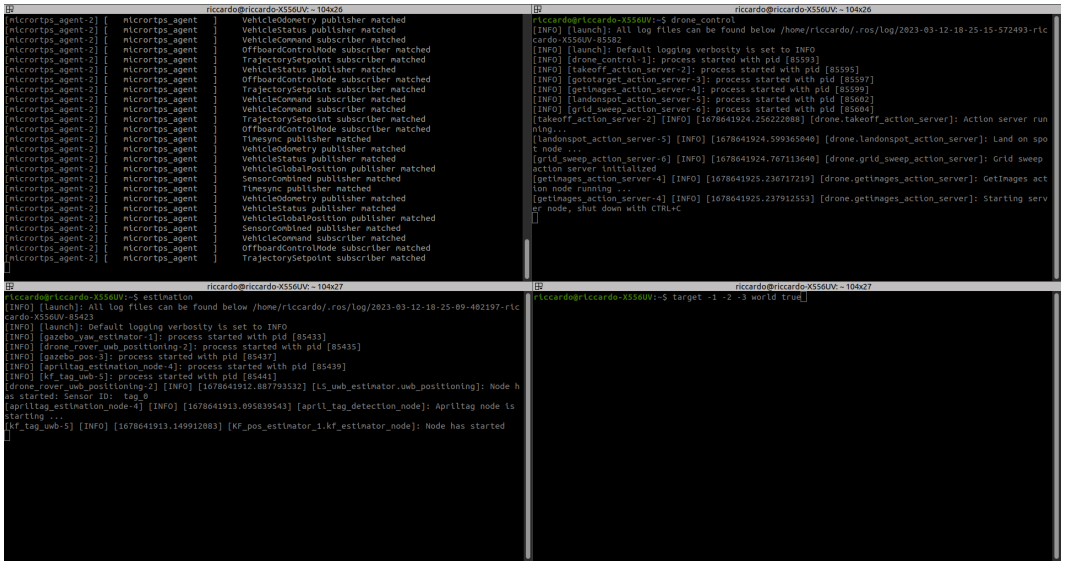


Figure 3.2: The figure depicts how the launch files shown in figure 3.1 are used when starting up the simulation environment. The top-left terminal is used to start the simulation environment, the top-right the action servers while the bottom-left the estimation algorithm and it's auxiliary nodes. A terminal window is left for the user to send the required commands.

  - MicroRTPS Agent, which is necessary in order to establish the communication with the PX4 flight stack and the ROS 2 nodes.

- Estimation launch file
  - `gazebo_yaw_estimator`: Node which publishes the yaw value of the rover in the world NED frame.
  - `drone_rover_uwb_positioning`: Node publishing the relative position of the UAV in the rover NED frame.
  - `apriltag_estimator_node`: Computes the relative position of the drone in the rover NED frame, similarly to the previous node.
  - `gazebo_pos`: This node is employed to collect the information related to the actual position of the UAV and the rover in the Gazebo simulation environment. The data gathered by this node is also used as ground-truth information during the Kalman filter algorithm tuning.
  - `kf_tag_uwb`: Hosts the actual Kalman filter estimation algorithm, through which the data fusion process take place, used mainly during the landing phase.

- Drone control launch file,
  - `action.launch.py`: Explained in detail in the next subsection.
  - The drone control node is used to implement the hover service capabilities and to provide a sort of fallback in case the action server were to fail.

Action servers launch file

As shown in the previous section each algorithm which is associated to a task performed by the UAV is started by a single python launch file in order to maintain a clean code and organization and to allow the startup phase to be less inconvenient for the user.

The action server are listed below:

- Takeoff → `takeoff_action_server.py`
- Go to target → `gototarget_action_server.py`
- Get images → `getimages_action_server.py`
- Area coverage → `grid_sweep_action_server.py`
- Landing → `landonspot_action_server.py`

The functionality and interfaces of each action server is described in details in the following section.

3.1.1 Reasoning behind the choice of ROS 2 Actions

As already explained in section 2.2.2, the employment of multiple ROS 2 actions instead of a single node concerning all control and navigation tasks offers a set of advantages. These consist mainly of:

- Possibility of offering the user with a request/response communication model, which is better from a user usage point of view with respect of the publish and subscribe experience offered by a simple node [12][28].
- A ROS 2 service paired with a single node containing all the control algorithms can also be used in order to offer the same request/response communication [28].
- This option has been discarded though because a ROS 2 service offers the client a single result instead of a steady feedback [28].
- Another reason is that of avoiding a single node and thus a single point of failure. In this case it has been chosen to associate an action server to each task while still offering a continuous running node in case of any problem, the drone control node.

It necessary to point out that the other two types of interfaces (topics and services) are still used for the development of the algorithms.

In particular topics are the core component for the communication of data between servers, a custom message regarding the drone status is implemented, the details of such implementation are explained in section 3.1.2.

Instead a service is provided in order to switch the state of the drone from performing an action to an hovering state. Furthermore, a more in depth examination is provided at section 3.1.2.

Moreover, a standard service offered by the `rcl_interfaces` (`SetParameters`) library is employed to select the additional sensor data to be employed by the estimation algorithm (the implementation of such service is showed in section 4.3.1), used by the precision landing action server.

3.1.2 Common capabilities to all the action servers

The action servers share some characteristics which are displayed in this section in order not to repeat each of them in every action server sub-chapter and instead focus on the specific features of each of the algorithms.

Hover service

The hover service is not directly implemented by any of the actions, but it is provided by the continuously running drone control node. All the proposed algorithms have the capabilities of requesting the drone to turning off the hovering state, for example when starting a new action task, or to turn on the hovering state at a specific position, when the task has been completed.

```
1 # Request
2 bool hover_req # true if hover requested
3 geometry_msgs/Point hover_pos
4 ---
5 # Response
6 bool hover_mode # true if in hover state
```

Source Code 3.1: Hover service interface definition. Via the `Hover.srv` file.

Whenever an action request the hover mode to be turned on it will also specify at which position in the world frame, by setting the `hover_req` to true and setting the position in the `hover_pos` as shown in the source code 3.1.

After this the drone will hover at the requested position which usually correspond with the lastly reached setpoint of the action task.

In this way the control node will publish a constant stream of setpoints to the dedicated topic with a frequency of 10 Hz.

Drone status topic

All the nodes composing the control section of the drone are subscribed to a custom topic called the status topic, defined by the `Status.msg` file (source code 3.2).

Each of the action server and the control node are subscribed to the status topic and can also publish on it, as shown in figure 3.3.

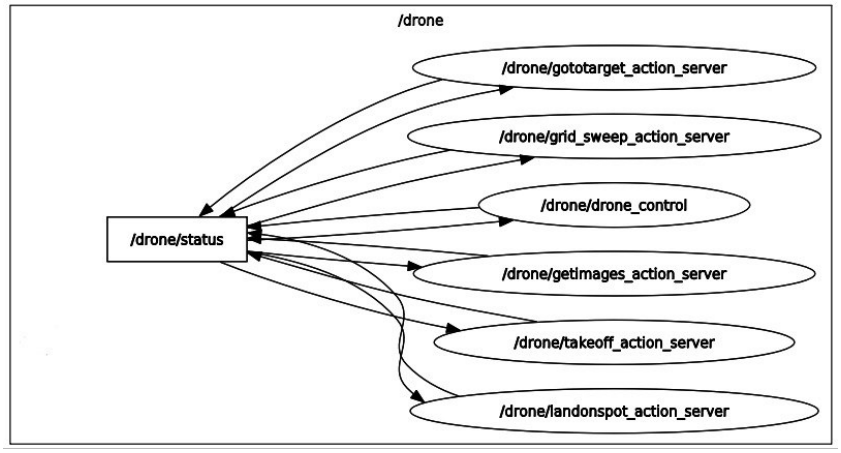


Figure 3.3: Rqt graph showing all the action servers and the drone status topic. The communication is bidirectional, all action servers are able to publish to the topic as well as gather the status information.

Each server then has the information regarding which action is being executed, if any.

Furthermore, if any goal request is made while the drone is completing a previously received goal the action server will reject the goal request.

The topic is defined as follows:

```
1    uint8 status
2    nav_msgs/Odometry odometry
```

Source Code 3.2: Drone status topic definition

As shown in the source code 3.2, the status message is quite simple. It exchanges the information related to the current state (in terms of task being performed) and the odometry of the UAV in the world frame. The status code is defined in the following table:

Action performed	**`status`**
Idle	0
Takeoff (in progress)	1
Takeoff (completed)	2
Hover	3
Go to target (in progress)	4
Go to target (completed)	5
Get images (in progress)	6
Get images (completed)	7
Landing (in progress)	8
Landing (completed)	9
Area coverage (in progress)	10
Area coverage (completed)	11
Abort	12

Table 3.1: Drone status code table

Goal rejection

In order to avoid a goal interfering with each other it is necessary to reject a goal request if another task is still in execution.

This is done through the knowledge of the drone status topic (section 3.1.2).

This algorithm component is executed in the `goal_callback` which is included in all action servers, the following pseudo code (algorithm 1) is employed to demonstrate this simple procedure, implemented in the callback function:

Algorithm 1 Pseudo code presenting the procedure employed in the `goal_callback` shared by all developed action servers.

Goal received by the action server → Stored in the `goal_handle` variable
if Drone status shows an idle state (drone status == 0) **or** Drone status show hover state (drone status == 3) **then**
  The goal is marked valid
  The goal is accepted and the action task can continue
else
  The goal action is marked invalid
  The goal is rejected
  The action task is not completed
end if

This procedure, while shared by all action servers, is slightly different in the case of the

takeoff action server due to the fact that the takeoff action is not necessary in case the UAV is in hover mode. In this specific case the takeoff request will be rejected if the drone is already hovering.

In every pseudo code displayed in the following sections this procedure will be substituted by the phrase *If the goal request is valid ...* in order to avoid repeating the same pseudo-code in every section.

Goal cancellation

Each action server has the capability of canceling the previously sent goal, even if accepted by the action server. This can be done either through the dedicated `abort` service or through pressing the `CTRL+C` key on the keyboard while on the client window of the terminal.

Once the goal request has been sent to the server the drone starts to move toward the target requested by the action server, if a cancellation is sent (trough a suitable callback) then the UAV sets back the hover mode via the control node.

Communication with the PX4 autopilot

Each action server has the possibility of communicating with the PX4 Autopilot by publishing the required trajectory setpoints, as already seen in section 2.3.2 the input to the low level controller can be either a position or a velocity array.

In this thesis work case the position setpoint has been employed for the simpler tasks, like the takeoff or reaching a given target while the velocity setpoints have been used when requiring either a trajectory tracking capability or more control in general.

The topic used to communicate to those controllers is `TrajectorySetpoint_PubSubTopic` while the PX4 odometry (`VehicleOdometry_PubSubTopic`) has been used as feedback for each action server except the precision landing one (see chapter 4 for details).

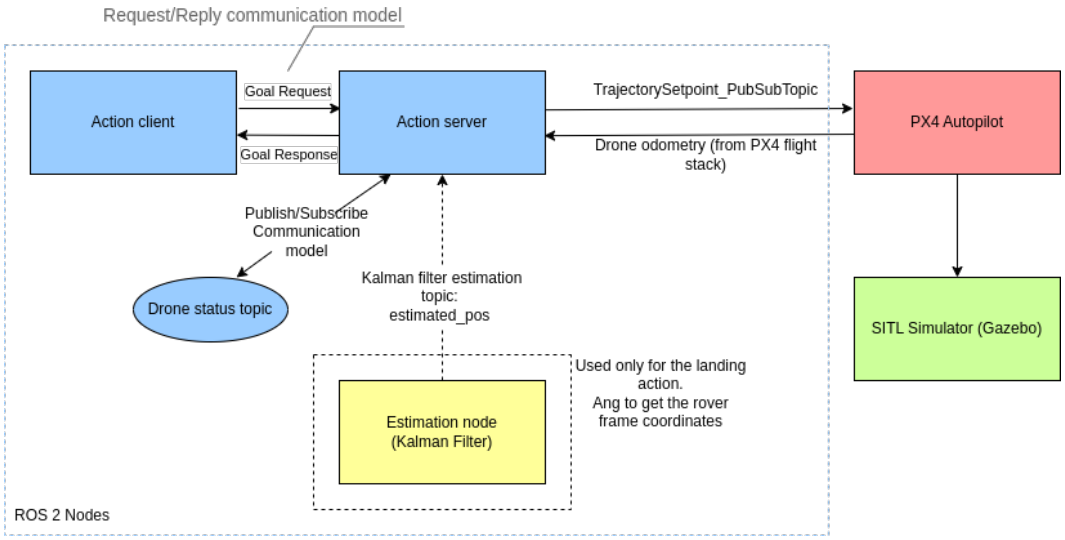


Figure 3.4: The image shows the general communication model of each action server.

3.2 Takeoff

The simplest action server, both from the user input point of view and from the algorithm one, is requested at the start of each mission.

The goal request, specified in the source code 3.3, contains only a boolean message which is set to true whenever a takeoff action is requested and the altitude of the takeoff, if different from the default one of 2 meters[1].

As already stated for every action which has been developed and therefore for the takeoff action, a client/server interface has been developed.

The client is operated by the user to send the required messages information to the action server through a terminal interface. The goal request messages are defined via a dedicate file called `Takeoff.action` (source code 3.3).

```
1    # Goal
2    bool takeoff_request
3    float64 takeoff_altitude
```

Source Code 3.3: Takeoff action goal definition

The takeoff action is the simplest of the actions, it gathers the current location of the UAV via the odometry of the drone, after which sends a setpoint request to the PX4 autopilot flight stack via the trajectory setpoint dedicated publishing topic.

The pseudo code of the action server is the following:

[1]The default value is implemented in the action client and not in the interface definition.

Algorithm 2 Takeoff action server pseudo code

Takeoff action server started
Wait for goal request
Goal request received
Generate the takeoff setpoint from the knowledge of the vehicle odometry and the required takeoff altitude
if The goal request is valid **then**
  Arm the drone
  while not The drone has reached the requested altitude **do**
    Publish the drone status code corresponding to the takeoff in progress (status code: 1)
    Publish the takeoff setpoint on the autopilot trajectory topic
    Send feedback to the client
  end while
  Publish the drone status code corresponding to the takeoff completed (status code: 2)

  Request the hover service to be set to true
  return Send result to the client
else
  Inform the client that the goal request is invalid or that the drone cannot takeoff
end if

The feedback sent to the action client is defined by the `Takeoff.action` file as well, and it is composed by the current position and the distance from the target, in this case the altitude requested.

```
1     # Feedback
2     geometry_msgs/Point current_position
3     string frame_id
4     float64 distance_to_takeoff
```

And finally the result is composed by just a boolean value regarding the action being completed or not.

```
1     # Result
2     bool takeoff_completed
```

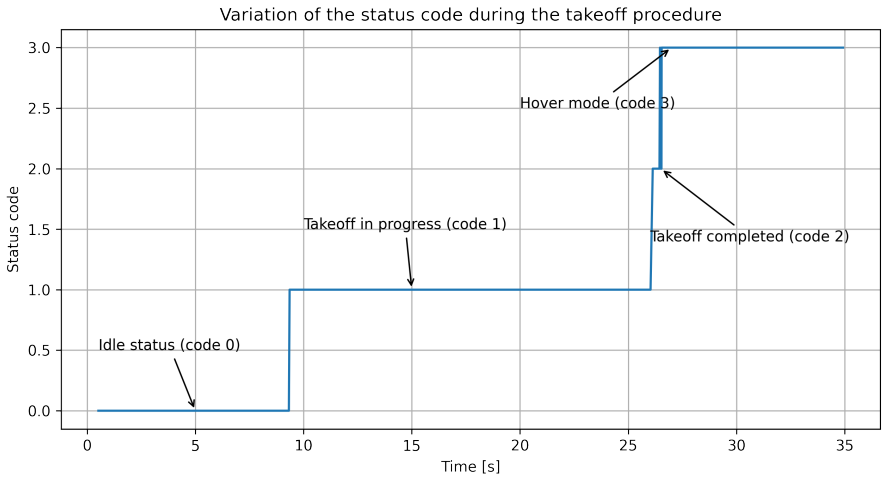


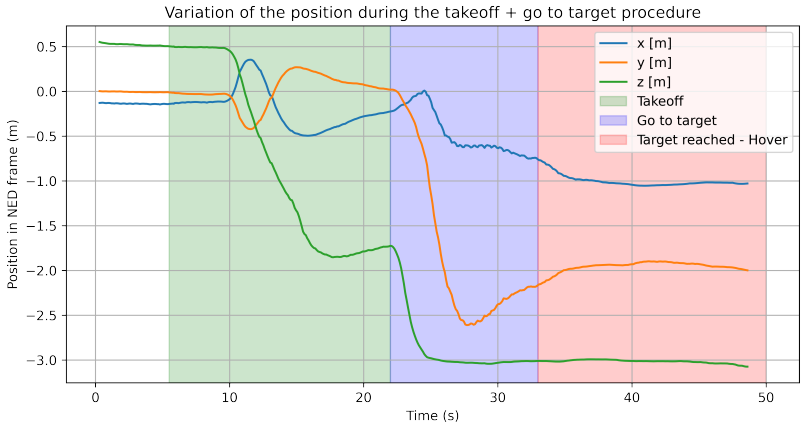
Figure 3.5: Figure showing how the status code changes during the takeoff action. As soon as the action request is received the status code changes to 1 (takeoff in progress) and similarly does when completing the takeoff, status code 2. When the action has been completed the drone enters hover mode and so the status code turns to 3.

3.3 Go to target

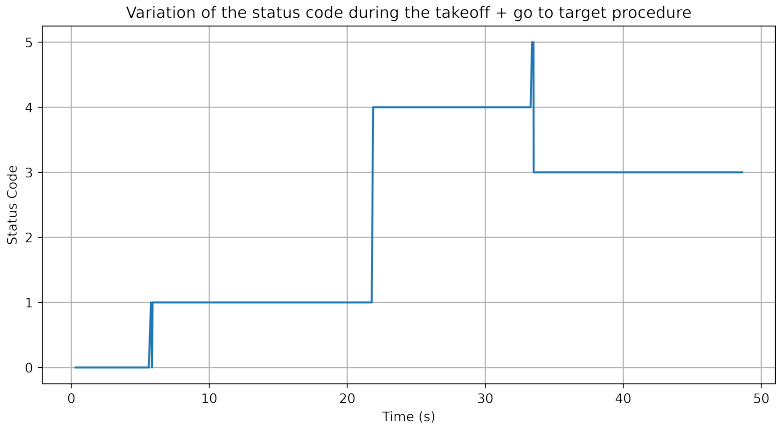
Similarly to the previous action the go to target one maintains a quite simple interface, the additional input required is the position of the target and its frame of reference.

Also, as the go to target action server shares some similarities to the takeoff algorithm it is possible to incorporate the server and to automatically perform a takeoff action if required by the user. To this end, as presented in the source code 3.4 and the implementation shown in algorithm 3, the `do_takeoff` flag is employed.

In order to do so, the drone has to be in the idle state (indicated by the status code 0, as shown in figure 3.6b). If the drone is not in idle state, i.e. the drone is already hovering (status code 3), and a takeoff is requested a warning is sent to the user and only the latter part of the action is carried out, the drone reaches the target.



(a) Variation of the drone position in the NED frame during the takeoff plus moving to target task. The target position is expressed in the NED frame through the coordinate vector (-1,-2,-3).



(b) Variation of the status code during the same task show in figure 3.6a.

Figure 3.6: Variation of the position and the status code during the takeoff and reaching target task.

The following source code shows that the `do_takeoff` flag is set to False as a default, if the user does not specify if a takeoff action is requested then it will not be done, as similarly with the takeoff altitude in the previous action.

```
# Goal
geometry_msgs/Point target_position
string frame_id
bool do_takeoff False # default: False
```

Source Code 3.4: Go to target action goal definition

The target position is specified through `geometry_msgs/Point` message type through which is possible to include 3D coordinates.

In order to specify which frame of reference is employed to determine the position of the target, the string entry `frame_id` is employed. As a default the `world` entry is used, this corresponds to the PX4 North-East-Down (NED) frame.

It is also possible to specify the `rover` frame, this requires the knowledge of the position of the rover, in order to know that the `estimation` section of the simulation must be started when using the launch files.

The steps taken by the action server are the following:

Algorithm 3 Go to target action server pseudo code

Go to target server is started
Wait for goal request
Goal request is received
if The goal request is valid **then**
  Set the hover service to false
  if The goal request contains the `do_takeoff` flag set to true **then**
    if The drone is in idle mode **then**
      Publish the drone status code corresponding to the takeoff in progress (status code: 1)
      Publish the takeoff setpoint to the autopilot trajectory topic
      → `TrajectorySetpoint_PubSubTopic`
      Send the feedback to the client
    else
      Send a warning to the user informing that the takeoff procedure already took place.
    end if
  else
    The takeoff procedure is skipped
  end if
  Set the hover service to false
  while not The drone has reached the requested target **do**
    Publish the drone status code corresponding to the go to target action in progress (status code: 4)
    Publish the target position on the autopilot trajectory topic
    Send feedback to the client
  end while
  Publish the drone status code corresponding to the go to target action completed (status code: 5)
  Request the hover service to be set to **true** at the specified target
  return Inform the action client that the target has been reached
else
  Inform the client that the goal request is invalid or that the drone cannot go to target
end if

As with the previous action the feedback and the result information are defined in the `Gototarget.action` file.

```
1     # Result
2     bool position_reached
3     ---
4     # Feedback
5     geometry_msgs/Point current_position
6     string frame_id
7     float64 distance_to_target
```

Source Code 3.5: Go to target action feedback definition

3.4 PID Controller

3.4.1 PID controller introduction

In order to provide the autopilot offboard controller, which responds to a given trajectory setpoints input, an external PID controller is employed.

This controllers computes the velocity input to be transmitted to the autopilot instead of sending directly the position request in order to have a smoother transition towards the target.

The controller is employed in the get images, area coverage and land on spot algorithms as a trajectory tracking controller.

The PID controller is employed because, even if it doesn't guarantee optimal control or control stability is broadly applicable and does not rely on the knowledge of the model [29].

As shown in a previous section (section 2.3.2) the autopilot flight stack provides with a position and attitude controllers which can be used to stabilize the drone. Instead, the proposed customizable high-level controller is used to compute the best velocity setpoints in the trajectory tracking section of each action server in which is employed.

3.4.2 PID controller architecture

A PID (Proportional integral derivative) controller is a mechanism which employs sensor feedback which is widely used. From the feedback the error value $e(t)$ is computed, which is the difference of the actual measured process variable $y(t)$ from the reference setpoint $r(t)$, the correction input $u(t)$ is applied computed from the PID terms.

In this thesis case the feedback may come from the already available PX4 Odometry messages, which then are compared to the reference position to be reached by the drone. Instead, in the precision landing case the error is computed directly through the relative position of the UAV with respect to the rover (more information is provided in section 4.4.2).

In this thesis case the control input $u(t)$ are the velocity setpoints $[v_x, v_y, v_z]$ sent to the autopilot flight stack, the interaction of the PID controller and the PX4 autopilot is shown in figure 3.8.

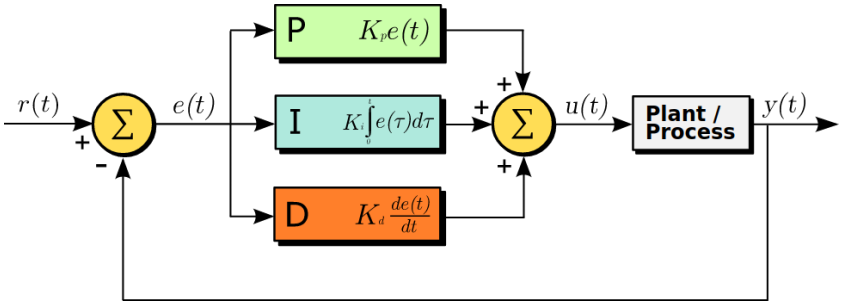


Figure 3.7: PID Controller block diagram [29]

Mathematical form of the control input

The control input expression is then:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

- K_p: proportional gain
- K_i: integral gain
- K_d: derivative gain
- $e(t) = r(t) - y(t)$: error

The positioning error value is then computed for each iteration of the PID computation, so each time the controller receives the information regarding to where the UAV should be positioned $r(i)$ and the odometry from the PX4 autopilot.

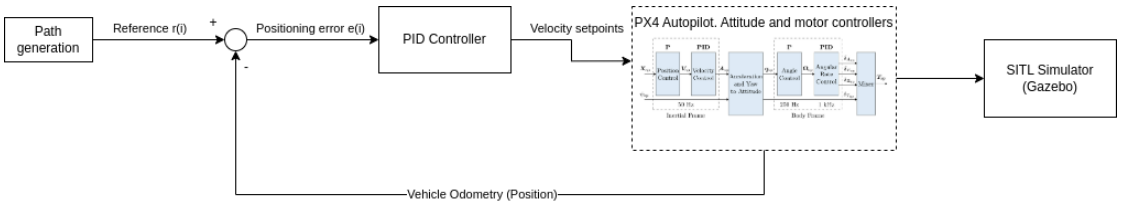


Figure 3.8: PID Controller scheme, the lower level controller used are showed more clearly in figure 2.5.

3.5 Get images

With this action server the drone has the capability of performing an orbit around a target, the user has the possibility of specifying the width of the radius and the number of images.

Differently from the previous cases, an higher number of entries is required to define the goal message, moreover the procedure followed in order to complete the task is more complex.

```
# Goal
geometry_msgs/Point target_position
float64 orbit_radius
uint8 num_images
string frame_id
```

Source Code 3.6: Get images action goal definition

The position of the target is expressed trough the same message type (`geometry_msgs/Point`) in order to maintain consistency through the interfaces. It is important to note that the position of the target will not be reached directly because the drone will orbit around it with a radius specified through the floating point message stored in the `orbit_radius` variable.

The target position entry is required to give the server the information of the position of the target, by default the frame of reference is the world NED frame but it is possible to specify a different one by sending the `frame_id` string, such as the frame centered on the rover.

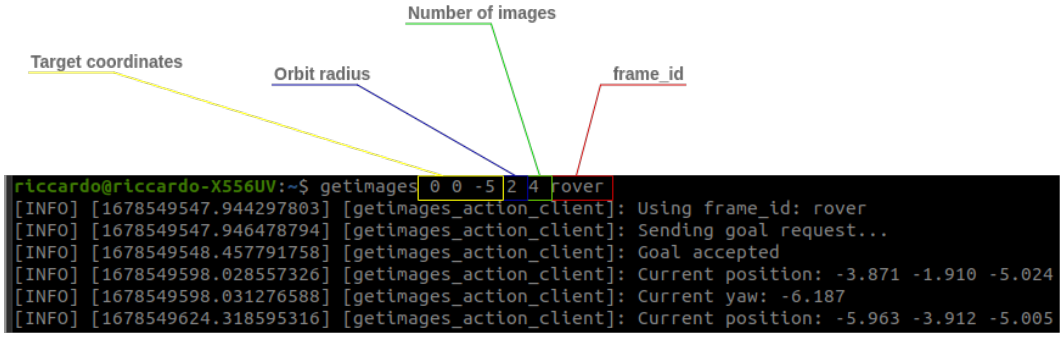


Figure 3.9: Terminal interface when requesting the get images goal. In this case the `rover` frame id is utilized, if the input would have been empty after defining the radius and number of images the reference frame employed would have been the local world frame.

Looking at figure 3.9 it is important to point out that the inputs and so the slots related to the coordinates of the target and the radius of the orbit are `float64` variables, the action clients takes care of the translation from integer to float if the user puts the input as integer as in the image above.

Still looking at the image, it can be seen how the image position are shown related to the local world NED frame even if the request has been done in the rover frame.

The integer message containing the information regarding the number of images to be acquired is used in the algorithm to compute the positions at which they need to be taken.

The corresponding indices, in the setpoint array of the orbit, are saved in an array called `image_index_arr`, its usage is described in algorithm 4.

3.5.1 Trajectory generation

Those informations are then sent to the action server and are used in order to generate the setpoints of the path that the drone will follow during the orbit. In order to do so they are sent as an input to the custom `generate_trajectory` function.

Since the function is employed to generate the trajectory meaning the circular path paired with the time information regarding when the setpoints will have to be reached, it's also necessary to specify the required tangential velocity (v_t) of the orbit.

Such value, while still editable by the user, is set by a ROS 2 parameter instead of being sent in the goal request, this is done in order to simplify the action client interface.

The path is generated by employing the following equations, the altitude setpoints information instead is kept constant during the procedure, and it is set by the user (as shown in source code 3.6).

$$\omega = v_t / r \tag{3.1}$$
$$x(t) = x_0 + r\cos(\omega(t)) \tag{3.2}$$
$$y(t) = y_0 + r\sin(\omega(t)) \tag{3.3}$$

It's also possible to get the feedforward velocities to be sent to the drone autopilot, with the following equations:

$$v_x(t) = -r\omega\sin(\omega(t)) \tag{3.4}$$
$$v_y(t) = r\omega\cos(\omega(t)) \tag{3.5}$$

With this information, although not used directly in the action server, is possible to compute the yaw angle to be sent to the drone autopilot, in order to make the drone face toward the next setpoint.

$$\psi(t) = \arctan\left(\frac{v_y(t)}{v_x(t)}\right) \tag{3.6}$$

Instead in order to maintain the drone heading toward the center of the orbit it's necessary to add $\pi/2$ to the computed value. With this yaw information it is possible for the ventral camera to point toward the target when getting the images.

It is then possible to see that the trajectory generation is a process completely done offline, in fact the UAV does not take into account the possible presence of obstacles while computing the trajectory setpoints.

Moreover, during the trajectory generation, the algorithms select the position at which to take the images, as the required number of images has been specified during the goal request and the orbit trajectory is known.

3.5.2 Trajectory tracking

Once the trajectory has been generated, the action server employs a PID controller in order to generate the correct velocity inputs to be sent to the PX4 autopilot on the trajectory setpoint topic.

The trajectory tracking is implemented in the `execute_callback` section of the algorithm, which is the following:

Similarly to all the actions, the feedback and the result information are pre-defined in the `GetImages.action` file.

```
1     # Results
2     bool get_images_completed
3     sensor_msgs/Image[] images
4     geometry_msgs/Point[] image_positions
5     float64[] image_yaws
6     string frame_id
7     sensor_msgs/Imu[] image_imus
8     sensor_msgs/NavSatFix[] image_navsats
9     float64[] image_altimeters
10    ---
11    # Feedback
12    sensor_msgs/Image current_image
13    geometry_msgs/Point current_position
14    float64 current_yaw
15    string frame_id
16    sensor_msgs/Imu current_imu
17    sensor_msgs/NavSatFix current_navsat
18    float64 current_altimeter
```

Source Code 3.7: Get images action feedback definition

The user has the possibility of choosing whether to receive the images as feedback, at soon as the drone gathers them, or getting them all at once when the drone completes the orbit.

The tracking of the path can be seen in the following image 3.10.

Algorithm 4 Get images action server pseudo code

Get images action server started
Wait for goal request
Goal request received
if The goal request is valid **then**
  Generate the trajectory → `generate_trajectory` function
  Get the positions at which to take the images
  Set the hover service to false
  Initialize the indices → $i = 0, j = 0$
  while not The drone has completed the orbit **do**
    Get the current position setpoints ← Previously generated path
    Get the current drone position from the odometry topic ← `VehicleOdometry_PubSubTopic`
    Given the current setpoint as $r(i)$ and the current position as x
    Compute the positioning error → $e(i) = r(i) - x$
    From the positioning error knowledge generate velocity setpoints with a PID controller
    Publish the drone status code corresponding to the get images action in progress (status code: 6)
    Publish the generated setpoints to the autopilot trajectory topic → `TrajectorySetpoint_PubSubTopic`
    Send feedback to the client
    if $i =$ image_index_arr(j) **then**
      Get the image from the `/camera/image_raw` topic
      Send the image to the client as feedback with the information regarding the position of the drone and the yaw.
      Increase image array → $j = j + 1$
    end if
    Increase setpoint index → $i = i + 1$
  end while
  Publish the drone status code corresponding to the get images action completed (status code: 7)
  Request the hover service to be set to true at the position corresponding to the end of the orbit
  Add the image array to the result
  return Send result to the client
else
  Inform the client that the goal request is invalid or that the drone cannot go to target
end if

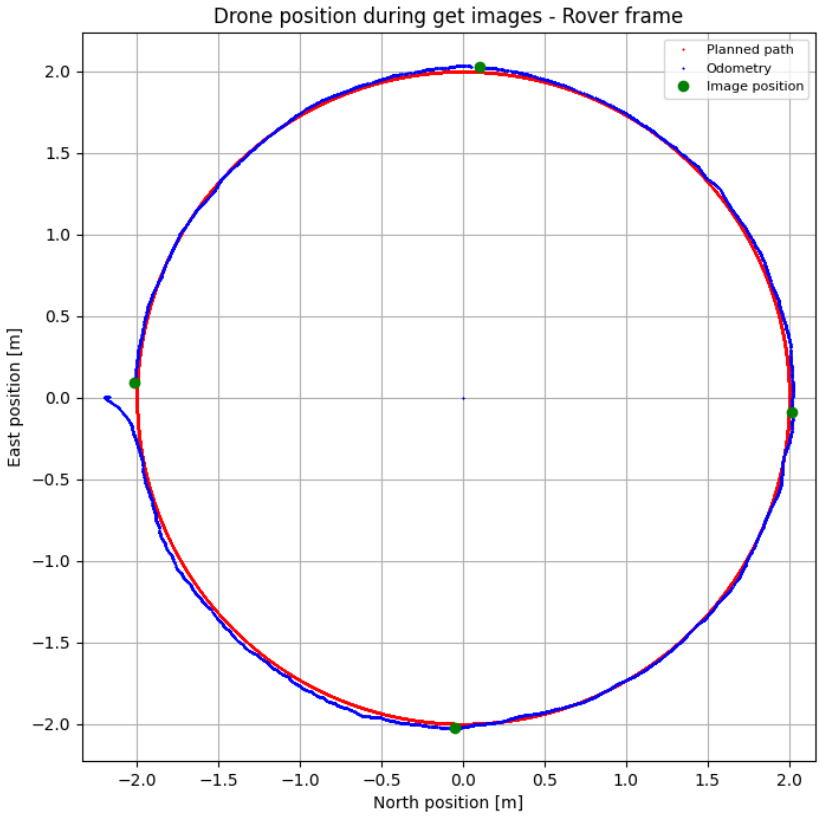


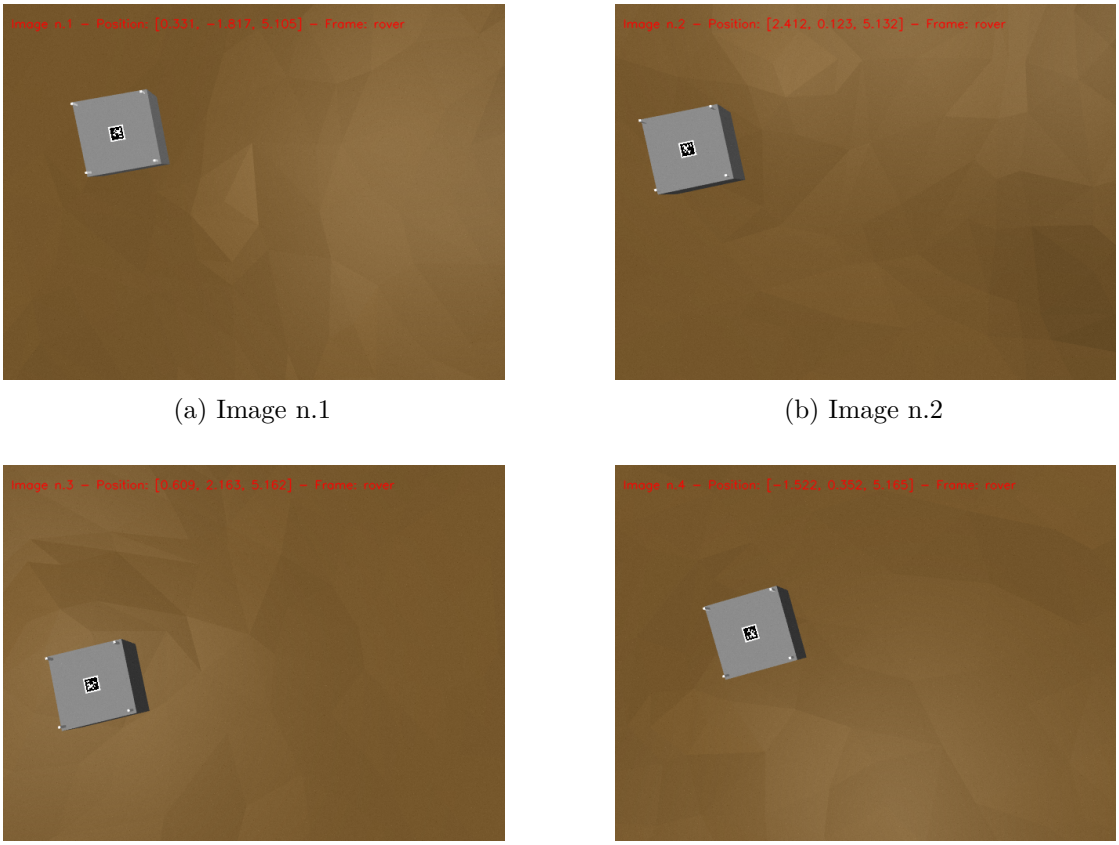
Figure 3.10: Get images action server trajectory tracking.

In order to track the generated setpoints a PID controller is employed, from which the velocity setpoints are generated with a custom frequency that can be chosen by the user, also in this case the value is defined using a ROS 2 parameter, in order to leave to the user some flexibility while maintaining a simple interface regarding the action.

During the simulation the frequency that has been chosen is of 100 Hz, which is way higher if compared to the one used in the `go_to_target` and `takeoff` actions, this is done in order to have a smoother trajectory following, but it can be easily set to a lower value in order to have a lighter simulation.

3.6 Area coverage

This action is called whenever the task requires for the user to obtain a map of the underlying terrain, the mapping is done through the imagery feed provided by the same topic as the previous action (`camera/image_raw`).



(a) Image n.1

(b) Image n.2

(c) Image n.3

(d) Image n.4

Figure 3.11

Figure 3.12: 4 images taken during an orbit, in this case the target corresponds to the rover, which is shown from each angle.

The user has to insert the variables shown at source code 3.8.

```
1     # Goal
2     geometry_msgs/Point[] grid_points
3     string frame_id
4     float64 resolution
```

Source Code 3.8: Area coverage goal interface definition

The user sends as input a set of points that define the perimeter of the area which has to be mapped, the information is completed with the `frame_id` information and so the reference frame used to define the coordinate system used for the `grid_points`.

The altitude is maintained at a constant level during the mapping procedure so a single value is requested when launching the action.

Instead, the `resolution` parameter is used by the path generation algorithm in order to define the distance between the straight segments composing the path.

```
riccardo@riccardo-X556UV:~$ gridsweep area_coverage
Enter number of grid points: 7
Grid point n. 1 :
Enter x coordinate: 0
Enter y coordinate: 0
Grid point n. 2 :
Enter x coordinate: 3
Enter y coordinate: 5
Grid point n. 3 :
Enter x coordinate: 7
Enter y coordinate: 4
Grid point n. 4 :
Enter x coordinate: 7
Enter y coordinate: 2
Grid point n. 5 :
Enter x coordinate: 4
Enter y coordinate: 1
Grid point n. 6 :
Enter x coordinate: 3
Enter y coordinate: 2
Grid point n. 7 :
Enter x coordinate: 0
Enter y coordinate: 0
Enter altitude: -5
Enter resolution: 0.25
[INFO] [1678637780.937539164] [grid_sweep_action_client]: Waiting for action server...
[geometry_msgs.msg.Point(x=0.0, y=0.0, z=-5.0), geometry_msgs.msg.Point(x=3.0, y=5.0, z=-5.0), geometry_
```

Figure 3.13: Terminal interface showed to the user when requesting an area coverage action. The user has to input the 2D coordinates of the points used to define the area. The action client takes care of translating the inputs into `geometry_msgs/Point` variables for the correct goal request, as seen in source code 3.8.

The same value is used also as the distance between the set of images taken by the drone during the action, this can be seen in figure 3.13 which shows which variables are requested to the user.

3.6.1 Path generation

In order to generate the path a library called `Python Robotics` has been used, this is an open source software project which focuses on autonomous navigation [30].

All the source code can be found in the dedicated GitHub repository [31]. In the action server only the `grid_based_sweep_coverage_path_planner.py` algorithm has been employed, although the library offers a large set of algorithms in the field of localization, path planning, mapping and path tracking.

The algorithm requires the set of vertex points of the perimeter and the resolution of the path as inputs, these are provided to the action server through the action client. The actual implementation of the algorithm takes place in the action server and returns a set of points that define the path to be followed by the drone.

In figure 3.14a the path generated by such algorithm for a square area of 5 m of side and 0.5 m of resolution is shown.

Path smoothing

In order to track a smooth path it is necessary to publish a high number of setpoints close together. As can be seen in figure 3.14a the number of points defining the area coverage

path is low and so the setpoints are spaced out. To this end a path smoothing step is implemented in the action execution callback, the result can be seen in figure 3.14b.

In order to provide the drone with a smoother path and to substitute the straight segments present when changing direction with curvilinear ones, the Numpy and Scipy library have been employed [32][33].

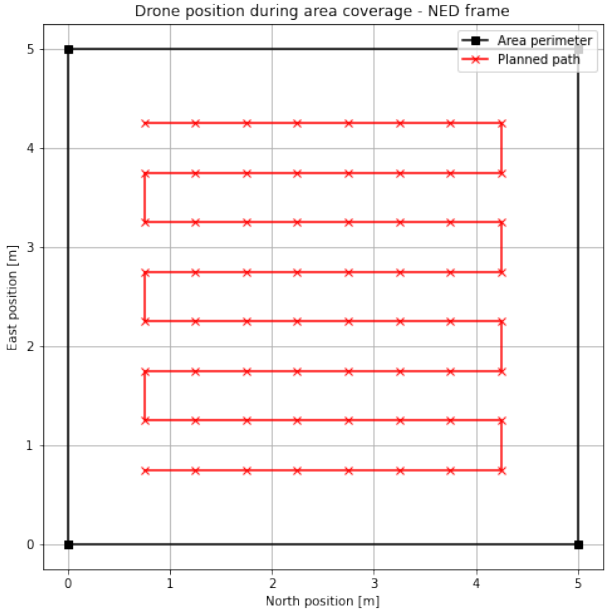
More specifically, two functions have been employed to generate the smoothed path. Those are `splprep` which is used to find the B-spline representation of a 1-D curve and `splev` which, given the knots and coefficient of the previously generated B-spline returns the 2D coordinates of the points representing the smoothed path. In the following pseudo code section (algorithm 5) the smoothing procedure has been described.

Algorithm 5 Path smoothing procedure

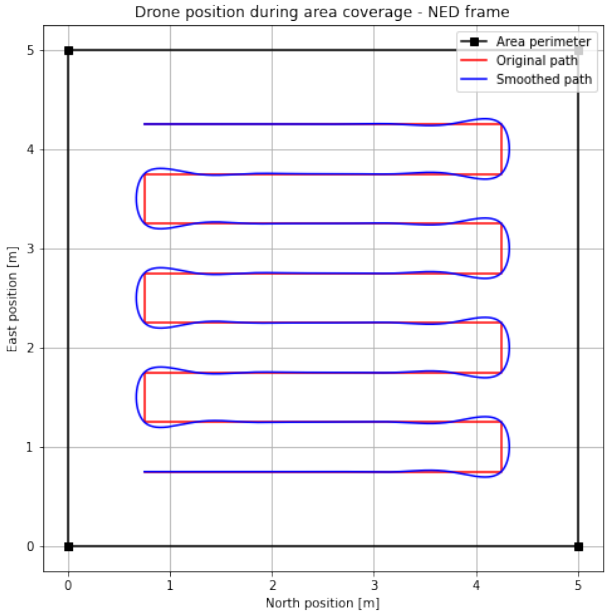
Input: Numpy array with the points coordinate defining the path → `pts`
Function: `smooth_path(pts)`
  Generate a B-spline given the path ← `splprep`
  Generate the smoothed path given the coefficients and ← `splev`
return Numpy array with the 2D coordinates of the points defining the smoothed path

The result of the smoothing process can be seen in figure 3.14b.

While it cannot be seen directly from the image, the number of setpoints describing the path is also increased, so that the drone has a smoother trajectory to follow.



(a) Path generated with python robotics algorithm, no smoothing applied. Each `x` corresponds to a point used to define the path.



(b) Path generated with the python robotics algorithm, in this case the smoothing has been applied.

Figure 3.14: Figures showing the generated path and the subsequent smoothing.

Position of the images

Since the goal is to provide to the user with a map of the area to be explored, it's necessary to take images of the surface underneath.

With the resolution entry of the action goal section the user has to set the distance between each rectilinear segments of the path. The same value is used in order to determine the distance between each position at which the drone has to take an image.

The equation written below, which is employed to compute the length of the path given an n number of setpoints, is used in a custom function called `pathlenght`. The coordinates of the setpoints composing the path are known from the offline computation of the smoothed path, proposed in the previous section.

$$L = \sum_{i=1,\dots,n} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2} \tag{3.7}$$

This information is then used in another custom function used instead to gather the position at which to take each image.

Algorithm 6 Procedure used during the computation of the position of the images to take during the area coverage

Input: Set of points composing the path → `pts`
Input: Required distance from one image position to the next → `d` (same value as `resolution` input)
Function: `path_image_pos(pts, d)`
for Each points defining the path **do**
  Given the path distance between two points as L (eq. 3.7)
  if $L \geq d$ **then**
    Save the index of the point to an array
  end if
end for
return Index array

At the end of this process the action server is provided with an array of integers (`image_index_arr`) which correspond to the position at which the drone has to send the image from the `camera/image_raw` ROS 2 topic and publish it on the feedback and finally the result topics defined in the action, as show in algorithm 7.

Feedback and results topics

As with the goal topics the type of messages sent for the feedback and the results are defined in the `AreaCoverage.action` file, as shown in source code 3.9.

```
1     # Result
2     bool sweep_completed
3     sensor_msgs/Image[] images
4     geometry_msgs/Point[] image_positions
```

```
5     float64[] image_yaws
6     string frame_id
7     sensor_msgs/Imu[] image_imus
8     sensor_msgs/NavSatFix[] image_navsats
9     float64[] image_altimeters
10    ---
11    # Feedback
12    geometry_msgs/Point current_position
13    sensor_msgs/Image current_image
14    float64 current_yaw
15    string frame_id
16    sensor_msgs/Imu current_imu
17    sensor_msgs/NavSatFix current_navsat
18    float64 current_altimeter
```

Source Code 3.9: Area coverage action feedback and result definition

3.6.2 Path tracking

Similarly to the previous action a PID controller has been employed in order for the UAV to track the path generated during the previous section.

The controller takes as input the positioning error and provides the UAV with the velocity setpoints as output, these are published on the `TrajectorySetpoints_PubSub` topic, a more detailed view of the implementation is provided in the dedicated section 3.4.

The complete pseudo code of the area coverage algorithm is presented below:

Algorithm 7 Pseudo code of the area coverage action server

Start the area coverage server
Wait for the goal request
Get the goal requests from the action client
Call the hover service and set it to false
Initialize the indices $\rightarrow i = 0, j = 0$
while not Drone has completed the grid sweep **do**
  Get the current position setpoints $\leftarrow$ Generated path
  Get the current drone position from the odometry topic $\leftarrow$ `VehicleOdometry_PubSubTopic`
  Given the current setpoint as $r(i)$ and the current position as x
  Compute the current positioning error $\rightarrow e(i) = r(i) - x$
  Generate the velocity setpoints from the positioning error $\leftarrow$ PID controller algorithm

  Publish the drone status code corresponding to the area coverage in progress (status code: 10)
  Publish the computed velocity setpoints to the autopilot trajectory topic $\rightarrow$ `TrajectorySetpoint_PubSubTopic`
  if $i =$ image_index_arr(j) **then**
    Get the image from the `/camera/Image_raw` topic
    Publish the image to the feedback topic
    Publish the current position and yaw of the drone as feedback
    Increase image array $\rightarrow j = j + 1$
  end if
  Increase setpoint index $\rightarrow i = i + 1$
end while
Publish the drone status code corresponding to the area coverage completed (status code: 11)
Call the hover service and set it to true in correspondence to the last area coverage setpoint
return Send the result of the action to the client

The drone, then, is able to follow the desired trajectory and to gather a set of image at the user-defined distance. The result of the tracking can be seen at figure 3.15. With the green marks are indicated the positions at which the images are taken.

The position of the drone is expressed in the world NED frame.

It is evident how the drone is able to track the computed path, the tracking happens at a low velocity due to the nature of the action.

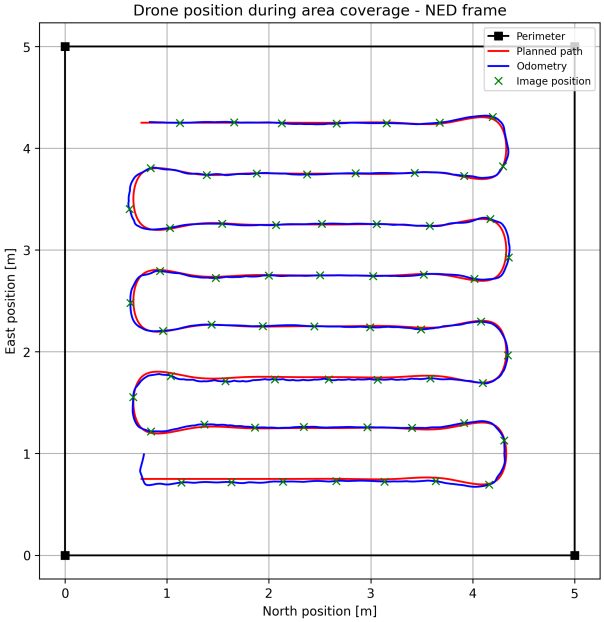


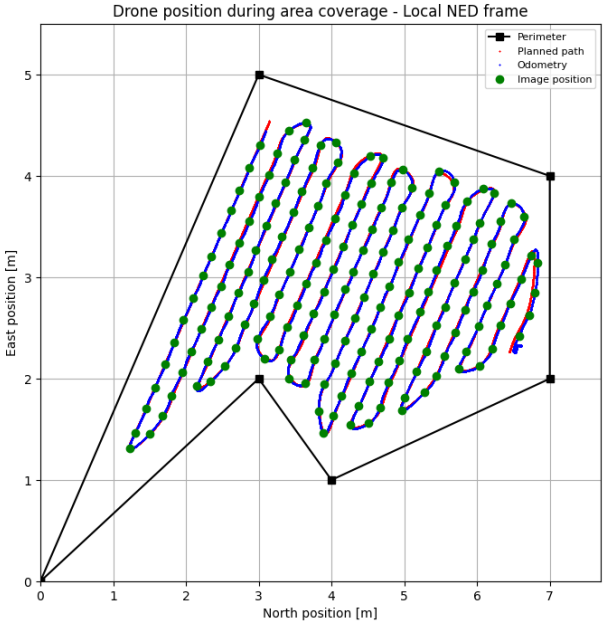
Figure 3.15: Drone effective position (blue line) with respect to the generated path (red line) during the area coverage. The position at which the images are taken are flagged with the green marks.

Irregularly shaped areas

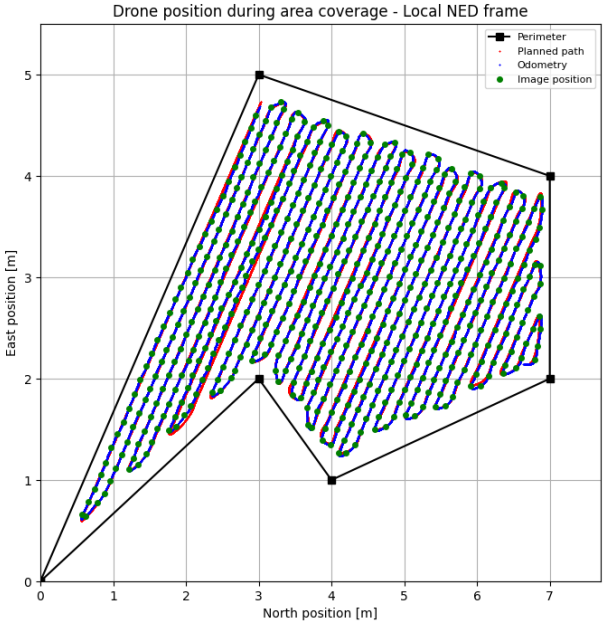
The action server also offer the possibility of covering convex or, in general, irregularly shapes areas.

As shown in figure 3.16 a lower (narrower) resolution results in a better mapping of the terrain, the irregular shape is covered better than with an higher resolution.

This comes at a cost though the number of images taken during the task increases, 185 with a resolution of 0.25 meters (figure 3.16a) versus 603 in the latter lower resolution (0.15 meters) case (figure 3.16b). This in turn will result in an higher time related to the mapping process.



(a) Resolution set to 0.25 meters.



(b) Resolution set to 0.15 meters.

Figure 3.16: Coverage of an irregularly shaped area with different resolutions.

Chapter 4

Precision landing and estimation

In order to estimate the relative position of the UAV with respect to the rover, a Kalman filter has been employed. The input of the filter employs the use of two additional sensor data to the ones already provided by the PX4 flight stack.

The first additional measurement comes from an ultra-wide band (UWB) system, from which it is possible to estimate the relative distance between the rover and the drone, while the second sensor is a camera, which is paired with an Apriltag marker placed on the rover landing platform, then through an estimation algorithm it is possible to estimate the relative position of the drone with respect to the rover.

All this sensor outputs are then fed to the Kalman filter which will return an estimate of the relative position of the drone with respect to the rover.

The starting point of this section has been the following theses: [34] and [35] as they also provide with a Kalman filter implementation for an UAV.

In the works cited only the ultra-wide band technology has been employed as input to the Kalman filter in order to retrieve the relative position between the rover and UAV. Then, when the Apriltag marker is detected, the control algorithm switches to the relative position estimate provided by the Apriltag algorithm.

In this thesis work instead the Apriltag estimation data is implemented as a Kalman filter input, thus allowing to select upon landing request which additional information is employed, then the simulation environment is used in order to test and analyze the result of the various selections.

Nonetheless the Kalman filter implementation and the organization follows the one provided in the cited theses although the modifications mentioned above were made in order to improve the flexibility of the algorithm.

4.1 Kalman Filter theory

First of all, the theory related to the Kalman filter has to be shown. A Kalman filter is a recursive algorithm which is used to estimate of unknown variables such as the state of a system, starting from a series of measurements over time paired with statistical noise and inaccuracies [36].

The algorithm works with a two-phase process. During the prediction step the filter estimates the state variables, given the system dynamic equations knowledge. In the update step, instead the measurements gathered from the sensors are used to update such estimates [36].

4.1.1 Prediction step

Below it is shown the model employed during the prediction step, the state x_k at time k is derived from the value of the state at time $k-1$. Then in order to show that these values are estimated the notation $\hat{x}$ is used.

$$\hat{x}_k = F_k \hat{x}_{k-1} + B_k u_k + w_k \tag{4.1}$$

- $\hat{x}_k$: state estimate at time k
- F_k: state transition matrix, applied to the previous state x_{k-1}
- B_k: control input matrix, applied to the control input vector u_k
- u_k: control input vector
- w_k: process noise, assumed to be a zero mean multivariate normal distribution $\mathcal{N}$ where the covariance matrix is Q_k: $w_k \sim \mathcal{N}(0, Q_k)$

During this step the filter provides with a prediction of the state of the system, using the 4.2 and 4.3 equations, this does not include the observation (z_k) information from the current k step.

$$\hat{x}_{k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k \tag{4.2}$$

$$\hat{P}_{k|k-1} = F_k \hat{P}_{k-1|k-1} F_k^T + Q_k \tag{4.3}$$

4.1.2 Update step

Instead, during the update step the model related to the observed measurements z_k is shown as follows.

$$z_k = H_k x_k + v_k \tag{4.4}$$

- H_k: observation model

- v_k: observation noise, assumed to be a zero mean Gaussian white noise with covariance matrix R_k: $v_k \sim \mathcal{N}(0, R_k)$

In the update section of the algorithm, the observation information regarding the current k timestep is taken into account refining the state estimate.
The update step is performed through the following equations:

Innovation residual:

$$\tilde{y}_k = z_k - H_k \hat{x}_{k|k-1} \tag{4.5}$$

Covariance innovation:

$$S_k = H_k \hat{P}_{k|k-1} H_k^T + R_k \tag{4.6}$$

Optimal Kalman gain:

$$K_k = \hat{P}_{k|k-1} H_k^T S_k^{-1} \tag{4.7}$$

Updated (a posteriori) state estimate:

$$x_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k \tag{4.8}$$

Updated (a posteriori) covariance estimate:

$$P_{k|k} = (I - K_k H_k) \hat{P}_{k|k-1} \tag{4.9}$$

Measurement residual:

$$\tilde{y}_k = z_k - H_k \hat{x}_{k|k} \tag{4.10}$$

Tuning the Kalman Filter

The tuning parameters of this algorithms are:

- Q_k: Covariance matrix of the process noise.
- R_k: Covariance matrix of the observation noise.

The value of such parameters must be appropriately selected, it is possible to do so trough a trial and error procedure. For this to be achieved, the result of the Kalman filter has been compared to ground-truth data coming from the simulation software, more details about the ground-truth data are shown in section 4.2 and 4.3.

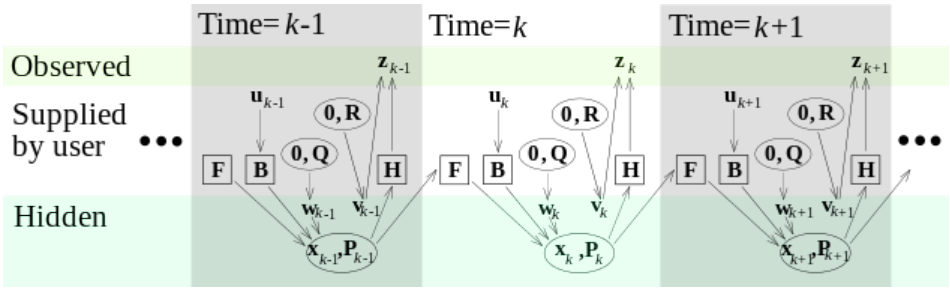


Figure 4.1: Model underlying the Kalman Filter. The matrices are represented by the squares and ellipses represent the normal distribution (Q and R are the covariance matrices). The values with no geometric box are the vectors (input, state, measurements)[36].

4.2 Kalman Filter implementation

4.2.1 Estimation node introduction

The estimation algorithm main goal is to provide the UAV with the relative position and velocity to the rover. In this way it's possible for the UAV to perform a precision landing and to perform most of the actions described in chapter 3. Also, it provides the algorithms the ability to perform the tasks by selecting the coordinates relative to the UGV position instead of getting the coordinates in the local world frame.

FilterPy Library

For the Kalman filter code implementation, the FilterPy Python library has been employed. This library implements the steps and computations shown in the previous section 4.1 into easily employable functions (e.g. `KalmanFilter.predict()` and `KalmanFilter.update()`)[37].

4.2.2 ROS 2 topics

As already stated the Kalman filter algorithm employs a number of measurements which are then filtered, such estimates come from a number of ROS2 nodes and subsequent sensors fitted on the UAV and UGV, this data flow is carried out through ROS2 topics, shown in the table below (the table does not include the namespaces which are instead shown in the `rqt_graph` image 4.2).

Measurement	ROS 2 Topic
Relative distance between UGV and UAV obtained through the UWB sensor.	`norot_pos`
Position of the UAV expressed in the world NED frame. Measurement offered by the autopilot flight stack.	`VehicleLocalPosition_PubSubTopic`
Estimated yaw of the UGV expressed in the world NED frame.	`estimated_yaw`
Relative position between the UGV and UAV retrieved through the Apriltag estimation algorithm	`estimated_pos`
Relative altitude of the drone, acquired through a laser altimeter	`DistanceSensor_PubSubTopic`

Table 4.1: ROS 2 topics employed in the Kalman filter node.

All such topics incoming to the Kalman filter estimation node can also be visualized through the `rqt_graph` tool.

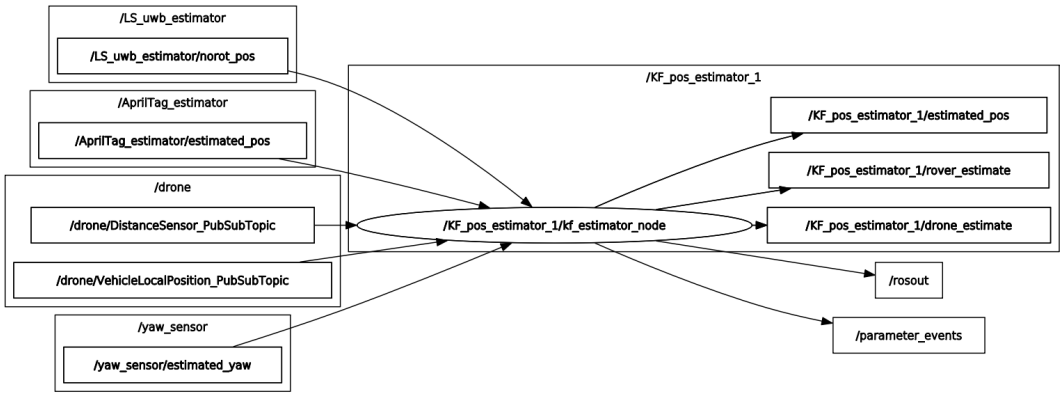


Figure 4.2: Image taken from the `rqt_graph` showing the incoming and outgoing topics related to the Kalman filter node. Topics are shown with a rectangular shape while the oval shape is related to the node.

Kalman filter node organization

As shown in figure 4.2 the Kalman filter estimator node is subscribed to each topic pertaining to the estimation algorithm. The data may come directly from sensor or through an algorithm like in the case of the AprilTag and the ultra-wide band sensor.

Ultra wide band measurements

The ROS2 node gathers the ultra-wide band data, through the `norot_pos` topic the position of the UWB tag, implemented on board of the UAV, is known.

The position, though, is expressed in the rover reference frame. Since the outgoing topic of the Kalman filter node is expressed in a local NED frame, it's necessary to rotate such measurement.

In order to perform the rotation from the UGV frame to the NED frame it is necessary to know the yaw angle of the rover, in the NED frame.

This computation can be implemented in the Python algorithm through the `scipy` [1] library more specifically with the `from_euler` method. Knowing the value of the UGV yaw ψ_{UGV} the rotation corresponds to the following rotation matrix.

$$R_{UGV}^{NED} = \begin{bmatrix} cos(\psi_{UGV}) & -sin(\psi_{UGV}) & 0 \\ sin(\psi_{UGV}) & cos(\psi_{UGV}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.11}$$

Having computed this, it's possible to rotate the `norot_pos` information in a NED frame.

$$\hat{p}_{UAV}^{NED} = R_{UGV}^{NED} \cdot p_{UAV}^{UGV} \tag{4.12}$$

The result of this computation is the position of the UAV in a NED frame centered in the rover origin.

The measurement used for the Kalman filter update step are thus the relative position of the UAV in the NED frame, shown as follows:

$$z_{UWB} = \begin{bmatrix} \hat{x}_{rel,UAV}^{NED} \\ \hat{y}_{rel,UAV}^{NED} \end{bmatrix} \tag{4.13}$$

As explained in the section relative to the update step (section 4.1.2) to the measurement are then added the covariance matrix R_{UWB}.

AprilTag measurements

The AprilTag data comes from the topic `/AprilTag_estimator/estimated_pos` published by `apriltag_estimation_node.py` which employs the `dt_apriltags` library in order to estimate the pose of the UAV from the camera feed, published on the `/camera/image_raw` topic.

[1] The precise library is the `scipy.spatial.transform`.

Figure 4.3: Incoming and outgoing topics from the AprilTag estimation node.

The `dt_pariltags` library is used in order to detect the pose of the AprilTag marker from the UAV camera feed.

Vehicle Local Position

The data relative to the position of the drone in the local (NED) frame is published directly by the PX4 autopilot, so no additional estimation node is necessary.

$$z_{PX4} = \begin{bmatrix} x^{NED} \\ y^{NED} \\ z^{NED} \end{bmatrix} \tag{4.14}$$

The Kalman filter algorithm receive the PX4 measurements in the local NED frame. So while the input data is used to update the filter in the dedicated section it will be necessary to translate the data to the local NED rover frame in the publishing section of the code.

UGV yaw

The rover is equipped with a plugin that publishes the odometry on the `rover_odom` topic, from which is straightforward, with the help of the `scipy` library to compute the yaw angle of the UGV in the world NED frame.

$$z_{\text{UGV yaw}} = \psi_{UGV} \tag{4.15}$$

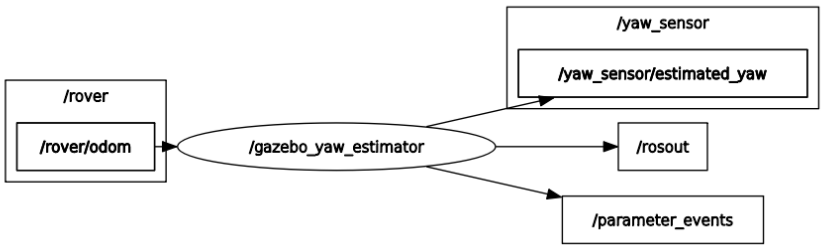


Figure 4.4: Rqt graph showing the incoming and outgoing topics of the rover odometry.

UAV Altitude

In the `.sdf` file, as explained in section 2.4.2, is implemented a plugin simulating a laser altimeter, the `libgazebo_ros_ray_sensor` gazebo plugin.

In the estimation section of the thesis, the `sensor_msgs/Range` message type has been chosen, this assures that a single value is returned to the algorithm and not a series of estimations (as shown in section 2.4.2).

4.2.3 Update step

Each of the previously explained measurements are fed to the estimation algorithms through a publish/subscribe communication model, moreover to each of the topics correlated to the measurements a callback method is associated.

In each callback methods the update step, explained in section 4.1.2, is provided by the `FilterPy` library and is implemented through the `KalmanFilter.update(z, R)` method.

The following pseudo code (8) shows the code that is implemented in a callback timer. The frequency of such timer is twice the set frequency of the prediction step. This value is adjustable via ROS2 Parameters which can be varied in the `estimation` launch file or via the dedicated terminal service. By doing so it's possible to balance the UAV application with a steady stream of estimations while tuning the frequency parameter in order not to overload the computation request for the algorithm.

The following update step is requested every 10 Hz. It will update the Kalman filter only if it has received a new measurement since the last update step. This is done only as a precautionary measure, as the measurements are fed to the algorithm with an higher frequency than that requested by the update step callback. This check is helpful, though, in the case of the measurement related to the apriltag, which are only available if the marker is visible by the ventral camera of the UAV. Also it is used in order to avoid using the incoming data if it has not been requested by the user (as explained in details in section 4.3.1 and 4.4.1).

Algorithm 8 Pseudo code demonstrating the update step of the Kalman filter algorithm

if UWB measurements are available and they have been requested via the ROS 2 paramters **then**
  Update the kalman filter with the z_{UWB} measurements and the covariance matrix R_{uwb}
end if
if PX4 measurements are available **then**
  Update the kalman filter with the z_{PX4} measurements and the covariance matrix R_{PX4}
end if
if UGV compass measurement is available **then**
  Update the kalman filter with the z_{UGVyaw} measurements and the covariance matrix $R_{compass}$
end if
if Altimeter measurement is available **then**
  Update the kalman filter with the $z_{\text{range sensor}}$ measurements and the covariance matrix $R_{\text{range sensor}}$
end if
if Apriltag measurements are available and they have been requested via the ROS 2 Parameters **then**
  Update the kalman filter with the z_{tag} measurements and the covariance matrix R_{tag}
end if

The `z` variable represents the measurement data, while the `R` variable represents the covariance matrix relative to the incoming data.

Each of the measurements has a different covariance matrix, all of the matrix are tuned in order to obtain the best estimation, the covariance matrices are R_{UWB}, R_{tag}, $R_{\text{range sensor}}$, R_{compass} and R_{px4}.

Since all matrices are diagonal it is possible to define them via a single value that can be associated to a ROS parameter for ease of use during the tuning process.

Algoritm troubleshooting

All the measurement are published asynchronously and with different frequencies with respect to the prediction step and to them.

During the development phase of the algorithm, the filter update was embedded in each sensor callback, so the filter was updated at different frequencies as soon as the new data was available. These frequencies were up to 100 Hz, so in order to have a lighter algorithm from the computation standpoint all the update steps were implemented in a single callback with a fixed and lower frequency of 10 Hz.

4.2.4 Prediction step

The prediction step implements the `KalmanFilter.predict()` method available through the FilterPy library [37]. Similarly to the update step it's performed at a fixed frequency.

In the same callback of that contains the `predict()` method it is present the publishing section of the algorithm, so that every time the algorithm publishes the most up to date information on the `estimated_pos` topic (as shown in Figure 4.2).

4.3 Kalman filter tuning

The tuning process of the Kalman filter algorithm has been carried out by modifying the value of the covariance matrices R associated to the observation noise and Q associated to the process noise.

To tune all these parameter a ground-truth information is used, this is gathered through the `libgazebo_ros_p3d` plugin which is used on the UAV and retrurns the position of the model in the gazebo simulator.

Since the gazebo reference frame is ENU while the reference frame used by the PX4 autopilot and for the estimation and navigation algorithm is NED it is necessary to perform the rotation described by equation 2.1. To this end a simple ROS2 node performing this rotation is employed which publishes on the ground-truth topic.

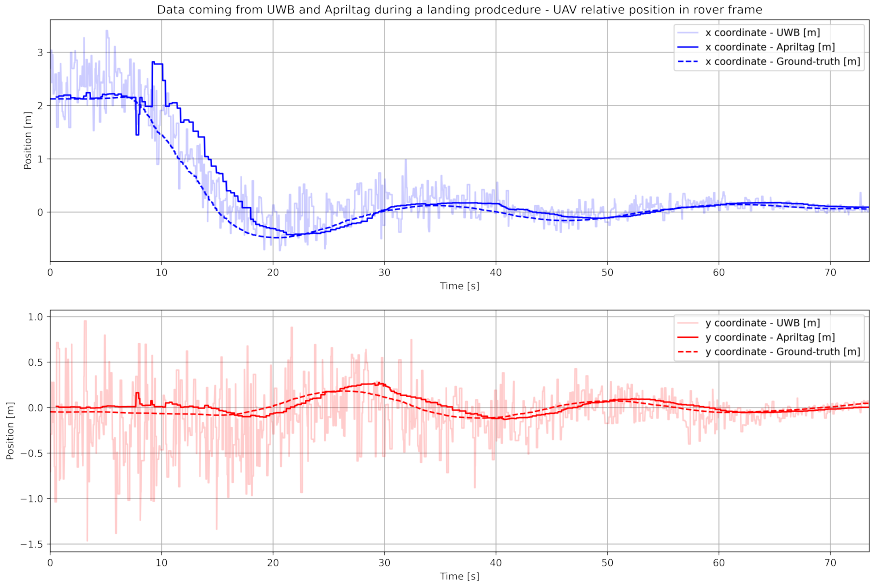


Figure 4.5: Comparison of the data incoming from the ultra-wide band and apriltag algorithms. Used during the tuning process to choose the R value.

In order to choose the best R value related to each incoming data stream, comparison have been made as showed in figure 4.5. As can be inferred from the image, the value related to the ultra-wide band will be greater with respect to the one related to the AprilTag, all the following testing will be done with a value of $R_{uwb} = 2m$ and $R_{tag} = 0.5m$, these value can be set and tuned in the estimation launch file `drone_rover_positioning.launch.py` (the launch file previously indicated in figure 3.1).

4.3.1 Estimation launch file

For ease of use all the nodes related to the estimation section of the code are grouped and launched simultaneously through single Python launch file, `drone_rover-_positioning.launch.py`. The organization may be seen in the following figure.

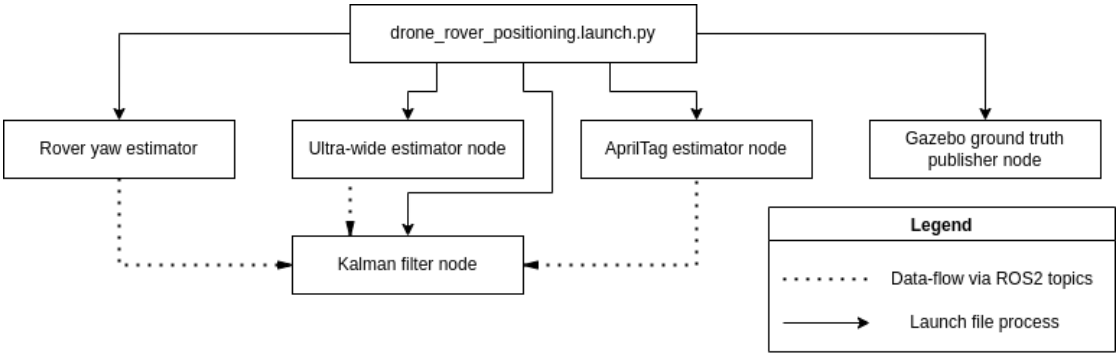


Figure 4.6: Estimation launch file organization

It is possible to see that the ground-truth node, and the related topic as also shown in figure 4.2, do not share directly data with the algorithm, it is just used for comparison purposes.

The usage of the ground-truth data will be show in section 4.5.

ROS 2 Parameters

As previously stated each of the covariance matrices is associated to a ROS2 parameter, it is possible to vary the parameter directly in the launch file without the need to modify it directly in the Kalman filter algorithm, moreover it is possible to request to change the parameter via either the command line interface or an external algorithm itself by employng the `SetParameters` service available through the `rclpy` ROS2-Python interface library. This service has been used extensively to change the sensor employed by the Kalman filter node during the comparison process.

4.4 Landing

While the landing action is not as complex from the algorithm point of view as the area coverage, since it is not required to track a trajectory or to provide the client with a set of images, is the most complex action from the interface organization standpoint.

The goal request, provided by the user, allows for the action server to select the usage of different sensor in the relative position estimation node.

As already specified in chapter 4 the Kalman filter node has the capability to select additional information coming from:

- The **ultra-wide band** sensor and thus the relative position estimation.
- The pairing of the camera and an **AprilTag** marker, again providing with the relative position between the drone and the rover.

By defining the action interfaces with the dedicated file (`LandOnSpot.action` the source code for the goal section is shown at 4.1) it's possible to analyze the goal request options.

```
1    # Goal
2    geometry_msgs/Point landing_spot
3    string frame_id
4    bool use_uwb
5    bool use_marker
```

Source Code 4.1: Land on spot action goal definition

Although the best case scenario is the one related to the usage of all the available information coming from the sensors. It is still necessary to be assured whether the algorithm is effective in case one of the sensor is not available in a real world setting.

And then its necessary to check what could be the possible shortcomings of the estimation algorithms and subsequently of the landing action in case not all the information is available to use.

From this requirement, then, it is possible to produce four cases corresponding to the usage of each additional data pairing.

4.4.1 Interface between the action and the estimation algorithm

The usage request of the two sensors is specified in the goal request, but it is necessary to point out that the client and the estimation node are not directly connected through the action interface.

It is necessary to add a prior step to the landing algorithm in which the `SetParameters` service is used in order to call the ones of the estimation node.

The goal request messages, already defined in the relative file, are then associated to ROS 2 parameters of the Kalman filter estimation node. Such parameters are:

- `use_uwb`: Which is a boolean value related to the usage of the ultra-wide band sensor in the Kalman filter node.

- `use_tag`: A boolean value expressing whether the node will be employing the apriltag information or not.

- `rover_position`: A double array message expressing the external estimate of the position of the rover in the world frame.

The rover position information is requested automatically by the client if the usage of the ultra-wide band is set to false. This information can be used both as a rough estimation in case if paired with the apriltag to give to the drone some sort of information of where to go to locate the apriltag, when this is located then a more precise information is fed through the estimation which is then used for the landing process.

In case neither the ultra-wide band and apriltag information are used by the estimation algorithm there is no possibility of estimating the relative position between the UAV and the rover. The landing action server then requires the landing position information, through the `landing_spot` message. This can be either the rover position information or the another spot available for landing.

4.4.2 Landing action pseudo-code

The landing action server is based on the one proposed in this thesis [34] although there are some modifications.

The adaptive, or gain scheduled, PID controller usage is shared and so is the usage of the thresholds associated to the approach. Instead the information related to the relative position and velocity coming from the Kalman filter depends on the goal request.

In the prior case the landing algorithm implements the apriltag information in the Kalman filter node instead of switching from a Kalman filter information not based on the marker information to the relative position computed solely on the marker whenever this one is detected.

The reasons for doing so are that by implementing the apriltag information directly in the estimation node allows to test the different cases explained in the previous section without making changes or adjustments to the source code itself, and thus using a single algorithm both for the estimation and the action part.

Thanks to the fact that the ROS 2 nodes are organized through actions the user has the possibility, through the action client interface, to select which additional data wants to use upon landing request.

Algorithm 9 Landing algorithm pseudo-code

The landing server is started
Wait for the goal request
Receive the goal request from the action client
if The goal request is valid **then**
  Request the hover mode to be set off, via the dedicated service
  Send the requested parameter value to the **estimation algorithm** via the `SetParameters` service
  while not Landing is completed **do**
    Publish the drone status relative to the landing in progress (status code: 8)
    Get the relative position to the rover ← **Kalman filter algorithm**
    Compute the velocity input through the adaptive **PID algorithm**
    Publish the computed inputs on the offboard control topic
  end while
  Publish the drone status relative to the completed landing (status code: 9)
  Disarm the drone
  Publish the drone status relative to the idle mode (status code: 0)
else
  Inform the action client that the goal requested is not valid or the server is not available
end if

Similarly to the goal request the feedback and result messages are defined in the `LandOnSpot.action` shown in the source code 4.2.

```
1     # Result
2     bool landing_completed
3     ---
4     # Feedback
5     geometry_msgs/Point current_position
6     string frame_id
```

Source Code 4.2: Landing feedback and result messages definition

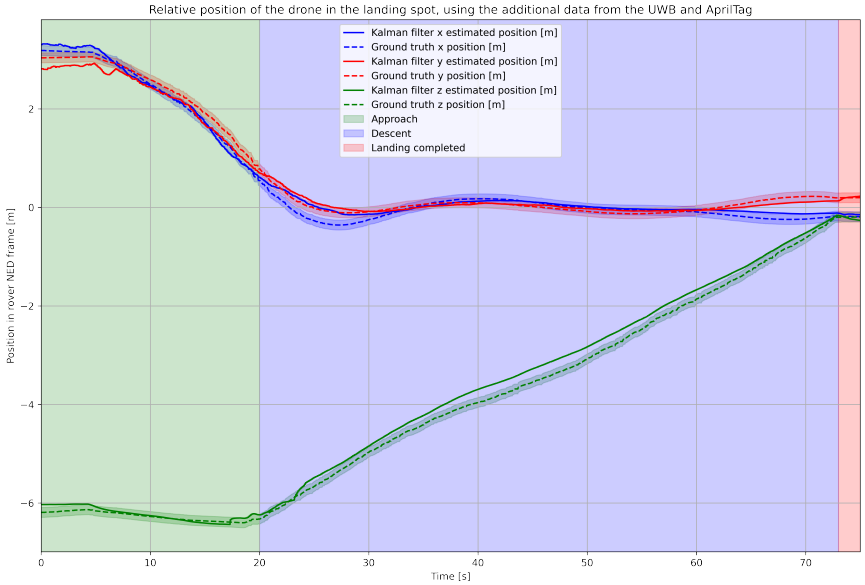


Figure 4.7: Distinction between the approach and descent phases in the landing algorithm.

4.5 Precision landing simulation results

In order to test the different capabilities of the estimation algorithm in different conditions represented by the different set of sensor employed during the landing action SITL simulations have been made.

4.5.1 Simulation with only ultra-wide band sensor

In this case the simulations is used in order to simulate the behavior of the estimation and landing action algorithms whenever only the ultra-wide band sensor is available while the AprilTag is not.

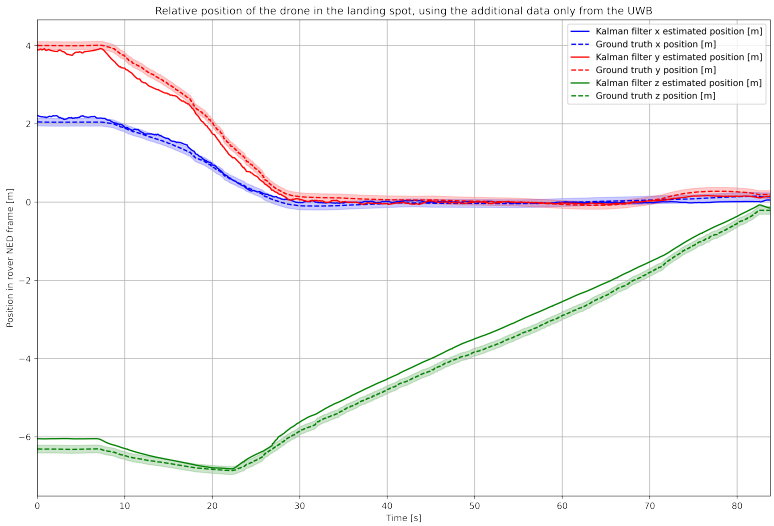


Figure 4.8: Kalman filter estimation output compared to ground-truth data. Case with UWB data and no AprilTag data.

This is imposed by the user via the ROS 2 parameter flags, `use_uwb` which is set to true while the `use_marker` set to false.

From figure 4.10 it is possible to see a visible discrepancy between the z estimated data and the real simulation data especially when the UAV is flying at an higher altitude, this effect is compensated when the drone lowers itself on the landing platform.

4.5.2 Simulation with only the AprilTag sensor

Similarly to the previous case the ROS 2 parameter are set when sending the goal request with the action client in this case the `use_uwb` will be set to false while the `use_marker` will be set to true.

The figures 4.10 and 4.11 refer to a test in which the rover position indication was deliberately incorrect respect to the real rover position. This has been done in order to test the algorithm capability in correcting the rover estimate once the marker is detected.

```
riccardo@riccardo-X556UV:~$ land_on_spot False True
[INFO] [1678876780.375210229] [landonspot_action_client]: Not using UWB
Enter rover position x : -3
Enter rover position y : -5
Enter rover position z : 0
[INFO] [1678876784.604779777] [landonspot_action_client]: Using apriltag
[INFO] [1678876784.605542180] [landonspot_action_client]: Rover position: [-3.0, -5.0, 0.0]
[INFO] [1678876784.610221074] [landonspot_action_client]: Goal accepted
[INFO] [1678876784.647833976] [landonspot_action_client]: Feedback: -0.025 -1.947 -6.131
[INFO] [1678876784.682580705] [landonspot_action_client]: Feedback: -0.024 -1.947 -6.131
```

Figure 4.9: Terminal window shown when a precision landing with no UWB sensor is requested.

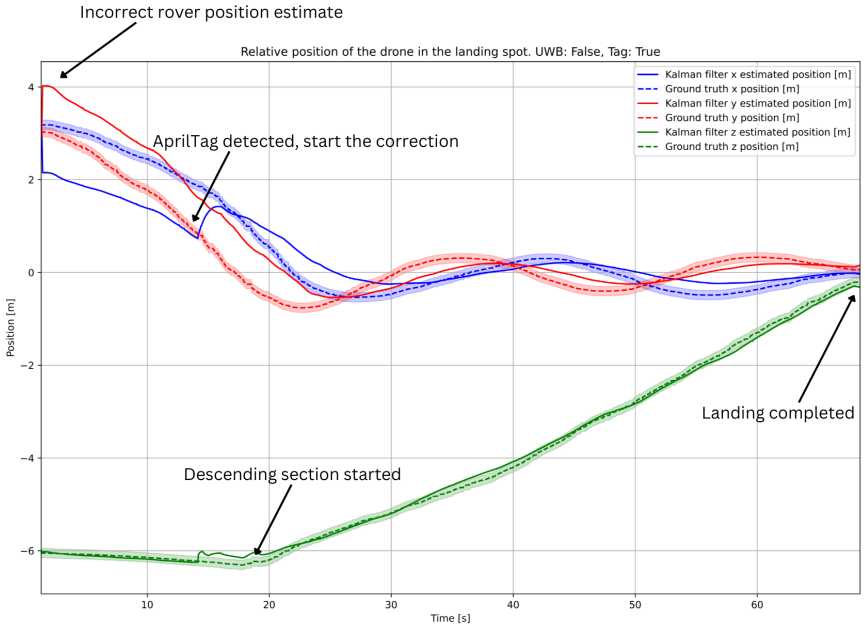


Figure 4.10: Kalman filter estimation output compared to ground-truth data. Case with no UWB data and AprilTag data. A confidence band of 10cm is placed around each ground truth data plot.

Differently to the previous case the estimation data related to the z variable is more precise if compared to the previous case this is due to the fact that the AprilTag data, especially in a simulation environment, is more precise than the ultra-wide band data.

A shortcoming of this approach, though, is that the UAV still has the necessity of hovering in close proximity to the AprilTag in order to identify it with the camera.

When launching the landing action, in this condition with no UWB sensor enabled, the drone has no knowledge of the relative position to the rover if it can't detect the tag. So it must receive a rough indication of that during the landing goal request, as shown in figure 4.9.

The rover position must be expressed in the local world NED reference frame, and the drone will start the landing action by moving toward this new target, after which the AprilTag will be detected and the effective rover position updated with real data coming from the estimation node.

In figure 4.11 it is show the estimation process of the position of the rover, the value coming from the Kalman filter algorithm is compared to a ground-truth measure. The rough position information of the rover comes from the terminal interface shown in 4.9, as it can be seen from the plot in figure 4.11 the one given to the algorithm is a rough indication and does not correspond to the actual value. It will be the estimation algorithm, with the use of the camera and AprilTag, that will correct this erroneous indication and proceed with a correct precision landing.

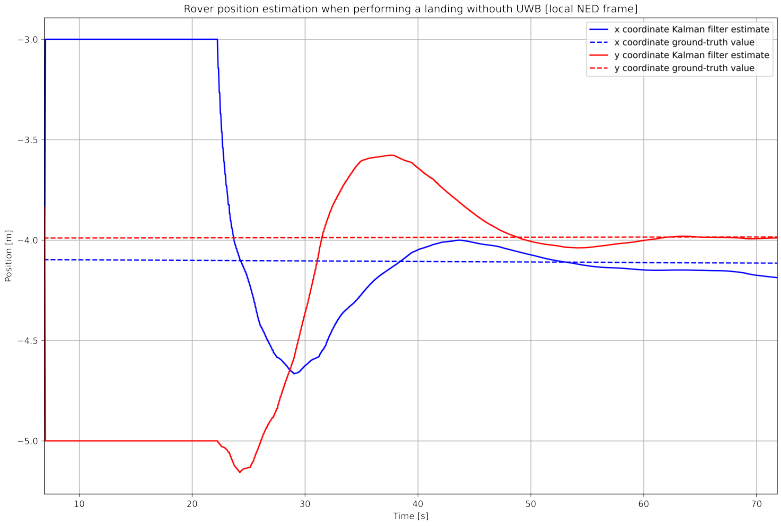


Figure 4.11: Estimated position of the rover during the precision landing request without the usage of the UWB data.

As shown both in figure 4.10 and 4.11 the UAV starts its approach phase with an incorrect information about the position of the rover, and thus its relative position to the landing platform. This is the corrected once the AprilTag is detected by the camera and the data gets fed to the estimation algorithm as shown in figure 4.12. A 10 cm band has been added in order to check the effectiveness of the correction.

4.5.3 Simulation with both UWB and AprilTag sensors

Finally, the best condition is when both the sensor are available this condition is simulated by setting both bool flags to true. This is also the default condition in the action client, if the flags are not specified both will be set to true.

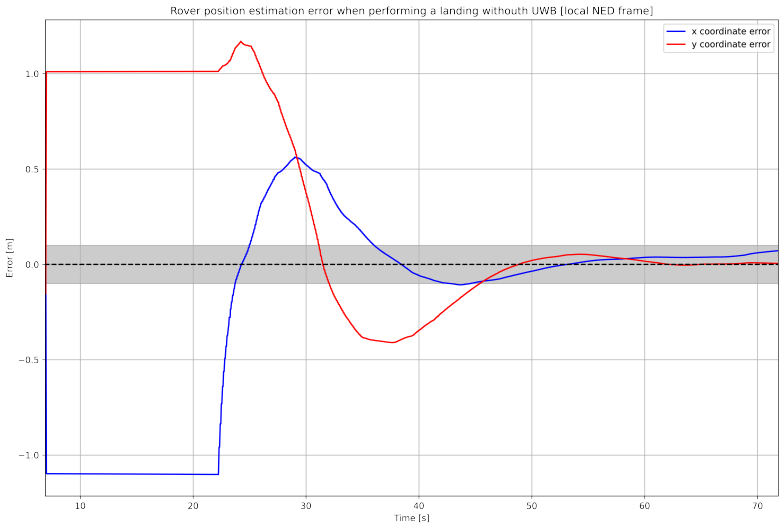


Figure 4.12: Rover position estimate error. Comparing the data coming from the Kalman filter algorithm and the ground-truth data.

The estimation algorithm with both ultrawide-band sensor and the data coming from the camera paired with the AprilTag marker shows the best performance in terms of precision (as shown in figure 4.13).

This had to be expected, moreover does not require the input of the indication of the rover position as this is not necessary due to the presence of the UWB sensor.

Comparison

This is the proposed solution regarding the position estimation, it possesses both of the advantages of the two previous solutions, them being a better estimation of the z coordinate of the drone given by the AprilTag estimation and avoiding the necessity of inputting the estimated position of the drone given by the ultra-wide band sensor. Additionally, this solution doesn't have the drawbacks of the previous cases.

4.5.4 Landing with no UWB and no AprilTag information

This case simulates the worst possible landing condition, in this case it's not possible to perform a precision landing as the estimation algorithm has no possibility of retrieving the relative position between the rover and the UAV.

Similarly to what is shown in figure 4.9 the user is requested to provide the landing action algorithm with the landing position coordinates, also shown in the goal request definition (source code 4.1) as the landing spot entry.

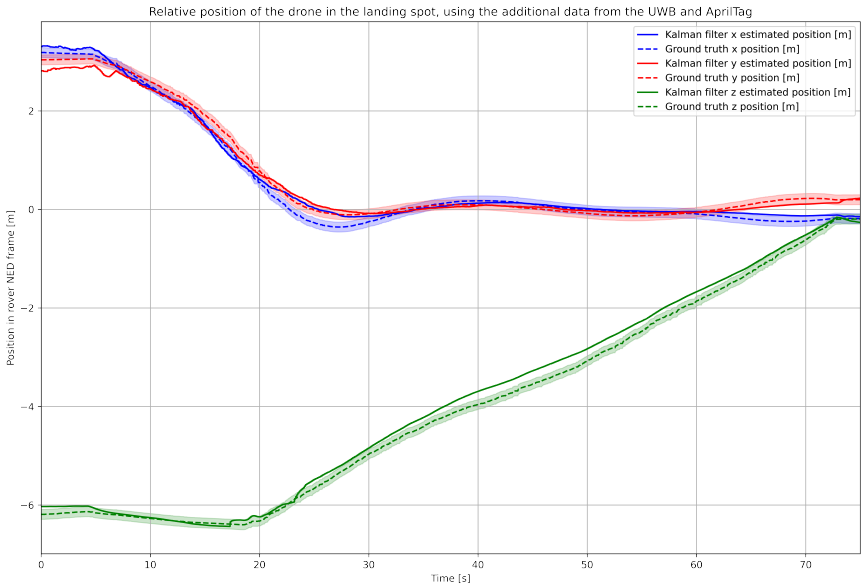
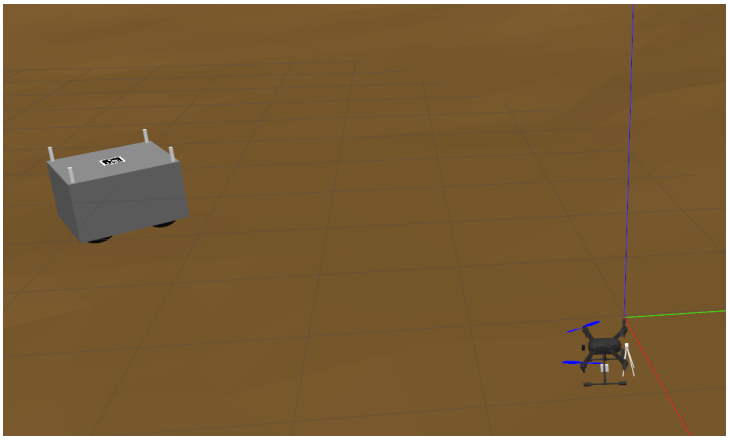
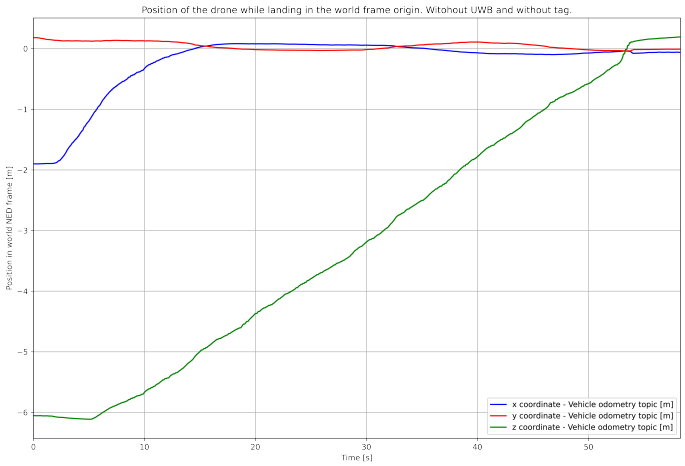


Figure 4.13: Kalman filter estimation output compared to ground-truth data. Case with both UWB data and AprilTag data.

After which the drone will perform a simple landing action, which only reaches the landing spot coordinates, which can be either the rover itself in the case the rover coordinates are known via an external method or another safe landing position.

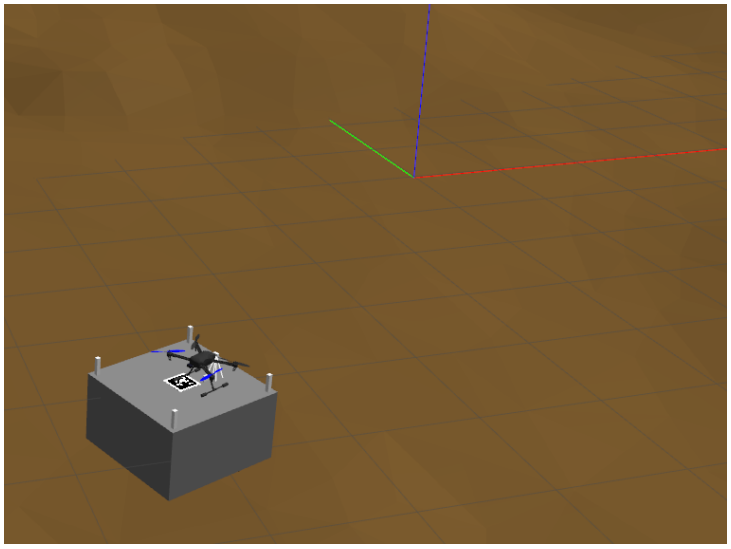


(a) Image showing the drone landed on the origin of the world local frame

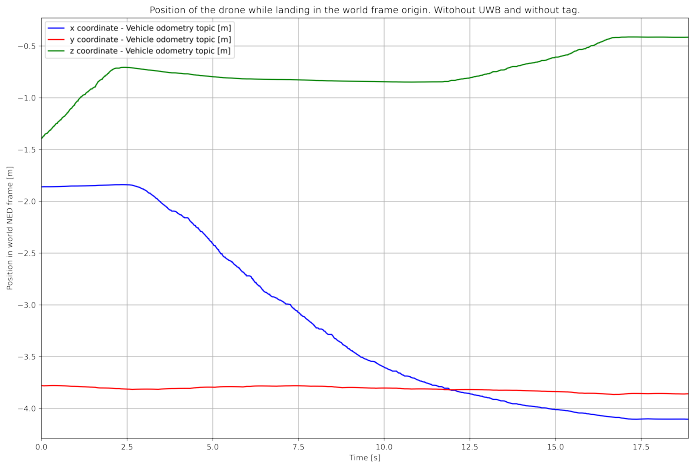


(b) Plot showing the vehicle odometry gotten from the drone PX4 topic.

Figure 4.14: Landing performed in a safe position, in this case corresponding to the origin of the local world frame.



(a) Image showing the drone landed on top of the rover.



(b) Plot showing the vehicle odometry gotten from the drone PX4 topic.

Figure 4.15: Landing performed on top of the rover.

Chapter 5

Conclusions

The thesis work presents a set of algorithms based on ROS 2 and the PX4 autopilot for offboard control of a UAV system.

Such algorithms have been developed in order to obtain an highly flexible and customizable simulation experience. All of the action server used to complete a mission tasks can be tuned by employing the ROS 2 parameters interface. Nonetheless the presented solution already proved to reach ideal results in the simulation.

The algorithms are based on both library and custom functions, for example the grid sweep function provided by the PythonRobotics library relates to the former case, while the PID controller for trajectory tracking regarding the latter case. In both the flexibility of the solution has been prioritized as opposed to more sophisticated solutions which require a better knowledge of the system and higher computation capability.

Moreover, the SITL simulation of such algorithms show how all of them provide satisfactory results, from the data gathering and trajectory tracking point of view.

Regarding the case of the precision landing algorithm the proposed solution is the one pairing the data gathered from the PX4 flight stack and additional data incoming from an ultra-wide band sensor and the camera with an AprilTag marker. This solution showed good estimation performance with a precision in the landing phase which was between the 10 cm confidence band with respect to the ground-truth data gotten from the Gazebo simulator.

Nonetheless, also the algorithms which didn't implement the additional data or just one data between the two were tested in SITL simulation, these also provided with satisfactory results but had shortcomings with respect to the proposed solution.

The simulation shows how the user can select each of the additional data pairings.

5.1 Future work

This thesis delivers a ready-to-use simulation environment, the communication interfaces between the UAV and a base computer (e.g. a rover), and a set of algorithms regarding the navigation, control and estimation. Such environment can be used to test the capabilities of a number of different quadcopters.

Once the drone model is selected it is necessary to test the proposed algorithms on real hardware, thus it will be required to modify some details of the algorithms, as in this case some plugin data is not provided by Gazebo but by some real world sensors.

Once the hardware has been defined it could also be possible to implement more sophisticated and system-specific solutions from both the control (e.g model predictive control) and estimation (e.g. extended kalman filter) section of the flight stack.

Appendix A

Simulation installation tutorial

A.1 Repository organization

A.1.1 drone_bringup

Includes all the `.launch.py` files necessary to launch the simulation environment, estimation and navigation capabilities of the drone.

A.1.2 drone_estimation

In here are contained the Kalman filter node, the apriltag and ultra-wideband estimation nodes, plus the Gazebo groudthruth position node which is mainly used during the testin phase for the validation of the estimation algorithm.

A.1.3 drone_rover_mars

In this folder are contained all the navigation and offboard control algorithms. Most of which are located in the `/actions` subfolder as the interface is achieved through ROS2 actions.

A.2 Installation

A.2.1 Requisites

Firstly, it is necessary to install the various repositories which are:

- PX4-Autopilot
- px4_ros_com
- px4_msgs

A.2.2 PX4 Autopilot

To do so, create a folder `mkdir px4_dev/` and inside it use the command:

```
git clone https://github.com/PX4/PX4-Autopilot.git --recursive
```

Which will clone the repository to the latest available version, since this simulation environment has been developed and tested with a specific version it's necessary to revert the changes with the command shown in the next section.

A.2.3 PX4-ROS2 Bridge

To setup the PX4 agent on the ROS2 side of the communication bridge it's required to clone the other two repositories. It's necessary to create another folder i.e. `px4_ros2_ws` and then another `px4_ros2_ws/src` in which to clone the two repositories.

```
git clone https://github.com/PX4/px4_ros_com.git
```

```
git clone https://github.com/PX4/px4_msgs.git
```

As with the source code of the autopilot the complete installation procedure can be found at this link.

A.2.4 Correct version download

This simulation environment has been developed with a specific version of the PX4 autopilot, in order to avoid problems it is useful to install the same version.

This can be don by running the following command, while in the same folder of the repository. This process has to be repeated for the PX4 autopilot source code and the component of the PX4-ROS2 Bridge.

```
git checkout <commit hash>
```

- PX4-Autopilot
  - Version: f15eefc
- px4_ros_com
  - Version: 3b577d6
- px4_msgs
  - Version: cb455c2

A.2.5 Complete the installation

Then in order to complete the installation and to install all of the dependencies of PX4 it's necessary to run the following command:

```
./PX4-Autopilot/Tools/setup/ubuntu.sh
```

The complete installation tutorial of the autopilot can be found at this link.
It is also necessary to install the FastRTPS/DDS middleware implementation. Installation tutorial link.

Troubleshooting

It may be necessary to clean up the installation after reverting to the previous version, after running the `ubuntu.sh` bash, it may be necessary to run this command:

```
cd px4_dev/PX4-Autopilot
make distclean
```

This will clear all the submodules, it should not be ran after modifying the firmware e.g. updating the iris model and the `empy.world` file.

Testing the installation

To test the installation it is possible to run the following command:

```
make px4_sitl gazebo
```

This will build and launch the default PX4 simulation with the iris model in Gazebo. If the installation is successful the model should be visible.

A.2.6 Modifications in the PX4 folder

All the modification files can be found in the folder `drone_rover_mars/extra`

Multiple run script

While in this folder `/home/<user>/px4_dev/PX4-Autopilot/Tools/`:

- Paste new `gazebo_sitl_multiple_run.sh`

Model files update

It is possible to copy paste the folder contained in `drone_rover_mars/models` and `drone_rover_mars/worlds` in the folders that can be found at `/home/<user>/px4_dev/PX4-Autopilot/Tools/s`

- `iris.jinja.sdf` and `model.config` in `../sitl_gazebo/models/iris`
- Create a new `rover_uwb` folder (in `../sitl_gazebo/models/`)

- `rover_uwb.sdf` and `model.config` in `rover_uwb` folder
- Create a new `curiosity_path` folder (in `../sitl_gazebo/models/`)
- Move all the files from `..drone_rover_mars/extra/models/curiosity_path` into the new folder
- `empty.world` in `../sitl_gazebo/worlds`

A.2.7 Updating the launch files path

In the folder `../drone_bringup/launch` the file `drone_rover_clients_agents.launch.py` has to be updated on line 26.

From:

'/home/riccardo/px4_dev_3/PX4-Autopilot/Tools/gazebo_sitl_multiple_run.sh'

To:

'/home/<user-folder>/px4_dev/PX4-Autopilot/Tools/gazebo_sitl_multiple_run.sh'

Moreover, the path used to save the position data and image data during the testing phase has to be updated in the files:

- `/drone_rover_mars/actions/getimages_action_server.py`
- `/drone_rover_mars/actions/gridsweep_action_client.py`
- `/drone_rover_mars/actions/gridsweep_action_server.py`

If using Visual Studio Code as Editor, it is possible to use the `Replace in Files` (or `CTRL+Shift+F`) function to update all the paths at once (VsCode function link).

From:

`riccardo/thesis_ws`

To:

`<user-folder>/<workspace>`

For example: `<user-folder>/px4_ros2_ws`.

A.2.8 Building the ROS 2 Workspace

This is the final step of the installation, the ROS 2 workspace should have this hierarchy:

```
src
├── data
├── drone_bringup
├── drone_estimation
├── drone_rover_mars
├── px4_msgs
├── px4_ros_com
├── README.md
├── ros1_bridge
├── ros2_px4_functions
├── ros2_px4_gazebo
└── ros2_px4_interfaces
```

While the `px4_ros2_ws` folder should have only the `src` folder.
At this point the workspace can be built by running the following command:

```
cd ~/px4_ros2_ws
colcon build --symlink-install
```

The full guide to build the workspace can be found here.
Before running the simulation it is necessary to source the workspace:

```
. install/setup.bash
```

A.2.9 Simulation aliases

In order to have an easier simulation startup, some bash aliases have been written. **These are not strictly necessary** in order to run the simulation but they do provide an easier usage. In order to do so it's necessary to add the following lines to the `.bashrc` file, by running the following command, from the home directory (run the `cd ~` if necessary).

```
gedit .bashrc
```

Once the file has been opened:

```
alias simulation='ros2 launch drone_bringup drone_rover_clients_agents.launch.py'
alias estimation='ros2 launch drone_bringup drone_rover_positioning.launch.py'
alias drone_control='ros2 launch drone_bringup drone_control.launch.py'
```

This will provide an alias related to the launch files contained in the bringup folder.

```
alias takeoff='ros2 run drone_rover_mars takeoff_action_client'
alias abort='ros2 service call /drone/abort_mission ros2_px4_interfaces/srv/AbortMission'
alias target='ros2 run drone_rover_mars gototarget_action_client'
alias getimages='ros2 run drone_rover_mars getimages_action_client'
alias land_on_spot='ros2 run drone_rover_mars landonspot_action_client'
alias gridsweep='ros2 run drone_rover_mars grid_sweep_action_client'
```

The abort mission service may no longer be necessary. It is reccomended to abort an action via the `CTRL+C` command as the code organization has changed during the developement.
It is still possible to run the abort mission service, it may be necessary to add this to a launch file (i.e. the `drone_control`) as its implementation was not strictly necessary.
In order to get the plots for this Master thesis project the PlotJuggler software has been employed. If it's installed another useful alias is this:

```
alias plotjuggler=' ros2 run plotjuggler plotjuggler -n'
```

A.3 Running the simulation

Once all the installation steps have been succesfully done it is possible to run the simulation.

During this part of the tutorial all the commands will be treated as if the aliases have been added, if this is not the case refer to the previous section to look up the complete commands.

If the `.bashrc` aliases have been added the simulation can be started by launching a terminal (a split screen one might be helpful, for example Terminator and then sending the following commands:

1. In one window of the terminal launch the `simulation` command, this will launch the Gazebo simulation environment.
2. In another terminal window, launch the estimation command. This command is required for any algorithms that need the relative position of the simulated UAV to the UGV position. For example, if you are testing only the go_to_target and takeoff algorithms, you can skip the estimation command, as it is not necessary and may consume a lot of computational resources.
3. Similarly to the previous case, in a third terminal window the `drone_control` command is issued. In any case, if the user wants to test any of the navigational capabilities or the estimation side of the simulation is necessary to launch these nodes.
4. Leave a fourth terminal window that can be used when issuing the client side of the operations, for example sending a request to the action servers.

An additional window may be helpful in order to run complementary software such as the already cited PlotJuggler or QGroundControl.

This should appear when launching the simulation command:

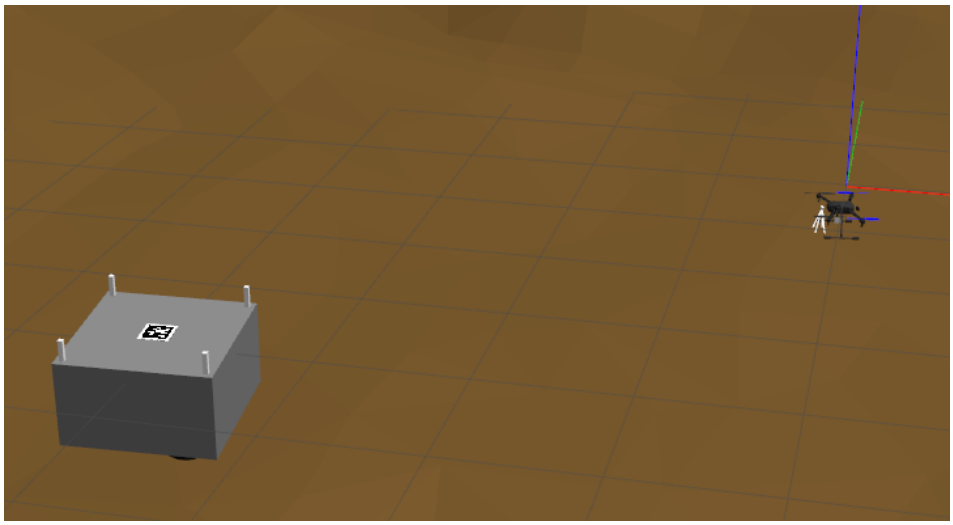


Figure A.1: Simulation environment in the Gazebo software.

A.3.1 Simulation commands

This section has been added assuming the aliases a have been set up in the `bashrc` file, it is possible to send the commands by following the following table.

In case the standard notation of ROS2 is followed, do not follow these instructions.

The standard command is the following:

```
ros2 action send_goal <action_name> <action_type> <values>
```

The options regarding the reference frame are:

- `world`, the default options if left empty
- `rover`, the NED reference frame centered in the position of the rover. To use this reference frame, launch the `estimation` file.

Command	Input 1	Input 2	Input 3	Input 4
`takeoff`	`<Takeoff altitude [m]>`			
`target`	`<string: Reference frame>`	`<bool: takeoff flag>`		
`getimages`	`<Orbit center coordinates>`	`<Orbit radius>`	`<Number of images>`	`<string: Reference frame>`
`landonspot`	`<bool: use uwb flag>`	`<bool: use apriltag flag>`		

Figure A.2: Table of the commands.

Takeoff + Go to target example

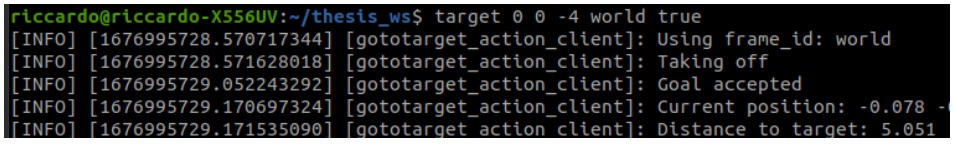


Figure A.3: Go to target terminal interface, with the aliases enabled.

All of the coordinate commands are sent following the NED frame convention (z axis in the down direction), in this case it's possible to see that the takeoff flag is set to true and so the drone will takeoff at the default altitude of 2 meters and the go to the target coordinates. In this case the target coordinates are expressed in the `world` reference frame.

In case it's necessary to specify the altitude of the takeoff it's necessary to use the `takeoff <altitude>` command.

Getimages example

```
riccardo@riccardo-X556UV:~/thesis_ws$ getimages -2.0 0 -4 2 4
[INFO] [1676995851.451178109] [getimages_action_client]: Using default frame_id (world)
[INFO] [1676995851.452150170] [getimages_action_client]: Sending goal request...
[INFO] [1676995853.714967002] [getimages_action_client]: Goal accepted
```

Figure A.4: Getimages terminal interface, with the aliases enabled and using the default (world) reference frame.

In this case the first three inputs are correspondent to the coordinates of the orbit, the following inputs is the radius and finally the number of image that the drone will take.

Since there is no string after the number of images, the reference frame is set to `world` by default.

```
riccardo@riccardo-X556UV:~/thesis_ws$ getimages -2.0 0 -4 2 4 rover
[INFO] [1676996506.094509218] [getimages_action_client]: Using frame_id: rover
[INFO] [1676996506.095548818] [getimages_action_client]: Sending goal request...
[INFO] [1676996506.102871894] [getimages_action_client]: Goal accepted
```

Figure A.5: Getimages terminal interface, with the aliases enabled and using the rover reference frame.

If instead the drone is required to take the images with a center expressed in the `rover` reference frame, the command above is used.

Area Coverage action

Organizing the area coverage task through the Python input function has been decided due to its complexity from the user's point of view.

The inputs the required during this action are:

- Specify the number of points required to define a closed perimeter for the mapped area.
- 2D coordinates of the points
- Mapping altitude of the drone. As with the previous actions, the altitude should be provided following the NED convention.
- Resolution of the grid sweep.

```
riccardo@riccardo-X556UV:~/thesis_ws$ gridsweep rover
Enter number of grid points: 5
Grid point n. 1 :
Enter x coordinate: 0
Enter y coordinate: 0
Grid point n. 2 :
Enter x coordinate: 5
Enter y coordinate: 0
Grid point n. 3 :
Enter x coordinate: 5
Enter y coordinate: 5
Grid point n. 4 :
Enter x coordinate: 0
Enter y coordinate: 5
Grid point n. 5 :
Enter x coordinate: 0
Enter y coordinate: 0
Enter altitude: -5
Enter resolution: 0.5
[INFO] [1676998891.005675914] [grid_sweep_action_client]: Waiting for action server...
```

A.3.2 Actions bagfiles

The folder data/bagfiles contains the simulation bagfiles that record the messages and topics exchanged during the simulation of the drone simulation project. The bagfiles can be used to replay the simulation data and analyze the performance of the estimation algorithm.

Bibliography

[1] E. Ebeid, M. Skriver, K. H. Terkildsen, K. Jensen, and U. P. Schultz, "A survey of open-source UAV flight controllers and flight simulators," *Microprocessors and Microsystems*, vol. 61, pp. 11–20, 2018.

[2] L. Meier, D. Honegger, and M. Pollefeys, "Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms," in *2015 IEEE international conference on robotics and automation (ICRA)*, pp. 6235–6240, IEEE, 2015.

[3] "JPL - Ingenuity Mars Helicopter." URL: https://www.jpl.nasa.gov/missions/ingenuity.

[4] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, "ROS: an open-source Robot Operating System," 2009.

[5] "Tutorials — ROS 2 Documentation: Foxy documentation." URL: https://docs.ros.org/en/foxy/Tutorials.html.

[6] "rqt_graph - ROS Wiki." URL: http://wiki.ros.org/rqt_graph.

[7] "ROS1 vs ROS2, Practical Overview For ROS Developers - The Robotics Back-End." URL: https://roboticsbackend.com/ros1-vs-ros2-practical-overview/#Why_ROS2_and_not_keep_ROS1.

[8] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[9] "Introduction - Programming Multiple Robots with ROS 2." URL: https://osrf.github.io/ros2multirobotbook/.

[10] "ROS 2 Design." URL: http://design.ros2.org/.

[11] "Changes between ROS 1 and ROS 2." URL: http://design.ros2.org/articles/changes.html.

[12] "Topics vs Services vs Actions — ROS 2 Documentation: Foxy documentation." URL: https://docs.ros.org/en/foxy/How-To-Guides/Topics-Services-Actions.html.

[13] "PX4 User Guide." URL: https://docs.px4.io/main/en/.

[14] "PX4/PX4-Autopilot: PX4 Autopilot Software." URL: https://github.com/PX4/PX4-Autopilot.

[15] "MAVLink Messaging | PX4 User Guide." URL: https://docs.px4.io/main/en/middleware/mavlink.html.

[16] "ROS (Robot Operating System) | PX4 User Guide." URL: https://docs.px4.io/main/en/ros/.

[17] "uORB Messaging | PX4 User Guide." URL: https://docs.px4.io/main/en/middleware/uorb.html.

[18] "RTPS/DDS Interface: PX4-Fast RTPS(DDS) Bridge | PX4 User Guide." URL: https://docs.px4.io/v1.12/en/middleware/micrortps.html.

[19] "ROS 2 User Guide (PX4-ROS 2 Bridge) | PX4 User Guide." URL: https://docs.px4.io/main/en/ros/ros2_comm.html.

[20] "GitHub - PX4/px4_ros_com: ROS2/ROS interface with PX4 through a Fast-RTPS bridge." URL: https://github.com/PX4/px4_ros_com.

[21] "GitHub - PX4/px4_msgs: ROS/ROS2 messages that match the uORB messages counterparts on the PX4 Firmware." URL: https://github.com/PX4/px4_msgs.

[22] "Offboard Mode | PX4 User Guide." URL: https://docs.px4.io/v1.12/en/flight_modes/offboard.html.

[23] "PX4 Architectural Overview | PX4 User Guide." URL: http://docs.px4.io/main/en/concept/architecture.html.

[24] "Controller Diagrams | PX4 User Guide." URL: https://docs.px4.io/v1.12/en/flight_stack/controller_diagrams.html.

[25] "Simulation | PX4 User Guide." URL: https://docs.px4.io/main/en/simulation/.

[26] "Gazebo Simulation | PX4 User Guide." URL: https://docs.px4.io/v1.12/en/simulation/gazebo.html#gazebo-simulation.

[27] "Using Vision or Motion Capture Systems for Position Estimation | PX4 User Guide." URL: https://docs.px4.io/main/en/ros/external_position_estimation.html.

[28] "Understanding actions — ROS 2 Documentation: Foxy documentation." URL: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html.

[29] "PID controller - Wikipedia." URL: https://www.en.wikipedia.org/wiki/PID_controller.

[30] A. Sakai, D. Ingram, J. Dinius, K. Chawla, A. Raffin, and A. Paques, "Pythonrobotics: a python code collection of robotics algorithms," *arXiv preprint arXiv:1808.10703*, 2018.

[31] "AtsushiSakai/PythonRobotics: Python sample codes for robotics algorithms.." URL: https://github.com/AtsushiSakai/PythonRobotics#citing.

[32] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[33] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[34] G. A. Pelissero, "Autonomous Precision Landing for UAVs on a Mars-like environment in ROS/Gazebo - Webthesis," Master's thesis, 2022.

[35] G. Scarati, "UAV precise ATOL techniques using UWB technology - Webthesis," Master's thesis, 2021.

[36] "Kalman filter - Wikipedia." URL: https://en.wikipedia.org/wiki/Kalman_filter?oldformat=true.

[37] "FilterPy — FilterPy 1.4.4 documentation." URL: https://filterpy.readthedocs.io/en/latest/.