POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Aerospaziale, Aeromeccanica e Sistemi

Collegio di Ingegneria Meccanica, Aerospaziale, dell'Autoveicolo e della Produzione

Design and simulation of a machine learning-based approach for an autonomous UAV for use in Agriculture 4.0 applications

Relatori:
Prof. Giorgio Guglieri
Ing. Nicoletta Bloise
Dott. Stefano Primatesta

Laureando:
Matteo Tonin

Anno accademico 2022/2023

"When life gives you lemons, don't make lemonade. Make life take the lemons back! What am I supposed to do with these? I'm gonna get my engineers to invent a combustible lemon that burns your house down!"

Cave Johnson, *Portal 2*

Abstract

The aim of this thesis is to design, build and simulate an algorithm which provides guidance and control for autonomous UAVs in Agriculture 4.0 within the PRIN project “New technical and operative solutions for the use of drones in Agriculture 4.0”.

The simulation scenario includes several targets which can represent plants, vegetables, or other vegetation where treatments deployed by the UAV is needed. The main objective of the developed algorithm is to guide the UAV, identifying and reaching all possible targets autonomously using information derived by depth cameras and other sensors. Another objective of the project is the application of the final algorithm with limited GPS or noisy signal conditions. This, in fact, is one of the most critical working scenarios and traditional guidance based on mission definition through waypoints could fail to achieve an acceptable level of positioning precision depending on the UAV’s mission profile.
The simulation environment is developed using ROS (Robot Operating System) framework and the Gazebo simulator. Simulation includes the UAV and a physical environment which should represent a working scenario as close as possible to the real one.
During the project will be developed an allocation algorithm, which objective is to map target’s positions in the local frame of reference, and a control method for three axes. Also, two Machine Learning (ML) applications are part of main challenge of the project. A U-Net, deep neural network, for target recognition will be implemented and a Reinforcement Learning (RL) algorithm, that describes the policy with which targets to reach sequentially are chosen, will be tested.

Final results show the algorithm works in the simulated scenario, with the UAV being able to allocate each target in the correct position in local frame and subsequently reach them one after another without passing over the ones already reached. This is the case if a policy of following always the nearest target is passed to the UAV. The Reinforcement Learning method developed presents good but not optimal results: the UAV is able to complete the mission without hovering over the same target twice, but the algorithm will not learn to choose the best path, i.e. the one which requires less time. Some further testing in this case is required or heavy modification of the algorithm are needed.

Table of contents

1 Introduction

The aim of this chapter is to provide the reader with an introduction to the topic of this thesis, the main scope of the project and the software components used. The general pipeline flowchart of the project, which explains the implemented methodology, is presented, and explained.

1.1 Project description

1.1.1 UAV precision agriculture

Traditionally, spraying techniques are used in the agricultural industry to apply pesticides, such as fungicides, herbicides, and insecticides, to plantations. Several parameters have an influence on effectiveness, cost and environmental hazard of pesticides spraying, such as crop canopy, crop height, crop volume, etc. With conventional sprayers, it's difficult to adapt the spraying technique to optimize the task for the above-mentioned parameters, resulting in spray loss in the form of spray drift and off-target deposition. These problems results in environmental pollution, human health hazards and the waste of costly products. The use of robotics and automation to carry out spraying falls under the term precision agriculture, which aim is to reduce pesticide residues, save costs, make crop protection more compatible and reduce hazards to humans and the environment.
In recent years, aerial spraying with UAVs has gained more interest as an effective alternative to achieve precision spraying. UAV spraying has several advantages or conventional methods:

- Provides spraying methods for tall crops, such as maize and cotton, and for pond crops, such as rice.
- Reduces spray loss in the form of spray drift by accurately positioning of the UAV using target recognition techniques.

In terms of application, tank refilling can be a real issue, but in recent years ultra-low volume sprayers have been introduced, reducing the need for frequent tank refills (Fiaz Ahmad, 2021).
Positional information is necessary for UAV spraying systems because of the need for accurate positioning. For some agricultural UAV based applications, such as yield monitoring, accuracies of less than 1 m can be achieved with differential GPS (DGPS). However, for spraying and other applications, accuracy below centimetres is required, thus making necessary the use of technologies such as real-time kinematic GPS (RTK-GPS) to meet this demanding requirement (Pérez-Ruiz, 2012).

RTK GPS works well and provides the robot, in this case the UAV, with a highly accurate position signal in real time. However, it has some drawbacks:

- Setup requires a base station whose exact coordinates must be known.
- The need for a base station with a clear view of the UAV.
- GPS signal outage, depending on environmental conditions, greatly affects its accuracy (In-Su Lee, 2005).
- High cost.

The main objective of this thesis is to extend on precision agriculture techniques by designing an algorithm to be implemented for a self-driving UAV for spraying operations with limited use of GPS information. During the project, machine learning techniques will be implemented, simulated, and tested to evaluate their usefulness in the proposed scenario.

1.1.2 Project software components

For this project, the mission objective is to autonomously reach five targets, randomly distributed or positioned with a criterion, in a working area. This objective can well represent what happens in a real scenario, where the UAV has to apply products only on some target plants and not on the whole plantation or follow a series of waypoints.
The first problem to be solved is how to develop the algorithm. Since a real UAV would be costly and carry the risk of damaging the system and the environment, the simulation option is chosen. Simulating the UAV and the environment allows to try different solutions without the risk of damaging the system, it's easier and always the first step for rapid prototyping of control systems and guidance algorithms. The main issue is the need to build a realistic simulator, that can represent both the behaviour of the UAV and its interaction with an environment.
The implemented solution is to simulate the UAV in the Gazebo simulator (Gazebo homepage, s.d.). Gazebo is a powerful simulator toolbox complete with development libraires, with the ability to generate several robots with a variety of sensors, such as cameras, GPS and IMUs. In Gazebo is also possible to generate an environment with 3D models, thus providing the needed UAV-environment interaction.
The Iris UAV, powered by PX4 (px4 user guide homepage, s.d.) autopilot, is used in the simulated Gazebo environment.
PX4 was chosen as the target firmware for the simulated UAV for the following reasons:

- Validated in real-world scenarios: commercial PX4-based UAVs are on use worldwide, with thousands of flight hours accumulated. This allows the development of a project that not only works

in a simulated environment but could later be implemented on a real machine, with minor changes to the source code.

- Open-Source Community: PX4 resources, documentation, code and troubleshooting methods are easily accessible to the user by consulting the official documentation or by referring to other users' suggestions on the official platforms. This allows for rapid debugging of the code and more secure project development.

As for the UAV the Iris model is the quadrotor of choice used for this project development. The reasons for this choice are listed below:

- Quadrotor design: In the UAV market, the quadrotor design is considered the simplest design for hovering UAVs. The clean design of Iris allows for faster simulation in Gazebo, as there are no fixed complex payloads pre-installed, and the collision/visual model is the simplest of any UAV available.
- PX4 compatibility: Iris is one of the quadrotors available in the PX4 package that is compatible with the autopilot firmware out of the box, without the need to create additional ROS plugins, which would have added an extra layer of complexity to the project.
- Several payloads are available: The Iris model comes with a GPS and IMU sensor already simulated in the base model, both of which are required for the project. Also, there are several Iris models already in the PX4 ROS package that come with a variety of cameras already configured. The availability of these additional payloads for the model is critical.

The ROS (Robot Operative System) middleware (ROS homepage, s.d.) is used to manage the streams of messages output by the PX4 autopilot or generated by multiple sensor sources. ROS is an open-source software composed of packages that define interfaces, components, and tools so that sensors, control systems and actuators can be easily built, connected, and simulated in Gazebo.
The MAVLink protocol (MAVLink Developer Guide, s.d.) is used for messages coming from and going to the PX4 FCU, as several ROS packages are available to read, write and generally manage them, ensuring great compatibility with the rest of the project.

1.1.3 Algorithm architecture

The main objective of the guidance algorithm is to provide the UAV's FCU with values in terms of velocities, both angular and linear, to follow in order to achieve the mission objective. The mission objective, as previously stated, is to reach all the targets spawned in the physical environment of the simulation.
The proposed solution method is closely related to how biological species perceive the environment; if the objective is to reach some targets scattered in an environment, we would probably try to memorise their positions and then proceed to reach them one by one. For humans and other species, it is quite easy to perform

a similar task, even if some of the targets are not visible at any given time, because we are able to memorise their approximate positions after seeing them. The same basic idea is implemented in the guidance algorithm, which has to encode a part aimed at seeing the target and giving commands to the UAV to reach it based on visual data, and a part aimed at memorising the target positions in local coordinates after seeing them during the simulation.
Therefore, the UAV must be able to recognise the targets through a machine learning based method known as image segmentation, so two depth cameras, able to give both the image and the pointcloud, are simulated as the payload of the UAV, better explained in chapter 2.2.

Figure 1.1 shows the complete guidance flowchart implemented in this thesis. In green, available information that can be obtained from the UAV's cameras and from the UAV's FCU is highlighted. While in blue components that provide guidance to the control loops are highlighted.
The upper part of the diagram shows the signals coming from the frontal depth camera image and point cloud up to the generation of the target local position message. The idea is to process the data stream from the frontal camera in real time using a U-net, explained in detail in Chapter 3, to obtain image masks, representations of the target's body in a single channel image. The masks are then processed using a contouring algorithm to obtain target centroids, the process is explained in Chapter 4.1. The next step is to store the target positions in the local reference frame, for reference systems see chapter 2.1.4. To achieve this, the target centroid in the camera reference frame, the front camera point cloud and the UAV position are processed by the mapping algorithm explained in chapter 4.2.
Once a mapping of the targets in the local coordinate frame has been obtained, the logic for selecting the target to be reached must be implemented. In this project, two selection methods have been implemented, one deterministic and one based on RL (Reinforcement Learning), both of which sequentially select one target after another without repeating the same one. The selection methods are better explained in chapters 6.1, 6.2.
Now a single target is selected, and its local coordinates are available to be used as a command signal. A PID control method for the x and y axes is implemented so that the error signal generated by the difference between the target (x, y) position and the UAV (x, y) position can be used to produce commands in terms of linear velocity and angular position.

It is also necessary to build an altitude controller so that the UAV can hover at a constant relative altitude even in the presence of uneven terrain, which could be a real case scenario. To achieve this goal, another PID control is implemented, z_PID, which takes as input an estimate of the UAV's relative altitude from the downward camera pointcloud and a reference value that represents the guidance. Using the error generated by these two messages, z control commands can be given to the FCU in the form of speed.
The x_y_PID and z_PID logic is explained in more detail in chapter 5.

Note that messages coming from GPS and IMUs are utilized in the allocation algorithm, however the idea is to provide guidance without or with limited use of the GPS message. In real operations, depending on the

environmental conditions, the GPS signal may be noisy, which also affects the position of the UAV, since the Kalman filtering takes multiple inputs (Vasko Sazdovski, 2005):

$$UAV\ state = Kalman(GPS, IMU_1, \dots, IMU_4)$$

This is an issue in the proposed method as the memorisation of target positions in the local frame implies the need for the UAV to know its position through the FCU. Since there is no other choice, the project development follows this route anyway, but tests in the presence of noise are obligatory to assert the robustness of the algorithm.

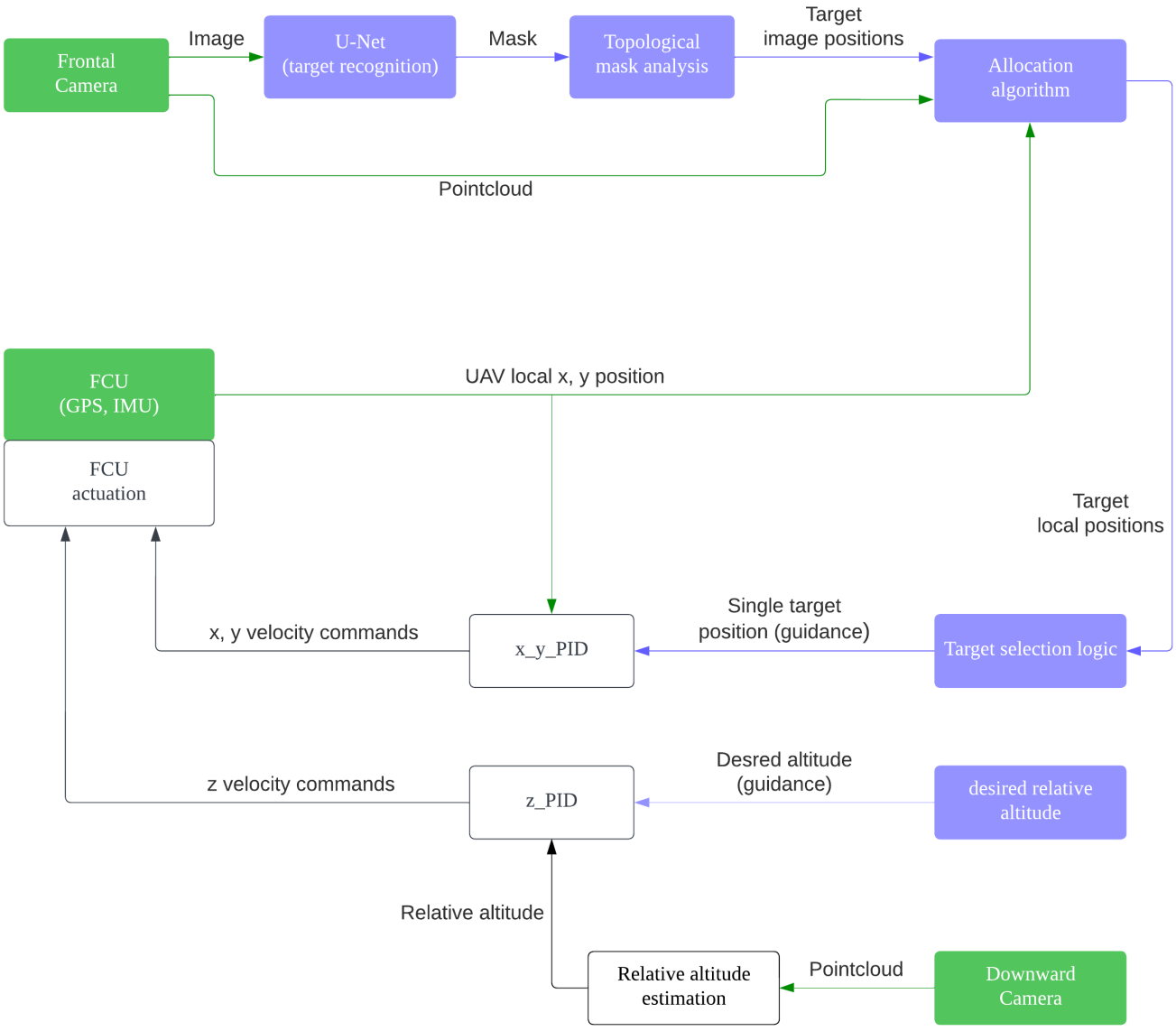


Figure 1.1: Complete project flowchart.

2 Simulation set up

In this chapter, the aim is to create the ROS workspace needed for the project, successfully launch the simulation generated in Gazebo and then spawning the Iris UAV with sensors such as IMUs, GPS and appropriate payloads such as cameras.
After having correctly generated Iris quadrotor in the world, code written in ROS logic is presented which aim is to allow the UAV to move in the simulated physical environment taking in input commands directly from the user's keyboard.

These actions are considered the basic for the simulation to work properly and the starting point to achieve results presented in the successive chapters.

2.1 ROS workspace initialization

As mentioned in the previous introductory chapters the project runs entirely in ROS, so the first step to complete is setting up a ROS workspace and installing necessary packages.
Then a list of the simulation components and configuration used is given to then start the simulation properly.

2.1.1 Workspace configuration

First of all, it is needed to source the *setup.bash* file so that environmental variable for both ROS and Gazebo are generated, then create the directory *catkin_ws/src*. *catkin_ws* is the working folder for the project, whereas *src* houses all user installed or created packages. Finally, through the command *catkin build* the ROS workspace is created.
With the following commands, in figure 2.1, in a Linux terminal is possible to generate the workspace as explained.

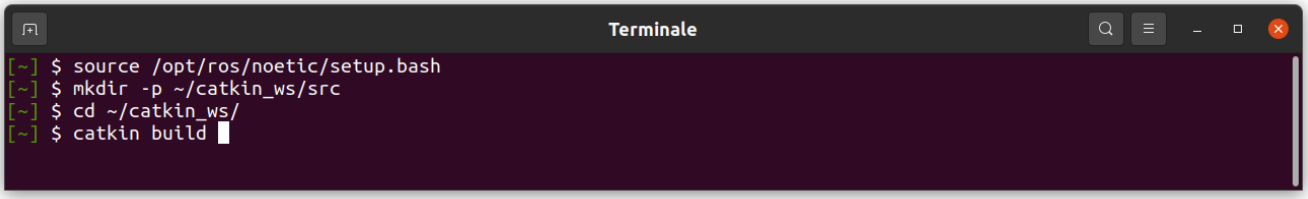


Figure 2.1: commands for ROS workspace initialization

The definition of ROS/Gazebo environmental variables through the command *source …/setup.bash* should be given each time a new terminal is opened, thus the command can be added directly at the end of Linux *~/.bashrc* file to speed up future ROS commands by terminal.

Several ROS packages are installed and built in the *catkin_ws* workspace just created, so that the final tree project folder should look as in figure 2.2.

- *Firmware (px4)*: this ROS package contains the firmware for the PX4 autopilot, various airframe models with relative plugins which simulates their dynamics and sensors models and plugins. Every ROS plugin is accessible by the user making the firmware completely customizable.
- *mavros* (mavros package summary, s.d.): this package provides a complete list of communication services and topics in MAVLink protocol (MAVLink Developer Guide, s.d.), with various quadrotor autopilots, including also PX4, the one used for this project. Without *mavros* topics and services, it will not be possible to communicate with the autopilot using python modules.
- *mavlink* (mavlink package summary, s.d.): this package is mandatory for the functioning of the *mavros* package, it is a communication library for various autopilots.

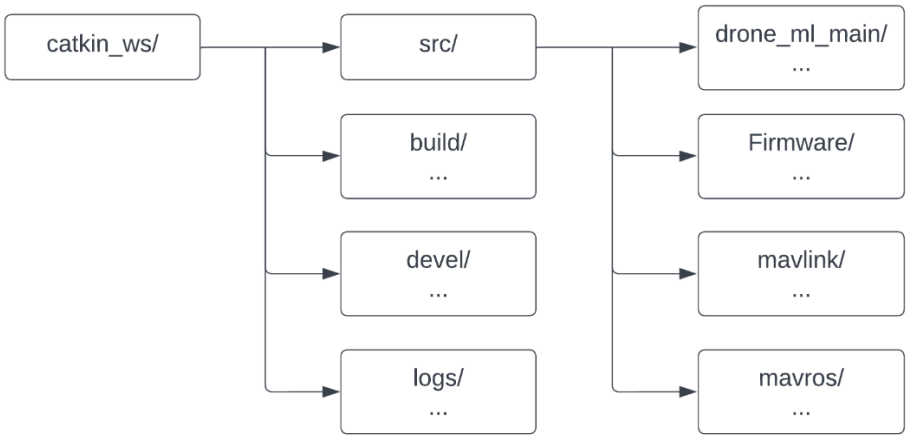


Figure 2.2: catkin_ws main ROS workspace subfolders tree up until two levels from catkin_ws main folder

Another ROS package, called *drone_ml_main*, is created to accommodate all the custom scripts and files, which subfolders tree is depicted in figure 2.3.
Dependencies of the package must be defined in files *CMakeList.txt* and *package.xml* files. In this case dependencies that have to be added for the project are:

- *gazebo_ros:* Gazebo services and topics in the ROS environment.
- *roscpp, rospy*: ROS interfaces with C++ and Python programming languages.
- *message_generation*: mandatory to define custom messages.

In *scripts*/*commander.py* are allocated the majority of the basic methods which allow the UAV to execute flying tasks like taking inputs from keyboard or follow and find targets and also some more basic tasks like switch between flight modes. Each script is briefly explained in the next chapters. The last command to finalize the building of the project is *catkin build* that allows to build the whole project and highlight possible packages installation errors. If the installation is done correctly, the terminal after the build command should look as in the figure 2.4.

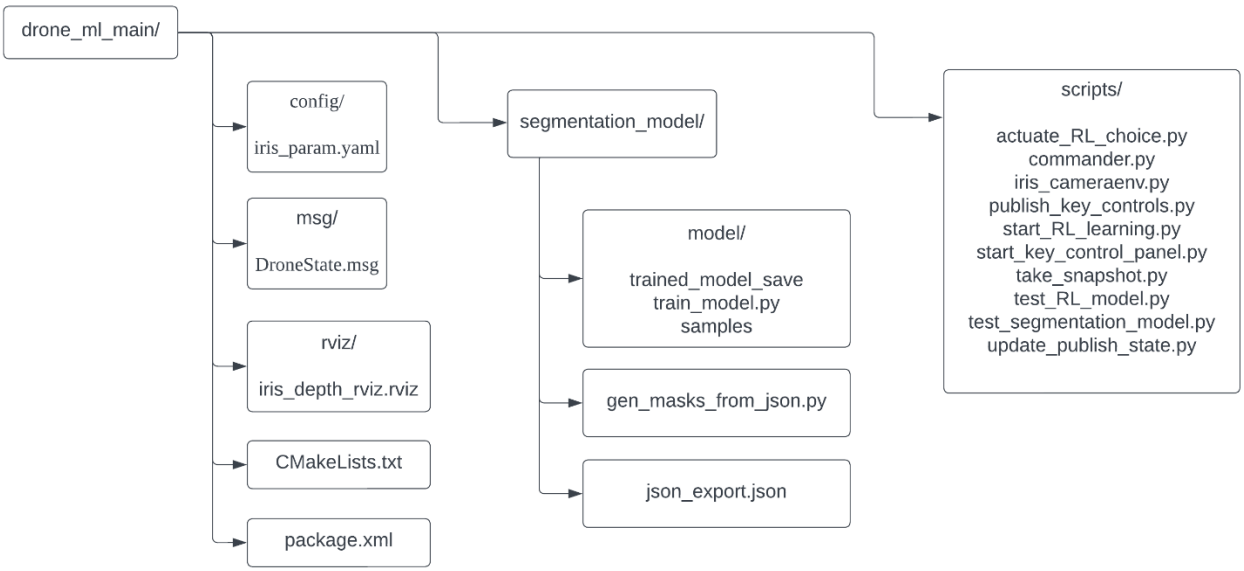


Figure 2.3: drone_ml_main complete subfolders tree and files

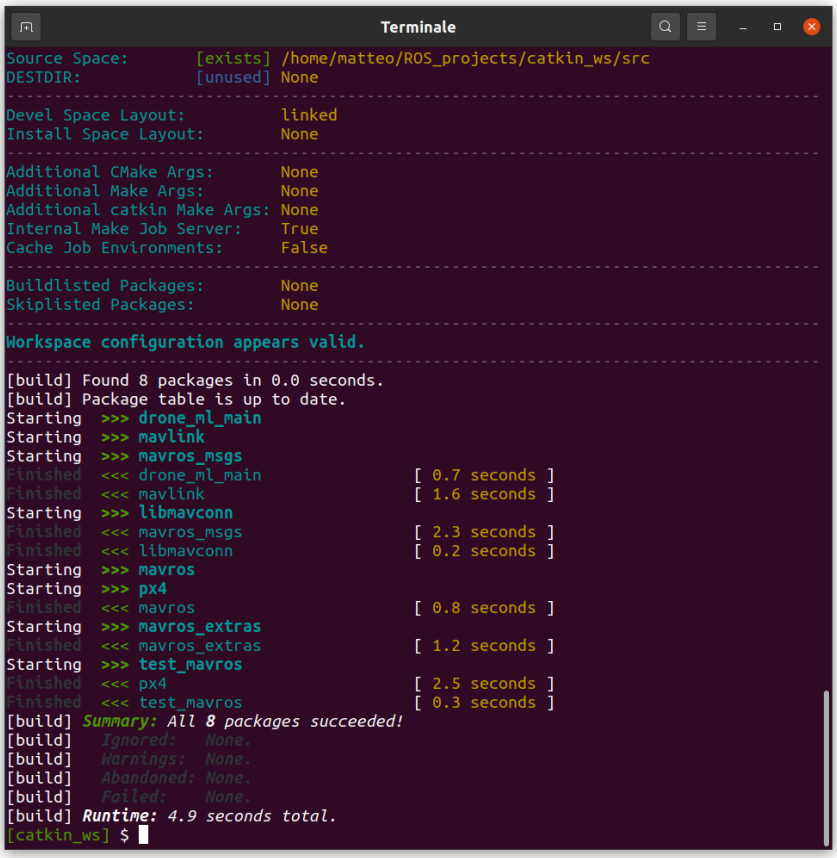


Figure 2.4: terminal screen after applying catkin build command on the whole ROS workspace.

2.1.2 Simulation components

In the chart in figure 2.5 are depicted all the file necessary for the simulation to run correctly as are called by the main launch file with the appropriate path relative to the machine utilized reported in the image.
In the base PX4 package are already present some launch files ready to use in the folder *Firmware/launch/…*, but for this project a new custom launch file has been created called *mavros_posix_sitl_CUSTOM.launch* to implement the wanted models, simulation parameters and world file.
Some simple modules in chart figure 2.5, which the launch file refers to, are briefly explained here, while the more customized and complex ones will be explained in detail in the next sections:

- *Iris_param.yaml* is a file in yaml format, which is a type of data serialization format used to create configuration files. In this case it is used as a data file which contains parameters to run the simulation. This allows for quickly change parameters without searching them in the various python scripts.
- *rcS* file is the PX4 Flight Management Unit initialization script, which resides in the PX4 firmware folder, and its use is to launch the PX4 autopilot bond to the Iris model.
- *px4.launch* is a *mavros* launch file that enables the *mavros* ROS module, this gives access to all the services and topics needed to communicate with the UAV.
- *empty_world.launch* (Gazebo) is launch file type which contains all of the default parameters necessary to launch the Gazebo simulator, and the commands to run both the gazebo server and client. One of the most relevant parameters listed in this file is also the physics model utilized by Gazebo, which in this case is set to “ode” model .

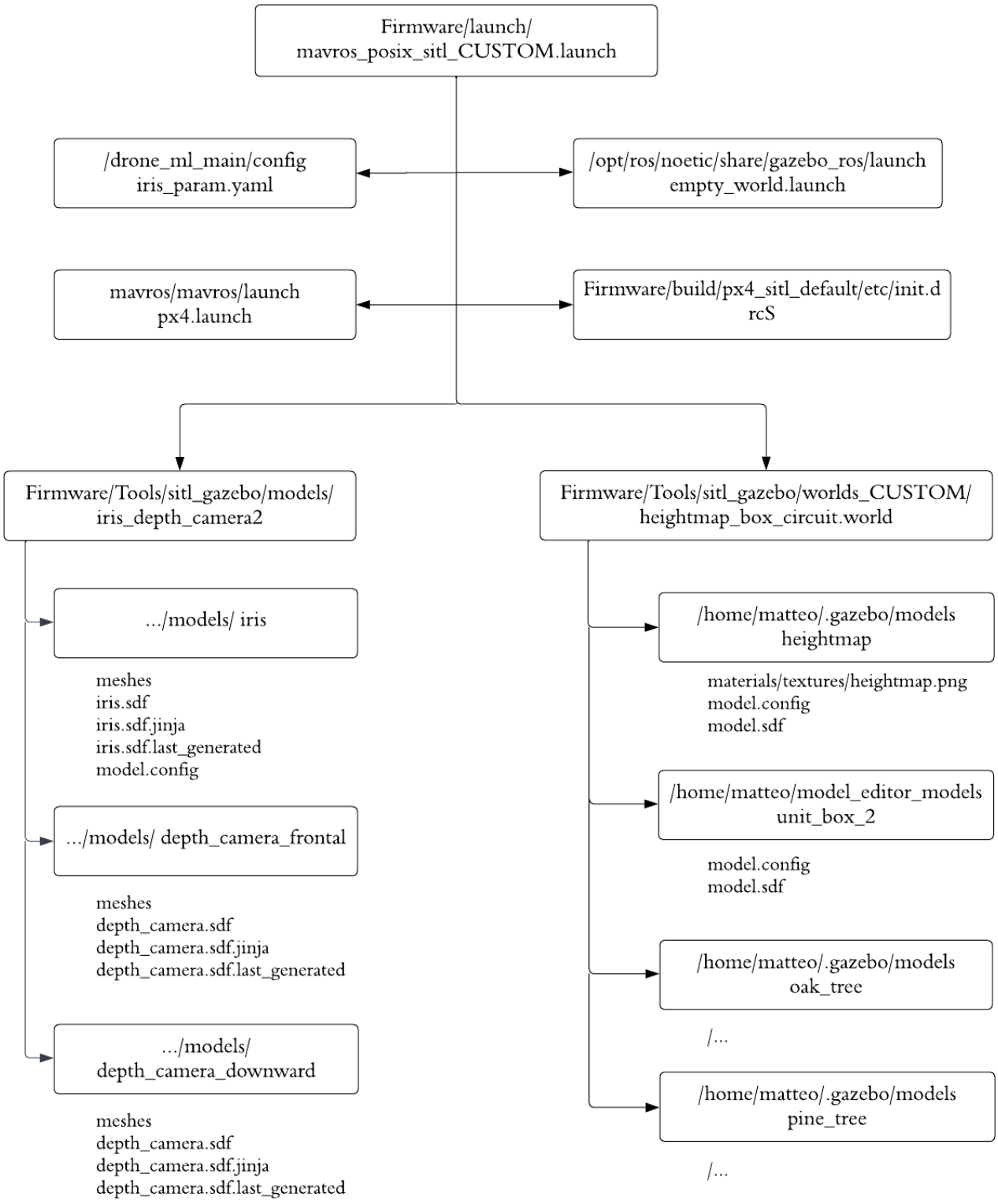


Figure 2.5 simulation file components called back by the launch file.

2.1.3 Starting the simulation

To start the simulation it is necessary to source the working folder Firmware utilizing the *Firmaware/Tools/setup_gazebo.bash* file and build Firmaware as the default PX4 folder. Then ROS path should be added both to the *Firmaware* folder and then to *Firmware/Tools/sitl_gazebo* folder. Finally, the custom launch file can be called using *roslaunch* ROS command.

Every step described before can be performed by the following lines of code in a Linux terminal, figure 2.6:

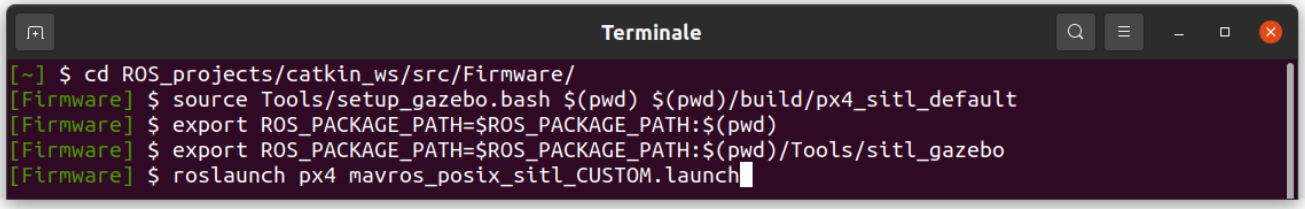


Figure 2.6: commands in Linux terminal to correctly start the simulation.

When the simulation starts it is important to notice the *roscore* ROS node is already running due to the *.launch* file configuration, thus is not necessary to launch a new one from terminal as suggested in the standard ROS procedure.

The Gazebo client is automatically deployed in a new Linux window and, if the simulation is launched correctly, should look like in figure 2.7, where the UAV is situated in local coordinates *(0, 0, 0)* at the center of the physical environment.

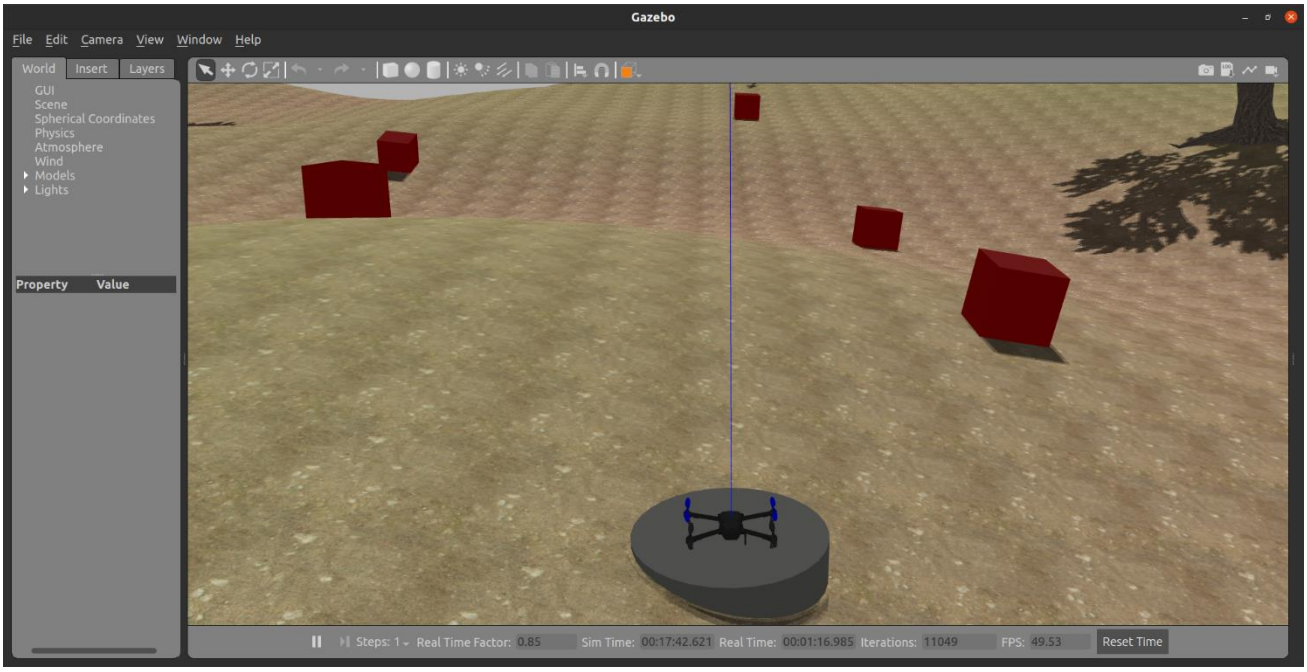


Figure 2.7: Gazebo client at the start of a simulation launch.

2.1.4 Reference systems

In this section a brief explanation on frames of reference used during the project is presented.

In figure 2.8a, the frame of reference body of the UAV, the one represented with longer and wider axes, and the camera frame of reference, with smaller axes, are depicted. These two frames are fixed to one another and rotated accordingly to values reported in chapter 2.2.2.
Differently form the standard body axes configuration often used in aeronautics, Gazebo Iris model utilized a body frame with axis having origin in the CG (Center of Gravity) of the UAV, x axes directed forward, z axis upwards, from the bottom to the top of the UAV model, and y axis that completes the right-handed triad. Instead, the camera reference frame is centered in the CG of the camera, and it has the x axis going to the right side, the y axis going downwards, with respects to the view direction of the camera model, and the z axis that completes the right-handed triad.
The local reference frame, depicted in figure 2.8b with the body frame, is fixed to the ground and centered in the origin point of the UAV [0, 0, 0].
The last frame utilized, which was not possible to depict, is used to give commands in the ROS topic *mavros/setpoint_raw/local*. The topic, if called with the appropriate *type_mask*, provides the PX4 autopilot with linear or angular velocities to follow in a frame centered in the UAV's body but with x-y plane parallel to the x-y plane of the local frame. This relative frame will turn in yaw as the body frame does. This allows to give commands easily without taking into account body and local frame relative rotations except the heading one.

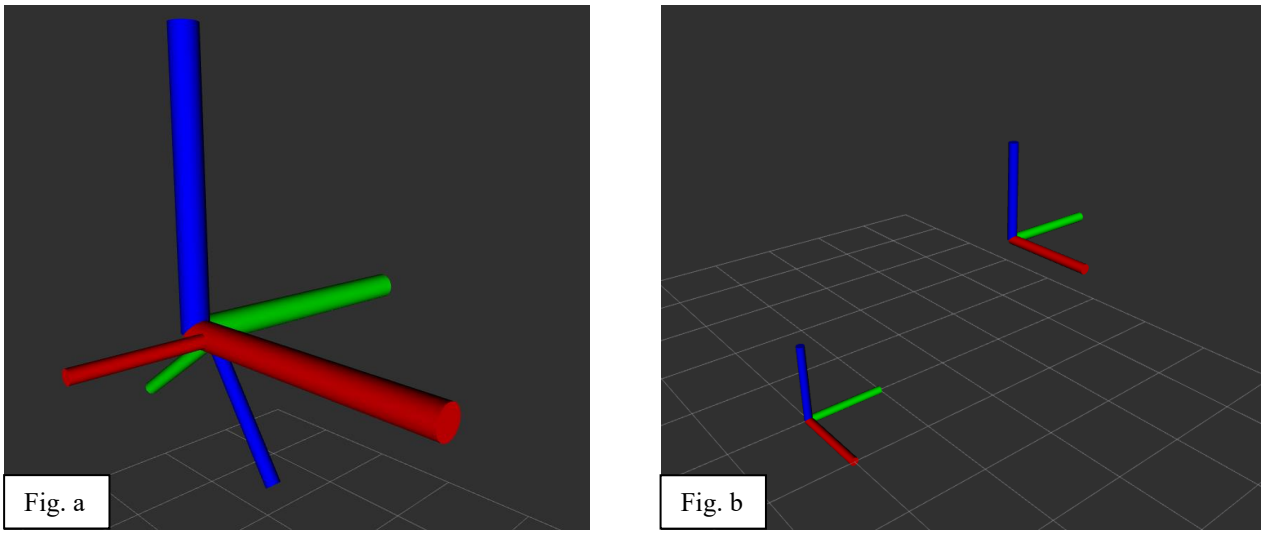


Figure 2.8:
figure a: on the left side, body UAV frame with thick axis and camera frame in smaller axis.
figure b: on the right side, local frame of reference axis in the lower left corner and UAV body frame on the top.

2.2 Camera models

In this section the design phase and implementation in Gazebo simulator of the UAV cameras chosen as payload are explained. Cameras parameters utilized in next project's phases are reported and camera models, in their final configurations, are visualized.

2.2.1 Design

Although a wide variety of Iris models with different combinations of payloads are available on the PX4 base package, for this project a custom configuration for UAV payloads has been chosen.
The specification choice for the reaching of the project objective:

- Target locking: being able to clearly observe the physical environment in front of the UAV to identify the presence of targets from afar.
- Maintained target visual contact: targets identified and followed should always be in the Field Of View (FOV) of the camera, even when the UAV is on top of the target during the application of gardening products.
- Relative altitude estimation: The depth camera should be able to view a large portion of terrain directly under the UAV so the relative altitude could be estimated.

It's not possible for a single depth camera to satisfy all the specifications, so in the first iteration of the Iris model two depth cameras were simulated as payloads of the UAV. One of the two depth cameras was attached under Iris and angled 75 degrees on the y-body axis. Its task was both to estimate the distance from the ground and maintain the visual contact with the target. The other depth camera was angled 45 degrees on the y-body axis so that the environment in front of Iris could be visible and the images of the camera were overlapped to avoid the visual loss of the target.
This first solution has been later discarded because required to unify the three channel bgr image and the pointcloud produced by both cameras into two messages. This action is both computationally demanding, since would have been required to work in real time, and difficult to code.
Instead, another depth camera configuration has been favored which consists in one depth camera pointing directly below the drone, which task is just to estimate the distance on the ground generating a point cloud. The frontal camera is angled 55 degrees referring to the y-body axis and generates both a point cloud and a bgr image, being able to visualize both directly in front of the UAV and under it.
This separation of tasks allows for a detailed optimization of camera parameters so that with FOV, and resolution and camera angles all the specifications can be respected.

2.2.2 Parameters and implementation

Camera parameters in table 2.1 are obtained through trial and error method checking result during test, visualizing cameras performances in Rviz (rviz package summary, s.d.) and moving the UAV using python modules explained in chapters 2.4 and 2.5.

	Frontal Depth Camera	**Downward depth camera**
Specification which complies to	- Target locking - Maintained target visual contact	- Relative altitude estimation
Angular position	(0 55 0) deg	(0 90 0) deg
Linear position	(0.1 0 0) m	(0 0 -0.04) m
Update rate	10 Hz	3 Hz
(Width, Height)	(112, 168) pixels	(10, 10) pixels
Horizontal FOV	90 deg	97 deg

Table 2.1: Table containing frontal and downward camera parameters as entered in the .sdf files.

For the choice on camera parameters, angular and linear positions, and FOV are chosen as function of the specifications explained before.
The update rate, width and height for the frontal camera is minimized so that less computational power is required when running the project in real time. There is a threshold on how low frontal camera resolution should be depending on the feature extraction on the U-net utilized for segmentation, better analyzed in chapter 3. The U-net must be able to recognize targets without losing accuracy.
Also, in the update frequency parameter there is a lower threshold which depends on how frequently commands updates should be given to the UAV FCU. Some tests in the simulated environment highlighted that an update rate under 7 Hz generates instabilities in drone behavior when in OFFBOARD mode.
For the lower threshold of the downward camera can be used for both update frequency and for camera resolution, this is because as mentioned before, the U-net does not take as input images from this camera but just from the frontal one. So, to evaluate height from the ground is sufficient to take data in a (10, 10) resolution 100 grid. For the update frequency in this case are allowed slower update rates on the z-controller so a 3Hz has been chosen to limit computational load.

As for models in the launch file in figure 2.5 the *iris_depth_camera2* model calls the base Iris model *.../models/iris*, which has not been modified and the two camera models *../models/depth_camera_forntal* and *.../models/depth_camera_downward* which are custom models created using *.../models/depth_camera* as base but with the different parameters discussed before.

2.2.3 Visualization

In Rviz the final configurations look as in the figure 2.9, where the frontal and the downward cameras are highlighted in red to make them visible, thus being able to notice their positioning and inclination from different angles.

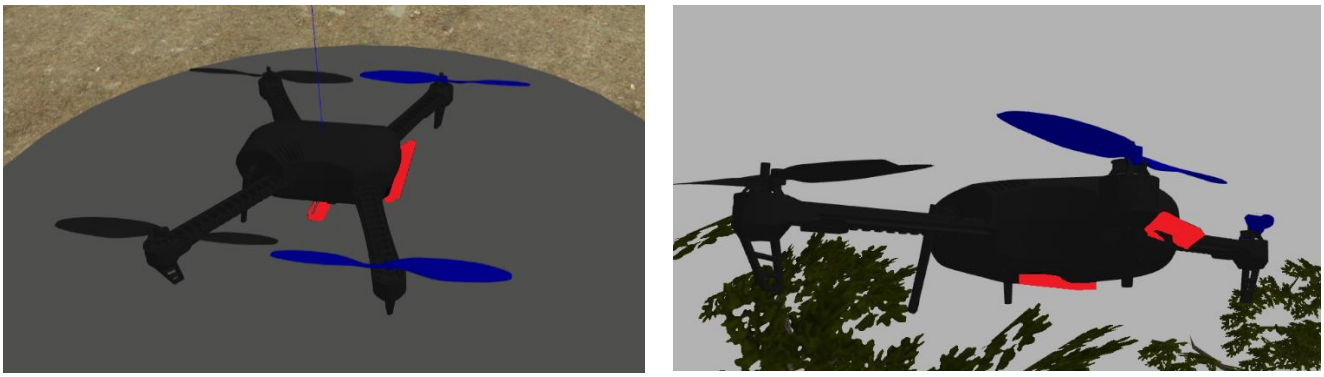


Figure 2.9: frontal and downward depth cameras highlighted in red fixed respectively on the frontal and lower part of the Iris UAV, as captured in the simulated environment.

For the downward camera, in Rviz both the pointcloud and the image are visualized in figure 2.10, where the UAV is near an oak tree model. Points shown are relative to every pixel evaluated by the downward camera in the (10, 10) grid. On top left the frontal camera image is depicted as a reference to understand the UAV position, and on the bottom right it is visualized the output image from the downward camera, which is not utilized in the project but is shown so that the pointcloud can be confronted to the tree position.

In figure 2.11, the point cloud and image generated by the frontal camera observing two tree models are depicted. As shown the point cloud is much denser than the previously shown downward camera and the image is more defined. Also, it is possible to see the point cloud shape extends both in front of the UAV and under it as specification requested.

Figure 2.10: Right side: pointcloud generated by the downward camera. Top left: image generated by the frontal camera. Bottom left: Image generated by the downward camera.

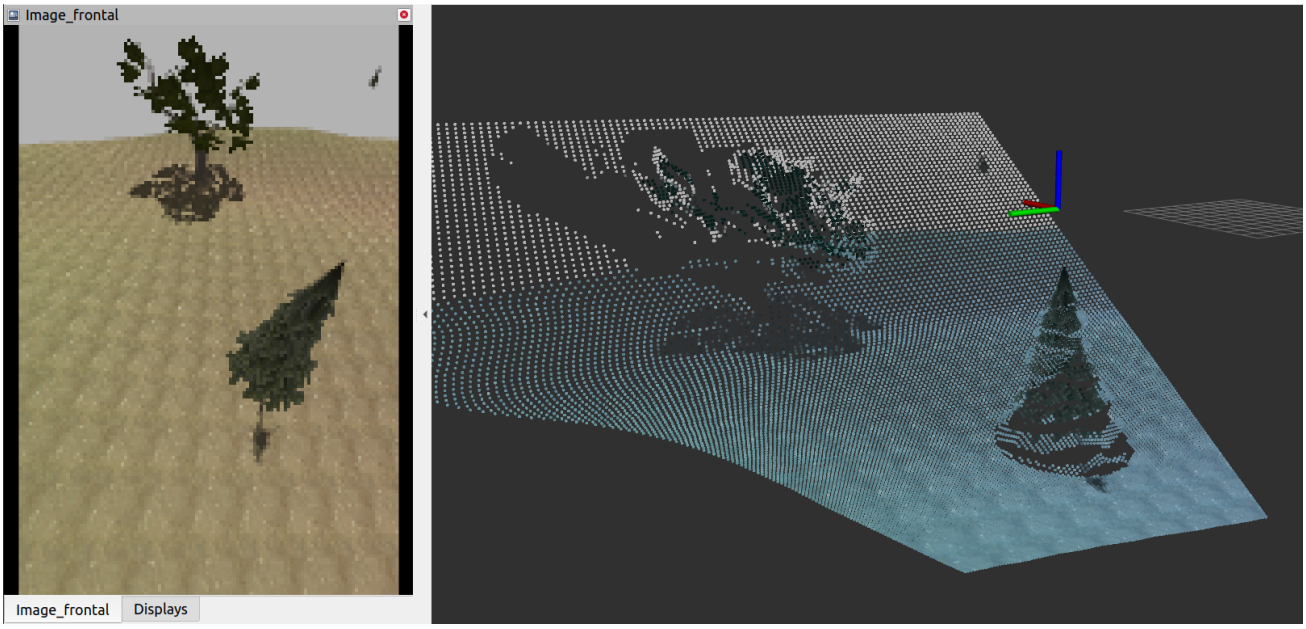


Figure 2.11: Right side: pointcloud shape when observing two tree models. Left side: image generated by the frontal camera.

2.3 Physical environment

In this section the design phase and visualization of the physical environment in Gazebo are presented.

2.3.1 Design

For this project the aim is to simulate an environment which can represent closely a real environment where the UAV could be hypothetically deployed. The ability to simulate a realistic environment is crucial for a future real application of the algorithm.
For instance, the segmentation model can be designed and trained quicker in the simulation environment to then feed a real data set after the design phase, guide algorithm can be tested quickly and safely in the simulation if represents a real working environment.
Unfortunately, it was not entirely possible to achieve this objective because of the lack of realistic models in the Gazebo libraries, so a list of minimum requirements for the environment has been created to ensure a standard for the design and testing phase of the algorithm:

- Uneven terrain: crucial to test the effectiveness of the z-controller, end to ensure the UAV can fly over targets at the correct distance.
- Non target models (trees): to provide a varied input source for the U-net model, to generate disturbances in the target recognition task.
- Varied ambience color scheme: both in the terrain and in the non-target models, to generate disturbances in the target recognition task.
- Shadow casting models: to make more challenging the target recognition with a disturbance which is close to the target itself.
- Targets to reach: it is mandatory for the design and test of the algorithm.

Trees models are not considered in this discussion as are basic models available in Gazebo standard libraries. In the next chapters, the generation of the terrain model and characteristics of targets are discussed.

2.3.2 Implementation and visualization

The terrain is generated staring from an heightmap, extrusion of a 2D image, which can be used to produce a 3D Gazebo model.
To generate the heightmap Gimp (Gimp homepage, s.d.) image manipulator software has been used, with the option “Render-Clouds”.

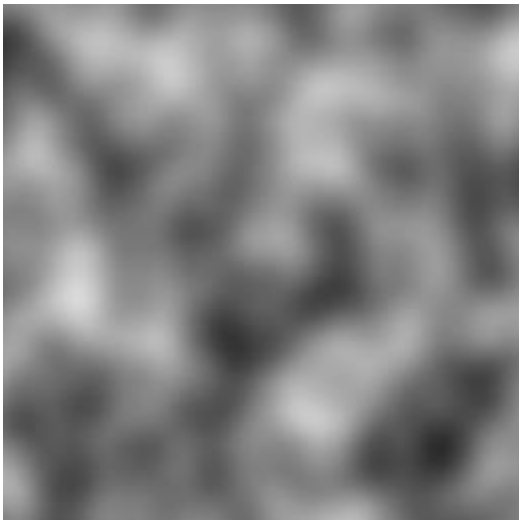


Figure 2.12: heightmap.png, distribution output of Render-Cloud of Gimp software.

Gimp outputs a bidimensional distribution of numbers saved as *hightmap.png*, image in figure 2.12. The number distribution generated has the property to be continuous in both axis, the result shapes are random generated continuous slopes and hills. As depicted in the figure 2.5, the heightmap model file is saved in */home/matteo/.gazebo/models* where *model.sdf* is present and recalls both the *heightmap.png* file and the textures files which are saved in the Gazebo file path */usr/share/gazebo-11/media/materials/textures*. In *model.sdf* it is also possible to scale the max height of the slopes of the model with the appropriate parameter.
Textures are then applied on top of the generated model to give a more realistic view from the frontal camera of the UAV. The textures are Gazebo textures already used in some other basic flat terrain models available in the basic package, which represents a grass field with yellow-brown-green shades, figure 2.13.
The complete environment can be seen in figure 2.14, where the ground model is generated with trees models and targets.

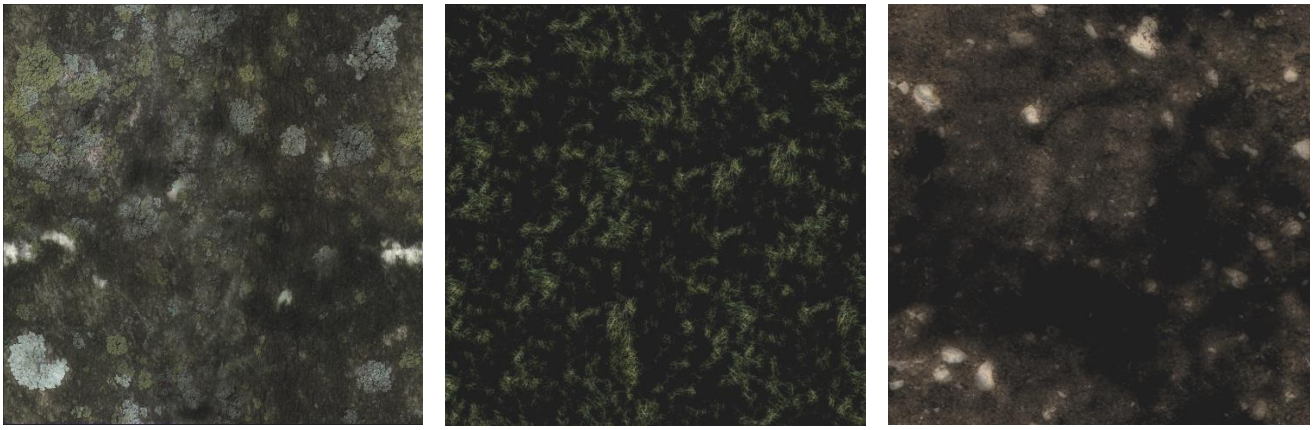


Figure 2.13: textures applied on the heightmap model.
From the left side to the right side: fungus_diffusespecular.png, grass_diffusespecular.png, dirt_diffusespecular.png

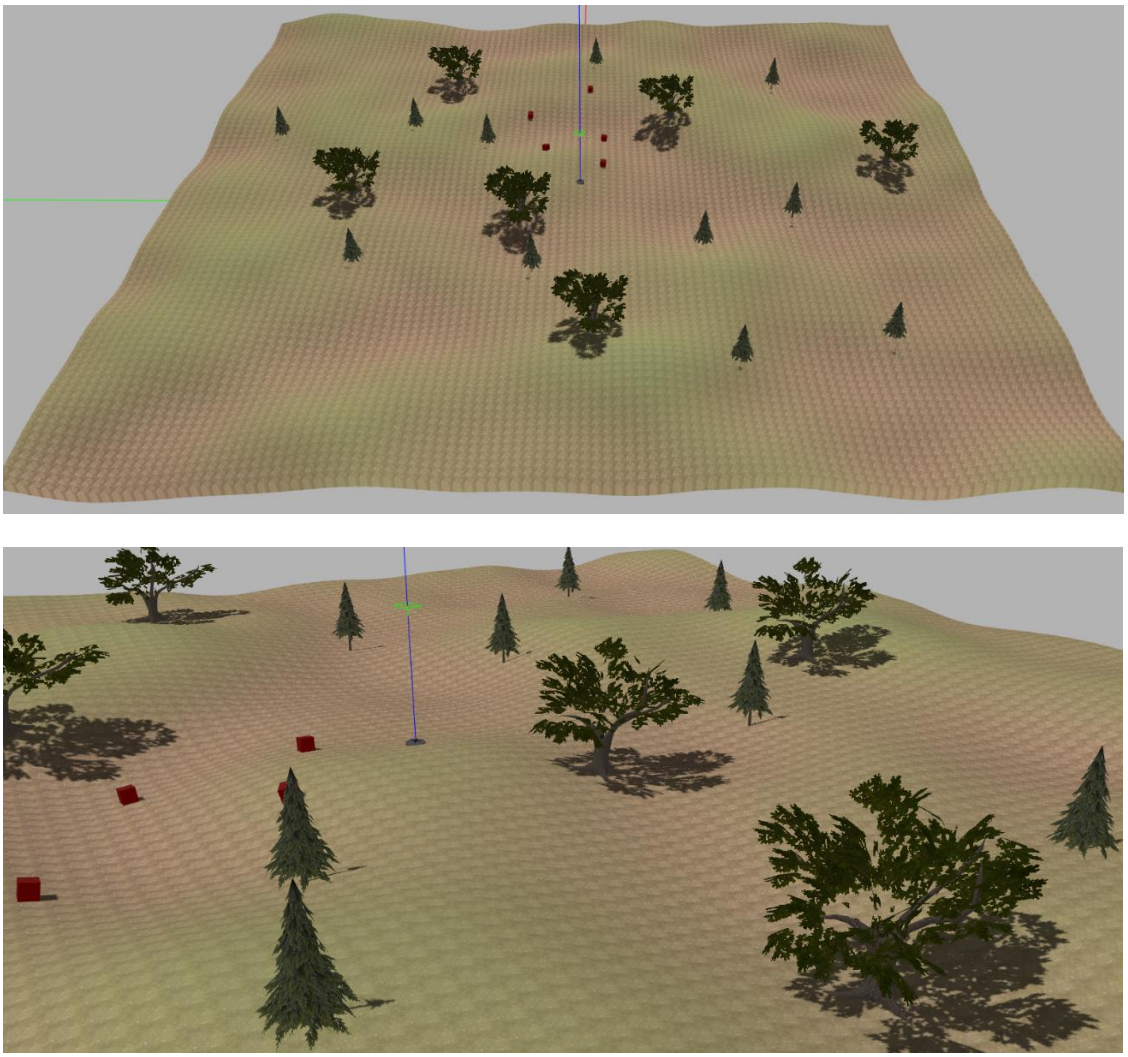


Figure 2.14: The physical environment generated in the Gazebo simulator, complete with oak and pine trees and targets.

2.3.3 Targets

As mentioned in the introductory chapter of the design choice of this project, red cubes objects are utilized as targets for the UAV to follow their positions.

The model for the target, as depicted in figure 2.5, is a custom model */model_editor_models/unit_cube_2* called by the *.launch* file. Although their positions are initialized at the start of each simulation run, the five cubes are moved in random positions at the start each of episode or to a preselect one.

In figure 2.15, targets simulated in Gazebo are depicted.

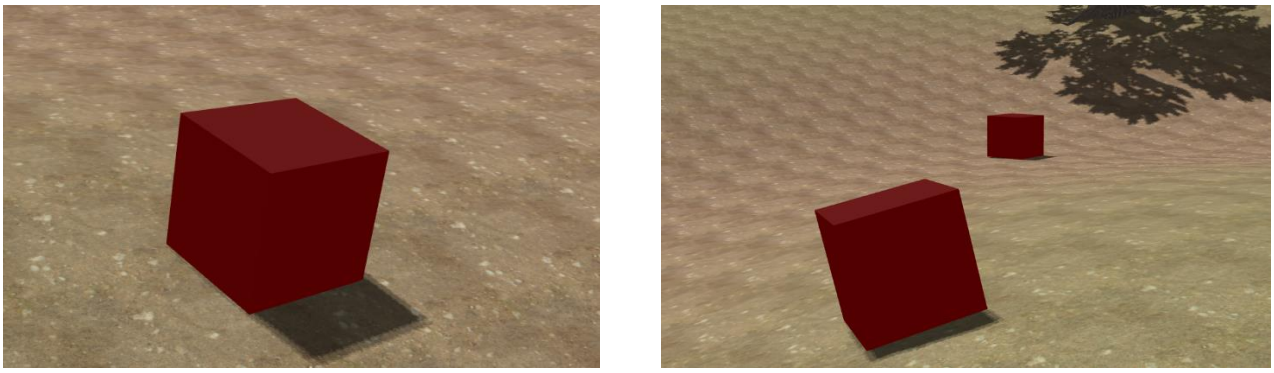


Figure 2.15: targets from different angles as seen in the Gazebo's simulation client.

2.4 Flight modes and UAV state

In this section the aim is to implement the reading of UAV basic information during the simulation and compose a state ROS message. Then build methods which allows the UAV to switch PX4 flight modes, action required to effectively allow the UAV to move in the simulation.

2.4.1 Introduction

First of all, basic information regarding the UAV is gathered in the UAV state message, so that are easily accessible in other more complex methods later implemented.
The information wanted for the UAV state message compositions are:

- Drone flight mode and state
- Linear position
- Angular position, attitude
- Linear velocity
- Angular velocity
- Relative altitude from the ground
- Global position in ellipsoid coordinates frame

Note that the UAV state message also includes the flight modes and the autopilot state here explained below.

As for flight modes, in the PX4 autopilot are available a multitude of flight modes, some useful if a RC controller is utilized in a non-simulated scenario which will be neglected now. Flight modes used in this project are listed and briefly explained below (mavros custom modes, s.d.):

- OFFBOARD: allows the UAV to actuate commands given using the *mavros* topic /mavros/setpoint_raw/*local* if a continuous stream of messages is pushed through this topic.
- AUTO.LOITER: the UAV will stop to follow commands updated through /mavros/setpoint_raw/local topic and will hover around the last coordinates known when the switching between flight modes is given.
- AUTO.RTL: the UAV will interrupt the mission, fly back to the initial take off point and land.
- AUTO.LAND: the UAV will execute a landing in the local coordinates in which is the drone when the flight mode switching happens.

The PX4 autopilot then has other two states, not to be confused with UAV states presented at the beginning of this section, to which can be set to. Armed state allows the autopilot to start rotors thus making possible the

execution of operations, disarmed state locks the rotors and makes impossible to take off from ground. If the UAV is on ground and are not given commands which result in a takeoff, the autopilot will automatically set itself in disarmed state.

2.4.2 UAV state message composition

To collect all the necessary information in real time *sripts/commander.py/updateStateAct()* function must be called. The figure 2.17 shows the flowchart representing the function logic.
On the left side, all the *mavros* topics read through a series of *rospy* subscribers are listed. The callbacks function cycled in *updateStateAct()* fill the *DroneState* custom ROS message, defined in the *catkin_ws/src/drone_ml_main/msg* as in figure 2.16, which then gets used in all later implementations to access the UAV states. Separately it is also saved the drone *State()* message, which contains the UAV state armed/disarmed.

DroneState.msg

msg > DroneState.msg

```
1  geometry_msgs/Vector3 position
2  geometry_msgs/Quaternion attitude
3  geometry_msgs/Vector3 attitude_eul
4  geometry_msgs/Vector3 linear_vel
5  geometry_msgs/Vector3 angular_vel
6  std_msgs/Float64 rel_alt
7  geographic_msgs/GeoPoint GLOBAL_position
```

Figure 2.16: DroneState ROS message definition.

All the velocity and positional value, except for the global position, are in local reference frame.
It is important to notice there aren't any ROS nodes and publishers active during this cycle, which means *DroneState* message will be only accessible in the python module in which the function is called.

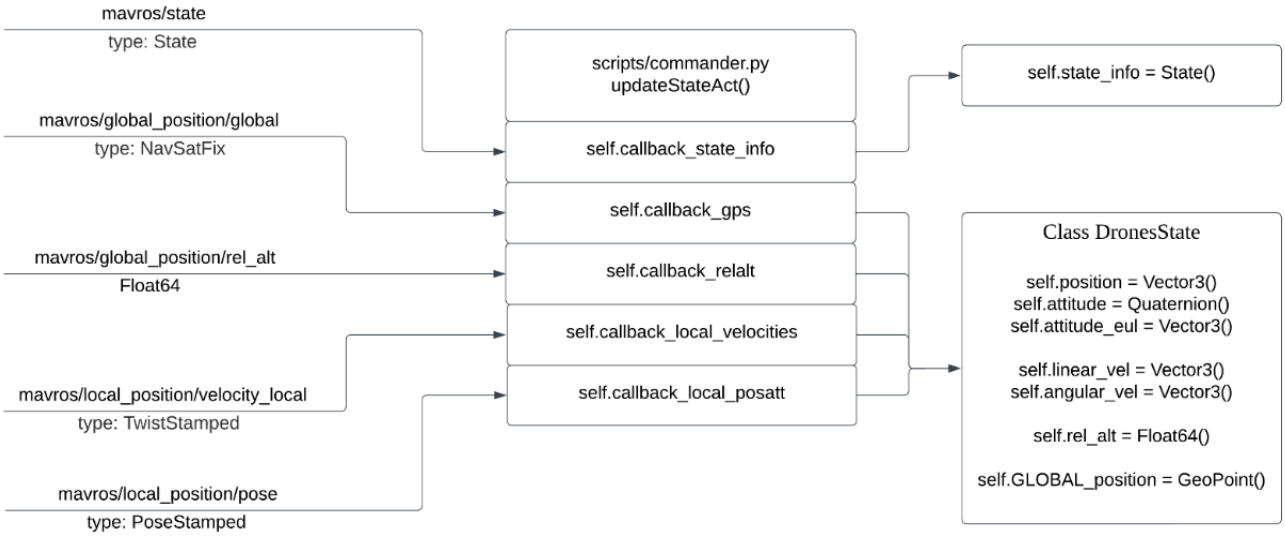


Figure 2.17: Flowchart representation of the updateStateAct() function.

2.4.3 Flight modes methods implementation

In figure 2.18 are represented all the python methods implemented in *commander.py/Commander* class which allows to switch flight modes, with the specification of all the *mavros* services used and what message type is passed to the service call. The information is then passed to the FCU of the UAV.
Below are listed functionalities of every method in figure 2.18:

- *setArm():* sets the state of the UAV to armed, which allows to actuate commands
- *setHoldMode():* sets the flight mode to AUTO.LOITER
- *setOffboardMode():* sets the flight mode to OFFBOARD
- *setDisarm():* sets the state of the UAV to disarmed, note the method call is only accepted when the autopilot has detected a landing, otherwise the method call will result in a failed UAV state change.
- *setForceDisarm():* sets the state of the UAV to disarmed, unlike the previous method this call disarms the UAV independently of the landing state. (e.g. If the UAV is flying and the method is called, rotors will stop to spin resulting in the crash of the UAV to the ground).
- *setAutoLandMode():* sets the UAV to AUTO.LAND mode.
- *setAutoRTLMode():* sets the UAV to AUTO.RTL mode.
- *setControlPosMode(x, y, z):* this method uses some other previously described methods as listed in figure 2.18. In this case mavros type_mask parameter is used to switch type commands to follow, from (x, y, z) axis velocity to 3D array of local coordinates to reach.
- *setControlPosYawMode(x, y, z, yaw):* similar to the previously described method, but the UAV in this case also follows a setpoint yaw command.

Other two methods are implemented which may result useful for future work, this time communicating directly with Gazebo simulator.

- *pausePhysics():* pauses the simulation, the simulation time is stopped and also the integration of motion equations of the UAV and other dynamic elements.
- *unpausePhysics():* reverts the effects of the previous command.

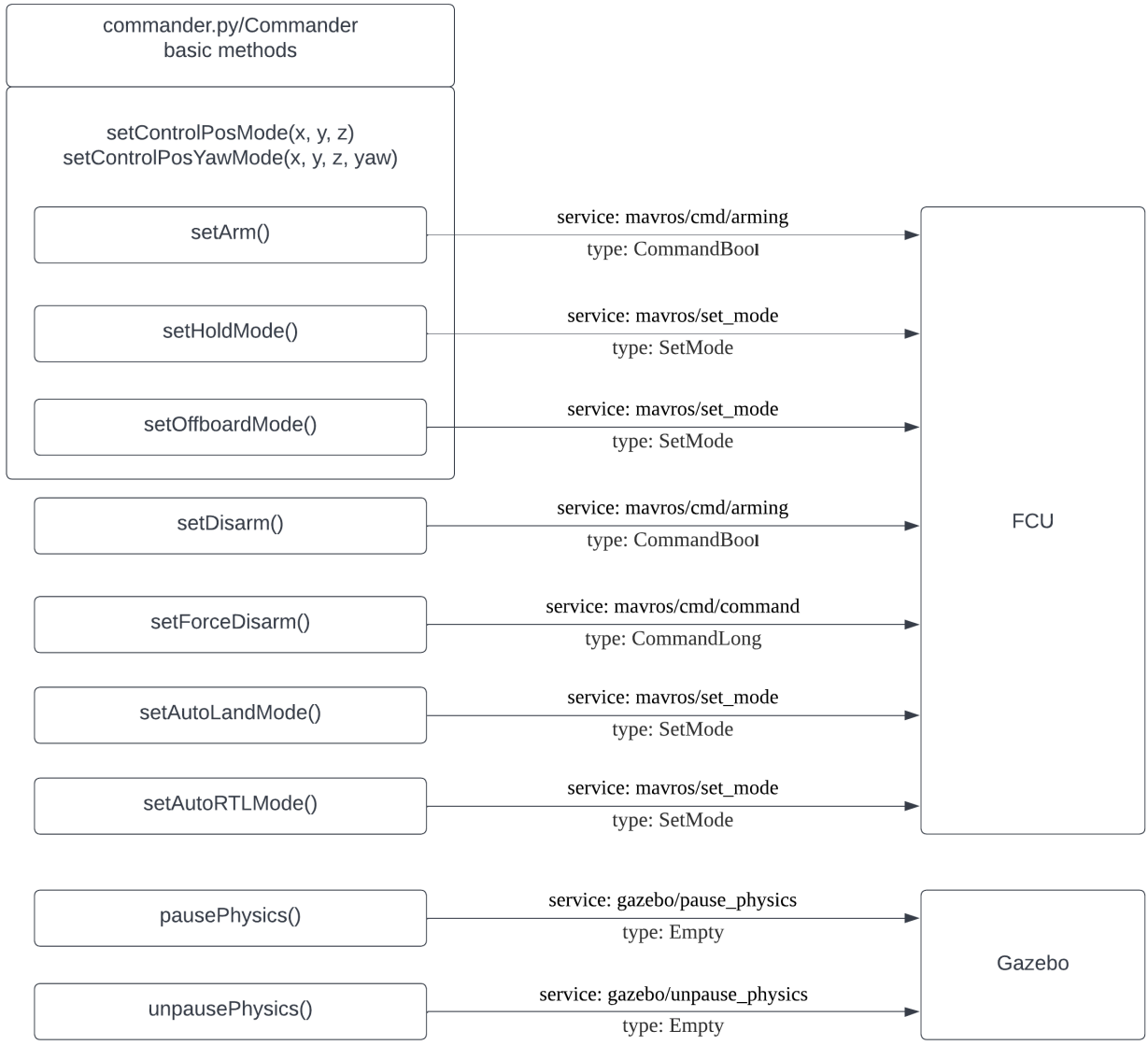


Figure 2.18: Diagram representing basic methods implemented and respective ROS services used to communicate with the UAV's FCU or to the Gazebo Simulator.

2.5 Key controls

Moving around in the environment the UAV in a natural way from keyboard is a basic functionality which is implemented as soon the simulation is ready to run. Having the ability to pilot the UAV is crucial for various reasons:

- Exploring the generated environment utilizing the point of view of the UAV.
- Taking snapshots with the frontal camera to generate a data set for the segmentation model later explained in chapter 3.
- Testing and designing the allocation algorithm later explained in chapters 4 and 6.

2.5.1 Implementation

For the development of the algorithm *mavros* topics are used to move the UAV. Figure 2.19 highlights which python modules are used, and which messages are passed between each other to permit the UAV to move.

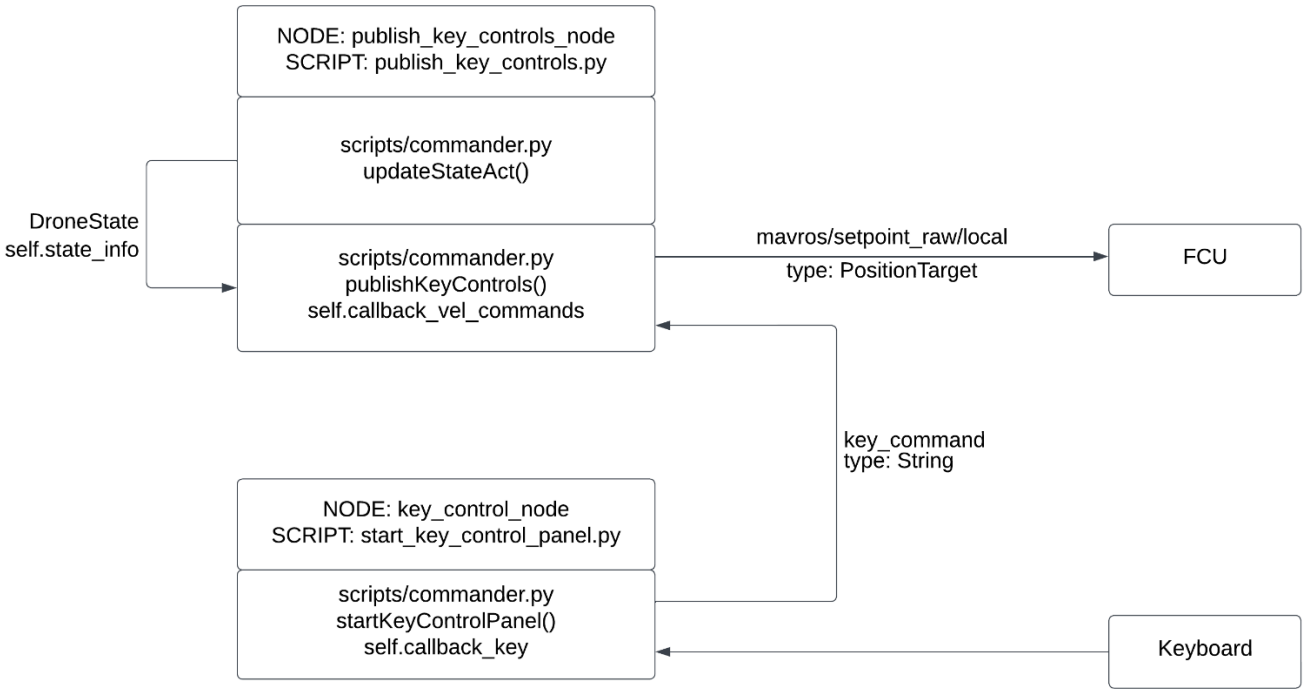


Figure 2.19: Flowchart representation of the moving drone python modules implementation.

To move the UAV in the simulated environment the user has to launch first the *publish_key_controls.py* script. In addition, the *start_key_control_panel.py*, opens a panel in which will be possible to type commands. The user enters commands from keyboard, which are detected utilizing the tkinter python package, (tkinter — Python interface to Tcl/Tk, s.d.) that allows to bind keys and call the *callback_key* function every time one of these keys is pressed.

In this case the callback publishes keyboard entries as String format in a ROS message, which then gets read in the *publish_key_controls.py* module. Every time a key command is received *callback_vel_commands* is called which maps every entry to an update on the message *self.position_targ_raw* published in the *mavros/setpoint_raw/local* topic. This topic communicates directly with the drone's FCU autopilot.

The message *self.position_targ_raw* published is of type *PositionTarget* which is a ROS message defined in the *mavros* package.

The *DroneState* message and *self_state* info get generated in the *updateStateAct()* method call, the state of the drone is the used in *callback_vel_commands* to correctly update the drone flight mode and state depending on the user's commands.

2.5.2 Command list and parameters

The table 2.2 reports all the command which are possible to give form keyboard, what every key does, and all the *commander.py/Commander* methods which are called back.

Some notes on the commands implemented:

- Directional commands: ('w', 's', 'a', 'd', 'Up (↑)', 'Down (↓)', 'Left (←)', 'Right (→)') automatically sets the UAV to OFFBOARD mode if is not already in in it.
- Directional commands: they stop the UAV in the motion axis if the opposing one is pressed while the UAV has a velocity in the other direction.
  (e.g. The UAV is proceeding onward because the user pressed 'w', if the user presses 's' then the UAV stops its motion in that axis. If another 's' is pressed the UAV starts to move backwards).
- Once pressed one of the directional commands the UAV continues to move without the need to maintain the key pressed.
- If the user presses 'g' then the UAV switches to OFFBOARD mode so that a 3D coordinate vector can be reached in local frame coordinates. After reaching the point with a tolerance the UAV is set to AUTO.LOITER mode.

Key	Description	Methods called
‘e’	Set the UAV to ARMED mode	*self.setArm()*
‘r’	Set the UAV to AUTO.RTL mode	*self.setAutoLandMode()*
‘g’	Set to OFFBOARD mode and asks the user to insert point to follow	*self.setControlPosMode()*
‘o’	Set to OFFBOARD mode	*self.setOffboardMode()*
‘h’	Stops the UAV and set to AUTO.LOITER mode	*self.setHoldMode()*
‘k’	Set the UAV position at starting coordinates WARNING: before take-off the user must wait sensors recalibration	*self.setModelStateDrone()*
‘x’	Sets the UAV to OFFBOARD mode Switches between normal/fast velocity modes	*self.setOffboardMode()*
‘q’	Completely stops the UAV in OFFBARD mode	---
‘w’	Sets the UAV to OFFBOARD mode Sets forward velocity > 0	*self.setOffboardMode()*
‘s’	Sets the UAV to OFFBOARD mode Sets forward velocity < 0	*self.setOffboardMode()*
‘d’	Sets the UAV to OFFBOARD mode Sets strafe velocity > 0	*self.setOffboardMode()*
‘a’	Sets the UAV to OFFBOARD mode Sets strafe velocity < 0	*self.setOffboardMode()*
‘Down (↓)’	Sets the UAV to OFFBOARD mode Sets the vertical velocity < 0	*self.setOffboardMode()*
‘Up (↑)’	Sets the UAV to OFFBOARD mode Sets the vertical velocity > 0	*self.setOffboardMode()*
‘Right (→)’	Sets the UAV to OFFBOARD mode Sets yaw rate < 0	*self.setOffboardMode()*
‘Left (←)’	Sets the UAV to OFFBOARD mode Sets yaw rate > 0	*self.setOffboardMode()*

Table 2.2: contains all the key bind from keyboard, mapped to the respective command and methods called back corresponding to the button pressed.

To move the UAV more effectively three flight methods have been implemented:

- Normal (slow) mode: allows the user to move the UAV with slow velocities and rates thus achieving precise positioning. Parameters followed by Px4 autopilot are velocities and yaw rates in the local frame of reference.
- Fast mode: allows the user to mode the UAV with high velocities and rates, thus being able to reach quickly far sections of the map. Parameters followed by Px4 autopilot are the same as the previous flight method.
- Point follow: utilizing the *type_mask* parameter (*setControlPosMode(x, y, z)*) is possible to switch what parameters the UAV has to follow, in this case the user inputs a 3d vector containing the position for the UAV to reach in local frame coordinates.

The figure 2.20 depicts graphically linear and angular velocity commands which are possible to give to the UAV from a side and a top-down view.

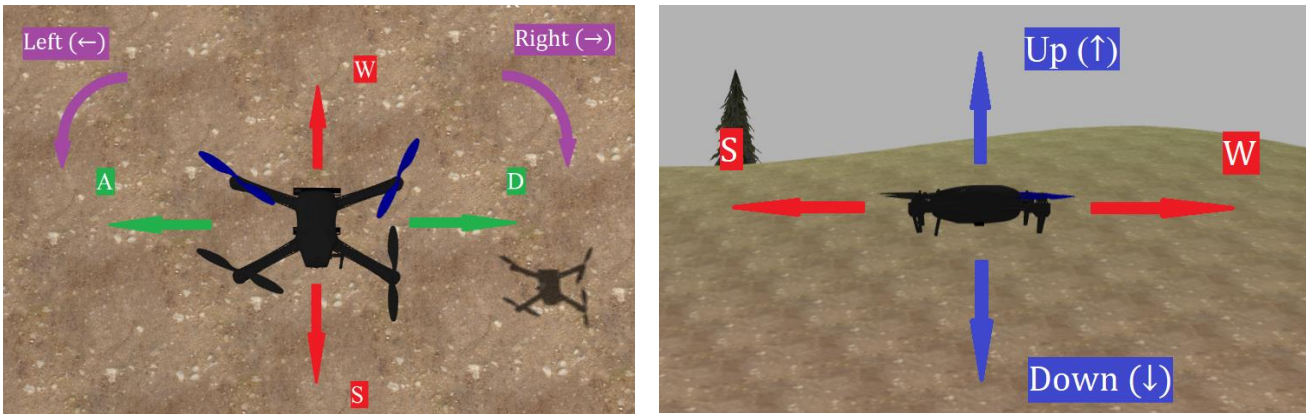


Figure 2.20: Graphical representation of rates and velocity commands available as inputs from Keyboard.

In figure 2.21 it is possible to observe the effects of point follow flight methos implemented, accessible for the user after pressing 'g' on the tkinter window.

The user should insert a 3D array containing (x, y, z) coordinates for the UAV to follow in the local coordinates frame. In 2.21a, the input array is (0 0 2), after the input the UAV takes off and reaches the desired coordinates, on top of the spawning area. In 2.21b the input array is (-5 -2 4).

In both cases the UAV reaches the desired coordinates correctly.

The time between the actual reaching of the point and the stabilization of the UAV with a tolerance of 0.07 m over it is tested to be always less than a few seconds. This value depends on the tolerances used and the dynamics of the UAV during the actuation of the command, thus can vary between being almost instantaneous, in the case the UAV is already in the right position, and a max of around 10 s in the worst cases, with a mean value around 2-3 s.

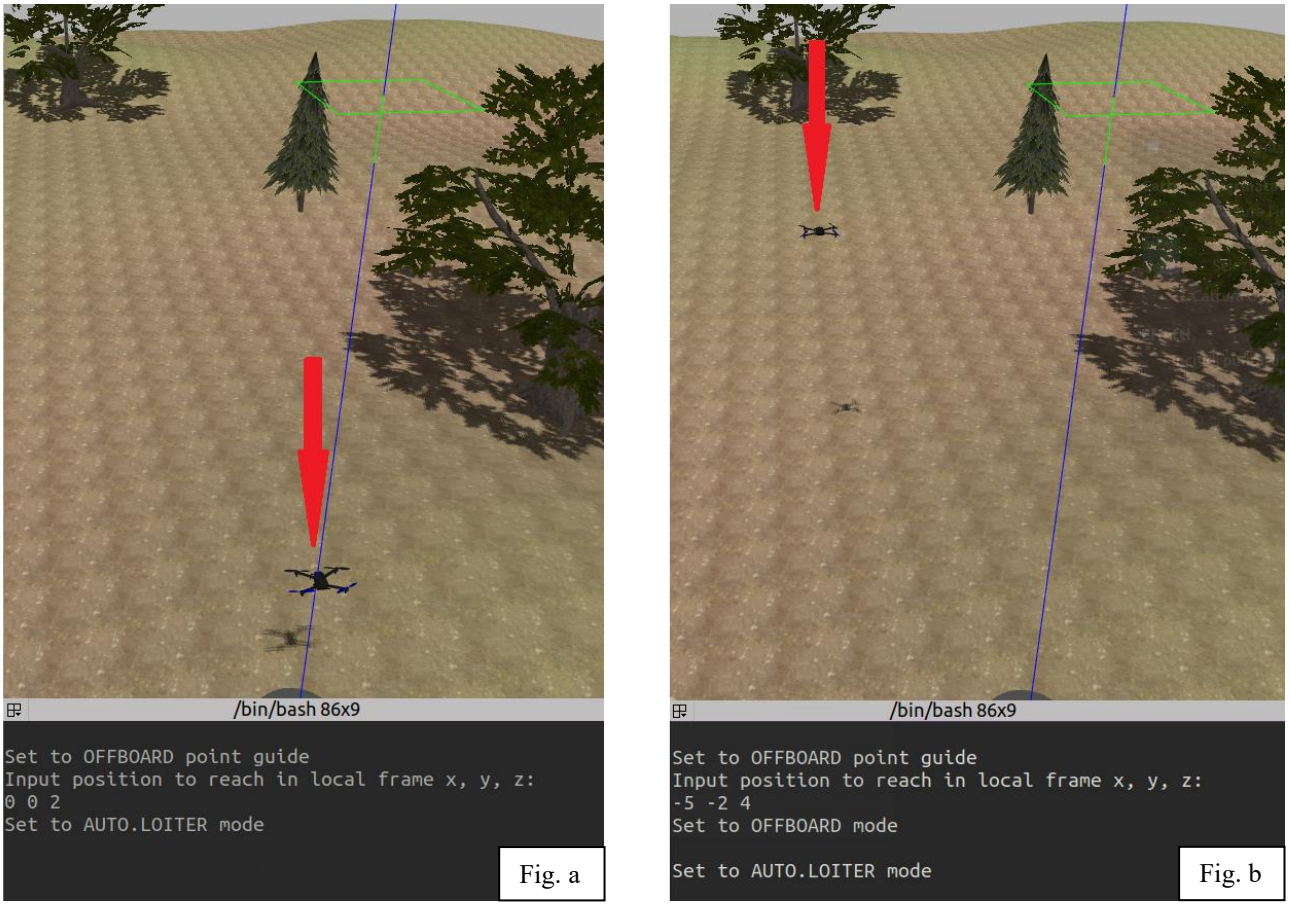


Figure 2.21: Point follow flight method implementation with terminals.
figure a: representing input position (0 0 2) and starting point of the test.
figure b: one representing (-5 -2 4) coordinates reached after the command.

In table 2.3 parameters are reported which are used to execute key controls modules for the UAV with a brief explanation of what they do.

Parameter name	Value	Notes
linvel_const	0.7 m/s	Linear velocity value for the UAV to follow in all three directions when flying in "Normal (slow) mode"
angvel_const	30 °/s	Angular rate for yaw in "Normal (slow) mode"
linvel_const_s	3 m/s	Linear velocity for "Fast mode"
sngvel_const_s	90 °/s	Angular yaw rate for "Fast mode"
pos_toll	0.07 m	Box tolerance, in all three directions, for the UAV to stabilize itself in the wanted position when flying in "Point follow"
vel_toll	0.04 m/s	Velocity tolerance in "Point follow mode". Necessary to avoid the overshooting of the right position
publishkey_rate	20 Hz	Publish rate for the ROS publisher in */mavros/setpoint_raw/local* topic

Table 2.3: parameters table for key commands python modules.

3 Segmentation model

In this chapter, the target recognition problem is explained, and a solution method is proposed, which consists in designing a CNN (Convoluted Neural Network) as a U-net. After defining the NN (Neural Network) architecture, the steps used to prepare and preprocess the data set needed during the training process are listed and explained. The final subsections of the chapter are dedicated to the testing phase, which is carried out directly in the simulated environment, after which the results are presented.

3.1 Concept

As mentioned in the introduction, using the output of the depth camera, which comes in two separate messages, one of pointcloud type and the other in image format, the software must be able to recognize targets that are randomly scattered throughout the physical environment. Target recognition is essential to obtain target distances and locate them in local coordinates, thus providing the necessary information to start developing control methods to follow target positions.
This section explains in detail the model used for the recognition task and its implantation.

3.1.1 Recognition task

The aim is to recognize targets in the simulated environment. The first step in defining a working method is to list a series of specifications which the software must comply to:

- Output type: the output of the model should be compatible with the next steps in the pipeline shown in figure 1.1, i.e. it should be able to provide information on both the presence of targets and their positions.
- Input type: data the method takes as input should be compatible with the output of the frontal depth camera of the UAV, so should take in data in Image format, PointCloud, a combination of both, or another data type obtained as post processing of these two data types.
- Computational speed: the recognition task has to be performed in real time, which imposes a strict constraint on the method of choice.

The method of choice that meets all the above specifications is to perform segmentation using a CNN (Convolutional Neural Network) in the form of a U-Net.
The output of the CNN U-Net is a single-channel grey-scale mask of the input image. Each pixel of the image is made up of values in the interval [0, 255], the NN estimates the probability that it belongs to the target [0, 1], this value is then multiplied by 255, thus obtaining the mask in grey scale channel [0, 255]. If the value for

each pixel is in the interval [200, 255], the pixel is said to belong to the target; if not, the pixel is said not to belong to the target. This allows great compatibility with the output of the frontal depth camera, since the data required by the CNN is just a NumPy formatting of the already available Image ROS message.
Compatibility with the rest of the pipeline is also ensured, since the output of the U-net is, as explained above, a mask containing information on both the presence of targets and their position in the camera's field of view, which allows to fuse the acquired masks and the pointcloud to obtain precise target positions.
The computational complexity is low enough to allow the task to be performed in real time, as most of the computational effort is devoted to training the U-Net, which is done offline.
Once the U-net has been trained, the only computations required when applying the model in real time are the multiplications between pixels and the stored weights of the CNN. Depending on the image quality, the task can be performed with little impact on the delay time between receiving the camera message and obtaining the target position as an (x, y) value in local coordinates.

3.1.2 U-net Neural Network model

The U-Net is a type of CNN which is state of the art for image segmentation tasks, preferred to the classic down-sampling CNN architecture in cases in which every pixel is required to have a label for localization purposes.
The U-Net is first introduced in (Olaf Ronneberger, 2015), specifically for Biomedical Segmentation tasks, but later implemented in other more general applications. The basic idea is to have a contracting network, very reminiscent of classic CNN networks, that performs down-sampling, followed by an expanding network that performs up-sampling.
The aim of the down-sampling layers is to extract image features, as in the classic CNN for labelling, which consists in updating the weights so that the filters can memorize target features in order to recognize them in the environment. The expanding part of the network instead performs up-sampling with a reverse architecture with respect to the classic CNN. The aim the U-Net is to use the filter data to generate the mask, which consists of a label for each pixel.
Another innovation introduced in the U-Net are skip connections, which perform concatenations on layers of the contraction part with layers of the expansion part. Skip connections make it possible to obtain a more accurate output by using information from previous layers that is lost in the down-sampling section.

The network used in this project is based on the original U-Net, but with some modifications. The model used is shown in figure 3.1. The differences in the implemented U-net with respect to the original one are a consequence of the task assigned to it.
The first difference is the shape of the input data, in the original U-Net the image input shape was [572, 572, 1], a square image with a single channel, whereas the implemented U-Net has a data shape of [168, 112, 3], a

square image with tree channels.

The shape [168, 112, ...] is needed to make the U-Net directly compatible with the camera output, while 3 channels are needed because the image is generated by the depth camera in bgr format.

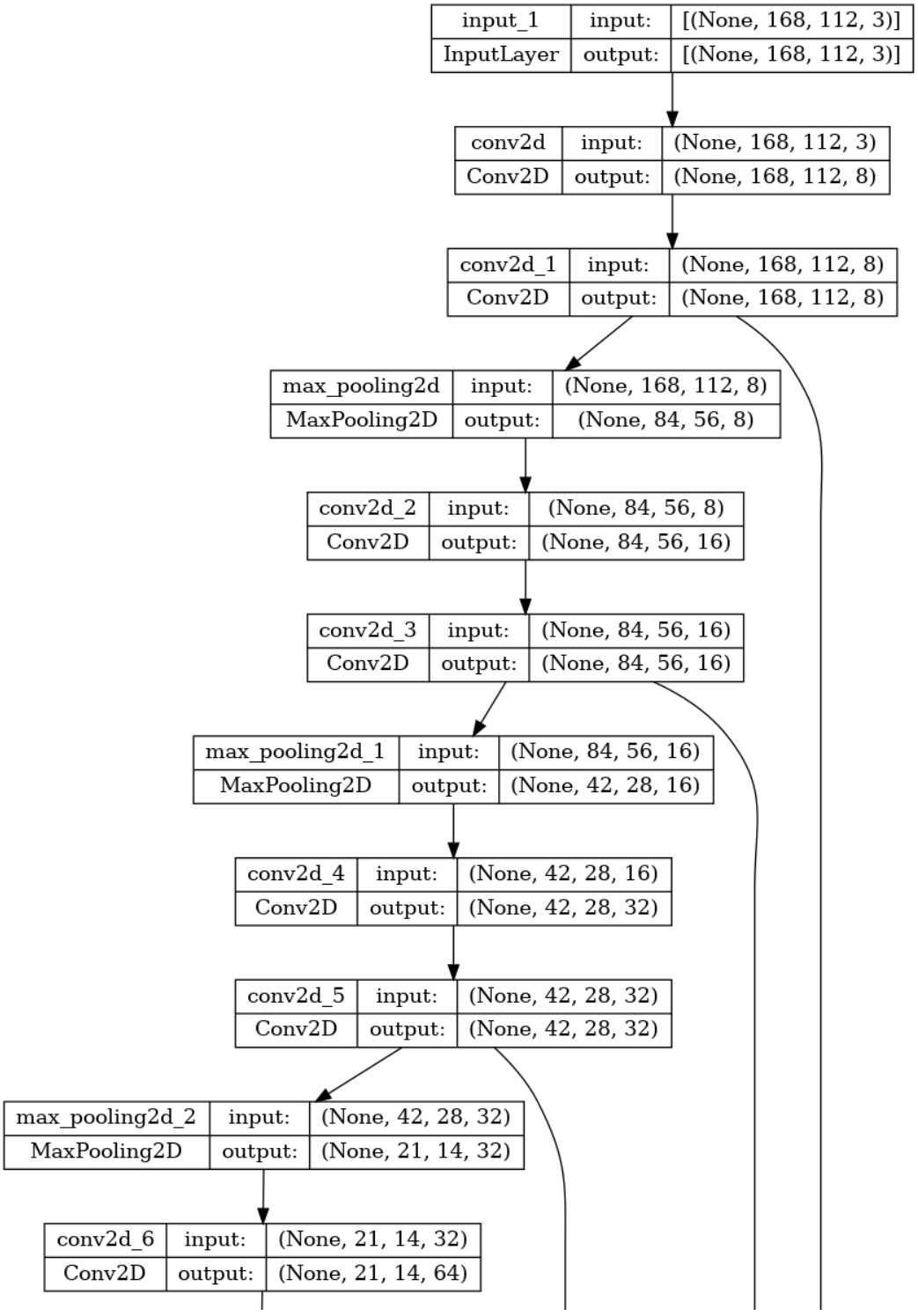
In the U-Net architecture, the depth of the network, i.e. the number of hidden layers implemented, depends on the smallest feature size of the targets that the U-Net can learn to recognize. The original U-Net takes as input an image of [572, 572, ...] and is compressed in the network bottleneck after filtering stages to [28, 28, ...] shape with a total of 26 layers. In the implemented U-Net, the goal is to maintain a similar bottleneck shape, regardless of the input shape. In the final architecture the input shape is [168, 112, ...] and in the bottleneck the data shape is [21, 14, ...] with a total of 17 layers. Tests have shown that having a smaller data dimension in the bottleneck doesn't lead to better performance on the given task.

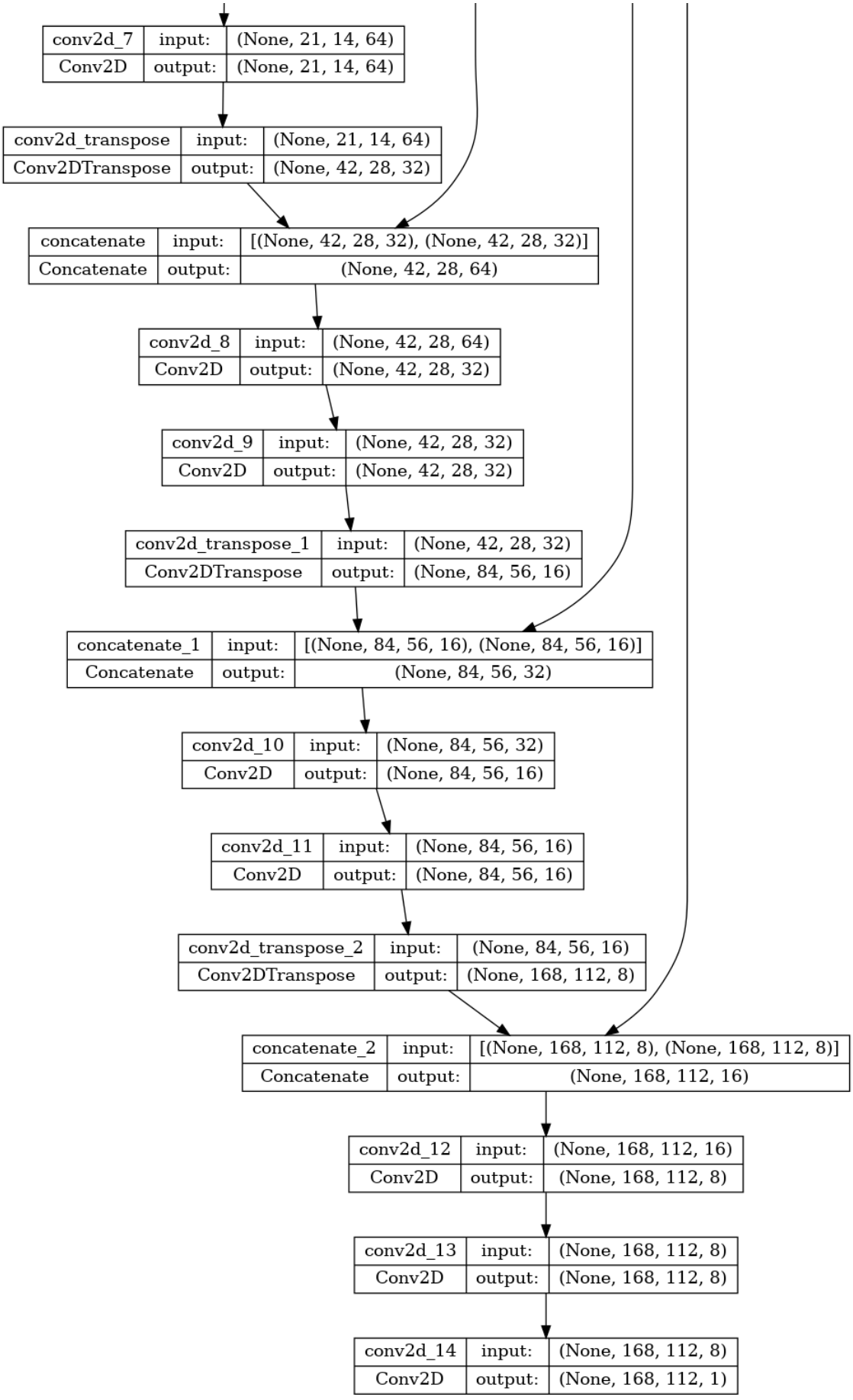
Another major difference is the number of feature maps in the convolutional layers, which are shown at each step in the graph in figure 3.1. The number of feature maps used depends on the complexity and number of features that the net has to store. The original U-Net has great generalization capabilities and can learn to recognize several objects, with a variety of features for each of them. In this case project, the task is to recognize a single type of target, so fewer feature maps are required.

The original U-Net has a maximum of 1024 feature maps at the bottleneck, whereas the implemented one has a maximum of 64.

Modifications on the original U-Net have been made to create a faster image recognition tool tailored to this project, to allow less computational complexity and reduce time delay in the real time application.

Figure 3.1: Modified U-Net model implemented in the project.




conv2d_7 | input: | (None, 21, 14, 64)
Conv2D | output: | (None, 21, 14, 64)
conv2d_transpose | input: | (None, 21, 14, 64)
Conv2DTranspose | output: | (None, 42, 28, 32)
concatenate | input: | [(None, 42, 28, 32), (None, 42, 28, 32)]
Concatenate | output: | (None, 42, 28, 64)
conv2d_8 | input: | (None, 42, 28, 64)
Conv2D | output: | (None, 42, 28, 32)
conv2d_9 | input: | (None, 42, 28, 32)
Conv2D | output: | (None, 42, 28, 32)
conv2d_transpose_1 | input: | (None, 42, 28, 32)
Conv2DTranspose | output: | (None, 84, 56, 16)
concatenate_1 | input: | [(None, 84, 56, 16), (None, 84, 56, 16)]
Concatenate | output: | (None, 84, 56, 32)
conv2d_10 | input: | (None, 84, 56, 32)
Conv2D | output: | (None, 84, 56, 16)
conv2d_11 | input: | (None, 84, 56, 16)
Conv2D | output: | (None, 84, 56, 16)
conv2d_transpose_2 | input: | (None, 84, 56, 16)
Conv2DTranspose | output: | (None, 168, 112, 8)
concatenate_2 | input: | [(None, 168, 112, 8), (None, 168, 112, 8)]
Concatenate | output: | (None, 168, 112, 16)
conv2d_12 | input: | (None, 168, 112, 16)
Conv2D | output: | (None, 168, 112, 8)
conv2d_13 | input: | (None, 168, 112, 8)
Conv2D | output: | (None, 168, 112, 8)
conv2d_14 | input: | (None, 168, 112, 8)
Conv2D | output: | (None, 168, 112, 1)

3.2 Training

This section explains in detail how the custom U-net is trained and implemented as part of the guidance algorithm for the UAV.

3.2.1 Data set

The data set is obtained directly in the simulated environment through the frontal camera of the UAV. This project choice was made to allow the UAV to access the same type of data both during the training and in its simulated mission to simplify the task, although targets and background are the same, angles and point of view and number of targets per image acquired may differ greatly during training and application.

To take snapshots of the targets the simulation in started in Gazebo with key control modules explained in the previous chapters. The python module *drone_ml_main/scripts/take_snapshots.py* allows to take a single snapshot and saves the instantaneous output the frontal depth camera sees. Captured images are saved in *drone_ml_main/segmentation_model/samples/images* in *.png* format, ordered as: [image0, …, imagen].
Since the neural network application is supervised learning, each data sample in the training set must be associated with a mask, consisting of labels for each pixel as explained in the previous chapters. To generate masks VGG Image Annotator (VGG image annotator webapp, s.d.) is used, which allows to select targets and produce a mask. After annotating every image, a *.json* file is exported from the online tool and saved in *drone_ml_main/segmentation_model/json_export.json*.
Using the python module *drone_ml_main/segmentation_model/gen_masks_from_json.py* with inputs images saved in *../samples/images* and the *json_export.json* file it is possible to generate the mask set saved as *.png* format with names [mask0, …, maskn]. Mask numbers corresponds to the image number, so that images and masks can be easily associated during the training process.
Some examples of images, with masks associated, are shown in figure 3.2.

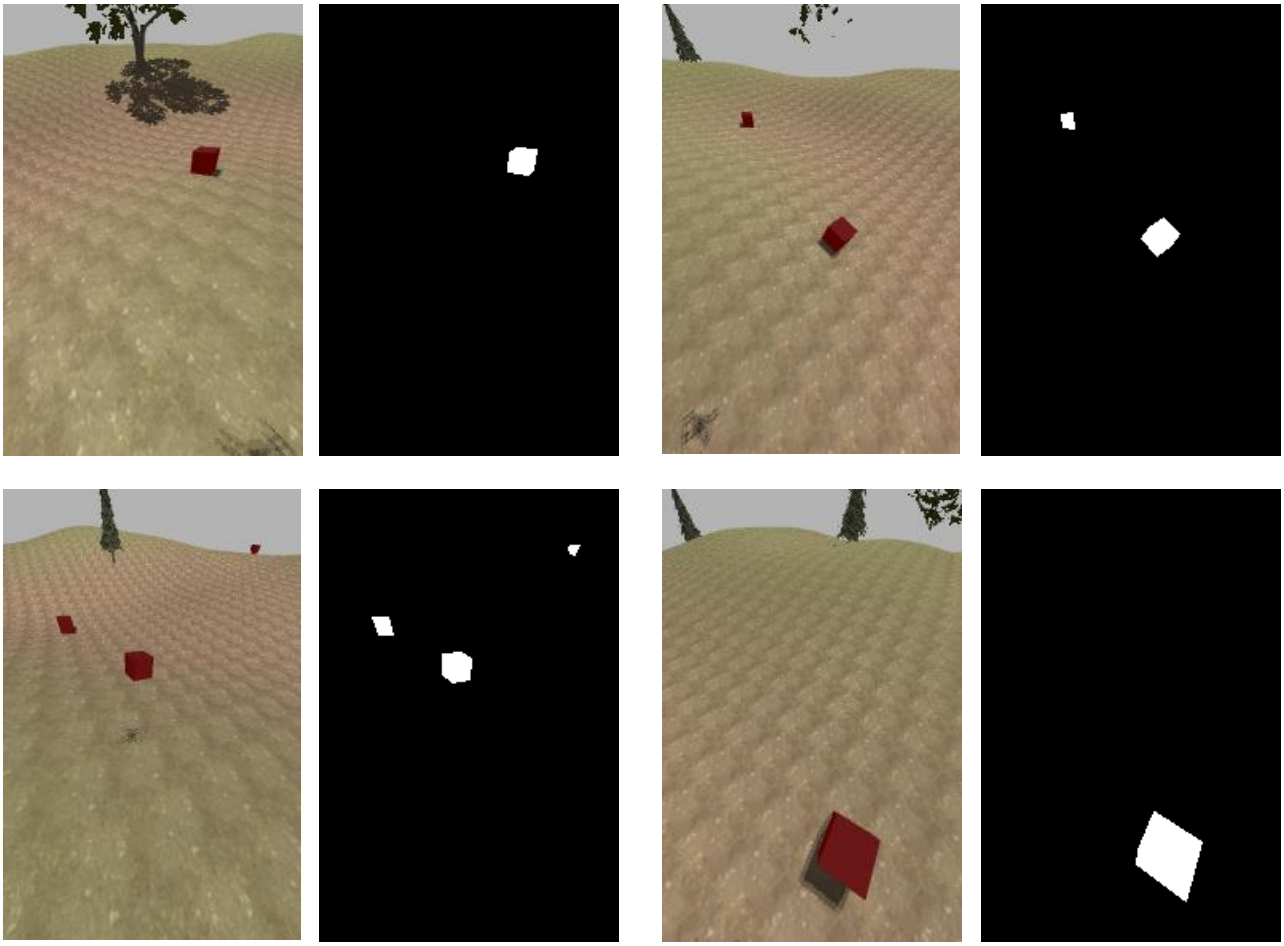


Figure 3.2: Image obtained from the frontal camera with their respective mask generated with VGG annotator software.

3.2.2 Data loader

The purpose of the data loader is to feed the neural network with the data set so that the training process can be initialized. It is common practice for networks whose goal is to label or recognize, to pre-process the data so that more data samples can be extracted from the original images, resulting in a better generalization.

In this case the data loader is coded using Tensorflow (Tensorflow homepage, s.d.) in *drone_ml_main/segmentation_model/model/train_model.py*, which already provides methods to perform the data augmentation process.

The coded data loader takes images and masks as input, generates the data set, splits the whole data set into two subsets, the training set and the evaluation set, and applies data augmentation techniques.

The goal of the training set is to provide a large data pool to train the neural network with, whereas the evaluation set aim is to evaluate the neural network results so that hyperparameters can be tuned accordingly. The evaluation set should be an unbiased set with respect to the training set, but both the training set and evaluation set are used in the training process. Finally, the test set is third subset of the complete data set

which goal is to provide a final evaluation of the trained model using a set of unbiased data samples, (Pietro Zanuttigh, 2021).
In this project the test set is represented by the actual stream of frames extracted from the frontal depth camera in the simulated environment during an online simulation.
The total data set consists of 55 images with 55 corresponding masks. The data set is divided into a training set consisting of 44 images and masks and an evaluation set consisting of 11 images and masks.
The evaluation set will remain the same during the next steps, while the training set will be modified during the data augmentation process. At each step, the training set is remapped with the modifications and added back to the original training set.
The modifications implemented in the augmentation process are listed below:

- Normalization by 255
- Brightness
- Gamma
- Hue
- Flip horizontally

An example of the augmentation process is shown in figure 3.3, where the original image is shown on the left, the modified image is shown in the middle column, and the associated mask is shown on the right.
The images and masks shown are in bgr, unlike those in figure 3.2, which were in rgb, because the actual frontal depth camera generates them directly in this format. So, to avoid the extra steps that would consist in changing the color channels, they are fed directly into the network in bgr. For this reason, the neural network must be trained with bgr images and not in rgb.
Also, in figure 3.3, the color scale is [0, 1] and not [0, 255], again because the network takes in input normalized images.
At the end of the modification process, the training set consists of 220 images. The masks are not modified, except in the flip image case, but only associated to the original image and the modified ones generated from the same image.

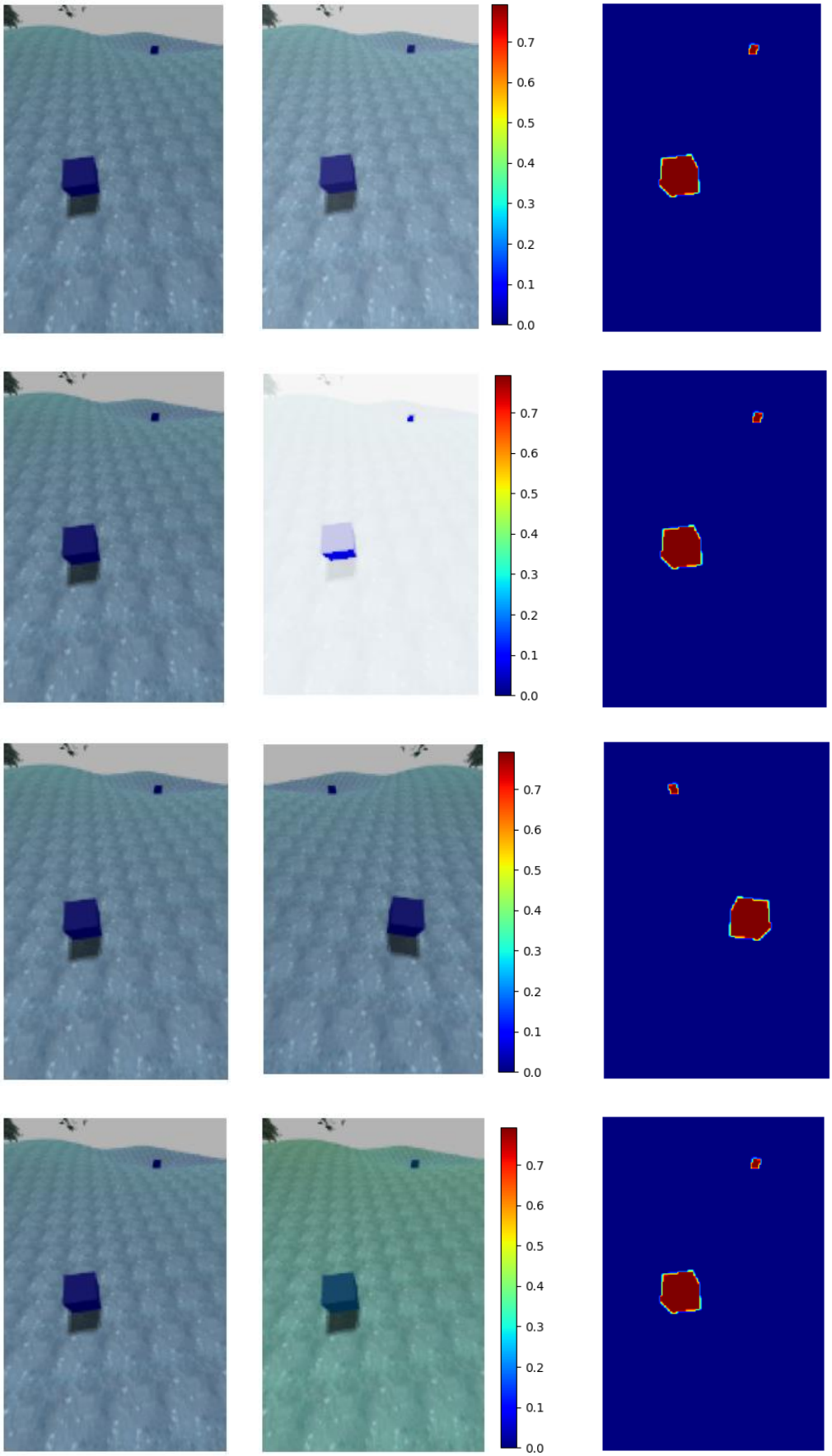


Figure 3.3: Example of the augmentation process, from top to bottom brightness modification, gamma modification, flipped horizontally and hue modification.

3.2.3 Training parameters

The neural network construction and training is made in Tensorflow and Keras (Keras homepage, s.d.) packages.

Parameters for training the U-Net, are shown in table 3.1.

Parmater	Value	Notes
Buffer size	10	Buffer allocated for picking random samples to generate the new training set. A data shuffling method.
Epochs	30	An epoch is an iteration of the entire training set, all the batches.
Batch size	5	Size of batches in which the training set will be divided in each epoch to feed the NN. At the end of each batch the neural network is updated
Optimizer	“adam”	Algorithm to change weights on the net in order to reduce the loss
Loss function	“binary crossentropy”	Loss is computed through the difference between true mask and prediction. The aim of the NN is to minimize the loss function value at each iteration
From_logits	False	Interpret the NN output as logits value. (True) Interpret the NN output as probability distribution (False)

Table 3.1: U-Net training parameters.

Before training the model, the training set is shuffled to avoid repeating the same samples in the same order at each epoch. The buffer size parameter determines how the shuffle is performed. Although it's usually a fundamental parameter, in this case, because many batches and many epochs are used, its value doesn't make much difference in the result.

Batch size determines how many batches the training set is divided into. In this case, a batch size of 5 is selected, implying a lot of batches and updates at each epoch.

The number of training epochs is the most important parameter, as it has the most impact on the final performance of the NN. Too few epochs will lead to underfitting, i.e. the NN has not learnt to recognize targets correctly, the error in the training set is high, in other words the training was ineffective.

Too many training epochs will lead the NN to overfitting, which means the NN recognizes correctly all the samples in the training set but fails to generalize its behavior when fed with different samples.

In this case, the number of training samples is small, which means that the NN must be fed several times to train it effectively (Pietro Zanuttigh, 2021).

Moreover, in this case, overfitting is not as much of an issue as underfitting because the samples in the simulated environment are very close to those fed to the NN during training. The best results were found with an epoch value of 30.

The loss function of choice is the Binary Cross Entropy loss (George Cybenko, 1999).
Let us define the true label distribution as p_i, whereas q_i is the distribution of the predicted labels.
Binary Cross Entropy it's defined as a regression logistic type function loss. Regression means the output is not just a label but the probability which indicates the belonging to a specific class. Logistics means the output probability of the sample being in one of the two classes is modeled after the logistic function:

$$q_{y=1} = \hat{y} = \frac{1}{1 + \exp(-\boldsymbol{w} \cdot \boldsymbol{x})}$$

Where is explicated the probability of obtaining the output $y = 1$. $\boldsymbol{w}$ is a vector containing weights of the NN, which is updated at each step, and $\boldsymbol{x}$ represents an input sample.
In the same way, the probability of obtaining $y = 0$ as output is simply:

$$q_{y=0} = 1 - \hat{y}$$

From the above explained notation can be noticed that distributions are limited in:

$$p \in \{y, 1 - y\}$$

$$q \in \{\hat{y}, 1 - \hat{y}\}$$

Now it is possible to assemble the Binary Cross Entropy Loss function, which is a measure of the difference between the probability distributions of the true labels and the predicted ones:

$$H(p, q) = -\sum_i p_i \log q_i = -y \log \hat{y} - (1 - y) \log (1 - \hat{y})$$

In this case i represents all the possible outcomes so that the whole probabilities can be mathematically described.
If we take as an example a single sample image from the training set, the idea is to compute the Binary Cross Entropy in all the pixels contained in the image to evaluate the difference between the NN evaluation and the real label. In this way, each pixel is assigned a predicted label, either belonging to the target or not belonging to the target.

An optimizer is an algorithm that aims to change the values of the NN weights according to the calculated total loss. In this case the optimizer of choice is the Adam optimizer (Diederik P. Kingma, 2015). Stochastic optimization methods are wieldy used with success in machine learning, but also in other scientific fields with great results. Stochastic means the objective function to be optimized, in this case the NN, consists of a sum of subfunctions evaluated on some subsamples of the data. Adam's goal is to make existing state-of-the-art stochastic methods, such as AdaGrad (Adaptive Gradient Algorithm) and RMSProp (Root Mean Square

Propagation) more efficient.

Adam adapts its learning rate based on the average of the first and second moments of the gradient at each step, making it simple to implement as requires very little parameter tuning.

Results shows that Adam works well in practice and has quickly become one of the currently recommended algorithms for training of deep NNs.

3.2.4 Training results

During the training process the metrics visualized were "accuracy" and the loss function value at each step, with which was possible to carry out parameter tuning.

Accuracy in machine learning applications is simply defined as:

$$accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

Computed across all pixel labels and on all the data samples.

Results in terms of accuracy and loss function value during the NN training are shown in figure 3.4.

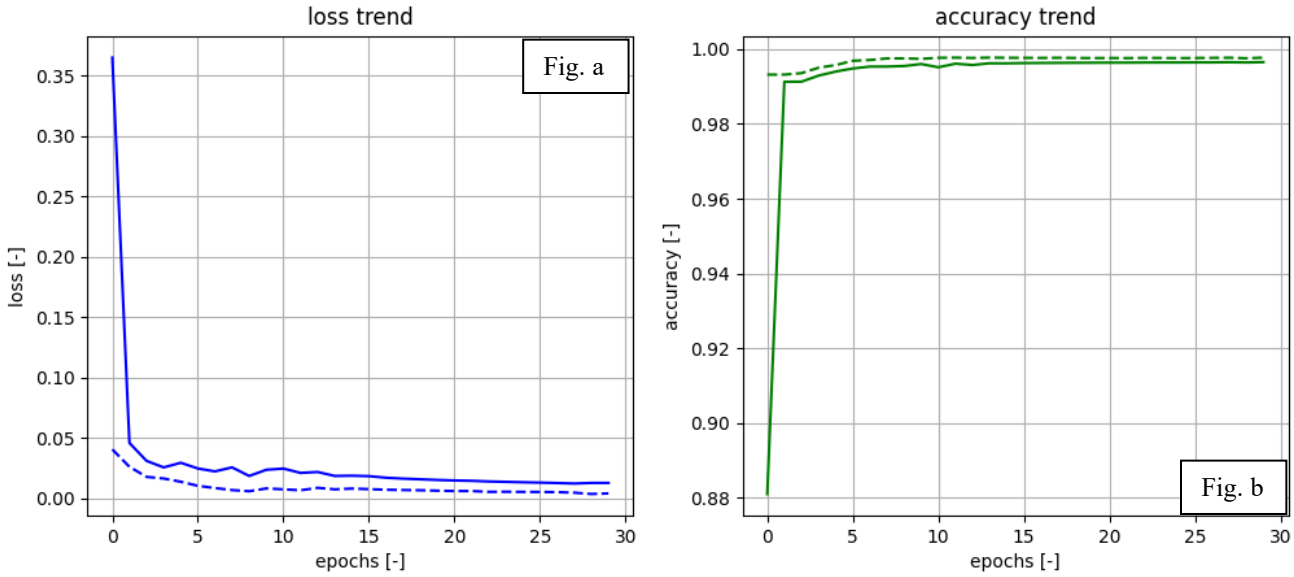


Figure 3.4: loss and accuracy during training of the U-Net with respect to epochs.
figure a: loss trend on training and validation with respect to epochs.
figure b: accuracy trend on training and validation with respect to epochs.
(—): loss computed on the training set
(--): loss computed on the validation set
(—): accuracy computed on the training set
(--): accuracy computed on the validation set

As shown in figure 3.4a, the loss value computed on both the validation and training sets decreases during training. Up to epoch 15 the loss plot, figure 3.4a, do not show a uniform decreasing trend, but after epoch 15 the trend is always decreasing. A low value of the loss function means that the distributions of the predicted labels and the true labels are close to each other, which is the desired result. In figure 3.4b, the accuracy trend computed on the training and evaluation sets is always increasing uniformly until epoch 20, after which a plateau is reached.
In conclusion, the training process is stable, at the end of which an optimal result is reached in terms of both accuracy and loss function.

3.3 Testing and results

This section presents the results of the U-net testing phase, with some notes on critical cases.

3.3.1 Testing

As mentioned in the previous chapters, there is no real test set of images available, so the testing phase can be carried out during the online simulation using key control Python modules to move the UAV in the physical environment.
Using the python module *drone_ml_main/scripts/test_segmentation_model.py,* the camera's output, available subscribing to */depth_camera_frontal/rgb/image_raw*, gets processed by the now trained U-Net in *TestSegm.cam_callback* and published in the topic *prediction_img*.
It is now possible to read and display images from the two topics side by side with Rviz to observe the image seen by the camera and the mask generated by the U-Net in real time.
A diagram of this process is shown in figure 3.5.

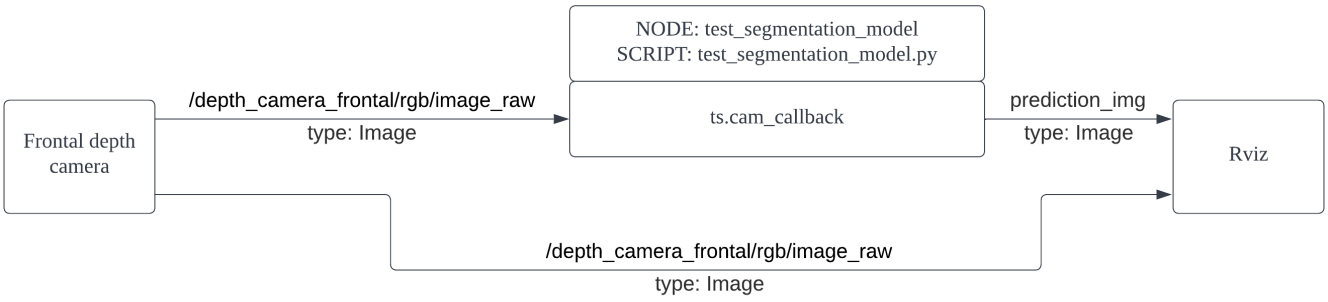


Figure 3.5: implementation of U-Net testing.

3.3.2 Results

Figure 3.6 shows images and masks generated by the U-Net. The results show that in these cases the neural network can correctly identify any target in the environment.

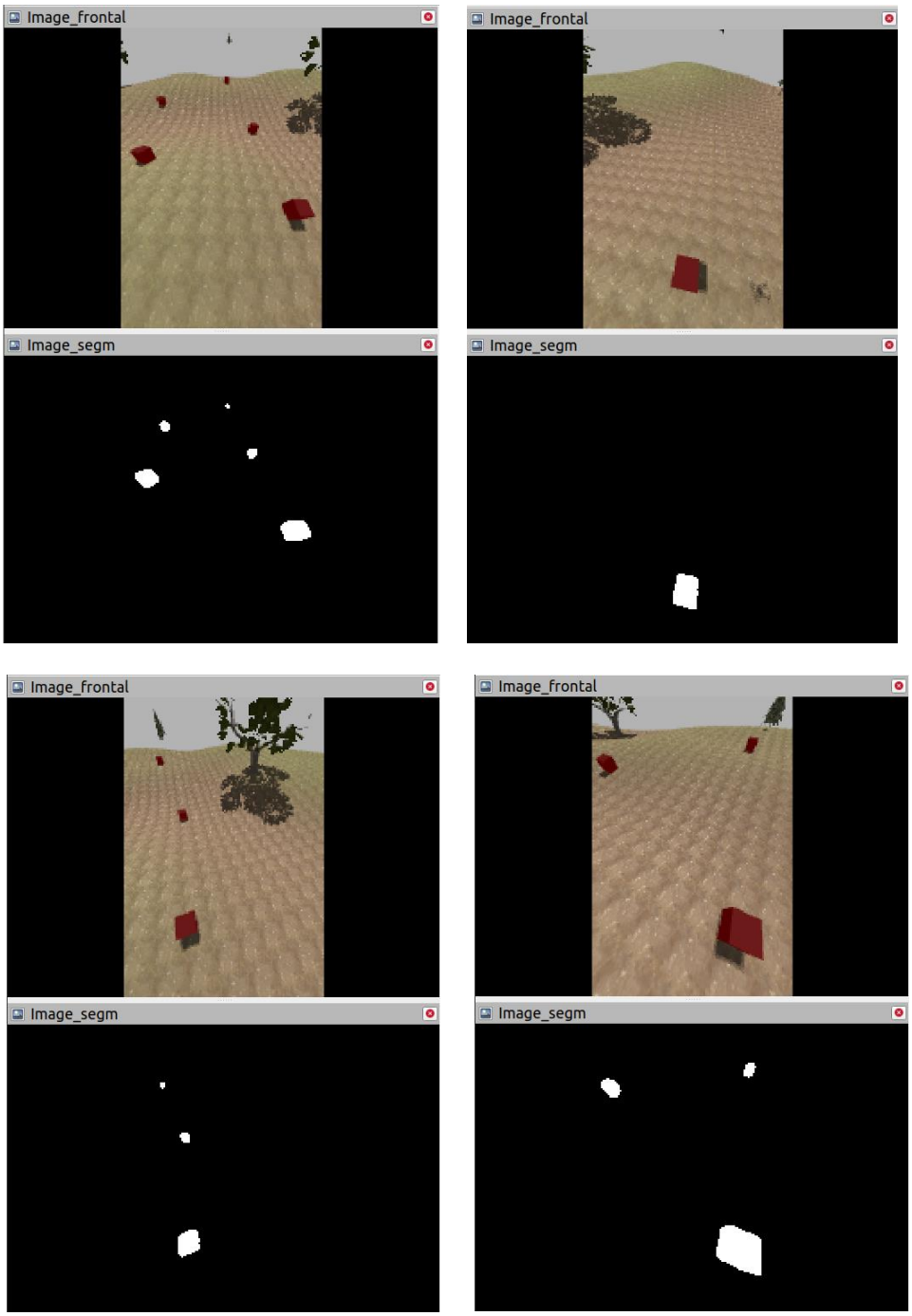


Figure 3.6: Example of results of the U-Net predictions. On the top of each image is depicted the input of the camera, while at the bottom there is the output of the trained U-Net.

It is possible to observe the accuracy with which the net is able to track targets. The closer the target is to the UAV, the better the NN can correctly predict its shape; if the target is far away, the NN is able to identify it, but in most cases is not able to reconstruct its shape. There are several reasons for this inaccuracy, including low image resolution due to the hardware limitations of running the simulation and python modules in real time, environmental disturbances, such as the colour gradient of the terrain, and the shadow of the target, which can be mistaken for the target itself.
As explained in more detail in chapter 6.4.1, shape imprecision in distant targets will have a bad effect on the classification algorithm.

Figure 3.7a shows an example of critical conditions that can cause problems later in the project. There are two targets close together, and from the UAV's perspective they are also superimposed. In this case the U-net correctly detects the targets but cannot distinguish between them and interprets them as a single target. Unfortunately, this is not a failure of the U-net's detection capabilities, but rather an environmental problem. The same case is seen from a slightly different perspective in figure 3.7b, in this case there is no superposition of any kind, so the problem does not appear.

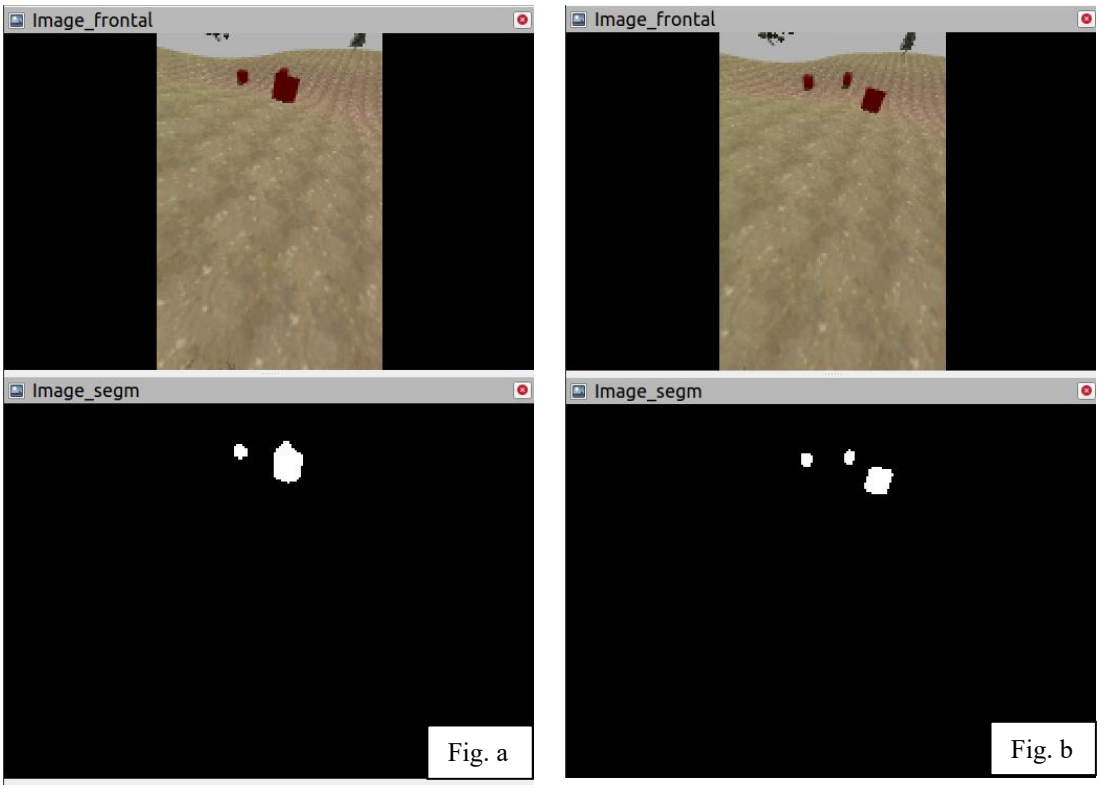


Figure 3.7: images as output of the frontal camera and after the U-Net processing.
figure a: on the left side two targets superimposed.
figure b: the same two targets but seen from a different angulation.

4 Allocation algorithm

This chapter aims at describing the project's pipeline, starting from the mask generation, explained in the previous chapter, and moving on to the allocation algorithm. The algorithm is designed to analyse U-Net outputs and generates a message which contains targets positions in the local frame of reference.

4.1 Topological mask analysis

After the U-Net is complete, the next step shown in figure 1.1, is to process the image masks and compute targets centroids. The aim of this section is to explain the process that occurs after the mask generation and leads to the determination of the target's centroids.

4.1.1 Concept

The basic idea is to exploit image masks generated by the U-Net, apply the algorithm described in (Satoshi Suzuki, 1985), already implemented in OpenCV (OpenCV homepage, s.d.) python package, thus obtaining both the number of targets seen by the frontal camera and centroids for each one of them.

The algorithm used performs a border following topological analysis on binary images, that consist of pixels than can have a value of either (1-component) or (0-component). Where (1-component) and (0-component) are two different values. The aim is to convert the binary picture into a border representation of it. Also, this algorithm has the advantage to enumerate borders when it finds them in the binary image, this feature is crucial as it is needed to count number of target and thus distinguish them.
For the algorithm explanation two definitions are important:

- Outer border point: in the same row the *(j-1)* pixel is (0-component) and the *j* pixel is (1-component).
- Hole border point: in the same row the *j* pixel is (1-component) and the *(j+1)* pixel is (0-component).

The border following algorithm raster scans all the image, when an outer border point or a hole border point is detected, the raster stops. If the pixel satisfies both the above conditions, then is regarded as a border starting point condition. If this is the case, then a unique number is assigned to the newly detected border and the border is followed marking every pixel in it. When every border pixel is marked, and the border is complete, the raster scan is resumed to find other borders in the image.
Now, it is possible to perform topological analysis and compute centroids of borders with OpenCV, which gives as output enumerated centroids.

4.1.2 Implementation

The implementation of the mask analysis algorithm is closely related to the target allocation task and the control methods used to follow the targets. For this reason, it is better to first show the entire implementation flowchart in figure 4.1, which shows the code architecture used during the final simulation, and then explain it step by step in the next chapters as new solutions are implemented.

As for the actual mask analysis algorithm implementation, with reference to the flowchart in figure 4.1, ROS subscribers are initialized in *updateBGRCam()* that subscribe to */depth_camera_frontal/rgb/image_raw*, which returns as output the frontal image, and *depth_camera_frontal/depth/points*. This topic then gives as output the PointCloud message of the frontal depth camera. Every time is received a format Image message *bgr_points_frontal_callback* is called. In *bgr_points_frontal_callback* the U-Net trained model is used to generate image masks and through OpenCV the contour algorithm is applied. After that, the *pixel_coord* message is produced, which is a 2D array with rows $i \in [0, 4]$ representing the target number and columns $j \in [0, 1]$ indicates the pixel coordinates of the centroid of the target.
If the camera detects less than five targets, then *pixel_coord* free rows are filled with $[-1, -1]$.
pixel_coord is then used in other methods for assigning targets and in the PID control methods.

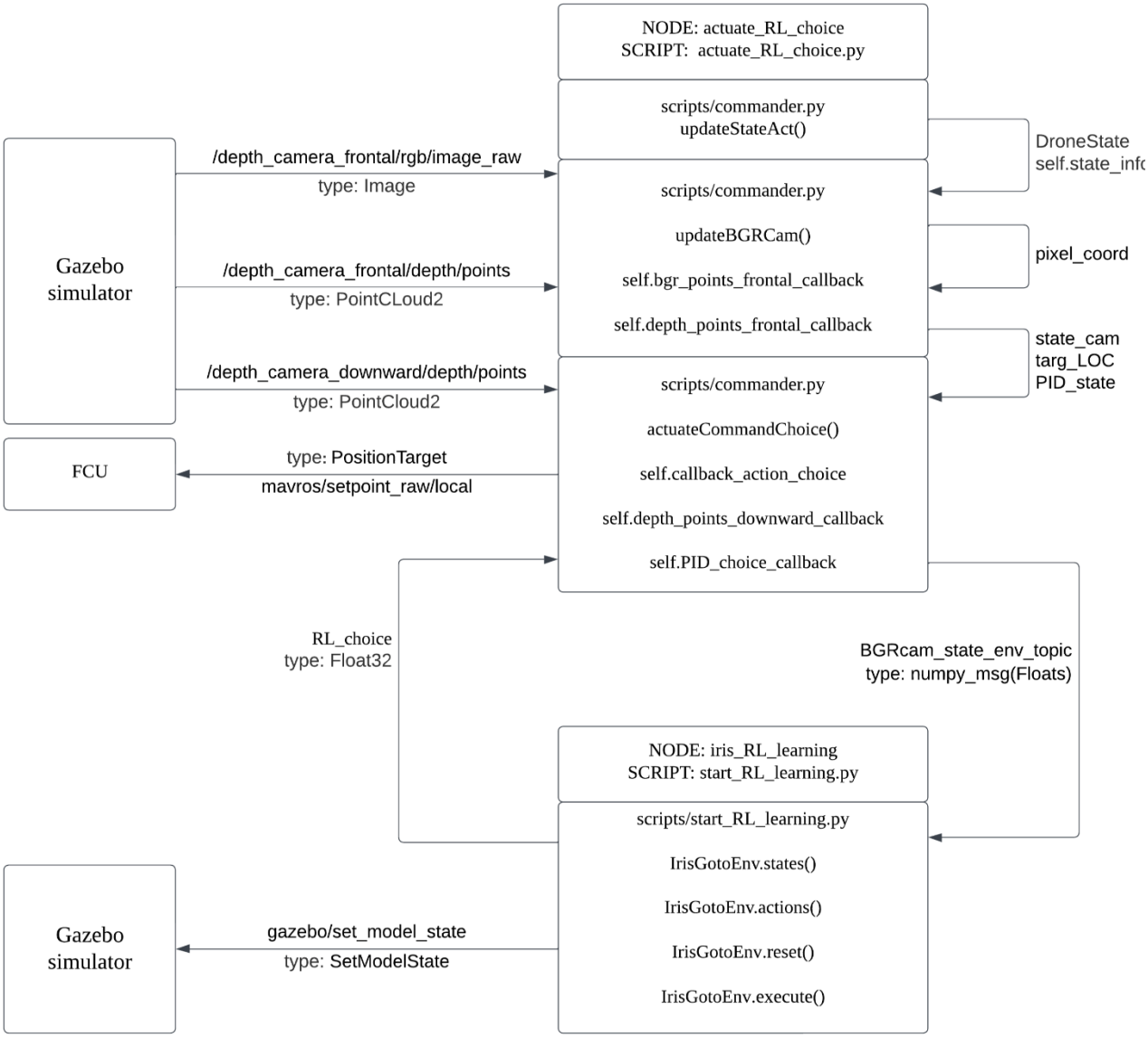


Figure 4.1; implementation of the final project as python modules and messages send and received.

4.1.3 Testing

For the testing phase, slight variations in the code are implemented so that it is possible to publish the processed and contoured image to Rviz. The results are depicted in figure 4.2, which shows examples of the original image and the contours obtained after topological analysis. Figures 4.2a and 4.2b show cases with multiple targets in visual contact, which the algorithm is able to recognise and find accurate contours. In figure 4.2c the UAV is close to a single target where contours are detected and drawn even in the presence of shadows cast by the same target. Figure 4.2d shows a close target half seen by the camera, the U-Net is able to recognize it anyway, since in the training set where implemented some similar samples, and contours are found. In this case, the calculated centroid will not be physically accurate, since it is based on the extracted

half contour, but this behaviour does not cause any issues in the final implementation of the guidance algorithm.

At the end of this phase, the message *pixel_coord* is correctly generated, enumerating the targets.

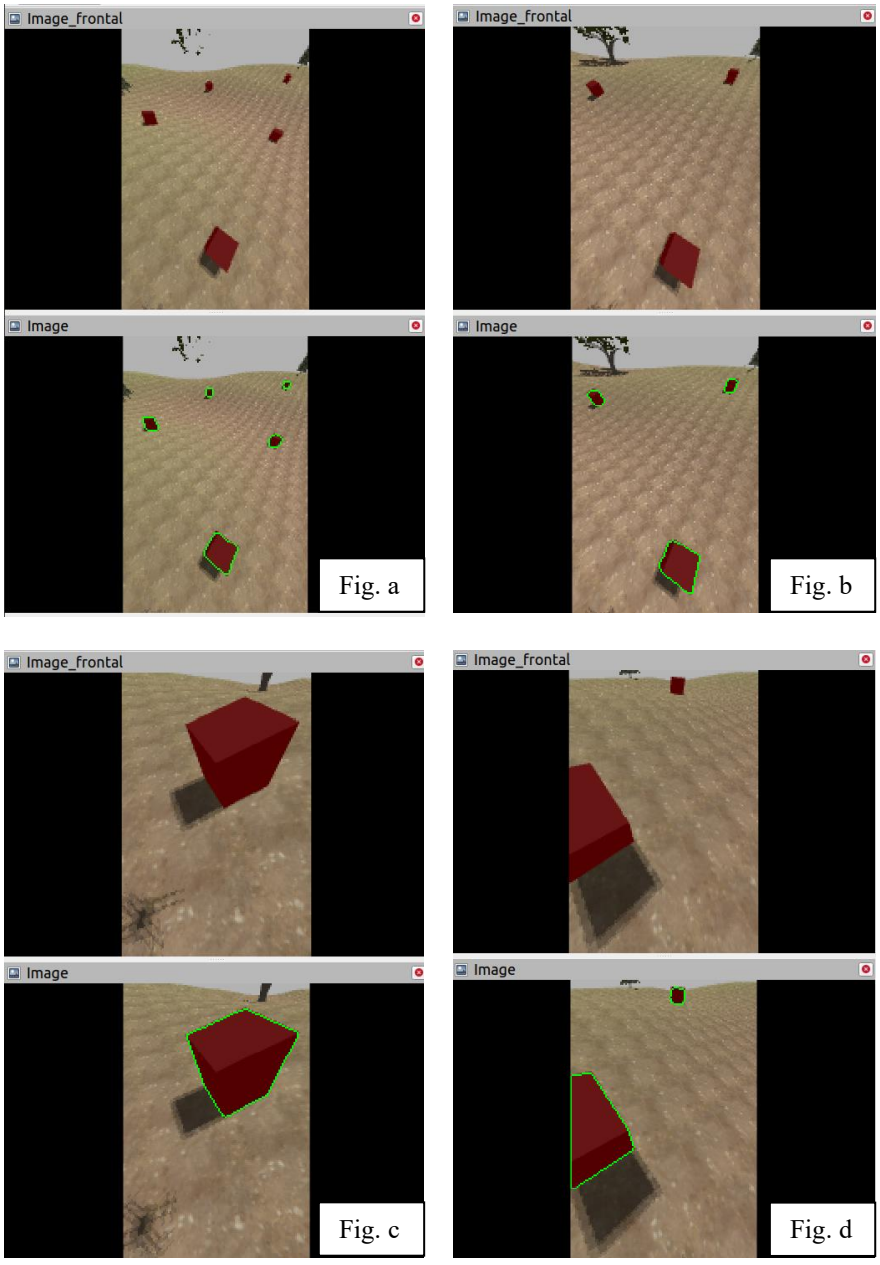


Figure 4.2: On the top of each image the output of the frontal camera, on the bottom the contoured image.
figure a: top left, multiple targets.
figure b: top right, multiple targets.
figure c: bottom left, single close target.
figure d: bottom right, single target half seen.

4.2 Target allocation algorithm

This section focuses on explaining the allocation algorithm developed to store the local position of targets, update their positions when are visible to the frontal camera, and simply store them when targets are not in line of sight.

4.2.1 Concept and operation

The first step is to take as output the targets centroids stored in *pixel_coord* and use the frontal depth camera point cloud message to find their positions in the relative camera frame. To obtain the target positions in the local frame, it is necessary to rotate and translate the data in relative coordinates, according to the frontal camera attitude and position. This is possible because the UAV position and attitude are extracted from the FCU after the Kalman filter, while the relative position and attitude of the front camera with respect to the UAV is known as defined in section 2.2.2.
Now the idea is to update a new message, *targ_LOC*, which memorizes the five targets in rows $i \in [0, 4]$ and contains target x and y local positions in columns $j \in [0, 1]$.
The idea is to let *targ_LOC* to be a static message, that is updated only when targets are detected and their *(x, y)* values are close enough to one of the *(x, y)* values already contained in *targ_LOC* in the previous step. Note that targets enumeration between messages *pixel_coord* and *targ_LOC* is different because of how the updating process works.
Although this idea theoretically could work in the simulated environment some issues arise:

- UAV dynamics: when a command is given, the UAV attitude changes from the neutral stable state to a new configuration so that the linear velocity command can be followed.
  The dynamic of the UAV influences heavily how accurate targets positions, obtained from the previously described pipeline, are.
- Distant targets: targets far away from the UAV are difficult for the U-Net to detect and their estimated position may be inaccurate as it depends on the shape of the target.

Tests in the simulated environment have shown that these problems can cause incorrect updates to *targ_LOC*. For example, new positions on the message may be updated with incorrect values, effectively creating a virtual target that does not exist in the simulated environment.
For these reasons here is introduced a *buffer_matrix* for the updating of *targ_LOC.* The *buffer_matrix's* task is to filter out possible uncertainties on the output targets localization so that *targ_LOC* is always updated with secure targets positional values.

The basic idea is having the *buffer_matrix* be a 3D array, which stores target numbers in rows $i \in [0, 4]$, memorizes *(x, y)* local target coordinates in columns $j \in [0, 1]$, and stores successive target locations taken directly after the target localization, which is the noisy signal that must be cleaned, in the last index $k \in [0, 9]$.

When targets are in view of the frontal camera a *buffer_matrix_slice* is generated, initialized with all rows *i* as [-1, -1]. The aim is assigning *(x, y)* positions of the targets acquired to the *buffer_matrix_slice* row *i.* The update logic is computing the difference between the acquired *(x, y)* and target positions contained in the last slice of the *buffer_matrix,* take the index which corresponds to the minimum difference.

- If the difference is under a certain tolerance, then allocate *(x, y)* to that index *buffer_matrix_slice* row.
- If the difference is over the tolerance, then allocate *(x, y)* values to a free *buffer_matrix_slice* row, containing [-1, -1].

In case less than five targets are detected at the same time, which is almost always the case, then missing *(x, y)* positions in *buffer_matrix_slice* are let with [-1, -1].

Then the newly sorted *buffer_matrix_slice* is appended at the end of the *buffer_matrix,* and the first position *k=0* slice of the *buffer_matrix* is delated.

In other words, the aim is to concatenate *(x, y)* values using matrix *k* row, so that *buffer_matrix* rows are filled just with values that should be similar to one another. Moreover *buffer_matrix* gets updated without changing its dimensions.

In the next step *buffer_matrix* is checked at each updating cycle, if one or more rows *k* are completely full, without any [-1, -1], then the average of the row *k,* with respect to the *j* positions, are computed. This result should be a secure value of target position (x_m, y_m) of the real target.

Now *targ_LOC* must be updated with new values obtained (x_m, y_m). *targ_LOC* message is initialized with all lines *i* with values [0, 0].

- If *targ_LOC* is already full of target positions, there are no [0, 0] in the array, then the index of the line in which the difference between (x_m, y_m) and target positions contained in *targ_LOC* is minimum is computed. In the next step the line of *targ_LOC* that corresponds to the index found is updated with (x_m, y_m).
- If *targ_LOC* is empty or partially full, at least a row containing [0, 0] is present, then two cases are possible.
    - If the difference between (x_m, y_m) and target positions already contained is under a tolerance, then the update works the same way as before.
    - If the difference is over the tolerance, then (x_m, y_m) is allocated to a new empty row.

This way is possible to allocate to *targ_LOC* only target positions which passes through the *buffer_matrix*, so that only sure values are used for the updating. Moreover, if a target is not in sight the row containing its

relative position is not updated but its *(x, y)* values remains static. This allows to both memorize filtered target positions and updating them when they are in camera sight.

4.2.2 Implementation

The purpose of this section is to explain the implementation of the code architecture in the final project flowchart already presented in figure 4.1.
When *updateBGRam()* is called, *depth_points_frontal_callback* is iterated, this happens every time the subscriber receives a pointcloud message in */depth_camera_frontal/depth/points* topic.
depth_points_frontal_callback takes as input the pointcloud message and *pixel_coord* array, previously generated in *bgr_points_frontal_callback*. The function implements the previously described allocation algorithm which, starting from *pixel_coord* and the frontal pointcloud produces as final output *targ_LOC*. Other two important messages, *state_cam* and *PID_state*, better explained in chapter 5.1.1, are generated during the iterations of *depth_points_frontal_callback*.

4.2.3 Parameters

Parameters necessary to run the target allocation algorithm are shown in table 4.1.

Parameter	Value	Notes
buff_toll	1.7 m	Tolerance with which gets sorted *buffer_matrix_slice*
targ_LOC_toll	2 m	Tolerance with which values are allocated to *targ_LOC*
buffer_matrix_lenght	15	Length, number of slices, in buffer matrix

Table 4.1: target allocation algorithm parameters.

The length of the *buffer_matrix* is fundamental parameter for the allocation algorithm. The parameter is closely linked to the update rate of the frontal camera, which is 10 Hz according to table 2.1. Since each time is reached a message in */depth_camera_frontal/depth/points* topic *buffer_matrix* receives a single update, this means every 1.5 s of simulation *buffer_matrix* is completely updated from the first to the last slice. If a new target enters the visual range of the frontal depth camera, then the algorithm has at least 1.5 s delay before actually updating *targ_LOC* message. The higher *buffer_matrix_length* is, the slower the target allocation will be, but also if *buffer_matrix_length* is high, a better estimation of the targets is given, and less possibilities of identifying wrong target there is.
A value of 15 is found to provide a good speed of allocation of true target's positions to *targ_LOC* without compromising its precision.

buff_toll is the tolerance value used to sort out *(x, y)* in *buffer_matrix_slice*, while *targ_LOC_toll* is the tolerance used for the allocation of mean values, output of the *buffer_matrix*, to *targ_LOC* message, according to the procedure explained in the previous chapter.
Below is a list of notes about these two parameters and how they interact is reported:

- Minimum target distance: *targ_LOC_toll* dictates the minimum distance, in x and y, at which the algorithm is able to allocate targets correctly. In this case if targets are closer than 2 m in x or y or both coordinates, the allocation algorithm can confuse the two of them and could start to update their position mixing up the two targets.
- *buff_toll* and the allocation speed: it has been tested that the higher *buff_toll* is the faster targets positions are given as output, but also the more imprecise they are, a compromise value which assures good results, taking into account this relation is found to be 1.7 m.

4.2.4 Examples

Figure 4.3 shows a snapshot of the *buffer_matrix* and the *targ_LOC* message 2 seconds after the start of the simulation. This is the ideal case where no noise is introduced into the position signals. The *buffer_matrix* in this case is ordered, for representative purposes, so that values of x and y are aligned.
When a row is completely full of similar values*, targ_LOC* is updated. Note that in this case each target position is updated at the same time, except for the last target which is not even allocated in targ_LOC, this is because the frontal camera has yet to see it.
Figure 4.4 shows a *buffer_matrix* snapshot in the same conditions as the previous case, but this time noise is introduced to the UAV position signals, making it more difficult for the algorithm to locate targets.
In this case it can be seen that '-1.00' values are present in each row, meaning that no values are updated in *targ_LOC* at the time of the snapshot as no similar values are detected for a long enough period of time.
targ_LOC, in this last case, contains two target coordinates because they were previously assigned with at least two full lines per target.

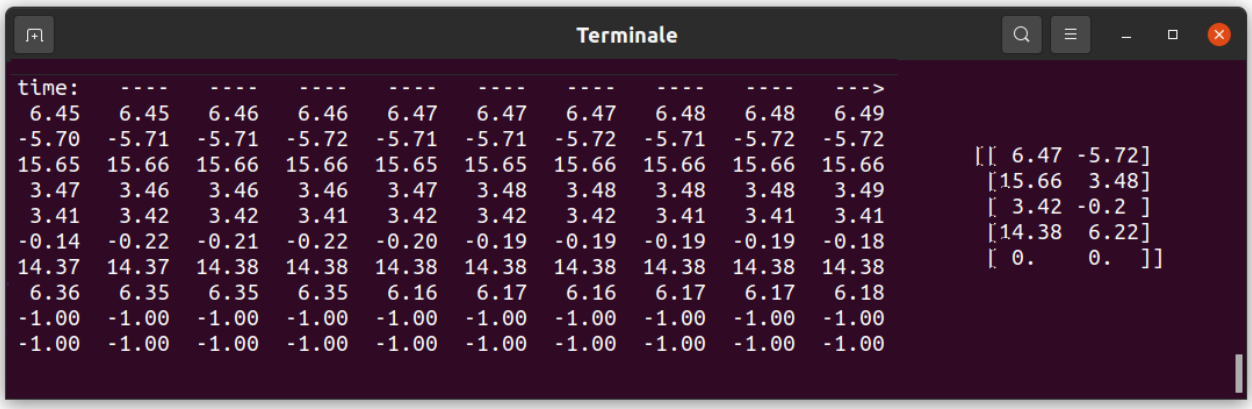


Figure 4.3: buffer_matrix working example. Every target is updated except the last one which is not seen by the frontal camera.

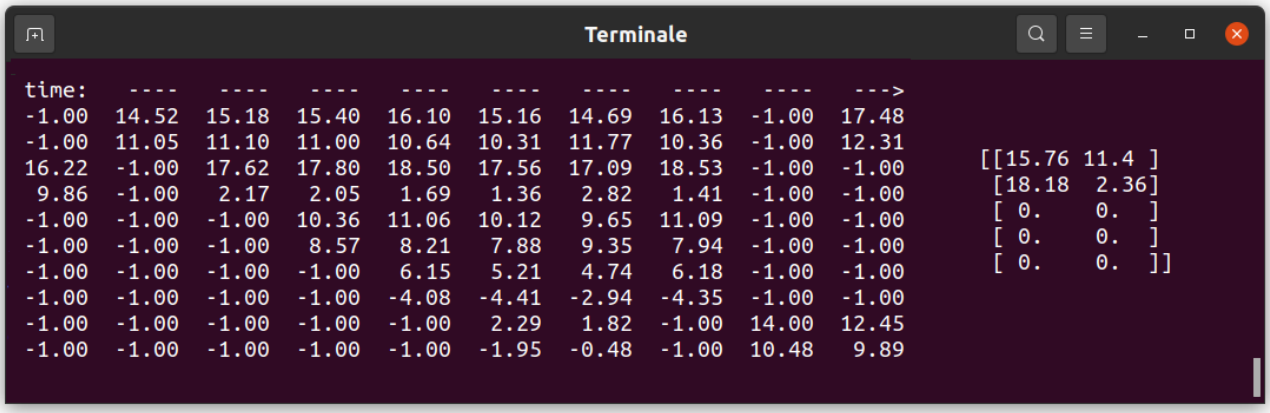


Figure 4.4: buffer_matrix working example with noise. None of the target is being updated, in each line there is at least on occurrence of [-1.00].

As the allocation algorithm is a fundamental part of the project, its results and performance are discussed separately in chapter 6.

5 Control method

In this chapter the concept and general chart of the control method implemented, which allows to reach targets, are presented. Then the targets sequential policies implemented, which are Reinforcement Learning and a deterministic one, are shown.

5.1 PID control

This section focuses on the PID control method used. As for the x, y control method, it takes into account the guidance obtained by the U-net and the subsequent pipeline, which gives as output the localisation of the targets in the local frame, as explained in chapter 4.
This chapter also explains the implemented z-axis control method.

5.1.1 Messages definition

Firstly, messages generated in the allocation algorithm loops are explained since are fundamental for the proper functioning of the control method.
PID_state is a 2D array generated using data in output by the mask analysis, it's a message which contains relative target positions as seen by the camera without any operation of filtering applied. More specifically positions in columns j of *PID_state* contains [y_cam_coord, x_cam_coord, total_distance] with $j \in [0, 2]$, whereas rows $i \in [0, 4]$ are for targets enumeration. Note data collected in *PID_state* is in the camera's frame of reference explained in 2.1.4. If the frontal camera identifies less than five targets then the *PID_state* array is shorter, has less rows, this means it changes its dimensions at each iteration.
Another important message for the control method implemented, which is not reported in the control chart below for simplicity, is *store_index* 1D array, which aim is to map targets positions from *targ_LOC* to *PID_state*. This way targets stored in *targ_LOC* and targets seen in *PID_state* by the frontal camera can be correctly associated. *store_index* is generated by computing the difference between *targ_LOC* and targets positions directly seen by the frontal camera, the same data which composes *PID_state*. Then the index, which corresponds the row of the minimum difference in both *(x, y),* is found. Finally, this index is associated to the current iterated *store_index* position. Free positions are filled with [-1] and *store_index* is obtained.
state_cam is a 2D array, with rows $i \in [0, 5]$ and $j \in [0, 2]$, used to communicate with the environment. In rows $i \in [0, 4]$ and $j \in [0, 2]$, the message contains the *targ_LOC*, whereas row $i = 5$ contains the UAV position. If is encountered a target position still not filled of *targ_LOC* then also *state_cam* is filled with [0,

0]. Also, in *state_cam* message target reached are flagged substituting the (x, y) target coordinates with [100, 100], this feature is better explained in chapter 5.2.2.
In summary four important messages will be used discussing this section:

- *Targ_LOC:* stores the targets positions in local frame of reference, represents the memory of targets positions the algorithm has.
- *PID_state:* stores targets position in the relative frame of the camera, represents what is seen by the frontal camera instantaneously.
- *store_index:* Keeps track of targets enumeration between *targ_LOC* and *PID_state*.
- *state_cam:* stores target locations in the local frame and is send to the RL environment. Also stores target's flags.

5.1.2 Concept and operation

A specific control method flowchart is shown in figure 5.1 in a "Simulink" style, where signals and gains are depicted. When a command is given, the *mavros/setpoint_raw/local* frame of reference, already described in chapter 2.1.4, is used. Colours in the chart indicates which control lines sends commands to which axis, blue for controls for the z axis, red for x axis, green for y axis and purple for yaw rates commands. Both feedback control lines are here regarded as outer loop, since they don't communicate directly with actuators, electric motors, but with PX4 autopilot, which then performs the inner control loops.

The discussion starts from the simplest control implemented, the one for the UAV's z axis, the blue lines, which is always active during the mission. The basic idea, as stated in the introductory chapter, is gather the pointcloud message from the downward camera, apply an average of the z coordinates of the points seen to get an average altitude at which the UAV is flying. Then compute the difference between a guidance value, the one desired which remains constant, and the state estimated one to apply a PID controller to the difference signal. The output is the velocity in z axis the autopilot must follow. Also, a derivative gain is applied directly to the linear velocity in z axis extracted from the FCU. Tests highlighted a more stable and smooth behaviour in presence of this value.
Note that to correctly compute the altitude estimation from the downward camera the UAV attitude must be taken into consideration. Thus, the relation below is utilized, where $(\phi,\ \theta)$ are respectively roll and pitch angles and *height*, *width* are number of data points given as output of the camera in the two directions. *Pointcloud(z)* is a 1D array representation containing all data points, in z camera frame axis, of the pointcloud message.

$$h\_avg\_eval = \frac{\sum_{i=0}^{height\ width} Pointcloud(z)_i * \cos(\phi) * \cos(\theta)}{height\ width}$$

For the x and y axis control method, firstly one of the targets memorized in *targ_LOC* is chosen through the execution of a policy later explained in sections 5.2, 5.3, thus *targ_LOC(n)*, relative to a single target, is obtained. Now two different control methods are applied depending on the case:

If the target is located in *targ_LOC* but not detected by the frontal camera, the idea is to make the UAV turn in the target's direction using *targ_LOC* information, and start moving in the target's general position so that it can be eventually detected by the frontal camera.

First, through trigonometric functions, is possible to relate the UAV's current position, obtained from the *DroneState* message, to the target's stored one, so the *yaw_des*, the guidance value, is obtained. A PID control is applied to the difference signal between *yaw*, which is the actual yaw angle of the UAV, and *yaw_des*. The result is the *yaw_rate* the autopilot has to follow. If the UAV manages to turn itself in the target's direction, within *yaw_lim* of difference error in yaw, and still the frontal camera does not detect it, then the velocity in the x axis is set to 2 m/s. The objective is searching for the wanted target.

If the target is detected by the frontal camera, the UAV switches control method and passes to a complete visual guidance. This is the case in which the target is allocated in *targ_LOC* and also detected by the frontal camera.

First the message *store_index* is used to map to *PID_state* the selected target in *targ_LOC*. For the x axis control the reference guidance value, *camera_x_targ,* is constant and depends on the distance, in the reference frame of the camera y axis, at which the UAV is hovering on top of the target. This value is equal for all the targets so is treated as a constant. The difference between *camera_x_targ* and *PID_state(x)*, relative position of the target in y axis camera's frame, is then used to implement a PID proportional value. The result is a command in the x axis.

Note that "/dist" exists because the command is divided by the total target distance contained in the third column of *PID_state.* In the case the target is detected by the frontal camera, then this control in the x axis is always active.

For yaw and y axis control, in the case in which the frontal camera is detecting the target, the best solution found is applying similar command patters a human would do if piloting the UAV manual. The idea is if the target is far then probably the best commands to give is trying to centre the target with yaw while proceeding towards it. Whereas if the target is close then move it with in x and y axis without manoeuvring in yaw.

The x axis commands are already implemented as explained in the previous paragraph, and remains always the same, while to apply y axis command and yaw first must be computed the total distance from the UAV to the target using *PID_state* message.

If the target is near, the distance is under the *cct* threshold, then a PID proportional command is applied to the signal *PID_state(n, y)* which carries the target position in the x axis camera frame. Since 0 is the guidance in this case: $p\_ty\_near\ (\ 0 - PID\_state(n, y)) = \ - p\_ty\_near\ PID\_state(n, y)$ so a "-" is applied. The

output is a strafe command in the y axis.

If the target is far, the distance is over the *cct* threshold, the same *PID_state(n, y)* signal is used and like before, since 0 is the guidance also in the yaw case, a "-" is applied. The output is a command in terms of yaw rate.

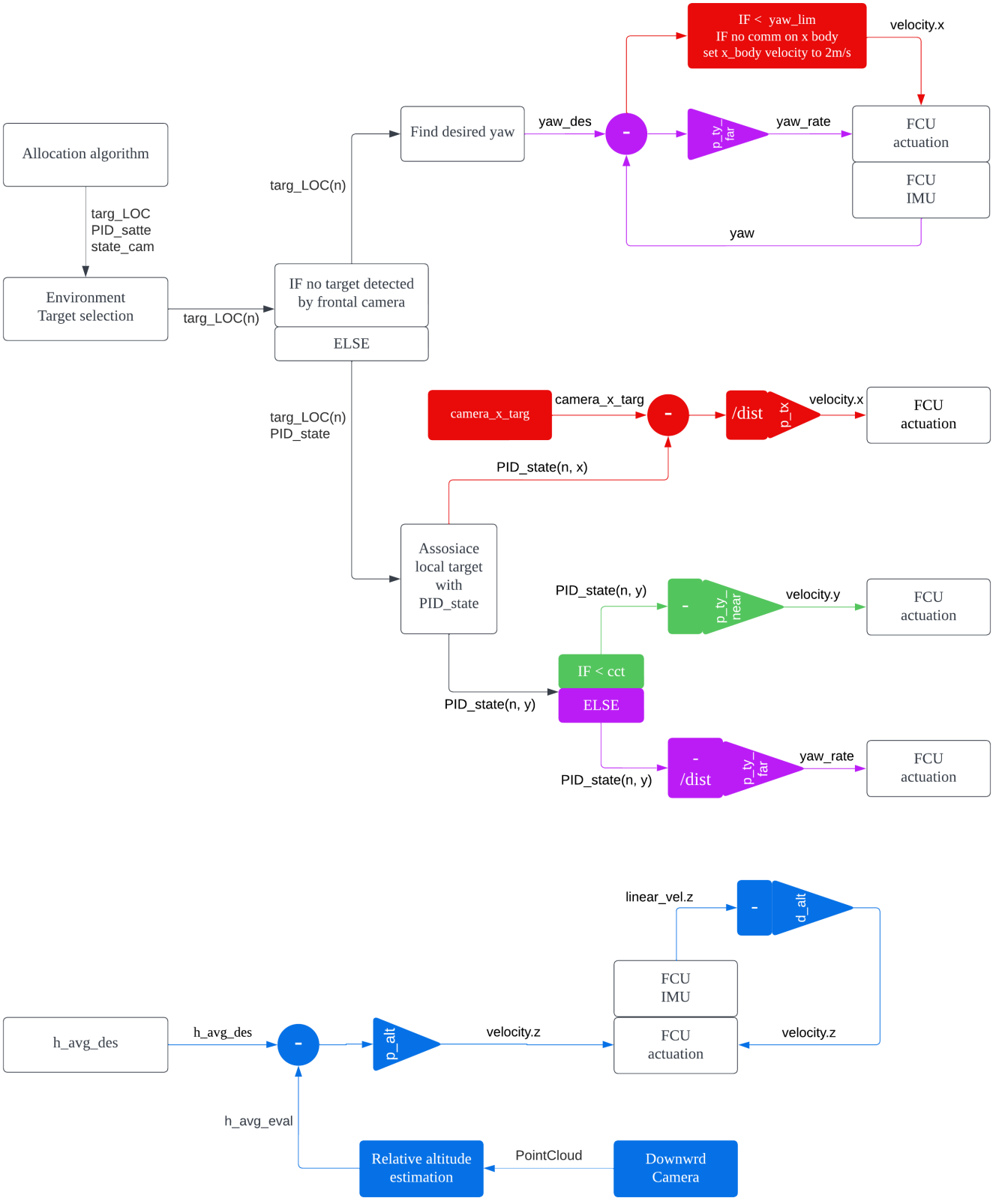


Figure 5.1: Control method flowchart.

5.1.3 Implementation and parameters

With reference to figure 4.1 now the implementation of controls in terms of code and python methods is explained with reference to figure 4.1.

As explained previously *state_cam, targ_LOC, PID_state* and *store_index* are generated during the *updateBGRCam()* method cycle. These messages are used by the *actuate_RL_choice()* method. Firstly *callback_action_choice* is called each time a new message is received in *action_choice_topic*. The purpose of this callback is to interpret actions in terms of commands. In table 5.1 are listed all available actions and what they do. In particular, for $actions \in [0, 4]$ a different target is selected, thus obtaining the coordinates in local position to chase, that is the previously mentioned *targ_LOC(n).*

Actions	Notes
action 0	Stops the UAV, selects target number 1 to be reached
action 1	Stops the UAV, selects target number 2 to be reached
action 2	Stops the UAV, selects target number 3 to be reached
action 3	Stops the UAV, selects target number 4 to be reached
action 4	Stops the UAV, selects target number 5 to be reached
extra action 100	Stops the UAV, deactivates the (x, y) control logic, sets the UAV to point follow mode and sets coordinates to the starting point, resets the *targ_LOC* and *state* message with [0, 0] at every row, moves targets to new random positions within the workspace.
extra action 101	Turns the UAV in yaw so that can scan the aera for targets before starting the mission

Table 5.1: action table.

PID_choice_callback, iterated at 10 Hz, takes as input the desired *targ_LOC(n)* to follow and applies the PID logic explained in the previous section. Commands are sent to the FCU in the usual topic *mavros/setpoint_raw/local*.

The method *depth_points_downward_callback*, iterated at 3 Hz as mentioned in the camera models section, instead applies the z-control logic, writing commands in *mavros/setpoint_raw/local topic.*

Table 5.2 lists the parameters mentioned in the operation of the algorithm. The results and commands generated by the implemented controls are reported in the last chapter, as they are closely related to the results of the whole project.

Parameter	Value	Notes
cct	7 m	Distance to switch between close and far flight controls
yaw_lim	15 deg	Tolerance of yaw to decrete if the UAV has turnd toward the target
camera_x_targ	1.6 m	Parameter relative to the camera frame, serves as guidance for the x axis once the UAV is detected by the frontal camera
pos_toll_outer	0.07 m	Tolerance in x and y camera axis used to determine if the UAV is on top of the target by the control system
vel_toll_outer	0.07 m	Tolerance in linear speed. UAV's speed in x and y axis must be under it for the target to be determined as reached.
h_avg_des	3.5 m	Desired average altitude to maintain
p_alt	3 s^{-1}	Proportional for z axis control (*trag_LOC* guidance)
d_alt	0.7 [-]	Derivative for z axis control (*targ_LOC* guidance)
p_tx	2.2 [-]	Proportional for x axis control (visual guidance)
p_ty_far	35 s^{-1} or deg/s	Proportional for y axis control (visual guidance far)
p_ty_near	0.5 s^{-1}	Proportional for y axis control (visual guidance near)
actuatecomm_rate	10 Hz	Rate at which commands are sent to the FCU autopilot

Table 5.2: control system parameters.

5.2 Reinforcement Learning policy

After a brief introduction to what Reinforcement Learning is, the specific model and implementation used in the project is presented.

5.2.1 RL introduction

Reinforcement learning (RL) is the third paradigm of ML methods, along with supervised and unsupervised learning. Supervised learning utilizes a training set made up by samples along with labels or result known a priori, as in chapter 3 for the segmentation task, whereas in unsupervised learning the training set is composed just by samples without labels or is not provided at all. Most of unsupervised learning algorithms are concerned with classification in different groups, like clustering applications.
RL applications are fundamentally different from supervised or unsupervised learning. A RL flowchart is shown in figure 5.2.

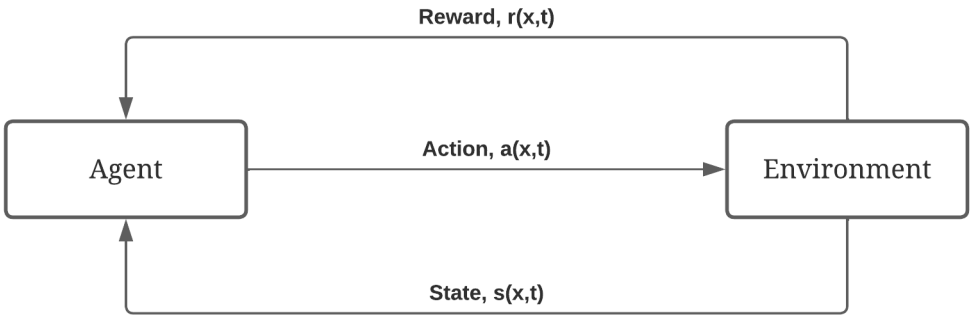


Figure 5.2: Reinforcement Learning flowchart.

RL is based on interactions between an agent, or actor, and an environment in a Markov Decision Process framework. The agent takes actions based on a policy, which is a probabilistic distribution which depends on the agent's current state.
The policy can be defined as $\pi(s,a) = Pr(a = a|s = s)$ which means, the probability that action a is taken knowing the state of the agent s. The action leads to a variation of the actor state in the environment. The environment chosen can be completely deterministic, as in the case of implementation of dynamical equations, or of probabilistic nature or a composition between the two. Based on the effects of the newly acquired state in relation to the environment a reward is assigned to the agent.

The optimization process objective is to find the best policy $\pi(s,a)$ that maximizes the number of long-term rewards acquired. Learning in RL framework is done by trial and error, a training set is no longer required, the policy is updated by trying different policies and updating it every time a better one is found based on subsequent simulations of the system, called episodes. The policy used during the training process can be always the best found at every episode or can have elements of randomness, to avoid local minima.
In most application the policy is described by a Deep Neural Network (DNN), which takes in input the actor state and outputs the action taken. During the optimization process weights of the net are updated so that a change in the policy is obtained (Barto, 2015).

5.2.2 RL model and concept

The main advantage in using a RL algorithm is that complex and nonlinear policies can be represented. In this case the objective is to find the best path between targets that minimizes the mission time at each episode. Now an explanation of components that build the implemented RL algorithm is given.

Each episode starts with the UAV at the starting point, at coordinates [0, 0, 2.5] m, and with a new configuration of targets in the physical environment. The goal of the mission is to reach all the targets without hovering over the same one two or more times; if this is achieved, the episode ends successfully. Each episode can also end without completing the mission if one or more of the other conditions listed in table 5.3 are met, see the last paragraph of this section for a more detailed explanation.

Each step starts when an action is taken.

One of the main components of any RL algorithm is the action space, which contains all the actions that the actor can take and that the policy can choose to actuate. In this case, the action space is defined as an array of five inputs [0, 1, 2, 3, 4], each action corresponding to one of the targets already detected and located in the *targ_LOC* message. Table 5.1 already lists all possible actions. A clarification should be made here: the action execution is part of the control system, whereas only the action space is part of the RL algorithm. The policy in this case is represented with a dense Neural Network that takes states as input and actions as outputs. The model is represented in figure 5.3. Note that the output of the Neural Network is a 64 neurons dense layer, this is done because Tensorforce (Tensorforce documentation, s.d.) automatically adds an additional layer with the same length as the action space, which in this case is 5.

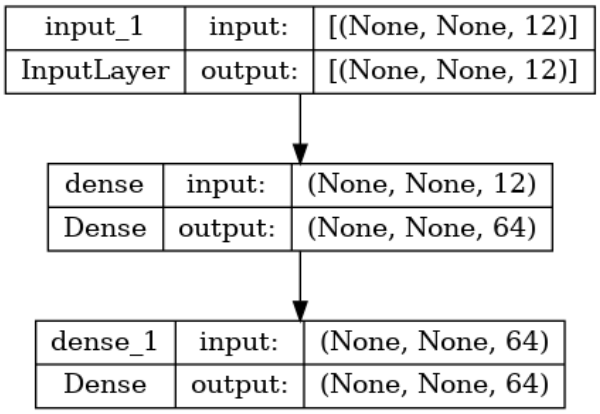


Figure 5.3: RL policy neural network architecture.

The state message is called *state* in the environment module and *state_cam* in *commander.py* module. Rows are $i \in [0, 5]$, indices $i \in [0, 4]$ contains *targ_LOC* message, which represents target locations in local coordinates. The last row $i = 5$ contains the UAV position. Every time a target is reached correctly then the position row corresponding to the target is updated with [100, 100] to flag reached one.

The idea is to give to the algorithm both positions of the UAV and targets so that, based on coordinates, it can evaluate distance and possible best path to follow. Reached targets positions in *state* message are then flagged with [100, 100] to add information about already reached targets.

Note that in this case the RL algorithm takes as input a flattened version of the *state* message described. This

is because in this case a CNN would not be a fitting model for the problem, but a completely dense net is selected, thus requiring a 1D array type input.

The environment main task is to assign to the actor a cumulative reward so that it can learn, meaning weights of the policy NN are updated, thus a better policy is achieved each time the updating takes place. Moreover, a list of checks to determine the end of the episode are implemented. These checks are also part of the RL environment.
Reward set is a key component of the environment, is the main parameter with which is possible to change actor behaviour. The positive reward set in this case is defined as a quadratic function in time. Every time the UAV is able to complete the mission, reaches all five targets, the time passed between the start and the end of the episode is computed. Using this time value, the reward is obtained by $250 - \frac{3}{100}\ time^2$. Whereas if the action extracted corresponds to a target already reached, at every step, then a negative reward of -10 is assigned.

It is fundamental to check at each step if the episode is terminated or if environment conditions still permit the episode to continue. This is done with a series of checks the UAV must comply to always, otherwise the episode is declared as finished and a new one is started.
In table episode checks with a brief explanation of what they do and their effects on the algorithm are listed.

Bool parameter	Notes	Effects
is_out_of_bonds	Detects if UAV is inside the workspace or if for some malfunctioning is outside.	Switches episode_done to True.
is_upside_down	Detects if the UAV flips itself. In this case it will not be possible to continue with the training and a restart of the simulator is obligatory.	Switches episode_done to True.
is_disarmed	Detects if the UAV is on ground and disarmed.	Switches episode_done to True.
time_is_up	Detects if the time limit for the episode is reached	Switches episode_done to True.
all_target_reached	Detects if the UAV reached of the five targets in the environment	Switches episode_done to True.
episode_done	Detects if one of the previous conditions is True	Determines the episode end and restarts a new episode

Table 5.3: Episode checks to determine the end of the episode at each step.

When the reset episode is called the episode is interrupted and several actions take place which aim is to move targets in new positions, move the UAV in the starting position, and reset messages.
More on episode reset is explained in chapter 6, since this functionality is heavily dependent on the specific

test, train or evaluation that is being run.
After episode reset a new episode is ready to start.

5.2.3 Implementation

For the implementation explanation the chart in figure 4.1 is referenced again.
To start the learning process the user has to launch *actuate_RL_choice.py* to activate the control system that allows targets to be reached, then has to launch *start_RL_learning.py* for the training to take place.

Through *BGRcam_state_env_topic* is sent the *state_cam* message, to the Reinforcement Learning module so that it can be evaluated when an action is extracted from the policy. When the action is selected, it gets sent to the control module through the *RL_choice* topic.
The RL algorithm is implemented using Tensorforce, a package for managing agent-environment interactions. Tensorforce allows to define an environment from scratch with maximum customization options and then manages the learning process after agent and hyperparameters selected by the user. (Tensorforce documentation, s.d.).
Below Tensororce compatible obligatory functions implemented are listed:

- states(): Defines the *state* message dimensions.
- actions(): Defines the *actions* space dimensions.
- reset(): Resets the actor and the environment.
- execute(actions): sends the action selected to the control module and waits until a new target is reached.

5.2.4 Environment parameters

Parameters that fully define the environment are listed in table 5.4 with some explanatory notes.
When an episode is reset, the targets are moved from their previous position to the newly generated position, thus are not respawned. This functionality is implemented this way because Gazebo allows objects to be spawned in an already running simulation, but only if a unique name is given to the object, making it easier to simply move them.
Target positions are generated by checking, in both x and y coordinates, whether targets are more than *cubes_toll* away from each other. Targets must be moved within the workspace, defined in table 5.4, with a margin from the outer boundary of [x_min_lim, x_max_lim, y_max_lim, y_min_lim] so that when the UAV reaches them the environment does not detect that it is out of bounds.

For the reward set, several options are implemented to test different solutions during the design phase. For example, a reward can be given if all targets are reached, or a negative reward can be given if the UAV leaves the workspace boundaries or flips upside down. In the final project, only the values listed in the table are used, but the defined environment can be easily implemented for other UAV related RL tasks with a few modifications.

Environment Parameters	Value	Notes
cubes_toll	2.5 m	Minimum distance with which can be moved targets from one target to the other both in x and y.
start_point	[0, 0, 2.5] m	UAV starting point at each episode in local coordinates.
start_yaw	0 deg	UAV starting yaw in local coordinates in local coordinates.
pos_toll_RL	1 m	Tolerance in x and y to detect if the UAV is on top of the target and give the reward.
yaw_rate_search	10 deg/s	Yaw rate with which the UAV scans the environment at the start of each episode. See chapter 6.
time_limit	100 s	Time limit for each episode.
max_roll	90 deg	Max Roll angle to detect if the UAV is upside down.
max_pitch	90 deg	Max Pitch angle to detect if the UAV is upside down.
x_max	22 m	Parameters identifying the workspace limit in which the UAV can operate.
x_min	-2 m	-
y_max	15 m	-
y_min	-10 m	-
z_max	10 m	-
z_min	-5 m	-
x_min_lim	5 m	Parameters which identifies limits in which can be moved targets.
x_max_lim	3 m	-
y_max_lim	3 m	-
y_min_lim	3 m	-
reward_target_reached	0 [-]	Reward if a target is reached.
reward_all_target_reached	$250 - \frac{3}{100}\ time^2$ [-]	Reward if a target reaches all the targets.
reward_out_of_bounds	0 [-]	Reward if the UAV exit out of workspace bonds.
reward_upside_down	0 [-]	Reward if the UAV flips itself upside down.
reward_action_out_of_bonds	-10 [-]	Reward if an action is selected that corresponds to an already reached target.

Table 5.4: Environment parameters.

5.2.5 Agent parameters

Parameters that define the agent are listed in table 5.5 with some explanatory notes.

Agent parameters	Value	Notes
n_actions	5 [-]	Length of the action array.
nepisodes	600 [-]	Episodes of training.
nsteps	30[-]	Maximum step number which is possible to try at every episode in training.
nep_eval	100 [-]	Episodes in evaluation.
nst_eval	10 [-]	Maximum step number which is possible to try at every episode in evaluation.
agent	"dqn" [-]	Agent specification used to make decisions.
memory	66 [-]	Replay memory capacity.
batch_size	64 [-]	Number of timesteps per update batch.

Table 5.5: actor definition parameters.

The agent selection determines the operation of the algorithm, in this case the deep Q-network is implemented and operates as described in the original work (Volodymyr Mnih, 2015).
The goal of the agent is learning to select actions so that the cumulative future reward across episodes is maximized, to accomplish this task the optimal action-value function is defined as:

$$Q(s,a) = \max_{\pi} E[r_t + \gamma\, r_{t+1} + \gamma^2\, r_{t+2} + \cdots | s_t = s, a_t = a, \pi]$$

The idea is to maximize the expected value of the sum in brackets across all possible policies $\pi = P(a|s)$. r_i is the reward at each time step *i*, whereas γ is a discount factor, that allows to discount rewards obtained from past steps and give importance to rewards obtained more recently. The action-value function depend on the state *s*, or observation, and action taken *a*. With these two information is possible to compute how valuable is an action depending on state and update the policy accordingly.
The experience replay method is implemented to stabilize the learning process when a Neural Network is used to describe the policy. Again, the experience replay is a method developed in (Volodymyr Mnih, 2015). At each step, the reward, action, previous and current states are stored in a database, and then at each update samples are drawn uniformly from the database to compose minibatches that are used in the actual update process. The aim is to minimize the loss function defined as:

$$L_i(\theta_i) = E_{(s,a,r,s')\sim U(D)}\left[\left(r + \gamma \max_{a'} Q(s', a', \theta_i^-) - Q(s, a, \theta_i)\right)^2\right]$$

Where values with " $'$ " are referring to the $t+1$ timestep whereas other values, without apex, are referring to t timestep. θ_i are parameters, weights that define the policy, at timestep t, whereas θ_i^- are targets parameters used to compute the difference between the action-value functions. Note that θ_i^- parameters are updated only

every C steps and not every iteration like θ_i. γ this time defines the agent's horizon in terms of Q value. At every step the action-value function is compared to a target function, they depend respectively on action and states at time t and action and states at time $t+1$, the target function is evaluated in all the possible future actions a' so that a horizon is obtained. This process is propagated using all (s, a, r, s') parameters extracted from the minibatch for the update. The aim is to chose θ_i so that the above reported relation is minimized.

For the implementation the memory parameter determines how many samples are stored in the database used to generate minibatches, whereas batch size is the capacity of every minibatch. In this case a memory of 66 and a batch size of 64 are used, so that at every update almost all the samples contained in memory are extracted. With a higher memory no differences in the result are highlighted.
Episodes used in training are highly variable from task to task, in this case a total of 600 episodes, are used for multiple reasons. Firstly, as shown in chapter 6, reward across episodes converges before the training ends, this means even with more episodes results may not change much. Moreover, as explained in conclusions, Gazebo is not an optimal simulator to train RL models, thus limiting greatly training episodes number.

5.3 Deterministic policy

This section explains the implementation of a simple deterministic policy that chooses the target to reach based on distance, as an alternative to the RL method.

5.3.1 Concept

In the previous chapter the RL policy method of target selection was explained, this time a different, simpler policy is implemented. This is done mainly to compare results of the RL policy to results of a standard deterministic policy. In addition, a deterministic policy, makes it possible to test more rapidly the allocation algorithm previously described, as it requires less workload on the hardware and errors can be spotted easily.

The idea is to select at every step the target, between the ones which are still not flagged as reached, that is closer to the UAV. This simple policy however does not minimize in every case the time for each episode to complete the mission, i.e. is reaching all the five targets.
In some cases, depending on targets disposition, can happen the minimum distance policy minimizes also the time required to complete the mission, but in others it won't.

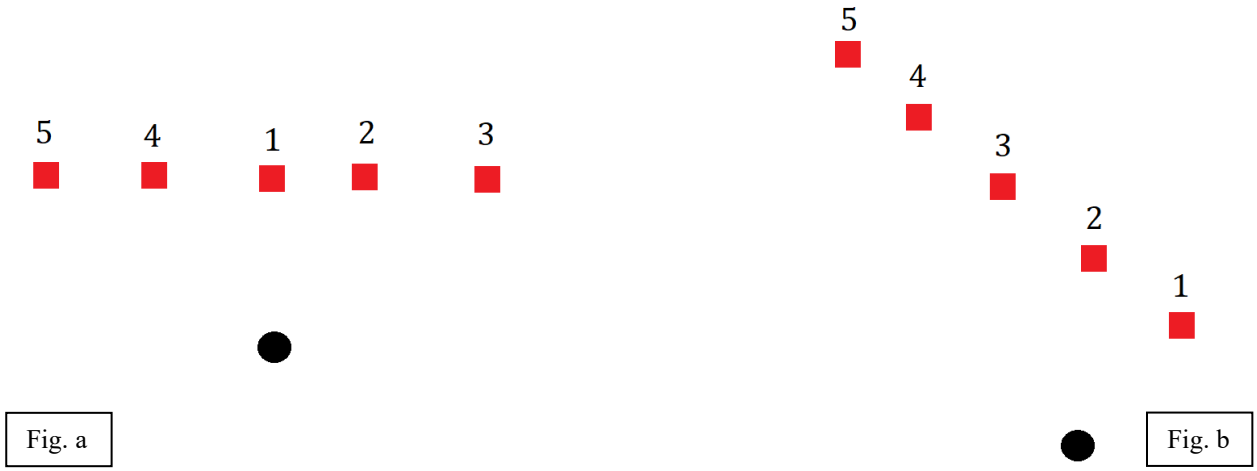


Figure 5.4: practical example proving the policy based on minimum distance is not always the best policy. On the left side figure a, on the right side figure b.

For instance, in the case represented in figure 5.3b, minimum distance target choosing also coincides with the optimal solution in term of time expended to finish the mission. In fact, the UAV will start from the closer side of the targets line and choses sequentially all the other targets as represented. However, in 5.3a is represented a case in which minimum distance choosing is not the best solution as the closer target is the one in front of the UAV, this leads to the UAV choosing a path which will overlap, thus obtaining a suboptimal solution in time expended to complete the mission.

5.3.2 Implementation

For the implementation, the idea is to keep the basic concept of the Reinforcement Learning state – action interaction but switching the policy with the deterministic rule. This allows the algorithm to run multiple scenarios one after another to simulate more cases and test the robustness of the algorithm.
The implementation chart, in terms of python modules, is depicted in figure 5.5.
It can be noticed the chart is similar to the RL one, the only difference is the *start_RL_learning.py* is replaced with *iris_follow_targets.py*. The new module does not contain *states()* and *actions()* methods since Tensorfoce is not required.

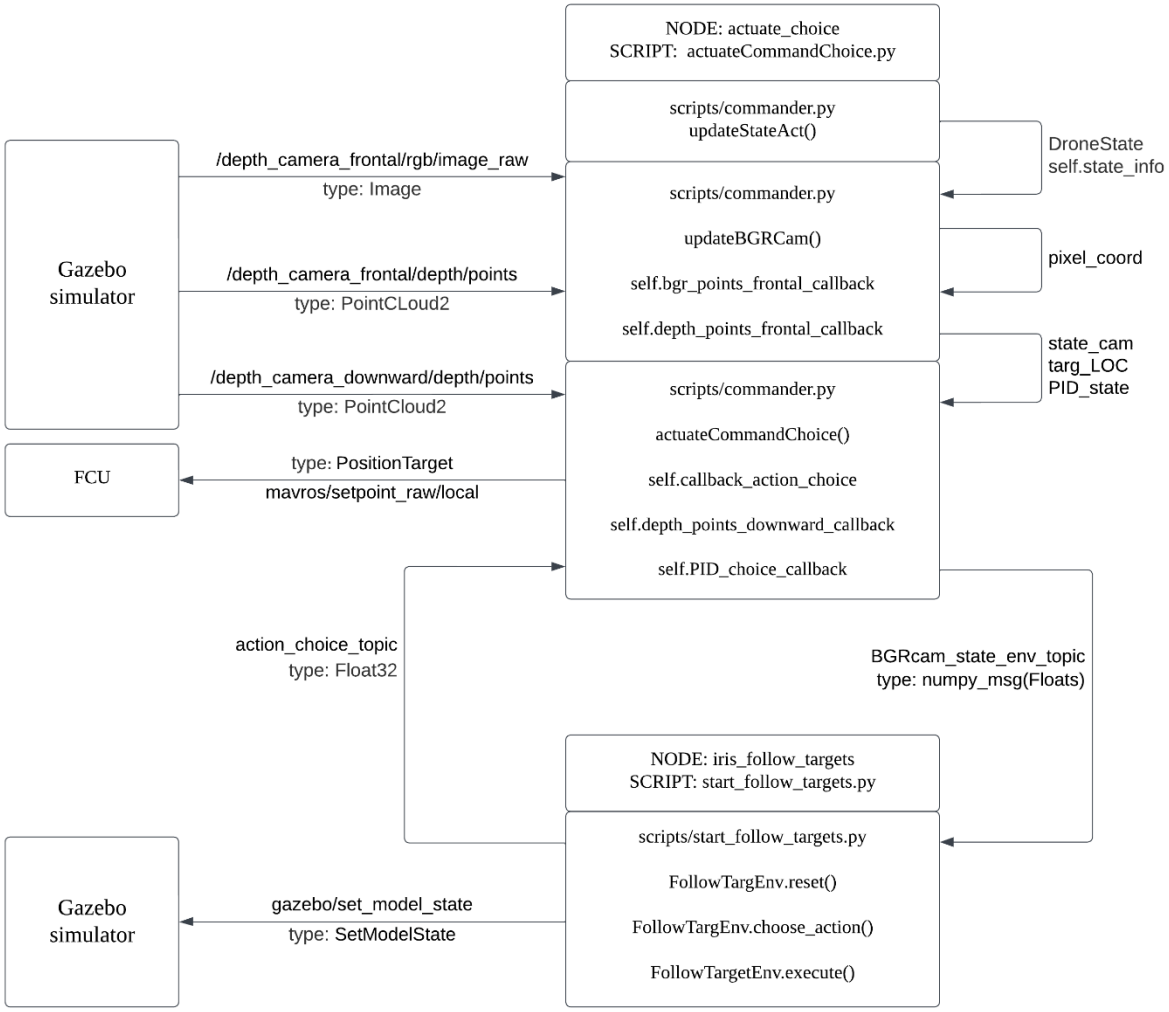


Figure 5.5: Implementation flowchart with the deterministic policy.

6 **Results**

In this final chapter, results are shown in the case of deterministic policy implementation, with and without noise applied to the UAV position signal to simulate a malfunction in GPS and Kalman filter. The results of the RL policy are also reported and analysed.

6.1 Deterministic policy

In this section, several results are shown in multiple case scenarios, such as targets in line and scattered targets. From the graphs presented it is possible to evaluate the control system commands passed to the FCU, described in section 5.1, and the performance of the allocation algorithm, described in section 4.2.

6.1.1 Test execution

The aim of this section is to describe preparation steps for the tests regarding the deterministic policy. First of all, results shown in the next sections are obtained without any noise added to the UAV position. This means these tests are relative to an ideal case in which there is no GPS noise and Kalman filtering, simulated in Gazebo, works without any issue.
When the test is started the *reset()* function is called, which, in this case performs the following actions:

- Targets are moved to a random generated location in the workspace limits or to predetermined locations depending on the specific test.
- *targ_LOC* and *state_cam* (*state*) messages are rest both with [0, 0] in all the rows.
- Through the extra message “101” the UAV will turn itself first to the right and then to the left and then turns again to the initial heading value to scan all the environment and see all targets.

Resetting the messages is fundamental as this way is possible to evaluate the performances of the allocation algorithm, such as how fast it can pick up targets and how precise it is.
After these actions are completed, the UAV starts to follow the target chosen based on the distance deterministic policy. The test ends either if it fails, the UAV does not pass one of the checks described in table 5.3, or if it completes the mission, reaching all five targets.

6.1.2 Vineyard row

In the first test targets are not moved randomly but collocated in a diagonal line through small modification in the algorithm. The aim of this test is to represent a line of targets, since in most of agriculture scenarios plants are arranged in this way for productive and process purposes. For instance, this test can represent a vineyard row where products must be applied.

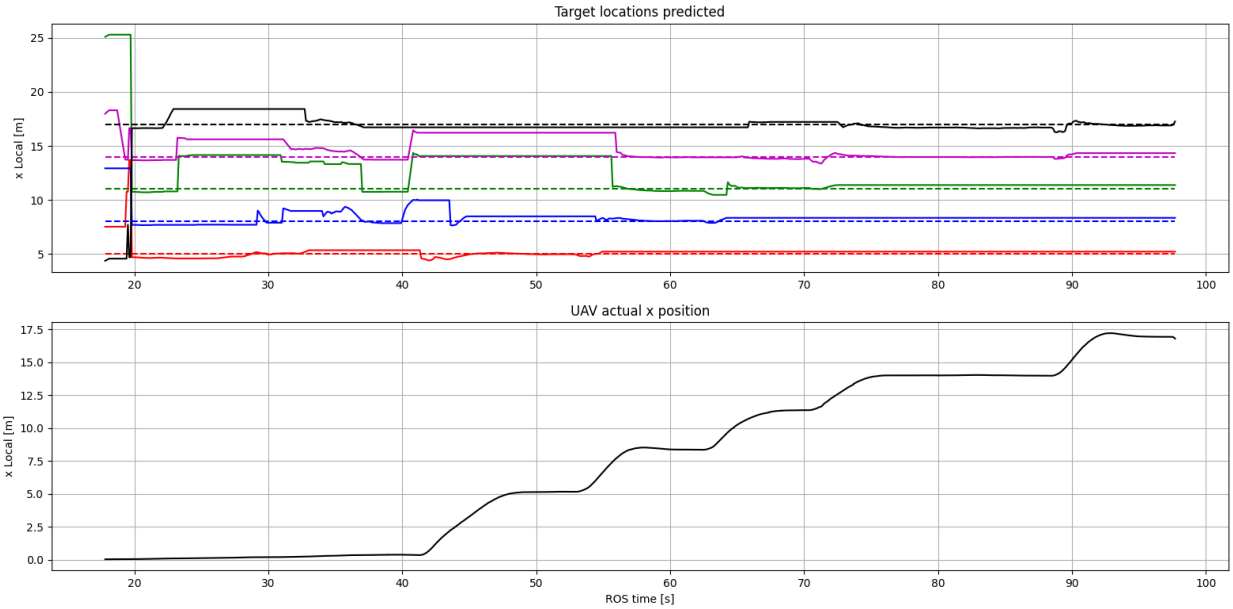
In figure 6.1 the real targets positions in dotted lines, which remains the same during the test, and the predicted target allocation algorithm ones, that varies in ROS time, are represented. The graphs depict the whole episode until the UAV manages to successfully reach all targets. Below each graph, for both x and y, the UAV position is shown. At around time 20 s targets are moved and messages are reset, in this case all targets are already in visual contact with the camera, thus all of target's positions are correctly pinpointed by the allocation algorithm with and error that varies in time. In the x graph, it appears the magenta and green curves overlaps thus confusing the two targets at around 25 s and 40 s marks. This does not happen as (x, y) target predicted coordinates are updated together if both of them are under a tolerance. In the y graph, in fact, at value 25 s does not show any significant error for the magenta and green curves, and in the 40 s, even though a bigger imprecision is present is not high enough to signify any malfunction or confusion between target located.
Black solid curves, representing the UAV position, are shown so that precision of targets predicted position can be related to the progress of the test. For instance, at 55 s mark the UAV gets closer to the targets corresponding to magenta and green lines, when the camera sees both targets again the algorithm adjusts their local position, greatly reducing the estimation error.
The relation that ties estimation error and UAV position is not always observable, since in most of the cases the algorithm is able to pinpoint with precision targets from the beginning and more importantly because visual contact with targets is not only dependent of UAV position but also on its heading.
In figure 6.3 the error in x and y is represented for every target which varies with the time. Another visual way for representing error in pinpointing the targets positions is as in figure 6.2, where blue ellipses width and hight are maximum errors in y and x respectively evaluated during the whole test. UAV complete flight path is also shown along with red targets scaled accordingly to the graph's dimensions. Targets are enumerated in the order in which they are moved during the reset of the environment. It is possible to relate target's number to the colour used in graphs (1 – red, 2 – blue, 3 – green, 4 – magenta, 5 – black).
In figures 6.4, 6.5 and 6.6 commands for the x axis, y and yaw and z axis, in the camera's frame of reference, respectively are depicted. Colours, in this case, are selected as in figure 5.1 so that every type of command can be easily recognizable. In figure 6.4 the dotted red line represents the detected target of choice to be reached, whereas dotted black line is the desired value, *camera_x_targ*. If in the camera's frame of reference, the chosen target is in the desired position then the target is directly under the UAV in x local coordinates. The solid red line is the command generated by the control system in terms of velocity.

It can be noticed that when a target is chosen then the target state spikes in the negative values and the commands acts accordingly to bring the camera detected target to *camera_x_targ*. This process repeats itself five times over. This happens only if the target is seen directly by the camera.

If the chosen target is not on the visual contact with the camera, then the control method is switched, as happens in this case around 50 s – 55 s marks. Figure 6.5a is relative to the command given in the y axis, whereas the figure 6.5b is relative to the yaw rate command. This time the desired value is always 0 when visual controls are applied, as in both y, when the UAV is moved with strafe commands, and yaw, when the UAV is moved in heading, the aim to align the target with the x body axis of the UAV.

In most of the cases the y commands are used more than yaw because targets are close one to another, however at marks 40 s and 52 s the UAV is not turned towards the target, so a command is given to turn so that the camera can see it. In this case the desired value to be reached is different from 0 and varies depending on target's angle with respect to the UAV's heading.

In figure 6.6 commands generated for the z axis are represented along with the desired altitude and the actual UAV's altitude. This is also the easier graph to read since there are not any switching of command modes.



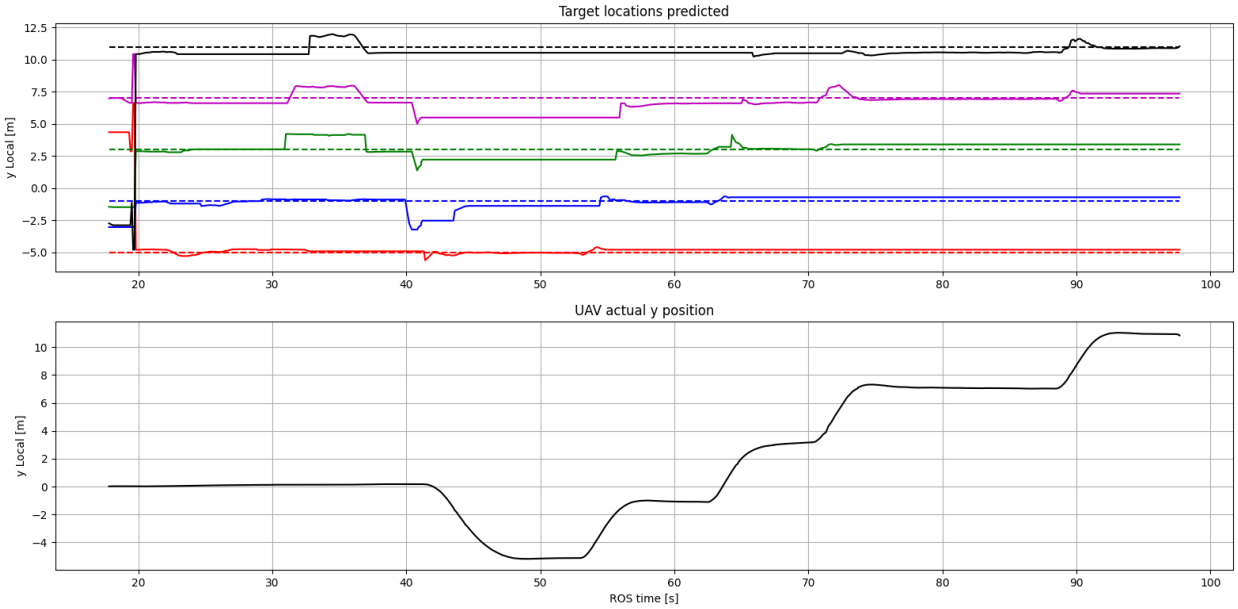


Figure 6.1: predicted target position and actual target position in x and y local coordinates.
From the top, the first and the third graphs:
(—): predicted target 1 position [m], (--): actual target 1 position [m]
(—): predicted target 2 position [m], (--): actual target 2 position [m]
(—): predicted target 3 position [m], (--): actual target 3 position [m]
(—): predicted target 4 position [m], (--): actual target 4 position [m]
(—): predicted target 5 position [m], (--): actual target 5 position [m]

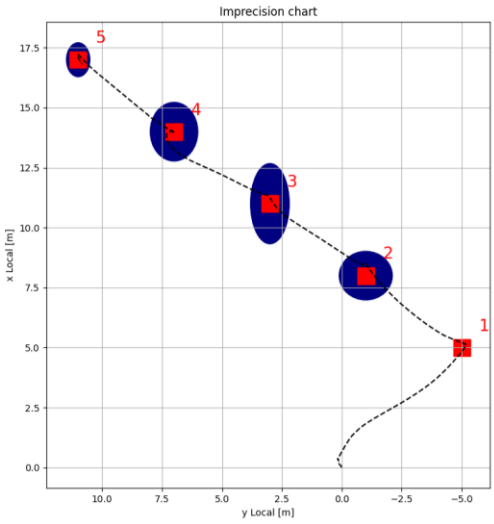


Figure 6.2: Top-down workspace graph.
Targets scaled, maximum errors in x and y, local coordinates, evaluated during the whole episode,
(--) UAV's trajectory.

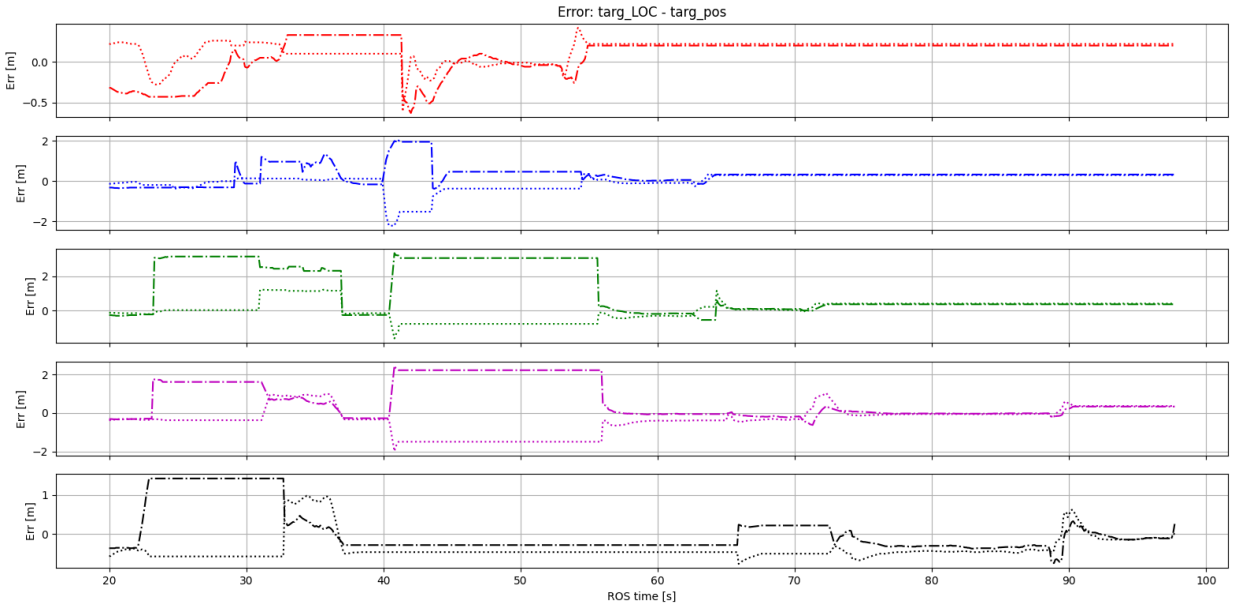


Figure 6.3: Error graph
(-.-): error on x local target 1 [m], (···): error on y local target 1 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 2 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 3 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 4 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 5 [m]

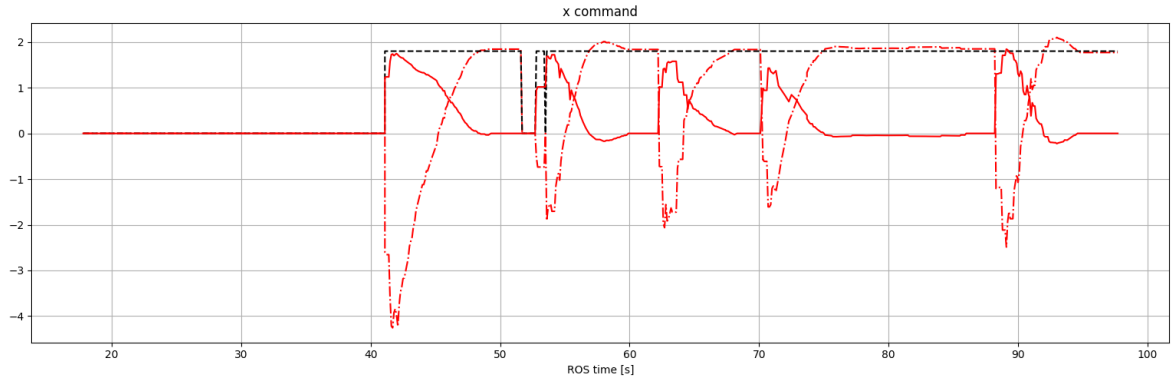


Figure 6.4: Command graph for x axis
(—): command on x axis [m/s], (--): y camera target state [m], (···): desired y camera target state [m]

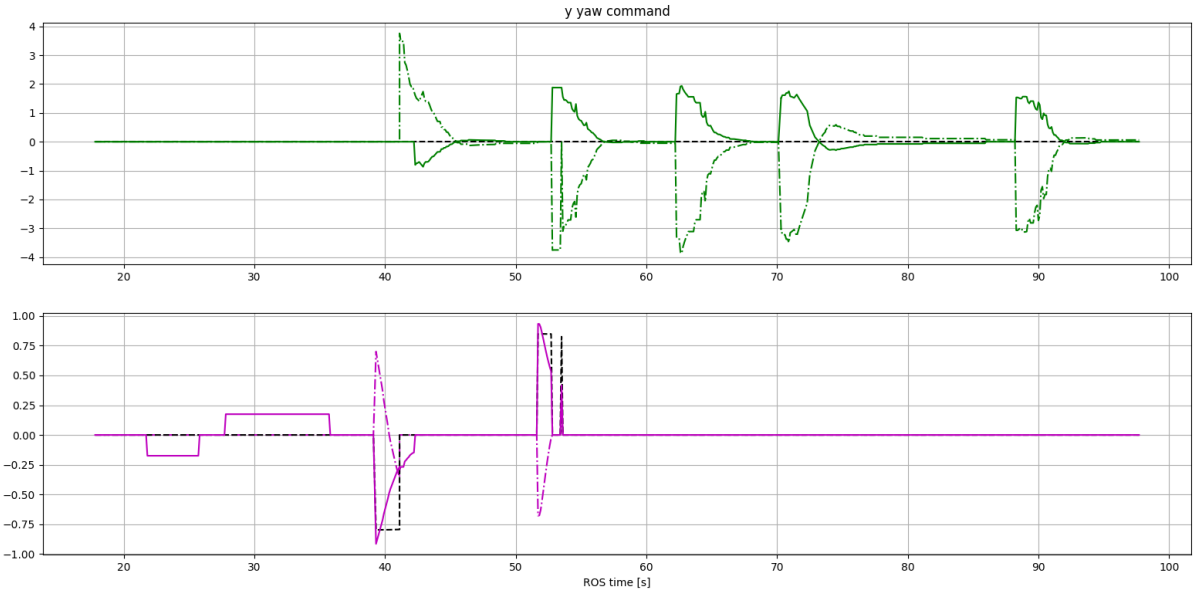


Figure 6.5: y and yaw rate commands. The first graph, figure a, represent commands to the y axis, while the second one, figure b, depicts commands in yaw rate.
(—): command on y axis [m/s], (--): x camera target state [m], (···): desired x camera target state [m]
(—): command in yaw rate [rad/s], (--): heading camera target state [rad], (···): desired yaw camera target state [m]

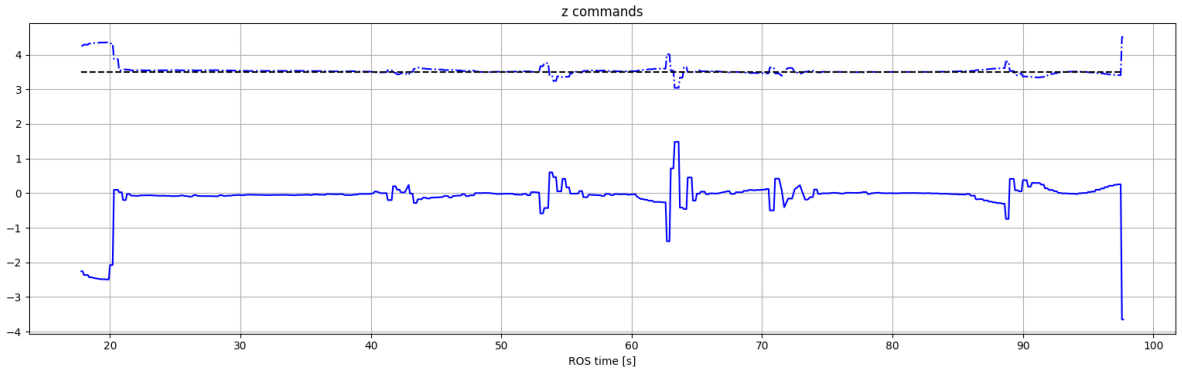


Figure 6.6: z control graph.
(—): command on z axis [m/s], (--): UAV's altitude [m], (···): desired altitude [m]

6.1.3 Scattered targets

This case is run as the previous one, but targets positions are random generated, thus providing with sparce targets in the workspace area.

As before, in figure 6.7 predicted targets positions and real ones are depicted for x and y axis. Also, in this case the algorithm is able to pinpoint, with errors reported in figure 6.9, all target positions.
Again, the magenta and blue lines deviate quite a bit from the exact target position, but, since the updating of *targ_LOC* message relies on both coordinates, the algorithm is not confusing them. In fact, for instance at 55 s mark the magenta line in the x graph moves towards the dotted blue line, however in the y case it moves towards the green line, this signifies the algorithm is making an error on the prediction but is not mixing two targets.
As in the previous case in figure 6.8 maximum errors evaluated in the whole test are represented. Although bigger errors are present with respect to the previous case, this does not compromise the mission. When the algorithm selects the target affected by an error it uses its approximative position to turn in heading towards its general position, and, if the target is still not detected by the camera, the UAV starts to move in its predicted position anyway. Eventually the target enters in the visual range of the frontal depth camera, thus causing the control method switching to only visual guidance. From this point on the stored position depicted in graphs is used only if visual contact is lost. As the target gets closer the error, in both visual control and the memorized position one, diminishes as it is easier to make an estimation, thus making the control system effective at reaching targets with precision.
In figure 6.10 the evolution of the predicted targets position error is visualized using snapshots of the test, the images are generated at the start of the test and each time the UAV successfully reaches a target.
At the beginning, 6.10a, the UAV is still, and almost all the targets are pinpointed with a low error. Only targets number 2 and 4, which are the farthest ones, are pinpointed with a greater error.
In 6.10b, the UAV moves to the closer target the number 3 however, since targets 2, 4 and 5 are still in the camera's visual range, their stored positions are still updated. With respect to the previous snapshots targets 2 and 4 position is estimated with a greater error, this is probably caused by the movements, dynamic response to command, of the UAV. Meanwhile target 5 position, which had an already small error, is estimated even better since the UAV is closer to it than before.
In 6.10c the UAV moves to target 5 and, in its path, enters in visual contact again with target 4 but this time is closer. This makes possible for the algorithm to correct for the previous error and estimate precisely target 4 position.
In 6.10d the UAV moves to the target 4 and sees again target 2, which allows to correct its predicted position.
In 6.10e and 6.10f, the UAV reaches both targets 2 and 1 thus completing the mission objective.

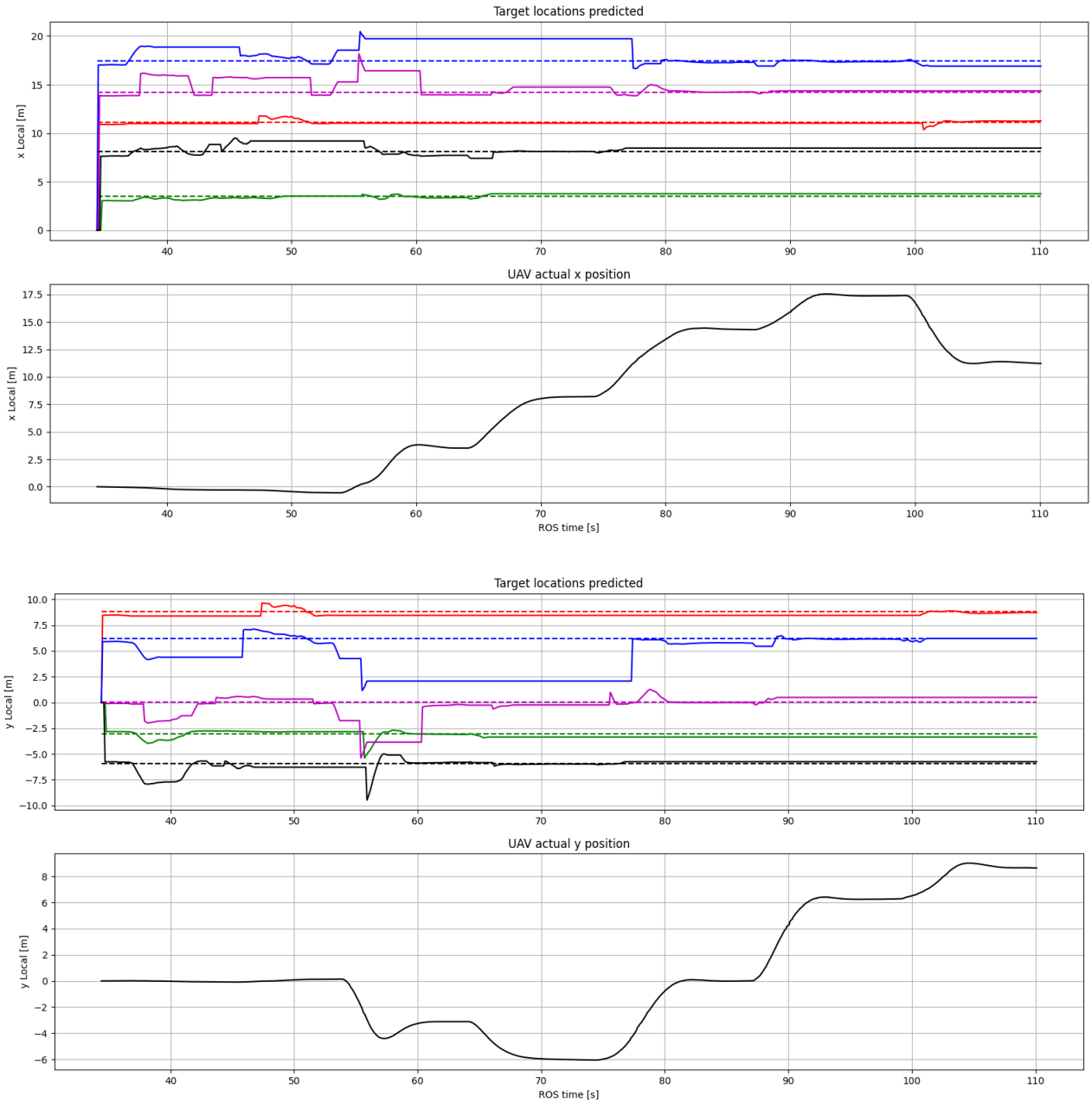


Figure 6.7: predicted target position and actual target position in x and y local coordinates. From the top, the first and the third graphs:
(—): predicted target 1 position [m], (--): actual target 1 position [m]
(—): predicted target 2 position [m], (--): actual target 2 position [m]
(—): predicted target 3 position [m], (--): actual target 3 position [m]
(—): predicted target 4 position [m], (--): actual target 4 position [m]
(—): predicted target 5 position [m], (--): actual target 5 position [m]

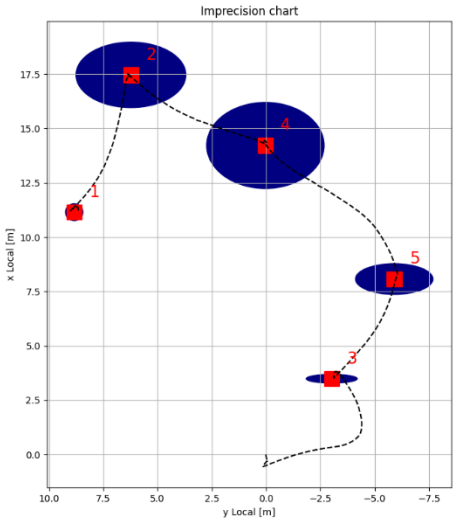


Figure 6.8: Top-down workspace graph.
Targets scaled, maximum errors in x and y, local coordinates, evaluated during the whole episode,
(--) UAV's trajectory.

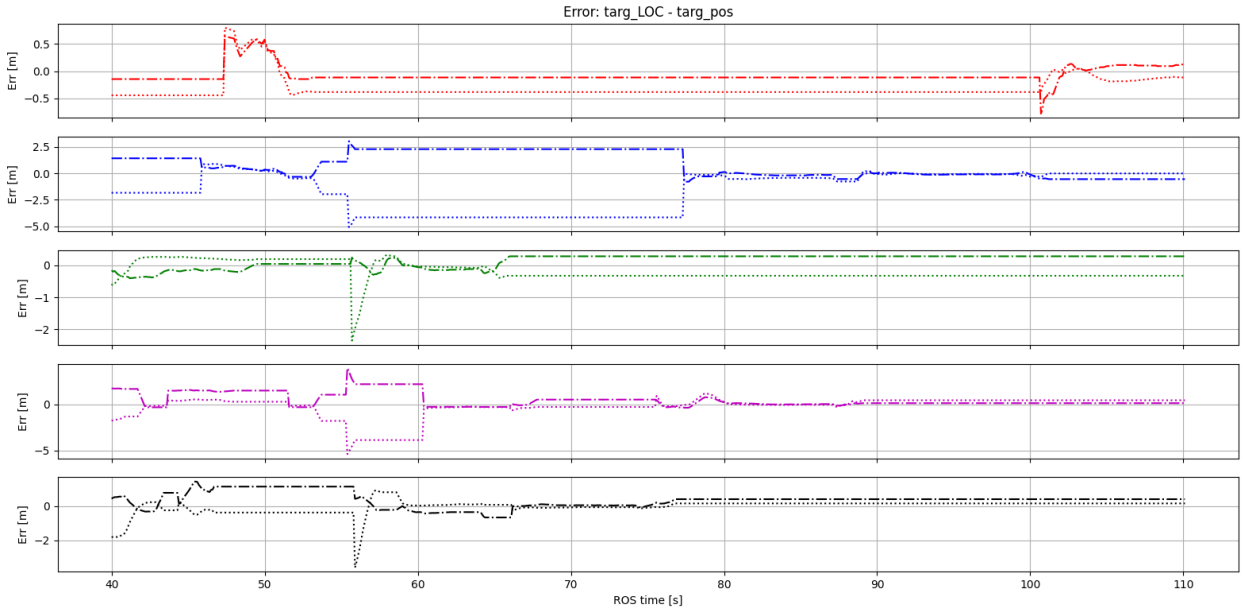
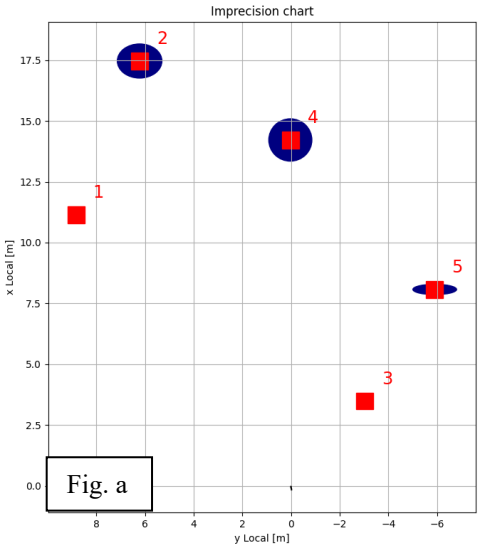
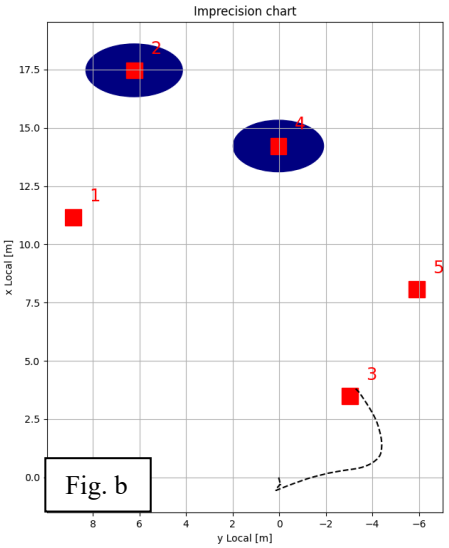
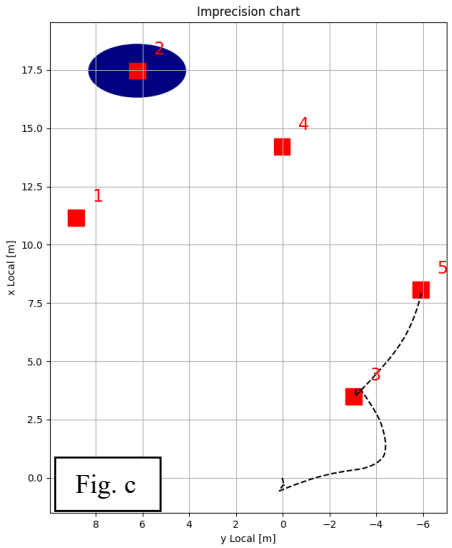
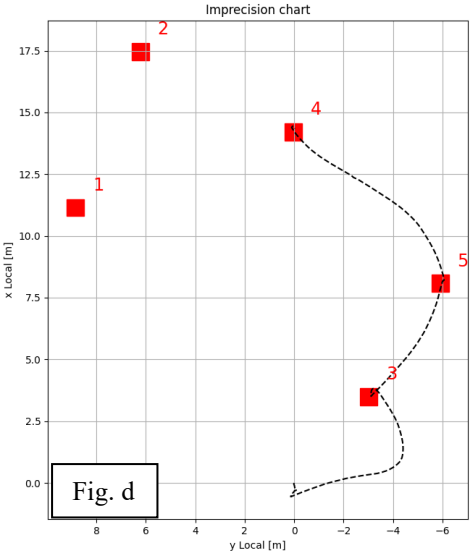


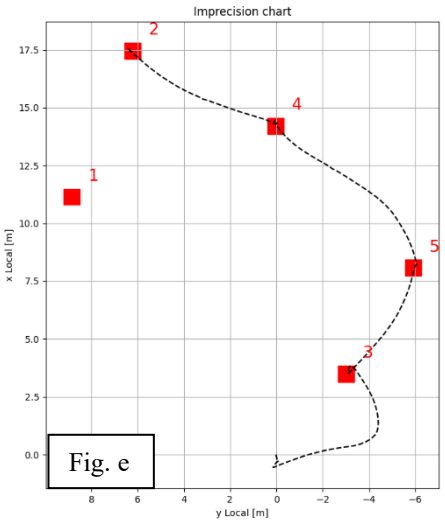
Figure 6.9: Error graph
(-.-): error on x local target 1 [m], (···): error on y local target 1 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 2 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 3 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 4 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 5 [m]











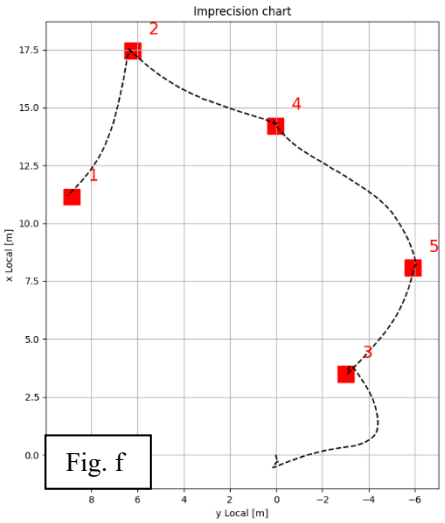


Figure 6.10: top-down view of the test with snapshots taken at the beginning and every time a new target is reached. Figure a: top left, figure b: top right, figure c: middle left, figure d: middle right, figure e: bottom left, figure f: bottom right.

6.1.4 Targets switching

This case is run exactly as the previous one, this means target positions are randomized at the beginning of the test.

In this case in figure 6.11, towards the start of the test at around 55 s and 75 s, when the UAV is still turning in search for targets, happens that some of them are reallocated in *targ_LOC* message in a different row position to the initial one. For instance, the black curve under the 60 s mark, which is relative to target 5, estimates the target position very close to the blue dotted line in both of the axis, but after seeing also the last target, the number 1, represented by the red lines, the black solid line jumps to the black dotted line, the correct one. Differently to the previous cases, in which this phenomenon does not show up, this time both the x and y estimation varies together from the blue dotted line to the correct black dotted line, thus meaning the target changed position in *targ_LOC*. This does not compromise in any way the mission as it happens before the UAV reaches the first target but can be a serious issue if it happens after on, when one or more of the switched targets are already reached. The issue arises as target reached are flagged enumerating them as they are presented in *targ_LOC* message, thus if two of them switch place the algorithm flags as reached the wrong one.
In figures 6.12a and 6.12b, the maximum errors on target's positions are depicted from the start of the test and after the correct reallocation of targets respectively. In 6.12a errors are large, specifically targets 2 and 5 ellipses overlaps one over the other in both coordinates. The overlapping can also be seen in figure 6.11, where target 2 is described by the blue lines while target 5 is described by the black lines.
In 6.12b, after the reallocation, errors are much smaller, the algorithm is able to pinpoint target's position correctly.

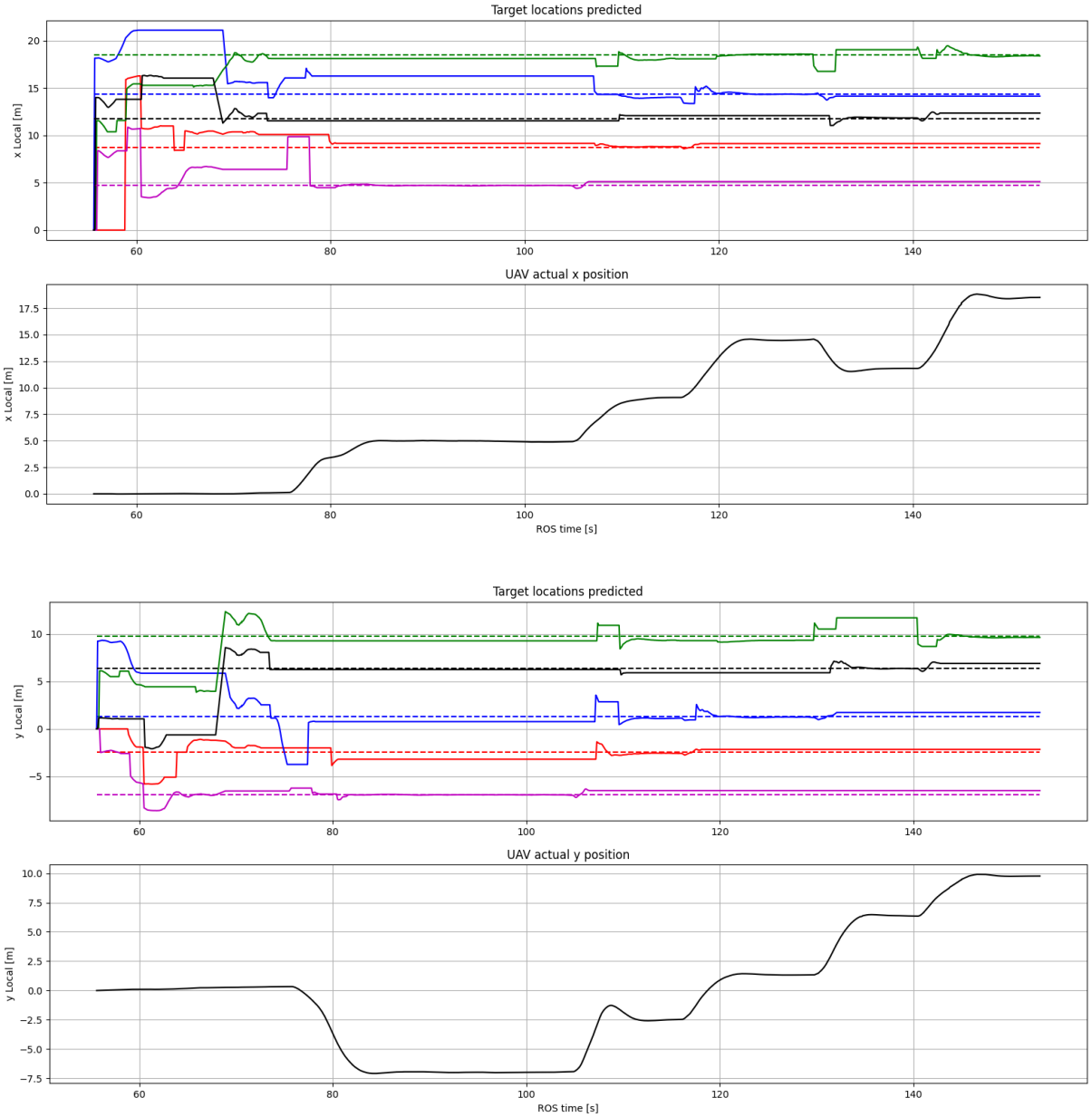


Figure 6.11: predicted target position and actual target position in x and y local coordinates. From the top, the first and the third graphs:
(—): predicted target 1 position [m], (--): actual target 1 position [m]
(—): predicted target 2 position [m], (--): actual target 2 position [m]
(—): predicted target 3 position [m], (--): actual target 3 position [m]
(—): predicted target 4 position [m], (--): actual target 4 position [m]
(—): predicted target 5 position [m], (--): actual target 5 position [m]

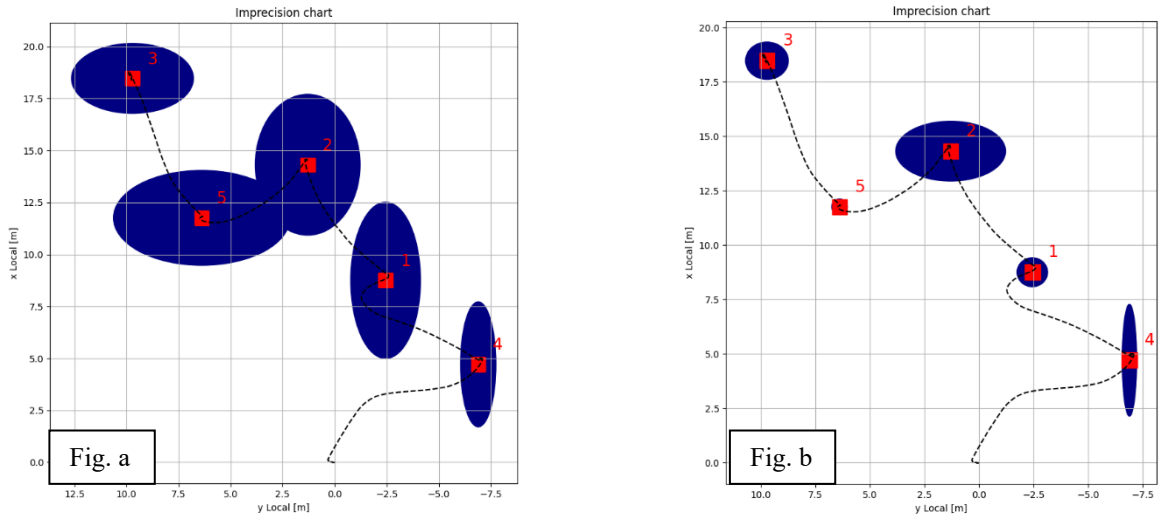


Figure 6.12: Top-down of the workspace, errors are evaluated before the targets are correctly located in the figure a, on the left side, and after in figure b, right side.

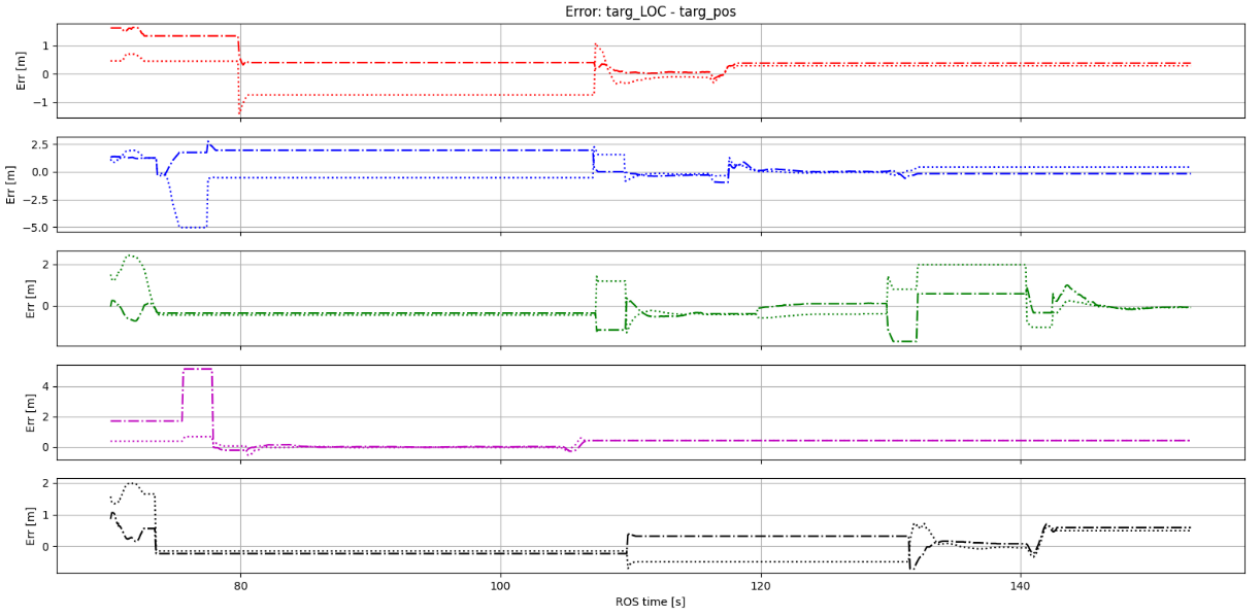


Figure 6.13: Error graph
(-.-): error on x local target 1 [m], (···): error on y local target 1 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 2 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 3 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 4 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 5 [m]

6.2 Deterministic policy with noise

In this case, noise is added to the UAV's position signal, and similar tests to the previous ones are carried out.

6.2.1 Test execution

In this case the test steps are the same as the ones described in chapter 6.1, however noise in the position signal extracted from the UAV's FCU is added.
Noise values are extracted from a normal distribution defined with $\mu = 0$ and σ variable. The disturbance signal is added directly to the message obtained from *mavros/local_position/pose,* effectively modifying the actual position of the UAV in *DroneState* message. The noise is actually added after the Kalman filter to consider the worst-case condition in which not only the GPS is affected by great errors but also IMUs outputs imprecise information. Furthermore, if the noise is added only on GPS signal it would be difficult to assess the allocation algorithm performances since the noise would change shape depending on the filtering process. The UAV position, as explained in the previous chapters, is used by the allocation algorithm to find correct targets positions, thus making it a crucial information for the control system operation when the target chosen is not in camera sight. However, once targets are correctly localized, and the UAV's camera sees the target to reach, then the disturbance signal should not have any impact on the control system part that acts when there is the visual contact with the target.

Another major difference on the algorithm here applied and then kept as part of the control system is the implementation of a *PID_buffer* variable. *PID_buffer* holds up five occurrences of requests of control mode switching from visual to following memorized target positions. When *PID_buffer* holds all the five requests then happens the switching, effectively generating a delay. However, the delay is none if the control switch is from following memorized targets to visual control.
This solution is implemented as, during tests with noise, when the UAV is in visual control mode close to the target, happens that the target is instantly lost, thus reverting back the control mode to follow memorized targets. This generates instantaneous commands in yaw rate that, although do not compromise the mission, makes the UAV behaviour different with respect to the expected one. With *PID_buffer* method the issue is completely solved.

6.2.2 Vineyard noisy row

The same test run in section 6.1.2, where targets are aligned diagonally, is implemented. This allows to compare results with and without disturbances in the position signal.

For this test a high value of $\sigma = 0.7$ is chosen to define the normal curve to extract noise values. In figure 6.14 the two normal curves, for both coordinates, are obtained as histograms with data collected from the whole test. Instead in figure 6.15 the UAV disturbed position in local coordinates is shown and in figure 6.16 the rate between the noisy signal and the ideal one is reported. 68.2 % of the noise generated is around $\pm 1\ m$ in both the axes, while the remaining percentage ranges from about [1, 2] m and [-1, -2] m, there is only one occurrence of a value which is over 2.5 m. As before results are shown in figures 6.17, 6.18 and 6.19.
The test proceeds similarly to the vineyard test without noise, the UAV is able to correctly locate each one of the targets and reaches them successfully.
Other more specific considerations are made in the next cases as some phenomena are more evident.

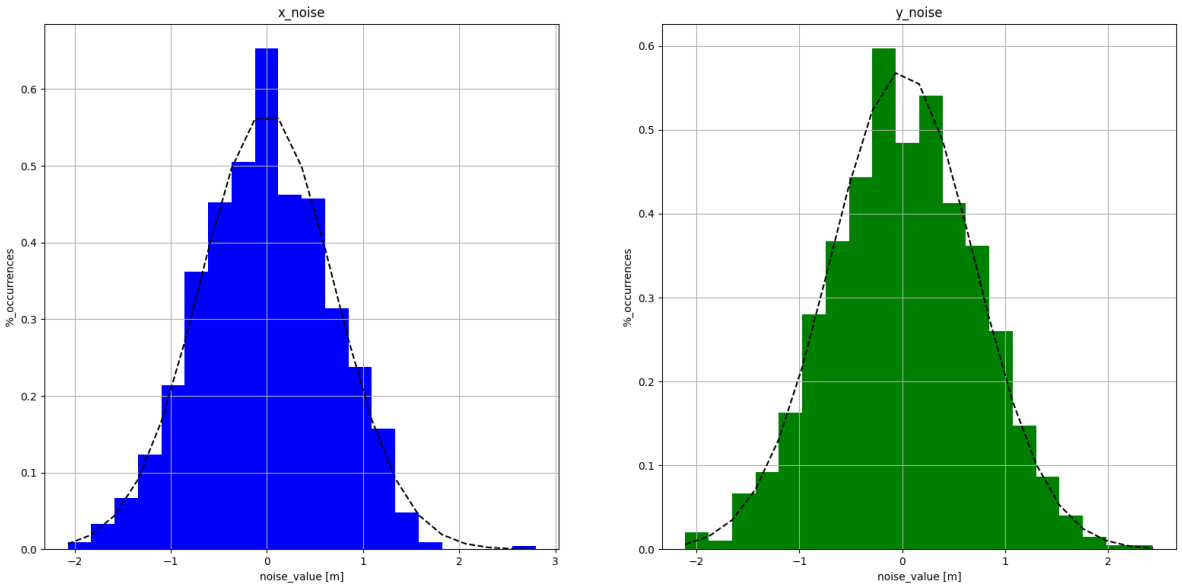


Figure 6.14: Normal distributions obtained as histograms for noise generation. On the y axis % of occurrences of the same value are shown. On the left side the x noise distribution, one right side the y noise distribution in local coordinates.
x local noise histogram [%], y local noise histogram [%], (--): corresponding analytical normal curve [%].

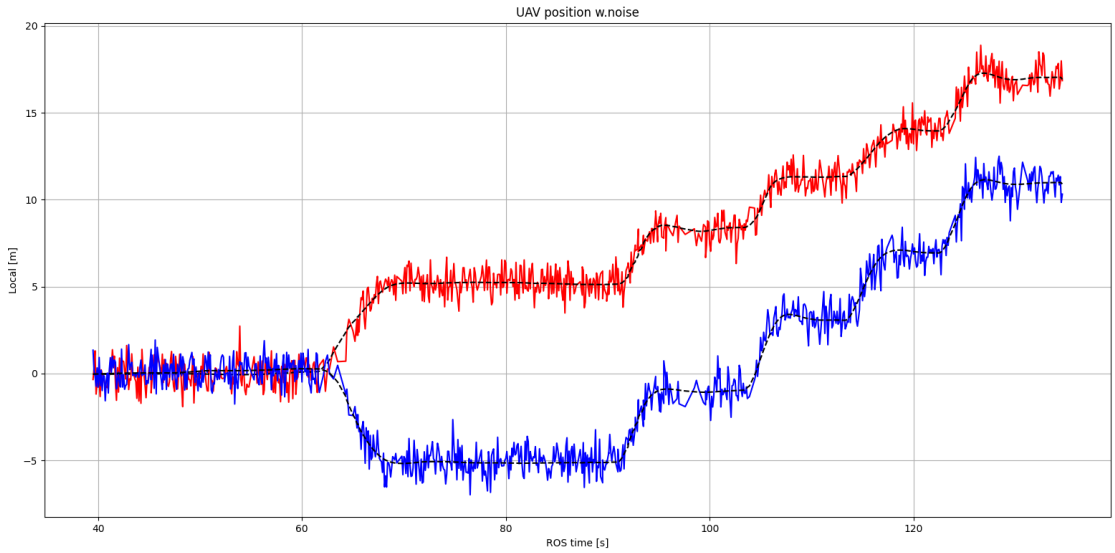


Figure 6.15: UAV's position on local coordinates with noise added.
(—): noise added on UAV's position x signal [m], (—): noise added on UAV's position y signal [m],
(--): UAV's position (x, y) [m].

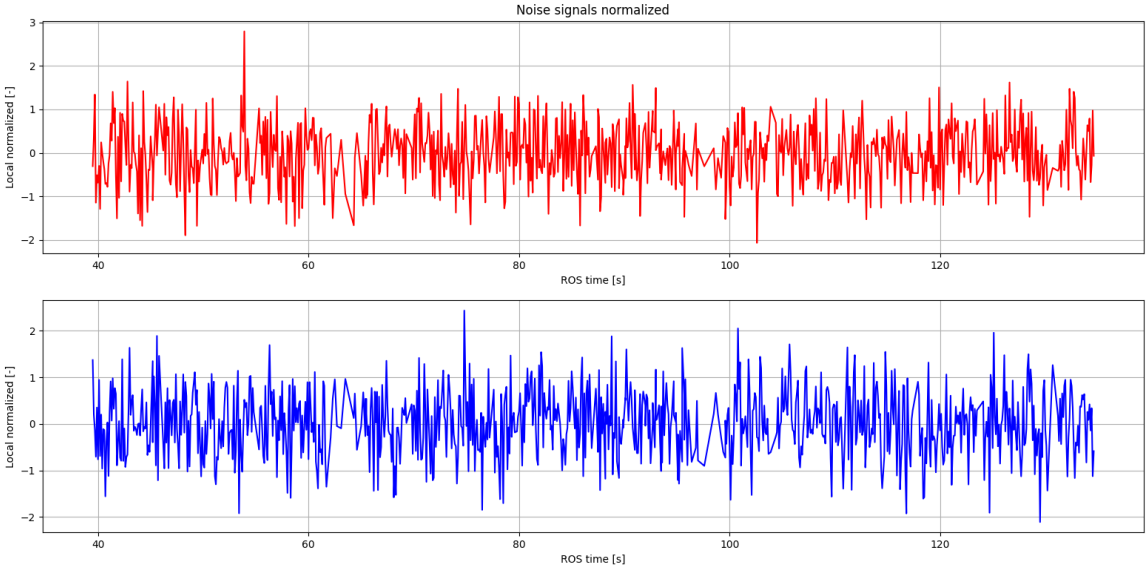


Figure 6.16: Rate between UAV's position with noise and UAV's ideal position.
(—): Rate for x local position [-], (—): Rate for y local position [-],

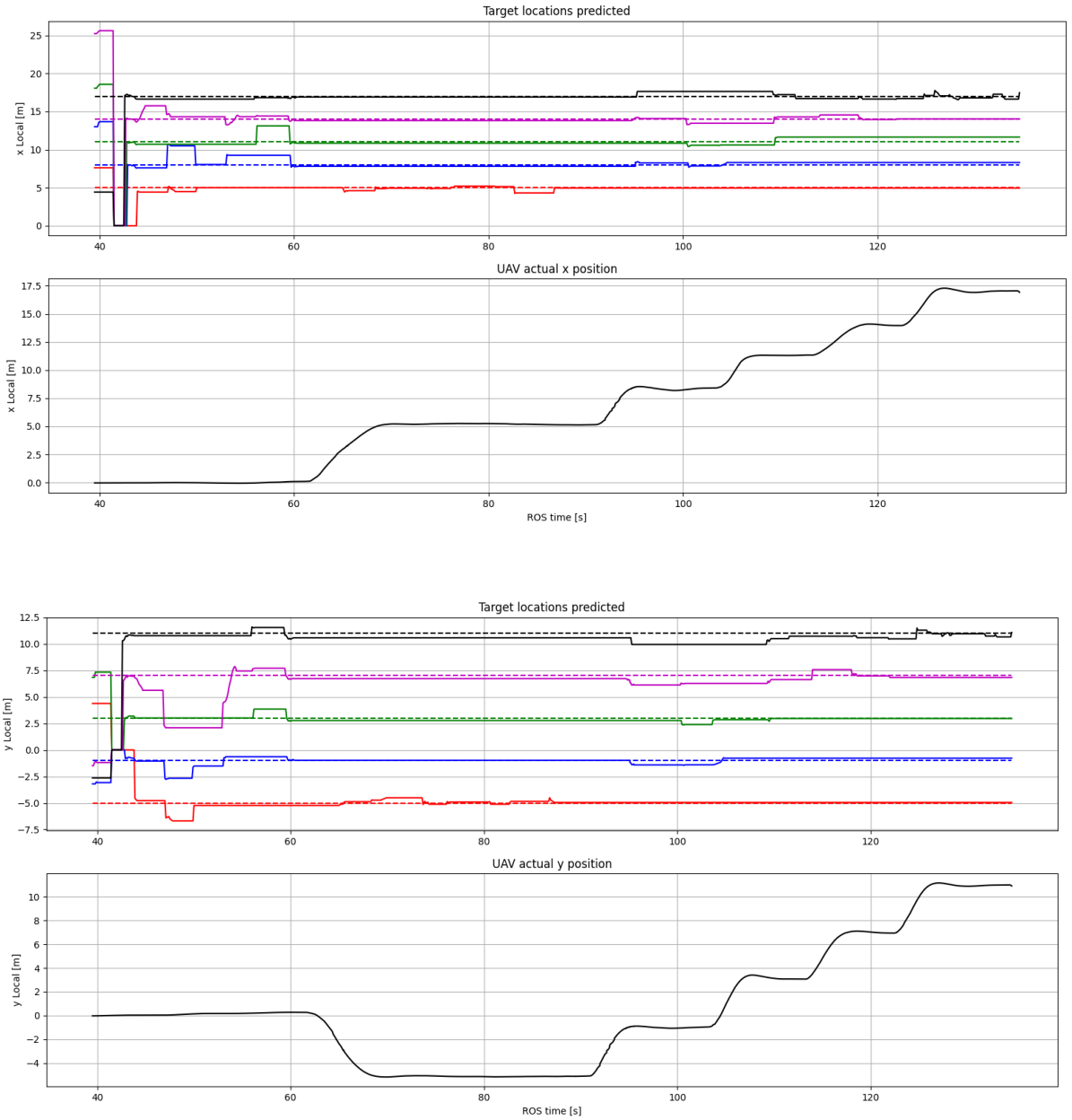


Figure 6.17: predicted target position and actual target position in x and y local coordinates. From the top, the first and the third graphs:
(—): predicted target 1 position [m], (--): actual target 1 position [m]
(—): predicted target 2 position [m], (--): actual target 2 position [m]
(—): predicted target 3 position [m], (--): actual target 3 position [m]
(—): predicted target 4 position [m], (--): actual target 4 position [m]
(—): predicted target 5 position [m], (--): actual target 5 position [m]

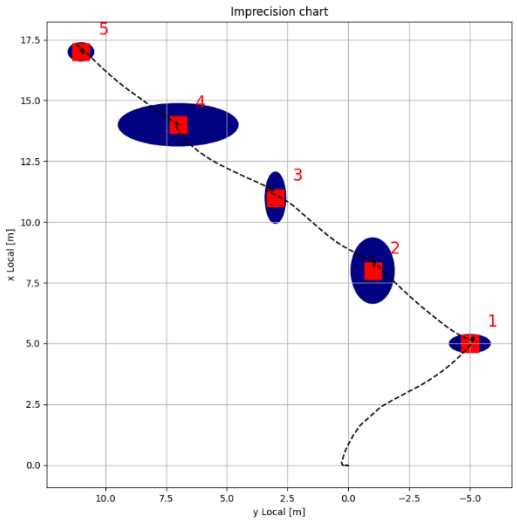


Figure 6.18: Top-down workspace graph.
Targets scaled, maximum errors in x and y, local coordinates, evaluated during the whole episode,
(--) UAV's trajectory.

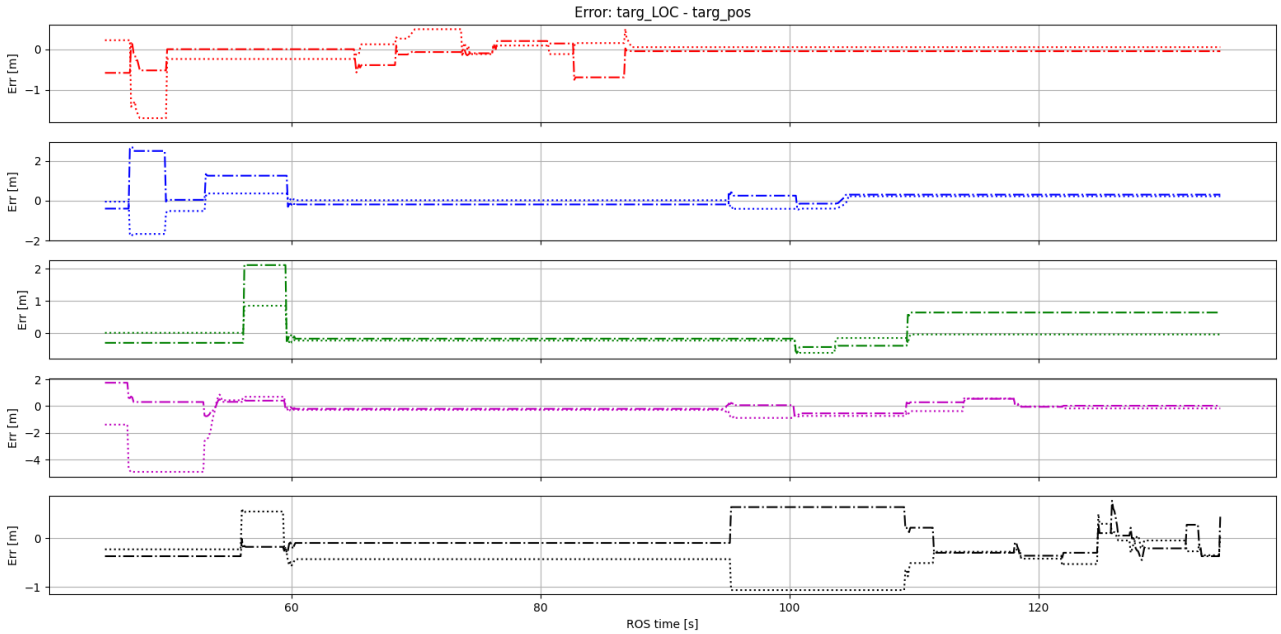


Figure 6.19: Error graph.
(-.-): error on x local target 1 [m], (···): error on y local target 1 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 2 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 3 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 4 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 5 [m]

6.2.3 Sparce targets noisy

This test is run like the one just presented, however targets positions are reset randomly in the workspace.

As before in figure 6.20, the normal distribution used to generate noise is reported. The normal distribution has the same $\sigma = 0.7$ as the previous case. Also, in this case the UAV is able to locate and reach each target in the workspace.

From these results emerges the allocation algorithm, in presence of noise in the position signal, struggles to allocate targets in the *targ_LOC* message. This happens because the noisy signal must be filtered by the *buffer_matrix*, which has to wait until a full line of similar values, within the tolerance *buff_toll*, is found. This ensures the generation of a correct value, with an error, of targets positions, however in the presence of noisy signal multiple values are discarded because not close enough to the previous registered one. The result is the allocation process, in presence of noise, is slower than the one without noise, this can be seen in figure 6.21, at around 50 – 60 s mark, where two targets are immediately located, whereas the others are identified after a certain delay.

Targets positions identified, in the noisy case, are more precise than the ones detected during the test without noise, proof of this can be seen in figures 6.22, 6.23. This happens because the noise distribution tends to average out other sources of errors like distance and the UAV dynamics that, in test without noise, produced most of the error. This relation is also noticeable in the previous test, however, is less evident.

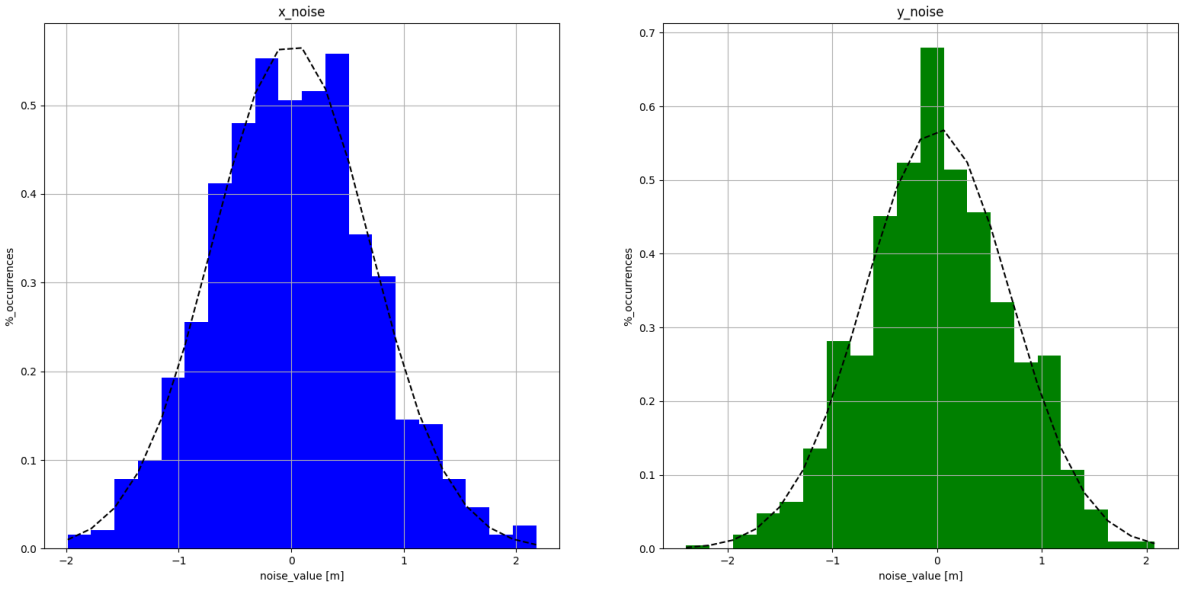


Figure 6.20: Normal distributions obtained as histograms for noise generation. On the y axis % of occurrences of the same value are shown. On the left side the x noise distribution, one right side the y noise distribution in local coordinates.
x local noise histogram [%], y local noise histogram [%], (--): corresponding analytical normal curve [%].

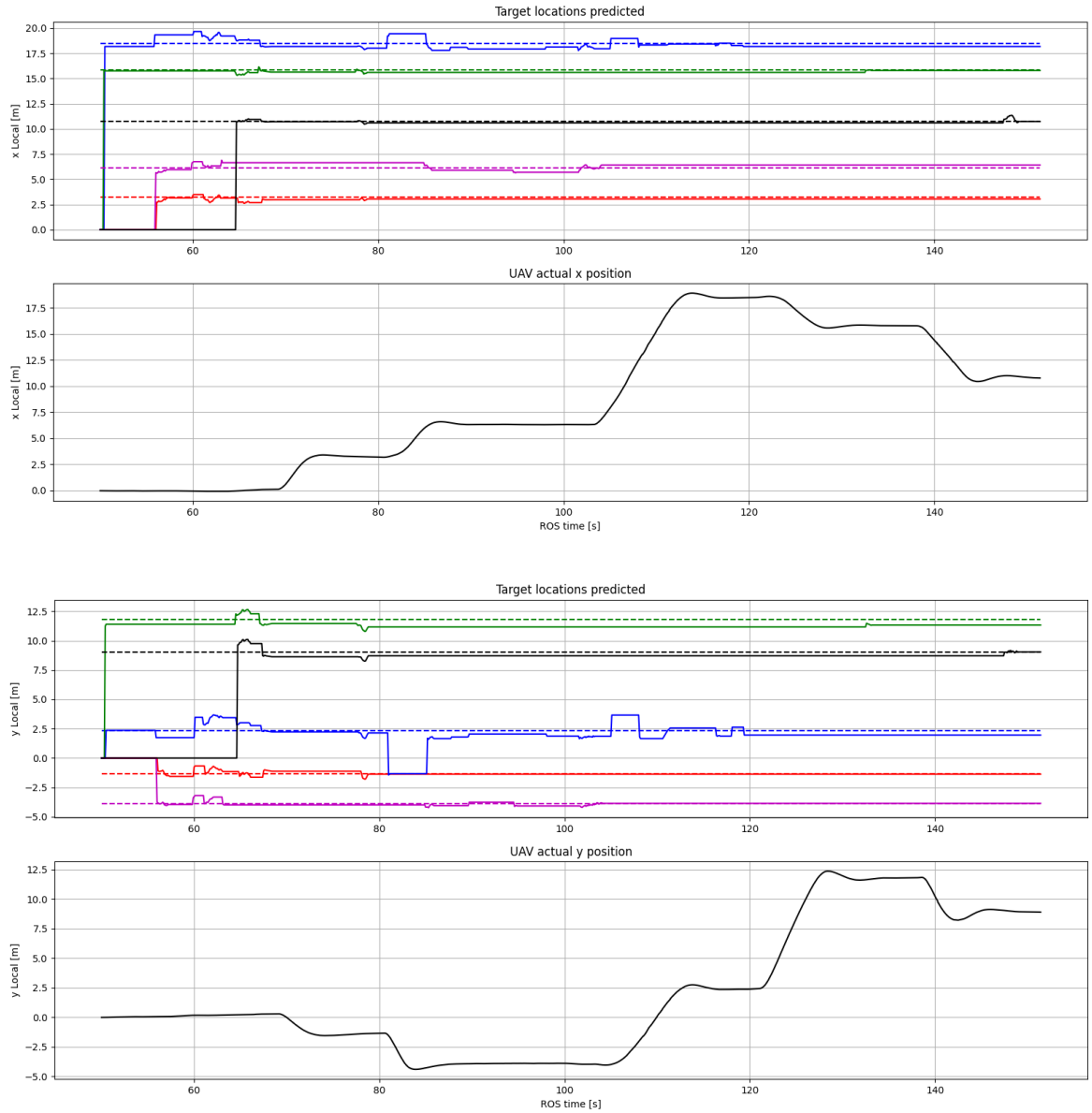


Figure 6.21: predicted target position and actual target position in x and y local coordinates. From the top, the first and the third graphs:
(—): predicted target 1 position [m], (--): actual target 1 position [m]
(—): predicted target 2 position [m], (--): actual target 2 position [m]
(—): predicted target 3 position [m], (--): actual target 3 position [m]
(—): predicted target 4 position [m], (--): actual target 4 position [m]
(—): predicted target 5 position [m], (--): actual target 5 position [m]

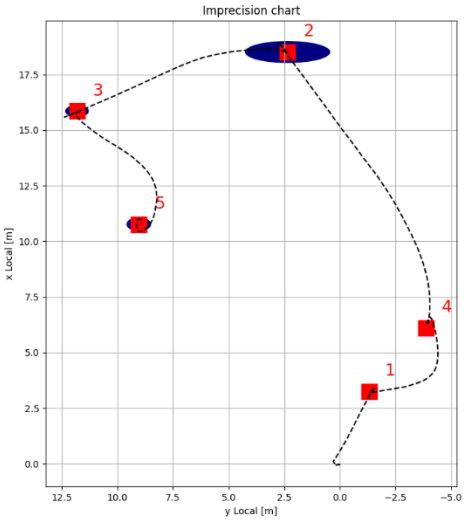


Figure 6.22: Top-down workspace graph.
Targets scaled, maximum errors in x and y, local coordinates, evaluated during the whole episode,
(--) UAV's trajectory.

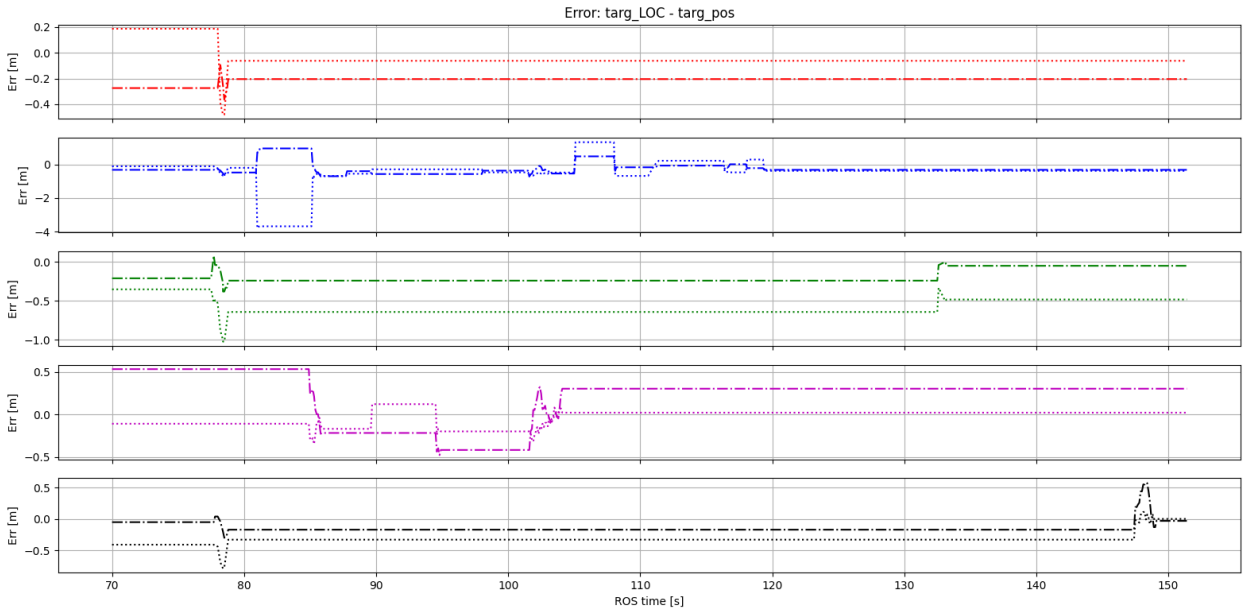


Figure 6.23: Error graph.
(-.-): error on x local target 1 [m], (···): error on y local target 1 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 2 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 3 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 4 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 5 [m]

6.2.4 Location error noisy

This test is run exactly like the previous one, however a value of $\sigma = 0.8$ is used to generate disturbances to the UAV position signal. Also $\sigma = 0.8$ is the threshold value identified for the noise generation, above it the allocation algorithm, in most tests, is not able to identify all the targets during the initial searching phase. Even with $\sigma = 0.8$ not all tests have a completely positive outcomes because in some of them not every target is correctly located. In this case the UAV manages to reach all the targets, however some issues arise.

Targets allocation in *targ_LOC* message is very slow, as shown in figure 6.24 the first target position is found at around 25 s mark, while the last is correctly located only after the 45 s mark.
In figure 6.26 the trajectory of the UAV is reported, as always along with maximum errors. It can be noticed that between target 2 and 5 the UAV follows a twisted path, the control system does not behave as expected. Target 5 is described in all the graphs by the black lines, in figure 6.24 and 6.27 it can be noticed the error with which the algorithm estimates its position is high. What happens is the UAV starts to move along its memorized position when target 5 is chosen, however, due to the intense noise in the position signal, as the UAV gets closer, the algorithm can't update quickly target 5's position. The algorithm is faced with conflicting information, a memorized value affected by great error and the inability to find a corresponding candidate to follow with only visual control system. This results in the UAV circling around the memorized target position, which is incorrect. In this case, eventually the algorithm is able to update target's 5 position after some time, thus managing to reach it.
In figure 6.28 y and yaw commands, from 100 s to 120 s marks, the timespan corresponding to the issue, are reported. The control system gives yaw rate commands to follow the memorized target.

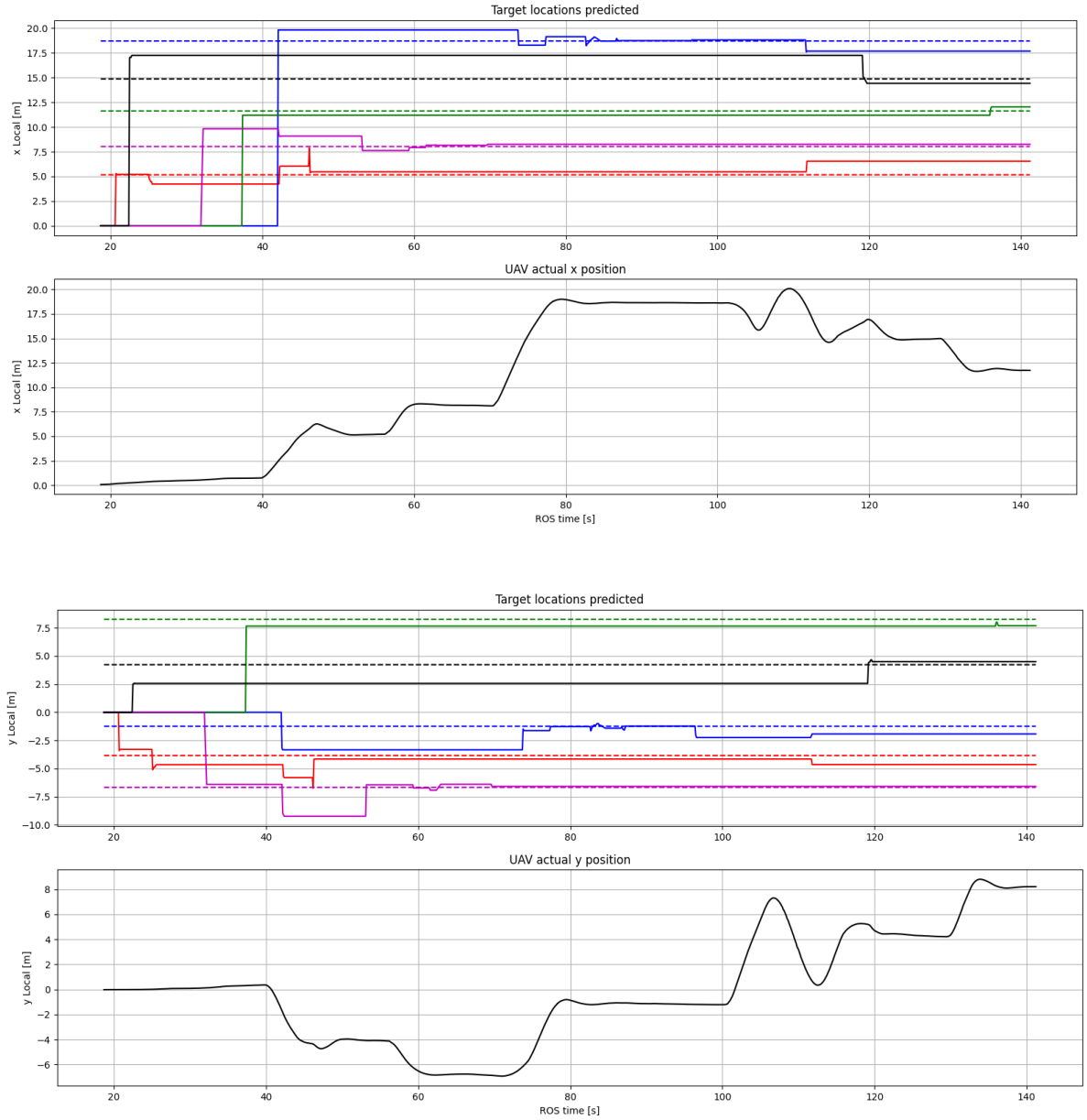


Figure 6.24: predicted target position and actual target position in x and y local coordinates. From the top, the first and the third graphs:
(—): predicted target 1 position [m], (--): actual target 1 position [m]
(—): predicted target 2 position [m], (--): actual target 2 position [m]
(—): predicted target 3 position [m], (--): actual target 3 position [m]
(—): predicted target 4 position [m], (--): actual target 4 position [m]
(—): predicted target 5 position [m], (--): actual target 5 position [m]

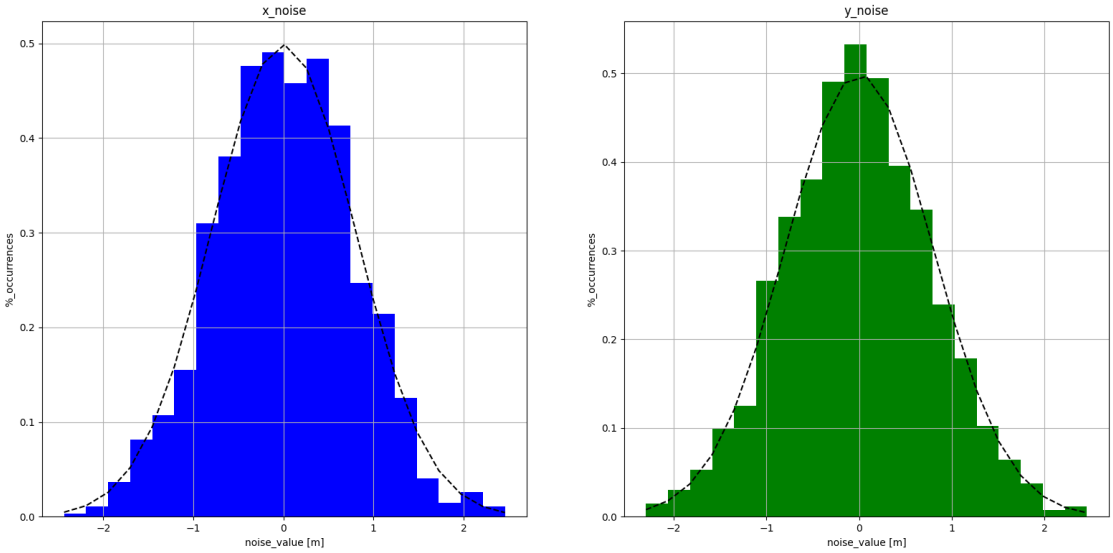


Figure 6.25: Normal distributions obtained as histograms for noise generation. On the y axis % of occurrences of the same value are shown. On the left side the x noise distribution, one right side the y noise distribution in local coordinates.
x local noise histogram [%], y local noise histogram [%], (--): corresponding analytical normal curve [%].

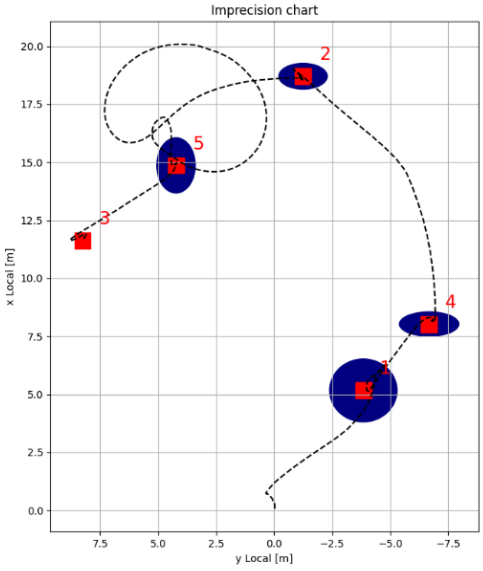


Figure 6.26: Top-down workspace graph.
Targets scaled, maximum errors in x and y, local coordinates, evaluated during the whole episode, (--) UAV's trajectory.

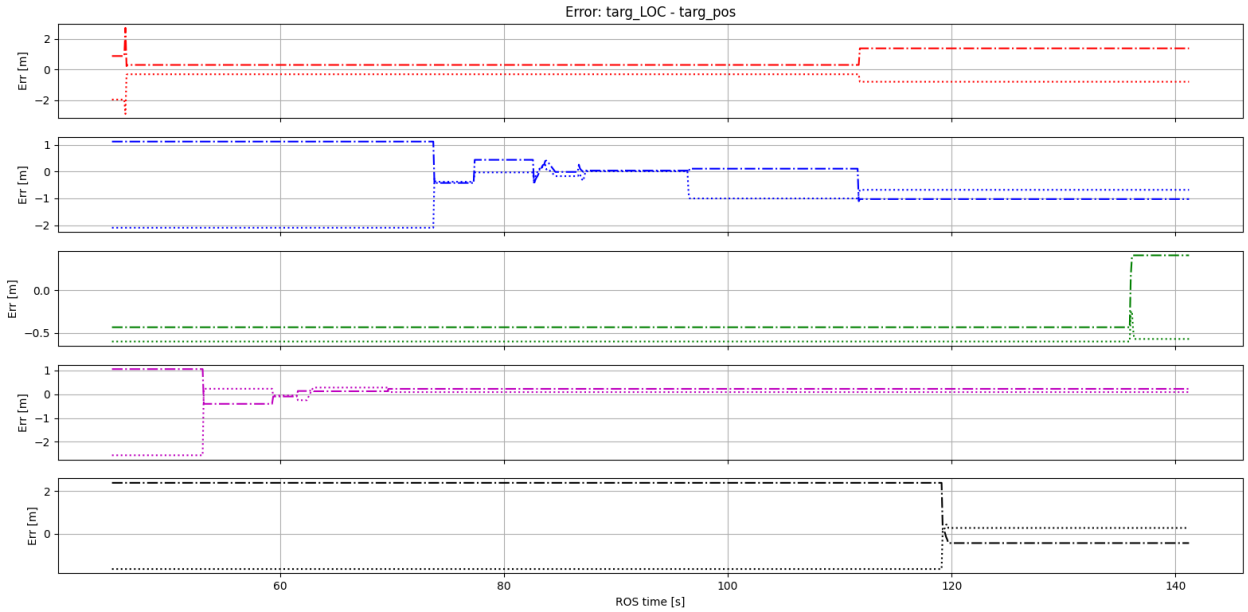


Figure 6.27: Error graph.
(-.-): error on x local target 1 [m], (···): error on y local target 1 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 2 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 3 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 4 [m]
(-.-): error on x local target 1 [m], (···): error on y local target 5 [m]

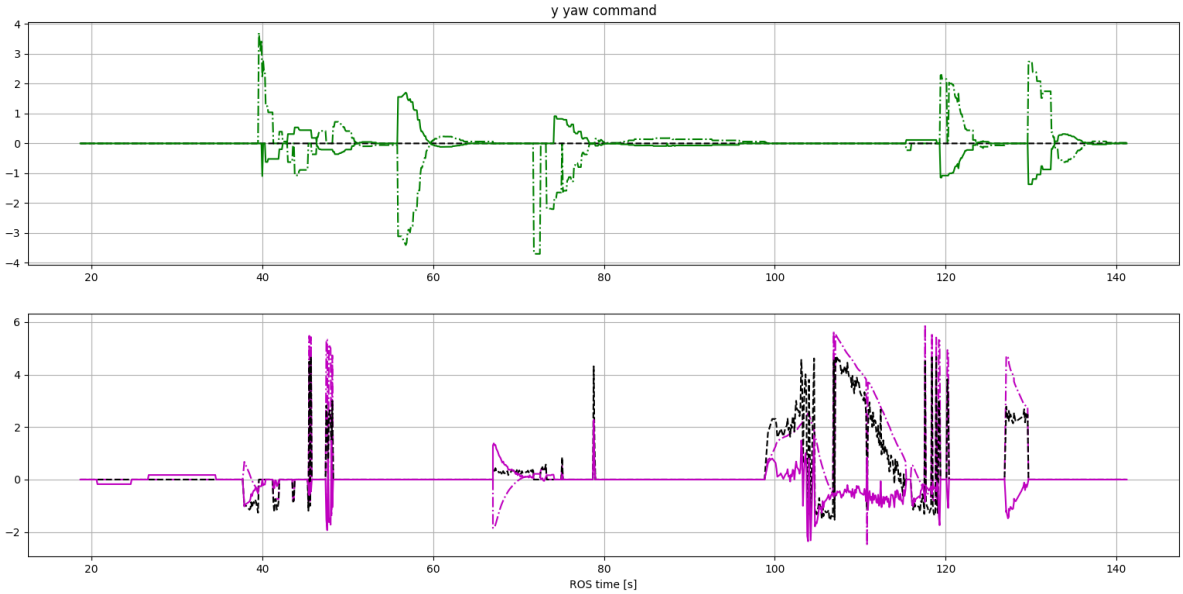


Figure 6.28: y and yaw rate commands. The first graph, figure a, represent commands to the y axis, while the second one, figure b, depicts commands in yaw rate.
(—): command on y axis [m/s], (--): x camera target state [m], (···): desired x camera target state [m]
(—): command in yaw rate [rad/s], (--): heading camera target state [rad], (···): desired yaw camera target state [m]

6.3 RL policy

This section explains the process of model training and evaluation in RL and shows the results.

6.3.1 Training and evaluation execution

Unlike before, since the aim of this section is to train and test the RL model policy, no noise is added to the position message. This is done to ensure that during the iterations of subsequent episodes there are no errors in the evaluation of the target positions, which could lead to either an incorrect update of the policy or, in the worst case, a complete simulation stop.
Another major difference from the deterministic policy section is that the target positions are now pre-initialised in the *targ_LOC* message. This is again done to avoid the algorithm not finding targets or misplacing them. This means that only the effectiveness of the RL algorithm is now tested, as the performance of the allocation algorithm has already been evaluated.
As a side note, although *targ_LOC* is pre-initialised, the allocation algorithm continues to update their positions throughout the simulation.
When the test is started the *reset()* function is called, which, performs the following actions:

- Targets are moved to a random generated location in the workspace limits.
- *state_cam* (*state*) message is reset with [0, 0] in all the rows.
- *targ_LOC* is reset with (x, y) target position in rows.

After these actions are completed, the UAV starts to follow the target, that gets chosen based on RL algorithm learning method if it is in the training phase, and with the best value action if it is during the evaluation phase.

6.3.2 Training

The training process is carried out in a total of 600 episodes and with parameters reported in tables 5.4, 5.5.

In figure 6.29 the cumulative reward across episodes is shown. In most episodes at the start of the training the UAV does not correctly complete the mission as chooses almost always wrong targets, the ones that has already reached, thus resulting in multiple negative rewards. after the first 100-200 episodes, through the updating policy process, the algorithm learns to never choose the same target twice thus completing the mission in most of the next episodes. In the last episodes the policy is able to consistently complete the mission, thus reaches every time the final reward, based on time. This allows to update the policy optimizing actions so that a lower mission time can be obtained. The training process is stable, but after 300 episodes is far slower seemingly reaching a plateau on reward across episodes.
In figure 6.30 a colormap representing rewards for each episode is reported. Episodes that are not completed only have negative rewards, dark coloured, whereas green rewards are positive ones, awarded after the mission is complete. The brighter the green colouring is the higher the reward is.

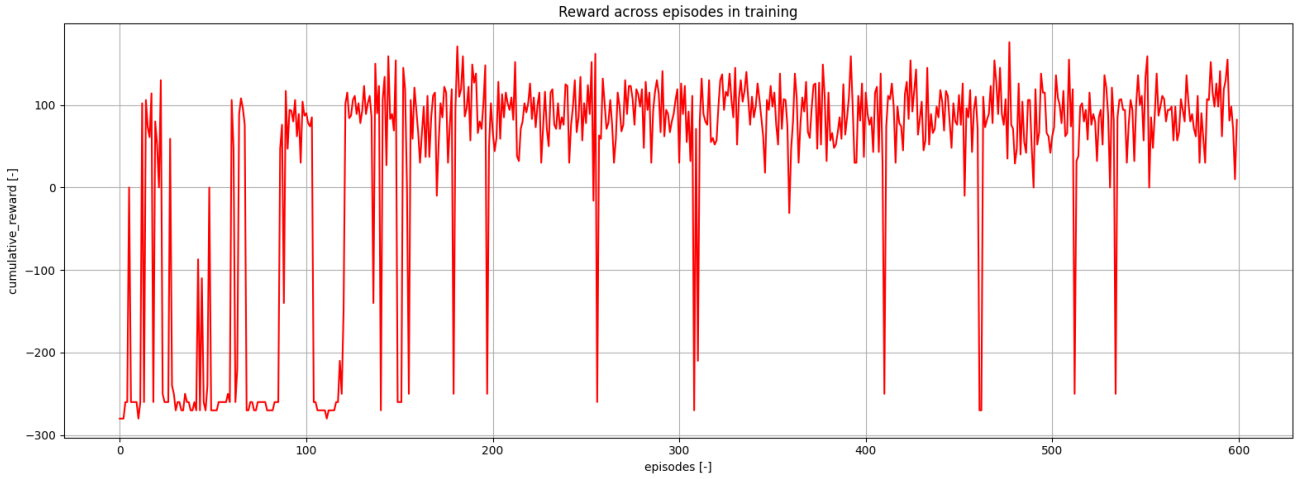


Figure 6.29: cumulative rewards across episodes during the training process.

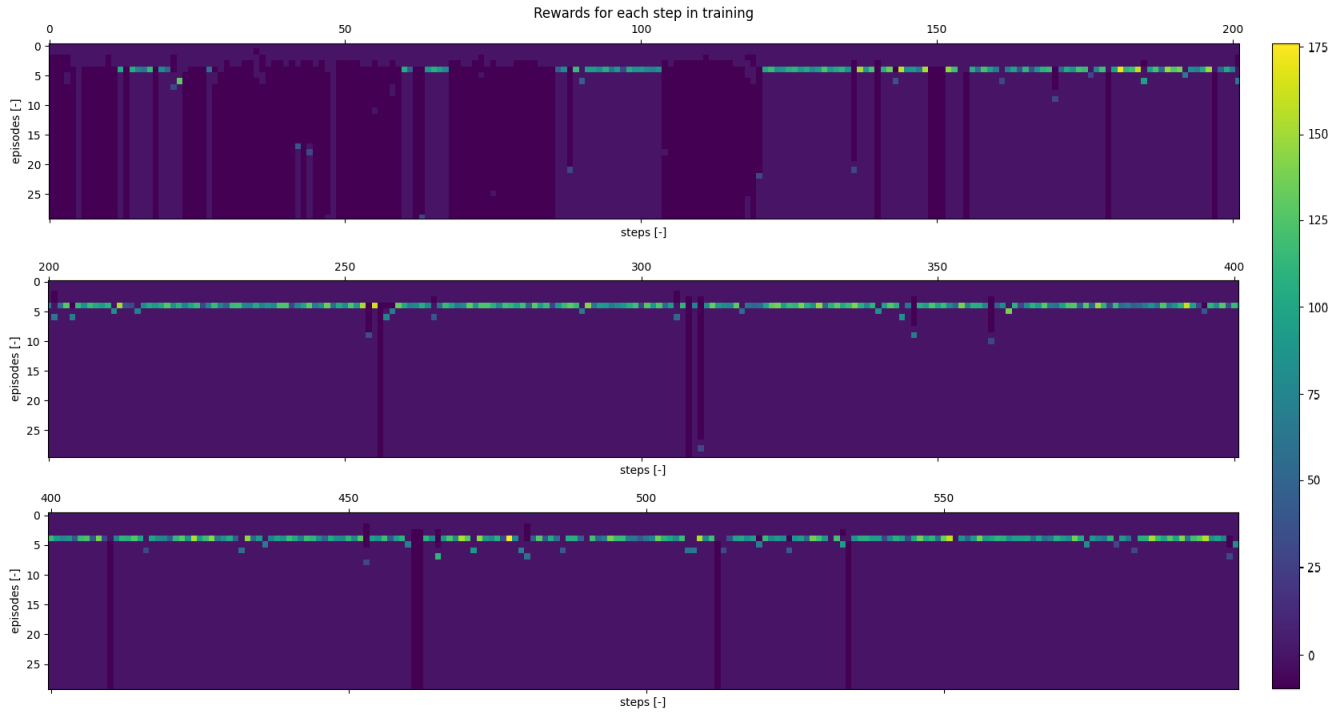


Figure 6.30: colormap representation of rewards at each step during the training process.

6.3.3 Evaluation

It's not easy to evaluate the trained policy performances since the target position change implies that, both in test and in the evaluation, some episodes have a reward cap higher than others. However, from evaluation episodes and results graphs, figure 6.31, observations, several conclusions can be drawn.

First of all, the policy learns to choose at every step targets that have not been already reached, this means the UAV does not pass two times over the same target. This result is achieved primarily through the flagging of reached targets rows in state with [100, 100], and since $\pi = P(a|s)$, making it easier to understand still available targets.

With previous trainings the policy used to learn to choose the same action sequence at every episode, independently of the state. With this training, however, this does not happen, and a different action sequence is chosen for every different episode. This means the algorithm actively choses actions on the state basis, which is the correct behaviour.

The best result should be a policy which chooses, accordingly to the state, the action sequence which minimizes the episode time. This is not completely achieved, as some different behaviour occurs. The policy chooses consistently as the first or the first two targets the ones closer to the starting point. This behaviour is beneficial in every episode. The next choices after the first, or the first two, are sometimes based on target groupings. The algorithm chooses the successive target that is in the general area of the UAV actual position.

Although this is not an optimal solution is the best one achieved and brings every time high value reward. However, some other times the next actions seem to be random and, although the UAV manages to reach all targets, episode time is high thus achieving a lower reward. This behaviour can be caused by the training, perhaps the algorithm did not learn to generalize some target configurations thus it does not develop the ability to adapt in these cases. In some rare episodes the UAV can't complete the objective, this also happens in the case a target configuration is very different to the ones generated during training.

A comparison between performances of RL algorithm and the application of the deterministic policy can be done. Results, in term of reward across episodes, are reported in the already discussed graph in figure 6.31, whereas results regarding the deterministic policy case evaluation are shown in figure 6.32. The average cumulative reward after an evaluation of 100 episodes is 77.41 for the RL algorithm, whereas the same result for the deterministic policy case is 141.29. Although the average in the case of the RL is lowered by four occurrences of failed missions, the application of the deterministic policy almost always performs much better in term of time spent at each episode.

Figure 6.31: reward across episodes during the evaluation phase.

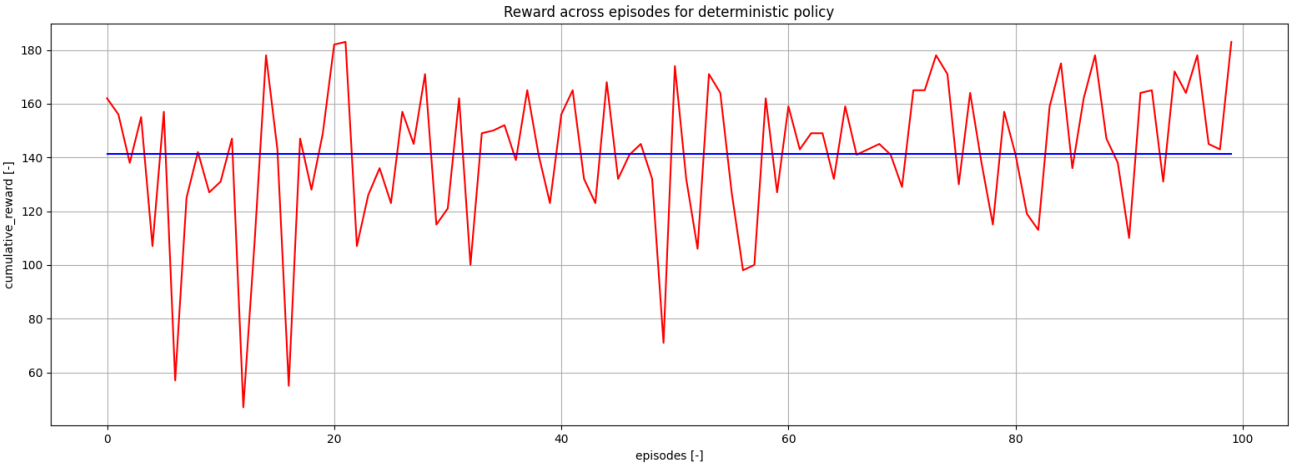


Figure 6.32: cumulative reward across episodes using deterministic policy.

7 Conclusions

In this final section, the full performance of the algorithm is described and a brief section on possible future work is reported.

7.1 Performances and future work

7.1.1 Performances

As for the implemented control system described in chapter 5, it manages to control the UAV by giving commands to the FCU in the form of an outer loop with respect to the PX4 autopilot, achieving the desired results. It is able to give effective commands both when the frontal camera has a direct line of sight to the selected target, and when commands are generated using memorised target positions. The control method switching is effective in managing the UAV's controls when it's far from the target and when it's close to it. The positioning error is less than 0.07 m in both the x and y axis with respect to the camera frame, although it can easily be reduced, but then more time is needed for the UAV to meet with the required new tolerance.

For the allocation algorithm, tests without noise show that it is capable of quickly storing target positions with errors similar to those shown in the graphs reported in chapter 6.2. The error depends mainly on the distance to the target, the further away the target, the higher the error, and on the dynamics of the UAV during flight. Some errors generated during the pipeline can accumulate, such as the superposition of targets during the recognition task. However, even this phenomenon rarely affects the outcome of the mission.
The UAV command response generates oscillations and changing in attitude which can result in a bigger error. However, the *buffer_matrix* implemented works well in mitigating these two sources of errors thus making the allocation algorithm robust at localizing targets each time.
Parameters which have a great effect on the allocation algorithm performances are *buff_toll*, *targ_LOC_toll*, and *buffer_matrix_lenght. buff_toll* and *targ_LOC_toll* values should be selected depending on minimum distance between targets, whereas the greater *buffer_matrix_lenght* is the more precise prediction on target locations are made, but the longer it takes for the algorithm to pick up new targets.
For now, the algorithm works best if all targets are seen and located before starting to reach them, as issues like the target switching discussed in chapter 6.4.1 can cause the mission to fail.
From tests done with noise, considering a normal distribution used to generate the disturbance signal, emerges the allocation algorithm is able to locate targets correctly also in noisy condition with a normal curve described with $\mu = 0,\ \sigma < 0.8$.

With the noisy signal is more difficult to pick up new targets, delays of several seconds have been found during test between the visual contact of the target to the memorization of its position. Also, once the target is located, errors are smaller than the ones in the case without noise, as explained in chapter 6.2.3.

The idea of having two parts of the whole algorithm which one memorizes targets, while the other one outputs only visual information and both are used in the control system, allowed the UAV to complete the mission in both policy cases. Thus, making this idea the key which allowed the designing of the whole project.

The RL policy obtained, based on a fully connected neural network, is able to correctly reach all targets scattered in the environment thus completing the mission but, in terms of time spent, in a not optimal way. The deterministic target choosing method, although it does not minimize the mission time, it is a great compromise between complexity and performances and, compared to the RL policy, achieves better results for now.
Through the design process of the RL methods some critical issues regarding the Gazebo simulator have been encountered:

- Large computing power required: Gazebo is a very precise simulator, representing with fidelity sensors behaviour, UAV dynamics, autopilots modes etc. this makes the simulation demanding and heavy to run. A simpler and more lightweight simulator would have been more effective for RL training as simulations can be run much faster than real time, reducing training time, which was not an option with Gazebo.
- Realistic sensor and autopilot behaviour: Gazebo simulates with fidelity sensors and the autopilot operations. This implies the need to implement autopilot mode switching, realistic tuned control systems, and other real UAV functionalities just to train the model. This results in a complex RL environment with a high probability of errors and warnings from the simulator, which are not always easy to assess and eliminate. Also, during simulations, are experienced different delays from when the UAV is on top of the target to the declaration of target reached. This depends on control system gains and UAV's dynamics, although is a real effect, is not beneficial for the RL training.
- No full reset possible: Once the simulation starts, is not possible to shut down and restart the whole simulation. Some other methods must be implemented like the one utilized, which consists in return the UAV back and move models. However, this method can cause issues, for instance if the UAV crashes into and environment model is not possible to reset and proceed the simulation.

Although the Gazebo simulator has been used with partial success, for these reasons is not suitable for effectively training RL models.

7.1.2 Future work

Starting from the U-Net utilized for target recognition, the next step is implementing in Gazebo realistic plants models and train the NN so that these models are recognized. Also, a catalogue of plants can be implemented, where different masks, depending on plant classification, are generated, so that only target plants of choice for the mission are reached and treated with appropriate products. This is feasible as the original U-Net in (Olaf Ronneberger, 2015) is able to generalize well also in the presence of multiple categories, although some modifications are needed. For instance, a CNN is utilized in the work of (Hehu Zhang, 2019) for foliage recognition of plants, in which useful CNN training advice are given.

Although not optimal, the deterministic policy assures best performances, the next step would be to implement the algorithm with the deterministic policy in a real scenario with hardware sensors and UAV. This is useful to evaluate actual performances and apply modifications to the algorithm to account for real effects. This also represents the last step of the design process.

The RL algorithm, although its performances are worse than the deterministic policy, is promising anyway. First, a new simpler simulator should be coded without real autopilot and sensors interaction, but that can represent an environment, for the sole purpose to train the RL model quickly. This way is also possible to run trainings in a brief time but with more episodes without errors, allowing to change hyperparameters quickly and try multiple different solutions in a fast way. Also, some environment agent interactions are understandable by the RL policy only after a much higher episode number than 600, depending heavily on the task. Then the trained policy should be implemented and tested in the Gazebo simulator to assess autopilot, sensors, and environment real effects interaction with precision.
RL allows to implement also other solutions, such as obstacle avoidance and other functionalities in an organic manner with modifications of the already coded environment, this also requires a different reward set and multiple trainings.

8 References

Barto, R. S. (2015). Reinforcement Learning: An Introduction.

Diederik P. Kingma, J. L. (2015). Adam: A method fo stochastic optimization. pp 1-2.

Fiaz Ahmad, M. S. (2021). Advancements of Spraying Technology in Agriculture. *ResearchGate*, pp 1-3, pp 8-9.

Gazebo homepage. (s.d.). Tratto da gazebosim.org: https://gazebosim.org/home

George Cybenko, D. P. (1999). The Mathematics of Information Coding, Extraction and Distribution.

Gimp homepage. (s.d.). Tratto da gimp.org: https://www.gimp.org/

Hehu Zhang, X. W. (2019). Research on Vision-Based Navigation for Plant Protection UAV under the Near Color Background. *Symmetry*.

In-Su Lee, L. G. (2005). The performance of RTK-GPS for surveying under challenging environmental conditions. *Earth Planets Space*.

Keras homepage. (s.d.). Tratto da keras.io: https://keras.io/

MAVLink Developer Guide. (s.d.). Tratto da mavlink.io: https://mavlink.io/en/

mavlink package summary. (s.d.). Tratto da ROS.org: http://wiki.ros.org/mavlink

mavros custom modes. (s.d.). Tratto da ROS.org: http://wiki.ros.org/mavros/CustomModes

mavros package summary. (s.d.). Tratto da ROS.org: http://wiki.ros.org/mavros

Olaf Ronneberger, P. F. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. pp 1-3.

OpenCV homepage. (s.d.). Tratto da OpenCV.org: https://opencv.org/

Pérez-Ruiz, M. (2012). GNSS in precision agricultural operation. *ResearchGate*, pp 11-13, pp 17.

Pietro Zanuttigh. (2021). *Machine Learning lectures*. Univeristà degli Studi di Padova, Padova.

px4 user guide homepage. (s.d.). Tratto da px4.io: https://docs.px4.io/main/en/

ROS homepage. (s.d.). Tratto da ROS.org: https://www.ros.org/

rviz package summary. (s.d.). Tratto da ROS.org: http://wiki.ros.org/rviz

Satoshi Suzuki, K. A. (1985). Topological structural analysis of digitalized binary images by border following. pp 32-39.

Tensorflow homepage. (s.d.). Tratto da tensorflow.org: https://www.tensorflow.org/?hl=it

Tensorforce documentation. (s.d.). Tratto da tensorforce.readthedocs.io: https://tensorforce.readthedocs.io/en/latest/

tkinter — Python interface to Tcl/Tk. (s.d.). Tratto da python.org: https://docs.python.org/3/library/tkinter.html

Vasko Sazdovski, T. K.-G. (2005). Kalman filter implmenetation for unmannedaerial vehicles developed withn a graduate course. *Elsevier Publications*, pp 12, pp 14.

VGG image annotator webapp. (s.d.). Tratto da University of Oxford INformation Engineering: https://www.robots.ox.ac.uk/~vgg/software/via/via.html

Volodymyr Mnih, K. K. (2015). Human-level control through deep reinforcement learning. *Letter*.