DIPARTIMENTO DI SCIENZE E SCIENZA DELL'INFORMAZIONE

CORSO DI LAUREA IN: INGEGNERIA INFORMATICA

Autonomous Quadcopter Flight

with PX4 and ROS2

THESIS TYPE: Experimental

Studente:

Massimo Girardelli

Relatore:

Prof./Dott. Luigi Palopoli

ACADEMIC YEAR 2019/2020

Abstract

This thesis is the design and development of a software application for the control of quadcopters and other flying drones.

Quadcopters have gained more attention in the past years due to their versatility and potential in various fields, including surveillance, aerial photography, and environmental monitoring.

Although ready to use solutions and similar open source projects exist, an easily configurable, with a graphical interface and indoor capabilities that responds to external inputs doesn't and this project aims to provide a working template that can be customized for different applications.

Controls can be sent by the GUI by people or via ROS2 messages

The end goal of the project is a proof of concept of all the different components working together, the ROS interface allows seamingless integration with many other devices in the robotic world.

This drone is capable of indoor flying with position provided by external beacons

SOMMARIO

ELENCO DELLE TABELLE

ELENCO DELLE FIGURE

ACRONYMS, ABBREVIATIONS AND IMPORTANT TERMS

PX4	PX4 is the Professional Autopilot. Developed by world-class developers from industry and academia, and supported by an active world wide community, it powers all kinds of vehicles from racing and cargo drones through to ground vehicles and submersibles.
ROS2	robot Operating System version 2, middleware that enables fast and easy messages communications between different nodes on a network
NED	orth-East-Down coordinate system where the X-axis is pointing towards the true North, the Y-axis is pointing East and the Z-axis is pointing Down, completing the right-hand rule.
FRD	ront-Right-Dwon coordinate system where the X-axis is pointing towards the Front of the vehicle, the Y-axis is pointing Right and the Z-axis is pointing Down, completing the right-hand rule.
SITL	Software in the Loop, a simulation tool that reproduces the flight controller behavior
LTS	“Long Term Support”, an LTS version of software recives supports for many years to come and thus is more preferred for compatibility
GUI	Graphical User Interface
OptiTrack	Motion capture system, used to detect the drone position with a tracker
Commander	Main computer from where the drone is controlled
Companion	Single board computer mounted on the drone

INTRODUCTION AND AIM OF THE THESIS

The aim of this thesis is to provide a system that allows drones to fly with the support of external position detection systems, in this case with Optitrack, Motion Capture Systems, ROS2, Pixhack flight controller and Rockchip ROCk 4 C+ board.

This thesis also provides an easily employable simulation environment to test the different flight options preseted by the commander

Thesis organization

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

• Hardware employed pros, cons and limitations

• Software employed introduction: description of the system used and all the software for this project

• Deployment: steps to take to deploy a system like this

• Simulation results: the proposed algorithms are tested through a SITL simulation, with the Gazebo simulator.

- Demo
- Flying results results: flight performance, obstacle, battery life,

CHAPTER I
HARDWARE:

This is the list of components used and some reasons behind the choices that were made

- **Holybro X500 V2 Development Kit**
  Main frame of the drone, it's a good all-in-one solution for a quadcopter, it mounts 4 920 Kv motors, perfect for this application, lower Kv motors offer more torque and less fan speed, which is desired in this application give the weight of the whole system, a higher Kv motor would strugle more.

  The Pixhawk 6C flight controller offers the right connectivity that we need for this esoteric application and easily provides the CPU and RAM taht we need.

- **5000 mAh Li-Po batteries**

  A massive 5000 mAh Li-Po battery powers the motors, the flight controller and the companion computer.

  The 4S configuration offers 14.8V, when charged it can reach 16.8V, the drone will strugle to lift below 14V.

  The exact model used is: "Dinogy Ultra Graphene 4S 5000mAh 80C"

- **ROCK 4 C+ (Companion)**
  A single board computer, similar to a Raspberry PI but with a 64-bit architecture.

  It's a hexacore ARM processor with 4GB of LPDDR4 RAM, which is plenty for our needs.

  The GPIO PINS are used for serial communication with the flight controller

  Connection is done with Wifi and a SSH shell and power is delivered via USB

- **Commander computer**
  Any computer that can run the commander programs, it's possible to use the Companion itself but there are no guarantees of recovering the system in case of disconnection.

- **OptiTrack**
  3D Motion Capture Systems mounted in the laboratory, the precision is <1 cm, latency is around 30ms, publish rate is 360Hz by default but the drone remains stable up to 50 Hz of data recieved in case of network congestion.

The 9 cameras placed around the flying area, provide an accurate position estimation. The position is sent through the network and must be decoded by a client.

CHAPTER II
SOFTWARE:

In this software chapter I will explain the systems and tools used to bring this project to life and the inner workings of the components to kae the drone fly.

System tools:

1. **Ubuntu 22**

   This was a very important decision since every distribution of Ubuntu has it's own ROS2 distribution and the latest LTS (for now) only supports Ubuntu 22

2. **ROS2 Humble**

   Middleware used to share all messages between applications and to launch most of the applications, ROS resolves around Nodes, Topics and Messages:

   - **Nodes:** a process interfacing with the ROS libraries, different nodes can run on different machines while being connected through networking and be visible to each other.
   - **Topics:** a pipe-like virtual structure within ROS that can be used to share messages between different nodes, it can be interfaced as a publisher or as a subscriber, each topic routes one specific message type.
   - **Messages:** data structures that travel through the nodes, it is not system-dependent and provides primitive types like booleans, integers, floats, arrays and nested messages types.
     Custom messages were built for this projec

   The optimal QoS (Quality of service) options were used:

   - Best effort, to reduce network congestion, most messages are ephemeral and gets overriden a few milliseconds later
   - Transient local: the publisher is responsible for delivering the last message tho subscribers who joined after it was published
   - Keep last: only keeps the last message, previous values are invalid anyway

   The **humble** distrubution was used as it was the LTS version supported at the moment of developing the project

3. **Python 3.10.12**

Python was used for the ease of developement and speed of deployment as it doesn't need compilation; python with PX4 has also much more documentation and examples than any other language.

Other programming languages that have better performance like c++ were considered but discarted due to minimal documentation, much more devloping time and a very slow overall deploment experience and unpleasant experience:

- ROS2 compilations have massive overheads, even for well written cmake files while python has none;
- a part of the code runs on the companion computer that, while powerful, is not really suited for continuous compilations

  (installing all the PX4 dependencies and building the optitrack package took more that 1 hour and a half)

The runtime overhead of python gets completly eclipsed by the network delay.

The version **3.10.12** is the offically supported by ROS2 Humble

4. **PX4**

   Open source flight control solution for drones.

   ROS is used by PX4 to provide an offboard control functionality, with a linux companion computer. In this way its possible to control the PX4 flight stack using a software outside of the autopilot. PX4 supports both versions of the robotic operating system, but while the communication between ROS 2 and the autopilot is done through a ROS2-PX4 bridge (MicroXRCEAgent).

   There are 2 ways to connect and read messages: ROS2-PX4 bridge or MavLink

   MavLik has a sligh different notation and needs it own libraries and messages while the ROS2-PX4 bridge only requires sourcing the message (done automatically when sourcing the whole ROS enviroment).

   The custom PX4 messages are used by most of the software used.

This list contains the 3 macro components used (Companion, Commander and Tracking) and software to launch to the whole system:

1. **Companion**

   The role of the companion is to run the **MicroXRCEAgent** service, the command to run is:

   ```
   sudo MicroXRCEAgent serial --dev /dev/ttyS2 -b 921600
   ```

   The client starts a ROS2 node that published topics with informations like current position, velocity, status, and subscribes to other topics to recieve commands, tracked position and keep-alive signals.

Communication with the flight controller is done on the serial port ttyS2 with a baud rate of 921600 to transmit all the data.

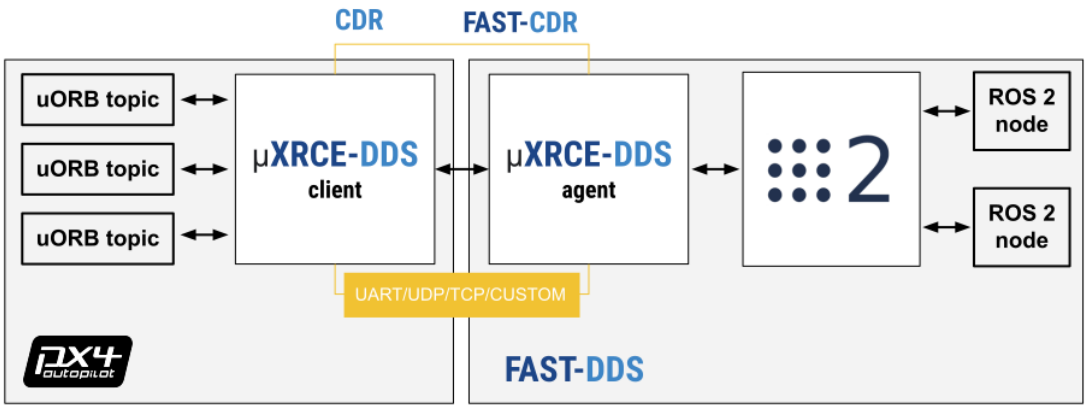


Figure 1: Flight controller on the left and companion pc on the right

For this particulat usecase a custom firmware was build to access particular informations like land detection, battery status.

2. Commander

The commander computer is responsible for running the following processes:

- **px4-py.py**

  (also known as controller from her on) the main process of the system, sends heartbeat signals to the companion to keep it alive, recives updates on position, velocity, status etc. from the companion, recieves data from the GUI, and most importantly elaborates all the data and messages to produce an action to send to the drone.

  This process is based on callbacks when a new message is detected, the main class OffboardControl(Node) keeps a record of the lates recieved.

  If there is no GUI (set in config.py), it procedes with the selected mode in the configuration file, otherwise it listens for action messages.

  Some actions are simply redirected to the companion (like arm, disarm, shutdown), other action that continue over time start a thread that compute and published mesasges to the companion over time and is stopped if a different a different command is sent.

  This this design the main thread constantly updates his own varables based on the lates message published by the companion, while every action over time is his own thread that based on the current status of the drone and can be stopped in any moment.

- **Gui.py**

  This graphical interface is a quality of life tool that considerably speeds up testing and validation since it makes possible to interface (in a human mode) with the controller, enabling runtime changes to the operating mode without editing and restarting the whole system.

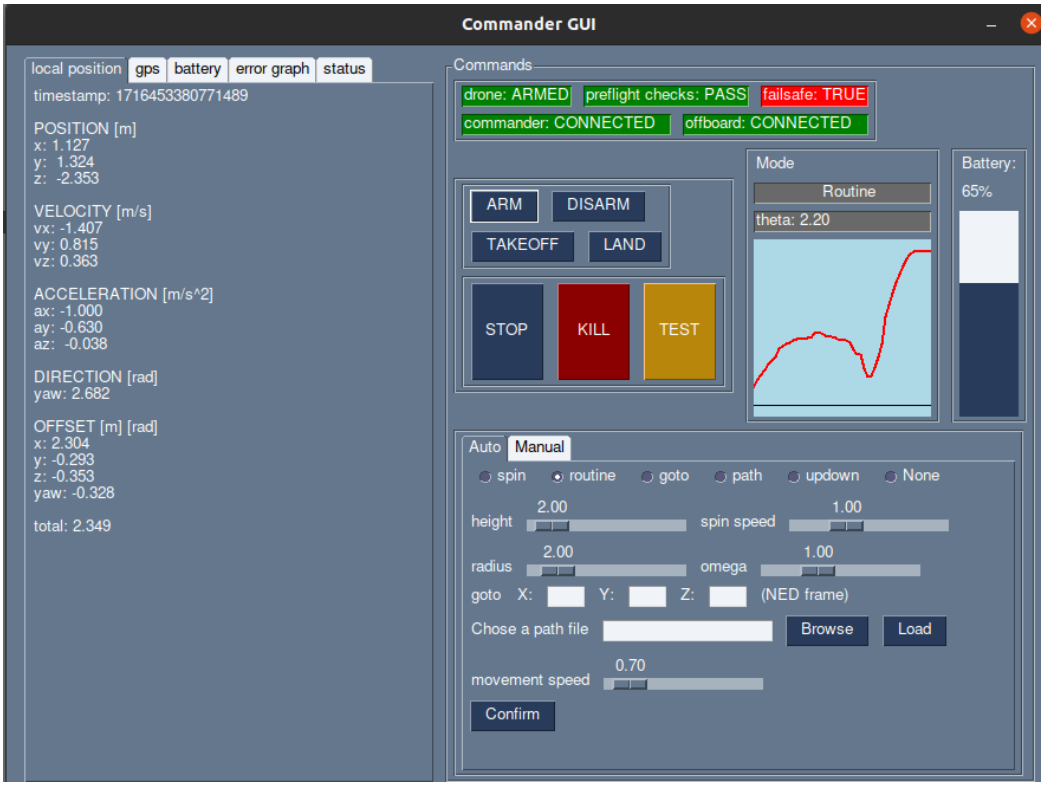


Figure 2: gui.py

It has many functionalities and offers all the relevant information on screen:

- **ARM** arms the drone if the above contitions are met
- **DISARM** disarms the drone before it takes off or when it landed, does not work mid flight
- **TAKEOFF** takes of to the height declared in src/config.py by the variable takeoff_height
- **LAND** lands at (0,0) local frame, position cannot be overriden, this is a limitation of the offboard mode because it would require a different mode to land normally on locaton, disconnecting from the commander
- **STOP** stops any activity and hold the position
- **TEST** button to test features, currently lands on a random position
- **KILL** emergency stop, terminates the flight immediately and stops the propellers, even mid air

USE WITH CAUTION

- **SPIN** spin on self with omega = spin speed slider
- **ROUTINE** starts hovering in circle around (0,0), at height = height , omega = omega slider,radius = radius
- **GOTO** goes to position x,y,z in NED frame (CAUTION WITH Z COORDINATES)
- **PATH** load a path file like src/medium.json and the drone fill follow all points with speed = moviment speed slider
- **UPDOWN** goes up and lands, mainly for testing
- **NONE** default, no activity, the drone will hover
- **GRAPH** shows arudimentary graph of position offset between the set position and the current position, gives a rough estimate of the offset

other commands and tabs are self explanatory

The gui and the controller comunicate with custom Commander messages made by me, the style is similar to the original PX4 messages with an integer defining the mode and a series of arrays where to pass thee data, it’s upto the reciver to correctly parsing the meaning

```
uint64 timestamp

#bool ready

#check config.py for modes
uint32 mode

bool ready

##description of parameters for each mode
# routine f1=omega, f2=radius, f3 = height
# path points = array of points
# spin f1 = omega, f2 = height
# updown no params
# goto fa1 xyz, f1 yaw, f2 speed
# landing fa1 xyz

float32 f1
float32 f2
float32 f3
float32 f4

float32[] fa1
float32[] fa2
float32[] fa3

commander_msg/CommanderPathPoint[] points
#int8 path_index

bool b1
bool b2
bool b3

int32 i1
int32 i2
int32 i3

```

Figure 3: Comamnder message for mode selection

This program is independent and the whole system can work without this piece of software since it doesn't comunicate directly with the companion but only to the commander, this is a design choice to improve compatibility and automation.

Every information is accessible with the command line or a program with a ROS interface and every action is replicable with a script

3. Tracking

The tracking component is needed for indoor flight and is expected by default, GPS flight need a change in the parameters of the flight controller

A previously mentioned the tracking system used for this project is the **OptiTrack** as it offers a sub-centimeter precision when correctly calibrated, but any other tracking device can be used as long as it can publish ROS pose messages

The software needed to run the tracking system are:

- **Motive**

  The proprietary software that is physically attached to the cameras, it published multicast messages int he network, must be configured with the shape of the tracker used.

  Must be interfaced witha client to acecs the date published.

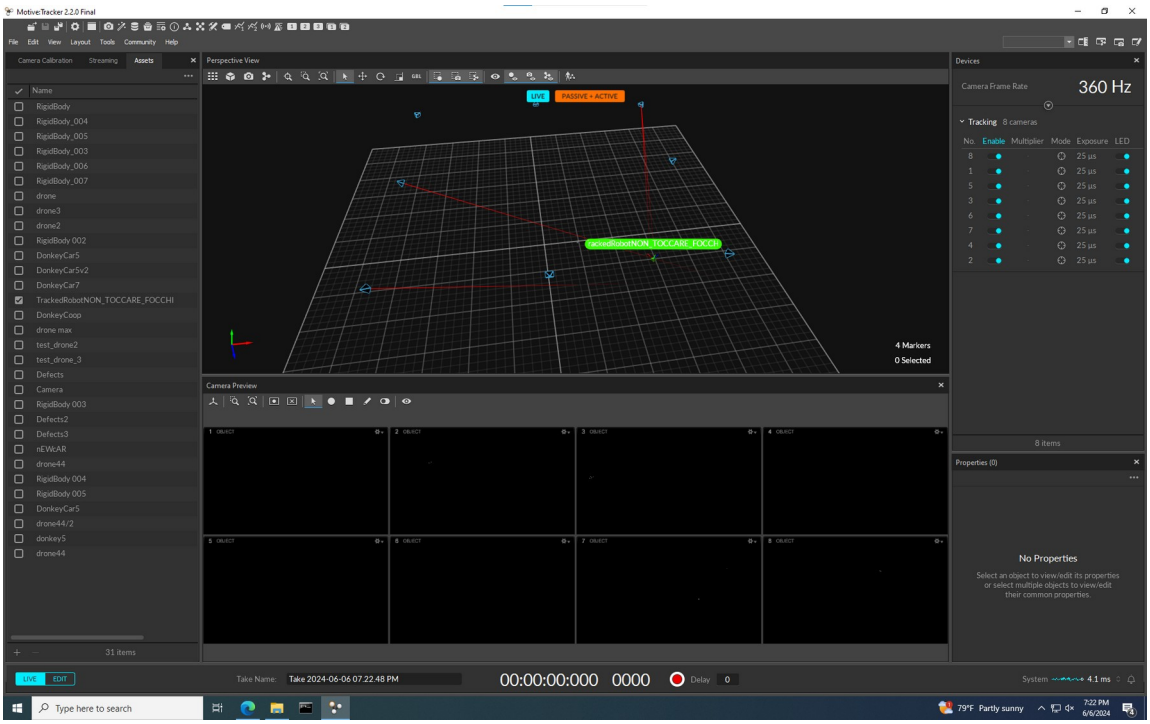


Figure 4: Motive software

- **optitrack_interface**

  python nat net client, it decodes the OptiTrack messages and publishes `geometry_msgs.PoseStamped`, a standard ROS message that can be accessed from the other programs

- **opti-to-px4.py**

  python program listens to the OptiTrack messages and converts `geometry_msgs.PoseStamped` to `VehicleOdometry` messages, a PX4 message type and throttles the output to s stable 100Hz

WHY?

Why doing all these conversion of messages?

Let’s procede with steps: the first conversion from Motive to optitrack_interface is need because Motive is a general pourpose trackign system and publishes it’s own format and not designed to work out of the with ROS so this step is unavoidable.

But in the second part from optitrack_interface to PX4 we convert ROS message to ROS message, can’t we just skip the middle part and do Motive-PX4?

The answer is yes, but not always.

Let me explain, this setup was done in an existing laboratory and a standard `PoseStamped` is recognized by any ROS program and is the most suitable for the data transferred, so an existing setup might alerady have this system setup.

There might be also the case of multiple objects detected or multiple drones, opti-to-px4 only relays the messages with a give ID to the drone

In any case I developed an all in one solution for a direct Motive-PX4 and it’s called **opti-to-px4-v2.py**

This is all the software that is used to make the drone fly which is either proprietary software, forked and modified code or developed from scratch by me, all details are in the main repository.

More programs are used in the simulation section.

CHAPTER II
SOFTWARE:

Each server then has the information regarding which action is being executed, if any. Furthermore, if any goal request is made while the drone is completing a previously received goal the action server will reject the goal request.

Goal rejection

In order to avoid a goal interfering with each other it is necessary to reject a goal request if another task is still in execution. This is done through the knowledge of the drone status topic (section 3.1.2). This algorithm component is executed in the goal_callback which is included in all action servers, the action is employed to demonstrate this simple procedure, implemented in the callback function:

This procedure, while shared by all action servers, is slightly different in the case of the takeoff action server due to the fact that the takeoff action is not necessary in case the UAV is in hover mode. In this specific case the takeoff request will be rejected if the drone is already hovering. In every pseudo code displayed in the following sections this procedure will be substituted by the phrase If the goal request is valid ... in order to avoid repeating the same pseudo-code in every section.

Goal cancellation

Each action server has the capability of canceling the previously sent goal, even if accepted by the action server. This can be done either through the dedicated abort service or through pressing the CTRL+C key on the keyboard while on the client window of the terminal. Once the goal request has been sent to the server the drone starts to move toward the target requested by the action server, if a cancellation is sent (trough a suitable callback) then the UAV sets back the hover mode via the control node.

Communication with the PX4 autopilot

Each action server has the possibility of communicating with the PX4 Autopilot by publishing the required trajectory setpoints, as already seen in section 2.3.2 the input to the low level controller can be either a position or a velocity array. In this thesis work case the position setpoint has been employed for the simpler tasks, like the takeoff or reaching a given target while the velocity setpoints have been used when requiring either a trajectory tracking capability or more control in general. The topic used to communicate to those controllers is TrajectorySetpoint_PubSubTopic while the PX4 odometry (VehicleOdometry_PubSubTopic) has been used as feedback for each action server except the precision landing one (see chapter 4 for details).

CHAPTER III
OFFBOARD ALGORITHMS:

Algorithms

organization In order to provide the UAV with the capabilities necessary to perform the tasks, from the simpler ones i.e. takeoff to the more complex i.e. taking images following a precise pattern and performing a precision landing, a number of algorithms have been developed. Furthermore by taking advantage of the different types of interfaces made available by the robotic operating system it has been chosen to organize such algorithms mainly through ROS 2 actions. The main tasks performed by the UAV in the simulation are:

- Takeoff
- Reaching a user-defined target
- Orbiting around a specific target while taking images
- Providing a coverage of a designated area while performing a grid sweep
- Precision landing To do so, an action server has been developed and associated to each task. Subsequently an action client, has been set up, in order for the user to request the tasks to be performed, thus creating a request/reply communication model between the action clients and server. Everyone of these server is then started via a single launch file for ease of use, and are paired with a continuously running control node. The organization is while the code organization is examined more in depth in the appendix A. Each of the main launch files starts some nodes and processes, described below:

- Simulation launch file:

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

- Configuration file:

```
    #tic of timer
    tic=0

    #mode, only one
    mode = MODE_ROUTINE
    #path procedure
    path_points=[0.0, 0.0, 0.0]
    path_index = -1



    #spin procedure
    spin_rad=0.05

    #updown procedure

    ##flight config
    #relative, negative means up (NED frame)
    #absolute with optitrack, negative means up (NED frame)
    takeoff_height = -2.0
    cruising_speed = 0.5

    #gui
    gui = True
    log=logging.INFO


    ################################################
    #opti-to-px4
    opti_to_px4_dt=0.01 #100 hz
    movement_speed=0.7
```

```
import logging


MODE_NONE=0
MODE_ROUTINE=1
MODE_SPIN=2
MODE_GOTO=3
MODE_PATH=4
MODE_UPDOWN=5
MODE_STOP=6

mode_dict={
    MODE_NONE: "None",
    MODE_ROUTINE: "Routine",
    MODE_SPIN: "Spin",
    MODE_GOTO: "Goto",
    MODE_PATH: "Path",
    MODE_UPDOWN: "Updown",
    MODE_STOP: "Stop"
    }

dist_threshold=0.3
#circle procedure
dt = 0.1
theta = 0.0
radius = 2.0
omega = 2.0
pose=0

#tic of timer
```

PID Controller

PID controller introduction

In order to provide the autopilot offboard controller, which responds to a given trajectory setpoints input, an external PID controller is employed. This controllers computes the velocity input to be transmitted to the autopilot instead of sending directly the position request in order to have a smoother transition towards the target. The controller is employed in the get images, area coverage and land on spot algorithms as a trajectory tracking controller. The PID controller is employed because, even if it doesn't guarantee optimal control or control stability is broadly applicable and does not rely on the knowledge of the model . As shown in a previous section (section 2.3.2) the autopilot flight stack provides with a position and attitude controllers which can be used to stabilize the drone. Instead, the proposed customizable high-level controller is used to compute the best velocity setpoints in the trajectory tracking section of each action server in which is employed.

PID controller architecture

A PID (Proportional integral derivative) controller is a mechanism which employs sensor feedback which is widely used. From the feedback the error value e(t) is computed, which is the difference of the actual measured process variable y(t) from the reference setpoint r(t), the correction input u(t) is applied computed from the PID terms. In this thesis case the feedback may come from the already available PX4 Odometry messages, which then are compared to the reference position to be reached by the drone. Instead, in the precision landing case the error is computed directly through the relative position of the UAV with respect to the rover . In this thesis case the control input u(t) are the velocity setpoints [vx, vy, vz] sent to the autopilot flight stack, the interaction of the PID controller

PX4 and ROS 2 for offboard control

ROS is used by PX4 to provide an offboard control functionality, with a linux companion

computer. In this way its possible to control the PX4 flight stack using a software outside of the autopilot.

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

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

In this thesis the second version of ROS has been employed thus requiring only to setup the connection to the autopilot through the PX4-ROS2 bridge.
From the communication standpoint the PX4 autopilot employs an RTPS protocol and a DDS middleware in order to interface itself with an offboard DDS application (such as ROS 2 nodes), in this way it is possible to exchange uORB2 topics sent by the client (the PX4 autopilot) to RTPS messages to the agent side (offboard computer) and vice versa. The connection between such devices is enabled trough an UART or UDP link. This translation and communication bridge is called the microRTPS bridge, the main components
of such architecture are shown in figure
In both cases the packages are populated at build time via the mentioned python scrips contained in the PX4 firmware. Since not all uORB topics are made available to the ROS
application as a default setting if specific messages are required to be transmitted from uORB to ROS messages then these scripts must be ran manually.
Both packages are available as GitHub repositories for ease of access an installation.
In order to perform it, is’s necessary to clone the repositories and then run the build command, as shown in the installation tutorial Appendix-A.

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

Offboard control

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

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

• At least a stream of > 2 Hz of setpoints commands that shall be sent to the autopilot from the companion computer.

• At least one of the following pose/attitude information has to be available (GPS, optical flow, visual-inertial odometry, mocap, etc.)

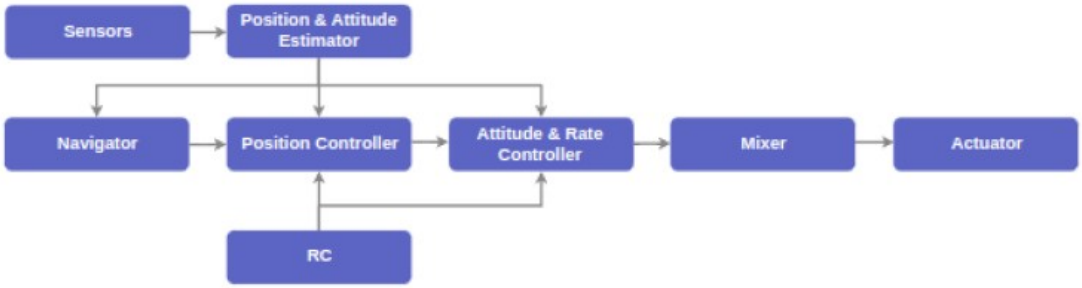
• The vehicle must be previously armed, and it must receive an appropriate stream of commands prior to the offboard mode engagement.

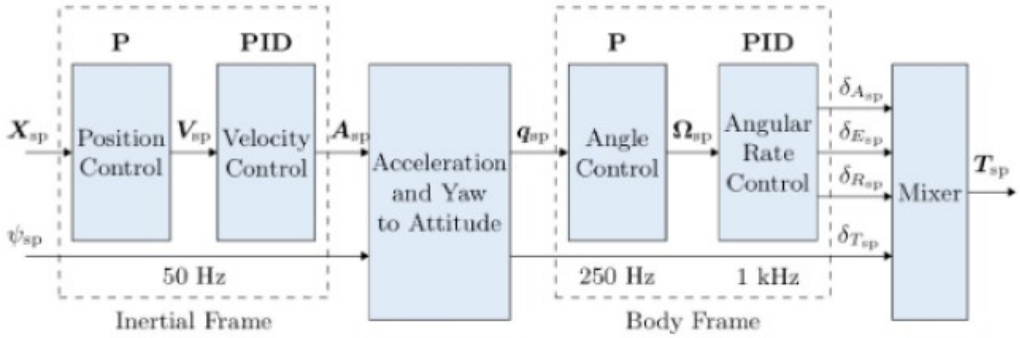
• RC communication must be disabled. If an RC command is issued to the autopilot

```
qos_profile = QoSProfile(
    reliability=QoSReliabilityPolicy.RMW_QOS_POLICY_RELIABILITY_BEST_EFFORT,
    durability=QoSDurabilityPolicy.RMW_QOS_POLICY_DURABILITY_TRANSIENT_LOCAL,
    history=QoSHistoryPolicy.RMW_QOS_POLICY_HISTORY_KEEP_LAST,
    depth=1
)
```

the stream of setpoints commands is stopped, then the vehicle will exit the offboard mode and enter a failsafe state, thus landing.

• Offboard mode requires also a continuos connection to a remote MAVLink system or a ground control station software (such as QGroundControl), otherwise if such connection is lost the vehicle will enter a failsafe mode and land.





CHAPTER IV
ALGORITHMS ORGANIZATION

In order to provide the UAV with the capabilities necessary to perform the tasks, from the

simpler ones i.e. takeoff to the more complex i.e. taking images following a precise pattern

and performing a precision landing, a number of algorithms have been developed.

Furthermore by taking advantage of the different types of interfaces made available

by the robotic operating system it has been chosen to organize such algorithms mainly

through ROS 2 actions.

The main tasks performed by the UAV in the simulation are:

• Takeoff

• Reaching a user-defined target

• Orbiting around a specific target while taking images

• Providing a coverage of a designated area while performing a grid sweep

• Precision landing

To do so, an action server has been developed and associated to each task. Subsequently

an action client, has been set up, in order for the user to request the tasks to be performed,

thus creating a request/reply communication model between the action clients and server.

Everyone of these server is then started via a single launch file for ease of use, and are

paired with a continuously running control node. The organization is displayed in figure

3.1 while the code organization is examined more in depth in the appendix A.

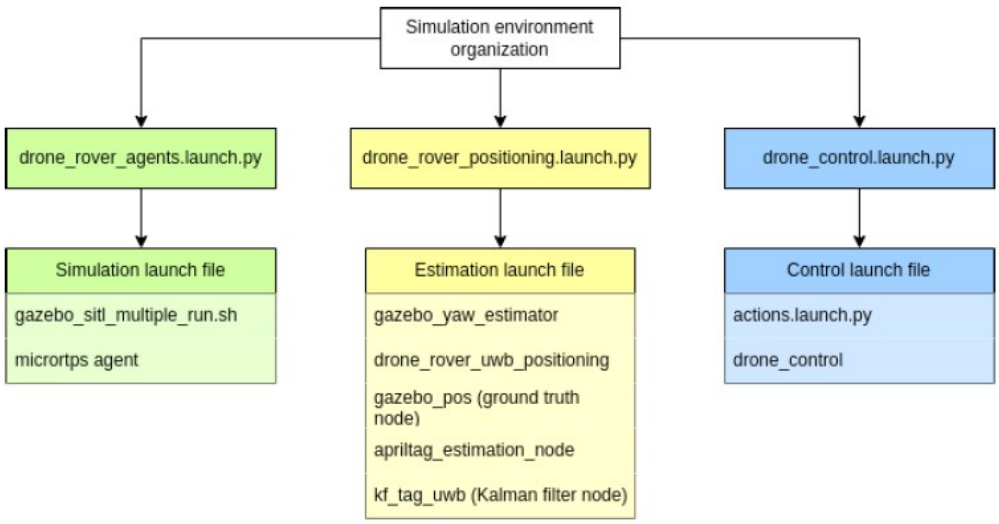
Each of the main launch files starts some nodes and processes, described below: • Simulation launch file

– The gazebo_sitl_multiple_run.sh is a bash script used to start:

* Launch the Gazebo simulation environment

* Generate the world model for the martian ground

* Position the drone model, which is a modified Iris model, in the Gazebo simulation



CONCLUSIONI

Inserire qui il testo delle conclusioni.

BIBLIOGRAFIA

Bianchi, G., Verdi, G. & Rossi, M., 2018. Titolo dell'articolo. *Journal Name,* 48(3), pp. 11-15.

Rossi, M., Bianchi, G. & Verdi, G., 2018. Titolo dell'articolo. *Journal Name,* pp. 10-15.

Verdi, G., 2017. Titolo del capitolo. In: R. Cristiano, a cura di *Titolo del libro.* Ancona: CasaEditrice, pp. 10-20.

ISTRUZIONI PER L'USO DEL MODELLO

Salvare due copie di questo documento, la prima da conservare come modello, la seconda da utilizzare per il proprio elaborato curricolare.

Parametri di riferimento del presente documento

Margini del documento: sinistro 3 cm; destro 3 cm; rilegatura 0,5 cm; superiore 3 cm; inferiore 3,5 cm

Numeri di pagina: in basso a destra (frontespizio e dedica non sono numerate)

Principali stili adottati
la raccolta di stili completa è visibile cliccando su **Stili**

Stili di paragrafo (il testo)

Normale: da usare per il corpo del testo; carattere *Times New Roman*; corpo 11 pt; interlinea 1,5; rientro prima riga 0,5 cm; nessuna spaziatura prima e dopo; giustificato

Citazione: da usare per le citazioni lunghe (almeno 4-5 righe); carattere come il corpo del testo; corpo 11 pt; interlinea singola; rientro prima riga assente; rientro a sinistra e a destra 0,5 cm; spaziatura prima 6 pt, dopo 12 pt; giustificato

Abbreviazioni: da usare per le sigle e le abbreviazioni nella tavola iniziale; carattere come il corpo del testo; corpo 11 pt; interlinea 1,5; prima riga sporgente di 2 cm; spaziatura dopo 6 pt; giustificato; mantieni assieme le righe

Stili di paragrafo (i titoli)

Titolo 1: da usare per i titoli delle parti principali del testo (tavola delle sigle, introduzione, capitoli, conclusioni, bibliografia, indice); carattere come il corpo del testo; corpo 16 pt; interlinea singola; rientro prima riga assente; spaziatura dopo 96 pt; allineamento centrato; anteponi interruzione (affinché il titolo cominci con l'inizio di una pagina)

Titolo 2: da usare per i titoli dei paragrafi; carattere come il corpo del testo; corpo 12 pt; grassetto; interlinea singola; prima riga sporgente di 0,5 cm; spaziatura prima 24 pt, dopo 12 pt; allineamento a sinistra; mantieni con il successivo; mantieni assieme le righe

Titolo 3: da usare per i titoli dei sottoparagrafi; carattere come il corpo del testo; corpo 12 pt; corsivo; interlinea singola; prima riga sporgente di 0,5 cm; spaziatura prima 12 pt, dopo 12 pt; allineamento a sinistra; mantieni con il successivo; mantieni assieme le righe

Titolo 4: da usare per i titoli dei sotto-sottoparagrafi; carattere come il corpo del testo; corpo 12 pt; tondo; interlinea singola; prima riga sporgente di 0,5 cm; spaziatura prima 12 pt, dopo 6 pt; allineamento a sinistra; mantieni con il successivo; mantieni assieme le righe

NB: I titoli dei capitoli, dei paragrafi e dei sottoparagrafi sono numerati automaticamente in modo strutturato (1, 1.1, 1.1.1). La numerazione dei capitoli prevede la dicitura automatica “Capitolo 1”, “Capitolo 2”, ecc. Nella tavola delle sigle, introduzione, conclusioni, bibliografia, indice la numerazione è stata eliminata

Stili di carattere

Enfasi (corsivo): da usare per la parola o le parole *in corsivo*

Enfasi (grassetto): da usare per la parola o le parole **in grassetto**

Enfasi (maiuscoletto): da usare per la parola o le parole IN MAIUSCOLETTO

Sigla corsivo: da usare per le sigle in corsivo come ad es. *CCL*

Sigla tondo: da usare per le sigle in tondo come ad es. DH

NB: Gli stili di carattere consentono un maggiore controllo del testo e lo rendono più stabile. Per rimuovere uno stile di carattere o una formattazione applicata a una o più parole il comando a tastiera è: CTRL+barra spaziatrice oppure CTRL+MAIUSC+Z

Bibliografia

La lista dei riferimenti bibliografici alla fine del documento è generata con il comando **Riferimenti** > **Bibliografia** > **Inserisci bibliografia**. È possibile aggiornare la lista dalla voce di menu contestuale (pulsante destro mouse) **Aggiorna campo**.

Lista delle tabelle

La lista delle tabelle è generata con il comando **Riferimenti** > **Inserisci indice delle figure**. Nella finestra di dialogo selezionare **Etichetta didascalia: Tabella**. È possibile aggiornare la lista dalla voce di menu contestuale (pulsante destro mouse) **Aggiorna campo**.

Lista delle figure

La lista delle figure è generata con il comando **Riferimenti** > **Inserisci indice delle figure**. Nella finestra di dialogo selezionare **Etichetta didascalia: Figura**. È possibile aggiornare la lista dalla voce di menu contestuale (pulsante destro mouse) **Aggiorna campo**.

Sommario

Il sommario è generato con il comando **Riferimenti** > **Sommario > Sommario personalizzato**. Nella finestra di dialogo selezionare **Formati: Da modello**. È possibile aggiornare la lista dalla voce di menu contestuale (pulsante destro mouse) **Aggiorna campo**.
SI RICORDI DI ELIMINARE LA VOCE “SOMMARIO” DAL SOMMARIO.