Université Joseph Fourier GRENOBLE

MASTER THESIS

Deployment of a simulation software architecture for a fleet of UAVs

Rafael Bailón-Ruiz

supervised by

Simon LACROIX

Research Director, LAAS-CNRS

LAAS CNRS

INSTITUT CARNOT LAAS CNRS

Friday 24th June, 2016

Abstract

SkyScanner is a project whose mission is to guide a fleet of Unmanned Aerial Vehicles (UAVs) to actively gather data in low-altitude cumulus clouds with the aim of mapping atmospheric variables. This report presents the work done for the SkyScanner project at LAAS-CNRS to design and implement a simulation software architecture for a fleet of UAVs. The new software package, based on the Robot Operating System (ROS) framework, integrates the in-house mapping and path planning algorithms with realistic simulators: Paparazzi open-source autopilot, FlightGear flight simulator and MesoNH atmospheric model. Then, the whole simulation loop is tested in order to identify potential weaknesses. Finally, some guidance and navigation enhancements are proposed in order to improve system performance.

Contents

List of Figures

Chapter 1

Introduction

1.1 The SkyScanner project

Meteorological scientists have nowadays great knowledge of atmosphere at macroscopic level, predicting events with accuracy. However at fine-grained level there are still lots of uncertainties, mostly at the boundary layer of clouds.

SkyScanner is devoted to the study and experimentation of using a fleet of drones to adaptatively sample cumulus-type clouds, following their evolution to study entrainment and the onset of precipitation [1]. The objective is to characterize the state of boundary layer below and surrounding a cloud, its atmospheric stability, lifting condensation level and updraft (upward-moving air current), shown in Figure 1.1.

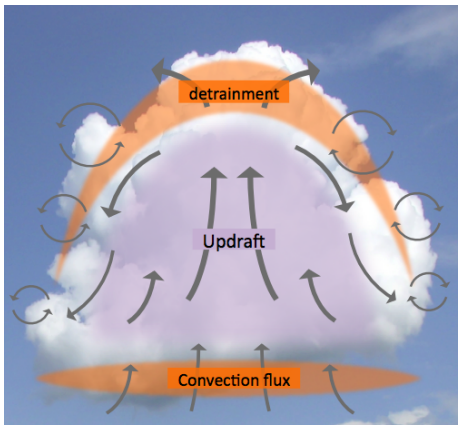


Figure 1.1: Coarse cloud schema. The physical processes at the boundary of the clouds remain not precisely understood by the atmosphere scientists.

The measurements of some parameters like aerosols can be taken from ground using LIDARs, that measure backscatter from the atmosphere, although other variables such as updraft can not be sampled. Thus, cloud microphysics can not be completely understood without in-situ measurements. Manned planes could accomplish in-situ measurements with on board instruments but are too fast and their size is too big to be able to focus on a single cloud nor taking fine samples. Also, there is a need to sample several places at the same time, which a single plane can not achieve. Furthermore, campaigns with manned aircrafts has a prohibitive cost for most research groups and institutions.

In recent days small fixed-wing unmanned aerial vehicles, also known as *drones*, have changed the landscape of atmosphere science. They are cheaper and smaller, and so they can be deployed faster, more often, in more situations and in bigger numbers. In fact UAVs are already being used by atmosphere scientists to follow some meteorological phenomena. But a single UAV is not sufficient to sample a whole cloud that can occupy a volume of up to one kilometer cube, which evolves through time: the benefits of exploiting *a coordinated fleet of drones* is obvious to gather in-situ cloud data.

The SkyScanner project aims at developing a fleet of drones to probe cumulus clouds. It is a joint effort of: CNRM-GAME[1] bringing cloud models and meteorological knowledge; ISAE[2] and ONERA[3], providing endured drones; and ENAC[4] and LAAS-CNRS[5] contributing to fleet control.

1.2 Previous work

Studying the evolution of clouds is intrinsically a 4D problem as they not only evolve in space but also in time. Already explored areas of the map loose value as time passes because of atmosphere evolution. Additionally, data are gathered over 1D manifolds (curves) of the four-dimensional space. Therefore, exploring a cloud by a fleet of drones is very different from the classic robotic exploration problem, where the robots perceive a whole surface or volume from one position: here the UAVs perceive only local data at their position. These particularities condition the developed solutions to create a map of the cloud and to explore it.

Mapping and path planning are interleaved processes: the mapping algorithm needs information about the unknown surrounding environment. On the other hand, the planning needs the map to generate paths that optimize the amount of gathered information and the spent energy.

The following subsections present the way these processes have been designed by LAAS within the SkyScanner project.

Mapping

The mapping task consists in reconstructing a continuous 4D (position and time) map from punctual and sparse measurements of wind[6]. It uses Gaussian Processes to solve this regression problem, a collection of random variables which have a joint Gaussian distribution. The interpolation mechanism of Gaussian Processes is also known as *kriging*.

Gaussian Process variables are defined by their mean $m(\mathbf{x})$, and covariance matrix, which is defined by a *kernel*, $k(\mathbf{x},\mathbf{x}')$:

$$\begin{aligned} m(\mathbf{x}) &= \mathbb{E}[f(\mathbf{x})], \\ k(\mathbf{x},\mathbf{x}') &= \mathbb{E}[f(\mathbf{x}) - m(\mathbf{x})(f(\mathbf{x}') - m(\mathbf{x}'))], \end{aligned} \tag{1.1}$$

The choose of a kernel function is important, as it sets a prior knowledge about the underlying process like its stationary or periodicity. At the same time the set of hyperparameters of the kernel function should be determined according to the actual sparse samples. This means that a continuous optimization process of the hyperparameters is necessary.

Then the inference process is performed with Equation 1.2, where $\bar{y}_\star$ is the mean and $\mathbb{V}[y_\star]$ the variance at point $\mathbf{x}_\star$ of the functions represented by the Gaussian Process conditioned by the $\mathbf{X}$ previous samples.

$$\begin{aligned} \bar{y}_\star &= \mathbb{E}[y_\star|\mathbf{x}_\star,\mathbf{X},y] = K(\mathbf{x}_\star,\mathbf{X})K(\mathbf{X},\mathbf{X})^{-1}y, \\ \mathbb{V}[y_\star] &= \mathbb{E}[(y_\star - \bar{y}_\star)^2] = k(\mathbf{x}_\star,\mathbf{x}_\star) - K(\mathbf{x}_\star,\mathbf{X})^\top K(\mathbf{X},\mathbf{X})^{-1}K(\mathbf{x}_\star,\mathbf{X}) \end{aligned} \tag{1.2}$$

Applying the inference process over multiple points allows to estimate a continuous function, like the one in shown in Figure 1.2, with an associated variance.

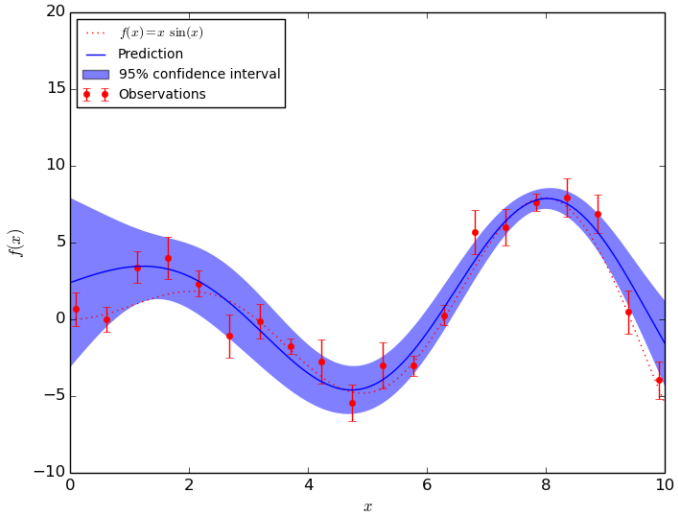


Figure 1.2: Gaussian Process with noise

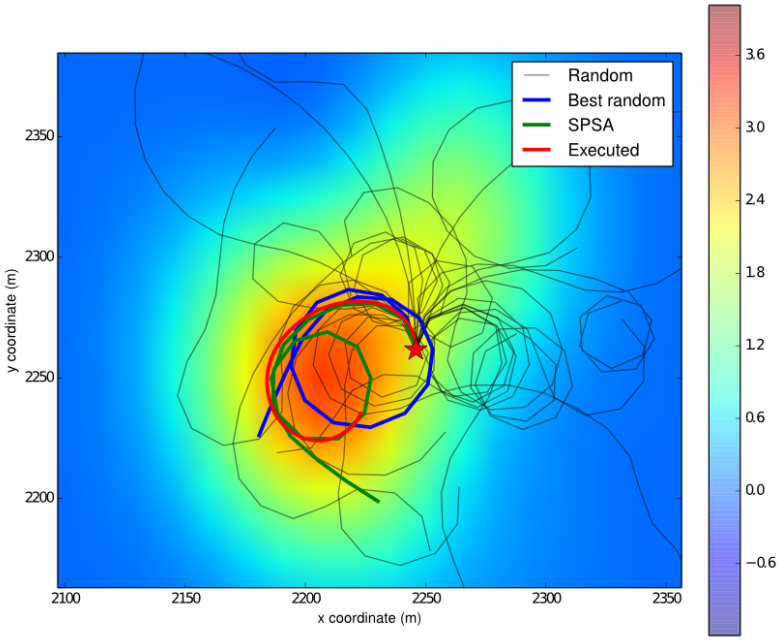


Figure 1.3: Illustration of the trajectory generation process in a realistic wind field, for a planning horizon $\Delta T = 20$ seconds. Projections on the xy plane for the random sampling initialization, optimized trajectory and final trajectory executed by the UAV are shown. The red star represents the initial position and the map colors shows the vertical component of the wind – with no particular unity, the redder being the highest.

Path planning

The path planning process defines paths that maximize the information gain within a specified area of interest while minimizing the energy consumption. It is a multi-criteria optimization problem that blends criteria related to *energy*, *information gain* and *region of interest*.

[1] Centre National de Recherches Météorologiques – Groupe d'étude de l'Atmosphère Météorologique
[2] Institut Supérieur de l'Aéronautique et de l'Espace
[3] Office National d'Etudes et de Recherches Aérospatiales
[4] École Nationale de l'Aviation Civile
[5] Laboratoire d'Analyse et d'Architecture des Systèmes – Centre National de la Recherche Scientifique
[6] Other variables such as temperature, pressure or liquid water content are relevant for the atmosphere scientists – however in this work we focus on winds, as it conditions the flight possibilities of the drones

The current solution is a centralized approach: computations are performed on a single ground station connected to all the UAVs (no communication constraints are considered). UAVs have indeed a low on-board computation power, and the only data that have to be exchanged with the ground station are scalar sensor data gathered at a few Hz, and trajectories once every few seconds.

As for aircraft dynamics, we suppose constant airspeed, limited turn radius and *perfect guidance*.

The overall planning happens in two stages: *Task allocation* at a coarse level and for each drone, an *optimal path generation*. Task allocation is under the control of the operator, who defines the utility of zones and allocate drones to those areas. Then the real path path planning process occurs. The algorithm plans trajectories using forward simulation taking in account the optimization criteria and the restrictions on UAV motions. The forward simulation is done using a basic fixed-wing aircraft model that only does banked turns. In this part, the optimization takes place in two steps: First, a blind random search of feasible paths is achieved at a finite, short, horizon (about 20 seconds): this generates a set of trajectories. The best one is selected, and then optimized with *Simultaneous Perturbation Stochastic Approximation* (SPSA) optimization algorithm.

1.3 Motivations

Mapping and path planning libraries have provided interesting results in a realistic simulated atmosphere (See Figure 1.4). Only the quality of the mapping algorithm were tested assuming the UAV can perform perfectly the predicted path. This means that the tests were not taking into account the actual behavior of a real UAV. Besides, no realistic flight dynamics simulator was included, and so the software library was not yet engineered to work with actual flight simulators.

So it is necessary to define a new software architecture allowing the integration of the previous work (on mapping and path planning) and flight simulators, in order to point out and identify possible weaknesses in the system. The architecture should be defined with the goal of being able to make as easy as possible the later transition into real tests with UAVs. It also seems important to assess the validity of the assumptions made by the path planner regarding the forward simulator UAV model.

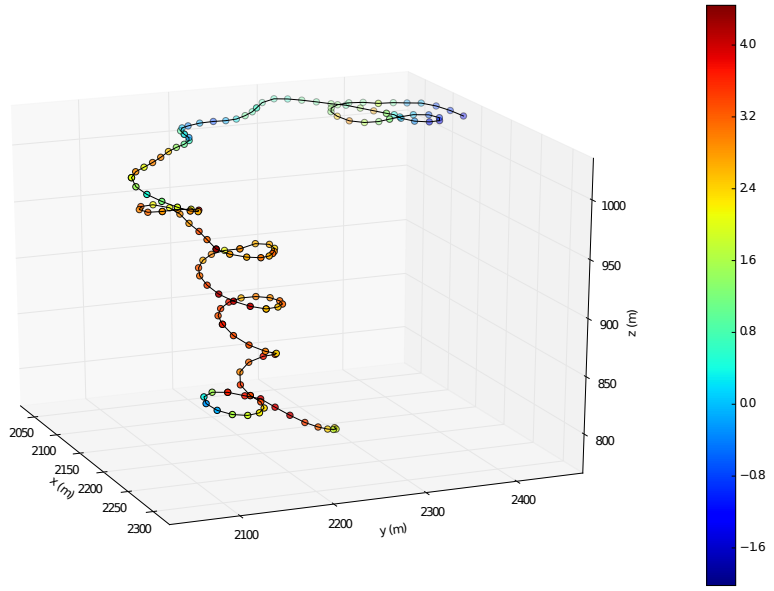


Figure 1.4: UAV trajectory produced by the mapping and path planning library with realistic wind environment. Red represents updraft and blue downdraft

1.4 Objectives

According to the motivations previously introduced, the goals of this internship are:

- Integrate the cloud mapping and cloud exploration algorithms within a more realistic environment
- Test the whole simulation loop in order to identify possible weaknesses of the algorithms
- Propose enhancements to the system

Later on the work, thanks to the implementation of the new simulation loop, some issues were detected. So some enhancement proposals were included as new objectives:

- Research about better guidance algorithms
- Implement some of them to have a deeper knowledge of the system issues

1.5 Requirements

In order to fulfill the objectives, a new software architecture for the SkyScanner project is proposed. It should be able to:

- Integrate the project's previous work
- Be prepared to handle a fleet of aircrafts
- Seamless transfer the algorithms to the real implementation
- Allow space to include the enhancements and new functionalities.

1.6 Report outline

On Chapter 2 the new software architecture that integrates the simulations and the developed mapping and exploring algorithms is presented, and its operation is detailed. Some results obtained with the whole architecture are presented.

Then, Chapter 3 comprehensively analyzes the guidance and navigation algorithms for fixed-wing UAVs. State of the art guidance algorithms are compared and enhancements are proposed.

Finally, Chapter 4 summarizes the main results and presents further work.

Chapter 2

Simulation Architecture

2.1 Robot Operating System

Robot Operating System (*ROS*)[1] is a software framework for robot software development. It provides an operating system-like environment, libraries and tools for heterogeneous computer clusters. ROS packages are composed of three distinct architectural elements: *nodes*, *topics* and *services*.

Nodes are were processing take place, like a process in a regular operating system. Nodes work together in a graph environment and communicate with others by streaming messages in *topics*, doing remote procedure calls to services, and getting and setting values in the *parameter server*. Nodes should be designed to do fine-grained task and then communicate with others to perform more complex tasks. The use of nodes reduces code complexity as the elements work isolated, only exposing the minimum amount of information to operate with others. Moreover, nodes written in distinct programming languages can coexist.

Topics are named buses over which nodes exchange messages. Topics are the way ROS decouples the production of information from its consumption. Nodes can anonymously publish and subscribe to them without being aware of who they are communicating with. Instead, nodes publish to the relevant topic and subscribe to those they are interested in. Topics define a many-to-many relationship where there could be multiple publishers and subscribers to a same one. Despite the link freedom, a topic is defined by a unique *message type*, assuring that only the right type of information will be exchanged.

Services are another communication paradigm. Instead of a many-to-many publisher/subscriber mechanism, it introduces request/reply interactions. A client calls the service and stays awaiting the answer. Services are provided by topics and defined as two topic-like messages, one for the call and the other for the answer.

There is also a fourth component, the *parameter server*. It is a multi-variate dictionary accessible globally so nodes and inspection tools can view and modify it. The parameter server is not designed for performance, but for static data such as configuration settings.

All four core components can be relocated to run in namespaces which are prepended to names and provide a hierarchical naming structure.

2.2 Time management

Most ROS nodes in the package must be synchronized so that the simulated aircrafts, the path planning and mapping algorithms share timely consistent data. ROS client libraries use by default the computer's system clock, known as "wall-clock" but this can be changed by publishing time into the `/clock` topic.

[1] http://wiki.ros.org/

In order to fine control simulation execution sometimes it is useful to be able to change the speed of time, usually to speed it up. SkyScanner's ROS package includes a node `/clock_generator` that handles ROS time (Figure 2.1). It sets the ROS clock by publishing the wall-clock multiplied by a constant into `/clock`, so ROS sees time faster or slower than the computer. In order to be able to set a custom time, the parameter `use_sim_time` must be set in the parameter server.

Using the ROS clock two timing messages are sent to nodes in topics:

- `/tick` running at 1 Hz, data gathering frequency
- `/fast_tick` running at 50 Hz, for deeper control loops

Then, there is the `/clock_control` topic that accept `/ClockControl` messages. Those control messages allow to hold the clock when some debugging is necessary.

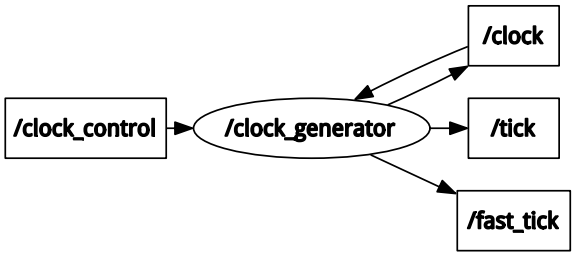


Figure 2.1: Clock generation node and related topics

2.3 Path planning and mapping

The *path planner and mapping* module is in charge of obtaining an estimated wind map given some measurements using a Gaussian Process, and at the same time, generating an optimal path to perform the mapping. The path planner selects a continuous track composed of circle arcs, feasible by the simplified model of a virtual fixed-wing UAV. Those arcs come as result of doing banked turns. Among a set of evaluated trajectories, the path planner picks the optimal path, taking into account some quality measures regarding energy consumption, information gathered and respect of exploration bounds.

This library was developed in former stages of the SkyScanner project and has been already used to obtain preliminary results. However it was not designed to work in a time managed environment, nor to communicate with realistic simulators: further architectural improvements were needed.

At each tick the module receives the position of every UAV and their wind measurement. The library updates the mapping according to these new data. As path generation takes noticeable time, some time management is needed and the process operates at a slower rate. The loop works as shown in Figure 2.2 for every drone in the fleet.

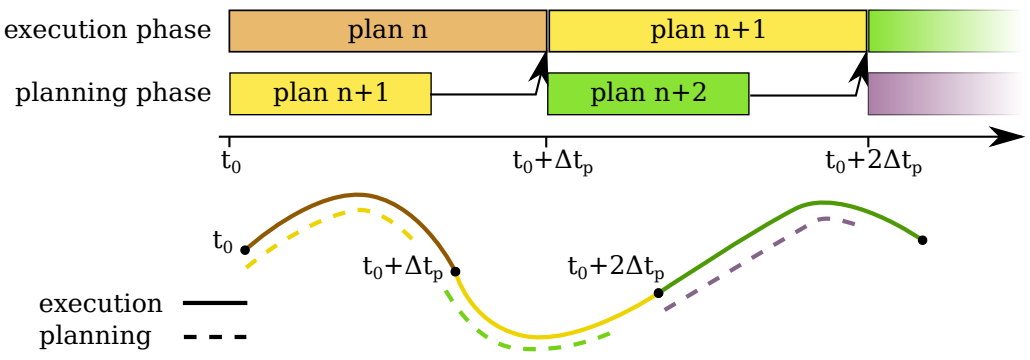


Figure 2.2: Planning chronogram

While a plan is being executed, the next one is being created. At the beginning of each cycle, e.g. $t = t_0$, the states of planner's virtual UAVs are updated to the expected position at $t_0 + \Delta t_p$, where Δt_p is the planner's cycle period. This time is actually the fraction of the planning that is executed. Then, a new plan for the next period is generated from this base position.

Planning time depends on the desired optimization level and is not bounded formally, but it is limited in practice. On the development computer, a 10 second horizon was a sufficient time span in order to have time to plan while not being too large – in which case the map is not reliable.

Figure 2.3 presents all input and output topics related to *pathplanner* node. From each aircraft namespace it takes position and wind measurements, then it does the processing synchronously with `/tick` messages. After that, the library gives two types of output: a set of commands for the plane, turn radius and propulsion power; and a sequence of expected states, position and attitude, which forms the track. Then, using both outputs, the path planning node forms a trajectory expressed as mathematical functions. Circles are converted to straight segments for radius over some threshold. Indeed most guidance algorithms perform better for lines and bigger circles can be approximated by segments without too much error.

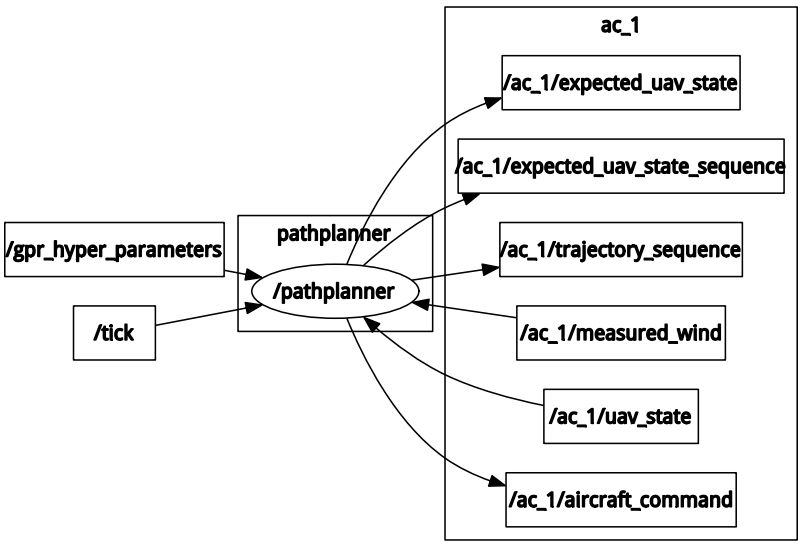


Figure 2.3: Path planner ROS graph

Hyperparameters optimization

The mapping algorithm models the atmosphere through a Gaussian Process and its prediction can be improved by adapting the process hyperparameters. Figure 2.4 shows the *gpr_optimizer* node which recalculates them at each new wind sample from any aircraft. Then it sends the result to the path planner node so it can update the values and profit from the optimization.

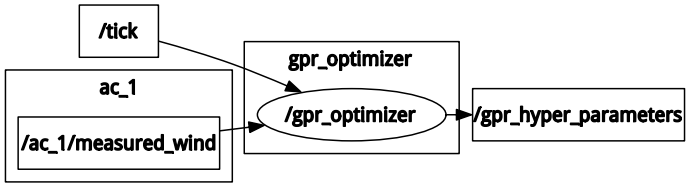


Figure 2.4: GP optimizer ROS graph

2.4 Interface with simulation backends

One of the requirements of the software integration is to be modular. In other words, the functionality of the whole software architecture must be separated into independent modules such that each one implements one task. They may be separated, recombined or replaced affecting minimally to the rest of elements.

First, the *path search simulator*, the one that the path planner uses to generate tracks, was integrated. It is being used to validate the planner behavior.

Then, an interface with *Paparazzi* was developed. At this stage of the project, we are using simulated planes with the JSBSim library.

Finally, communication with the free flight simulator *FlightGear* was written to overcome some limitations in paparazzi that are introduced later in this report.

On the other hand, the data from *MesoNH* atmospheric simulator is used together with flight simulators in order to provide a realistic meteorological environment.

Path search simulator

The planning algorithm needs to look ahead to generate some path according to the restrictions. There is a simple aircraft model in the algorithm that is used to generate trajectories according to the expected characteristics of the real UAVs. The user has to provide the model parameters according to the actual UAV, and the path planner creates a feasible trajectory given this model and the estimated map.

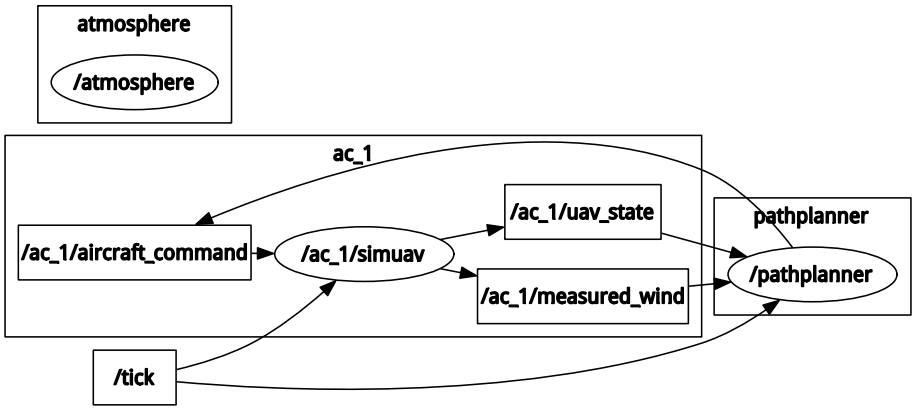


Figure 2.5: Path search simulator (`simuav`) ROS graph

This *sui generis* simulator has been also implemented as an external simulator, with the purpose to assess the quality of the mapping algorithm used in planning. As shown in the ROS graph of Figure 2.5 the virtual aircraft, represented as the `simuav` node, takes the low level aircraft commands as inputs. Without any kind of guidance stage, this means that if the mapping is doing a good work, the resulting UAV state should be the same as the expected UAV state by the planning. Differences between the actual position and the expectation are due to mapping not matching the real wind.

MesoNH

The CNRM-Game research group, using the in-house developped MesoNH simulator[2], has provided realistic simulated data of cloud microphysics in a cumulus generation scenario at fair weather conditions. This simulation includes information about three dimensional wind, pressure, humidity, temperature and liquid water content, which defines the presence or not of a cloud.

[2] http://mesonh.aero.obs-mip.fr

As wind is the atmospheric variable that significantly affects the flight, it is the only one introduced into the ROS architecture loop. In fact, Paparazzi does not allow introducing other meteorological parameter but wind speed. However, SkyScanner aims to include in the future the rest of the atmospheric variables in the process, but only for cloud mapping purposes – they will not influence the flights generation or exectution.

Let's introduce the *atmosphere* ROS node, which is running the `/get_wind` service. This service queries wind speed from MesoNH off-line simulated data, stored in a network drive. Then, there are other topics, *paparazzienvironment* and *flightgearenvironment* which are in charge of setting the wind. As shown in 2.6, they provide position and time to *atmosphere* and put the corresponding wind speed into the simulation. Other flight simulators could be included following the same procedure.

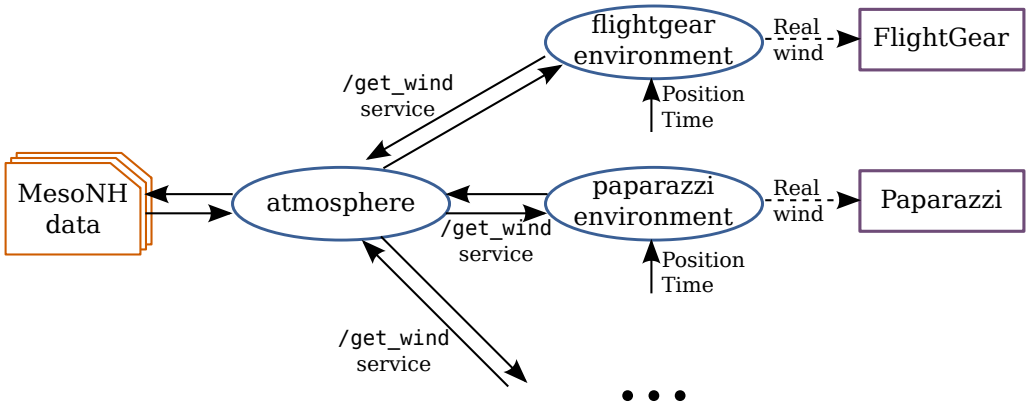


Figure 2.6: Interface between MesoNH data and flight simulators

Paparazzi

Communication between UAVs and Paparazzi ground control software are made through the Ivy bus protocol. Involved systems put on the bus messages that are broadcasted along the channel. Then, subscribers filter the information they are interested in by using regular expressions. Each paparazzi aircraft sends periodical telemetry messages with its position, attitude, energy consumption and task execution status (among others). The ground control station or other custom software can subscribe to these messages and put back on the up-link bus commands for the planes.

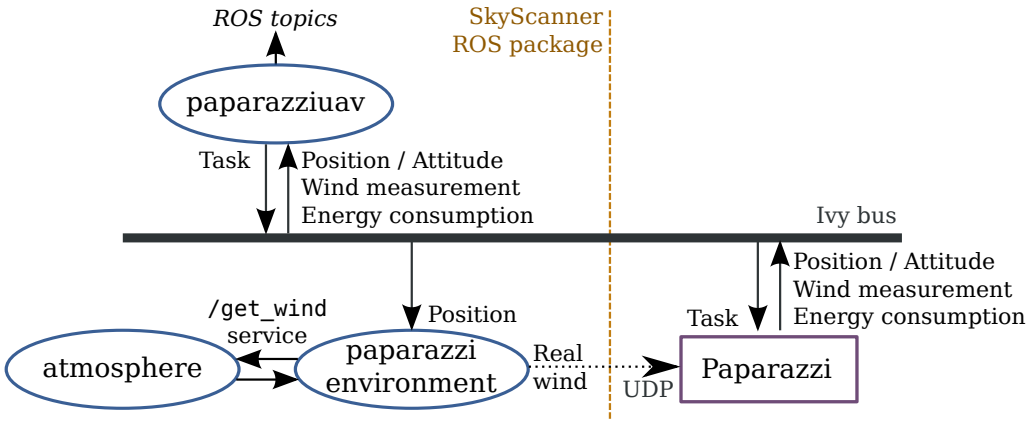


Figure 2.7: Interface scheme between Paparazzi and SkyScanner ROS package

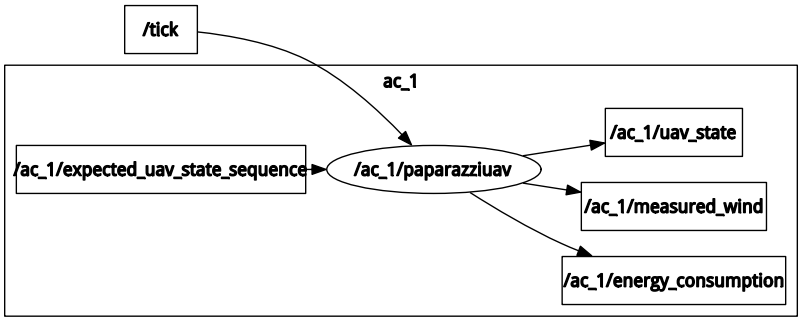


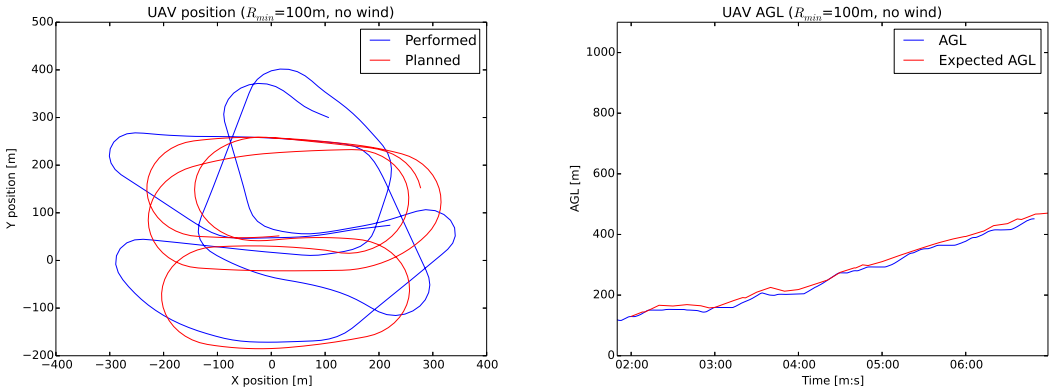
Figure 2.8: ROS graph of paparazziuav

The SkyScanner ROS package communicates with the Ivy bus and exposes the data we are interested in into ROS topics. Paparazzi puts on the bus all the messages received from the planes' telemetry link. This includes position, attitude and energy consumption which are the ones the *paparazziuav* reads. As information from all aircrafts is in the same channel, each node filters the ones that matches the aircraft id. Figure 2.8 shows the ROS graph of a *paparazziuav* node and related topics.

On the other hand, *paparazziuav* puts task messages which can be waypoint, circle, segment and path. Despite the variety of trajectory shapes, the possibilities are quite reduced: Path tasks are just a sequence of five segments at constant altitude and circle tasks are not chained one after other automatically. Even worse, paparazzi uses the carrot-chasing algorithm as guidance. This path following method is suitable for tracking circular and linear trajectories that are steady, but it doesn't perform well for dynamical ones like those produced by the SkyScanner's path planner.

As a sequence of circles cannot be used as trajectory by paparazzi, and after testing with waypoints, segments an paths, it was the segment task the one that performed the best. So *paparazziuav* sends sequences of straight lines to *paparazzi*, but the results weren't satisfactory.

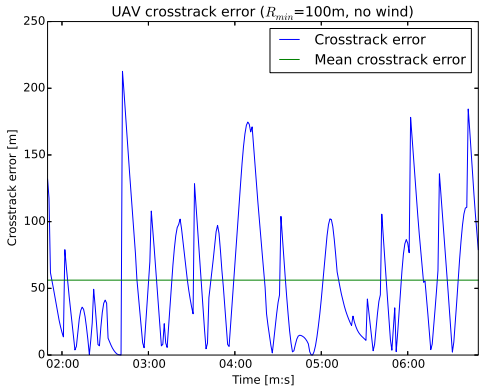
Figure 2.9 shows a simulated *easystar* UAV in paparazzi trying to track the generated path. The minimum radius was forced to 100m to plan the trajectories, and wind effects were deactivated. Even with these conditions, the plane is not able to keep the track and it can be readily stated just with a quick look that the result is not satisfactory.



(a) Horizontal projection

(b) Vertical profile

Figure 2.9: Planned and performed trajectory of an *easystar* UAV in paparazzi's simulator without wind



(a) Instantaneous cross-track error

Max $[m]$	Mean $[m]$	Min $[m]$	σ	$CF = \sigma / \bar{x}$
213	56	0	46.65	0.83

(b) Statistics after a 5 minutes flight

Figure 2.10: Cross-track error of trajectory in Figure 2.9

A quantitative analysis of the ability to follow the track is done using the *crosstrack error* as perfomance measure, defined as the distance between UAV's position and the expected trajectory. Figure 2.10 shows the crosstrack error of the trajectory in Figure 2.9. Table 2.10b proves the weak ability to track the path: even if the mean error seems not to big, taking in account that the expected path is going around an area of 400 by 400 meters, it is significant. The coefficient of variation (CF) is almost 1, the aircraft is almost acting erratically.

Looking at those unsuccessful results made us realize that more research was needed in order to achieve at SkyScanner's objective. Either Paparazzi may not be the best option to guide the aircraft or the paths are not feasible for the plane. As the UAV is able to follow a stationary circumference of the same radius, we supposed that the main problem is in Paparazzi's guidance system which it is not able to track at the required rate. At the same time we cannot forget that Paparazzi autopilot does not allow chaining circles and the are being substituted by small segments.

FlightGear

Due to problems regarding guidance in Paparazzi, we decided to explore other options and try to implement a back-end for a different flight simulator. FlightGear was chosen for several reasons. First, it can use JSBSim as Flight Dynamic Model library, the same one as Paparazzi. In fact JSBSim was developed specifically for FlightGear in mind. Second, it is easy to find aircraft models for it. Third, most simulation parameters, commands and states can be controlled externally with remote protocols. Finally, it is open source software so it can be installed on every computer without restriction, and the implemented code can be inspected.

Flight Gear simulation state, inputs, outputs and settings are organized as a tree where they are categorized. Every value can be read and modified from the program itself or external software. Communication between FlightGear and other software can be achieved by several means: UDP, TCP, HTTP or Telnet. HTTP and Telnet servers in FlightGear have a slow update rate of the values, but they have the advantage of being easy to use. HTTP allows to show data in a web browser and Telnet provides the capability of monitoring and changing values through the command line using client software available in all operating systems. On the other hand, communication can happen at faster rate using UDP and TCP. However, the way to send and receive information is more strict. It is necessary to define which information is going to be sent and received in a XML file, that is loaded when launching FlightGear.

As data transfer between FlightGear and the SkyScanner ROS package should happen at 50 Hz at least (recall the frequencies of operation in Section 2.2), HTTP and Telnet protocols proved to be insufficient. UDP was chosen as protocol over TCP because it is stateless. In UDP there is no connection to maintain, only messages being sent or received, so communication is more robust facing interruptions.

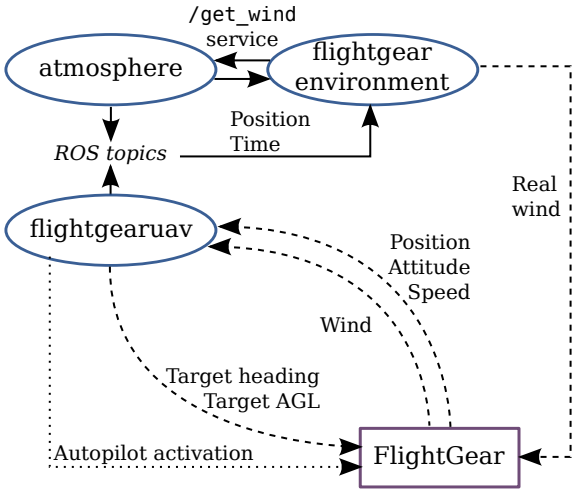


Figure 2.11: Interface scheme between FlightGear and SkyScanner ROS package

Values shown as dashed lines in Figure 2.11 correspond to the ones sent by UDP. Going into details, each line corresponds with a socket: position, attitude and speed are received together at `/fast_tick` frequency; then, the heading and height above ground level (AGL) setpoints in another socket at the same frequency and finally, wind is sent and measured back in two different channels running at 1 Hz. Wind injection can be deactivated if we want to use Flight Gear's atmospheric models.

Other control messages that are not sent evenly but arbitrarily, like autopilot activation or deactivation, are transmitted using Telnet as they don't have to be defined in the XML files at launch time. They are shown as dotted lines in Figure 2.11.

Furthermore, let's point out an important difference between Paparazzi and FlightGear interfaces. The former sends command to UAVs in *tasks* defined by geometrical curves like segments or circles, but the latter sets a *target heading* and a *target AGL*. That is due to the fact that Paparazzi is a whole autonomous flight control system and FlightGear just a flight simulator. This means that to use FlightGear we need to introduce a *guidance* node which translates geometrical trajectories into heading and AGL setpoints. An overview of this node is provided here but it is explained in detail later in Chapter 3.

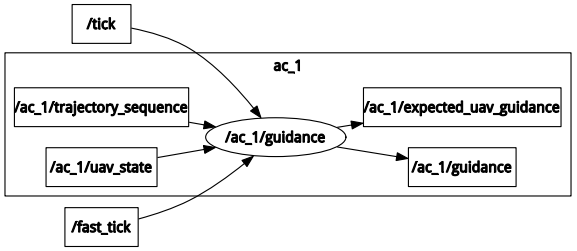


Figure 2.12: `guidance` node ROS graph

The *guidance* takes the mathematical expression of trajectories and the actual position of the aircraft and compute the estimated heading that will lead the UAV to the right path. At a first stage it determines which part of the track to follow, that is, switch between circles and segments following some criteria (temporal mainly). Later it calculates some heading according to the guidance algorithm applied to the current piece of path. AGL setpoint is directly taken from the expected position, leaving the task of tracking it to FlightGear, which is able to do so.

2.5 Execution loop

The fact that until this point and through this report most of the time only one UAV is considered, is just for the sake of simplicity of figures and explanations. As stated in Requirements section of Chapter 1, the upcoming simulation architecture should be able to manage several aircrafts, which os actually possible with the SkyScanner ROS package (See Figure 2.13). Any (reasonable) number of UAVs can be launched within the simulation, allowing each one to have their own set of parameters, and without changing anything of their structure. Every aircraft defined to be launched runs in a different namespace. Involved nodes, topics and parameters are prepended by an aircraft code so no conflict is present. Common nodes like *paparazziuav* or *paparazzienvironment* nodes run on their own namespace but they read and put messages of aircrafts on their namespaces.

Figure 2.14 shows the whole ROS architecture for two different flight simulation backends. They are running one aircraft in the *ac_1* namespace. This name can be whatever sequence accepted by ROS. In this particular case, *ac_1* was chosen because when using the Paparazzi back-end, it is mandatory to define the aircraft id of the plane to appropriately filter the messages in the ivy bus. On the other hand, in the FlightGear case no special id is required but to set the hostname and ports of the UDP sockets. The namespace schema permits also to have isolated parameter sets.

The system could work with aircrafts running in different flight simulator backends. Even if this doesn't have any special utility for us at the moment, it shows how the new architecture is modular and adaptable to distinct situations.

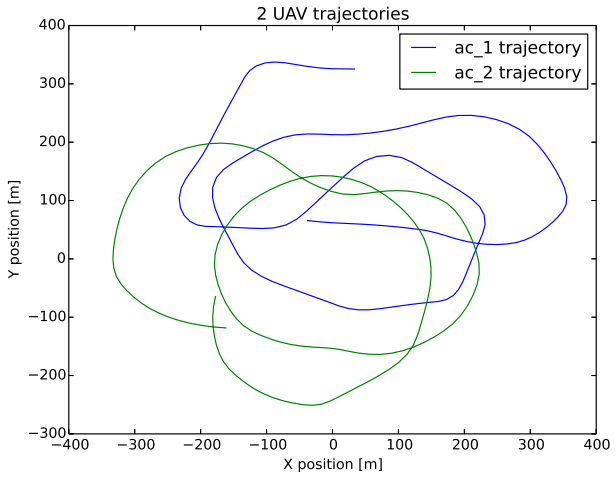
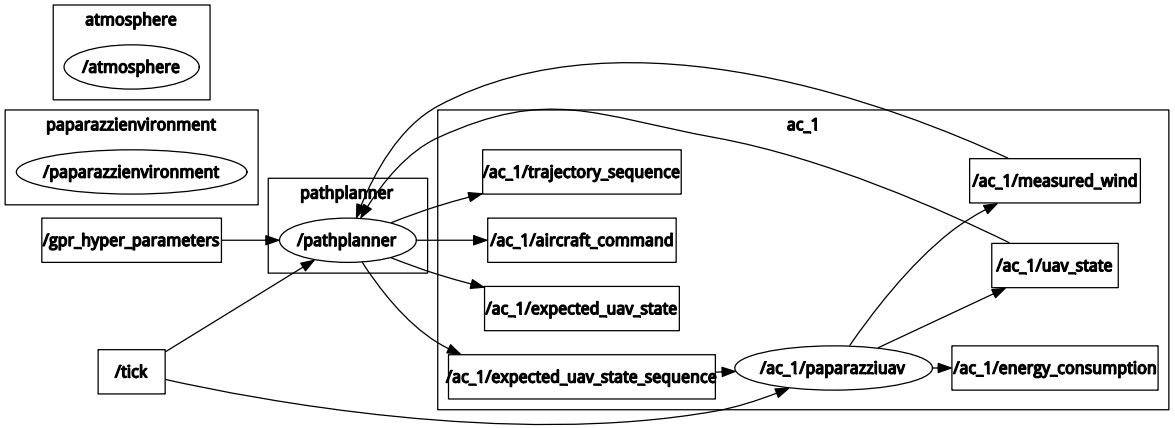
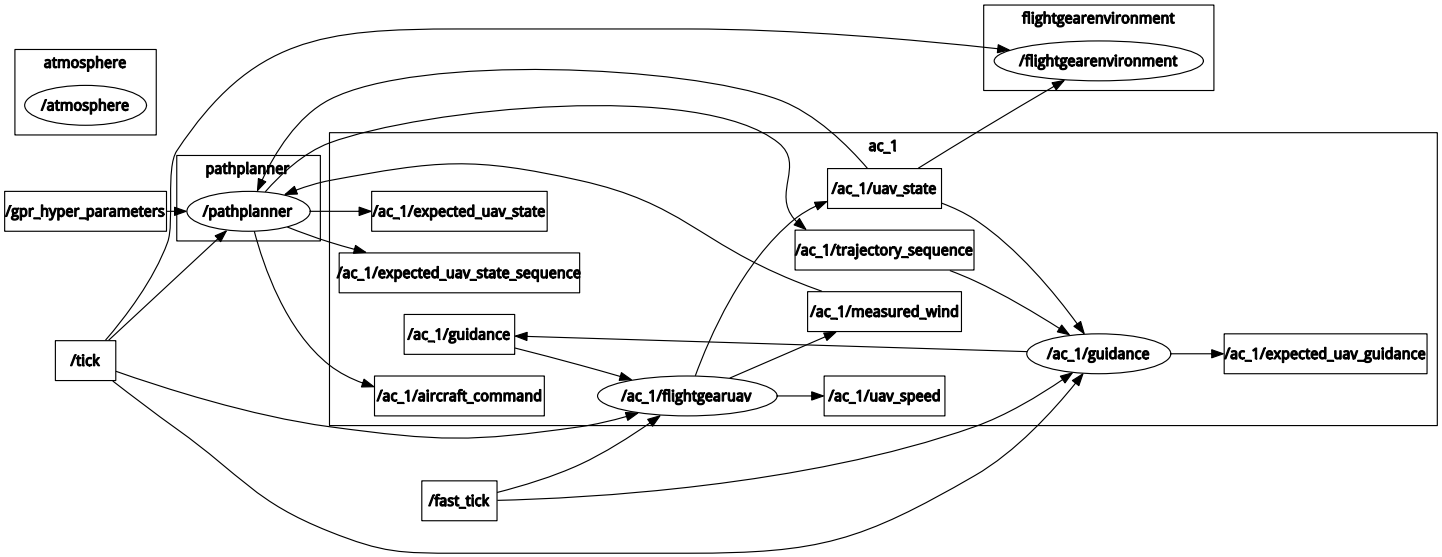


Figure 2.13: Trajectories of two aircrafts simultaneous simulation



(a) Paparazzi ROS graph

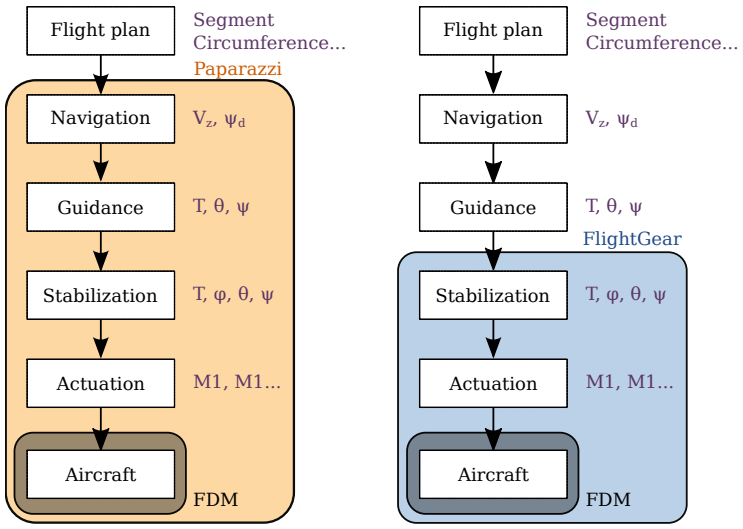


(b) FlightGear ROS graph

Figure 2.14: ROS graph of the complete execution loop using two flight simulation backends

Chapter 3

UAV Navigation and Guidance



(a) Paparazzi **(b)** FlightGear

Figure 3.1: Control layers of autonomous aerial vehicles

The *guidance* or *path-following* problem is to determine the heading angle that accurately tracks a given path. Figure 3.2 shows the schema of circumference tracking: the distance d is the *cross-track error*, ψ is the actual heading of the UAV and θ_d is the direction of the *line-of-sight (LOS)*, the trace of the path. Then, the objective of path-following is to minimize the heading error and the cross-track error, in other words, to make $|\theta_d - \psi| \to 0$ and $|d| \to 0$ for $t \to \infty$.

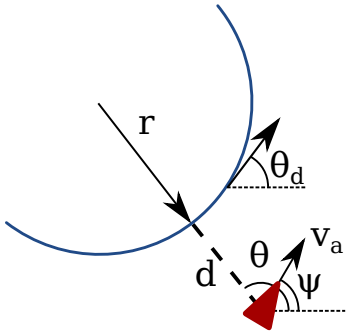


Figure 3.2: Guidance problem definition for circumference path.

There are multiple well-known solutions for the guidance problem in steady two-dimensional paths, but as the SkyScanner's path planner proposes highly dynamical 3D trajectories we must analyze which algorithms is adapted for the desired performance requirements, to detect flaws on those algorithms or on path generation and to propose improvements to both if possible.

Some path-following algorithm types, used widely by UAV autopilots, are compared in [2] but those are studied only for tracks at constant altitude as 3D guidance is still a challenge. There are some works on three dimensional approaches like in [3] where a sliding-mode guidance system is proposed for constant speed missile-like vehicles; or [4] which introduces a 3D extension for some constrained optimal guidance laws. But they have limited applications, so there is still much work to do in order to find generic and robust algorithms.

The usual approach to overcome the limitation to 2D guidance, is to decouple horizontal and vertical displacements [5]. This assumption is completely valid for steady tracks and it is applied to all generic autopilot systems. However it may be not valid because of the type of paths we are dealing with.

For the tests made with FlightGear we used the *malolo 1* UAV model which is the closest model we have found to the one used in Paparazzi (*easystar*). Despite they have similar size, the *malolo 1* is heavier, 8 kg, compared to the weight of the *easystar*, 1 kg. This affects mainly the turn radius when flying so performance cannot be compared directly. But it can give an insight into the effects of the guidance stage in the whole command loop.

3.1 Navigation

Navigation corresponds to the second top layer in Figure 3.1. It translates definitions of tracks into a target heading, the line-of-sight, and target vertical speed to feed the guidance.

The algorithm depends on the type of curve: waypoint, straight line, circumference... For waypoints the heading setpoint is the angle that forms the line going from the UAV to the target. In the case of straight lines, the heading is the orientation of the path which is a step input for the guidance loop.

For other curves, the target heading is the derivative of the curve at the closest point to the aircraft. It requires calculating the minimum distance from a point to a curve, which requires solving an equation for any non-trivial situation. The equation may not have an explicit solution and require the use of root-finding algorithms. Hopefully, it is still easy to solve it for circumferences: the heading setpoint is perpendicular to the segment that joins the center and the aircraft. From the guidance algorithm point of view, it consists in following a ramp function with a slope depending on circle radius.

3.2 Stabilization

Under the guidance layer, *stabilization* takes a heading setpoint and commands the actuators. Heading change is performed doing a *banked turn*, in which the aircraft inclines towards the inside of the turn. In FlightGear this control layer is implemented as a 2-stage cascade PID controller, as shown in Figure 3.3. The inner loop controls the aircraft roll, φ, changing aileron position. The outer loop sets the roll to handle the heading ψ. φ is limited to ± 20 degrees.

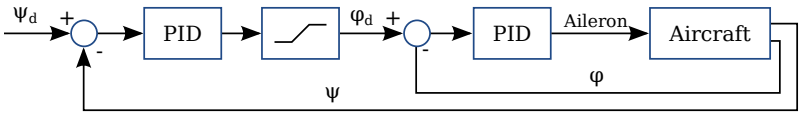


Figure 3.3: Aircraft stabilization control loop scheme.

Figure 3.4 shows the response of the stabilization control loop to a 90 degree step response. We can notice a first-order response superposed by a small ripple, caused by the most internal control loop, and a 1 second time delay with negative overshoot.

The settling time for 95 % of the setpoint is 13 seconds. If the UAV flies at 15 m/s, this means that it will perform an arc of 195 m while this period of time. As the step goes from zero to 90 degrees, we can say that the minimum turn radius it can perform is around 125 meters.

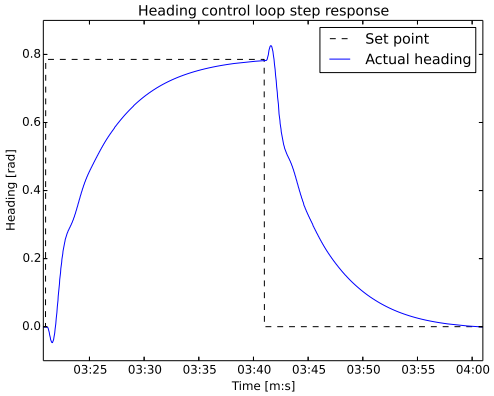


Figure 3.4: Heading stabilization step response for a *malolo 1* UAV.

3.3 Carrot-chasing guidance

The *carrot-chasing* guidance algorithm introduces a *Virtual Target Point (VTP)* that must be followed by the UAV. Like a donkey follows a carrot attached to a stick, the UAV updates it's heading to pursuit the VTP which moves ahead over the path at a fixed distance. This is the guidance algorithm included in Paparazzi and therefore the first one it has been used by the SkyScanner project.

The performance of the carrot-chasing algorithm depends on the VTP distance to the UAV. For short distances the vehicle cannot track the trajectory as it moves straight to it, resulting in overpassing and making a sinusoidal trajectory along the track. For greater carrot distances, the UAV converges to a straight line path. Having a larger distance to the carrot prevents the aircraft to overpass the path but increases the settling time.

For circles, as for segments, a short carrot distance causes the UAV to follow a sinusoidal path along it. Increasing it makes the trajectory converge but unlike lines, a too big value provokes the UAV to travel around the circle without never reaching it (See Figure 3.5). Values considered small or big depend greatly on the radius of the circle and the aircraft following it. In Paparazzi, the VTP is measured in units of time. As it presumes constant speed it is considered equivalent.

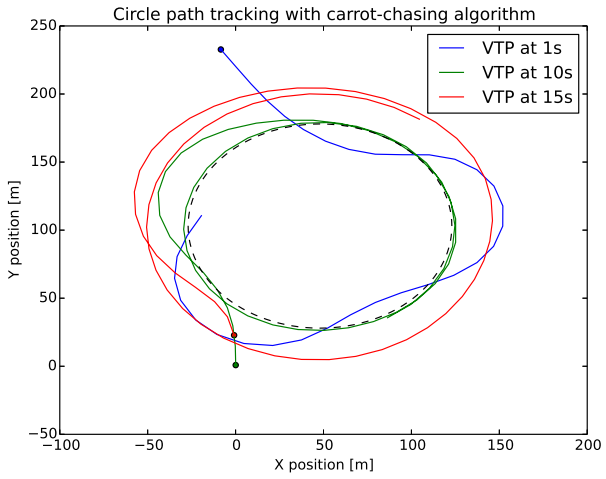


Figure 3.5: Circumference tracking comparison for different Virtual Target Point distances.

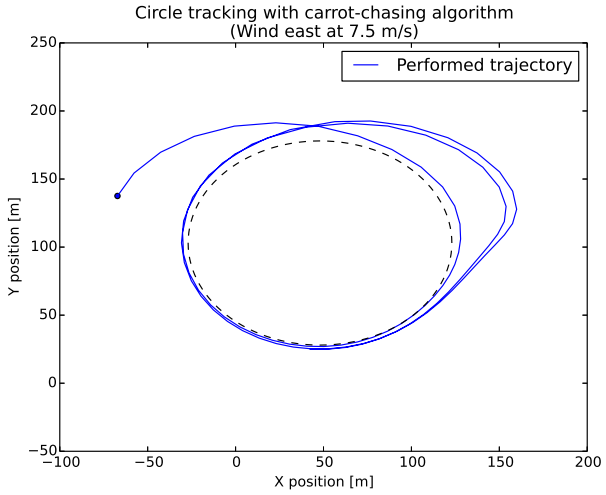
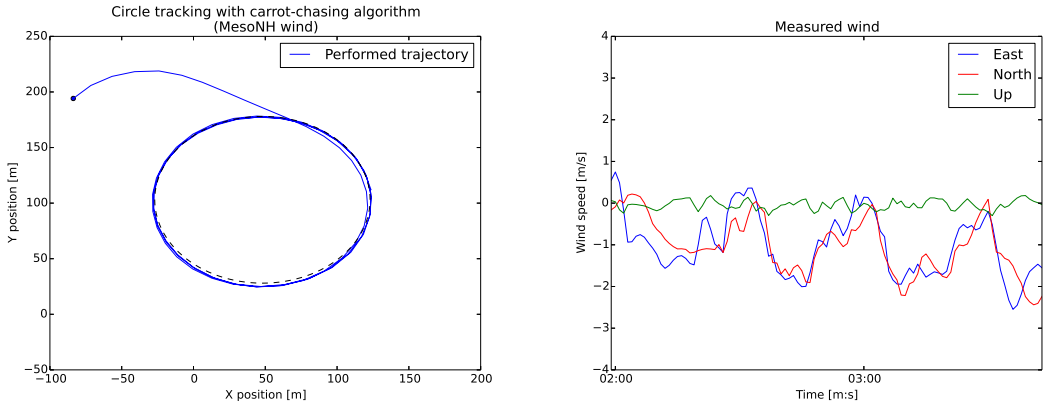


Figure 3.6: Circumference tracking with constant 7.5 m/s east component wind.



(a) UAV trajectory

(b) Wind speed components

Figure 3.7: Planned and performed trajectory of an *easystar* UAV in paparazzi's simulator without wind

Introducing wind into the environment means putting a disturbance to the system and when it is present, the constant speed presumption no longer applies. In Figure 3.6 the UAV goes faster when moving in the same orientation of wind and slower when moving facing it. This results in not performing a circle as planned. When applying realistic wind, with smaller magnitude, it doesn't affect too much to the trajectory. We only observe a minimum displacement in south-west direction.

The carrot chasing algorithm usually performs the worst among state-of-the-art guidance laws in terms of robustness to external disturbances [2]. Some wind compensation algorithms are proposed in the literature such as wind estimation through a model[6] or changing the VTP distance according to a fuzzy logic algorithm[7], but they cannot compete with other algorithms applying the same provision.

3.4 Pure pursuit and LOS guidance

Another geometric method is the *pure pursuit and line-of-sight (PLOS)* guidance algorithm, which is a combination of two guidance laws. The *pure pursuit* law drives the aircraft to the destination and the LOS guidance steers it toward the line-of-sight.

The pure pursuit guidance law is given by:

$$\psi_p = k_p(\theta_d - \psi) \tag{3.1}$$

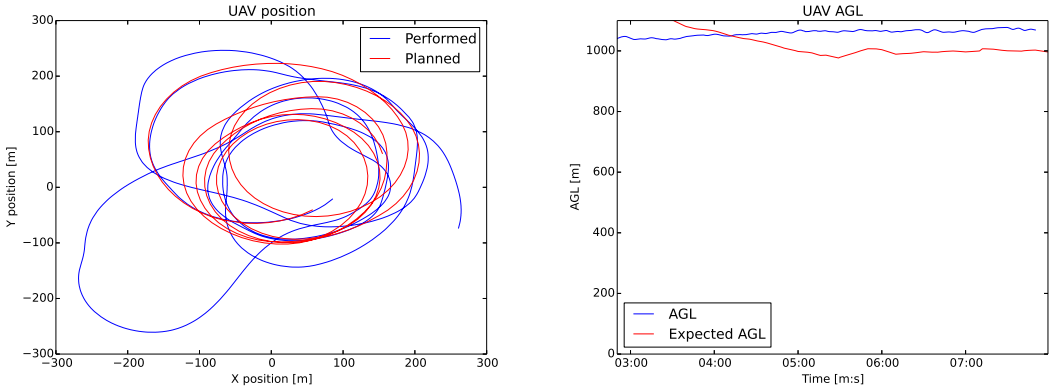
On the other hand, the LOS guidance law is:

$$\psi_l = k_l d \tag{3.2}$$

A linear combination of Equation 3.1 and Equation 3.2 makes the UAV follow the path:

$$\psi_d = k_p(\theta_d - \psi) + k_l d \tag{3.3}$$

As stated in [2], the PLOS guidance law is stable for every $k_p > 0$ and $k_l > 0$ and should have less crosstrack error compared to the carrot-chasing algorithm.



(a) Horizontal guidance. $k_p = 100$, $k_l = 3$.

(b) Vertical profile

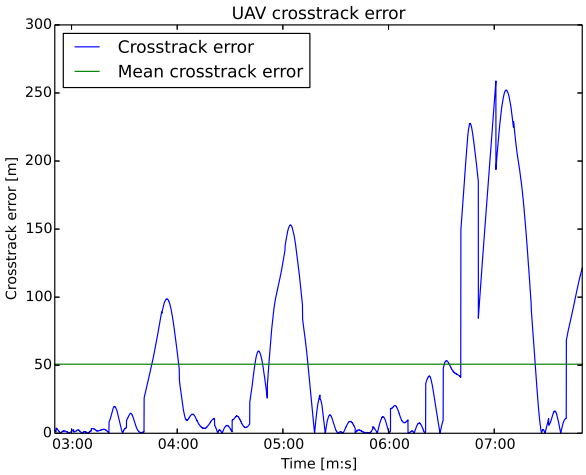
Figure 3.8: Planned and performed trajectory of a *malolo 1* UAV in FlightGear simulator with realistic MesoNH wind and PLOS guidance.

It can be seen in Figure 3.8 that the path is being tracked better than with Paparazzi's guidance in Figure 2.9 but sometimes it gets lost. Altitude is not being tracked at all, the reason may be decoupling the horizontal and vertical movement is not valid. However we suspect some issues in the aircraft model as height above ground level is almost not reacting to changes in altitude setpoint.

Regarding the crosstrack error, Figure 3.9 its mean value is similar to the one obtained with paparazzi and introduced in Figure 2.10. However we notice half of the time the crosstrack error is quite low while in the other half it is much bigger. In addition, the error function is not continuous (there are big jumps around minute 07 : 00). An in depth look at this situation made us realize that this is due to two reasons: first, in Figure 3.10 we can see the airspeed is not constant and it can vary from $14,5\ m/s$ to $19,5\ m/s$. Second, transitions between pieces of track are not smooth from the point of view of guidance. The path is continuous, but the parameters defining, turn radius and circumference center, are not.

3.5 Vector field guidance

Another approach to guidance are *vector field* (VF) guidance laws. Vector fields are potential functions which minimum lies on the desired path. For each point in space there is an associated guidance vector which guides the aircraft to path. If the UAV is able to follow the orientations of the vector, it arrives to the track. The stability of these algorithms is assured by Lyapunov stability criteria.



(a) Instantaneous cross-track error

Max $[m]$	Mean $[m]$	Min $[m]$	σ	$CF = \sigma / \bar{x}$
259	50.7	0	68.421433	1.348393

(b) Statistics after 5 minute flight

Figure 3.9: Cross-track error of trajectory in Figure 3.8

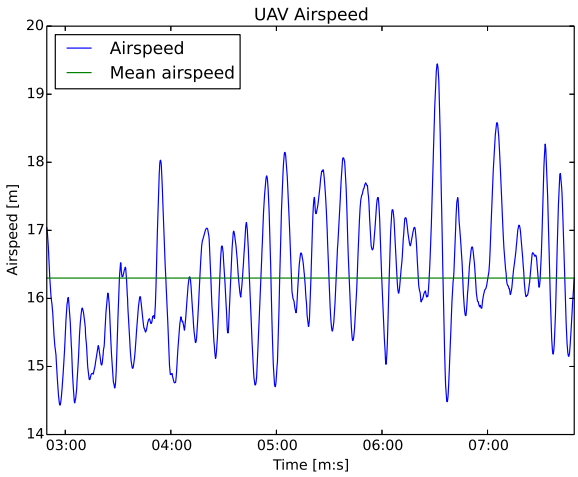


Figure 3.10: Measured airspeed during the flight of Figure 3.8.

The following algorithm [8] is currently being implemented in Paparazzi by the people of ENAC, but we have used it with FlightGear. This control law is designed to command a vehicle in speed. However, FlightGear's autopilot doesn't allow to control aircraft's heading rate of change. So it has been adapted, with the help of one of its authors, to give the heading as output.

Let's define a path $\mathscr{P} \subset \mathbb{R}^2$ described implicitly by:

$$\mathscr{P} = \{(x, y) : \varphi(x, y) = 0\} \tag{3.4}$$

Where $\varphi : \mathbb{R}^2 \to \mathbb{R}$ is C^2-smooth and in some vicinity of $\mathscr{P}$ one has

$$\nabla\varphi(x, y) \neq 0 \tag{3.5}$$

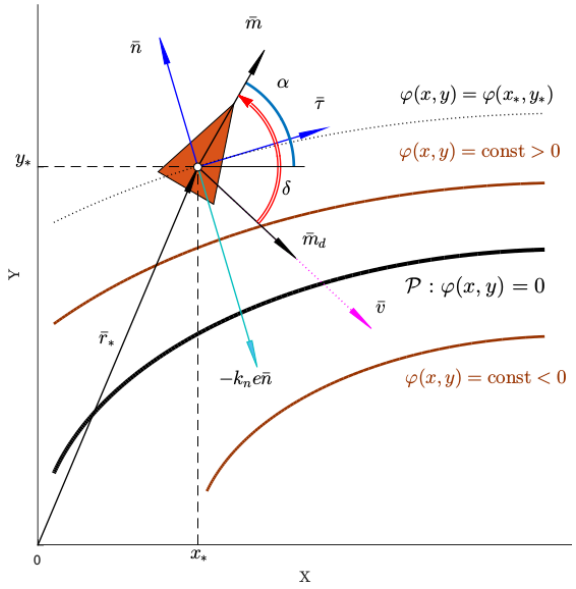


Figure 3.11: Geometrical definitions of vector field algorithm mathematical components.

As illustrated in Figure 3.11 the whole plane $\mathbb{R}^2$ is covered with sets of $\varphi(x,y) = c$. At any point (x_*, y_*) where Equation 3.5 holds, $\bar{n}(x_*, y_*) = \nabla\varphi(x_*, y_*)$ is a normal vector to the path and the unit tangent vector $\bar{\tau}$ is chosen as the basis $[\bar{\tau}, \bar{n}]$ is orthonormal.

Instead of defining an euclidean distance to the path, we consider the *tracking error*

$$e = \psi[\varphi(x,y)] \in \mathbb{R} \tag{3.6}$$

where $\psi : \mathbb{R} \to \mathbb{R}$ strictly increasing, differentiable with $\psi(0) = 0$.

The goal is to design a control law that eliminates this tracking error. We introduce the potential function

$$V(x,y) = \frac{1}{2}e^2(x,y) \tag{3.7}$$

Its derivative regarding the kinematic constraints of the UAV is

$$\dot{V}(x,y) = e(x,y)\psi'(\varphi(x,y))\nabla\varphi(x,y)^\intercal \tag{3.8}$$

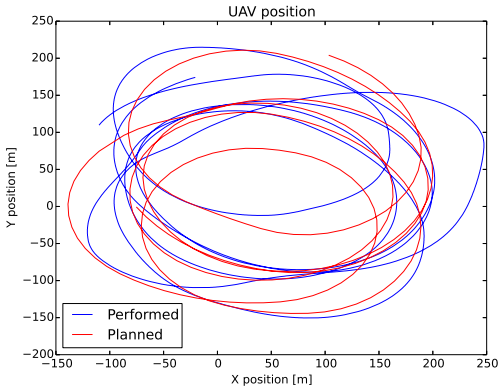
The V function is decreasing, so $|e|$ too, provided $e\bar{n}^\intercal\bar{m} < 0$. Since the UAV must also move at the direction of the path LOS, $\bar{m} \approx \frac{\bar{\tau}}{||\bar{\tau}||}$. At the same time it should be orientated in a way the tracking error is reduced. So the vector field $\bar{v}$ is given by

$$\bar{v}(x,y) = \bar{\tau}(x,y) - k_n e(x,y)\bar{n}(x,y) \tag{3.9}$$

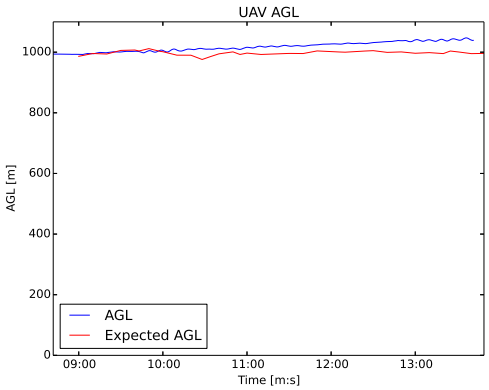
The *guiding vector field*, the desired orientation, is hence

$$\bar{m}_d = \frac{\bar{v}}{||\bar{v}||} \qquad \forall\, \bar{v}(x,y) \neq 0 \tag{3.10}$$

After testing the vector field guidance algorithm (see Figures 3.12 and 3.13), we faced similar issues as with PLOS guidance. Most of the time the UAV stayed close to the path but a few times it got lost, greatly increasing the error. In addition, the airspeed assumption didn't hold this time (Figure 3.14).

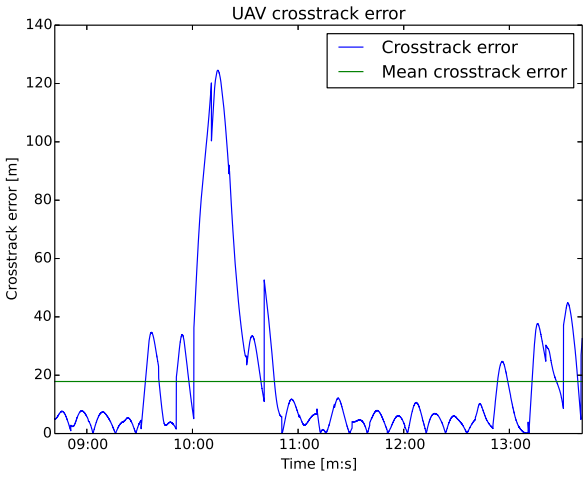


(a) Horizontal guidance. $k_n = 6 \cdot 10^{-5}$



(b) Vertical profile

Figure 3.12: Planned and performed trajectory of a *malolo 1* UAV in FlightGear simulator with VF guidance.



(a) Instantaneous cross-track error

Max $[m]$	Mean $[m]$	Min $[m]$	σ	$CF = \sigma / \bar{x}$
124	17.8	0	25.9	1.45

(b) Statistics after 5 minute flight

Figure 3.13: Cross-track error of trajectory in Figure 3.12

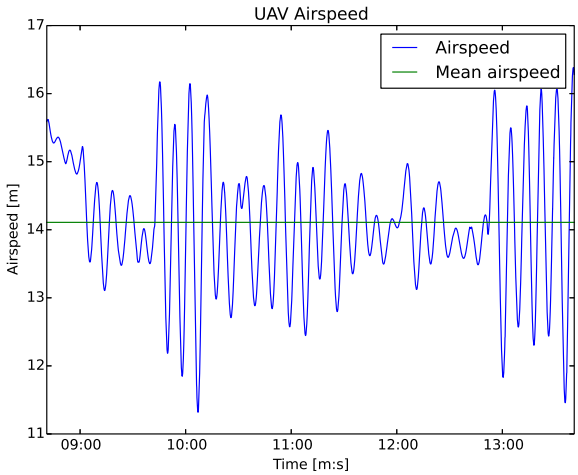


Figure 3.14: Measured airspeed during the flight of Figure 3.12.

Chapter 4

Conclusion and future work

In previous chapters we have introduced the mission of the SkyScanner project. Then, we explained the previous effort on mapping and path planning and how these algorithms work. We justified the need of a new software architecture that could help validate the behaviour of the algorithms.

A new software package, based on the Robot Operating System (ROS), has been implemented and deployed. It integrates several realistic simulators: Paparazzi and FlightGear as flight simulators, and MesoNH to gather atmospheric data. The software architecture was designed with modularity in mind and the easy of transitioning to the operation with real UAVs.

The software integration allowed to identify that Paparazzi cannot make the UAVs follow the tracks provided by the path planner. This is due to problems with the guidance algorithm (*Carrot-chasing*) and the navigation module, which wasn't designed to handle these types of tracks.

Because of the issues, a new objective was added to the internship. In order to solve the problems, we did a survey about different guidance algorithms with their advantages and weaknesses. After that, we implemented two of them (*PLOS* and *Vector Field*), and we added communication with the FlightGear simulator. This showed us that there are guidance algorithms that perform better than the *Carrot-chasing*. However, there where still problems with path-following: we identified that the generated paths are not totally feasible by real UAVs.

As a consequence, for the remaining of the internship, we are exploring other types of curves for the planning process, that should be more easily tracked. Fir this purpose, we will exploit the insights about guidance that we have acquired by testing various solutions. Regarding Paparazzi, we are in close contact with its developers, and they are now improving the guidance and navigation modules to be able to handle more complex trajectories.

Appendix

Improvements to the ROS framework

The ROS framework provide some tools to inspect and supervise running ROS packages. The *rqt* software is the graphical way to present this information and it works through plugins that bring the functionality. There is a plugin to show relationships between nodes and topics (*rqt_graph*), another one to inspect topics (*rqt_topic*) and some other one to plot numerical values of topic messages against time (*rqt_topic*). However there is not any plugin to plot one message field against another, like when plotting a trajectory, so it was required to develop a new plugin for this task.

Based on *rqt_plot*, an improved version named *rqt_plotxy* was implemented. It takes two. The SkyScanner's new software architecture use it amply to plot UAV trajectories. *rqt_plotxy* takes messages from both topics and matches them in time, using the time stamp if a header is included or by reception time otherwise.

A request to include *rqt_plotxy* in the default set of *rqt* plugins has been sent to ROS developers.

Statistics node

In order to have an online performance measure of the system, a statistics module has been included. It is optional to run it and can be launched and stopped at any time.

The */stats* ROS node takes, for each aircraft, the actual state and the expected state and trajectory. With the first it calculates the error in altitude and with the second the crosstrack error. The crosstrack-error is determined as the euclidean distance from the aircraft to the expected path.

Finally it publishes these performance measures in a */stats* topic for each UAV, so it can get recorded or processed by other topics.

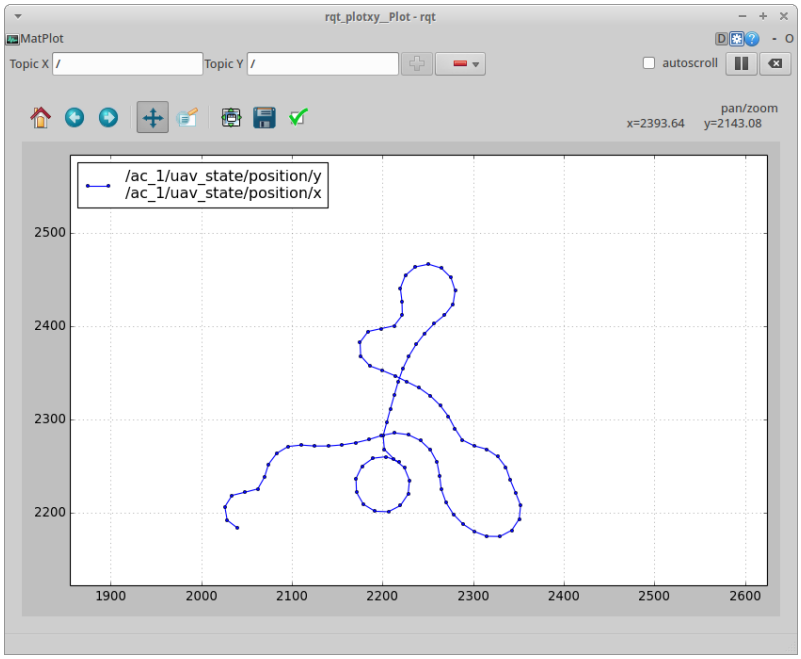


Figure 1: Screenshot of *rqt_plotxy*

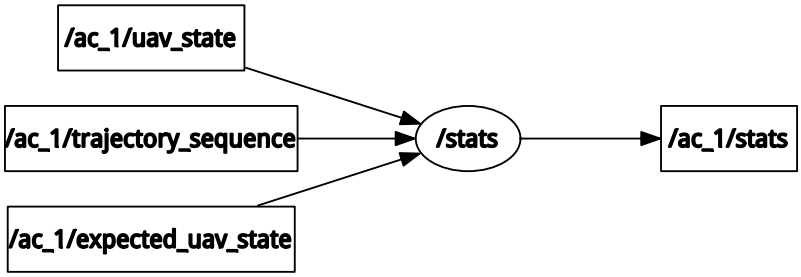


Figure 2: ROS graph of *stats* node

Bibliography

[1] A. Renzaglia, C. Reymann, and S. Lacroix, "Monitoring the Evolution of Clouds with UAVs", presented at the IEEE International Conference on Robotics and Automation, May 16, 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01275324/document (visited on Apr. 6, 2016).

[2] P. B. Sujit, S. Saripalli, and J. B. Sousa, "Unmanned Aerial Vehicle Path Following: A Survey and Analysis of Algorithms for Fixed-Wing Unmanned Aerial Vehicless", *IEEE Control Systems*, vol. 34, no. 1, pp. 42–59, Feb. 2014, ISSN: 1066-033X. DOI: 10.1109/MCS.2013.2287568.

[3] J. Song and S. Song, "Three-dimensional guidance law based on adaptive integral sliding mode control", *Chinese Journal of Aeronautics*, vol. 29, no. 1, pp. 202–214, Feb. 2016, ISSN: 1000-9361. DOI: 10.1016/j.cja.2015.12.012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1000936115002411 (visited on Jun. 2, 2016).

[4] F. Tyan, "Unified approach to missile guidance laws: a 3D extension", *IEEE Transactions on Aerospace and Electronic Systems*, vol. 41, no. 4, pp. 1178–1199, Oct. 2005, ISSN: 0018-9251. DOI: 10.1109/TAES.2005.1561882.

[5] T. I. Fossen, K. Y. Pettersen, and R. Galeazzi, "Line-of-Sight Path Following for Dubins Paths With Adaptive Sideslip Compensation of Drift Forces", *IEEE Transactions on Control Systems Technology*, vol. 23, no. 2, pp. 820–827, Mar. 2015, ISSN: 1063-6536. DOI: 10.1109/TCST.2014.2338354.

[6] H. E. Núñez, G. Flores, and R. Lozano, "Robust path following using a small fixed-wing airplane for aerial research", in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, Jun. 2015, pp. 1270–1278. DOI: 10.1109/ICUAS.2015.7152420.

[7] S. A. H. Tabatabaei, A. Yousefi-koma, M. Ayati, and S. S. Mohtasebi, "Three dimensional fuzzy carrot-chasing path following algorithm for fixed-wing vehicles", in *2015 3rd RSI International Conference on Robotics and Mechatronics (ICROM)*, Oct. 2015, pp. 784–788. DOI: 10.1109/ICRoM.2015.7367882.

[8] Y. A. Kapitanyuk, A. V. ProskurnikovAnton V., and M. Cao, "A guiding vector field algorithm for path following control of nonholonomic mobile robots", 2016, submitted.