

POLITECNICO DI MILANO

School of Industrial and Information Engineering

Computer Science and Engineering Course

Dipartimento di Elettronica, Informazione e Bioingegneria



Consensus based control for a Unmanned

Aerial Vehicle formation flight

Advisor: Prof. Marco LOVERA

Co-Advisors: Eng. Pietro PANIZZA

Eng. Mattia GIURATO

Thesis by:

Alex DELBONO Matr. 850114

Academic Year 2016–2017



*A Pietro, una persona vera ed altruista come poche...*



# **Ringraziamenti**



# Abstract

This thesis is about the synchronized flight of a formation of multirotors which execute a mission. The mission is characterized by a trajectory for each drone which composes the formation. The formation must be able to deal with unforeseen events, which can compromise the outcome of the mission. In order to do that, the drones must exchange information with each others and a support network is therefore needed.

The purpose of this thesis is to present the algorithm for synchronized formation flight called “Consensus algorithm” and implement it in a simulated environment and in a real system with heterogeneous drones. We want to verify that the theoretical results can be applied in a real distributed system, with not ideal network performances.

In the first part, we explain the algorithm used during the experimental work, while in the following chapter the software and hardware used are presented. In the last chapters, we show in detail the structure of the software and the experiment conducted. In particular, many simulations will be presented to confirm the quality of the “Consensus algorithm”. Finally, a comparison between the simulated results and the ones obtained in the real environment is proposed.



# Sommario

La tesi tratta di volo sincronizzato di una formazione di multirotori che eseguono una missione, caratterizzata da una traiettoria per ogni drone che compone la formazione. La formazione deve essere in grado di reagire a eventi inaspettati, che possono compromettere l'esito della missione stessa. Per raggiungere tale scopo, i droni devono scambiare informazioni, attraverso una rete di supporto.

L'obbiettivo della tesi è di presentare l'algoritmo utilizzato per il volo sincronizzato, chiamato “Algoritmo di Consenso” e implementarlo sia in un ambiente simulato, che in un sistema reale, composto da macchine eterogenee. Si vuole verificare che i risultati teorici possano essere applicati in un sistema reale distribuito, dotato di una rete di comunicazione dalle prestazioni non ideali.

Nella prima parte della tesi, verrà esposto l'algoritmo usato durante il lavoro sperimentale, mentre nel capitolo seguente saranno presentati l'hardware e il software. Negli ultimi capitoli saranno evidenziati nel dettaglio la struttura del software e gli esperimenti condotti. In particolare, saranno mostrate alcune simulazioni, in modo da confermare la bontà dell’“Algoritmo di Consenso”. Infine, nell'ultimo capitolo, sarà proposta una comparazione tra i risultati delle simulazioni e quelli ottenuti mediante l'implementazione in un sistema reale.



# Contents

<b>Ringraziamenti</b>	<b>5</b>
<b>Abstract</b>	<b>7</b>
<b>Sommario</b>	<b>9</b>
<b>List of figures</b>	<b>17</b>
<b>Introduction</b>	<b>19</b>
<b>1 Consensus theory</b>	<b>25</b>
1.1 Parametrized trajectory . . . . .	27
1.2 Virtual time . . . . .	30
1.3 Consensus law . . . . .	32
1.4 Network topology . . . . .	36

1.5 Convergence property . . . . .	37
<b>2 System architecture</b>	<b>39</b>
2.1 Hardware . . . . .	39
2.1.1 Pixfalcon . . . . .	40
2.1.2 Intel Edison . . . . .	42
2.1.3 RaspberryPi Zero . . . . .	42
2.1.4 Motive Optitrack . . . . .	43
2.2 Software . . . . .	45
2.2.1 Ground station . . . . .	46
2.2.2 Raspberry Pi Zero . . . . .	47
2.2.3 Intel Edison . . . . .	47
2.2.4 Both companions . . . . .	47
2.2.5 Pixfalcon . . . . .	48
2.2.6 Simulation . . . . .	48
2.2.7 Additional software . . . . .	49
<b>3 Consensus node</b>	<b>51</b>
3.1 Start and stop services . . . . .	53

3.2 Consensus variable callback . . . . .	56
3.3 Local position callback . . . . .	57
<b>4 Simulation results</b>	<b>63</b>
4.1 Trajectory following . . . . .	64
4.2 First disturbance . . . . .	68
4.3 Second disturbance . . . . .	70
<b>5 Experimental results</b>	<b>75</b>
5.1 Trajectory following . . . . .	76
5.2 First disturbance . . . . .	78
5.3 Second disturbance . . . . .	80
<b>Conclusions</b>	<b>83</b>



# List of Figures

2.1	Pixfalcon board	41
2.2	Edison board	42
2.3	RaspberryPi Zero board	43
2.4	Motive Optitrack screenshot	44
2.5	Optitrack marker	45
2.6	ANT-1 drone	45
2.7	Hexa drone	46
3.1	Input and output of the node	53
3.2	Custom service structure	55
3.3	Custom message structure	56
3.4	Consensus variable class	58
3.5	Class used to manage the position and velocity of the drones	59

---

3.6	Class used to manage a generic trajectory . . . . .	61
4.1	Iris model . . . . .	64
4.2	Trajectory . . . . .	65
4.3	Evolution of the trajectory in time of the first drone . . . . .	66
4.4	Target following drone 1 . . . . .	66
4.5	Target following drone 2 . . . . .	67
4.6	Positions of the two drones in time . . . . .	67
4.7	Disturbance . . . . .	68
4.8	Target following drone 1 . . . . .	69
4.9	Target following drone 2 . . . . .	69
4.10	Positions of the two drones in time . . . . .	70
4.11	Disturbance . . . . .	71
4.12	Target following drone 1 . . . . .	71
4.13	Target following drone 2 . . . . .	72
4.14	Positions of the two drones in time . . . . .	72
5.1	Target following drone 1 . . . . .	76
5.2	Target following drone 2 . . . . .	77

5.3	Positions of the two drones in time . . . . .	77
5.4	Target following drone 1 . . . . .	78
5.5	Target following drone 2 . . . . .	79
5.6	Positions of the two drones in time . . . . .	79
5.7	Target following drone 1 . . . . .	80
5.8	Target following drone 2 . . . . .	81
5.9	Positions of the two drones in time . . . . .	81



# Introduction

In recent years, the field of robotics had developed rapidly. The growing applications of mobile robots and drones in many fields has brought an important increase in the number of robotic vehicles. Many commercial solutions are available, but, in many cases, they offer a product which needs an expert pilot. For this reason and due to the high prices, the diffusion of mobile robots is limited. In the latest years, these problems were mitigated and we now find lower prices and easier-to-use products. This will boost the diffusion of non-professional robotic vehicles in our houses and in our daily activities. Applications of UAVs are the ones related to exploration and data collection. Indeed, drones are fundamental in inspection of unknown areas, such as forests or unreachable terrains. Moreover, the monitoring of industrial artifacts constitutes a valuable application of UVAs, such as for instance, solar panels, wind turbines, high-voltage cables or tubes can be supervised using aerial vehicles. The multimedia production is the field in which the drones are most present. Indeed, most of the machines are equipped with high-definition cameras in order to provide professional videos.

The power consumptions of the machines has been significantly reduced in the recent years and we are now able to build smaller and lighter vehicles. In partic-

ular, in the aerial field there are commercial products which weight about 200g with a flight time of 20 minutes. These improvements allow us to develop more advanced features, such as trajectory planning, obstacles avoidance, formation flight. The development of autonomous UAVs is fundamental for humanitarian response around the world. When a natural calamity happens, the UAVs can play a crucial role by delivering essential goods or finding missing people. Moreover, the operations can be done without risking the safety or the life of the rescuers, because they can act remotely or plan an autonomous mission.

The trajectory planning is essential for the development of autonomous vehicles, because the generation of a feasible trajectory for a mission is required, otherwise no mission can be done. The trajectory generation is a difficult problem, because of its computational complexity. Indeed, many suboptimal algorithms have been developed in order to reduce this complexity. There are different classes of planning algorithm and they can be summarized as follows:

- Artificial Potential Field: these algorithms assign a value of the potential for every point in the map. The goal has the lowest (highest) value and the obstacles have high (low) values. Then, the robot tries to descend (climb) the potential, in order to reach the goal.
- Sampling Based Planning: random samples are used to find a path from the starting point to the goal. Advanced random samples techniques can be used in order to reduce the complexity of the algorithm.
- Grid Based Planning: this kind of algorithm overlays a grid on the map, so every configuration corresponds with a grid pixel. The robot can move from

one grid pixel to any adjacent grid pixels as long as that grid pixel does not contain an obstacle.

- Reward-Based Planning: the robot can apply different actions in every state of the world. The outcome of an action can be not deterministic and after every action, the robot gains a reward. The objective of the robot is to maximize the sum of the rewards.

Another important aspect is the obstacle avoidance. The trajectory is usually planned offline, when the mission is not started yet. So, the trajectory does not take into account the fact that an unplanned obstacle might interfere with the mission. In this case another algorithm is implemented to run online. This kind of algorithms are called obstacle avoidance algorithms and they plan again locally the trajectory, in order to get rid of unforeseen events. These algorithms need the drone to be equipped with proximity sensors, which provide information about the surrounding environment. Different methodologies have been developed for obstacles avoidance. The main ones are summarized below without details.

- Artificial Potential Field: as before, the idea of this approach is that obstacles exert a virtual repulsive force, given by the potential field, to push away the robot from them while the goal position generates a virtual attractive force to guide the robot to it.
- Virtual Force Field method (VFF): is the combination of the Artificial Potential Field with the concept of Probabilistic Occupancy Grid maps.
- Fuzzy Controller for Obstacle Avoidance: as the name says, a fuzzy controller is used to derive the variables for the vehicle's orientation and accel-

eration control, depending on the current perception of the robot's sensors

- Vector Field Histogram method (VFH): improved version of the VFF method.
- VFH+, VFH\* method: improved versions of the VFH method, but more computational costly. The most recent one is the VFH method, which uses an A\* search.
- Traversability Field Histogram (TFH): The local path planner bases on the VFH concept, but is extended by the information provided within the Traversability Map.
- Dynamic Window Approach: the algorithm takes into account the dynamic and kinematic constraints of the robot and it is similar to the VFH+ algorithm. The method is called dynamic window approach and considers only admissible velocities which can be reached within the next time interval and which allow the robot to stop safely.
- Curvature Velocity Space method: This method chooses a point in the linear-angular velocity space which satisfies some constraints and maximizes an objective function. This objective function tries to move the robot close to the commanded direction at the highest feasible speed, while travelling the trajectory with the largest clearance from obstacles.
- Beam Curvature method: improvement for the curvature velocity space method.
- Nearness diagram Navigation: it is similar to the VFH method, but uses a polar histogram to derive actions to be taken for the robot.

When multiple machines have to fly in a formation, a synchronization mechanism is needed. For example, if one of the drones has to deviate from the planned trajectory (for instance, because of an unplanned obstacle), the formation must be preserved. To reach the synchronization, vehicles must communicate with each other and exchange information. Notions from graph theory are needed to analyze the behaviour of the system and demonstrate the convergence properties.

This work is focused on the presentation of a distributed synchronization algorithm, which we call “Consensus Algorithm”. Every drone must exchange its mission progression with its neighbors and must adjust its mission progression based on the information received from the other vehicles. The algorithm is a distributed algorithm and it is executed on every vehicle.

In the first chapter, we will see the theoretical aspects of the “Consensus Algorithm” and then, in the second chapter, we will provide a general overview of the hardware and software used to deploy the algorithm. In the third chapter we will take a look at the software implementation of the main part of the system, providing classes and snippets of code. Finally, we will explain the simulated and experimental results of our work in chapters four and five. We will apply the algorithm to a real system and we will present our results and its performances.



# Chapter 1

## Consensus theory

The literature about consensus has grown significantly in the last few years because of the increasing presence of autonomous vehicles. This, in accordance with the new improvements in robotics, has brought to a growing interest in consensus between multiple agents which have to accomplish a mission in cooperative or adversarial scenarios.

Consensus theory has its roots in graph theory and control theory and it can be used to coordinate a mission in order to achieve synchronization between the vehicles even during unforeseen events which force one or more components of the mission to change the planned trajectory or task. In this case, the other components identify this variation and they act to preserve the synchronization.

A consensus algorithm is a distributed algorithm which can sometimes be simulated in a centralized fashion because of the reduced computational power of the machines involved in the mission, which are equipped with low power hardware to

account for the crucial issue of power consumption. In this scenario, a centralized server simulates the algorithm and communicates with all the machines.

The most common consensus application is a spatial and timing consensus: in this implementation, each vehicle of the formation has to travel along a specified trajectory and the completion of the mission occurs when all the agents reach the final positions of their spatial paths. The algorithm has to guarantee that the difference between the ending time at which all the vehicles finish their tasks is minimized and asymptotically goes to zero, when the execution time goes to infinity and no other unforeseen event happens. We consider formations of Unmanned Aerial Vehicles (UAVs), but the key concepts can be freely applied to other categories of robots when they are able to follow a trajectory. This study is focused on this kind of application, and further simulation results and experimental achievements are presented in Chapters 4 and 5.

In the next sections we refer to the main work of Cichella et al.[5] in the field of UAV consensus, in order to provide an homogeneous state of the art about consensus. The paper considers all the details needed to build a consensus system. The main components of this kind of system are listed and explained below:

- Parametrized trajectory
- Virtual time
- Consensus law
- Network topology.

## 1.1 Parametrized trajectory

The trajectory is a spatial path with an associated timing law and it is used to identify the position of the center of mass of our agents at a given time. First of all, the trajectory used during our experiments are polynomials curves. In particular, we use Bézier curves, because of the existence of many computationally efficient algorithms designed for this kind of curves, such as algorithms to efficiently compute the minimum distance between two Bezier curves and the existence of a closed-form solution for the arc lengths of the paths.

First of all, let  $I = \{N, E, D\}$  denote a right-handed inertial frame with  $N$ ,  $E$  and  $D$  unit vectors along north, east and down respectively. The vector  $p = (x, y, z) \in I$  denotes the position of the center of mass of the vehicle. Let  $B = \{X_B, Y_B, Z_B\}$  be right-handed body fixed frame centered in the center of mass of the vehicle. The orientation of the rigid body is given by rotation matrix  $R(\phi, \theta, \psi)$  where  $\phi$ ,  $\theta$  and  $\psi$  are the roll, pitch and yaw Euler angles respectively. We can state the following definition of a generic trajectory  $\mathbf{p}_{d,i}(t_d)$  for  $N$  vehicles:

$$\mathbf{p}_{d,i} : [0, t_{d,i}^f] \rightarrow \mathbb{R}^4, \quad i = 1, 2, \dots, N \quad (1.1)$$

where  $t_d \in [0, T_d]$ , with  $T_d := \max\{t_{d,1}^f, \dots, t_{d,N}^f\}$ , is the time variable of the trajectory, while  $t_{d,i}^f \in \mathbb{R}^+$  are the individual final mission times of the vehicles obtained during the planning phase. Usually all these final times are equal and therefore  $t_{d,1}^f = \dots = t_{d,N}^f = T_d$ , but we introduced the notation for the sake of

generality. Obviously, the trajectories need to be collision-free and must comply with spatial and temporal constraints due to the dimensions of the vehicles and their maximum velocities and accelerations. The range of the function is  $\mathbb{R}^4$  because, in the case of UAVs, we want to control  $x, y, z, \psi$ . The trajectory can account also for *roll* and *pitch*, and therefore, the function might take images in  $\mathbb{R}^m$ , where  $m$  is the number of dimensions considered. In the following sections, we will refer only to the position and  $\psi$ , but a more general theory can also be developed.

We now parametrize the trajectory using a dimensionless variable  $\zeta_i \in [0, 1]$ , related to the time  $t_d$ . In this way we can specify a function  $\theta(\cdot)$ , which represents the timing law associated with the spatial path  $\mathbf{p}_{d,i}(\zeta_i)$ . We can specify this timing law using the dynamic relation:

$$\theta(t_d) = \frac{d\zeta_i}{dt_d} \quad (1.2)$$

where  $\theta(t_d)$  is a smooth and positive (the parameter increases when the time increases) function.

As it allows a one-to-one correspondence between the time variable  $t_d$  and the parameter  $\zeta_i$ , an analytical expression for the function  $\zeta_i(t_d)$  is desirable. Using

the timing law defined in (1.2), the map  $\zeta_i(t_d)$  is given by the integral:

$$\zeta(t_d) = \int_0^{t_d} \theta_i(\tau) d\tau \quad (1.3)$$

Usually, all these functions are defined as polynomials in order to make the evaluation process quicker and easier, since multiplication and addition are the basic operations in a digital processor. However, it is not mandatory to use them and a generic shape for the functions can be designed.

For the spatial path we use quintic Bezier curves for the  $x$ ,  $y$  and  $z$  coordinates, while we use a third degree polynomial for the  $\psi$  of the drone since we need less elaborate shapes. Since we are using drones with a decoupled control of  $x$ ,  $y$ ,  $z$  and  $\psi$ , the polynomials are completely independent.

The timing law associated to these curves is a Bezier third order polynomial. So, in our case, we have five polynomials which have to be evaluated in real time in order to provide the set points. The frequency at which we evaluate the trajectory is  $5Hz$ .

The trajectory which describes the mission of a single drone is given by:

$$\begin{cases} x_i(\zeta_i) = \sum_{j=0}^5 \bar{x}_{i,j} b_j^5(\zeta_i) \\ y_i(\zeta_i) = \sum_{j=0}^5 \bar{y}_{i,j} b_j^5(\zeta_i) \\ z_i(\zeta_i) = \sum_{j=0}^5 \bar{z}_{i,j} b_j^5(\zeta_i) \\ \psi_i(\zeta_i) = \sum_{j=0}^3 \bar{\psi}_{i,j} b_j^3(\zeta_i) \\ \zeta_i(t_d) = \sum_{j=0}^3 \bar{\zeta}_{i,j} b_j^3(t_d) \end{cases}$$

where  $\bar{x}_{i,j}$ ,  $\bar{y}_{i,j}$ ,  $\bar{z}_{i,j}$ ,  $\bar{\psi}_{i,j}$  and  $\bar{\zeta} \in \mathbb{R}$  are the desired control points of the spatial path and  $b_n^m$  are the (up to degree  $m$ ) Bernstein basis polynomials. We do not enter in details about the Bezier curves and Bernstein basis polynomials, because it is not the objective of this work.

We have defined all the elements of a trajectory and we do not go into detail about the trajectory generation phase. Further information about boundary conditions and flyable trajectory, which satisfy the dynamic constraints of the vehicles, can be found in [5] and is extensively analyzed in [4], [24], [26].

## 1.2 Virtual time

Given  $N$  collision free trajectories, we want each vehicle to follow a *virtual target*, moving along the path computed offline by the trajectory generation algorithm. The objective can be achieved introducing a *virtual time*,  $\gamma_i$ , which is used to evaluate the trajectory and can be adjusted online to reach the synchronization

even when external disturbances occur. Thus, the position of the  $i^{th}$  virtual target is denoted by  $\mathbf{p}_{d,i}(\gamma_i(t))$  and the  $i^{th}$  vehicle tries to follow it, by reducing to zero a suitably defined error vector using control inputs.

Considering the trajectory  $p_{d,i}(t_d)$  produced by the trajectory generation algorithm, we consider the virtual time  $\gamma_i$  as a function of time  $t$ , which relates the actual time  $t$  to mission planning time  $t_d$ .

$$\gamma_i : \mathbb{R}^+ \rightarrow [0, T_d], \quad \text{for all } i = 1, 2, \dots, N. \quad (1.4)$$

We can now define the virtual target's position, velocity and acceleration, which have to be followed by the  $i^{th}$  vehicle at time  $t$

$$\begin{aligned} \mathbf{p}_{c,i}(t) &= \mathbf{p}_{d,i}(\gamma_i(t)) \\ \mathbf{v}_{c,i}(t) &= \dot{\mathbf{p}}_{d,i}(\gamma_i(t), \dot{\gamma}_i(t)) \\ \mathbf{a}_{c,i}(t) &= \ddot{\mathbf{p}}_{d,i}(\gamma_i(t), \dot{\gamma}_i(t), \ddot{\gamma}_i(t)). \end{aligned} \quad (1.5)$$

With the above formulation, if  $\dot{\gamma} = 1$ , then the speed profile of the virtual target is equal to the desired speed profile computed at trajectory generation level. Indeed, if  $\dot{\gamma} = 1$ , for all  $t \in [0, T_d]$ , with  $\gamma_i(0) = 0$ , it implies that  $\gamma_i(t) = t_d$  for all  $t$  and thus:

$$\mathbf{p}_{c,i}(t) = \mathbf{p}_{d,i}(\gamma_i(t)) = \mathbf{p}_{d,i}(t) = \mathbf{p}_{d,i}(t_d).$$

In this particular case, the desired and commanded trajectories coincide in every time instant and also the velocity profiles coincide with the ones chosen at the trajectory generation time. If instead  $\dot{\gamma}_i > 1$ , it implies a faster execution of the mission; on the other hand,  $\dot{\gamma}_i < 1$  implies a slower one.

We can now normalize  $\gamma_i$  in order to have a range which is  $[0, 1]$ . We simply need to divide all by  $T_d$ . In this way, we could use Bezier curves in order to represent the spatial path. This kind of curves offer interesting properties for computing minimum distances between two of them and allow the computation of smooth trajectories.

The second order derivative of  $\gamma_i$ ,  $\ddot{\gamma}_i$ , is a free parameter used to achieve the consensus. In the next section we will introduce the control law which commands its evolution during time and we will explain how it is possible to implement a distributed algorithm.

### 1.3 Consensus law

Now, we formally state the path following problem. We define as  $\mathbf{p}_i(t) \in \mathbb{R}^4$  the position of the center of mass of the  $i^{th}$  agent and its  $\psi$  and since  $\mathbf{p}_{c,i}(t)$  describes the commanded pose to be followed by the agent at time  $t$ , the errors are defined

as:

$$\begin{aligned}\mathbf{e}_{p,i}(t) &= \mathbf{p}_{c,i}(t) - \mathbf{p}_i(t) \in \mathbb{R}^4 \\ \mathbf{e}_{v,i}(t) &= \mathbf{v}_{c,i}(t) - \dot{\mathbf{p}}_i(t) \in \mathbb{R}^4.\end{aligned}\tag{1.6}$$

Then, the objective reduces to that of regulating the error defined in (1.6) to a neighbourhood of zero. This task is solved with an autopilot capable of following the set points computed from the desired trajectory at specified instances of time.

The virtual time is the parameter used to reach consensus between multiple vehicles. In fact, since the the trajectories are parametrized by  $\gamma_i$ , the agents are synchronized at time  $t$  when:

$$\gamma_i(t) - \gamma_j(t) = 0 \quad \text{for all } i, j \in \{1, \dots, N\}, \quad i \neq j.\tag{1.7}$$

We can also control the rate of progression of the mission using a parameter  $\dot{\gamma}_d \in \mathbb{R}$ , which represents the velocity of the virtual time with respect to the real time. All the agents share this variable and they proceed at the same rate of

progression if:

$$\dot{\gamma}_i(t) - \dot{\gamma}_d(t) = 0 \quad \text{for all } i \in \{1, \dots, N\}. \quad (1.8)$$

Adjusting  $\dot{\gamma}_d$ , we can decide the speed of the mission: for instance, if we set  $\dot{\gamma}_d = 1$  and (1.7) and (1.8) are satisfied for all the vehicles, then the mission is executed at the speed originally planned in the trajectory generation phase. If instead we use  $\dot{\gamma}_d > 1$  or  $\dot{\gamma}_d < 1$  we carry out the mission faster or slower. This term can be changed in real time in order to avoid moving objects or unplanned obstacles, which make it necessary to change the path of one of the agents. For the purpose of consensus, the parameter is only a reference command, rather than a control input.

Now, we introduce the coordination control law which regulates the evolution of  $\ddot{\gamma}_i(t)$  during the time and determines  $\gamma_i(t)$ :

$$\begin{aligned} \ddot{\gamma}_i(t) &= \ddot{\gamma}_d(t) - b(\dot{\gamma}_i(t) - \dot{\gamma}_d(t)) - a \sum_{j \in \mathbb{N}_i} (\gamma_i(t) - \gamma_j(t)) - \bar{\alpha}_i(\mathbf{e}_{p,i}(t)) \\ \dot{\gamma}_i(0) &= \gamma_d(0) = 1 \\ \gamma_i(0) &= \gamma_d(0) = 0 \end{aligned} \quad (1.9)$$

where  $a$  and  $b$  are positive coordination control gains, while  $\bar{\alpha}_i(\mathbf{e}_{p,i}(t))$  is defined

as:

$$\bar{\alpha}_i(\mathbf{e}_{p,i}(t)) = \frac{\mathbf{v}_{c,i}^T(t) \mathbf{e}_{p,i}(t)}{||\mathbf{v}_{c,i}(t)|| + \epsilon} \quad (1.10)$$

with  $\epsilon$  being a positive design parameter,  $\mathbf{e}_{p,i}$  the position error vector defined in (1.6) and  $\aleph_i$  the set of the neighbors which can communicate with the  $i^{th}$  vehicle (we will see details later). In equation (1.9) we have four terms. The feedforward term,  $\ddot{\gamma}_d$ , allows the virtual target to follow the acceleration profile of  $\gamma_d$ . The second term,  $-b(\dot{\gamma}_i(t) - \dot{\gamma}_d(t))$ , reduces the error between the speed profile imposed by  $\dot{\gamma}_d(t)$  and  $\dot{\gamma}_i(t)$ , which corresponds to the control objective given in (1.8). In particular, if  $\dot{\gamma}_d(t)$  is equal to one, then the virtual target converges to the desired speed profile chosen in the trajectory generation phase. The third term,  $-a \sum_{j \in \aleph_i} (\gamma_i(t) - \gamma_j(t))$ , ensures that all the vehicles are coordinated with their neighbors as specified in (1.7). Finally, the fourth term,  $-\bar{\alpha}_i(\mathbf{e}_{p,i}(t))$ , is a correction term used to take into account for the path following errors of the agent. Indeed, if the vehicle is behind its target, the term is non-zero and the target slows down in order to wait for the real vehicle.

With this control law, we want our vehicles to be synchronized and to proceed at a desired rate of progression, in order to accomplish the mission even when some unforeseen disturbances occur during the execution phase.

## 1.4 Network topology

To achieve the time-coordination objective, agents must exchange information over a supporting communication network. To analyze the information flow, we need to consider some tools from algebraic graph theory, whose key concepts can be found in [2].

We assume that a vehicle  $i$  exchanges information with only a subset of all vehicles, denoted as  $\aleph_i(t)$ . We assume that arcs of the network are bidirectional and that there are no network delays. The information exchanged is composed by the virtual time of the agents,  $\gamma_i(t)$ .

The topology of the graph,  $\Gamma(t)$ , that represents the communication network, must comply with the following constraint in order to guarantee the convergence of the consensus algorithm:

$$\frac{1}{NT} \int_t^{t+T} QL(\tau)Q^T d\tau \geq \mu I_{N-1}, \quad \text{for all } t \geq 0 \quad (1.11)$$

where  $L(t) \in \mathbb{R}^{N \times N}$  is the Laplacian of the graph  $\Gamma(t)$  and  $Q \in \mathbb{R}^{(N-1) \times N}$  is a matrix such that  $Q1_N = 0$  and  $QQ^T = I_{N-1}$ , with  $1_N$  being a vector in  $\mathbb{R}^N$  whose components are all 1s. In (1.11), the parameters  $T > 0$  and  $\mu \in (0, 1]$  represent a measure of the level of connectivity of the communication graph. This condition requires the graph  $\Gamma(t)$  to be connected only in an integral sense, not pointwise in time. Therefore, even if the graph were disconnected during the mission at some

interval of time, the convergence of the consensus algorithm would still be possible. With this condition, we can capture also packets dropouts, loss of communication and switching topologies, which can all occur during the mission, but these events do not necessarily break the convergence property.

## 1.5 Convergence property

The control law given by (1.9) guarantees that the error of the consensus algorithm converges to zero exponentially. It can be shown that the maximum convergence rate is given by the sum of the convergence rate of the path following error and the term

$$\frac{a}{b} \frac{N\mu}{T(1 + (a/b)NT)^2} \quad (1.12)$$

which depends on the control gains  $a$  and  $b$ , the number of vehicles  $N$  and the quality of service of the communication network, characterized by the parameters  $T$  and  $\mu$ . If we fix the gains and the number of the vehicles, the convergence rate depends only on the amount of information which the agents exchange each other over time.



# **Chapter 2**

## **System architecture**

In this chapter, we will describe the hardware and software architecture of the system. Most of the software is completely decoupled from the hardware part and can be run on different machines, allowing portability and reusability. On the contrary, the code is developed only for the specific hardware; mostly the code which is more related to the specific functionalities of the hardware itself. We will have an overview of the hardware used and then we will show a general software architecture and its deployment on the machines involved.

### **2.1 Hardware**

The hardware used to run the system is heterogeneous and we will show in detail the machines involved in the project.

A desktop pc, whose characteristics are listed below, will be our ground station:

- Processor: Intel Core(TM) i5-6500 CPU @ 3.20GHz
- Memory: 16 GB
- Network: Intel Gigabit CT Network Adapter
- Storage: 230 GB

On this machine, we will run a virtual machine which has the following specifications:

- Processor: 1 Core
- Memory: 4 GB
- Storage 25 GB
- Network: Virtual adapter

The flying vehicles involved in the mission are of two kinds, but both adopt the same general configuration, even if with different hardware. Indeed, they are equipped with a flight control unit connected to a companion microcomputer by the serial port. The microcomputer communicates with the ground station through the Wi-Fi connection.

### 2.1.1 Pixfalcon

The flight control unit adopted is the Pixfalcon, (Figure 2.1) which belongs to the family of the Pixhawk [18].

Its specifications are the following:



Figure 2.1: Pixfalcon board

- Main System-on-Chip: STM32F427
  - CPU: 180 MHz ARM Cortex M4 with single-precision FPU
  - RAM: 256 KB SRAM (L1)
- Failsafe System-on-Chip: STM32F100
  - CPU: 24 MHz ARM Cortex M3
  - RAM: 8 KB SRAM
- Wifi: ESP8266 external
- GPS: U-Blox 7/8 (Hobbyking) / U-Blox 6 (3D Robotics)
- Connectivity:
  - 1x I2C
  - 1x CAN (2x optional)
  - 1x ADC
  - 4x UART (2x with flow control)
  - 1x Console
  - 8x PWM with manual override

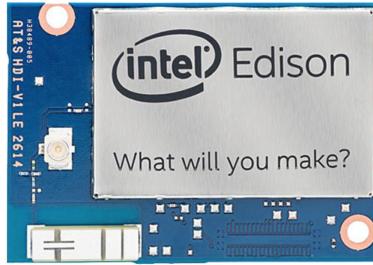


Figure 2.2: Edison board

- 6x PWM / GPIO / PWM input
- S.BUS / PPM / Spektrum input
- S.BUS output

### 2.1.2 Intel Edison

The companion computers are of two types. The first kind is the Intel Edison [14] (Figure 2.2) which is a general purpose computer with the following specifications:

- Atom 2-Core (Silvermont) x86 @ 500 MHz
- Memory: LPDDR3 1 GB
- Storage: 4 GB EMMC

### 2.1.3 RaspberryPi Zero

The second kind of companion is the RaspberryPi Zero [10] (Figure 2.3), with the following specifications.

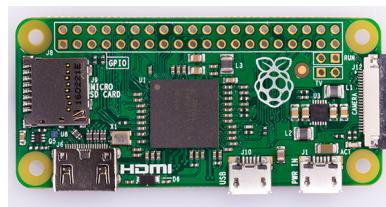


Figure 2.3: RaspberryPi Zero board

- Processor:
  - Broadcom BCM2835
  - contains an ARM1176JZFS (ARM11 using an ARMv6-architecture core)
- Memory: 512MB LPDDR2 SDRAM
- USB On-The-Go port
- Mini HDMI
- 40pin GPIO header
- CSI camera connector

#### 2.1.4 Motive Optitrack

The motion capture system is Motive Optitrack [20], we use eight Optitrack Prime 13 cameras [21], arranged on a square as show in Figure 2.4. The cameras define a volume, which is a cube of 5 meters per side, and the drones can fly without obstacles inside it.

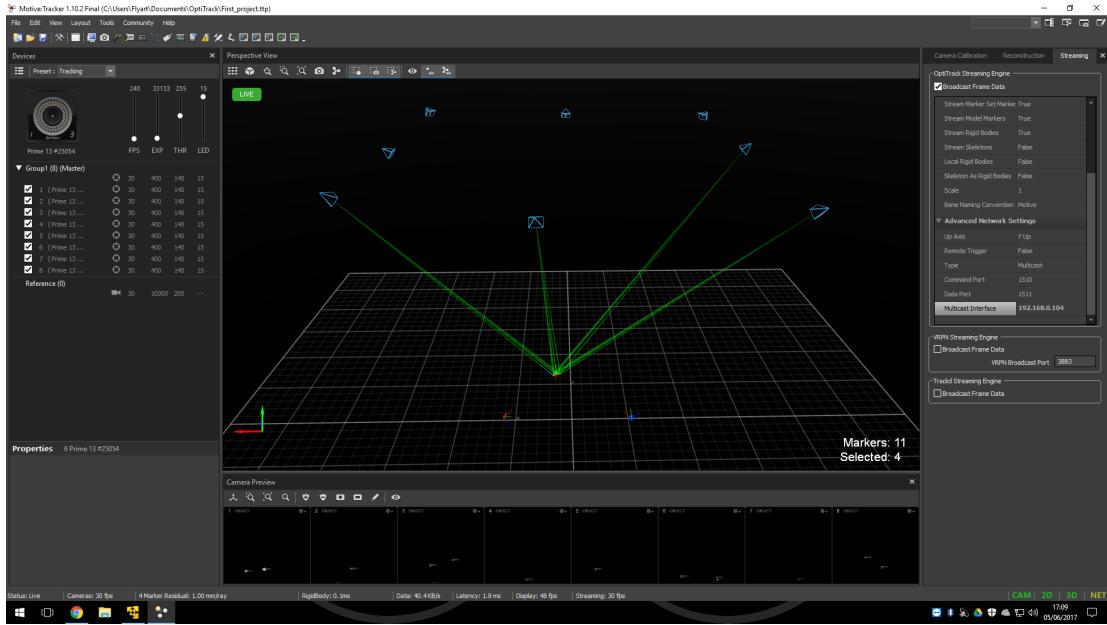


Figure 2.4: Motive Optitrack screenshot

The cameras are connected to a Netgear Prosafe 28PT GE POE [22] switch through Gigabit Ethernet cables.

In order to be seen by the Optitrack system, every drone must be equipped with markers (Figure 2.5) with different configurations, which differentiate the drones from each other.

The two models of drones which we will use are presented in Figures 2.6 and 2.7. The first one is the ANT-1 model, a 200g drone, which is equipped with the Raspberry Pi Zero and the Pixfalcon. The second one is the Hexa model. It is provided with the Intel Edison board and the Pixfalcon board. It is larger and its diameter is approximately 40cm. As we can see from the figures, the ANT-1 model has four propellers, while the Hexa has six.



Figure 2.5: Optitrack marker



Figure 2.6: ANT-1 drone

## 2.2 Software

We now list the software adopted to execute the algorithm in a real environment and in the simulated one.

The principal software used to manage the distributed architecture is ROS Kinetic Kame [7]. ROS is a robotic middleware with a structure which is mainly publisher-subscriber which can manage more machines in a distributed environ-



Figure 2.7: Hexa drone

ment. The central part of the ROS architecture is a node called ROS core, which manages the topics of the system and the subscriptions. The ROS core offers also other functionalities, such as the Parameter Server or the possibility to advertise services. The Parameter Server is a central infrastructure, which is responsible for storing configuration parameters loaded by the nodes of the system. These parameters can be retrieved by other nodes and used if necessary. Instead, a ROS service is a sort of remote function call. One node can advertise the service, which can be called by any other node. The call is synchronous, so the caller is blocked until the callee has executed its callback function. The ROS architecture is based on queues, threads and callback functions, but most of the provided tools hide part of the implementations of the distributed environment.

### 2.2.1 Ground station

The ground station runs Windows 10 Pro [19] and the software used to virtualize a Desktop machine is VMware [25]. On the virtual machine is installed Ubuntu 16.04 LTS [3] in order to run software needed and available only for Unix systems.

On Windows operating system we launch the Motive Optitrack software [20], which allows to calibrate and control the cameras for position tracking. It then provides the streaming of the positions of the markers identified by the cameras and sends it to the Ubuntu operating system using a multicast IP address. Here, the information is converted by a ROS node and sent through the ROS topics, which are read by the drones. In this way, each drone knows exactly its position. This conversion node is an open source node called Mocap which can be found on GitHub [13]. On Ubuntu side, we launch the ROS core, which manages all the ROS nodes and topics.

### 2.2.2 Raspberry Pi Zero

The Raspberry Pi Zero executes a dedicated version of Debian operating system, which is Raspbian. The version used is Raspbian Jessie 4.4 [9].

### 2.2.3 Intel Edison

The Intel Edison runs a version of Debian called Jubilinux, at version 0.1.1 [6].

### 2.2.4 Both companions

Both companions, the Raspberry and the Edison, are provided with ROS Kinetic and both have to execute some ROS nodes in order to communicate with the other drones.

Both of them run Mavros nodes [16], which can be downloaded from GitHub and manage the conversion of the information taken from the ROS topics to the serial port and vice versa. Indeed, the ROS messages are converted into Mavlink messages and sent through the serial port to the Pixfalcon autopilot. The same is done for the Mavlink messages from the autopilot, which are published on ROS topics.

The second kind of ROS node run by the companions is a custom consensus node, which loads the desired trajectory and sends the next set point to the Mavros node. This node will be analyzed in detail in Chapter 3.

### 2.2.5 Pixfalcon

The Pixfalcon FCU is flashed with PX4 Pro Autopilot [17], an open source firmware downloadable from GitHub. The release used is the v1.5.5.

### 2.2.6 Simulation

The simulation part is developed with the utilities provided by the PX4 firmware. In particular, the physical engine used is Gazebo [8], which is an Open Source project.

### 2.2.7 Additional software

We use Matlab R2016\_B [15] to process the data, to plot the graphs and to validate some theoretical results.

This document is written in L<sup>A</sup>T<sub>E</sub>X [23] and the code IDE used is Atom [1], while the versioning control platform used are GitHub [11] and GitLab [12].



# Chapter 3

## Consensus node

In this chapter, we will examine the structure of the consensus node. We will see its main components and we will show some snippets of code in order to make clearer which parts are involved.

As shown in the general architecture (chapter 2), the consensus node is developed as a ROS node, which subscribes and publishes messages to different topics. Moreover, the node offers some ROS services used to start and to stop the trajectory following algorithm or the consensus algorithm.

The structure of the node takes into account the main architectural patterns used in the software development field and it was designed to allow the maximum degree of usability and customization. However, since it has to be executed on machines with limited amount of resources, one of the most important metric taken into account is the efficiency of the code.

The node functionalities are enclosed into a C++ class which initializes all the

ROS elements and prepares the node to receive the start and stop commands. The initialization is done by the class constructor when the object is created. First, in order to apply the consensus dynamic equation, we need the current position of the UAV. The Px4 board already publishes the estimated local position on a topic. Therefore, all we need to do is subscribe to that topic to retrieve the messages with the required information. Second, we want to publish the consensus variable of the drone in the topic used by all the other UAVs, because, having obtained the others' consensus variables from the same topic, we are able to compute the proportional consensus error. Third, since we want to compute the position error and weight it for the target velocity, we need the next set point and the next desired velocity profile. Finally, we compute the acceleration of the consensus parameter using the consensus equation (1.9) and we publish the next set point. We will see the details through the code. All these elements can be summarized and shown in Figure 3.1.

The subscription of a ROS topic works through a callback function, which accepts as parameter the pointer to the new message. Since in our case we have multiple subscriptions and we must advertise the start and stop services, we need to implement a multithreading architecture which takes care of the concurrent accesses to the state of our object. Three threads have been used and their functions are listed below:

- Start and stop services
- Consensus variable callback
- Local position callback

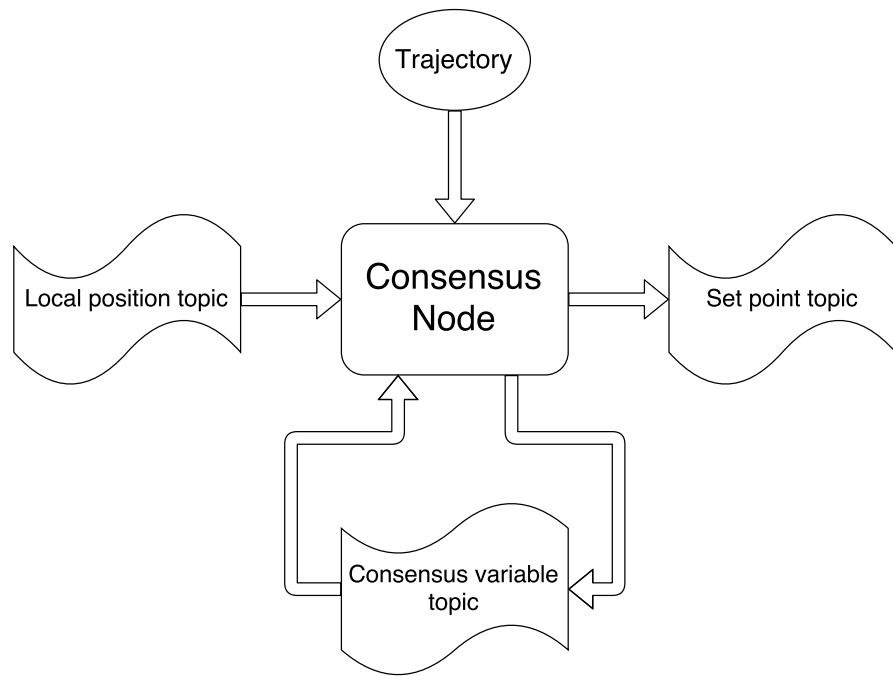


Figure 3.1: Input and output of the node

### 3.1 Start and stop services

The node can work in two different modes:

- trajectory-only
- consensus

On the one hand, in the trajectory-only mode, the node computes the next set point and sends it to the UAV, without considering the possible existence of other UAVs in the mission. Its only objective is to follow the trajectory and reach its final position trying to respect the time constraints imposed by the trajectory. On the other hand, in the consensus mode, the node does the same computation as before, but it also publishes its consensus variable and reads all the other ones. It

then considers this information and adjusts its consensus variable. The consensus mode includes the trajectory-only, and can be started even if the trajectory-only has already been launched, while the opposite is not true. When the consensus mode is stopped, the trajectory-only is stopped as well. Once the mission has been accomplished, the current active mode stops automatically. This means that one of the two modes can be restarted freely without having to stop the previous one.

The services are implemented by using the ROS Service class, which manages the whole infrastructure needed for calling the service. The call of a service is synchronous and the caller is blocked until the service function is terminated. In this case, we offer two services: one for starting and stopping the trajectory-only mode and the other for the consensus mode. It is possible to customize the service call in order to pass different numbers and types of arguments to the service function and to define the response. For the two services, we have defined the same parameters that are shown in the Figure 3.2.

The message is composed by two parts: the request and the response. In the request we need a boolean field in order to know if we want to start or stop the algorithm. The response consists in a boolean variable, which represents the success of the operation, and an exit code, which identifies the eventual problems occurred. The constants for the exit codes are directly specified in the definition of the service.

On the one hand, the trajectory-only service starts or stops the thread which, taken a local position, computes the next setpoint. On the other hand, the consensus one starts or stops the same thread as before and the one which retrieves the consensus variables from the other quadrotors.

```
# Command for enabling or disabling
# trajectory and consensus tasks

#####
## Request fields
bool cmd
___

#####
### exit code constants
# everything ok
uint8 SUCCESS = 0
# trajectory following already
# active and consensus activated
uint8 ESCALATION = 1
# nothing to do
uint8 ALREADY_DONE = 2
# other errors
uint8 FAILURE = 3

#####
# Response fields
bool success
uint8 exit_code
```

Figure 3.2: Custom service structure

```
Header header
string owner
float64 gamma
```

Figure 3.3: Custom message structure

## 3.2 Consensus variable callback

The thread responsible for collecting the consensus variables of all the other UAVs, is managed by ROS and it executes a callback function when a new message is published on a specific topic. This topic is used by all the drones to publish their consensus variable,  $\gamma_i$ , and it accepts a custom message which contains only a string with the name of the owner of the variable and the value itself. The message has also a header, which contains general information such as timestamp or message ID. The structure of the message can be seen in the Figure 3.3.

The callback function receives the information from the topic and updates a local view of the variables of the neighbors. This information has a timeout validity, because we do not want to consider values which are too old. Indeed, if we considered too old values and a problem in the network caused a loss of packets, our drone might think that the other drones have significantly different values of the consensus variables and might therefore wait for them. This is why, it is better to discard these values and remove the neighbors after a timeout.

In order to store this information, we use a thread safe support class, which provides a procedure to check if the variable is expired or not. The signatures of the methods of the class are presented in the Figure 3.4. We use a container to

store the values of the neighbors and we always check if the value has expired or not before using it.

These variables are used in the consensus law (1.9) as  $\gamma_j$  of the neighbors and are used to compute the proportional error. We can see that the expiration interval of the values can model the fact that the network topology can change. Indeed, if a link between two drones vanished because, for instance, they were too far from each other, after a timeout (which is equal to the expiration time), the neighbor would be removed from the container and the drone would not take into account the old neighbor. The timeout can even ignore a failure of a drone: if a machine had a critical problem and did not send its consensus variable, the other drones would remove it from their neighbors and continue their mission without problems.

### 3.3 Local position callback

In the position callback function we apply the consensus law. First of all, we store the actual position of the drone in an object of a custom class, whose signature is presented in Figure 3.5. In this class, we also include, besides x, y and z, the yaw of our vehicle. All the operations defined over the class consider also the orientation.

Then, we compute the synchronization term which corresponds to the sum of the difference between the current  $\gamma_i$  and all the  $\gamma_j$  of the neighbors. In order to do this, we iterate over a container, which stores the values, and we incrementally form the synchronization term.

```

class GammaParameter {
    private:
        std::string owner;
        double gamma;
        double acquisition_time; //In seconds
        std::mutex mtx;
        static double expiration_interval; //In seconds

    public:
        GammaParameter();
        GammaParameter(const GammaParameter& gp);
        GammaParameter(std::string owner,
                      double gamma,
                      double acquisition_time);

        ~GammaParameter();

        static void setExpirationInterval(double expiration_interval);
        static double getExpirationInterval();
        GammaParameter& setOwner(std::string owner);
        GammaParameter& setGamma(double gamma, double acquisition_time);
        GammaParameter& setData(std::string owner,
                               double gamma,
                               double acquisition_time);

        std::string getOwner();
        double getGamma();
        bool isExpired();
        GammaParameter& getData(std::string *owner_ptr,
                               double * gamma_ptr,
                               bool * exp_ptr);
        GammaParameter& operator= (const GammaParameter &gp);
};


```

Figure 3.4: Consensus variable class

```

class DronePose {
private:
    double x, y, z, yaw;
public:
    DronePose();
    DronePose(double x, double y, double z, double yaw);
    DronePose(const DronePose &dp);
    ~DronePose();

    double getX() const;
    double getY() const;
    double getZ() const;
    double getYaw() const;
    void setX(double x);
    void setY(double y);
    void setZ(double z);
    void setYaw(double yaw);

    double module() const;
    DronePose& operator= (const DronePose &dp); //Assignment
    DronePose operator+ (const DronePose &dp) const; //Sum
    DronePose& operator+= (const DronePose &dp);
    DronePose operator- (const DronePose &dp) const; //Difference
    DronePose& operator-= (const DronePose &dp);
    DronePose operator- () const; //Unary minus
    double operator* (const DronePose &dp) const; //Scalar product
    DronePose operator* (double scal) const; //Product with constant
    DronePose& operator*=(double scal);
    DronePose operator/ (double scal) const; //Ratio with constant
    DronePose& operator/=(double scal);
};


```

Figure 3.5: Class used to manage the position and velocity of the drones

The next step is to form the  $\bar{\alpha}_i$  term. Firstly, we need the next set point, which must be obtained evaluating the trajectory with the actual value of  $\gamma_i$ . Our trajectory is represented by a class, and its signature is shown in the Figure 3.6. We then simply compute the *position\_error* as *set\_point - position*. We also need the desired velocity, which can be obtained using the suitable function of the trajectory class. At this point, we have all the terms needed to compute  $\bar{\alpha}_i$  as defined in 1.10.

Since we have all the elements, we can apply the consensus law and find  $\ddot{\gamma}_i$ . We simply need to have the coefficients  $a$  and  $b$  and the references  $\ddot{\gamma}_d$  and  $\dot{\gamma}_d$ .

One of the last steps that we need to do is updating  $\dot{\gamma}_i$  using  $\ddot{\gamma}_i$  and  $\gamma_i$  using  $\dot{\gamma}_i$ . We compute the interval of time,  $dt$ , between the last update and the current update and we do the math as:

```
dgamma += ddgamma * dt ;
gamma += dgamma * dt ;
```

Finally, if we are operating in consensus mode, we need to publish the value of  $\gamma_i$  to the right topic, otherwise we simply ignore it. An operation which is always necessary is the publication of the setpoint message for the autopilot of the UAV, in order to allow the drone to follow the trajectory and reach its final destination.

```
class Trajectory {  
private:  
    std::vector<TrajectorySegment *> segments;  
    std::string drone_id;  
  
    double filterAndConvertTime(double t) const;  
    TrajectorySegment* getRightSegment (double t) const;  
  
public:  
    double min_time, max_time;  
  
    bool loadXML(std::string dorne_ns, char * document);  
    void cleanAll();  
  
    DronePose evaluateNED(double t) const;  
    DronePose evaluateENU(double t) const;  
    DronePose operator [] (double t) const; //Default return ENU  
  
    DronePose evaluateVelNED(double t) const;  
    DronePose evaluateVelENU(double t) const;  
};
```

Figure 3.6: Class used to manage a generic trajectory



# Chapter 4

## Simulation results

In this chapter we will show the simulations conducted to evaluate the consensus algorithm.

As said in the Chapter 2, the software used is Gazebo. We ran SITL (Software in the loop) simulations, through the utilities provided by the PX4 firmware. It provides models of the main topologies of aerial vehicles, such as plane, VTOL, Tailsitter VTOL and quadrotor. We will use the quadrotor model called Iris, which is shown in the picture 4.1.

The PX4 firmware is run on a simulated hardware and all the ROS nodes are executed on the same computer. The physics is simulated by Gazebo and all the components are interfaced through Gazebo plugins. In this way, the model can interact with all the external simulated components.

The Gazebo model is specified, using SDF language, in a Gazebo model file and the dynamic parameters are listed in it and the geometry is included as well.

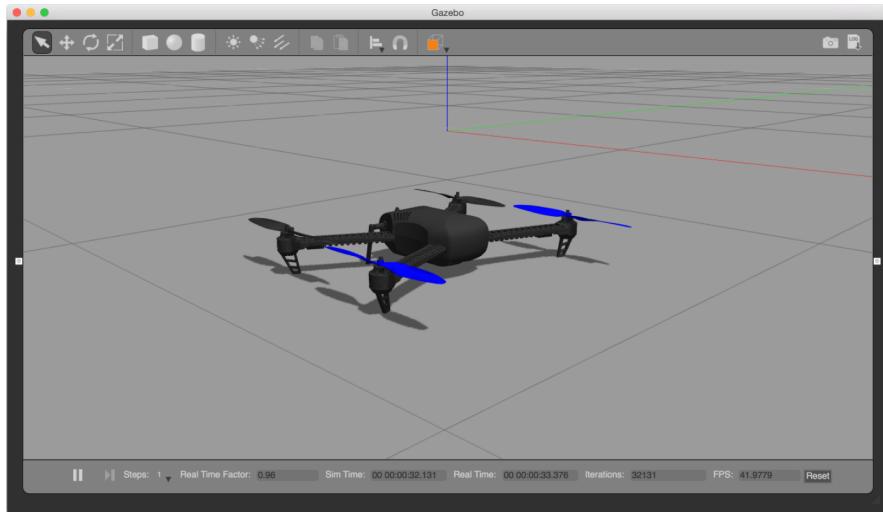


Figure 4.1: Iris model

The simulations that we will show are of three kinds. Firstly, we will present only the trajectory following problem of a formation of two drones. Secondly, we will send a disturbance to one of the drones and we will stop it in its position. Thirdly, we will introduce a disturbance which will make one of the drones go back following its trajectory backward. We see how the consensus algorithm reacts to these disturbances and forces the other drone to preserve the formation. Although only two drones have been used because of the computational load of the simulation, but the concept can be extended freely to an arbitrary number of drones.

## 4.1 Trajectory following

In the simulation two drones have been used. The trajectory of the mission is shown in 4.2.

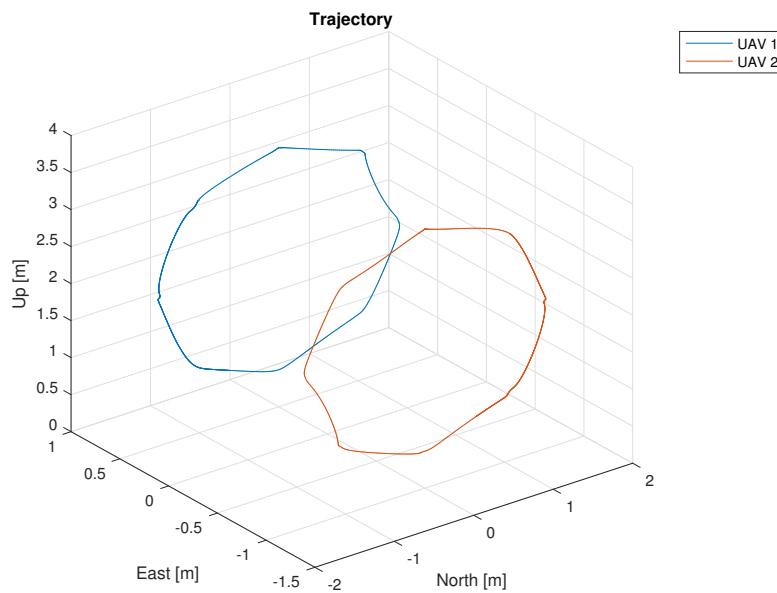


Figure 4.2: Trajectory

Both drones start in the upper point of their circle and then they move along the trajectory in opposite directions. The mission terminates when both drones reach their starting point. The evolution of the trajectory in time of the first drone can be seen in the Figure 4.3.

How the two drones follow their trajectory is shown in the Figures 4.4 and 4.5.

There are some delays due to the fact that the autopilot needs time to follow the target, but the drones can follow if successfully. In fact, they arrive at their final position at the same time.

If we also consider the Figure 4.6, we can see that both drones are synchronized during the execution of the mission. In the next cases, we will add disturbances in order to make the effects of the algorithm more evident.

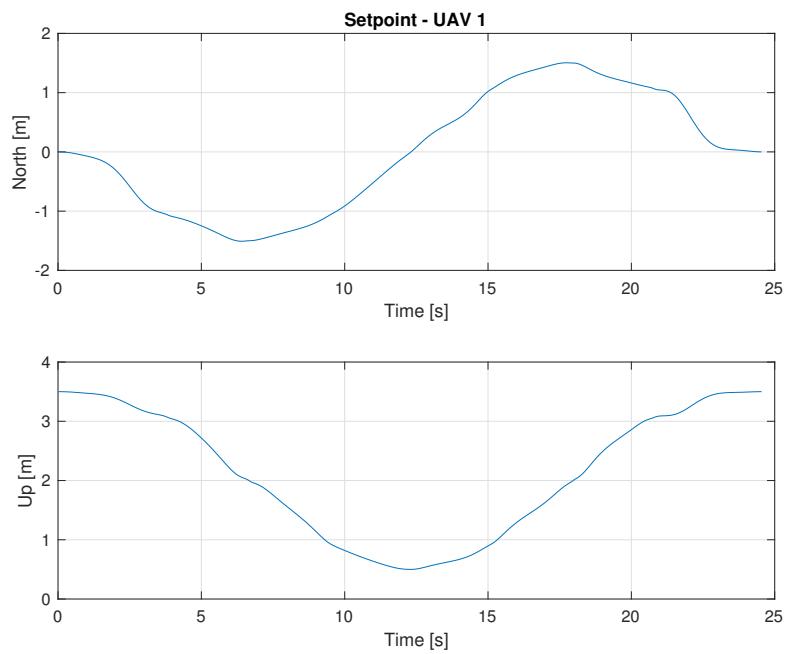


Figure 4.3: Evolution of the trajectory in time of the first drone

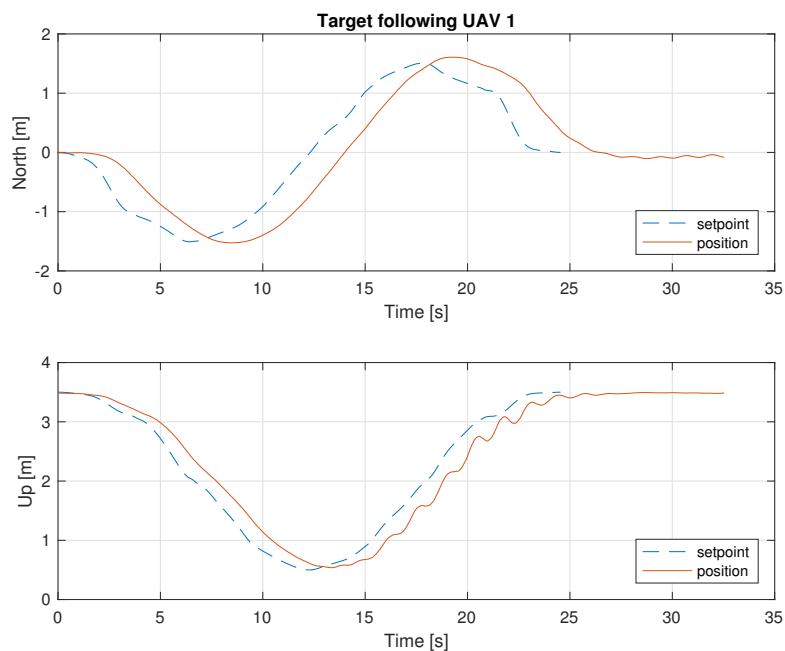


Figure 4.4: Target following drone 1

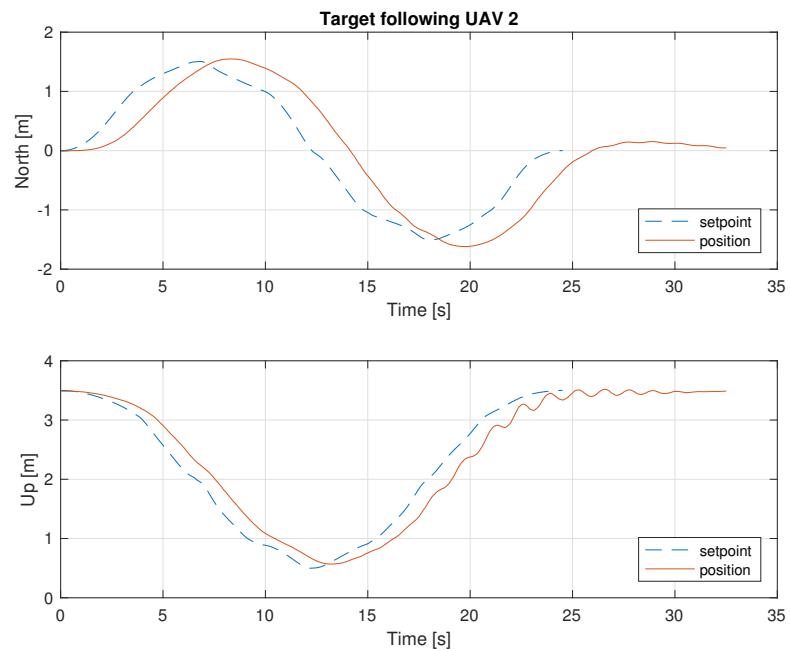


Figure 4.5: Target following drone 2

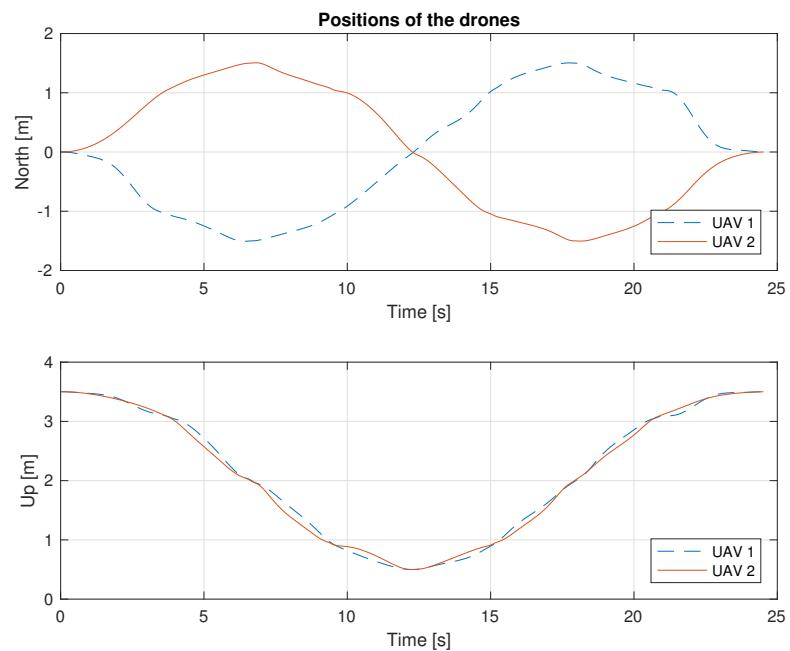


Figure 4.6: Positions of the two drones in time

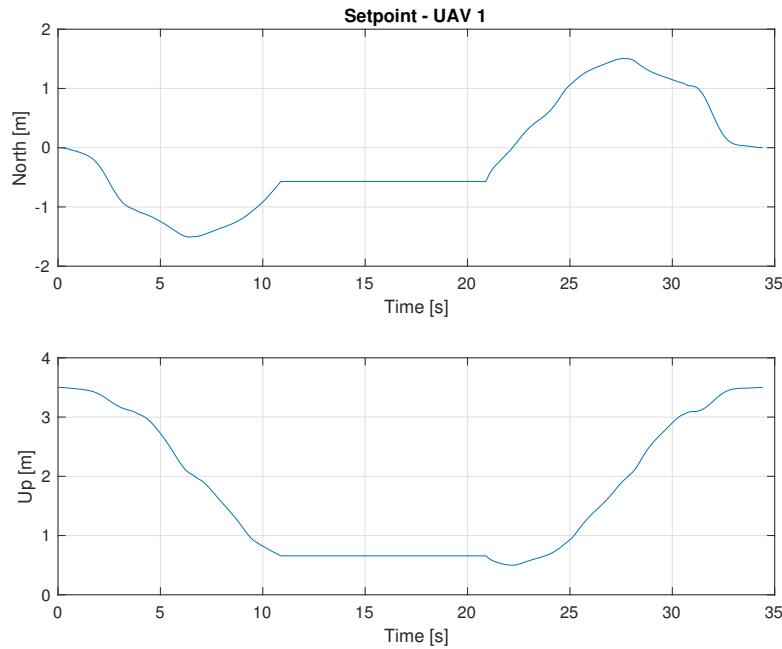


Figure 4.7: Disturbance

## 4.2 First disturbance

In this scenario, we use the same trajectory as before (Figure 4.2), but in this case we stop one of the two drones and the other will follow it. The Figure 4.7 shows the disturbance applied to the first drone at time 11s.

We can now see how the two drones execute the mission. The second drone tries to go on when the first is interrupted, but then the consensus stops it. The plots are shown in the Figures 4.8 and 4.9.

Finally, the Figure 4.10 is a graph representing the overlapped positions of the two drones.

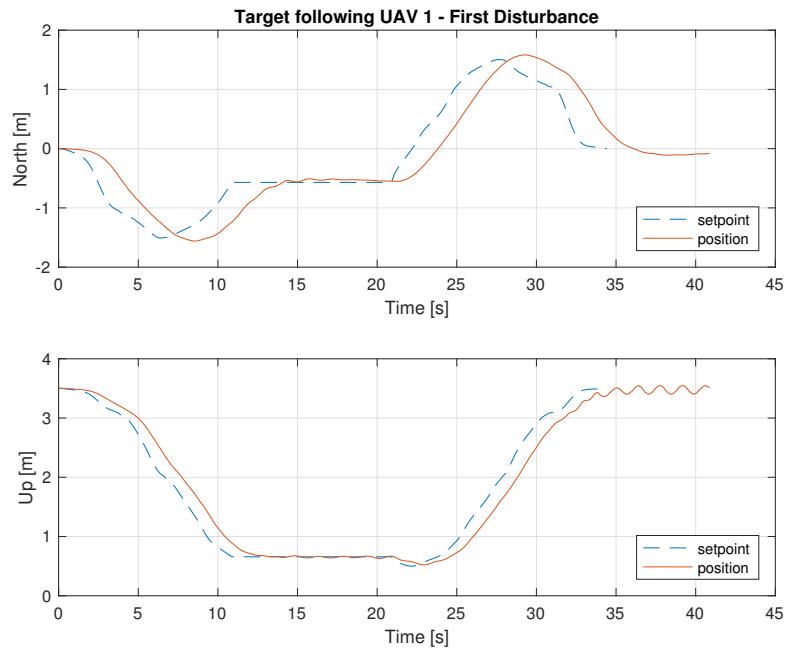


Figure 4.8: Target following drone 1

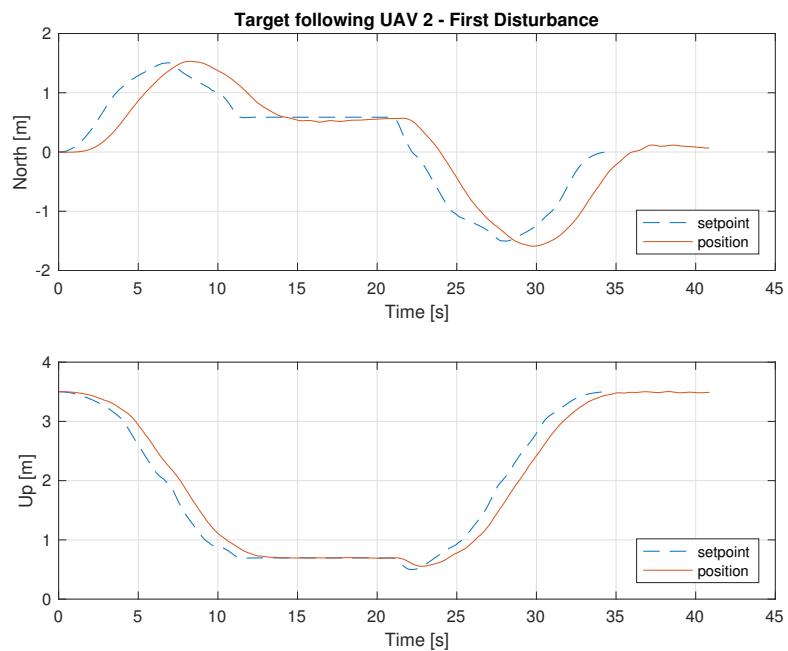


Figure 4.9: Target following drone 2

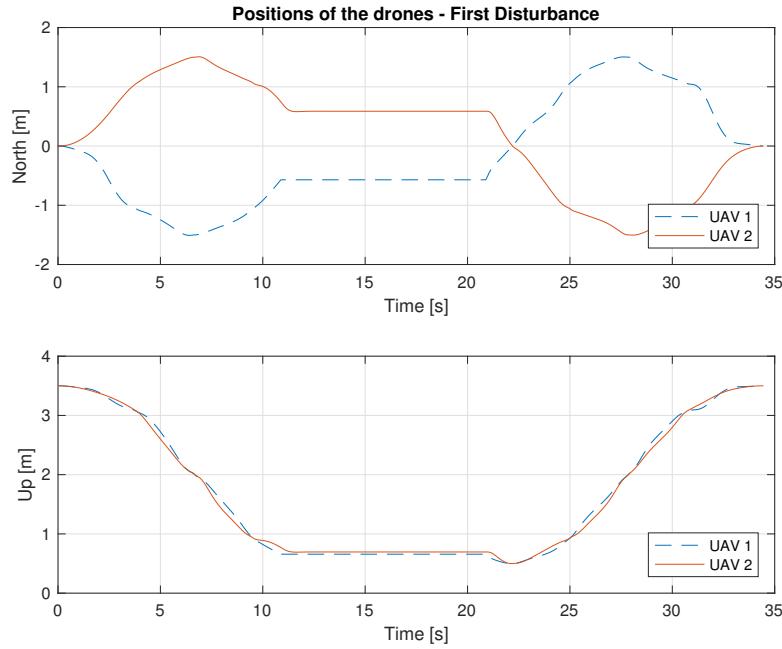


Figure 4.10: Positions of the two drones in time

### 4.3 Second disturbance

The trajectory used is always the same (Figure 4.2), but this time the disturbance is different. Now we force one drone to go back through the trajectory which has travelled so far. After the disturbance, the drone can resume the trajectory and complete the mission. We can see the effect of the disturbance on the trajectory in the Figure 4.11.

At time 11s the disturbance starts and the drone begins to go back. At time 26s, the drone has returned to the position where the disturbance is started.

In this situation, the other drone recognizes that the other machine is going backward for an unknown reason and starts to follow it. We can see how the mission is done by the two drones in the Figures 4.12 and 4.13.

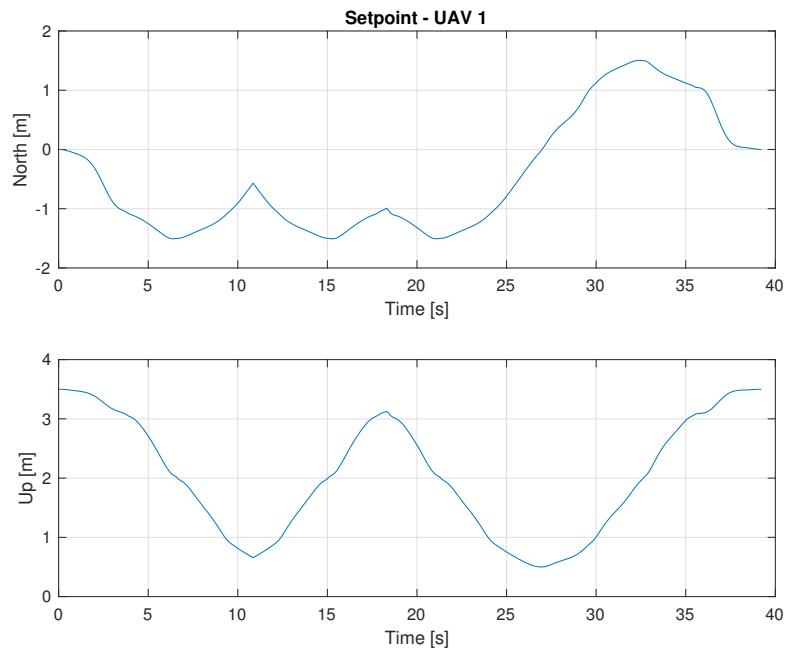


Figure 4.11: Disturbance

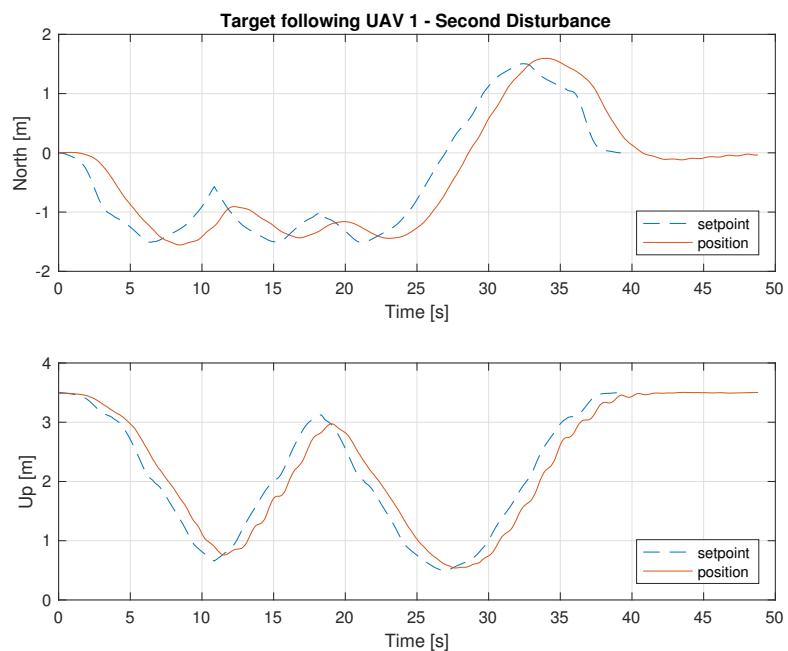


Figure 4.12: Target following drone 1

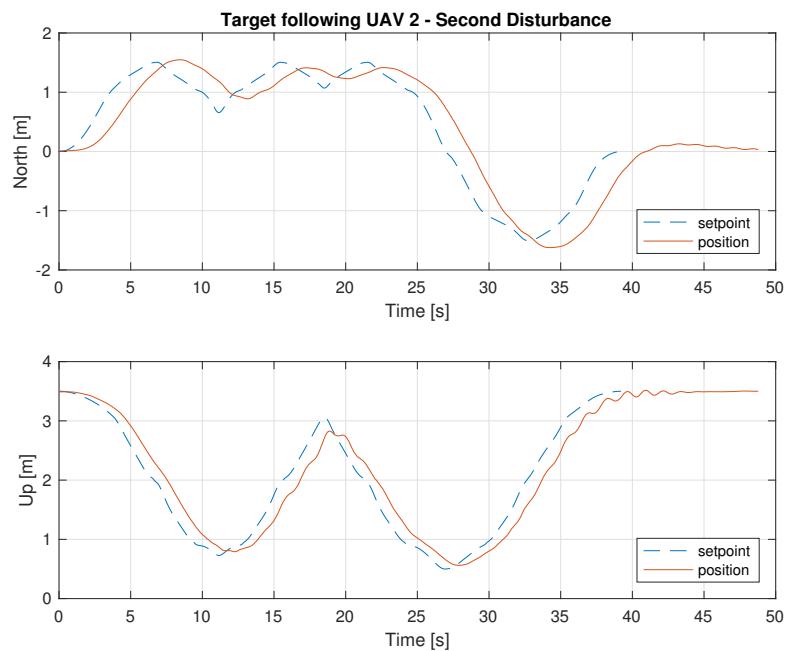


Figure 4.13: Target following drone 2

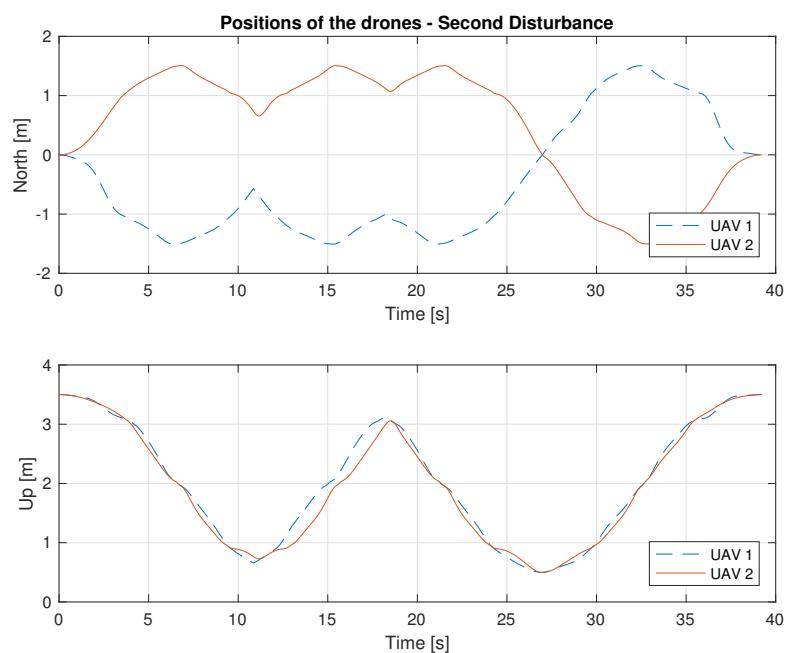


Figure 4.14: Positions of the two drones in time

Finally, the Figure 4.14, is a graph representing the overlapped positions of the two drones.



# Chapter 5

## Experimental results

In this chapter we present the results obtained applying the simulations presented in the previous chapter to a real system.

We want to highlight that the algorithm works even in the real system and the results obtained are compatible with the simulated ones. In this scenario, we need to take into account that the network is not ideal and the data may suffer delays and inaccuracy due to the complex clock synchronization of the machines involved in the experiment.

As in chapter 4, we will run the experiment three times. Firstly, the formation has to follow the trajectory without disturbances. Secondly, we run the algorithm and then we stop one of the drones, while the other one will try to go on, but then stop. Thirdly, we will introduce a disturbance which will make one of the drones go back following its trajectory backward. The consensus algorithm reacts to these disturbances and forces the other drone to preserve the formation.

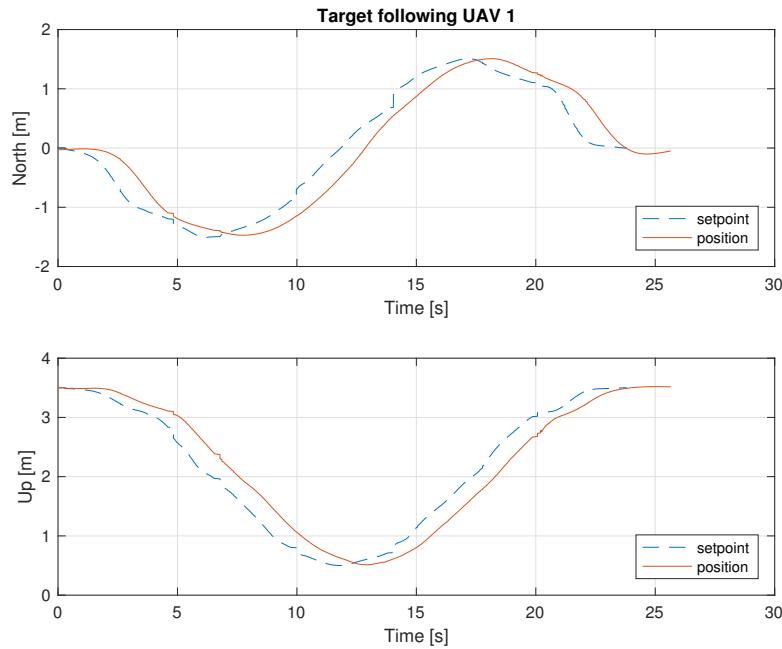


Figure 5.1: Target following drone 1

The trajectory used is the same as the simulated one (4.2), and the drones start on the top points of their trajectory, following the circle in opposite directions. They must be symmetric and finish the mission at the same time.

The three cases will now be presented in detail.

## 5.1 Trajectory following

The two drones will follow the trajectory, as shown in the Figures 5.1 and 5.2.

In this case as well, there are some delays due to the fact that the autopilot needs time to follow the target, but the drones can follow if successfully. In fact, they arrive at their final position at the same time. The two drones remain synchronized during the whole mission as we can see in the Figure 5.3.

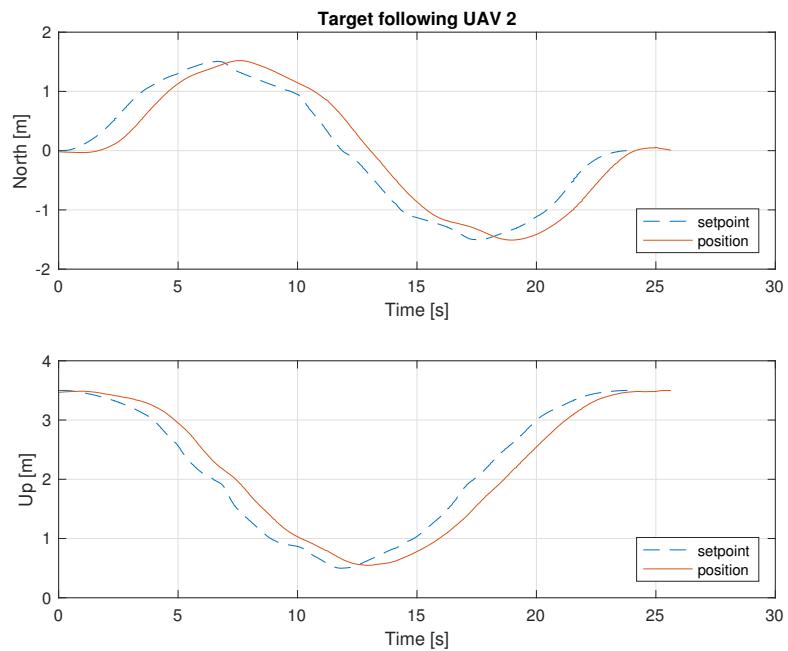


Figure 5.2: Target following drone 2

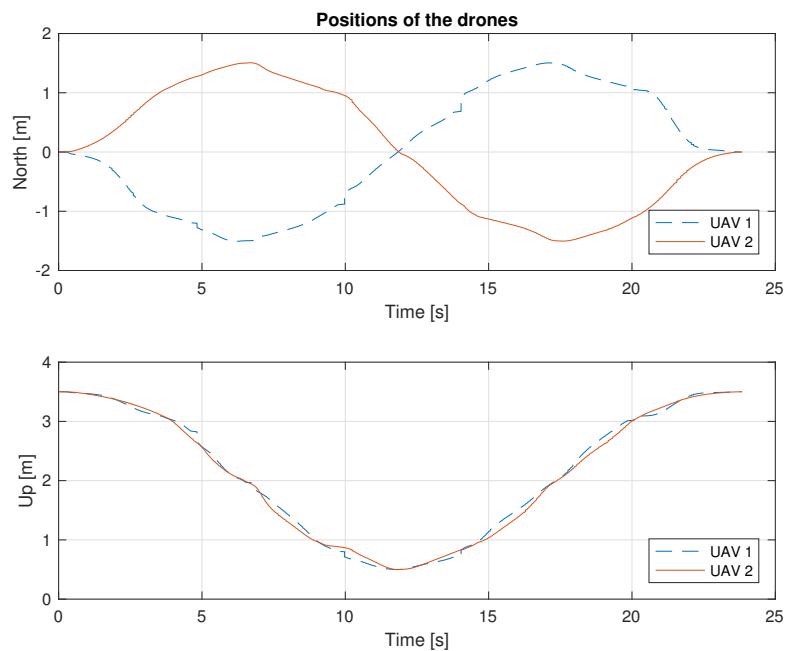


Figure 5.3: Positions of the two drones in time

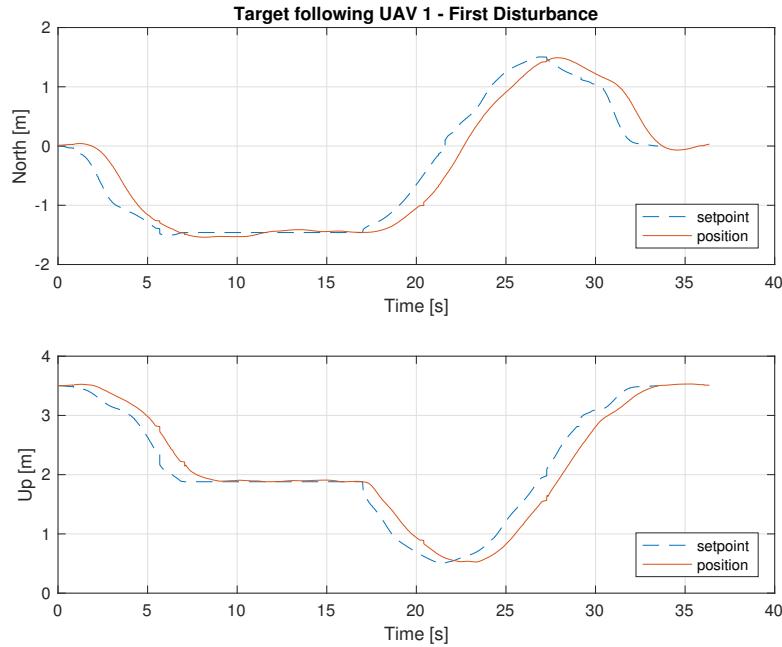


Figure 5.4: Target following drone 1

## 5.2 First disturbance

When we add a disturbance to one of the drones, the algorithm rejects it and preserves the formation. In this experiment we introduce the disturbance a time 6s and it terminates 10s later. The mission can be seen in the Figures 5.4 and 5.5.

The disturbance causes the other drone to stop and wait until the disturbance is over. We can see the synchronization in the Figure 5.6.

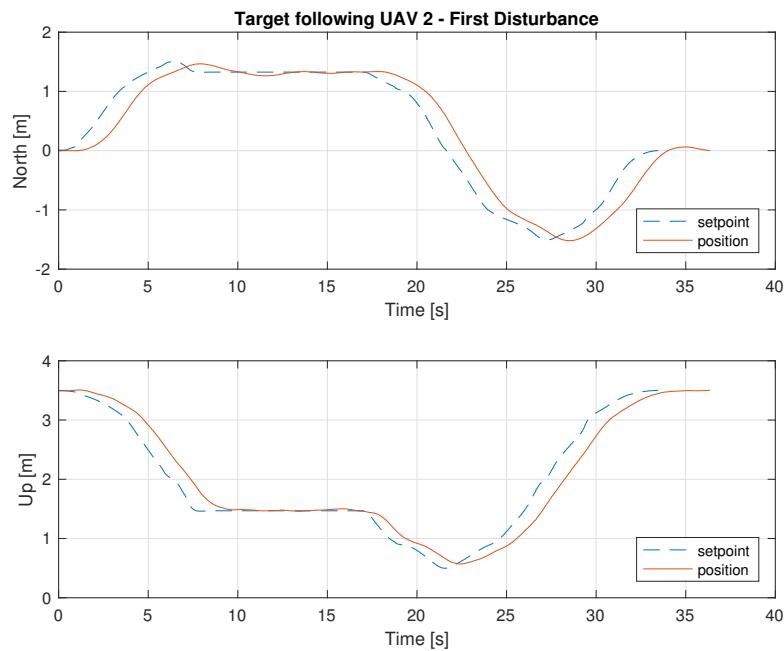


Figure 5.5: Target following drone 2

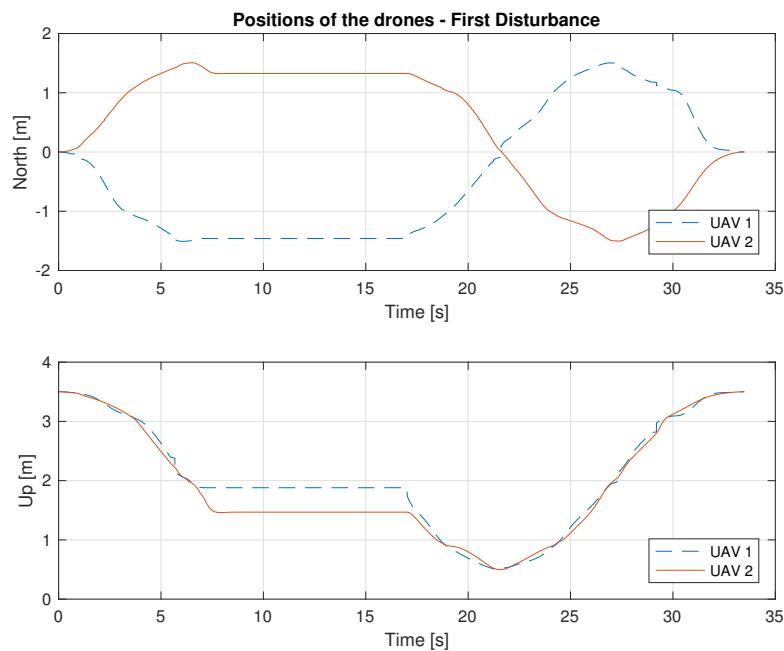


Figure 5.6: Positions of the two drones in time

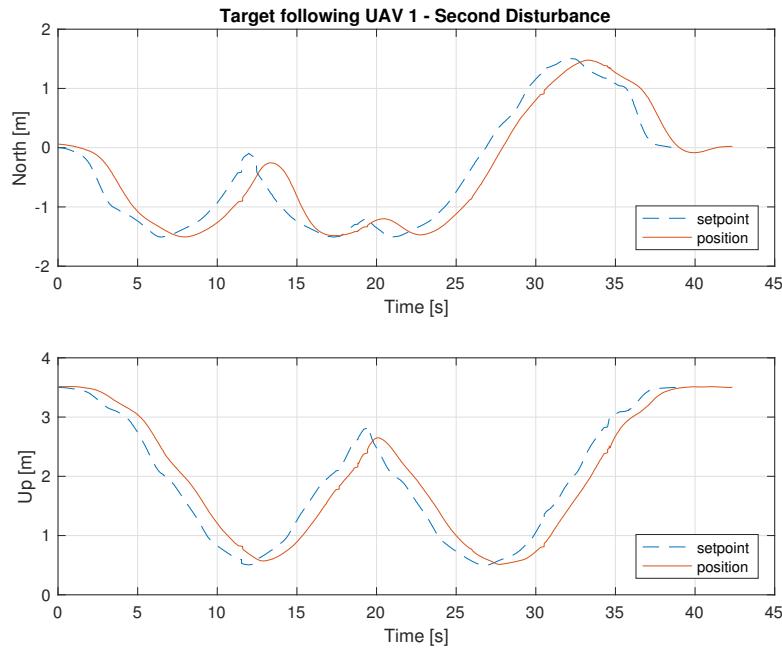


Figure 5.7: Target following drone 1

### 5.3 Second disturbance

The last disturbance is the one which forces a drone to go back along the trajectory. We can see how the setpoints change when the disturbance is active and how the drones follow them. The Figures 5.7 and 5.8 present the experiment.

The disturbance causes the other drone to stop and go back, following the other, until the disturbance is over. The synchronization is shown in the Figure 5.9.

The experimental results are less accurate than the simulated ones, but the overall behaviour is preserved. Indeed, if we compare the results, we can see that the formation is maintained in both cases. The plots are very similar even if the simulated model of the drones does not reflect the real drones and even if the

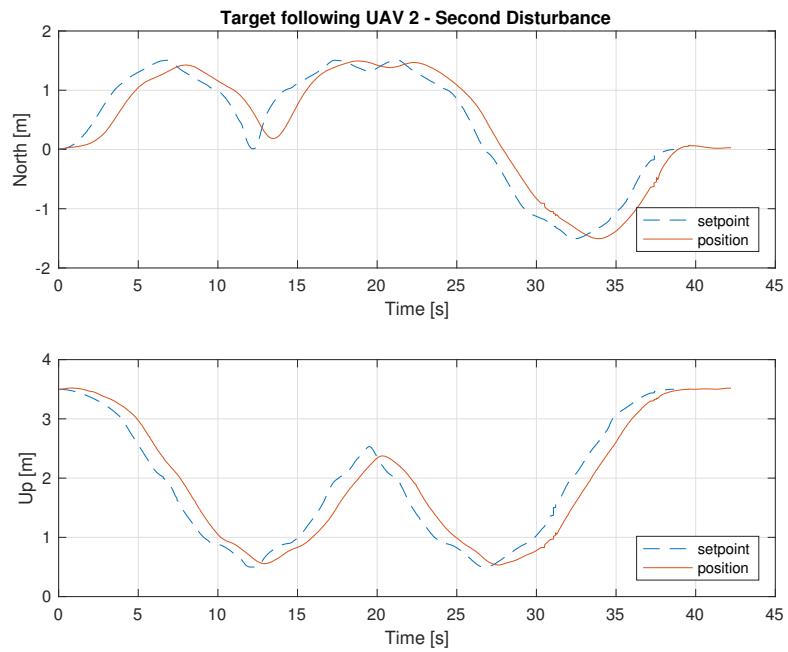


Figure 5.8: Target following drone 2

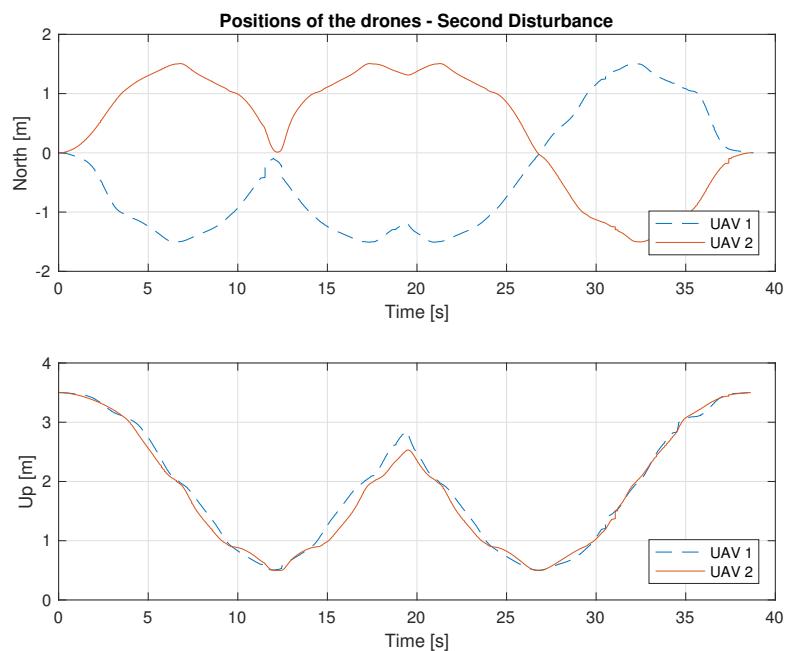


Figure 5.9: Positions of the two drones in time

drones used are heterogeneous.

# Conclusions

The results obtained in our experiments can be applied in many fields in which a formation of UAVs has to operate. In this thesis, we have shown how the consensus algorithm is deployed on a real system and which performances can be achieved.

The implementation allows us to plan and fulfill a cooperative mission in which the synchronization is one of the most important aspects. The system is robust to network delays and loss of packets and to unexpected failures of the vehicles involved. Moreover, the formation, if equipped with an obstacle avoidance algorithm, overcomes the presence of unexpected obstacles during the execution of the mission.

Further studies can be carried on in order to integrate recovery procedures in case of network failures or the loss of a machine. It is possible to develop obstacle avoidance algorithms or any kind of online procedures acting directly on the free parameters of the algorithm presented. In particular, it is possible to change the values of  $\ddot{\gamma}_d$  and  $\dot{\gamma}_d$ , in order to modify the velocity of the execution of the mission. These two values can be also used to build more complex functionalities on top of the consensus algorithm.



# Bibliography

- [1] Atom. Atom. <https://atom.io>.
- [2] N. Biggs. *Algebraic Graph Theory*. New York: Cambridge Univ. Press, 1993.
- [3] Canonical. Ubuntu 16.04 LTS. <https://www.ubuntu.com>.
- [4] R. Choe, J. Puig-Navarro, V. Cichella, E. Xargay, and N. Hovakimyan. Trajectory generation using spatial Pythagorean Hodograph Bezier curves. *in Proc. AIAA Guidance, Navigation and Control Conf., Kissimmee, FL*, 2015.
- [5] V. Cichella, R. Choe, S. B. Mehdi, E. Xargay, N. Hovakimyan, V. Dobrokhodov, I. Kaminer, A. M. Pascoal, and A. P. Aguiar. Safe coordinated manuvering of teams of multirotor unmanned aerial vehicles. *IEEE Control Systems Magazine*, August 2016.
- [6] Emutex. Jubilinux. <http://www.jubilinux.org>.
- [7] Open Source Robotic Foundation. ROS. <http://www.ros.org>.
- [8] Open Source Robotics Foundation. Gazebo. <http://gazebosim.org>.
- [9] Raspberry Pi Foundation. Raspbian Jessie 4.4. <https://www.raspberrypi.org/downloads/raspbian/>.

- [10] Raspberry Pi Foundation. Raspberry Pi Zero. <https://www.raspberrypi.org/products/pi-zero/>, 2015.
- [11] GitHub. GitHub. <https://github.com>.
- [12] Gitlab.com. GitLab. <https://about.gitlab.com>.
- [13] Kathrin Grävel and Alex Bencz. Mocap Optitrack. [https://github.com/ros-drivers/mocap\\_optitrack](https://github.com/ros-drivers/mocap_optitrack).
- [14] Intel. Intel Edison Compute Module. <https://software.intel.com/en-us/iot/hardware/edison>, 2014.
- [15] MathWorks. Matlab. <https://www.mathworks.com/products/matlab.html>.
- [16] L. Meier. Mavros. <https://github.com/mavlink/mavros>.
- [17] L. Meier. Px4 Firmware. <https://github.com/PX4/Firmware>.
- [18] L. Meier. Pixhawk autopilot. <http://www.pixhawk.org>, 2008.
- [19] Microsoft. Windows 10 Pro. [https://www.microsoftstore.com/store/mseea/it\\_IT/pdp/Windows-10-Pro/productID.320433200](https://www.microsoftstore.com/store/mseea/it_IT/pdp/Windows-10-Pro/productID.320433200).
- [20] Motive. Optitrack. <http://optitrack.com/products/motive/>.
- [21] Motive. Prime 13. <http://optitrack.com/products/prime-13/>.
- [22] Netgear. Prosafe 28PT GE POE. <https://www.netgear.com>.
- [23] Latex project. Latex. <https://www.latex-project.org>.

- [24] V. T. Taranenko. Experience of Employment of Ritz's, Poincare's, and Lyapunov's Methods for Solving Flight Dynamics Problems. *Moscow: Air Force Engineering Academy Press*, 1968.
- [25] VMware. VMware. <https://www.vmware.com>.
- [26] O. A. Yakimenko. Direct method for rapid prototyping of near-optimal aircraft trajectories. *Journal of Guidance, Control, and Dynamics*, 23(5):865–875, Sept.-Oct. 2000.