Master Thesis
Máster Universitario en Ingeniería Industrial

Development of a ROS environment for researching machine learning techniques applied to drones

Author: José Andrés Millán Romera

Tutors: Jesús Iván Maza Alcañiz, Aníbal Ollero Baturone

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Master Thesis
Máster Universitario en Ingeniería Industrial

Development of a ROS environment for researching machine learning techniques applied to drones

Author:
José Andrés Millán Romera

Tutors:
Jesús Iván Maza Alcañiz, Aníbal Ollero Baturone
Profesor titular y catedrático

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Master Thesis: Development of a ROS environment for researching machine learning techniques applied to drones

Author: José Andrés Millán Romera
Tutors: Jesús Iván Maza Alcañiz, Aníbal Ollero Baturone

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

0

Agradecimientos

Es incontable la cantidad de gente que me ha aupado hasta este momento, casi literalmente. No me tiembla el pulso al ecribir que, pese a lo arduo del camino, he sido absolutamente feliz cada uno de los días que lo he andado. Y la culpa es vuestra.

Muchas son las razones por las que me alegro del camino elegido. La más importante: ahora sé quién quiero ser y de qué clase de personas me quiero rodear. Y en eso, veo que soy yo el que ha acertado.

Muchas gracias a mis maestros Iván Maza y Ángel Rodríguez, y al GRVC al completo, por saber guiarme a modelar el proyecto, a saber discurrir las necesidades y a tomar las perspectiva correcta para ver la solución óptima. Y por proyecto me refiero a mi crecimiento como ingeniero.

A mi familia. Gracias por la alegría que siento cada vez que reconozco en mí mismo algún rasgo vuestro. Gracias por confiar en mí cuando ni yo lo hacía, por enseñarme a relativizar los malos momentos y por estar a mi lado en la distancia. Gracias por las dos pequeñas grandes revoluciones de nuestra familia, que me hacen esforzarme por ser un buen ejemplo que rejuvenece a su lado.

A mis amigos, a los ubetenses que hace tanto que no recuerdo cuándo os conocí, a los que me acogisteis en esta maravillosa ciudad y a los que he descubierto en estos años. Gracias por enseñarme a compartir mis virtudes y a reirme de mis defectos, a entender que la vida son más cosas, a descubrir qué me gusta y por qué quiero luchar.

Gracias a la mejor recompensa a una de mis más difíciles decisiones. Gracias por enseñarme que, cuando se está seguro de que algo tiene que salir bien, saldrá bien si nunca se bajan los brazos. Gracias porque contigo aprendo a construir el futuro que quiero vivir y disfruto cada día diseñándolo.

Y por último, muchas gracias a las personas que piensan que el conocimiento hay que compartirlo. Ya sea en una conversación o en internet, con una idea o con un consejo. Porque son ellas las que hacen que la humanidad avance, que la ciencia tenga un impacto positivo y que gente como yo podamos algún día sumarnos a mejorarlo.

José Andrés Millán Romera
Sevilla, 2019

Abstract

The first part of this dissertation presents ROS-MAGNA, a general framework for the definition and management of cooperative missions for multiple Unmanned Aircraft Systems (UAS) based on the Robot Operating System (ROS) [42]. This framework makes transparent the type of autopilot on-board and creates the state machines that control the behaviour of the different UAS from the specification of the multi-UAS mission. In addition, it integrates a virtual world generation tool to manage the information of the environment and visualize the geometrical objects of interest to properly follow the progress of the mission. The framework supports the coexistence of software-in-the-loop, hardware-in-the-loop and real UAS cooperating in the same arena, being a very useful testing tool for the developer of UAS advanced functionalities. To the best of our knowledge, it is the first framework which endows all these capabilities. The document also includes simulations and real experiments which show the main features of the framework.

ROS-MAGNA is used to develop and test a machine learning tool. The information generated during a mission is used to train neural networks of different architecture for navigation purposes. The data treatment and training processes are accomplished in a testbench to select the best solution from different datasets. Tensorflow is the framework selected to implement every deep learning algorithm along with its Tensorboard tool for training understanding.Furthermore, an API with the pre-trained is used during a real mission in real time.

The third part of this dissertation is the design and integration of a voice control assistant inside ROS-MAGNA. Employing diverse online and offline tools, oral commands are processed to perform changes to the mission state and performance and to retrieve information.

Índice Abreviado

Abstract III
Índice Abreviado V

List of Figures XI
List of Tables XIII
List of Codes XV

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 1
1.3 Organization of the contents 2

2 State of the Art **5**
2.1 ROS-MAGNA 5
2.2 Machine Learning Research 6
2.3 Voice Control Assistant 8

3 Main Software Tools **9**
3.1 Python 9
3.2 Robot Operating System, ROS 9
3.3 UAV Abstraction Layer, UAL 11
3.4 SMACH 12
3.5 GAZEBO 14
3.6 Tensorflow 14
3.7 WIT 15

4 ROS MAGNA **17**
4.1 Master 18
4.2 Ground Station 20
4.3 Ground Station State Machine 23
4.4 Worlds 29
4.5 UAV Manager 35
4.6 UAV - Navigation Algorithms Interface 43
4.7 UAV Data and Configuration 46
4.8 Experiments 48

5 Machine Learning Research **55**
5.1 Introduction 55
5.2 Steps previous to training 56
5.3 Neural Network boilerplate 59

6 Voice Control Assistant **69**
6.1 Introduction 69
6.2 Designed WIT app 70
6.3 Frontend 70
6.4 Backend 71

7 Conclusions and future work **73**
7.1 Conclusions 73
7.2 Future work 73
Bibliography 75

Contents

Abstract III
Índice Abreviado V

List of Figures XI
List of Tables XIII
List of Codes XV

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 1
1.3 Organization of the contents 2
1.3.1 ROS MAGNA 2
World 2
Ground Station 2
UAV Manager 2
UAV Navigation Algorithm Interface 3
Master 3
Contributions 3
1.3.2 Machine Learning Research 3
Data treatment 4
Neural network boilerplate 4
1.3.3 Voice Control Assistant 4

2 State of the Art **5**
2.1 ROS-MAGNA 5
2.2 Machine Learning Research 6
2.2.1 Introduction 6
2.2.2 Subfields 6
2.2.3 Most common structures 6
2.2.4 Applications 7
2.3 Voice Control Assistant 8

3 Main Software Tools **9**
3.1 Python 9
3.2 Robot Operating System, ROS 9
3.2.1 Challenges 9
3.2.2 System 10
Architecture 10
Communications 10
Specific tools 11
3.3 UAV Abstraction Layer, UAL 11

3.4 SMACH 12
3.4.1 Introduction 12
3.4.2 Philosophy 13
3.5 GAZEBO 14
3.6 Tensorflow 14
3.6.1 Intro 14
3.6.2 Work methodology 15
3.6.3 Tensorboard 15
3.7 WIT 15

4 ROS MAGNA 17
4.1 Master 18
4.1.1 Motivation 18
4.1.2 Hyperparameters 18
4.1.3 Master functions 18
Definitions 18
Creation and control of storage folders 19
Initialisations 19
Batch of simulations 20
Processes termination and data storage 20
4.2 Ground Station 20
4.2.1 Motivation 20
4.2.2 GS functions 21
Initialization 21
Starter functions 21
Finisher functions 22
4.3 Ground Station State Machine 23
4.3.1 Motivation 23
4.3.2 SM building system 23
Introduction 23
State Machine Structure Elements 24
4.3.3 Defined States 25
Introduction 25
New World 25
Spawn UAV 25
Wait 25
Take off 25
Land 25
Save CSV 26
Basic move 26
Follow path 26
Follow UAV at distance or position 26
4.3.4 Example of state machine structures for missions 26
Introduction 26
All take off 26
Follow paths sbys 27
Queue of followers AD / AP 28
Battery recharge 29
Safe stop and descent 29
4.4 Worlds 29
4.4.1 Motivation 29
4.4.2 Hierarchy of classes 31
Worlds 31
Volumes 32

Generic Geometry 33
Specific Geometries 34
Free Space Pose 34
Obstacle 35
4.4.3 Auxiliar classes 35
Transformation Broadcaster 35
Rviz Marker 35
Rviz PolygonArray 35
4.4.4 Built Worlds 35
4.5 UAV Manager 35
4.5.1 Motivation 35
4.5.2 UAV Manager functions 37
4.5.3 Initializations 38
Listeners and publishers 38
4.5.4 Commanders 38
Land command 39
Take-off command 39
Go to Waypoint command 39
Set velocity command 39
4.5.5 Roles 39
Path Follower 39
UAV Follower At Distance 40
UAV Follower At Position 40
Basic Move 40
4.5.6 Miscellaneous 41
Evaluator 41
Parse for CSV 42
Save data 42
Store data 42
GS state actulization 42
Geometricals 42
4.5.7 UAV-M State Machine 42
Motivation 42
SM building system 43
Available states 43
4.6 UAV - Navigation Algorithms Interface 43
4.6.1 Introduction 43
4.6.2 Initialisation 44
4.6.3 Guidance 44
4.6.4 Simple Guidance 44
4.6.5 ORCA 3D 45
4.6.6 Neural Networks 45
4.6.7 Miscellaneous 46
Hover 46
Upper and Lower Saturation 46
Neighbour Selection 46
4.7 UAV Data and Configuration 46
4.7.1 Introduction to UAV Data 46
4.7.2 Initialisation 46
4.7.3 Listeners 47
UAV position 47
UAV velocity 47
UAV UAL state 47
Depth camera 47

Preemption command 47
4.7.4 Motivation of UAV Configuration 48
4.7.5 Initialisation 48
4.7.6 Interface definitions 48
4.8 Experiments 48
4.8.1 Introduction 48
4.8.2 Experiment I: State machine safety branches 49
4.8.3 Experiment II: SIL, HIL and real joint mission 51
4.8.4 Experiment III: Delay on computation due to number of simulated UAVs 53

5 Machine Learning Research 55
5.1 Introduction 55
5.2 Steps previous to training 56
5.2.1 Manager 56
Motivation 56
Hyperparameters 56
Training performance 56
5.2.2 Data pickling 57
Data retrieving 57
Data parsing 58
5.2.3 Data preprocessing 58
Data mathematical preparation 58
Dataset partition 59
5.3 Neural Network boilerplate 59
5.3.1 Graph building 59
Initialisation 59
Fully-conected stage 59
Convolutional stage 60
Stages assembly 60
Output retrievement 62
5.3.2 Graph trainer 62
Auxiliary graph branches 62
Training 63
5.3.3 Example of use 63
Auxiliary graph branches 63

6 Voice Control Assistant 69
6.1 Introduction 69
6.2 Designed WIT app 70
6.3 Frontend 70
6.3.1 Introduction 70
6.3.2 Speech recognition process 71
6.4 Backend 71
6.4.1 Introduction 71
6.4.2 Entities to command 71
Entities translation 71
Ground Station service message construction 72
Service call 72

7 Conclusions and future work 73
7.1 Conclusions 73
7.2 Future work 73
Bibliography 75

List of Figures

3.1	ROS challenges.	10
3.2	ROS structure.	10
3.3	Sample of an RQT graph.	11
3.4	Example of an Rviz visualization.	12
3.5	Example of a State Machine implemented with SMACH.	13
3.6	Example of GAZEBO simulation.	14
3.7	WIT example.	15
4.1	ROS MAGNA general structure	17
4.2	Master node relations diagram.	18
4.3	GS node relations diagram	21
4.4	Visualization of an "all take-off" Concurrence	27
4.5	Visualization of Follow paths sbys mission step	28
4.6	Visualization of Follow UAV At Position mission step im	29
4.7	Visualization of Battery Recharge mission step	30
4.8	Visualization of Safe Stop mission step	30
4.9	Worlds modelling diagram	31
4.10	Layout of an example world	32
4.11	World 1: Intersection	36
4.12	World 2: Tube	36
4.13	World 3: Solar plant	36
4.14	World 4: GAUSS Project Use Case 1	37
4.15	UAV-M node relations diagram	37
4.16	Experiment I: Battery low, GAZEBO	49
4.17	Experiment I: Battery low, RViz	50
4.18	Experiment I: Imminent collision detected, GAZEBO	50
4.19	Experiment I: Imminent collision detected, RViz	51
4.20	Experiment II: SIL, HIL and real UAVs	51
4.21	Experiment II: Mission development visualized with RViz	52
4.22	Experiment II: Real Canamero solar plant	52
4.23	Experiment3	53
5.1	General structure for the Machine Learning Research	55
5.2	Tensorboard graph for one layer and complete fully-connected stage	60
5.3	Tensorboard graph for one layer and complete convolutional stage	61
5.4	Tensorboard graph for a complete model from input to output	61
5.5	Tensorboard graph for one single computation	62
5.6	Tensorboard graph for complete training and final use process	63
5.7	Gym world empty	64
5.8	Loss function set 1 for empty world	65
5.9	Loss function set 2 for empty world	66

5.10 Gym world with an obstacle 67
5.11 Loss function set 1 for obstacled world 67
5.12 Loss function set 2 for obstacled world 68
5.13 Histograms of weights and biases of the optimal found architecture 68

6.1 General pipeline for the voice control command 69
6.2 Teaching sentences examples 70

List of Tables

4.1	Hyperparameters.	18
4.2	Parameters defined in Master	19
4.3	Possibilities offered when storage conflict	19
4.4	Initialisations of Master	20
4.5	Simulation data stored	20
4.6	Master-GS channels	22
4.7	UAV spawn parameters	22
4.8	Mission threads closing steps	22
4.9	Steps to create a state machine	23
4.10	State Machine Structure Elements	24
4.11	Defined steps	25
4.12	Defined complex structures	26
4.13	Volume class definition	32
4.14	Generic geometry class definition	33
4.15	Specific geometries dimensions	33
4.16	Functions of Geometries	34
4.17	Specific geometries shape	34
4.18	UAV-M commander functions	38
4.19	Roles and its inputs	39
4.20	Basic Move definition	41
4.21	Basic Move directions mapping	41
4.22	Grond Station State Machine states	43
4.23	Solver algorithms	44
4.24	ORCA 3D parameters	45
4.25	Inputs to the NN by roles	45
4.26	Topics subscribed by UAV-D module	47
4.27	Parameters defined on the UAV-C module	48
5.1	Machine Learning Research Hyperparameters	57
5.2	Neural network architecture parameters	57
5.3	Crumble of dataset variables to FNN inputs	58
6.1	WIT designed entities	71

List of Codes

4.1	Storage folders logical cascaded structure	19
4.2	All Take-off, Land and Save mission steps hierarchy	27
4.3	Follow Paths SbyS step hierarchy	27
4.4	Queue of Followers AD/AP step hierarchy	28
5.1	Data retrieving process	58

1 Introduction

The trial and error of humankind have created dramatic changes since we first learned how to make fire. Some of them exploded into our consciousness causing considerable fear. Others, however, were pursued for decades until a glimmer of light was seen for the first time. Nevertheless, all of them share one feature in common: could any of them been used to solve the timeless dilemmas of our history or could they have lead us to a point of no return and catastrophic end. It seems that in fact both could be true and both have got us to this point in history where we find ourselves now.

Robotics and Artificial Intelligence are, without doubt, the next great tectonic shift for our race. It will change our society from the root upwards. Social structures, relations, work, leisure, safety and security, communications, transport, all will be completely different in a hundred years from now. Hence, as has been before at every inflexion point, we have the opportunity to propel ourselves to a more equitable, peaceful and advanced world. However, it is also our duty to guide this new force employing moral integrity and virtue and make sure its control is in the right hands. Or, even better, in everybody's hands.

1.1 Motivation

This project is not about all these philosophical questions. But they were what motivated me to accomplish it. The awareness of this reality and the opportunity to access all the knowledge here deployed, has created that inflexion point for me and my career as an engineer. Here was where my new passion for Robotics and Artificial Intelligence began, and as with every new passion, it enabled me to fulfil a lifelong ambition: to fly. That is the link to the second main idea of this project. The ability to fly has also permitted me to experiment with the latest generation of Unmanned Aerial Vehicles.

In recent years, the use of UAVs or, more commonly known as drones, has experienced unprecedented growth globally. That is, among other reasons, thanks to the high improvement of computational power to control such an inherently unstable aircraft. These kinds of devices open up opportunities, not only to enhance leisure but for a vast amount of useful applications such as risky aerial manipulation, transportation or surveillance.

Besides it has exciting potential, the increase in drone use has created a very challenging problem. We will have to deal with an increasingly crowded airspace which will require precise traffic management, as well as technological improvements and safety guarantees. This is why Artificial Intelligence and unmanned vehicles are bound together.

1.2 Objectives

The knowledge learned thanks to this project has reached much further than the initial objectives in many engineering areas. However, it has been due to clear initial objectives.

First of all, the willingness to learn Artificial Ingetillence basics theory and application with the most powerful framework. The most important algorithms such as neural networks may be used onboard UAVs in order to provide them with a safer and more intelligent decision capability for several problems such as collision avoiding.

That reason derived the second main purpose of this project: a framework where to train and test the algorithms on simulated and real UAVs. With time, this objective broadened given the increasing potential.

Therefore, the aim has been to improve its capabilities to fulfil the requirements of the projects it has been used for. For instance, these improvements have been increasing the number of behaviours, of autopilots, structures for complex cooperative missions and many other features.

Eventually, it raised the possibility of controlling the mission with voice commands to facilitate user interaction. Provided the characteristics of ROS-MAGNA, it presented a straightforward implementation so it became the third main objective.

1.3 Organization of the contents

1.3.1 ROS MAGNA

ROS-MAGNA (ROS-based Multi-AGent mission maNAgement), presented on chapter 4, is a general framework to manage cooperative missions for multiple UAS based on ROS. Multi-UAS mission specifications are implemented as state machines that control the behaviour of the different UAS independently of its autopilot on-board. In addition, a virtual world generation tool has been designed to offer an overall visual understanding of the state of the main geometrical elements and its progress on the mission. The framework is focused on offering useful testing tools on the whole development pipeline of UAS advanced functionalities, from simulated software-in-the-loop and hardware-in-the-loop to real UAS even at the same time and arena.

This project presents a general but deterministic approach to manage cooperative missions regarding every aspect that affect the performance. From the definition of the surrounding world, the configuration of the UAVs, the characterisation of the interface with onboard algorithms and the determination of the state machine that specifies the mission, the researcher would easily integrate, simulate and bring to real tests its research. The architecture is entirely modular and scalable to any size and complexity of the world, number of UAVs and depth of SM of the mission.

In order to provide an extensive but clear overview, a typical case of use is described now. An insight throughout all the modules, implemented in ROS nodes, is simulated. Supposed we are part of a research lab focused on urban delivery. Our task consists on, avoiding collisions with terrain obstacles and other known airbornes, get from one point to another in a short area of the city. In the next part of this introduction, the different modules that implement this platform are described throughout.

World

This task corresponds to section 4.4. First of all, a map of the land restrictions should be split into single volumes within its respective geometries and obstacles, modelled and introduced inside the Worlds database. After that, the different desired paths should also be split into single waypoints. Certain uncertainty as different land and take-off points or the position of obstacles, would be accomplished. This trajectory information should also be referenced within volumes description.

That modelisation of the world is simplified by a JSON interface so from easy to complex worlds are easily defined. Some examples of worlds are presented as an explanation, but a bigger range of creativity is affordable.

Ground Station

Once a physical environment has been described, it is time to define a mission. This task is accomplished by translating the information stored in another JSON file into states concerning tasks or actions for each UAV or the Ground Station. The tasks transmitted to a drone would be either for own movements or for own parametrization such as change on algorithms used or role accomplished. The corresponding part of the document is section 4.2.

After that, those steps are assembled employing transitions that link them. Progressively, an architecture of logical cooperative actions that pursuit a global aim is built. Those missions are defined as a graph or tree with several branches to deal with the variety of possibilities the mission would face. Concurrencies, sequences and simple state machines may be used, even nested, to shape the full cooperative mission.

UAV Manager

The single actions that a UAV can accomplish need to be defined. The way it will talk to the autopilot, or the information it has stored from other airbornes define when and how the mission is afforded. This task is deeply explained on section 4.5

Besides the typical movements that every drone should be able to do, other complex behaviours can be provided in order to gain versatility for more critical missions. Those are called roles. Furthermore, the

sensors it is provided and the use it does to are is defined in this module as well. At the end of every mission, this module is in charge of translate and store all the data generated into a single file for future treatment.

A variety of single actions are already implemented such as take off to a height, land and hover at a waypoint. A range of more complex behaviours is offered to the user as well. Those roles consist of following a list of waypoints in the air, follow another UAV at a certain distance or follow him at a specific relative position. The first kind of role to follow a path seems to be the most useful for us.

Initially, the mission is designed and tested on Software-in-the-loop (SIL), simulating the real world physics as well as the autopilots. Lately, the computation of the autopilot may be executed on the real hardware, so the next level of the process is testing on Hardware-in-the-loop (HIL). Finally, the mission is completely exported to the real field, executing on computers only the ROS MAGNA original software.

UAV Navigation Algorithm Interface

Now we come to the point of selecting or introducing the algorithm that is going to run inside each UAV in order to make it achieve its targets safely and successfully. That task is for the UAV NAI module, which is going to decide, for instance, the velocity of the UAV once it is aware of the real situation on its surrounding. This module is the central core of this platform, and the rest of the modules are nothing else than peripherals. Every new algorithm to be proven will be uploaded to the companion computer of the drone and will ultimately decide its next movements.

Not only the lazy algorithm of going straight forward to the target is implemented. Additionally, the ORCA algorithm has been imported and assembled, and the module is provided with an extra interface with external nodes that smooth the performance of path following. All the capabilities may be learned on section 4.6.

An interface to deal with Artificial Intelligence agents is offered in order to become this environment a so-called AI gym. For this purpose, the use of fully connected and convolutional neural networks is also available. Fully connected and convolutional neural networks have already been trained in this very same ROS MAGNA, but any further exchange of information is feasible for the interface. With this plethora of choices, several research applications find a framework to fulfil the duty of going throw the city.

Master

Finally, the Master node is responsible for defining hyperparameters and the storage of the data of every collection of simulations. It is the most used module while working with this platform as it acts as frontend. Every decision explained in the modules above, was just an addition to the possibilities. The Master must decide, at the beginning of each batch of simulations, what features define them among the possibilities offered by each module. The management of the storage where drones are going to deposit its performance data after the simulation and the definitions of each dataset are as well its issue. Master is explained on section 4.1.

Contributions

To sum up, every crucial actor for a multi-UAV mission has been developed and presents its first solutions. There is a lot to improve and so will be. In the next chapter, every module is explained in detail for a complete understanding. Moreover, the parts where the modules are proper for external ampliation or scalability will be described. With that, I offer the possibility to anyone with good ideas and willing to contribute.

ROS-MAGNA is, at the time of writing, being used on European projects of the Robotics, Vision and Control Group (GRVC) of the University of Seville. Two of them, SAFEDRONE and GAUSS are devoted to the development, implementation and testing of Unmanned Traffic Management. SAFEDRONE project is centred on real experiments from the point of view of a final user, whereas GAUSS focuses on the Air Navigation Service Provider. The third project, called INSPECTOR, will produce a final solution for a fully autonomous solar plant inspection carried by a team of UAVs.

1.3.2 Machine Learning Research

With the goal in mind to create a tool to deal with diverse Machine Learning algorithms, chapter 5 is designed not to get stacked to a single neural network architecture. Consequently, the whole solution is divided into submodules that solve different requirements of the pipeline in a sequential structure. For different algorithms researched, each one of the submodules may be altered or even substituted. In addition, the already created software may be a link in a larger chain such as reinforcement learning.

The research is divided into four subparts are Manager, the Pickler, the Preprocessor and the Neural Network boilerplate. The Manager module, corresponding to section 5.2.1, is in charge of the full process

and controls that when and how each of the other tasks are executed. As well, it decides the parameters that define the other processes, so iterates between the different experiments accomplished.

Data treatment

About the Pickler in section 5.2.2, its function is to retrieve the dataset from the database created by ROS-MAGNA. A selection of the data contained in the database must be done depending on what algorithms and mission the knowledge are required. Hence, the database information must be mapped to predefined structures suitable to be incomes for the networks. Consequently, for each different network studied, a fixed order must be defined and implanted.

The Preprocessor module, explained in section 5.2.3, receives the data generated and structured by Pickler and applies a mathematical transformation over it. This way, that data improved and normalized in order to facilitate the learning inside the networks. Results of the preprocess are lately used to revert the operations.

Neural network boilerplate

Before the data is fed into the neural network, that must be defined as a Tensorflow graph. Several branches compound that graph to define the network itself as well as auxiliary computation for learning processes or debugging. This process is accomplished by the Graph Builder module. Summing up, this module initialised a default graph and defines the placeholders and variables, constructs the operations of the Fully-connect and Convolutional stages and assembles them up. Over them, concatenates the optimizer. Finally, it is appended a branch for final use providing standalone inputs and retrievement of a single output. All those operations are accompanied by summaries and inside name scopes, that facilitate later understood of the success of the process. This process is depicted on section 5.3.1.

Finally, that graph is given to the Trainer which retrieves the input and outputs on the dataset created by Preprocess and feds them into the graph sliced into batches. Furthermore, the branch of the optimizer is executed, which updates the weights and biases until several steps take part. As the last task, the best network parameters are stored for future use that may be comparison research or real use on UAVs. This part is explained in section 5.3.2.

1.3.3 Voice Control Assistant

The task of controlling a multi-UAV cooperative mission requires high concentration and presents several sources of critical errors. A way to facilitate the job of the commander of the mission may result in a beneficial increase of efficiency and suitable fleet quantity. One of the possible ways to improve its experience may be by abstracting from the keyboard commands with complex language. This part of the project is focused on providing a solution to control missions implemented on ROS-MAGNA using its own architecture.

On one side there is section 6.3, the software stays listening for incoming voice commands. When they are detected and stored, speech is translated to text, whose content is later extracted into pieces of information. This process is accomplished, in order, by PyAudio library, Google Speech Recognition library and WIT API. This task is accomplished by the Frontend module, called this way as it acts as a user interface.

On the other side, it is found 6.4, the information extracted from the voice command is again translated to explicit movements or mission changes. With that metainformation, a message is composed to send a service to ROS-MAGNA over the ROS architecture. The Backend module is a node that makes this transformation and acts as a client service to send the command. When the whole process is accomplished the Frontend starts listening again.

2 State of the Art

The purpose of this chapter is to explain the background in which this project has been conceived. It facilitates a better understanding of the decisions taken, the chosen inputs and the tools employed during the definition and development of the project. The chapter is subdivided into the same parts that this project is divided into as those come from very different fields whose state-of-the-art is worth being explained separately.

2.1 ROS-MAGNA

Until now, no many works can be found about multi-UAS architectures implemented under ROS for the definition and management of missions. For example, ATLAS [29] is a framework developed for ROS that only addresses cooperative localization for Unmanned Aerial Vehicles (UAVs) based on cameras and fiduciary markers. On the other hand, the multi-UAV control testbed presented in [34] is focused on a GPS waypoint tracking package and a centralized task allocation network system (CTANS) developed under ROS. In the field of central control frameworks, an interesting event-based real-time Nonlinear Model Predictive Control (NMPC) Framework with ROS Interface for multi-robot systems is presented in [22].

Regarding teleoperation, the TeleKyb framework [25] is an end-to-end ROS software framework for the development of bilateral teleoperation systems between human interfaces and groups of quadrotor UAVs. Also in the area of teleoperation, considering the fact that the operation of multi-UAV system may need multiple cooperative operators, an algorithm based on position information and color information is described in [49] to identify multiple operators. The results of hand gesture recognition of multiple operators with UAV control under ROS are also presented.

Some works related to virtual reality and multiple-UAS can be also found. In [44], Unity-based virtual reality interfaces are developed for immersive monitoring and commanding interfaces, able to improve the operator's situational awareness without increasing its workload. Three applications are presented: an interface for monitoring a fleet of drones, another interface for commanding a robot manipulator and integration of multiple ground and aerial robots. Also for Unity, ROSUnitySim [37] presents an efficient high-fidelity 3D multi-UAV navigation and control simulator in GPS-denied environments.

Ground Control-like open software is at disposal of every user. Examples like QGroundControl [9] and Mission Plan [7] present perfect solutions in which the user may visualize the position and state of the airborne, change mode or arm it and upload or receive missions from the simulated or real autopilot. Those tools also provide customization of the features of the drone and the radio control.

For the global control of aerial traffic, also called U-Space, some worldwide projects are currently developing researches under the name of UAS Traffic Management (UTM) such as SAFEDRONE [11], GAUSS [4] and CORUS [2]. With the aim of aiding this kind of projects, some applications have been designed in order to provide applications that integrate the globar information such as geofences, fighplans and trackings of near and global flights. The main softwares currently operating are Unifly [12] and Airmap [1].

Closer to the topic of ROS-MAGNA, some works present an approach for the control of a UAS swarm that performs a task in a coordinated way. A guide to using ROS to fly a Bitcraze Crazyflie 2.0, a small quadcopter platform, both individually and as a group is presented in [27]. In [26], it is proposed a support system for supervision of multiple unmanned aerial vehicles by a single operator supervising its cooperative behaviour.

The FlyMASTER project [31] develops a software platform for cooperative swarming by high-level control and supervision of multi-agent UAS systems that is platform agnostic, capable of communicating with any flight controller that implements the MAVLink.

2.2 Machine Learning Research

2.2.1 Introduction

Machine learning (ML) is such an extensive definition that its SoA is not so clear to limit. It is defined as the use of algorithms and computational statistics to learn from data without being explicitly programmed. It is a subsection of the artificial intelligence domain within computer science. While the field of machine learning did not explode until more recently, the term was first coined in 1959 and the most foundational research was done throughout on the decades of 70 and 80. The rise of Machine learning to prominence today has been enabled by the abundance of data, more efficient data storage and faster computers.

2.2.2 Subfields

The machine learning field of knowledge may be subdivided into subfields concerning the nature of the input data, mathematical operations used and the purpose of its application.

A supervised learning algorithm takes labelled data and creates a model that can make predictions given new data. That can be either a classification problem or a regression problem. In a classification problem, there might be test data consisting of photos of, for instance, animals, each one labelled with its corresponding name. The model would be trained on this test data and then the model would be used to classify unlabeled animal photos with the correct name. In contrast, in a regression problem, there is a relationship trying to be determined among many different variables.

Unsupervised learning deals with data that has not been labelled or categorized. The goal is to find patterns in the data in order to derive meaning. Two forms of unsupervised learning are clustering and dimensionality reduction. Clustering is grouping alike data. Data in one group should have similar properties or features to one another. However, when compared to data of another group they should have highly dissimilar properties. Dimensionality reduction compresses data by removing random variables and holding onto principle ones without losing the structure and meaningfulness of the dataset. Dimensionality reduction makes the data easier to store, quicker to run computations over, and easier to view in data visualizations.

Reinforcement learning uses a reward system and trial-and-error in order to maximize the long-term reward. There are an exploration and exploitation tradeoff. In order to handle this, reinforcement learning algorithms integrate a level of randomness called an epsilon-greedy strategy. Epsilon is the percentage of states where the agent would take a random route and knowingly miss out on a reward. Generally, reinforcement learning algorithms begin more explorative and as the reward systems of the game are better understood, the algorithm then leans towards exploitation. The framework for reevaluating the probability in each state is based on a Markov Decision Process (MDP).

The area of machine learning that has seen the most significant results has done so by mimicking the human brain is Deep learning. It utilizes neural networks which, just like the human brain, contain interconnected neurons that can be activated or deactivated. Deep learning can fall into supervised and unsupervised learning subsections of ML. Input or multiple inputs are passed into a neural network which then processes them into one or many outputs. The neural network itself is a network of neurons grouped into layers.

Deep learning gets its name from the number of layers, also called depth. The core matrix operations that are performed on the data layer by layer are multiplication of the input by a weight, adding a bias, and then applying an activation function to the result. The output of that computation is passed along to the next neuron and the process is repeated until it gets to the end. With each subsequent pass through the entire network, a cost function adjusts the weighted connections in order to reduce error and improve the model. [38]

2.2.3 Most common structures

An overview is taken about the main layouts and types of neural networks on the currently exploited on the SoA taking a deep sight into the main features the differentiate them.

Perceptrons are considered the first generation of neural networks. Those are simply computational models of a single neuron wich feeds information from the front to the back. Training perceptrons usually requires

back-propagation, giving the network paired datasets of inputs and outputs. Inputs are sent into the neuron, processed, and result in an output. The error being back propagated is often some variation of the difference between the input and the output. Given that the network has enough hidden neurons, it can theoretically always model the relationship between the input and output. [46]

In 1998, Yann LeCun and his collaborators developed a really good recognizer for handwritten digits called LeNet [33]. It used back propagation in a feedforward net with many hidden layers, many maps of replicated units in each layer, pooling of the outputs of nearby replicated units, a wide net that can cope with several characters at once even if they overlap, and a clever way of training a complete system, not just a recognizer. Later it was formalized under the name convolutional neural networks (CNNs). They are primarily used for image processing but can also be used for other types of input such as as audio. A typical use case for CNNs is where you feed the network images and the network classifies the data.

Recurrent neural networks (RNNs) [24]are basically perceptrons. However, unlike perceptrons which are stateless, they have connections between passes, connections through time. RNNs are very powerful, because they combine two main properties. On the first hand, its distributed hidden state that allows them to store a lot of information about the past efficiently. On the second hand, the have non-linear dynamics that allows them to update their hidden state in complicated ways. With enough neurons and time, RNNs can compute anything that can be computed by a computer. However, they can oscillate, settle to point attractors or behave chaotically. RNNs could potentially learn to implement lots of small programs that each capture a nugget of knowledge and run in parallel, interacting to produce very complicated effects.

In [39] it introduced a new breed of neural networks, in which two networks work together. Generative Adversarial Networks (GANs) consist of any two networks (although they are often a combination of Feed Forwards and Convolutional Neural Nets), with one tasked to generate content (generative) and the other has to judge content (discriminative). The discriminative model has the task of determining whether a given image looks natural (an image from the dataset) or looks like it has been artificially created. The task of the generator is to create natural looking images that are similar to the original data distribution. This can be thought of as a zero-sum or minimax two player game.

Autoencoders are neural networks designed for unsupervised learning, i.e. when the data is not labeled. As a data-compression model, they can be used to encode a given input into a representation of smaller dimension. A decoder can then be used to reconstruct the input back from the encoded version.

Many more structures are presenting very interesting capabilites. Just to mention: Long-Short Term Memory networks (LSTMs), Gated Recurrent Units (GRUs), Hopfield network (HN), Boltzmann Machine or Deep Belief Networks. [32]

2.2.4 Applications

The applications of machine learning imply extremely diverse areas of knowledge. On the world of computer vision, it is used for semantic segmentation, image classification, object detection or image generation. Visual odometry or navigation and motion planning are the greatest applications in robotics. For audio, its generation, classification and events detection are the most suitable use cases. Natural language processing is used for machine translation, language modelling, sentiment analysis or text classification. Other important advantages are for time series with its classification, analysis, forecasting and prediction.

Machine learning is also currently used in other disciplines not so related to this project. Some examples would be Medicine, for drug discovery, lesion and brain segmentation or tumour discovery. Just to enumerate, areas like graphs, music, reasoning, computer code, knowledge base, adversarial studies, recommendation systems or casual inference are benefiting from these algorithms. Some tools used are word embeddings, representation learning, transfer learning, domain adaptation and data augmentation.

The research with games deserves a particular insight. Deep Blue (IBM) was a chess-playing computer which won its first game against a world champion in 1996. Later in 2017, AlphaGo algorithm beat the world champion at Go, a way more complex game with a number of possible board states so extensive that it is not possible to be fully contained on a computer.

Focusing on nowadays games influence, Atari released the code of some of their games along with a common API as an interface for the player inputs and outputs. Some other platforms are using Continous Control games. Furthermore, worldwide games that are currently been played, like Starcraft, Starcraft II or Dota as being real-time strategy games. [3]

2.3 Voice Control Assistant

Several projects on the literature are currently attempting to develop a general purpose voice bot capable of receiving queries and return information during a fluid conversation. For many years, this kind of technology has produced poor approaches, always very focused on the single application and completely constrained by it. On the last few years, the field has suffered a hight improvement thanks, among other reasons, to the renaissance of Artificial Intelligence and much more powerful computation capabilities. Several bots are now used to facilitate user experience and reduce costs by offering an almost human speech, succeeding by far on the Turing test (differentiability of the machine against a human).

Therefore, the state-of-the-art studied on this subchapter is focused on speech recognition, as the application is a task more particular for each case. The speech recognition process can be subdivided into three stages: voice detection, speech to text and natural language processing. On the next paragraphs, those areas are studied.

Most modern speech recognition systems rely on what is known as a Hidden Markov Model (HMM). This approach is based on the assumption that when a speech signal is viewed on a short enough timescale, for instance, ten milliseconds, can be reasonably approximated as a stationary process, whose statistical properties do not change over time and can be studied and compared for voice detection. One example of many libraries that conduct this process is [40]. Most rest of the libraries cited later has its own voice detector.

The power spectrum of each fragment is mapped to a vector of real numbers known as cepstral coefficients. The final output of the HMM is a sequence of these vectors. To decode the speech into text, groups of vectors are matched to one or more phonemes, which is a fundamental unit of speech. This process requires training, since the sound of a phoneme varies from speaker to speaker, and even varies from one expression to another by the same speaker. A special algorithm is then applied to determine the most likely word (or words) that produce the given sequence of phonemes.

In many modern speech recognition systems, neural networks are used to simplify the speech signal using techniques for feature transformation and dimensionality reduction before HMM recognition. Voice activity detectors (VADs) are also used to reduce an audio signal to only the portions that are likely to contain speech, preventing the recognizer from wasting time analyzing unnecessary parts of the signal. [23]. Many software development keys (SDKs) may be found in the literature, being the most highlighted example Google Cloud Speech [6].

In the early days, many language-processing systems were designed by hand-coding a set of rules, for instance by writing grammars or devising heuristic rules for stemming. In the late 1980s and 1990s, much natural language processing research has relied heavily on machine learning. The machine-learning paradigm uses statistical inference to automatically learn such rules through the analysis of large corpora of typical real-world examples (a corpus (plural, "corpora") is a set of documents, possibly with human or computer annotations). [13]

Some examples of those libraries are WIT [14], which is the one used for this project, apiai [30] and Pocketsphinx [41]. SpeechRecognition library [21] acts as a wrapper for several popular speech APIs as the previously exposed, which makes it highly flexible.

3 Main Software Tools

In this chapter, the main tools employed will be described. The logical sequence of explanation is as follows: an introduction of the tool and a justification of its presence in this project, an insight of its working philosophy, a light overview of how it is implemented in the project and additionally, relevant auxiliar functionalities.

3.1 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed. [8]

Nowadays, Python is used for quick prototyping but with an effective system integration. It has raised him to be one of the top 10 programming languages most used almost since its creation in 1991 by Guido van Rossum. Today, it is ranked as top 1 by IEEE [https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages]. In fact, cutting-edge technology companies such as Google, Boston Dynamics or Amazon have declared Python as a key language for their new developments.

In which concerns to this project, Python is the universal language. Used in every single part, different Python-based libraries take part in the solution of problems which will be more specifically explained in previous sections. Although some used tools obviously make use of other different languages such as C++, every piece of code specifically written for this project is implemented in Python 2.5.

3.2 Robot Operating System, ROS

The Robot Operating System is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms. ROS was built from the ground up to encourage collaborative robotics software development. For example, one laboratory might have experts in mapping indoor environments, and could contribute a world-class system for producing maps.ROS was designed specifically for groups like these to collaborate and build upon each other's work. [10]

3.2.1 Challenges

In order to provide a whole solution for a multi-degree-of-freedom multi-robot world, the system must undertake the conjunction of a range of common characteristics depicted on Figure 3.1.

The first challenge is plumbing all the information that must flow between parts of robots, between robots or between any atomic agent which would take part in the solution. That flow must be well structured not only in the network layer with a well-defined structure but also in the transport layer with a specification of messages and agent roles. The second issue is to provide different tools to accomplish the necessity of

implementing such a complex application and understand what is going on. These tools must facilitate the characterisation and representation of architectures, networks, plans and robots implemented.

Every single agent, whatever its task be, needs to make use of different capabilities to fulfil its task. It may be a very specific solution for a single application or a widen required one. Therefore, it brings up the third hurdle: a common system should provide a sort of capacities for faster development and reuse of best solutions. Directly derived from the last one, the fourth challenge is the creation and maintenance of an ecosystem materialized as a community of expert and beginner developers who update and improve ROS and help each other.

Figure 3.1 ROS challenges. The framework integrates a range of transversal features required for control for robots.

3.2.2 System

The solution offered by ROS is in constant development and evolution but has a base structure which will be briefly described for a minimal understanding of the rest of the project. It is intended to be read for those who have never worked with ROS and I highly recommend them to get to well-know it via tutorials and implement it on their robot applications.

Architecture

On one hand, there is the distributed network architecture. A simplified diagram is presented in Figure 3.2 As described before, it is based on atomic agents who represent an element of the robot called nodes. Avoiding confusion, it must be clarified that an element of the robot not only is referred to a physical part such a joint or a sensor but also an intelligence one such an algorithm or a visualizer.

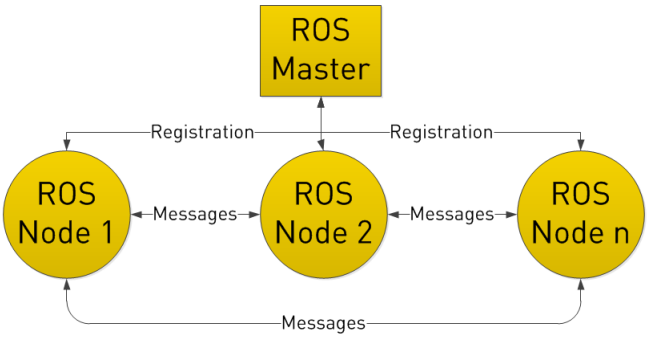

Figure 3.2 ROS structure. A central node called master manages communications between itself end the rest of slave nodes.

Those nodes are concerned with its represented robot's element by means of the reading of its status (i.e. sensors) and the capacity of changing it (i.e. controlling actuators). Thus, nodes are the gateway between each robot's element and the rest of them. Inside each node, different capabilities would be implemented as required and used sometimes or constantly during its lifetime.

Communications

On the other hand, the communication between nodes is what gives ROS its real power. There exist three different kinds of communication depending on the nature and the interval of time it must is active. Topics are the first one and consist of an open container in which data can be dropped at any time by a publisher and instantly read by a subscriber.

The second type is a service and is a little bit more tricky than topics. A node called server offers to the rest of the topics the opportunity of requesting him to accomplish a task for them. The node which is interested in it becomes a service client, makes the request, the server node completes it and returns a response with

information about the result of the fulfilled task. Therefore, a service takes a short period and just takes place once (although it would be triggered by the same or other clients more times).

The third type of communication is called an action and is not actually a new one but a combination of the both just explained. A service is requested by a client and started by the server. Then, a topic is opened in order to provide feedback between those two nodes. The action would take whatever duration it requires to be fulfilled and then the service is finished with the sent of the response.

A common characteristic of every type of communication is the need for a message description. The message defines the structure of the data that will flow in any movement of information. Thus, any type of topic, service or action corresponds to its message structure.

Specific tools

Amongst every function designed to deal with the network architecture described above, there exist some other capabilities which may help to take a whole overview and to visualize what is actually happening in this virtual environment called ROS. The first one is called RQT, a Qt-based framework for GUI development for ROS. One of the most useful tools that this package provides is RQT GRAPH, shown in Figure 3.3. It is used to visualize the node-based network described in the last subsection. There, it is easy to distinguish between nodes and its interconnections materialized by topics. The graph shows which nodes are publishing and which ones are subscribed to them. But not only network graphs can be shown, RQT IMAGE VIEW offers the capacity to display a different kind of images such as RGB or depth if they are travelling through a topic.

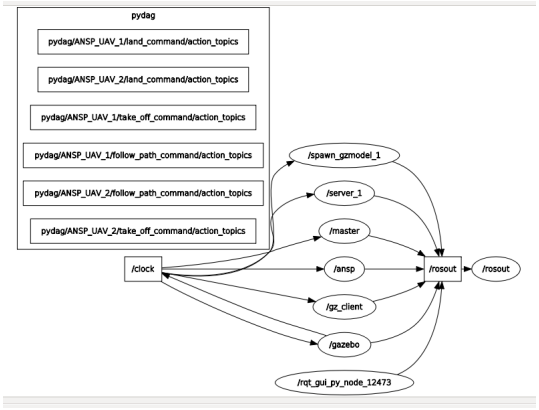

Figure 3.3 Sample of an RQT graph containing several nodes (ellipses) and its topic communications(rows and boxes).

Another extremely useful ROS tool is Rviz, presented on [17]. This package offers a 3D visualisation in which different types of ROS information may be displayed. Sensor outputs such as a point cloud will take place in its real 3D position what makes very easy to understand this information. In advance, three axis reference frames (TF) will also be placed so the correspondence between robot physical elements' position and orientation is visualized and concerned with other kinds of information. As a final virtue of Rviz, not only actual flowing information can be displayed but stored bags of past simulations also. Therefore, critical situations may be visualized several times for deep examinations.

3.3 UAV Abstraction Layer, UAL

This library implemented by GRVC research group from the University of Seville allows the user to control multiple UAVs in real or simulated experiments without prior knowledge of the specific autopilot based on ROS [43].

Before going deeper on UAL, the ROS package MAVROS [16] should be explained. Its purpose is the communication with the autopilot of the UAV, i.e., implements a node whose robot's element associated is the UAV's autopilot. The communication is achieved via MAVLINK protocol, which was developed for micro aerial vehicles due to its low weight capacity and therefore, light communication devices.

Back to UAL, it would be said that this package is a bridge between MAVROS and the user. Abstracting him from the numerous params, modes, topics and services that it uses, UAL offers a way shorter interface

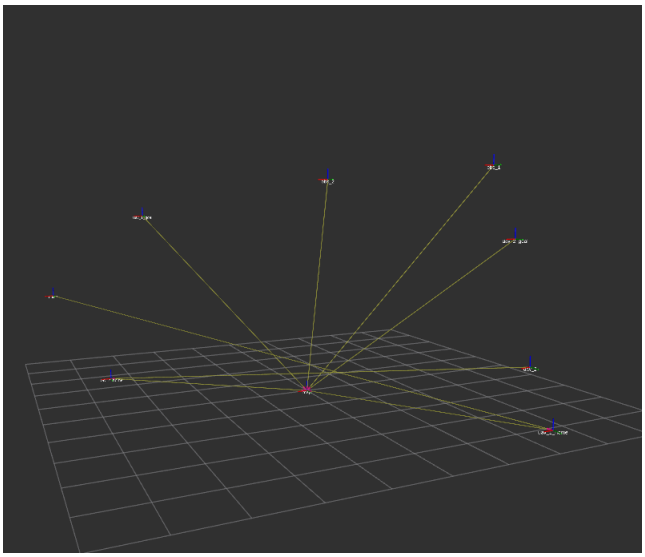

Figure 3.4 Example of an Rviz visualization. Transforms (TFs) are placed around the map origin in order to precisely place robots and its diverse components.

with the autopilot. It allows the user to command a waypoint in the space with local or global coordinates or as well the desired velocity, always in offboard mode. UAL also offers information about the state of the UAV and takes control of it until the action specified has been accomplished.

This project makes use exclusively of UAL for control and command of rotary-wing UAVs but instead uses directly MAVROS for fixed-wing UAVs. This situation is due to non-compliance of UAL with this kind of airborne, but if an upgrade was made, it would be done in this project as well.

3.4 SMACH

3.4.1 Introduction

SMACH is a task-level architecture for rapidly creating complex robot behaviour. At its core, it is a ROS-independent Python library to build hierarchical state machines. Is essentially a new library that takes advantage of very old concepts in order to quickly create robust robot behaviour with maintainable and modular code. SMACH is useful when you want a robot to execute some complex plan, where all possible states and state transitions can be described explicitly. This basically takes the hacking out of hacking together different modules. [18]

It would be thought that a state machine is not essentially required. And it may be true... at the very beginning. A multiagent mission definition is not a trivial challenge and exponentially gets complexity as more and more agents are gifted with more and more capabilities. Controlling the state of a mission as a compound of the state of every single agent and deciding what to do next depending on that may is a tricky task if states and movements between them are not flawlessly bounded. An example of state machines may be found on Figure 3.5 by visualising it on SMACH Viewer [19].

In addition, another highly beneficial point of a state machine is the possibility of reusing states that perform the same behaviour at different times of the mission, such as hovering at the current point of the space. This is trivial when linking closed boxes but maybe very tough when making plumbers on a blurred code.

Its implementation in ROS, although it is purely ROS-independent, makes its integration inside the environment even easier. Tools such as ROS-oriented types of states facilitate significantly the interface and the issue of deciding how to implement a task inside a state. And what is more impressive, there is the possibility of visualising the state machine and its interactions but also control current states during a mission.

3.4.2 Philosophy

SMACH would be a little bit confusing while first reading tutorials but the reader quickly gets into the idea of first defining states and later assemble them all together. There exist plenty of different atomic states and types of state machines. Additionally, the user may customize them or create new ones from scratch

Talking about states, basic ones consist only of a python class with an initialization and a bigger piece of code to run when the state is active. Likewise, there exists another type of state even shorter that only executes a callback when the state is activated. More complex ones consist of a built-in implementation of ROS action servers or clients, which either trigger or accomplish them.

One step above in the hierarchical structure, we found the most simple type of state machine. This flow diagram is composed of single states, whatever its type among the above described, interconnected by means of so-called transitions. Each transition is the linking from the state's outcome and another state. Due to the existence of different outcomes from one single state, it will point out to different states depending on the transformation of the data throughout. Of course, every state machine has its own outcomes.

But how is the data flowing throw the net? "User data" inputs and outputs are strictly defined for each state. Consequently, the user can decide what information is going to receive the state at execution time, in other words, what variable's value.

Briefly highlighting some types apart from the basic one, concurrence state machines allow the user to activate multiple nodes at the same time. It implies the problem of dealing with multiple state outputs for only one state machine output. Other kinds of state machines are sequences and iterators.

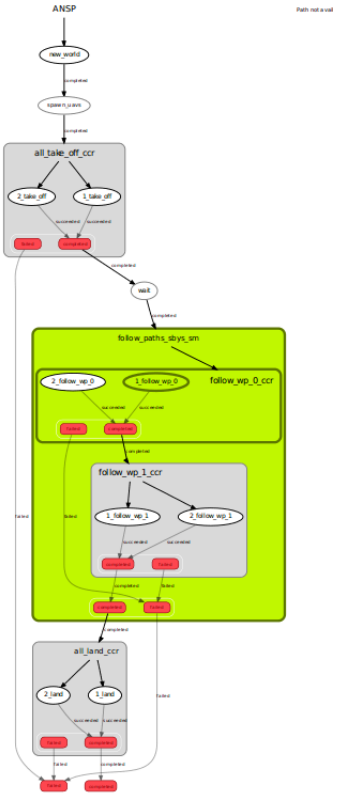

Figure 3.5 Example of a State Machine implemented with SMACH. Nodes (ellipses) are associated into nested state machines (boxes) and related to each other by its outcomes (rows). The green state machines are currently active.

Putting together all these structures, the complexity of the diagram is as high as the user requires. This is performed by means of nesting state machines inside each other, always controlling transitions and user data flow.

3.5 GAZEBO

Gazebo is a well-designed simulator that makes possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. This software offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. It owes his power to a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces. Best of all, Gazebo is free with a vibrant community. [5]

None of the utilities described so far would make sense without a link with the real world. It is brought up by Gazebo, which is one of the most realistic simulators and the most spread one for ROS developers. This program offers the researcher the possibility of testing its developed robot or algorithm with high confidence for future real experiments. Not only physics engine and visualisation are offered by Gazebo but also a huge amount of scenographic and robotic models ready to be used with all physic description implemented. Figure 3.6 presents an example of a GAZEBO simulation.

In which concerns to this project, Gazebo is the only simulator used as part of the legacy received from UAL. In fact, all communication with the simulator is already implemented by UAL and only a few changes have been required. Talking about models, all models available for this project are built-in Gazebo and, more precisely, rotary-wing UAVs were already customized by UAL.

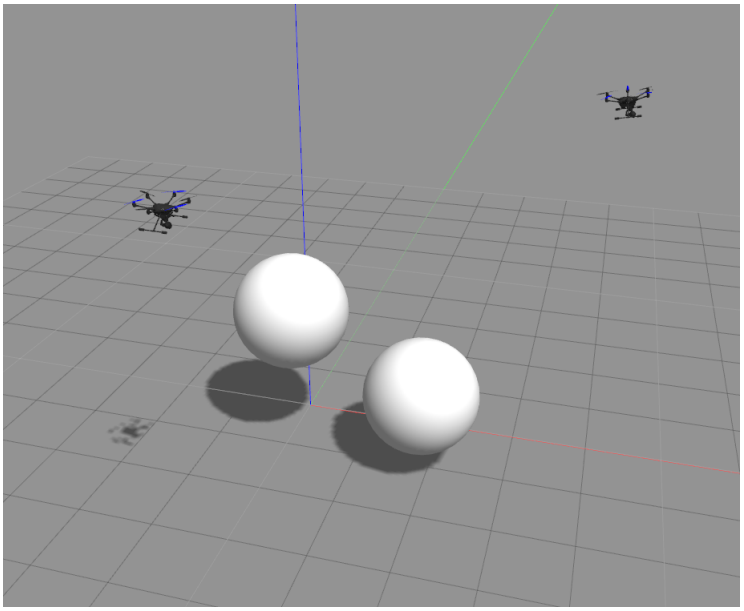

Figure 3.6 Example of GAZEBO simulation where two UAVs are placed around two spherical obstacles.

3.6 Tensorflow

Tensorflow is an open source software library for high-performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains. [20]

3.6.1 Intro

Well-known for every beginner or master on machine learning and artificial intelligence in general, Tensorflow is one of the most spread libraries for complex and distributed computation. As a standard, there is almost no algorithm that cannot be accomplished with this library. This makes it perfect for complex data architectures such as neural networks, either simple fully connected or more computational resources requiring as convolutions, recurrent or GANs.

The capacity of distributing operations amongst available devices is even more beneficial when only one computer is making use of a physical engine as Gazebo and calculating heavy algorithms at the same time. Consequently, CPU/GPU based Tensorflow is essential for this project. Not to mention that making use of such a library is way time saver than implementing neural networks or any other machine learning algorithm by scratch. The clarity and simplicity meant for Tensorflow make easy to learn deep learning and to develop new customized structures while perfect understanding of data, or tensors, are flowing and transforming.

3.6.2 Work methodology

As an overview of Tensorflow's projects philosophy, it should be remarked that first of all a graph must be defined. A graph consists of every variable or constant tensor, placeholders for future data, done amongst them and some other features. All together characterize how some inputs experiment a progressive transformation to give some outputs as a result, always in the form of tensors. Furthermore, the training process must also be codified by an objective function, its training algorithm and its evaluation.

After graph definition, the second step consists of training characterization. How dataset is being sliced and introduced in the graph at every step of training, which sections of the dataset are linked to each placeholder of the graph or how many times the training process is going to take part.

3.6.3 Tensorboard

As a last exceptional advantage of Tensorflow, Tensorboard tool should be highlighted. It consists of a graphical environment for a better understanding of the whole process modelled by Tensorflow. Along with the graph definition, a sort of naming for every part of it would be provided by the user so Tensorboard intuitively displays every single part of it: constants, variables, placeholders and the whole sort of operations that relate them. This is a great tool to understand and debug algorithms.

Once the graph is controlled, and the training process has been accomplished, all data generated, if named, is displayed in a variety of diagrams. This is very useful for comparing weights and biases or for characterizing how the accuracy of the algorithm improves as it is trained.

As if this were not enough, different training processes can also be compared in order to decide which hyperparameters such as learning rate or network structures fit better in each case.

3.7 WIT

A web application that provides natural language for developers. Wit.ai makes it easy for developers to build applications and devices that you can talk or text to. Their vision is to empower developers with an open and extensible natural language platform. Wit.ai learns human language from every interaction and leverages the community: what's learned is shared across developers. [15]

The core of WIT is the transformation of a natural language sentence in decomposition of single pieces of atomic information as shown in Figure 3.7. This is not about splitting into words, is about extracting the real intent of the sentence and classify the relevant information it offers. All this process is computed in remote servers so the application needs to be online during the request.

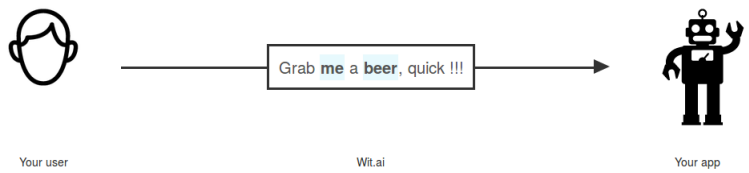

Figure 3.7 WIT example where a user sends a command to WIT, which interprets the sound and translates into information intelligible by the app.

At the creation of the WIT application, the developer is expected to introduce as many examples as possible in order to teach WIT the intent and the information it should extract from each segment of the sentence. A huge amount of training sentences will make the application more robust so it gets to know all the future possible queries.

As a result of this and once it is trained, WIT will be able to receive a never seen sentence and return a list of pieces of information, its value and its confidence. The user consequently introduces the most expected values into his own application.

4 ROS MAGNA

ROS-MAGNA (ROS-based Multi-AGent mission maNAgement)is a general framework to manage cooperative missions for multiple UAS based on ROS. Multi-UAS mission specifications are implemented as state machines that control the behaviour of the different UAS independently of its autopilot on-board. In addition, a virtual world generation tool has been designed to offer an overall visual understanding of the state of the main geometrical elements and its progress on the mission. The framework is focused on offering useful testing tools on the whole development pipeline of UAS advanced functionalities, from simulated software-in-the-loop and hardware-in-the-loop to real UAS even at the same time and arena.

In this chapter, every module that compounds ROS MAGNA is going to be explicitly and widely explained. In the end, the reader will have obtained a complete comprehension of the tasks that each node performs, the flow of data between them, how the user is expected to interact, the options offered and the way to improve or enlarge its facilities. The order followed here differs from the order of the introduction. The reason was the intention of presenting a logical use overview. Now, that insight is known, so a more structured order will make it easier to explain how it works.

A detailed diagram for a better comprehension of the workflow of the framework is shown in Figure 4.1. On it, ground and aerial segments are separated, letting Master and Ground Station nodes as well as some other utilities on the land, and the nodes that govern the UAVs, onboard the aircraft. A deep insight into all these components is studied in the next sections.

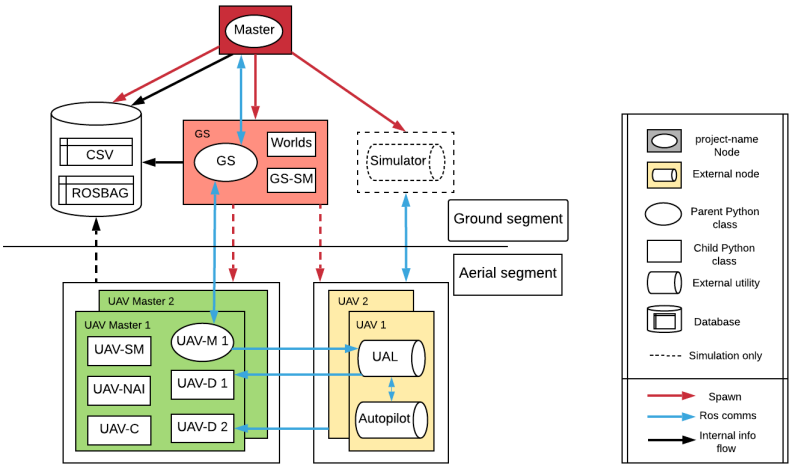

Figure 4.1 ROS MAGNA general structure. Separation of the ground and aerial segment within its node components, utilities and ROS communications..

The source of ROS-MAGNA can be found in GitHub [45]. For further technical information, the Wiki is at the disposal. It contains more specific information such as the name of ROS topics, services or actions as well as the message that defines them.

4.1 Master

4.1.1 Motivation

The Master node consists of a single Python class and is placed at the very top of the structure diagram of ROS MAGNA. Its interactions are presented on Figure 4.2. This is the only node that remains active from the beginning of the collection of missions performed until the last node of the last mission is deactivated. Inside this module, almost every hyperparameter is created, first connections with the adjacent tools as ROS and gazebo are made, folders to store the data to be generated are created (if already exist, the user is asked about how to proceed) and a first definition of the collection of simulations is also stored.

Most of the enumerated processes must be effectuated for every new mission, so the most important duty of this Master module is to start the sequence of program, model and nodes spawnings that constitute the whole framework for every simulation. As a consequence, it is also its issue to control if the performance has been acceptable and store summarizing information at the conclusion.

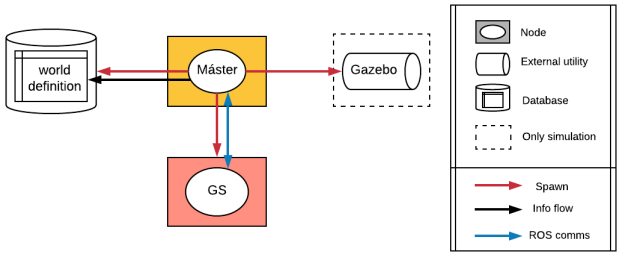

Figure 4.2 Master node relations diagram. This node spawns the database (DB), GS and the GAZEBO if required. It stores information on the DB and maintains communication with the GS.

4.1.2 Hyperparameters

Every collection of missions is conceived to be different from another. The determination of that variations is settled via the hyperparameters shown in Table 4.1, and must be defined at the beginning before any execution is realised.

Table 4.1 Hyperparameters that define the main features of a single bunch of simulations or a real mission. Those are shared by every node on the architecture.

Hyperparameter	Description
World	Type of the world or sceneario created.
Subworld	Subype of the main world.
Mission	Global mission that characterizes the state machine.
Submission	Subtype of the global mission.
n dataset	Number of the dataset to create.
n simulation	Number of simulation where to start inside the dataset.
N iter	Bunch of simulations developed in the defined dataset.
SMACH view	Flag to decide if SMACH introspector is activated.

4.1.3 Master functions

Definitions

The initialization of this class brings the global definitions of the parameters of the project. The Table 4.2 details each parameter and its definition.

Table 4.2 Parameters defined in Master as frontend that are shared as hiperparameters with rosparam.

Master parameter	Description
Hyperparameters	Global mission that characterizes every UAV's role
Gazebo client	Run visualizer for Gazebo
Rviz	Run visualizer for ROS information
Home path	Root path inside the device of the user
Save data flag	Decision to store the mission data generated. Useful when debugging
Save rosbag flag	Decision to store the ROS data generated. Useful when debugging

Creation and control of storage folders

The main tasks of the Master node is to create, if it does not already exist, the data storage folders. Once the home path is defined, it creates three levels of paths nested by the world and mission type, the provided name, the dataset and the simulation. The structure to construct the storage path is presented on Code 4.1. When it has all the information, the function DataExistanceChecker controls, before the definition of the mission dictionaries, if that simulation already exists. If it were not, required nested folders are created, and the process goes on as assumed. The problem is presented if that simulation is already created and there is a risk to delete it. In that case, the user is asked what to do as explains Table 4.3.

Code 4.1 Storage folders logical cascaded structure.

```
Code
Data Storage
  Simulations
    World
      Subworld
        Mission
          Submission
            Name
              Dataset id
                Simulation id
```

Table 4.3 Possibilities offered to the user on the terminal when a storage confliction is detected.

Option	Description
Continue	Delete all previous data and spawn and spawn new folders
Append	Add new simulations data alongside the old ones.
Do not save	The step for storing data is skipped.
Abort	Simulations are skipped and no more nodes are started.

Initialisations

When all parameters are known, and the mission has been established, a variety of initialisations must be carried. In the first place, if on SIL or SIL, Gazebo is launched as the physics engine where every model will be spawned. A new node is created, and consequently, ROS MAGNA starts its life. Once the contact with ROS is established, the hyperparameters can be uploaded in order to make them available for the rest of the upcoming nodes.

As a last initialisation, the Master node advertises a ROS service to be used in the future by Ground Station node. This communication channel will determine the end of the current mission and the Master will continue to the next one.

Table 4.4 Initialisations at the beginning of Master for every new mission accomplished.

Action	Description
Gazebo	ROS launch
Ground Station node	ROS launch the with spawner
Simulation termination acceptance	Advertise a ROS service for Ground Station

Batch of simulations

Everything is ready now for the core part of this module: iterations to perform as many simulations as defined on the way that has been set. Before that, the user may decide not to continue if a conflict with storage existed. Hence, that flag must be checked. Everything executed in the rest of this section is useful once for the current simulation so it will be repeated every time a new one is started.

The number selected for the simulation is updated as a hyperparameter, so the rest of the nodes know where to store its generated data. A ROS launch command starts the Ground Station node. The simulation finished flag is deactivated waiting to be activated by the Ground Station node. A timer is initialised to assess the time elapsed for a performance.

Master sleeps while the mission is being developed. There exist two possibilities to exit from it. On the one hand, everything fits and the targets are achieved. In that case, a notification from Ground Station node will be received via ROS service and next simulation will take part. On the second hand, if something does not work as it should and agents get stuck, the mission will elapse too much time. This situation activates the exit flag itself without waiting for Ground Station, and the simulation is preempted, shutting down manually the rest of the nodes.

Processes termination and data storage

It is an obligation to care that every started process es adequately closed. At the end of every simulation, a range of processes are feasible to remain alive. This would cause fatalities for the next simulations and hence Master manually finishes them on a secure way. These processes may be the autopilots or Mavros (external node that implements the protocol to talk to PX4) nodes in the first level. In the second one, there take place the server of UAL and Python. The reason for thhese two cascaded levels is due to the probabilities of the failure and the severity of its consequences.

The chosen level of killed nodes varies. If the "too much time elapsed during simulation" flag is raised, a purge until the second is yielded. In contrast, if the simulation was acceptably accomplished, it remains on the first level. Once every mission of the dataset has been performed, another second level purge runs to let clean the computer processor.

Finally, at the end of the batch of iterations, the information that summarises the whole bunch is gathered is stored. First, a dictionary is created and then, the information about the performance of every simulation is read and appended to the dictionary. That information corresponds to the name of the simulation, the success achieved and any message that may have been included.

Table 4.5 Data stored summarizing a batch of simulations.

Data	Description
Nº simulation	Identifier for the rest of data
Success of simulation	Activated if a collision occurred at any instant
Message	Any kind of extra information for a more verbose definition

4.2 Ground Station

4.2.1 Motivation

This node is spawned by Master every new iteration. Inside it, there is a sort of utilities for the management and control of the multi-UAV mission. Ground Station (GS) is the central node around which the rest of the nodes play its role. Composed of two different Python classes that will be explained later, this module

commands when each UAV accomplishes a task. That way it ensures that UAV nodes do not execute actions freely, but the whole government is a well-orchestrated mission. Therefore, GS does not directly contribute value to the mission data but is essential for it to be properly generated. As the central controller, it has to maintain continuous contact with the rest of the agents. The main interactions are shown in Figure 4.3.

On one side, GS must inform Manager of the end of the mission. On the other side, it spawns and gives functional orders to the single nodes related to the drones. All those actions depend on the current state of the mission. Consequently, that state is defined by the state machine.

The mission state machine is so complex that another Python class is defined explicitly for that purpose, different than the GS itself. The construction of that state machine, and hence its Python class, will be explained in a different section. That separation corresponds to the necessity to know well primarily how the GS core class does work and what possibilities offers.

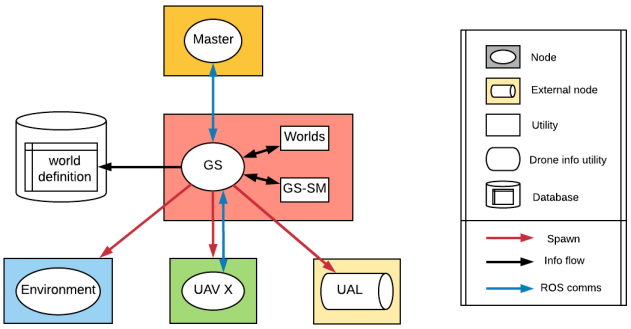

Figure 4.3 GS node relations diagram. Environment, UAV and UAL nodes are stored, information exchange is done internally between node submodules and externally with the rest of ROS nodes.

4.2.2 GS functions

Initialization

When the module is launched, a set of initialisations must be executed after starting the ROS node. As it is done in the rest of the modules, the first process performed is to get the definition of the mission from the ROS hyperparameters.

Some flags are set by default. In order, the simulation success is set to "true" and will not be changed until the assessment of the performance does not detect any collision. Besides, a list composed by the state of the drones is set to land as all of them will be spawned in that state. Finally, another list including the information of a collision occurred concerning every drone is set to negative for the moment. All of them are suitable to be changed during the mission.

Now everything is ready to init the node and start listening to the different ROS channels employing GS listener function as described in Table 4.6. That attention to the ROS network is composed by the advertisement of a server for the UAV-Manager (UAV-M) nodes to actualise its respective state. That state has information about the current status but also about the occurrence of collisions.

Furthermore, the listener function starts to listen to both static and dynamic Transformation topics within the information of different reference frames in the space referred to UAVs and obstacles' position. That information is required to compose a rosbag with the geometrical information of everything befallen during the mission.

The last step of the initialisations is composed by the creation of the state machine object and its execution. Once the mission has been executed and is finished, a finisher function ends the nodes and other processes, and the rosbag is also closed.

Starter functions

The first function is used to set the spawn received features of each UAV for UAL and simulation agents to use them later. Those features are its model, its position and its yaw orientation. This ROS parameter is later read by UAL and defines what will be spawned. In order to accomplish that, that parameter is gotten first. Subsequently, the received features are copied to the saved param in its respective field and uploaded to update it. All that information used for spawning is summarised in Table 4.7.

Table 4.6 ROS channels opened by the Master to maintain contact with GS during a whole mission.

Channel	Description
UAV state actualization	Service where drones send its current status once at change.
Static transformations	Topic which to read obstacle and origin positions from.
Dynamic transformations	Topic which to read every drone position from.

Table 4.7 GS spawned parameters to inform UAVs of its features and shared with the rest of nodes as hyperparameters.

Parameter	Description
Initial position	The vertical projection of the first waypoint of its path
Initial yaw	The orientation to desired for its first objective
Model	The kind of airborne that must be spawned

Besides, it is offered a utility for the spawn of the drones. The first task to accomplish it is reading the specifications of each one of the drones provided in both the JSON of the general UAV model configuration and the specific mission UAV configuration. Those files store information about everything that has to be spawned along with the respective node. Variables such as the use of UAL, the simulation nature or the autopilot type are amongst the different specifications required. Once that information has been gathered, it is written in the general XML that copes with every accepted configuration.

A final starter function is left. Its goal is the creation of the single storage folder for the current simulation number. It follows a cascaded methodology to define the final path constructed from the hyperparameter information. The cascaded workflow is due to the possible problem of the inexistence of any of the parent folders, in which case, it is created before continuing using children. Once the whole path is complete and checked to exist, the rosbag is created at the bottom of it corresponding to that single simulation number. From now on, all the listened transformations explained above may be successfully stored.

Finisher functions

There exist three ways for the GS to decide that the mission has finished. On the basic one, every UAV informs about the success of the performance of the last state of the state machine of the mission. If no information from the UAVs arrives within the maximum expected time for the mission, it is preempted. However, there exist the possibility that any of the drones informs about a collision. On that case, the mission is also aborted, informing as well to every agent to finish its current state and land.

Finisher functions are called once the State Machine has finished its execution. They have the mission of closing every thread and node that has been opened for the current simulation, saving the global simulation information on disk and informing of the end to the Master.

The main function of this task is Die. It is a short sequence of other functions calls in a logical order. Table 4.8 shows that sequence and all of them are detailed after it.

Table 4.8 Mission threads closing steps.

Step	Description
Save world definition	Storage of how the mission has been performed
Finish UAVs nodes	Send a ROS topic message to every drone to finish its activity
Finish UAVs models	Call a ROS service to Gazebo
Erase all obstacles	Call a ROS service to Gazebo
Termination command	The kind of airborne that must be spawned
Signal shutdown	Terminate activity and close this node

The SavingWorldDefinition function accomplishes the storing of the data. It is required to update the hyperparameters with information data, so the actual ones are retrieved from ROS parameters. Success

and collision lists have been built during the execution of the mission state machine. Subsequently, those lists are retrieved and added to the global parameters dictionary as one single variable. That complete hyperparameters variable is both uploaded to the ROS params and stored in the CSV file, updating it.

As a second step of the finishing part, it is found a function that ends the UAVs nodes. Each drone individually advertises a ROS service to receive the die command; this will be further explained in the UAV-M section. The GS node needs to be a client of those services and send a dying request to all of them. The UAVKiller function accomplishes that task.

When there are no more UAVs nodes, GazeboModelsKiller applies a Gazebo service to finish its models. That call is different from the ones done internally as the information required is only the model of the drone, and an identifier of the number of that model there existed at its creation moment plus one.

Once there are no drones, neither its nodes nor its model in Gazebo, obstacles are also deleted. As will be explained in the Worlds module, every obstacle is handled by a Python class. That class offers a method to delete its model. The deletion of all of them is accomplished directly by a method of the Worlds class.

Gazebo is empty this far. It will be closed, but it is not the duty of GS as its goal is only for current mission and it would be useful for more of them. The only task that is left is to inform Master employing simulation termination services, accomplished by SimulationTerminationCommand. That function merely creates a client for that service and sends a positive request.

Last assignment to finish with GS is its termination. The node ends, and everything that it brought up is cleaned.

4.3 Ground Station State Machine

4.3.1 Motivation

This far, it has been explained every possibility offered by the Ground Station. Besides, the way of use and how it communicates with the rest of nodes are also detailed. However, the question of how a mission is composed and orchestrated has not been answered yet.

As it was revealed in the hyperparameters, the name of the mission and its submission were settled at the creation of the Master node. It points out to a JSON file in which the whole mission structure, it means its state machine, is defined. The main issue now is how to transform that structure in fully defined steps concerning every UAV and, what is more, its interconnections.

The best tool offered by ROS for this purpose is SMACH, as explained in the Tools section. Table 4.9 explains the different steps to be accomplished in the creation of a state machine with SMACH.

Table 4.9 Steps that must be taken into account to properly create a state machine that safely defines a full cooperative mission.

Step	Description
States identification	Definition and identification of atomic states
States inputs and outputs	Identification of data required and generated
ROS Action states	Transformation of states into that type to facilitate the application
State grouping	Concern batch of states and how they relate with each other
Complex hierarchies identification	Relate groups of nodes with built-in structures such as Concurrences or nested SMs
Structure dataflow	Define how data is transformed throughout the complex hierarchy
Recurrent complexity	Amplify the complexity up to required nesting different hierarchies

SMACH presents a coding philosophy of definitions and inclusions that will not be further explained here as it is a library itself issue and is no in scope. In this subsection is to be described how that library is used to create the desired State Machine (SM).

4.3.2 SM building system

Introduction

The basis followed for this challenge is started with de definition of a GS SM class inside which the hole state machine is to be defined. It receives as heritage every method and attribute from GS class so, in essence,

is merely a part of it. In advance, the execution of the states varies depending on its type. For callback states, a single piece of code is executed when it is activated. For Single Action states, the action goal is built by a function, and another one splits the result. The chosen type is explained for every kind of the next structures, but the content of those state functions is only a decomposition and a direct call to UAV-M central utility.

In the initialisation, a global state machine is created, and the rest of the structures are defined inside. This global SM is executed once for mission accomplishment. Some initialisations are executed here to give a value to begin by such us assigned UAV ID for a state, waypoint number to follow and ID of pursued UAV. After that, the GS SM definition list is read. For every single element, the function "add sm from CSV" is in charge of splitting its information into a nested State Machine. This step will be further explained in the next subsection.

Finally, the flag hyperparameter to use SMACH View is consulted. In case the flag is raised, it is started the introspection server for the dynamic graph visualisation. The arguments it needs are a ROS direction where to publish and the object of a State Machine to be inspected. The introspection is executed at the moment of creation, and it stays asleep until the execution of the state machine when it starts reading and sharing.

State Machine Structure Elements

The function Add SM from JSON reads the definition of a part of the mission and codes it into SMACH information. It also requires the current SM at its actual state and the parent definition.

Primary, the type field is checked to decide what kind of structural element is included in the state machine. The different possibilities can be consulted on Table 4.10. The complex structures are characterised by having states inside, so the function calls itself recursively as deep- it means, as many times- as it is defined in the JSON definition.

Those state machine structure elements are described below to get a full insight of which functionalities offer and what are useful for.

Table 4.10 Elements that define the nested structure of a state machine. An unlimited depth may be achieved by introducing any element inside other as many times as required, following its defined internal structure.

Structure element	Description
CallBack State	Simplest state that only executes once a piece of code.
Simple Action State	Action associated state that does not finish unit it has been accomplished.
General State Machine	Basic state machine that activates a single state and has multiple outcomes.
Concurrence	State machine that has accepts multiple active states.
Sequence	State machine that facilitates the interconnection of a sort of states in a row.

Initially, if the structural element is one of the three kinds of state machines exposed, it is used a function called IdsExtractor that updates the field "ids" from the definition. This field defines the replications applied to the state inside it. If an integer or a list defines the field, it is standardised as a list. In contrast, if it is "all", the adjacent field "ids var" is also read so to understand what to fulfil the "ids" field with. For instance, if "ids var" is settled as "uavs", the "ids" field becomes a list containing the id of every UAV that takes part on the mission. This field is later used for the name of the structure and to assign the values of the parameters.

According to those type of elements, the field "occurrences" is the one to define the child parameters, so the one used for the definition to make the own recurrence. Due to the working philosophy of opening and closing of state machines, every time a new element is added it is done at the currently opened branch and level of the full state machine.

Concurrence and Sequence state machines require further explanation given its more complex nature. In one hand, a Concurrence needs a specific definition of outcomes of the various branches it has activated to define the Concurrence its own. Also, each new branch is added to the SM as if an ordinary one were inside an ordinary SM, without any other state pointing out to it. On the other hand, a Sequence presents its connector outcome that gathers one after each other in a row the added states or state machines.

4.3.3 Defined States

Introduction

Once known the next step of the mission, the "add sm from CSV" function must add an SM or a single state to the mission sequence. Here are described all the possibilities implemented and Table 4.9 summarises them.

The architecture of the SM is defined inside the piece of code dedicated to each kind of step. However, the callback definitions for the different states are situated apart as different kinds of state machines employ them. Those state definitions will be detailed in the next section.

Table 4.11 Built-in states suitable to fulfil the defined mission state machine and its purpose.

State	Description	Type
New world	Definition and spawning of scenario obstacles.	CallbackState.
Spawn UAV	Spawning of mobile agents.	CallbackState.
Wait	Asleep while specified time or user dashboard input.	CallbackState.
Set parameter	Change internal parameter such as algorithms used.	SimpleActionState.
Take off	Lift off a single UAV.	SimpleActionState.
Save CSV	Storage of simulation data.	SimpleActionState.
Land	Descend to land.	SimpleActionState.
Basic move	One axis movement instruction for a drone.	SimpleActionState.
Follow path	Navigate to a sequence of waypoints.	SimpleActionState.
Follow UAV at distance	Navigate to a UAV up to a distance	SimpleActionState.
Follow UAV at position	Navigate up to a position relative to a UAV.	SimpleActionState.

New World

Given the fact that only one process is required to define and locate every volume and every UAV, New World is a single state. As it is the first one defined and added to the SM, SMACH takes it as the initial state. Due to its simplicity, the execution of spawning and model uploading would have been included along with this state. However, that option is discarded to provide versatility if new worlds would be defined or spawned separately later in the same mission.

No inputs are assigned as this is a fixed process and every required variable has already been defined as hyperparameter.

Spawn UAV

Once the information of a new world is thoroughly established, agents are brought to reality by this state. The reason why those two states must always be at the beginning is that, without UAVs and its nodes, GS would be pointless without agents to harmonise. Similarly, it is a predefined process that does not require any other definition as input or user data.

Wait

Sometimes, it is required to let the system recover from a computational requiring process. As well, it would be useful to the mission for the user to be aware of the situation until the next step or forever. Whatever its utility, those single steps are defined as the previous ones as state callbacks. The state type is provided as parameter differing between waiting for a specified time or a user dashboard input.

Take off

A ROS action state accomplishes the lifting off task. Due to the fact that the drone is prepositioned on a specified point, only the desired height must be described in this step. The state will end once that height has been reached

Land

Similarly to Take-off, Land is a ROS action state. However, as now it is not required a height, no parameter is given for the definition of the goal. Furthermore, in the future would be added options such as a point in the horizontal land plane where to land.

Save CSV

This one is the state to dump into a memory file the whole data frame structure that has been fulfilled during the whole mission. It is an internal process, so nothing respecting to ROS occurs.

Basic move

Keeping in mind the utility of a basic move, it seems reasonable the use of a simple structure such a standalone state. A pure callback state would accomplish this, but an action state provides all the specification that this issue requires. The goal formation of that action would receive a dictionary characterising the basic move and the identification number of the agent. That information is transformed into the action message.

Follow path

As its name shows, the flight plan for this state is fixed from the beginning, and every parameter is determined. Once the concerned UAV is defined, two more parameters are required. The first one is a list containing the direction on the path inside the world structure. I.e., the volume, geometry, pose set and any other explicit specification. With that, the path is completely defined, but there is one more parameter about the performance. The nodes that smooth the path may be tunned to define the order of the generated spline. Thus, this field defines that parameter as well as if the smoothing is required.

All those parameters are the ones that fulfil the action message sent to de UAV.

Follow UAV at distance or position

Similarly to the Follow path function, the definition of this state receives the parameters from the JSON such as the following type, the distance/relative position and the identity of the pursuit UAV. It is also defined a period of time in which the task is performed.

4.3.4 Example of state machine structures for missions

Introduction

In this section are exposed some examples of complex structures suitable to be part of a mission. Table 4.12 provides a list of every structure defined up to now. A selection of three of them has been selected to explain its maximum potential.

Table 4.12 Complex structures created as associations of the above states and different types of state machines. Them may correspond to one, two or many UAVs on the same structure.

Step	Description
Spawn UAVs	Query and spawning of mobile agents.
All take off	Concurrence to start to fly at the same time.
Follow paths sbys	Each agent receives a list of waypoints and accomplishes it point by point together.
Queue of followers AD	First agent follows a path and the rest follow another one at a distance.
Queue of followers AP	First agent follows a path and the rest follow another one at a position.
All save CSV	Individual storage of simulation data.
All land	Concurrence to descend to land at the same time.
Battery recharge	Follow path to home, descend, wait and vice versa.
Safe stop and descent	Hover during a short time and land.

All take off

The complexity of this structure is quite a bit higher. A concurrence state machine is selected as the leading network. In other words, different states are initial, and there is more than one active at the same time. Each of the states corresponds to the take-off of an agent as described in Code 4.2 and Figure 4.4. The type of state used for that task is a Simple Mission State as concerns directly to a movement accomplished by a ROS action server on each UAV-M.

First of all, as different outputs will occur from each of the active states, an outcome dictionary is required. This dictionary points out each general concurrence output with each joint possibility of single states outputs. In this case, all UAVs have assigned the success output to set the whole mission step as succeeded.

For each UAV there is appended an instance in the outcome dictionary as well as a new state in the occurrence. That fact provides the independence of the number of UAVs present on the mission. In advance, they must receive a new name for every state to avoid identification conflicts. Consequently, each one receives the identification number of its UAV as the first part of the name.

Finally, inputs for the definition of the ROS action are provided and are suitable for take-off height.

Code 4.2 All Take-off, Land and Save mission steps hierarchy.

```
Global mission State Machine
  - Other mission steps -
  (All Take-off / Land / Save) Concurrence
    UAV 1 SimpleActionState
    UAV 2 SimpleActionState
    UAV 3 SimpleActionState
    - Other UAVs -
  - Other mission steps -
```

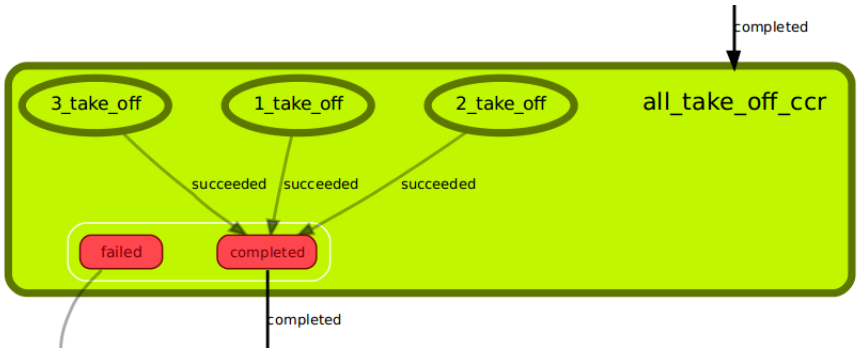

Figure 4.4 Visualization of an "all take-off" Concurrence. At the same level and at the same time, a state is concurrenced for as many as UAVs as on scene.

Follow paths sbys

Follow Path Step By Step constitutes along with Follow UAV AtDistance/Position the most complex type of hierarchy implemented. No new concept is introduced, but all the known ones are nested.

The aim is to introduce a singular concurrence for every new waypoint each UAV takes of its path. As many concurrencies as long the waypoint path is, are concatenated. Hence, the behaviour of waiting for other agents to get to its goal until it gets to the next command is achieved.

The outcomes dictionary must be defined for every occurrence but it would be reused as the performance must be the same. Furthermore, the only actual difference with structures above is the reading of that path length and adding one more occurrence for each waypoint. Moreover, the outcomes dictionary that links the last concurrency output and the state machine that contains the whole sequence of concurrences also needs its outcome mapping. This structure is well described in Code 4.3 and Figure 4.5.

Regarding the single steps, those are defined as Simple Action states. It is because following a path is associated with an action in the UAV-M. For each one of the states, the identification of the UAV and the number of waypoints must be provided. The next goal to adjudicate is obtained by hyperparameters definition.

Code 4.3 Follow Paths SbyS step hierarchy.

```
Global mission State Machine
  - Other mission steps -
  Follow Paths SbyS State Machine
    Waypoint 1 of paths Concurrence
      UAV1_WP1 - Follow Path (1) SimpleActionState
      UAV2_WP1 - Follow Path (1) SimpleActionState
```

```
      UAV3_WP1 - Follow Path (1) SimpleActionState
    Waypoint 2 of paths Concurrence
      UAV1_WP2 - Follow Path (2) SimpleActionState
      UAV2_WP2 - Follow Path (2) SimpleActionState
      UAV3_WP2 - Follow Path (2) SimpleActionState
    - Other waypoints on path -
  - Other mission steps -
```

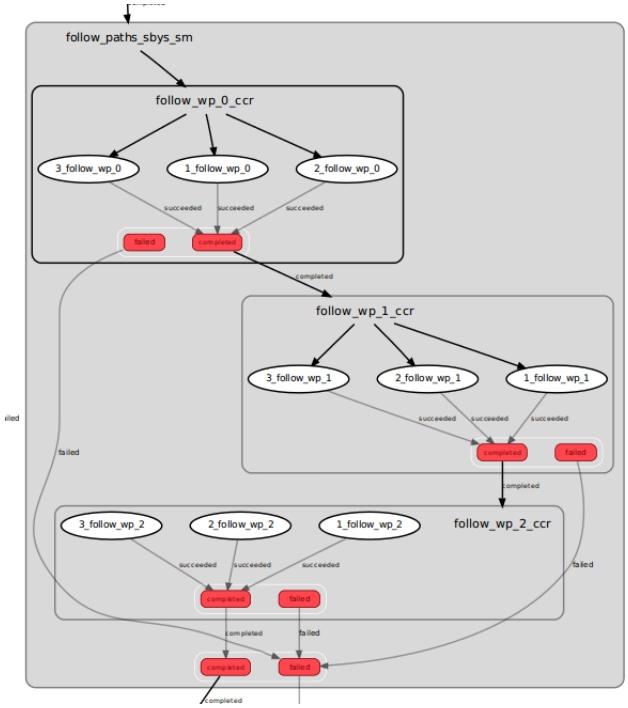

Figure 4.5 Visualization of Follow paths sbys mission step. For every new waypoint of the paths, a concurrence SM is created with a state for every UAV inside. Later, every new waypoint concurrencies are concatenated to shape the whole path.

Queue of followers AD / AP

Whereas in the last mission step, every agent waited for the others to accomplish every new waypoint target, now the first drone executes its whole path. At the same time, a second UAV follows him at a certain distance or a specific position. Moreover, so does the third pursuing to the second. In consequence, they constitute a queue of followers.

For that first drone, a pure action state is enough to make it accomplish its task. As explained before, its ID is passed, and he gets the path to follow from hyperparameters. For the rest of the drones, its identity number, as well as the ID of the followed agent, needs to be provided. With that information, another simple action service is implemented to deal with that kind of roles. A more accessible overview of the offered structure is shown in Code 4.4 and Figure 4.6

There exist one simple action state for both At Distance (AD) and At Position (AP) behaviours. The similarities among them, let this explanation be the same for both although that difference of behaviour must be taken into account. In one hand, for ADs, the distance must be provided to the goal. On the other hand, for APs, a position composed by a bias of [x,y,z] is required to determine the translation of the objective from the real target position.

Code 4.4 Queue of Followers AD/AP step hierarchy.

```
Global mission State Machine
  - Other mission steps -
  Follow Paths SbyS State Machine
```

```
    UAV1 - Follow Path SimpleActionState
    UAV2 - Follow UAV AD/AP (1) SimpleActionState
    UAV3 - Follow UAV AD/AP (2) SimpleActionState
    - Other UAVs-
  - Other mission steps -
```

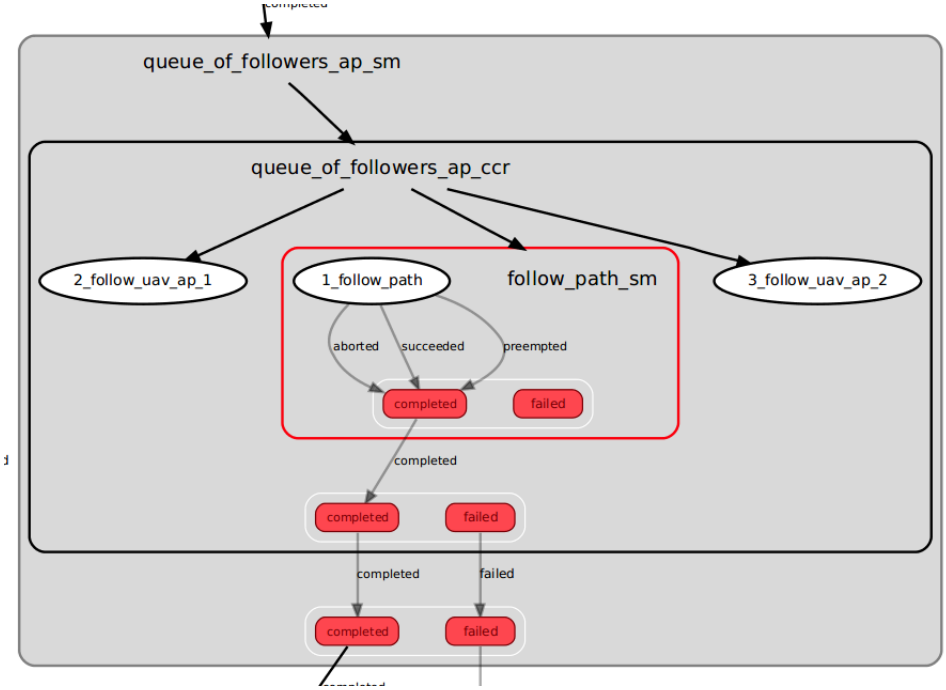

Figure 4.6 Visualization of Follow UAV At Position mission step. While UAV 1 follows an ordinary path, UAV 2 follows it and UAV 3 follows UAV 2.

Battery recharge

On a long mission, it would happen that a UAV runs out of battery at a performance state. Here presented structure shows the workflow to follow to take the drone back home, wait to be recharged and come again to the point of the mission it was at the critical time. A sequence of Follow path, land, wait and take-off states, already presented, is followed to achieve this objective. At any time, collision or critical events are considered and redirected to its respective outer state machine. A more accessible overview of the offered structure is shown in Figure 4.7.

Safe stop and descent

Finally, the safety implementation that should present any SM for a real simulation is accomplished in this section. Note that every state on the image corresponds to a concurrence, so every drone engages the actions. The image shows an overview of the full state machine. Once on the central state of the mission in which every agent is on its task, if any critical event, collision or low battery on the final step occurs, every drone is relocated on a hovering state so that any collision is avoided. Once stopped, the SM awaits for the pilot to take control. In the case that does not happen within a short time, every UAV lands wherever it is over, saving the data of the mission. A more accessible overview of the offered structure is shown in Figure 4.8

4.4 Worlds

4.4.1 Motivation

The primary goal of this module is the creation of virtual reality. That virtual reality contains every piece of information about the real or simulated world that the UAVs and the GS are going to be provided with. For the simulation case, Gazebo offers a great variety of different scenography in favour of making the virtual

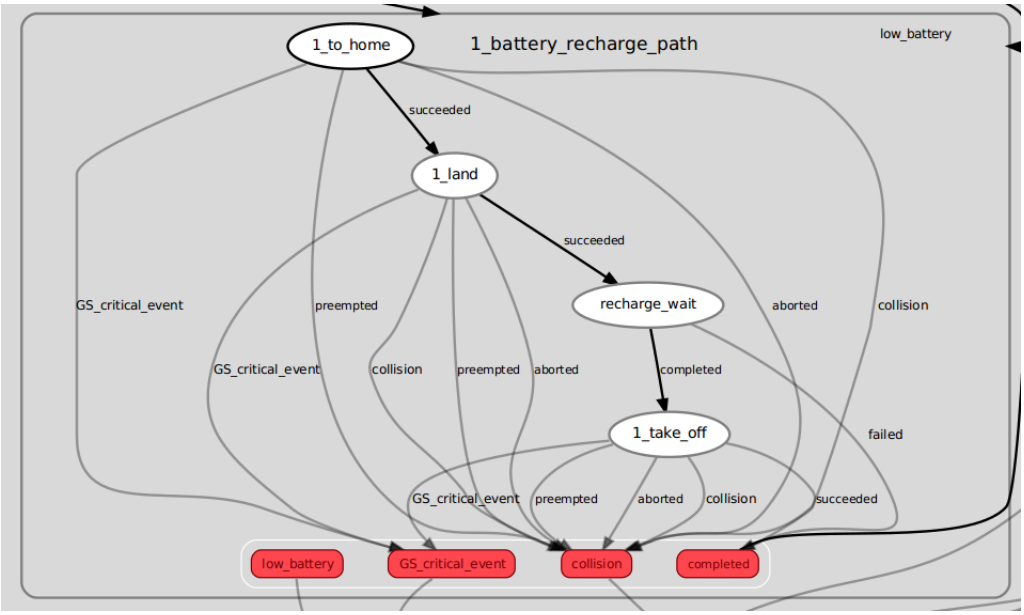

Figure 4.7 Visualization of Battery Recharge mission step. Once entered on this state (top row of the image), a sequence of states take the UAV to home, land, recharge wait and finally take off, and the mission comes to the prior state.

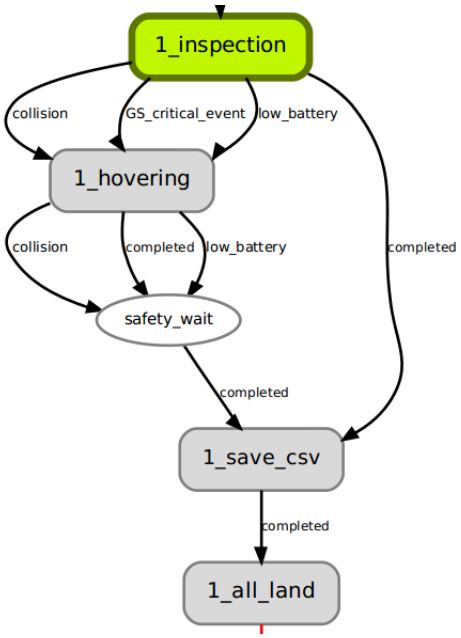

Figure 4.8 Visualization of Safe Stop mission step. Every box on the image represents a deeper state machine inside. When the inspection returns a critical outcome, a hovering state followed by a wait are activated so the pilot can take control safely. If there have been no changes, data is stored and all drones land.

world as similar as possible to the real one. That entails the difficulty of positioning and modelling every object at its minimum detail. In a task like the goal of this project, the excessive specification would mean a useless complication. Hence, minimalistic objects such as boxes, cylinders, spheres or prisms are going to complete the range of obstacles as a more sophisticated description would only require an assembly of these original shapes.

The scenario where the play is going to take place is initially an empty 3D space so volumes, geometries and models can be placed wherever it is required. They may have any size and any orientation as long as

they leave free space for the UAVs to fly around. Therefore, thinking wisely on the layout of the scenario is essential for a mission to have possibilities of success or even for an algorithm to have troubles enough to prove its value. Therefore, this module is a useful tool to modify it easily.

4.4.2 Hierarchy of classes

The way to model the world is depicted in Figure 4.9. It consists of the decentralisation of single volumes over the three dimensions. Those volumes are defined by several geometries within and each geometry, by different kind of elements. The advantage of this approach resides in the use of the global position of the volume to place locally each of the geometries it presents inside, wherever the volume was placed. Furthermore, the objects inside each geometry are locally placed as well concerning the position of its parent geometry. Initially, the idea would seem quite complicated but gets clarified as the cascaded architecture is decomposed.

Digging more in-depth on the hierarchy, spatial points from each of the geometries would be extracted in diverse manners. Those are achieved by a poses set such as 3D equidistant matrix, by extraction of the edges of the shape, by defining logical paths such as zigzag on a plane or directly selecting the points by their coordinates. Once those spatial points and its orientation are defined, each one is concerned to a Free Space Point (FSP) or an obstacle is placed on it. An FSP is suitable to be selected later as part of larger flight plans or inputs for an algorithm. For its part, obstacles are spawned on the simulator.

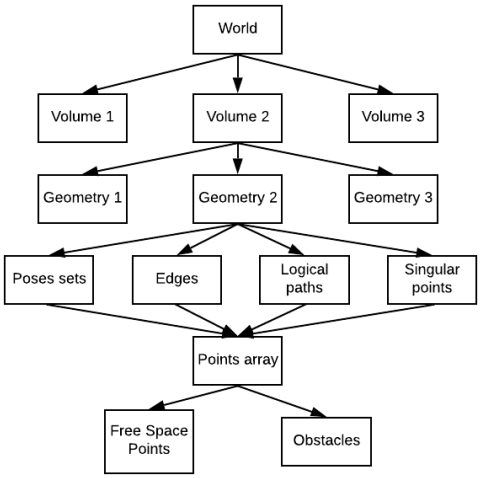

Figure 4.9 Worlds modelling diagram. It is a hierarchy structure scaded in on volumes, geometries, poses sets, and free space points or obstacles.

Implementing the exposed structure of the definition of the World, an example has been invented in order to clarify the concept in Figure 4.10. The world frame is on the left of the image, from which two reference frames of volumes are defined. Around them, several geometries of different shapes are referenced. Into the geometries, free space poses (FSP) and obstacles are placed randomly, within logical paths such as zigzag, by coordinates or on the vertexes.

That modelisation seems to solve the problem of environmental understanding. However, who and how creates it is to come in the next paragraphs. The definition of a world is expected to be created inside a JSON file. A JSON file would be described as nested lists and dictionaries fulfilled by basic value types. Making use of such a powerful tool, the whole information of a world can be stored based on specifications defined for the designations of this project.

Once a world file has been chosen, those rules are followed by the Worlds module taking, as a result, the same architecture built up by python objects instead. Each one of them is responsible for one element on the specified hierarchy, and it must manage its information during the entire mission.

Worlds

On top of the pyramidal structure is found the Worlds class from which the rest of elements are expanded or nested. This class is not concerned with any specific 3D object but is thought to gather all of its child ones.

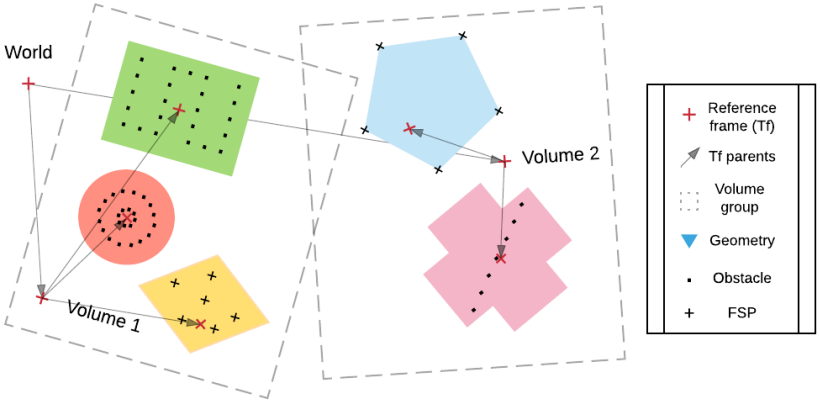

Figure 4.10 World frame on the left. Two reference frames of volumes defined from world. Around them, several geometries of different shapes. Into the geometries, free space poses (FSP) and obstacles are placed randomly, within logical paths such as zigzag, by coordinates or on the vertexes..

Initially, an ID number is assigned to the world so it would be possible to work with plenty of them. As always, the hyperparameter retrievement is accomplished at the beginning from ROS params. Amongst those hyperparameters, the name of the scenario and the home path have been read, so the JSON file is determined and accessible to be uploaded. It is now time to start translating from JSON information to the python objects.

Reading throw the volumes list, an object is created from each one receiving its definition to continue the nesting. A list of all the obstacles of the worlds is fulfilled to make them accessible all at once. Once all volumes have been created, the hyperparameters are actualised containing the new information generated. For positioning, it is employed the Transforms library (TF) broadly used in ROS.

One last capacity of this class is the inclusion of the possibility to retrieve FSPs from a specified volume, geometry and kind of point generation as explained in the architecture.

Volumes

The purpose of this class is to localise, build and gather different geometries as long as giving some standard features to all of them.

Name, prefix, permits and the origin are recovered from the input definition as defined in Table 4.13. The list containing the already created transforms is also saved. To define the origin in ROS information so the rest of 3D elements may be referred to it, a transform is created on the specified position. As before, the list of geometries is followed and an object of its specified shape is appended to the list of geometries, providing it with its definition. The transformations of those children are stored on its list and so is done with the transform.

This class also presents functionalities to externally get the list of transforms and a geometry specified by name. Besides, a single waypoint specified by its coordinates may also be requested.

Table 4.13 Volume class parameters that define the main features to place and differentiate it along with the child elements.

Parameter	Description
Name	Unique identifier of the volume
Prefix	Short unique identifier of the volume
Permits	Features about capacities of the volume
Origin	Position and orientation of the local frame referred to the global frame
Geometries	List of child geometries that define the volume

Generic Geometry

This Generic Geometry class is a heritage for the specific geometry classes that are to be discussed in the next section. Therefore, it must be thought of as a range of standard tools shared by every different geometry. As a consequence, the functions it presents would seem initially unconnected, but the whole chain that characterises the Geometry capabilities is the conjunction of Generic and any Specific Geometry objects.

Amongst those common tools, there is the definition of general parameters of both the class itself and its Rviz Marker associated. Those parameters are defined in the geometry definition provided that can be found in Table 4.14. Hence, the first step of the initialisation is to read those parameters and create their own ones. Specifically for name and prefix, the prefix of its parent is appended at the beginning of the string to maintain the hierarchy nomenclature. Secondly, the associated TF transformation is spawned so to reference the rest of the elements of the geometry to it. Thirdly, the RViz marker object is created to improve its visualisation and understanding.

On the last part of the initialisation, the poses sets selected in the definition are created. Thus, the single definition of every one of them is followed to create it using the rest of the functions. For its part, the definition of each pose set may be found on Table 4.15.

Table 4.14 Generic geometry class definition. Parameters that define the main features of any type of geometries independently of its shape.

Parameter	Description
Name	Unique identifier of the geometry
Prefix	Short unique identifier of the geometry
Shape	Basic geometry type
Origin	Position and orientation of the geometry frame referred to the local volume frame
Dimensions	Characteristic size values
Color and Alpha	RGB and opacity of the RViz marker
Pose sets	List of poses sets to perform inside de the geometry

Table 4.15 Specific geometries dimensions.

Specific geometries dimensions. Parameters that define the main features that depend on the shape and according to the type of pose

Several functions are offered to get the value of different elements such as one obstacle of the list or the transformation. The list is so extensive that only the most important and influent ones are explained here. The whole list of functions can be observed in Table 4.16.

The RawDensityMatrix function provides a mathematical 3D matrix of the dimensions of the selected poses set. That 3D matrix is fulfilled with True or False values on each of the elements distributed on a randomly depending on its density. Another function called GenerateObstacleFromDensityMatrix receives a boolean 3D matrix, and a poses 3D matrix and mixes them to return a list of objects placed on each pose whose value on the boolean matrix presents a True. GenerateObstacleFromPosesList presents a very similar functionality, this time, for every element of a provided list of poses is now created an obstacle associated. As it has been done for obstacles, similar functions make the same task but return Free Space Poses instead. To make use of every function described so far in this paragraph, the function GeneratePosesSetMatrix reads the definition of the poses set and selects which function is appropriate for each specification.

Apart from matrices, GenerateRandomPoses asks for a pose in any random place inside the geometry and locates on it an obstacle or an FPS as specified on the definition of the poses set.

Table 4.16 Functions used by the Geometries classes to define its own parameters by translating the information contained on the JSON file. It is specified the type of class that offers the function.

Function	Provider class
Make Marker	Generic
Erase Marker	Generic
Get Transforms	Generic
Get Obstacles	Generic
Get FS Global Pose from Matrix/List/Path	Generic
Raw Density Matrix	Generic
Generate Obstacle/FSP from Density Matrix/List	Generic
Generate Poses sets Matrix	Generic
Generate Random Poses	Generic
Generate Random Dimensional Values	Generic
Pose From Array	Generic
Generate ZigZag	Generic
Poses Density Matrix	Specific
Random Poses	Specific

Specific Geometries

The geometries so far defined comprise a cube, a sphere, a cylinder, a torus and a prism. Inside each one of those classes, the dimension of the RViz marker is defined according to the specifications of each shape. Each class defines its dimensions according to the characteristics of the volume it represents. Therefore, each element inside the dimension field of the definition represents a real dimension of the geometry as depicted in Table 4.17. There is no other difference in the initialisation step.

The main reason to create different classes resides on PosesDensityMatrix and RandomPoses functions. The first one divides each of the three dimensions that define de geometry into similar segments. At the end of each segment, along with every dimension, a pose is placed, so those poses are equitably distributed. The RandomPoses function makes use of the above-explained dimensions to randomly provide a value for one of them between the respective limits that define the geometry.

The prism class has a sort of auxiliary functions provided its more complex geometry. Using the library Sympy, geometrical elements -such as interactions with polygon or segments- are computed to extract the inclusion or limits of its shape.

Table 4.17 Specific geometries shapes currently implemented and its main dimensions.

Shape	Dimensions
Cube	Cartesian lengths
Sphere	Cartesian radiuses length
Cylinder	Cartesian base lengths and height
Prism	Height and vertex coordinates

Free Space Pose

The Free Space Pose is an available position of the space that can be accessed for different purposes. The interface to operate with it is this class.

Initially, the parameters that define the identity of the FSP are obtained from the poses set definition, but the name needs to be assembled from the prefix of the parent, the "pos" tag and its identity number. A TF broadcaster locates the frame for the object so it can be referenced globally.

Two functions for getting the transform and the global pose are offered. These functions are beneficial to relate frames in the sequent steps. Hence, they are called from their parent classes.

Obstacle

This class is focused on managing the information of a single obstacle and spawning it if required. Thus, everything concerned with that obstacle deals with this class or with information generated by it.

The parameters that define the obstacle are provided in the poses set definition and are saved to the class at the beginning of the initialisation. To define the name, the prefix, the tag "obs" and the identification number are concatenated. The path where the XML files of the basic shapes are stores is retrieved from ROS hyperparameters and built up. Depending on what shape has been selected, its concerning file XML file is rewritten to dynamically define its dimensions. That is a way of dealing with the fixed nature of XML definitions.

The definition of the own marker is also accomplished, and the services for spawning and deleting the GAZEBO model are created. A TF broadcaster locates the frame for the object so it can be referenced globally. Once that information has been obtained, the object is spawned on the simulation, and the RViz marker is also built.

As specified for the Geometry classes, some auxiliary functions are provided to solve explicit requirements such as erasing and spawning the model, get transform and global pose or make de RViz marker.

4.4.3 Auxiliar classes

Transformation Broadcaster

This class is focused on managing, during the whole mission, every action that should be accomplished about one single specific TF.

First of all, the message to be broadcasted starts its building by defining the header. Secondly, the pose is also introduced on it depending on how it is defined whether a list or a pose. In case it would be a list, it would come from a matrix or an array, so its length is checked to decide which way of reading to choose. Finally, the new pose is appended to the list with every pose of the volume.

This class offers two functionalities for its operability. The Broadcast function directly sends the message containing every TF so far defined and another function to retrieve that TF list.

Rviz Marker

An RViz marker is a simple visualisation of a real object for better comprehension, so this class only needs to interact with ROS for its information to be depicted. Once it has been spawned, it can be actualised, so dynamic interaction would be required. That interaction is the primary duty of this class.

About the initialisation, the definition of the marker and the hyperparameters are retrieved from its known sources. That information is translated to the message that defines the marker inside its corresponding field. Once the message has been fulfilled, it is spawned.

Additionally, a function to actualise the pose and another one to erase it is offered. Each one is focused on modifying the original message according to the specifications and publishing it.

Rviz PolygonArray

As well as RViz Marker, this function is in charge of creating a visualisation. In contrast with a marker, the polygon array is used to show the dimensions of a prism. For every lateral and base face, a new polygon is added to the list to deal with a unique visualisation. The functionalities are very similar to the marker as they deploy the same utility of the chain.

4.4.4 Built Worlds

Some examples of worlds used in the different projects are shown in Figure 4.11, Figure 4.12, Figure 4.13 and Figure 4.14.

4.5 UAV Manager

4.5.1 Motivation

None of the nodes described so far would have sense without the Unmanned Aerial Vehicle Manager node, and this section explains its central core. The task of this node is to focus on a drone and manage all its information and movements. This node is equipped with all the communications with ROS to receive

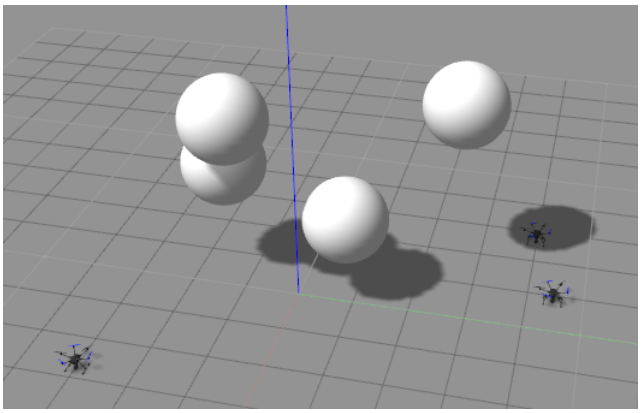

Figure 4.11 World 1: Intersection. On the central cylinder, some obstacles are randomly placed and the UAVs must cross throughout it in order to test collision avoidance.

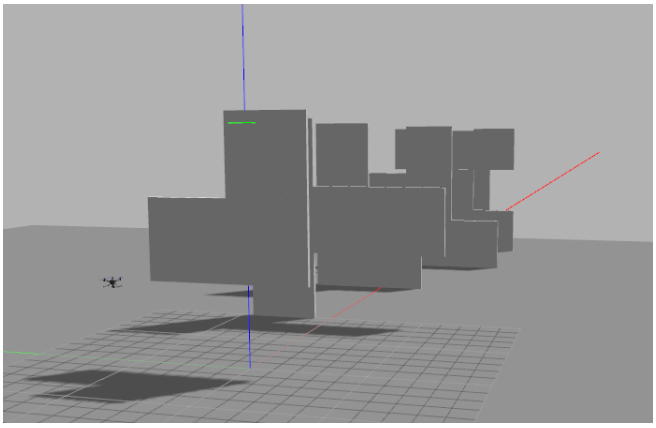

Figure 4.12 World 2: Tube. The sequence of walls with randomly placed bricks for a UAV to cross it using vision.

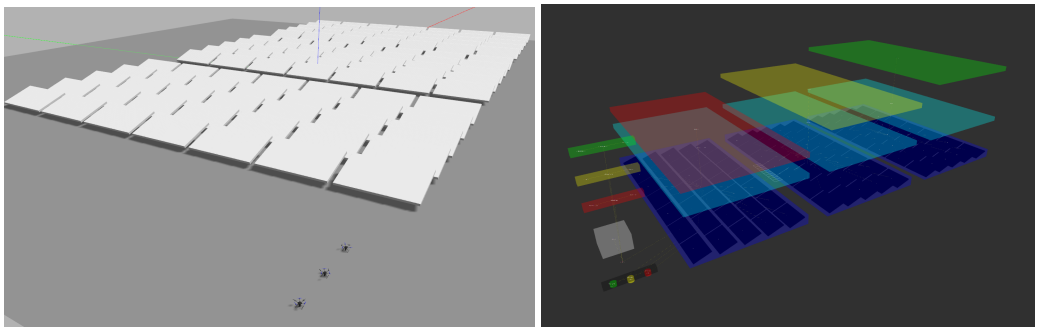

Figure 4.13 World 3: Solar plant. Three different zones with solar panels on the lower level, the inspection level and the third level represent the approximation height. On the inspection polygon, a zigzag path is employed for area seeping..

information from its sensors and the state of other drones. Also, it can talk with UAL to send it instructions of actions.

The decision making, it means, who decides what to do at every moment depending on the state of the mission is Ground Station as was previously seen. That is informed via ROS actions advertised by this node and is, thus, its duty to accomplish them as accurately as possible. The measurement of the performance and the saving of the whole mission data is as well its duty, and a task performed inherently by the node. All the instruments that surround this core of the node are modelled in the diagram Figure 4.15.

In this section, different utilities from inside the UAV Manager are to be defined such as how does it

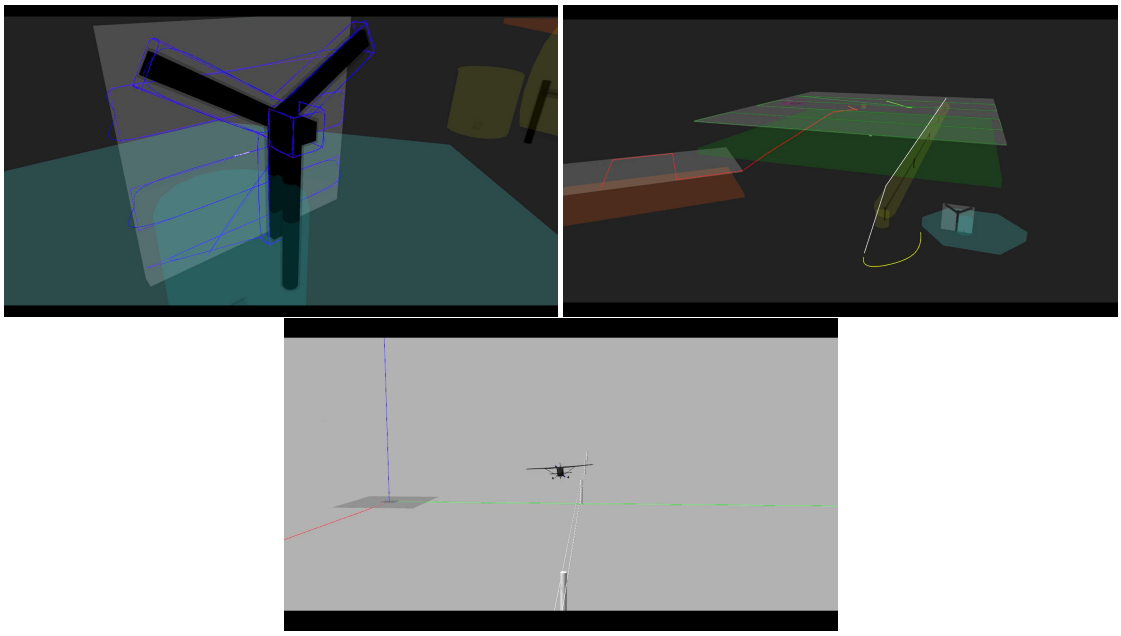

Figure 4.14 World 4: GAUSS Project Use Case 1. This project presents a challenge of different applications of UAVs. Between them, it can be found three areas sweeping (top right), linear surveillance and a full wing turbine inspection(top left). A simulation in GAZEBO is shown on the image below.

accomplish the different roles. Other functions like gathering information from others and own drone, the decision of the optimal velocity to impose or the State Machine that rules on the drone, are implemented in other modules that will be exposed later in its section.

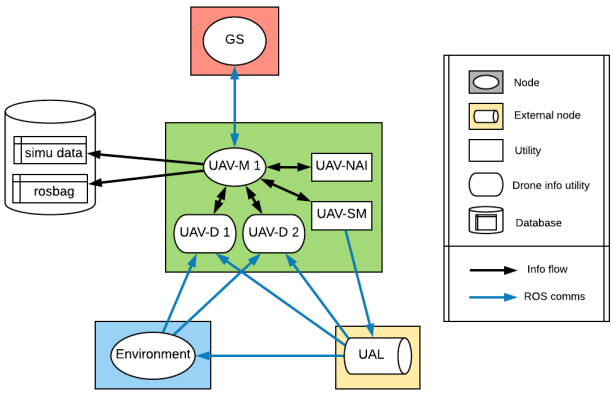

Figure 4.15 UAV-M node relations diagram. ROS communications and maintained with GS, UAL and Environment. Internal flow is accomplished between the modules of the node and external data exchange with the database.

4.5.2 UAV Manager functions

As it was explained in the complete architecture overview, not all of the functions are executed from inside but is the SM that, commanded by the Ground Station, comes to this central function to activate the behaviours. Those behaviours are controlled and executed from this central module talking to the UAV corresponding interface while taking into account the information about external entities received from the rest of the modules.

First of all, some initialisations are defined to get an extensive survey of what information flow is this node dealing with. Consequently, the rest of the possibilities offered by the GS module are detailed to understand the advantages provided.

4.5.3 Initializations

Initially, the individual identification provided at the initialisation as well as the retrieve of all the hyperparameters is effectuated to get a global consciousness of the operation. Along with those global parameters, it is created the object that will manage the velocity that the UAV is adapting every instant. That is called the Navigation Algorithm Interface (NAI) and is the subject of another module.

Now is the turn of value initialisations for a group of variables that need an initial state. That is the case of the data frames that collect the whole mission information. Furthermore, the distance to the goal is set far away, the time pursuing the goal is started for the timer and the goal itself is also initialised empty. At the end of this group, can be found the initialisations of two timers for elapsed time between data storing and to check elapsed time in the calculation of velocity each instant.

It is now time to wait some time until the computer recovers from the spawning of this UAV model and nodes. Hence, two periods of sleep are programmed. On the one hand, every drone awaits the same time depending on the total number of drone. In another side, its ID will determine another additional sleeping time, so they get progressively advertised to receive orders. Another group of single valuing is executed to assign a first state of the drone to "landed", to select that at the beginning a collision has not occurred and that a command to die did not arrive.

At this point in the initialisation process, every crucial information has been obtained or defined so the node is started and the ROS communications shall start from now on.

After that spawning, two lists of the objects that gather the information of each UAV must be created. The first one consists of the objects that deal with the initial configuration (UAV-C) such as the model, the second one, copes with de data (UAV-D) received dynamically along with the whole mission. As a consequence, the two lists have a length equal to the number of total drones that take part in the mission. One element of each one is concerned with the same UAV assigned to this UAV-M, here called "main" UAV. In this basis, the ID number of its assigned drone and the ID of the main drone are given while creating every UAV-D object. Similarly, as the NAI module, UAV-D and UAV-C modules will be explained in detail in its section.

Finally, the last part of the initialisation of a UAV-M node corresponds with the start of listeners, later explained. The last action accomplished is the definition of the state machine and its execution.

Listeners and publishers

As long as it concerns this central module, the listening part is reduced to the communication with the GS to send its command to finish the simulation. The action that GS will execute for that aim is merely to complete its activity and end the node.

Regarding now to the publishers, this time four publishers are defined. Gathered in pairs, the first one publishes to RViz the raw waypoint path as selected for the goal and the smoothed path in the case. The second pair cope with the commanded position or velocity, depending on the actual control type, so it talks directly to UAL.

4.5.4 Commanders

A sort of commands to the autopilot is at disposal thanks to UAL. The GS commander functions handle the interaction with the respective UAL functions, adapting them to its requirements. Commanders will not be directly executed but constitute a utility that will later be used by the different roles. A list with every commander function and its particular parameters associated is offered in Table 4.18.

Table 4.18 UAV-M commander functions to accomplish the different tasks and the parameters that define it.

Commander function	Parameters
Land	None
Take off	Height
Go to waypoint	Target Pose
Set velocity	Target Pose, Hover
Set home	None

Land command

The Land command is the simplest one as it only needs the blocking parameter of its UAL service associated. Hence, the rest of the settings to talk to the correct service are the UAV ID name that was got in the definition hyperparameters.

Take-off command

The only difference that Take-off command offers with respect to Land is that it needs a height where to stop once on air. That is passed as a single integer and directly introduced into the message of the service request to UAL without any transformation.

Go to Waypoint command

As its name explains, UAL affords itself the task of taking the UAV to a position and orientation specified. To accomplish that, it only requires the Pose object that defines the target. To double the security of getting to the target and also to have the possibility to decide the radius of acceptance, this function implements a second control above UAL. Once the service request has been sent, this control consists of keep controlling the distance to the goal employing another utility function. Once that distance is lower than the acceptance radius, the drone is supposed to be on its goal position.

Set velocity command

This last type of commander is the one selected to make use of the NAI as it always defines the velocity to apply. Consequently, it is the most peculiar one given the fact that it is the one ever used for the performance of the roles. As the goal it takes is to be defined separately by each one of the roles, this is not passed as an argument. However, that goal is a variable of the class already owned by this function.

Regarding the hover option, if it is activated, the Hover functionality of the NAI is called. That function returns a velocity twist with all members equal to zero.

If on the contrary, hover flag is deactivated, the core of this velocity command is used. The Guidance function of NAI object is called giving to it the list of all UAV-Ds information and the target position. As the return, the selected velocity twist is obtained. It is now time to check if data should be saved. Depending on the time elapsed since the last storage, the state of the UAV and the role, it will be kept or not. For the path follower role, it will only save the data when it is going to a new waypoint but will not if it is hovering. Instead, the rest of the roles are constantly saving.

Finally, the velocity twist resulting from the two options above is passed as the argument to the client of the UAL service to set the new velocity.

4.5.5 Roles

The different roles are in a layer above the command functions. They are focused on generating a target position, so the velocity command does the work to get to it. Furthermore, one of its adjacent tasks is to evaluate the performance of the role accomplishment. A list with the roles implemented and their inputs are presented in Table 4.19.

Table 4.19 Roles that the UAV may adopt during a state of mission and the inputs required to define it.

Role	Inputs
Path Follower	List of waypoints
Follow UAV At Distance	Target ID, Distance
Follow UAV At Position	Target ID, Relative position
Basic Move	Move type, Dynamic, Direction, Value

Path Follower

The implementation of this role is split in the accomplishment of different targets. The initialisations required every time a new path is followed, consisting of the publication of it for RViz and to contact with the path smoother nodes. For this last step, if the smooth path mode is different to zero, the original target path is sent to one of the nodes employing a service. The response consists of a new splined path. That new path is again

published to RViz and also sent along with the smooth mode to the node that deals with the smooth velocity constant transmission.

The very next step is to iterate between every single waypoint in the path. Hence, the rest of this subsection is about one single target that is changed at its termination.

The current target is retrieved from the path list of goal Poses by its position on it, and an additional variable is equally defined to be part of the storage collection. After that, the state of the UAV is changed to advertise the movement and the number of the target. This change is also notified to the Ground Station. That new target position is also broadcasted as static transform by the already explained function dedicated to it.

Once informed, the central process of deciding the new velocity starts entering on a loop until the distance to the goal provided by the utility function is lower than the accepted radio on target. Inside it, an evaluation is preliminarily done after the calling to Set Velocity commander function deactivating the hovering. As the goal and the role were defined above, NAI already knows what specifications should be followed.

Once the drone is on its objective, the Set Velocity function is called again but, this time, with hovering flag active to leave the drone holding at its target. A new state is set to inform that the drone is currently on its target and its state in the GS is also actualised. Finally, further evaluation is done given the fact that another drone would still be moving.

This process is repeated as many times as the path is long. Hence, now another target waypoint would be picked up.

UAV Follower At Distance

Initially, the distance selected is saved in a global variable to be later saved with the rest of the instant data. The state is actualised to declare that a UAV is being followed, its identity and that the type is at a distance. As always, the GS is informed of that change.

The way of finishing the role execution will be a topic message from the GS. Therefore, the execution enters on a loop until that message is received. Inside it, a copy of the actual position of the target UAV is retrieved and selected as goal Pose. In case the solver algorithm was ORCA [48], the target point is forwarded in future by adding to it a piece of the velocity. Thus, a naïve prediction is accomplished, and the algorithm attempts somehow to deal with speed. Once the goal is completely defined, it is broadcasted as a static transform.

It is now the time to select the velocity. While the distance to the target is not lower than the defined target distance, Set velocity function is called, and an evaluation accomplished. Once the target is achieved, hovering is activated waiting for it to move away and start pursuing again. This behaviour is continuous until the GS sends the order to finish it.

UAV Follower At Position

This role is very similar to Follow UAV AD, so most of this information is repeated from its explanation. The reason why it is expressed again is in case the reader is only interested in this role, in order not to make him read both.

Initially, the distance selected is saved in a global variable to be later saved with the rest of the instant data. The state is actualised to declare that a UAV is being followed, its identity and that the type is at a distance. As always, the GS is informed of that change.

The way of finishing the role execution will be a topic message from the GS. Therefore, the execution enters on a loop until that message is received. Inside it, a copy of the actual position of the target UAV is retrieved and selected as goal Pose. To that new goal, a bias defined on the function inputs is added on the three axes to perform the translation over it. In case the solver algorithm was ORCA, the target point is forwarded in future by adding to it a portion of the velocity. Thus, a naïve prediction is accomplished, and the algorithm attempts somehow to deal with speed. Once the goal is completely defined, it is broadcasted as a static transform.

It is now the time to select the velocity. While de distance to the target is not lower than the defined target distance, Set velocity function is called, and an evaluation accomplished. Once the target is achieved, hovering is activated waiting for it to move away and start pursuing again. This behaviour is continuous until the GS sends the order to finish it.

Basic Move

The best way to understand the definition of the basic movement is at sight of Table 4.20 that explains the fields and the different possibilities that each one can be assigned to.

Table 4.20 Basic Move types and and the fields that caracterize it as well as the different options for each one.

Field	Options
Movement type	Takeoff, Land, Translation, Rotation
Dynamic	Position, Velocity
Direction	Up, Down, Left, Right, Forward, Backward
Value	Float

The first ascertainment is about the type of movement field. If it is assigned to a take-off movement, the value field should contain the desired height. In that case, the Take-off commander function is called, and that Height is given. However, if the type of movement is Land, its commander function associated is called without any other observation to the rest of the fields.

Another possibility of type would be a transition or a rotation. Those queries would carry more parameters associated. A change matrix for linear and angular changes is initialised within all its settings equal to zero. Afterwards, regarding the direction selected in its field, the value field will be mapped with a single member and its written inside the change matrix. This mapping is described inTable 4.21. Each member of the array has its positive and negative direction assigned.

Table 4.21 Basic Move directions mapping from translation and rotation words to axis.

Translation	Mapping
Axis X	[Forward, Backward]
Axis Y	[Right, Left]
Axis Z	[Up, Down]
Rotation	**Mapping**
Axis Y	[Up, Down]
Axis Z	[Right, Left]

Finally, the dynamic is checked. In one hand, it could be Position. In that case, the actual position of the drone is copied, and to each of its axes, its corresponding member of the changes matrix is added. The last step is calling the Path Follower role function, so it takes the drone to its destiny. On the other hand, if the dynamic selected is Velocity, a twist is directly created from the changes matrix, and it is passed to the Set velocity twist, so the drone adopts the desired twist.

4.5.6 Miscellaneous

It is now the time for a set of utilities that, although do not contribute to a single UAV movement, they are essential to the development of a mission.

Evaluator

Preliminary initialisations are taken into account to determine the minimum distance to other objects and agents as long as to state that at the beginning, no collisions have occurred against them. On the first instance, two lists within the distances are created and permanently actualised inside the UAV-D class, and are now retrieved. Those lists refer to the relative distances to the nearer UAVs and static obstacles with a lenght defined by hyperparameters. Consequently, it es checked the inclussion of the possition within the world limits defined on the world JOSN. The last test is about an eventual GS critical event advertisement.

Once that information has been gathered, each component of them is checked to be under the minimum distance previously determined. In the affirmative case in one single element of both lists, the global collision flag would be raised. The second part of the evaluation is related to the target. While the goal has not been increased, the distance to it and the time elapsed since it started being the objective are also calculated and stored.

All of those data are now gathered in a single evaluation dictionary. That is a better way to store it latter as a single data frame. As this information is about a single instant, it will be assembled later for the whole mission information.

Parse for CSV

Useful for the next utility, this function is focused on extracting the critical information of Pose and Twist. The way it effectuates this extraction is creating simple lists that can be easily stored and iterated. For example, from a Pose, the first component of the list is a list with [x,y,z] attributes of its position and the second, [x,y,z,w] components of its orientation.

Recursively, a list of UAVs would also be provided. As every UAV is expected to be composed by a Pose and a Twist, the function may call itself recurrently to develop a complete parsing of the whole list. As a result, this option would give a more elaborate list within the already explained single records.

Save data

The utility of this function is to collect all the relevant data of every instant and assemble a unique variable. The followed work frame is to fulfil single dictionaries containing all desired variables of a single moment. Once it is completed, it is appended at the top of the data frame heritage from the previous instants of the current mission. For complex objects different from lists of dictionaries, the function Parse 4 CSV, explained above, is employed.

Initially, the selected velocity for the current instant given the actual situation is parsed. Secondly comes the positional and dynamical information of every UAV including itself. This information is general to every UAV independently of its role. The rest of the choices concern to the goal and are suitable to vary depending on it.

For path followers, the pose of the next waypoint is parsed and its twist set to zero to complete the goal information. Concerning UAV followers At Distance, the goal is fulfilled with the position of the UAV pursued and the distance selected to follow it. Finally, about UAV followers At Position, the final position of the goal once the bias has been applied, is the chosen to be the goal data.

The last step for this data frame is its creation or ampliation with a new instant. Besides this data frame, another one is simultaneously created with the information received from the depth camera in case its flag would be raised. Its creation methodology is the same.

Store data

The collection of data from every instant is now stored in memory inside a single file. That data frame is already wholly defined. Therefore, only issues left are the definition of where to create the file. In the first place, the obstacles number definition is selected and formed amongst the two possibilities that come from the two ways of obstacles distributions. Secondly, given the wold simulation information, the path is built, and the data frame is stored on it. This same process would be followed to store the data frame concerning to depth camera besides the main one.

GS state actulization

This function is called to keep also informed the GS every time the current state of the UAV is updated, or a collision has taken part. Its only task is to receive that new state, wait for the service advertised by the GS to be available and send it along with a possible collision occurrence.

Geometricals

Lastly, a variety of small functions is supplied to get some geometrical information in a fast manner. First, three utilities concerning euclidean distances are found. These calculate the distance to the goal, a generic one between two Poses and the last one for the distance to an obstacle from world definition, expressed as a list.

It is also provided one more function that returns the actual value of the module of the velocity.

4.5.7 UAV-M State Machine

Motivation

On the previous section, all Ground Control functionalities have been described. Therefore, the reader has a clear comprehension of suitable actions and roles. But not about how all those events fit in a timeline. A mission is a set of the already known actions accomplished logically. The user has decided that logical way and the orchestra director is set to be the GS. Said the reader would ask himself why a new state machine for the drone is required.

The reason is simple: a drone cannot accomplish every instruction at any time independently what is currently doing. Control of the actual state must be performed to let new guidelines affect only in case it should be available. Another reason would be the division and determination between the different behaviours the UAV may afford. Those must obey a strict sequence of steps.

SM building system

As explained in the GS SM, the intrinsic work frame of SMACH will not be deeply explained in this document as it would be out of scope. But a clear view of its use is provided for a user to be completely capable of getting the best of it. The structure of the SM creation is similar in the way it defines, and later, the definitions of the different state machines are explained.

The basic structure would be visualised as a star-shaped state machine with a central state as the core and different Simple Action Server states situated on every corner. That structure shorted in Table 4.22.

Table 4.22 Grond Station State Machine states implemented that accomplish movements of the drone or more complex behaviours.

State	Definition
Action server advertiser	Publishes the servers that trigger the rest of states
Take-off	Start flying at a given height
Land	Get landed with a vertical descenst
Basic move	Single axis, dynamic movement
Follow path	Get to a sequence of points in the space
Follow UAV AD	Pursuit another agent at a given distance
Follow UAV AP	Pursuit another agent at a given translation
Save CSV	Store mission data in a file

Available states

The first and central state is called Action Server Advertiser (ASA) and is a pure Callback state whereas the rest are the Simple Action States. Consequently, the opening defined is the central one. About the rest, when they are created, not only its state is included on the overall state machine but it is also created a list with all of the Simple Action States. The reason to do this is to make then callable to start and stop advertising all at once by the central state.

Accordingly, no outcome from ASA is pointed to the rest. The transition to them is activated directly by a server request. On the contrary, when those Simple Action Service (SAS) are completed, this time the outcome dictionary always drives back to the central ASA. The definition of that ASA dictionary and, as a consequence, the inclusion of them on the SM is the same. Hence, the following explanations are focused directly on either simple or action callback.

Not all of them are described as it may occur that its implementation is not sophisticated enough. In those cases, the execution consists of a change of state, the GS actualisation and the call to the GS function concerned to its task and already explained deeply on the Ground Station section.

Finally, a last detailing about the centre ASA state is that its task is accomplished by the iteration over all SAS stored on the global dictionary. For all of them, the server is run. Once all are advertised, a rosspin is executed to listen to all of them.

4.6 UAV - Navigation Algorithms Interface

4.6.1 Introduction

Navigation Algorithms Interface (NAI) module is the core of this project. The reason why all these implementations are useful is thanks to the ability provided by NAI to try new algorithms. The other modules have also been a challenge and very useful for them concerning tasks. But is this functionality what makes the UAV-M decide what action to take depending on the environment that surrounds him. Without it, no movement would happen, and missions and controllers would be irrelevant.

A brief insight into NAI would split it into the different algorithms that it would make use of to accomplish the task of receiving state awareness and give back a new velocity to adopt. The inputs for that algorithms vary depending on the algorithm in the first instance but also on the current scenario or behaviour assumed.

NAI is implemented as a Python class fed by GS to provide it with every information required such as the hyperparameters. Therefore, every instant that GS requires a new velocity, it only needs to request it providing the current situation. The solver algorithm is structured at the initialisation. That fact allows the algorithms to perform tasks that require tracking of values over time.

A list containing the different algorithm that is currently at disposal is shown in Table 4.23.

Table 4.23 Built-in solver algorithms currently on use on the missions depending on the test that must be accomplished as well as its definition of use.

Algorithm	Function
Simple guidance	Purely aims the velocity to the goal without awareness of the environment.
ORCA 3D	Reciprocal collision avoidance dyncamics algorithm from dynamic data.
Fully Connected Neural Network	Simple DL Collision avoidance from dynamic data.
Convolutional Neural Network	DL collision avoidance from depth camera data.

4.6.2 Initialisation

The predominant focus of this module on each of the algorithms makes the initialisations be lighter than in other cases. In fact, the only two pieces of information required as inputs from GS when initialised are its UAV ID and the role to obey. This is due to its characteristics of being concerning only to that single agent. After that, as always, the general information about the current mission is obtained from the ROS hyperparameters.

Eventually, when the chosen solver algorithm is a neural network, some more complex initialisations are required. In that case, a Tensorflow session is created and the network graph is acquired depending on hyperparameters. As it is expected to be a pretrained neural network, a checkpoint of its internal parameters is uploaded, updating them. As the last step of this specific initialisation, input and output tensors are extracted to be reused later during the execution of the deep learning algorithm.

4.6.3 Guidance

This section is focused on redirecting the guidance request to the specialised function depending on the solver algorithm. This task is to be accomplished every instant the drone needs to make a decision, so the information of all the agents and the goal are provided with the request. These pieces of information are also updated on the class attributes to be used later.

4.6.4 Simple Guidance

This first algorithm is devoted to providing the most desirable velocity given the place where the goal and the UAV itself are. The rest of the environment is not taken into account so the ideal output provided should only be used into an ideal empty environment. How to accomplish this most straightforward issue by approximation is shown in the next paragraphs.

If the flag for smooth path mode is set to a value different than zero, this whole process is accomplished by the external nodes that implement that function. In that case, The returned velocity would be directly the one provided by them. In contrast, if the value of that flag is zero, the values of the desired speed module, speed to adopt when on goal and the distance where to start the transition gradient between both velocities are set. Consequently, the distance to the target is computed and its norm extracted.

Gathering all that information, a check of distance to the target is lower than approximation distance is done. In that case, a proportional part is subtracted from the complete module of the velocity. That way, a linear relation is built between distance and speed is formed to gain smoothness on the approximation.

The next step consists of the creation of a vector with the direction inherited from the previously created vector to the goal but, this time, with the just computed velocity module. Also, the direction to the target is calculated for the case it would also be desired to guide the heading. Ultimately, a Twist object is created containing the linear velocity with the selected module and the error on the yaw angle.

4.6.5 ORCA 3D

As ORCA 3D [48] is an external algorithm, a provided API is used to deal with it. The first step fulfilled is defining the parameters asked to perform the labour. Its name and explanation are shown in Table 4.24. As this API does not support static obstacles, those are treated as agents that maintain its velocity equal to zero and are not willing to change it at all. Thus, no parameters refer to them. The radius assigned to the obstacles, as it is a virtual creation, is externally defined on the world definition.

Table 4.24 ORCA 3D parameters that the algorithm needs to be fed with and its function.

Parameter	Function
Time step	Positive time step of the simulation.
Neigbour distance	Maximal distance (centre to centre) to other agents taken into account.
Maximum neighbours	maximal number of other agents the agent takes into account.
Time horizon	Minimal amount of time for which the agent's velocities are computed.
UAV radius	Non-negative radius of the agent.
Maximum speed	Maximum speed the agent would adopt.

Once the required parameters have a value assigned, the simulation of the environment of the ORCA API is initialised providing the params. After that, an auxiliary function, that will be explained later in this section, is used to select what agents and static obstacle are the closest. When the Simple Guidance function studied above provides the most desired velocity, is the moment of passing information to ORCA.

Providing all the information about agents and obstacles gathered up to now, agents to the ORCA environment are appended one by one. As it was said before, static obstacles are defined as stopped agents that cannot change its velocity. On the contrary, for each one of the new agents, its current velocity is selected as its desired one. Once ORCA knows everything about the actual situation, it is the time to take a step forward in the environment.

The new situation in the ORCA environment provides newly selected velocities for every agent introduced. As NAI is only working on one of them, the velocity concerning to him is extracted and transformed into a Twist variable. Eventually, if the heading guidance is required by hyperparameter, the actual angular difference in yaw is passed as part of the twist.

4.6.6 Neural Networks

The Neural Network solver is the most important one and the reason for this project. Its real accomplishment depends almost exclusively on its second part called Machine Learning Research (MLR), but ROS MAGNA is required to a have a link with it. And that link is this subsection of NAI module. Thanks to the MLR, ready-to-use Fully Connected Neural Network or a Convolutional Neural Network are offered. That means that they are provided already trained, there is only need to give the inputs and receive the output. The selection of the type CNN architecture is subject to the depth camera use.

A clear overview of the inputs of the different types of roles is presented on Table 4.25. The output is not extracted as it always the same: the selected velocity. This definition is the first task to be accomplished by the function once the role is defined.

Table 4.25 Inputs to the NN in their fixed order and depending on roles. Most of the fields are divided into x,y and z.

Path follower	UAV follower AD	UAV follower AP	Path follower with depth camera
Own velocity	Own velocity	Own velocity	Own velocity
Relative goal pose	Relative goal pose	Relative goal pose	Relative goal pose
Others relative pose	Goal velocity	Goal velocity	Depth camera info
Others velocity	Distance	Others relative pose	
Obstacles relative pose	Others relative pose	Others velocity	
	Others velocity	Obstacles relative pose	
	Obstacles relative pose		

Once the inputs have been selected, it is time to extract them from the information of the actual situation. Furthermore, it is imperative to assemble the data in the required order and a single 1D vector. Hence, every feature must be split on axes.

In the beginning, the position and velocity of the main UAV may be determined as they will be useful no matter the role to perform. Moreover, it is imperatory to calculate relative positions to itself.

Once those are computed, an iteration over every input is accomplished. Primarily, if the feature cannot be directly obtained from the variables that define the actual situation, it is calculated. And secondly, it is appended to the input vector that is being built. A clarification about the inputs concerning "others" and "obstacles" must be taken into account: every neighbour information is iteratively placed respecting the order for as many are selected to be taken into account.

Once the input vector is complete, a Tensorflow session is run fed with them as the single input. The required variable to be calculated is the only output. This distinction of "single" is due to the existence of batches of training data that must be clear differences with single predictions online.

The selected velocity is now converted into a Twist variable a returned to be adopted by the drone.

4.6.7 Miscellaneous

Hover

This function has the most straightforward implementation. As hover utility is to let the drone on a standby state on a waypoint of the space, no movement is required. That means a linear and angular velocity equal to zero in all three axes. That simple Twist variable is created directly and provided as output.

Upper and Lower Saturation

Sometimes a sharpening of the possibilities in a range of values such as the velocity is required. This function provides thresholding of any amount by a maximum and minimum. Its implementation primarily consists of checking if the entry value is outside the limits and allocate it on top of the crossed boundary if required.

Neighbour Selection

Computing avoidance to every actor, inert or not, would require a prohibitive computational power. Even more, it would not make any sense if some of them are far enough. That is why this function is useful: it helps to select what agents and obstacles are more dangerous than others due to its proximity.

The own position is retrieved, and for every element on the UAV-Ds list, the Euclidean distance between both Poses is calculated. That provides a list with the length equal to the total number of drones within the distances to own position. Consequently, that list is sorter, so the smallest distances appear on the smallest indexes. Absolutely the same is done for the list of static obstacles, and a similar sorted list is provided.

Both lists are returned to the function that requested them. This way, that function will be able to slice the list at whatever point to take into account any number of nearest neighbours.

4.7 UAV Data and Configuration

4.7.1 Introduction to UAV Data

Up to now, the central core of the ROS MAGNA environment is well known. There are, though, two last modules very relevant on the performance. The first one is this UAV-D Python class inside the UAV-M module. Each of the objects created from this class will be focused on one single drone. Also, that single drone would be the one the GS is concerned to or another one. Therefore, this extraction offers the GS a simple way to manage the data of every agent taking part in the mission.

UAV-D is mostly passive, it means, its mission is to gather data which is generated outside on the ROS network. That information is processed and translated to the drone data so UAV-M may easily access to it. However, the module would be a use of that data to transform and republish it to, for instance, improve its visualisation.

4.7.2 Initialisation

As it is the habit, the first task accomplished is the reception of the arguments received. In this case, the ID of the UAV on the scope and the ID on the range of the GS also referred to as main UAV.

On the next step can be found some definitions of ADS-B issue are required but will not be further explained as would be out of the goal of this document. Furthermore, three flags must be initialised to

first values such as the preemption flag, the state provided from UAL and the decision of making use of Rviz so some additional topics for visualisation will be offered. Using its specific function, as always, the hyperparameters are gotten from ROS parameters. As final general initialisations, the CV bridge to deal with the data from the depth camera topic is defined from its library.

It is now the time to create a transform broadcaster that will be used subsequently to render the position received from the different sources.

Finally, if the Rviz flag has been raised and this UAV-D is focused on the main one, some other variables will be required. Those are the first definition of the followed and its header definition, the flag of having changed the state to clear the path and, as last, trajectory and velocity topic publishers.

4.7.3 Listeners

Given its passive behaviour, the rest of the body of this module is composed of listeners and its callbacks. Those can be observed in Table 4.26 and will be shortly described inside the next subsections. To clarify, as each one is focused on one UAV, the " " tilde symbol refers to the namespace of its assigned drone.

Table 4.26 Topics subscribed by UAV-D module from which it retrieves it own information. It is accompanied with the description and if it is used on every simulation or depending on the mission specifications.

Topic	Description	Use
~/UAL - pose	Current stapmed local, cartesian pose.	Always
~/UAL - velocity	Current stapmed cartesian twist.	Always
~/UAL - state	Codification about landed, armed or flying amongst others.	Always
~/Environment - raw ADS-B	Specific ADS-B information of actual navigation variables.	If required
~/r200 - raw depth camera	Depth value about each pixel provided by Gazebo.	If required
~/magna - preempt command	Command from ANSP to finish actual behaviour.	If required

UAV position

Initially, the position is stored on the attributes. After that, if the Rviz flag is raised and the assigned UAV is the main, some other tasks are computed. First, if the flag for the change of state has been arisen, the own path is cleared, and the flag deactivated. Consequently, the actual position received is appended to the path and altogether is published.

Out of the Rviz issue, if the communications are chosen as ADS-B based and the UAV is not the main one, the own pose is broadcasted. This is performed by the transformation of the Pose type information received into a transform type.

UAV velocity

As before, the velocity is stored on the attributes. Also, if the Rviz flag is active, the frame to which is referred the Twist is changed to be onto the one for the UAV. The reason for this is when it is now published, that twist would be observed on Rviz and clearly understand the velocity of each UAV.

UAV UAL state

This one is the most straightforward callback as the data received is directly passed to the attribute, so the GS can know if the UAV will accept new communications or still have to wait.

Depth camera

A CV bridge is required to cope with this kind of information. It must be read the message received and its encoding extracted to accomplish this task. That encoding is then passed to the bridge, so the pixel data is correctly provided. Finally, that depth information is converted into an array for better and normalised use.

Preemption command

When in a constant behaviour like Follow UAV AD, any external command is required to finish the action. If the message from the GS about a critical event is received, its flag is arisen, and the state of the current behaviour is preempted.

4.7.4 Motivation of UAV Configuration

This architecture deals with a wide range of possibilities on the configuration of the UAV. Some examples of it would be the autopilot onboard or the possibility of an agent to be real or simulated. All those specifications may be restricted to the model of drone chosen, as it would be the case of the autopilot. All of them may be found in Table 4.27. However, the simulation or not of a UAV is subject to the mission. That is the reason why a class is dedicated to this kind of things as it retrieves and gathers all the information in standard variables suitable to be used by the rest of the project in one single way.

Table 4.27 Parameters that define the configuration of UAV-C module. The description defines its use and the definer, if it is directly predefined by the model chosen for the UAV or if that information must be provided on the mission definition.

Parameter	Description	Definer
Autopilot	Company owner of the navigator system.	Model
UAL use allowed	UAL implemented for defined autopilot and defined model.	Model
Security radius	Distance for collision detection.	Model
ID	Single identification for the UAV.	Mission
Model	Referring to the autopilot, determining the frame type.	Mission
Mode	SITL or real experiment.	Mission
Marker color	Color and alpha for RVIZ visualisation.	Mission
UAL use	Option to directly use the autopilot interface.	Mission

As one object of this class is concerned with one of the agents of the mission, any UAV-Manager should create one UAV-Configuration for any neighbour with whom it would like to know about. Of course, in an ideal situation, a fully standardised interface as UAL this would not be required. Nonetheless, each model and the autopilot would offer singularities. Hence, it is a better option to rely on an intermediate bridge more customizable inside this architecture.

4.7.5 Initialisation

The only parameter required at the initialisation is the ID of the referred drone. With that identification and the ROS parameters such as the mission and submission, a path to the JSONs is determined. Those JSONs define the mission, from which initially the model is retrieved, and the fix configuration of that model. Gathering both, every piece of information required for the whole mission is determined. Once every feature required is defined, its corresponding functions are used.

Once the model is defined, another JSON is read concerning it. The information there provided is common to every airborne of that model, corresponding to parameters such as the autopilot, the safety radius or the maximum speed acceptable.

4.7.6 Interface definitions

A single value corresponds to a type of autopilot, a kind of model and the use of UAL. Depending on the information offered by any autopilot, it opens several topics to read or write it which would be out of the scope of UAL. As well, a model would be equipped with a sensor independent of the autopilot. In addition, UAL itself offers a range of topics and services that need to be initialised in a conventional way. For all those possibilities, several dictionaries according to ROS nature groups, are fulfilled in each case.

4.8 Experiments

4.8.1 Introduction

This section is intended to provide some examples of the use and potential of ROS-MAGNA. Initially, two mission experiments, with its ad-hoc world, are offered to emphasise how parts of the state machine have its influence on the UAVs. Finally, another experiment is carried in order to compute the influence of the number of agents simulated in the performance. Videos of all experiments are at disposal at GRVC webpage:

ROS-MAGNA. These are the media presented on the ROS-MAGNA ICUAS paper, one of the outcomes of this project.

4.8.2 Experiment I: State machine safety branches

On this first experiment, the potential of using a state machine for the definition of the mission and its capability to deal with unexpected critical events is demonstrated. Two UAVs are simulated on GAZEBO. The first one of them is in charge of package delivery throughout a space with static obstacles. The second UAV must follow 1 at different relative positions. The central core of the state machine implements these tasks, which are not deeper studied, but the safety ones explained on the rest of the section.

Once the mission has started, both drones use ORCA to avoid obstacles and each other. The first delivery accomplished with success, however, on the way to the second delivery point, UAV 1 runs out of battery. This moment is depicted on Figure 4.18 with an image from GAZEBO and SMACH. Given that event, the SM enters on a safety branch that leads drone 1 back home while drone 2 maintains its behaviour of following it at different relative positions. Once UAV 1 has landed at home, it is awaited for the user to inform the SM of full charge by pressing a button. A screenshot from RViz of that moment is presented on Figure 4.19. Consequently, the main task is resumed and the delivery process continues as normal. A video of the whole mission is offered on GRVC webpage: ROS-MAGNA: Simulation I.

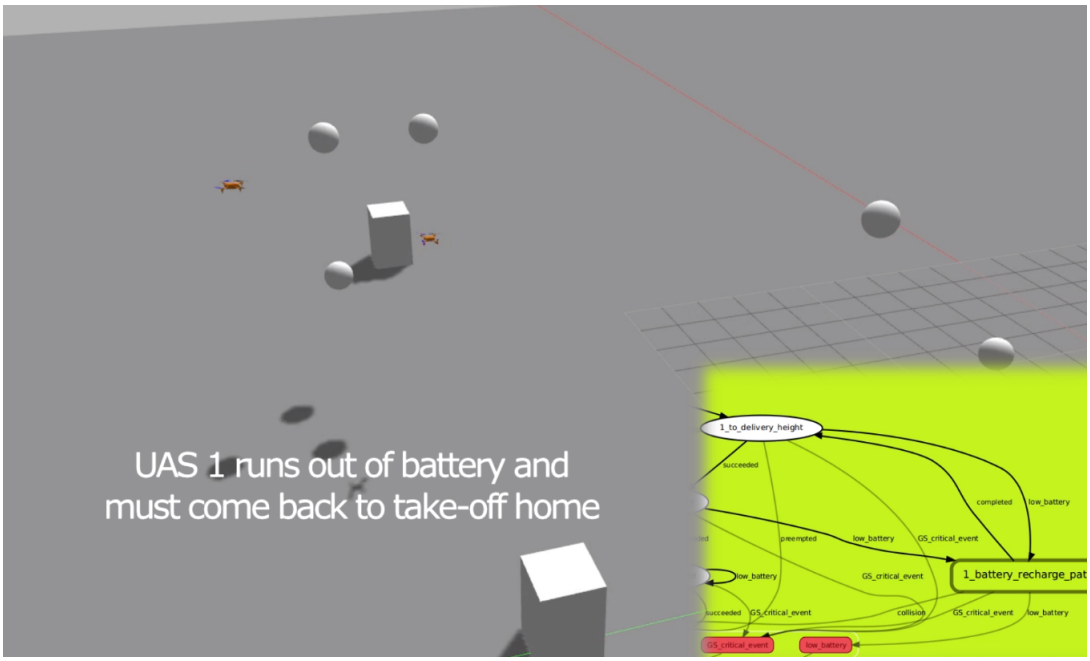

Figure 4.16 GAZEBO simulation of the moment in which UAV 1 runs out of battery and UAV 2 is following it at a relative distance. On the bottom right corner, the mission state machine has switched to the corresponding nested state machine to come back home for recharge.

To deal with the possibility of failure of any of the collision avoidance algorithms, the same delivery mission is repeated without the use of any of them. As it is supposed, given the dangerous zones with static obstacles, there is a moment in which drone 1 detects that proximity threshold with one of the obstacles has been transgressed. That moment is depicted on Figure 4.18. On it, the short distance between the front drone and the sphere is evidenced in GAZEBO and, on the bottom left corner, the state machine has changed state following the collision row to a hovering state. Both drones start hovering and a wait state is executed to let a pilot take manual control of them. As it does not actuate, the next action is landing both UAVs, which are able to get safely to the ground. A screenshot of this final movement in RViz is shown in Figure 4.19. The whole process of the mission may be better-understood thanks to the video on GRVC webpage: ROS-MAGNA: Simulation II.

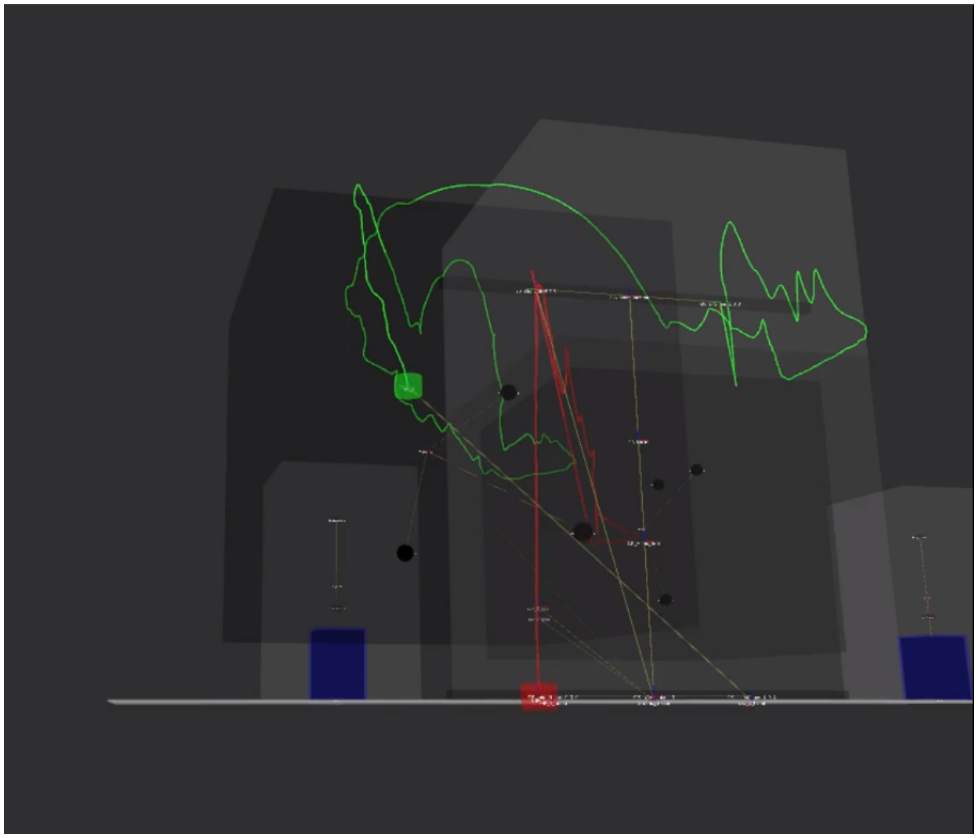

Figure 4.17 RViz visualization of the delivery environment (purple boxes), static obstacles (black spheres) and paths followed by UAV 1 (red) and UAV 2(green) at the moment drone 1 has landed for recharge and drone 2 is following it at a relative distance.

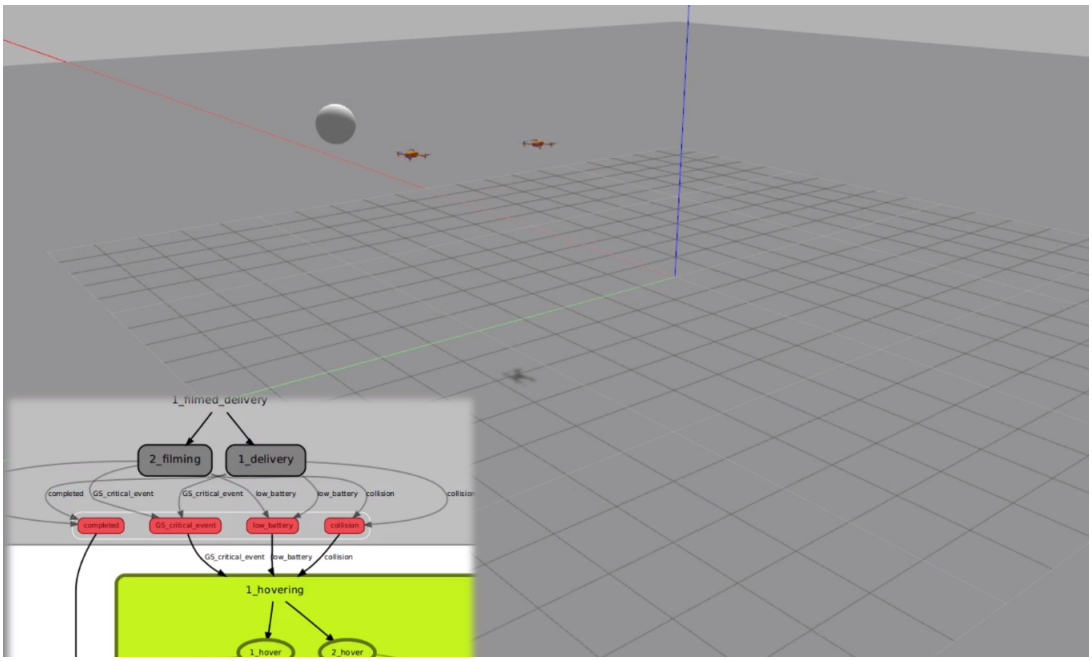

Figure 4.18 GAZEBO simulation of the moment in which UAV 1 detects an imminent collision and UAV 2 is following it at a relative distance. On the bottom left corner, the mission state machine has switched to the corresponding branch for hovering, waiting and eventually landing.

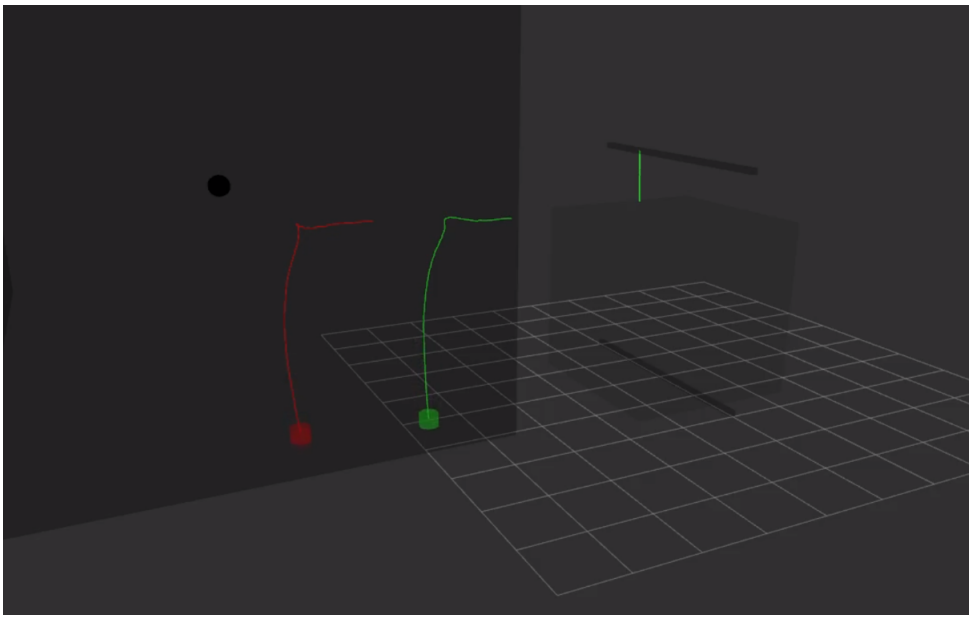

Figure 4.19 RViz visualization of the paths followed when drone 1 (red path) has detected proximity to an obstacle (black sphere) and both UAVs have landed safely.

4.8.3 Experiment II: SIL, HIL and real joint mission

The second experiment is intended to demonstrate the capability of ROS-MAGNA to support, on the same mission and the same fictional arena, software-in-the-loop, hardware-in-the-loop and real drones without important implications to the performance.

The HIL drone is an IRIS model with PX4 autopilot simulated with GAZEBO. The SIL UAV is an f550 DJI model whose autopilot is real hardware but its inputs are simulated on the DJI Assistant software. Finally, a full real f550 DJI model is also employed, performing a real flight on Escuela de Ingenieros at the University of Seville. The three of them are presented on Figure 4.20.

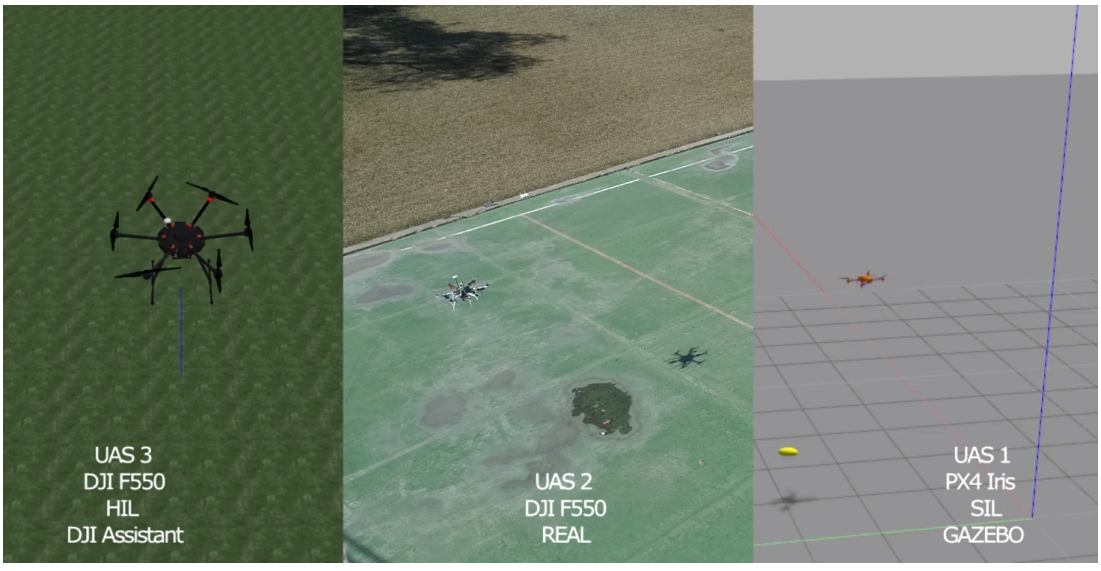

Figure 4.20 Visualizations of the three UAVs employed on the mission. On the left, SIL DJI f550 model with real autopilot hardware and inputs simulated by DJI Assistant software. On the centre, real DJI f550 performing a real flight at the University of Seville. On the right, a SIL PX4 Iris model fully simulated on GAZEBO.

Given the higher risk of the mission with real UAS, no critical events are simulated but simple path

following behaviours. A world simulating a real solar plant is modelled, providing there the zones where the panels at as well as inspection and approximation zones for the three of the panels groups associated with each UAV. The world and performance may be better understood with Figure 4.21. It can be appreciated the purple zones of the panels, cyan zones for inspection and yellow, green and red zones for approximation. The same colours are used to visualize paths expected and performed by each UAV. At the backend of the image, several safety heights are plotted for each drone, so the displacement to the panel volumes is accomplished at different heights for safety. The video containing the full mission may be found on GRVC webpage: ROS-MAGNA: Experiment I.

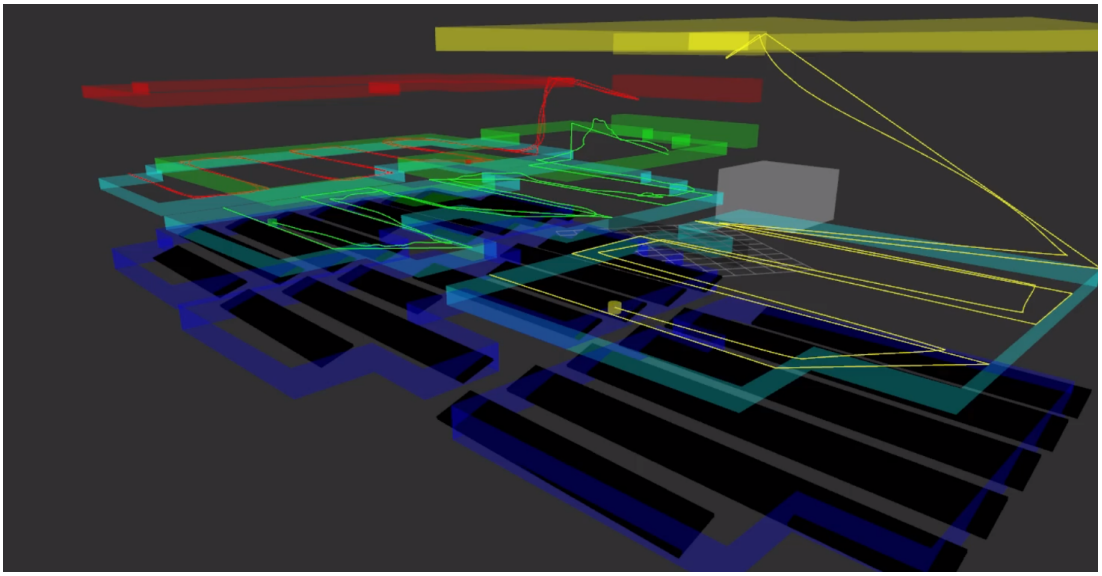

Figure 4.21 Mission performance visualized on RViz. Panels zones (purple), inspection zones (cyan), approximation zones (yellow, green and red) and corresponding UAVs paths with their associated colours. At the backend of the image, take off zone with different safety heights.

The real Canamero solar plant is located on Extremadura, whose shape is exposed in Figure 4.22. The full process of building the modelled world is visualized in a video on the same webpage.

Figure 4.22 Google maps screenshot of the real Canamero solar plant at Extremadura used for world modelling.

4.8.4 Experiment III: Delay on computation due to number of simulated UAVs

The Navigation Algorithm Interface (NAI) that runs inside each drone has the duty of computing high consume computations. One of them would be an iteration of the ORCA algorithm. It may has a strong, negative impact on the development of the mission if a high number of these processes are running on the same device. In some occasions, turns out to be definitively negligible given the deep alteration of the results. Due to this phenomenon, it is very useful to make a short study to quantify that impact in order to be aware of the limits.

Under that idea, the next experiment is based on comparing the time to the response that a single UAV lasts when a move command is received when it is the only drone simulated or when it is accompanied of one and two more.

On Figure 4.23, a MATLAB plot of the results is exposed. The graph is divided into three different subplots, each of them showing the behaviour of UAV 1 when there are one, two and three drones on the mission, respectively. For all of them, the coloured bars at the top represent stages on the command process. The first one, on purple, is concerned about the goal definition on the Ground Station node. The green one, also on the GS, is the time waited to send the command. The white intersection between green and cyan is the time elapsed during ROS communications (service) which not always is perceived by the rosbag tool. The last bar, on cyan, points out the process occurring on the UAV node in order to perform the task. Finally, the blue line during the whole subplot shows the current position on X-axis of the corresponding UAV.

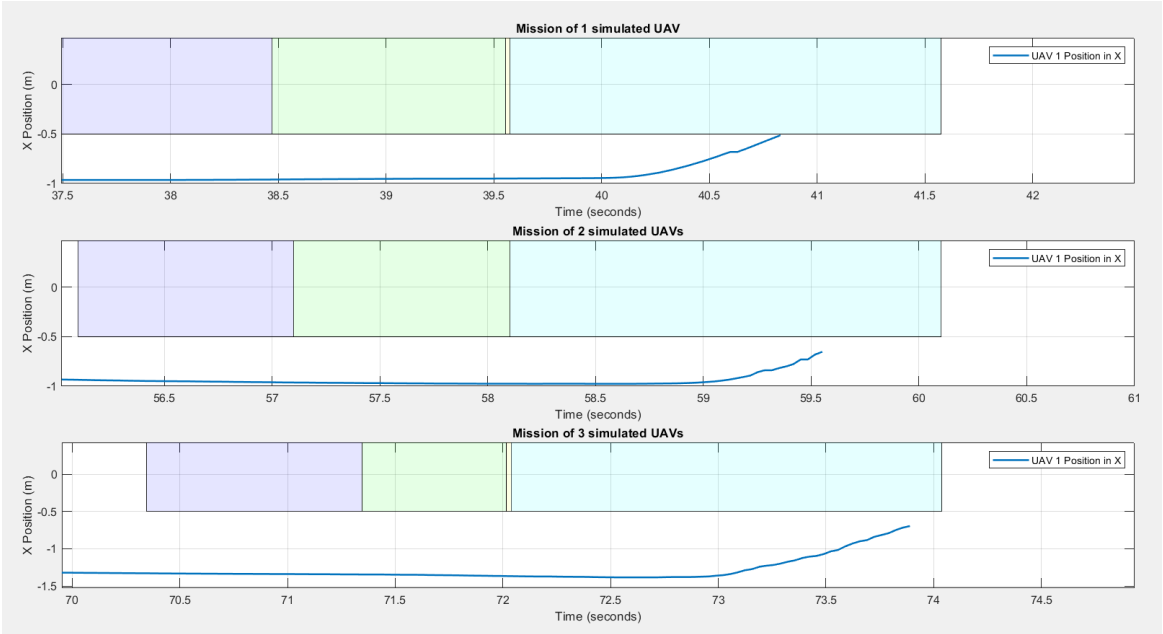

Figure 4.23 MATLAB plot of the Experiment III. Three different subplots for 1, 2 and 3 UAVs on a mission but only plotting information of UAV 1. For all of them, the coloured bars at the top represent stages on the command process: goal definition on the Ground Station node(purple), time waited by GS to send the command (green), process occurring on the UAV node in order to perform the task (cyan) and curren position of UAV 1 on X-axis (blue line).

All subplots are identically scaled on X-axis for comparison and the initial moment of command reception has been aligned. It is easy to realize that, the most UAVs on the mission, the more on the right the blue line starts moving upwards, changing its state according to the command. In light of the outcome, this experiment demonstrates that there exists an influence of the number of UAVs simulated on the reaction time.

As an effort to dig deeper for the cause of this phenomenon, the time elapsed on one single iteration of the ORCA algorithm has been also quantified. The mean averages, on milliseconds, are [0.340,0.359,0.386]. This is, with little doubt, an example of the reasons that a chain of similar processes on the same iteration and a concatenation of that iterations result in the happening above described.

Other possible measurements would be interesting, for example, to communications delay. However, given the diverse casuistry, it has been discarded as none of the experiments would represent a feature as

general as the purpose of this project.

5 Machine Learning Research

5.1 Introduction

The main reason for developing the previous chapter was exclusively this research. All those modules and implementations, which took much more time, were always focused on providing a helpful work frame. The ROS environment meant the basis where this next chapter is built upon, providing it with all the information and features required to develop, train, and test new algorithms.

A Machine Learning (ML) research is a very general definition. It is actually intended to be this way. The goal was not to pursue a single objective neither to focus on a single algorithm. In contrast, it was wanted to explore the different existing alternatives in order to study and learn about the challenges, requirements and solutions they have been born to provide. This is, essentially, a project to get inside the Artificial Intelligence world by means of self-experience. As was referred in Navigation Algorithm Interface section of the previous chapter, is there where all Machine Learning algorithms are introduced into the simulations and, therefore, into the UAVs. The overall architecture is presented on Figure 5.1

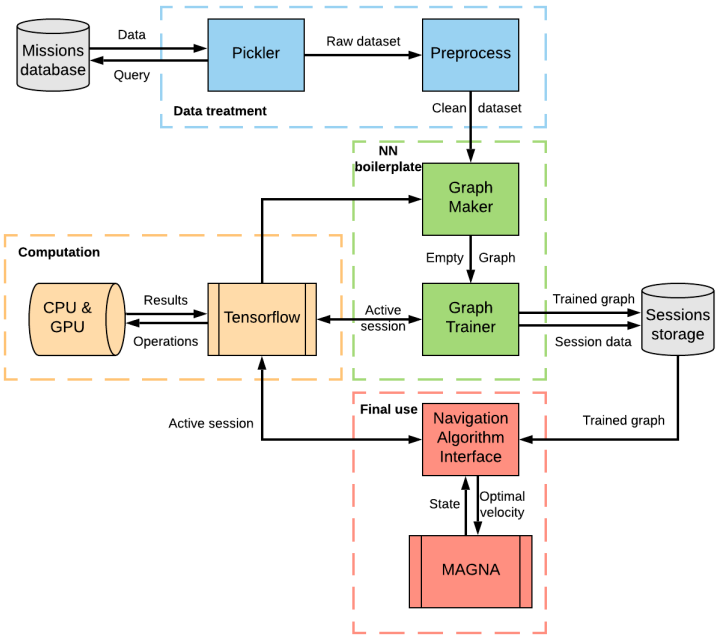

Figure 5.1 General structure for the Machine Learning Research. The workflow is straightforward from the database, suffering transformations on the treatment stage after entering on the training process inside the boilerplate. After that, the network is ready for final use on a real application. Tensorflow offers the capabilities to deal with machine learning elements and also manages the computing devices. Every information generated is stored in the sessions database.

Attempting to that goal, Deep Learning algorithms have been developed in the first place. As a first step, simple fully-connected Neural Networks have been produced to achieve basic goals such as following a static path or a moving target. For that purpose, all information of the surrounding environment is supposed to be well known and deterministic. The second step is to go deeper into the knowledge stored by the net and append convolutional layers. That way, a more complex net is able to accept the pixel information from a depth camera, analyse it and decide the best direction given a goal predefined. Taking into account the supervised learning nature of the problem, an auxiliary algorithm called ORCA is employed to become the teacher of the nets.

As a last, promising step forward, those deep networks will be in future work used inside a more challenging algorithm: Deep Reinforcement Learning. This time, the problem is not about using another algorithm and learn from its actions. The RL agent must learn by itself attending to the reward he gets from its own actions. The possibilities of application improve exponentially but the scope is kept the same as for the previous supervised learning: following static or dynamic goals.

Having reached this point, an overview of the scope of the research has been offered. It must be highlighted its learning purpose to understand that no beyond state of art discoveries were awaited. It would come later in time. By now, a priceless foundation is accomplished and a basis for further research has been settled.

5.2 Steps previous to training

5.2.1 Manager

Motivation

Similarly as happened to the ROS MAGNA Environment, this part of the project needs a central core that orchestrates the rest of executions. This Manager module is the one commissioned to define and perform in order for every pickling, preprocessing and training processes. Each one of them is implemented as a basic module defined in a Python class.

Hyperparameters

Its first task brings the definitions of the different hyperparameters, shown in Table 5.1. Those are used to determine every bounding information for single processes. When they are created as a list, an iteration amongst its elements is performed in order to realize different pieces of training concerning each of its values.

In the first place, there can be found a sort of definitions that characterize the path from where the dataset is to be retrieved from. Continuously, training parameters are settled in different lists to define the various trainings that will be performed and compared. Finally, specific characteristic for both fully connected and convolutional architectures are reported. If any this parameters is a list, the training process will be repeated once for each parameter on the list in order to research its influence.

The last one of the definitions here required is concerned with the architecture of the neural network. Two different lists one for each stage, fully-connected and convolution, are defined. In the case of the convolution, a list of dictionaries is deployed to contain several parameters. Each element of the list will correspond to a single architecture and each architecture, is a list with the parameters of each layer. The possible parameters to tune are presented in Table 5.2

Training performance

Once the dataset has been retrieved, the "future velocity instants" parameter (FVI) is the only one that compromises the performance of Pickler and Preprocessor tasks. Due to this reason, its selection along the concerning list is accomplished before those both processes. Once an FVI has been chosen, Pickler and Preprocessor are accomplished. Its implementation will be deeply explained further in this section. It is possible to decide whether execute or not this two tasks using a flag given the fact that once the data is selected and preprocessed, it may be useless and costly to remake the same computation to obtain the previous result.

The next step is to initialise Graph Builder and Trainer. Finally, three more lists must be iterated in order to make trainings for different values of learning rates and FC and Covnet architecture definitions. Once a value has been selected for each one, the Graph is created and the training process takes part.

Table 5.1 Hyperparameters that characterize the main features of the Machine Learning Research implied on data retrievement and training.

Hyperparameter	Description	User
World	World folder of the storage to choose for supervised dataset.	Pickler
Subworld	Subworld folder of the storage to choose for supervised dataset.	Pickler
Mission	Mission folder of the storage to choose for supervised dataset.	Pickler
Submission	Submission folder of the storage to choose for supervised dataset.	Pickler
Nº Dataset	Where to retrieve supervised dataset.	Pickler
Nº neighbours	Number of closest agents taken into account	Pickler. & Preprocessr
Teacher role	What roles of the mission to choose within supervised dataset.	All
Teacher algorithm	What navigation algorithm to choose within supervised dataset.	All
Batch size	Number of instants from which to learn every time.	Graph Maker & Trainer
Learning rate	Gradient influence on weights tunning.	Graph Maker & Trainer
FC hidden layers	Length of each hidden layer.	Graph Maker
Covnet params	Dictionary to define layer parameters.	Graph Maker
Nº steps	Number of baches used for training.	Graph Trainer
FVI	Future velocity of instants in advance to feed the neural network.	Pickler

Table 5.2 Parameters that define the architecture of a convolutional neural network on its convolutional and fully connected part.

Parameter	Description	Network stage
Hidden neurons number	List with a number of neurons for each layer.	Fully-connected
Image size	Width and height of the image.	Convolutional
Num Channels	Number of channels of the image.	Convolutional
Patch size	Width and height of the kernel.	Convolutional
Depth	Number of channels of the kernel.	Convolutional
Padding	Addition pixels virtually added.	Convolutional

5.2.2 Data pickling

Data retrieving

The first step is focused on accessing the database created by the ROS Environment and extract only the information related to the current training. For that task, previously have been defined the hyperparameters. They determine where is allocated that information and what filtering should be applied.

There are two ways to provide the Hyperparameters depending on the use done to the module. On one hand, it might be provided directly from the Manager module, where they have been defined. On the other hand, it would be executed from ROS and in that case, the hyperparameters would be retrieved as ROS parameters.

Thanks to that information, an introspection to the database is affordable. The first filtering has been primarily done by the path definition but a deeper one is required as shown in Code 5.1. Throughout every simulation inside the path, the ones that offer a world definition are selected. This step is important due to the eventual problems that may occur during a simulation that make impossible the creation of such a vital information file.

For all those who present the definition of the world, a second level of filtering is passed to extract only the succeeded ones.

Code 5.1 Data retrieving process.

```
Extraction of hyperparameters depending on the source whether ROS params or by
    arguments
Build, from hyperparams, the path where to retrieve the dataset from
Check every simulation contained in the search path
  Check if there is a world definition inside
    Introspect that world definition and check the succeed field to be positive
      Look for UAVs with the desired role and algorithm
        Load data, parsing it into a pickle
        Append it to the global pickle with the rest of simulations
Create the pickle file
```

Data parsing

The selected information is then parsed into instants, represented as rows in the CSV dataframe. Consequently, each column of the row is cut up into numerical single elements as exposed in Table 5.3. The order followed is also crucial as inputs to the Fully-connected Neural Networks (FNN) must always be the same. In addition, each role has its own input structure as in needs to be fed with different information. On the case of Convolutional Neural Network (CNN), the input image is the only input, on pixels, received, so no extra structure is required to be defined.s

Table 5.3 Order in which dataset variables are crumbled and mapped to FNN inputs depending on the behaviours studied.

Variable	Partitions	Teacher role
Own velocity	[x,y,z]	All
Relative goal pose	[x,y,z]	All
Relative neighbours position	[[x,y,z],[x,y,z],...]	All
Neighbours velocity	[[x,y,z],[x,y,z],...]	All
Goal velocity	[[x,y,z],[x,y,z],...]	Path Followers
Distance	d	Path Follower at distance
Distance relative to goal	[x,y,z]	Path Follower at position

In contrast, the output of the created neural networks is always the velocity to adopt for all of them. This velocity is also split, selecting the corresponding row taking into account the number of future instants to be forwarded.

To deal with this cup up of dataset matrices into inputs, the function LoadAgentData is employed. It has built in some lists with the order of the inputs for each role, so an iteration over it is performed to know what treatment must be performed to each element so to split it into numerical variables. As a result, for each instant, it is obtained a vector of the length of the inputs of the NN containing float numbers corresponding to one instant. When every instant is assembled, the variables is ready to be stored in a pickle file for later treatment.

A new folder is then created inside the Machine Learning Research database and the pickle file is stored on it corresponding to the current working parameters.

5.2.3 Data preprocessing

Data mathematical preparation

As it is common, hyperparameters are retrieved in order to generate the path where to retrieve the pickle file from. A dictionary containing mathematical information of the dataset is computed using Numpy library. These values are used to perform transformations on the data. This preparation is required at this point so the learning process is facilitated mathematically, no extra information is biased and the convergence is faster.

Primarily, the mean value is calculated and subtracted to the whole dataset so the new mean is set to zero. After that, maximum and minimum values are compared. The one with maximum absolute value is used to normalize data by dividing every element by that value. Finally, dataset rows were shuffled and simulation instants are separated as the training is independent of them.

Dataset partition

The dataset must be split into three parts: train, valid and test. The first one is the largest and actually used while training. The train dataset is also lately cut up into batches. Valid dataset is used while training to compare the current performance of the network trained with the train dataset. Finally, the test dataset is required at the end of the training process to compute a final average performance metric in new data never used during training.

Percentage values of each dataset are currently set to 70%, 20% and 10% respectively. However, those would be suitable to be provided by parameters.

As the last task of the Preprocessor module, a new pickle file is created into the corresponding path of the MLR storage. That file contains the three different datastets as long as the labels for each one, it means, the output velocity in this case.

5.3 Neural Network boilerplate

5.3.1 Graph building

Initialisation

The initialisation of this module built upon a Python class is achieved by retrieving the preprocessed pickle file from the database after composing the path using hyperparameters. As explained in the previous section, this file contains a dataset split into three subdatabases called the train, valid and set. Each one of them is also divided into inputs (called dataset) and output (called labels).

When a new training process is required, the function TrainNewOne is required. It receives every parameter tunned on the Manager module inside the lists that define the different pieces of training.

As its name explains, Tensorflow (TF) working philosophy is based on tensor (matrices of multiple dimensions) operations. Those operations are represented as a graph where each node is an operation and each row represents a tensor. The great advantage of working with graphs is that there are not compound of a straightforward algorithm but a lot of them may coexist. It means, it is not a monodirectional computation. Any tensor of the graph may be asked to be computed if the necessary previous data is fed. Therefore, it may be computed, and even actualised, a branch of the graph without even have read another one.

Hence, the very first task is to open a default TF graph inside wich the rest of processes will be accomplished. Mean and standard deviation values for randomly generated weights are set to 0 and 0.1 respectively as default. Those would also be parameters of research but it has not been still implemented. Depending on the existence of the convolutional stage, the corresponding image parameters such as image size and total name of pixels are required to be defined. Furthermore, the inputs for each stage are also divided.

For each one of the parts of the incoming dataset and its labels, a placeholder is created. This is an initially empty variable whose variable is defined later on the training but whose size and type is already defined.

Finally, constant variables store the mean and maximum levels computed during preprocessing steps. It is easy to understand that these values are required when the output of the neural network needs to suffer the inverse process.

Fully-conected stage

Two lists for weights and biases of neurons are initialised empty. A function is specifically defined to gather the creation of this stage provide a common name scope to facilitate the visualisation later.

Inside that function, an empty list is created to contain the future outputs of every layer. Furthermore, the list containing the number of neurons on each hidden layer is extended. This is accomplished by concatenating another layer at the beginning of the same length as the number of inputs to this stage and the same process at for the last layer of the length of the outputs. Those initial and final layers would not be called hidden as input and output values of the network are easily interpretable for humans.

All parameters are defined to perform the actual computations of tensors. For every layer in a loop, its corresponding weights are randomly initialised within a truncated normal distribution and the biases are set to zero. The output of the prior layer is multiplied with the weights and the bias summed. To the result, the activation function is calculated and stored as the output of the current layer to be used later by the next one.

Histograms of weights and biases of every layer are also created for later visualisation in the debugging. The resultant graph is presented on Figure 5.2. On the left, it may be appreciated the operations of one single layer in order from down upwards. On the right, the full stage composed of four layers concatenated.

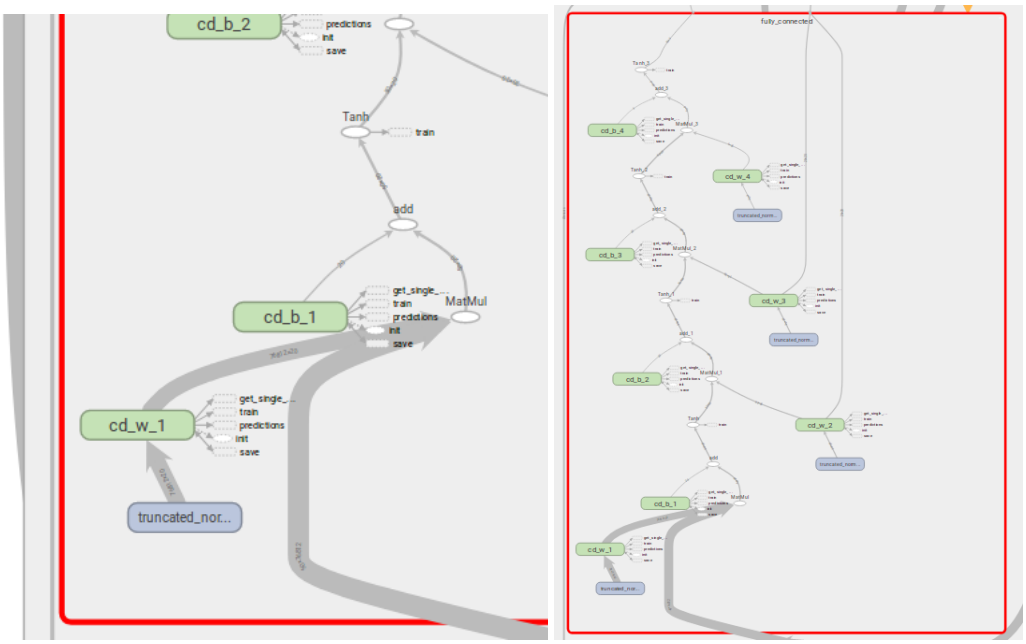

Figure 5.2 On the left, Tensorboard graph for one layer with respective multiplication and addition of inputs, biases and weights. On the right, the concatenation of four layers completes the fully-connected stage.

Convolutional stage

Given the fact that the pixels of the image have been flattened, it is now required to reshape it into its original size. As it was done for the fully-connected stage, biases and weights lists are initialised empty. The next step is to define the function that builds up the corresponding graph under a name scope.

For every new layer, its corresponding parameter dictionary is retrieved and the layer computed. The weights are also initialised randomly in a truncated normal distribution and the biases are set to zero

The graph of each layer is created by the concatenation of three different steps. The first one is the convolution itself, provided by a TF built-in function that receives the output of the prior layer and weights along with kernel definition. Its output is summed up to the biases and a ReLu operation is implemented over it. Moreover, the result is pooled by a kernel of the same padding. The output of the layer is stored on the corresponding list, ready to be used by the next layer. Finally, histograms of weights and biases are stored.

The resultant graph is presented on Figure 5.3. On the left, it may be appreciated the operations of one single layer in order from down upwards. On the right, the full stage composed of four layers concatenated.

Stages assembly

The complete model is assembled by a function that makes use of the two ones just defined for each stage, depending on the architecture type. It also introduces a name scope that gathers them.

On one hand, if the architecture is simply composed of a fully connected stage, the task of this function is only to call the corresponding function and provide the same output. On the other hand, if both stages are required, a first call to the convolutional stage function must be done in order to create it, as it is the first one the input data goes through. Consequently, provided the outputs of the convolution and the rest of the inputs, the fully- connected stage is built up.

The output before the post process is also included as a TF diagram as it has a very high value to understand the behaviour.

Figure 5.4 shows the joint of the whole model. It may be appreciated the retrievement of inputs fed, its transformation as the flattening of the image, the model, and its logits outputs.

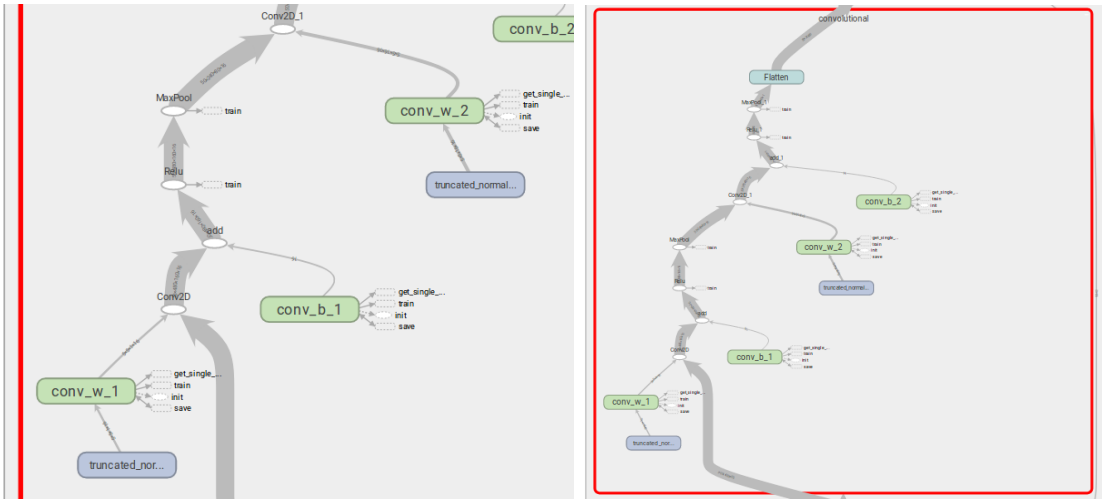

Figure 5.3 On the left, Tensorboard graph for one layer with respective convolution with weights applied to the image, addition of biases, ReLu and maxpooling processes. On the right, the concatenation of two layers completes the convolution stage.

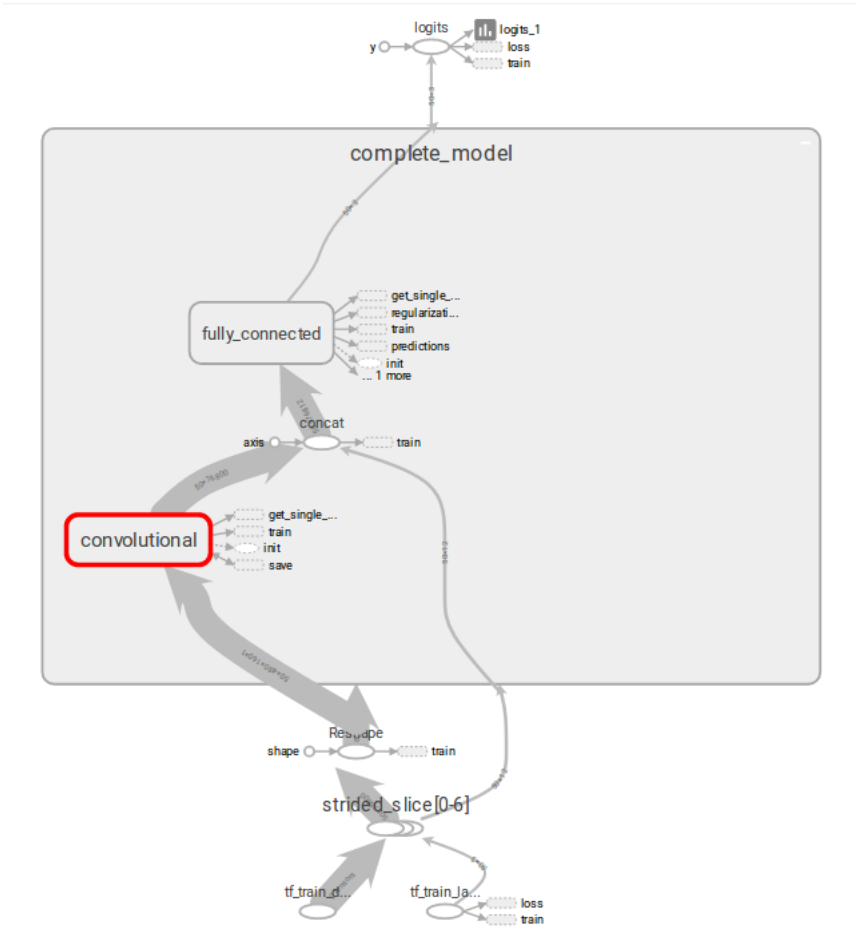

Figure 5.4 Tensorboard graph for a complete model from input to output. It represents the concatenation of the convolutional stage and FC inputs to the FC stage.

Output retrievement

As the purpose of this section is to return a velocity provided the current state of a drone and its surroundings, this function focused merely on that. The neural net would be supposed to be already trained and ready to offer the optimal velocity. As ever, this is done under its corresponding name scope.

The preprocess operations must be performed to the inputs before they entered on the neural network, so the previously computed mathematical parameters are used again. When inputs are normalized, they are fed to the graph and the input of the complete model is required. That value contains the optimal velocity, but need to suffer the inverse treatment.

Figure 5.5 presents where this new tensor is located inside the graph. The single input is fed and the outputs of the model are used to get a single velocity.

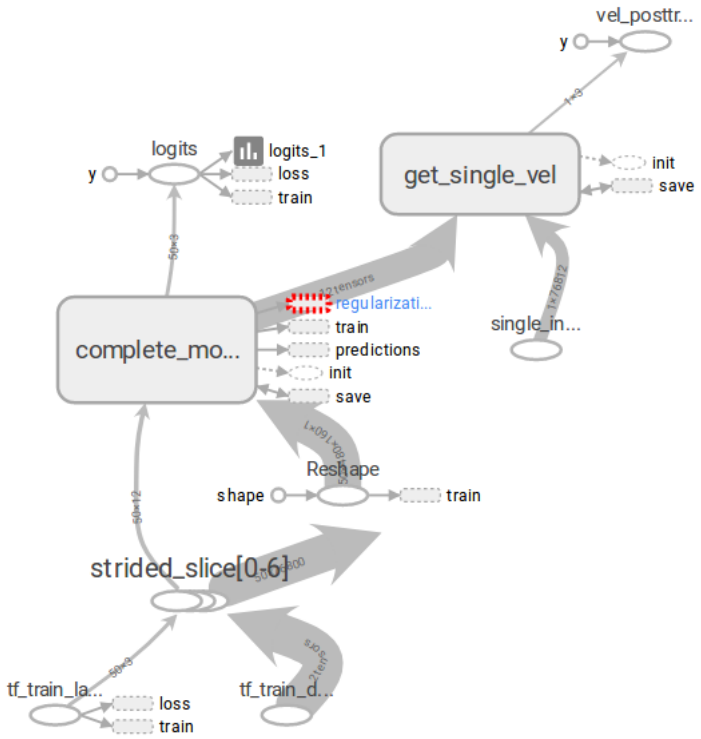

Figure 5.5 Tensorboard graph for one single computation when the network is used on the final application. The network receives a single state of the UAV, the data is transformed during the complete model above explained and the output is mapped to an optimal velocity.

5.3.2 Graph trainer

Auxiliary graph branches

The training algorithm is composed of some extra computations around the basic neural network that must be defined inside the graph. Each one has received its own name scope to facilitate the localization of each step.

The regularization is processed by a built-in TF function. The weights of each layer are multiplied by a fixed constant and summed up. The resultant regularized loss is used in the optimization process explained below.

As it is well known, a loss function is a function to be minimised. Hence, every component of it tends to be reduced by the optimizer algorithm. For this purpose, the loss function is defined as the mean square error between the outputs of the NN and the ground truth labels. That way, the optimizer minimizes the error and the NN outputs get more similar to the labels. In addition, in the loss function, the regularized loss in added to the error so the weights tend to be small.

The chosen optimizer is gradient descent as is the most common in the literature. This is as well a function already implemented on TF so nothing else must be accomplished but provide it with the learning rate predefined and the computed loss.

Finally, predictions for valid and test parts of the dataset are defined, as it was done during the already explained process for the training dataset.

Training

Once the graph is completely defined, it is the time to train it. This process must be done inside a Tensorflow session, which is the very first step to accomplish.

The global variables of the graph, such as randomized weights, are initialised. Besides, a saver TF utility is created and a TF summary merges the created variables to form the graph. The path to the session storage is built using every important parameter that defines this single training process. The saver is used to store the model graph.

A for loop is used to iterate over the steps of the training process that have been chosen. The training set is sliced into batches of the predefined steps, so only one batch is used on this step. Of course, this process is performed for the dataset as for the labels.

WIth the inputs must be introduced into a feed dictionary so TF know the placeholder where to deposit each variable. The optimizer, the loss and the train prediction are computed so the weights and biases of the whole neural network are updated toward optimisation. Information about some of the training steps is periodically printed. The current lost and accuracy are shown. The accuracy is computed by comparing the results with the labels seemingly to the training process. At the end of the training, the same information is also presented to know the final results.

The final graph containing every single operation of the whole process is demonstrated in Figure 5.6. On it, the prior graphs are contained as well as the computations designed in this chapter.

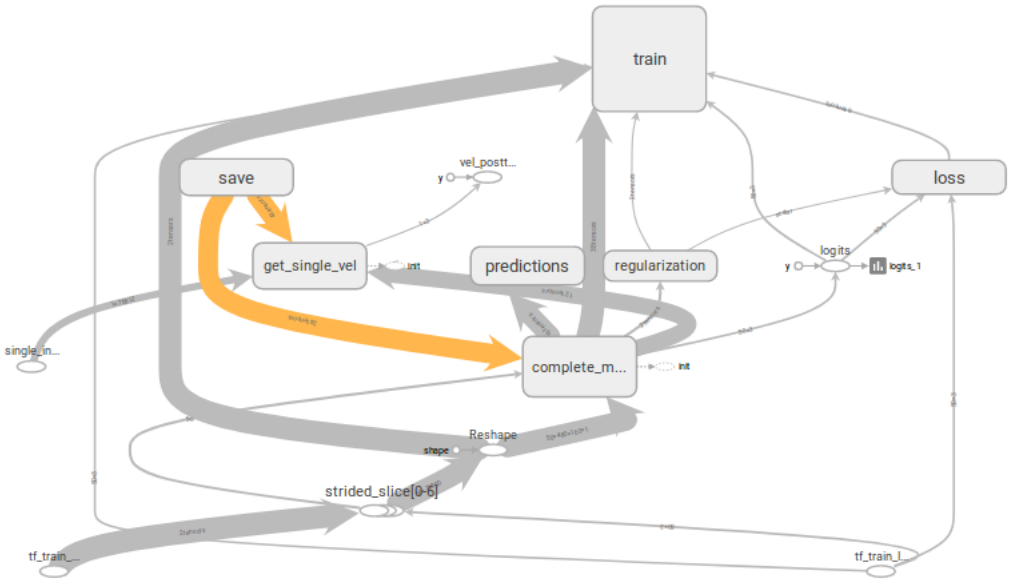

Figure 5.6 Tensorboard graph for complete training and final use process. Containing inside and wrapped on simple boxes, the whole process above explained is aided by processes required on training such as prediction, regularization and the calculus of the loss function.

5.3.3 Example of use

Auxiliary graph branches

In this section, it is explained step by step how this framework should be used to train and test a new neural network.

Initially, ROS-MAGNA is employed to generate an appropriate dataset suitable for the purpose of the target mission. Therefore, a new world and mission must be carefully designed so its characteristics represent the ones on the application and every element on it present a utility. In this example, we want to train a UAV in avoiding another one while following a path. For that purpose, the world is modelled as four cylinders placed on the four corners of a square. Besides, two UAVs must execute a mission for following a waypoint path. Every waypoint is randomly placed somewhere inside of one of the four cylinders. That way, conflicts on both trajectories are created in very different dimensions. ORCA algorithm is used for collision avoidance taking into account one single neighbour, as presented on Figure 5.7.

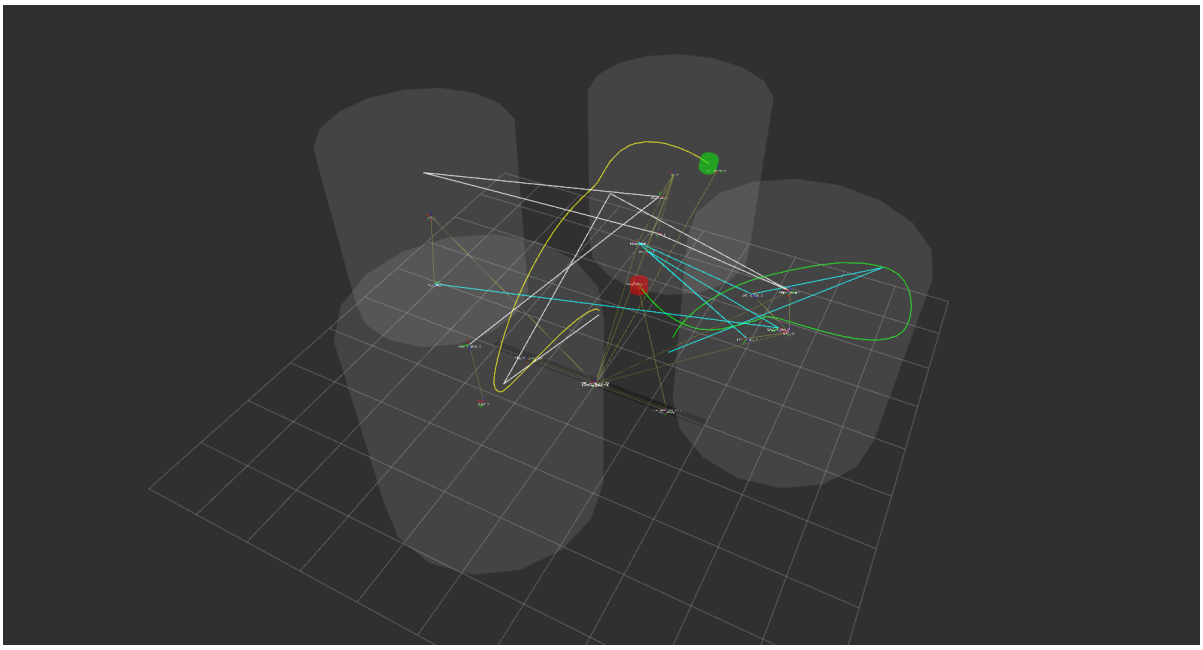

Figure 5.7 Four black cylinders are placed on the corners of a figurated square. On each, several paths are randomly placed to form the goal path of each one of the drones. From the UAVs, represented as coloured cylinders, its followed path depicts the movements accomplished for approaching the targets while avoiding close the close UAV and the obstacle.

80 simulations are executed and its data stored. Back on the Machine Learning research framework, the world, mission and other hyperparameters such as teacher algorithm or teacher or as path are selected. A list of architectures is defined to make different pieces of training for comparison. The batch size is set to 500, the learning rate to 0.01 and the number of steps at 250000. After the training process, Tensorboard is employed to understand what has happened. Figure 5.8 depicts some examples of the cost function value over the whole iterations during the training. The final cost value compared is the smoothed one given it represents the average and is not so affected by the high variability over iterations.

Structures with 2 hidden layers present a quite similar final cost, although the one with 30 neurons per layer has slightly better performance: 0.01696 against a 0.01706 which are translated into test validation accuracy values of 91.5% and 91.3% respectively. Those results would be acceptable compared with the performance found on the SoA on this kind of applications, but further tuning is to be applied towards optimisation.

Switching now to the 3 three hidden layered, 10-20-10 structure gets a value of 0.01985, and 20-30-20, 0.01906 with test validation accuracies of 90.1% and 90.4%. The deepen on the structure has surprisingly turned out to mean worse results. On an attempt to improve them, the steps parameter is moved from 250000 to 500000, as seen in Figure 5.9, with a clear improvement of 0.01588 and 0.01539 for the cost and 92.1% and 92.2% for the test validation accuracy. Those values are quite more acceptable, which allows settling this structure as a good candidate for the final use.

Complicating now a bit the scenario, an object is placed on the centre of the area where the two drones are moving. That obstacle is randomly placed inside a cylinder. Furthermore, the former cylinders are used no more and are substituted by a torus concentric to the cylinder of the obstacle. As before, random waypoints

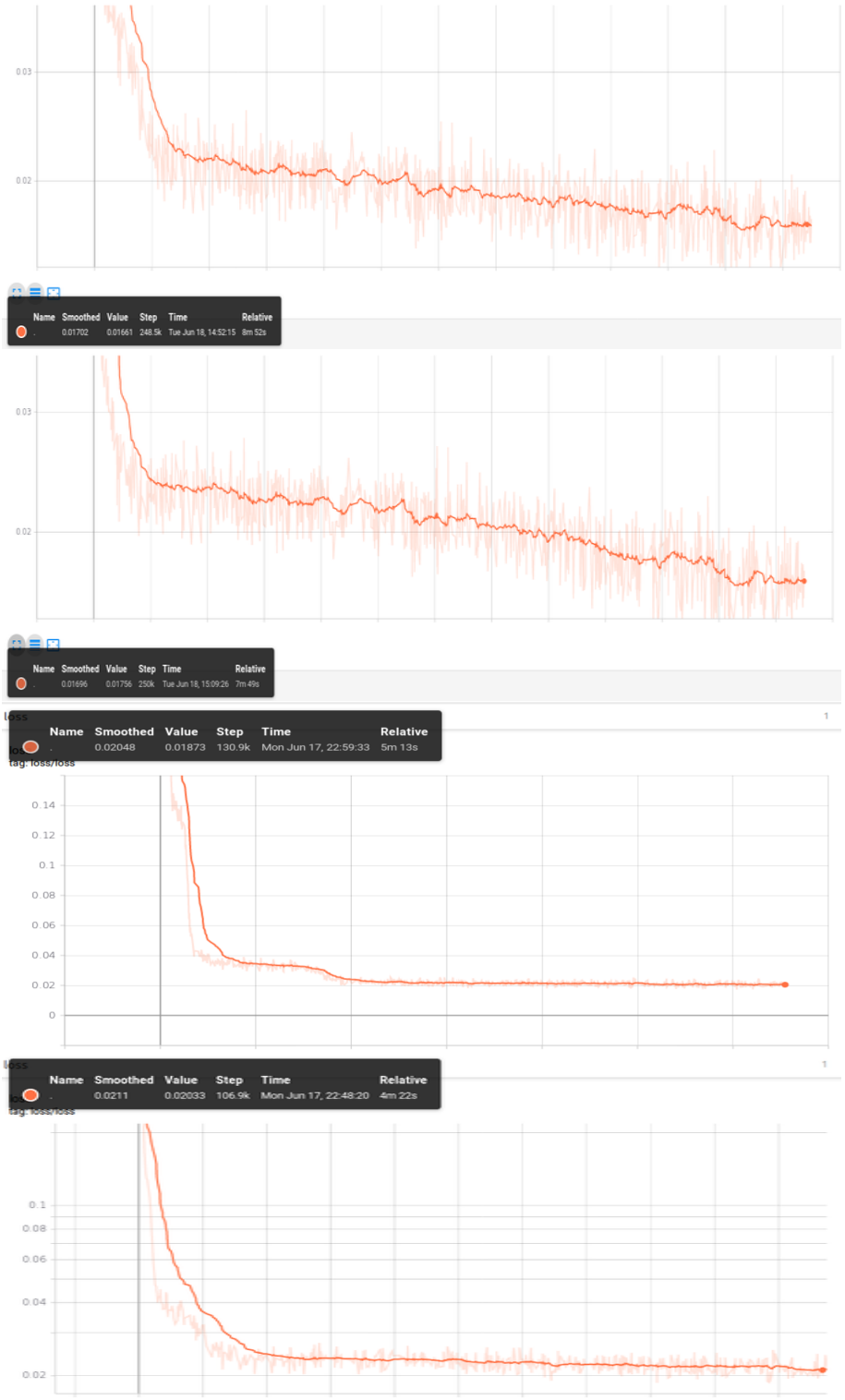

Figure 5.8 Loss functions for different architectures in descendent order inside the lists. LR:0.01, Layers: [10-10,30-30,10-20-10,20-30-20], Steps:250000, final loss: [0.01696, 0.01706, 0.01985, 0.01906] and test accuracy: [90.1%, 91.3%, 90.1%, 90.4%].

Figure 5.9 Loss functions for different architectures in descendent order inside the lists. LR:0.01, Layers: [10-20-10,20-30-20], Steps:500000, final loss: [0.01588, 0.01539] and test accuracy: [92.1%, 92.2%].

inside the tours form the path for each drone, see Figure 5.10. ORCA algorithm is used to avoid the other UAV and the obstacle at the same time.

As on the prior setup, two-hidden layered structures are tested with 100 inputs for batch and a learning rate of 0.01. Picture Figure 5.11 shows the results. The one with ten neurons is trained over 250000 steps with a poor result of 0.02068 of loss and an accuracy of 85.2% as expected given the increased difficulty of the avoining problem due to the object. That value must be uplifted after using it on a final application. The next research is done by incrementing to 30 neurons and 500000 steps, with an improvement to 0.01942 in loss and 85.6% in the test validation.

After a few iterations, the performance has reached values of 0.01678 in loss and 89.0% in test accuracy as depicted in Figure 5.12. The changes accomplished are a learning rate from 0.1 and a structure of 10-20-10. Furthermore, the most optimal network found, a 20-30-20 throws a 90.1% accuracy and a loss of 0.01101.

To put an end on the Machine Learning research example, a very interesting Tensorboard tool is shown. No further investigation has been made based on it, but it would be exceptionally useful the next steps of optimisation when a better insight into what is happening to the data throughout the network. Figure 5.13 depicts histograms of bias and weights over iterations so its displacement is accessively understood.

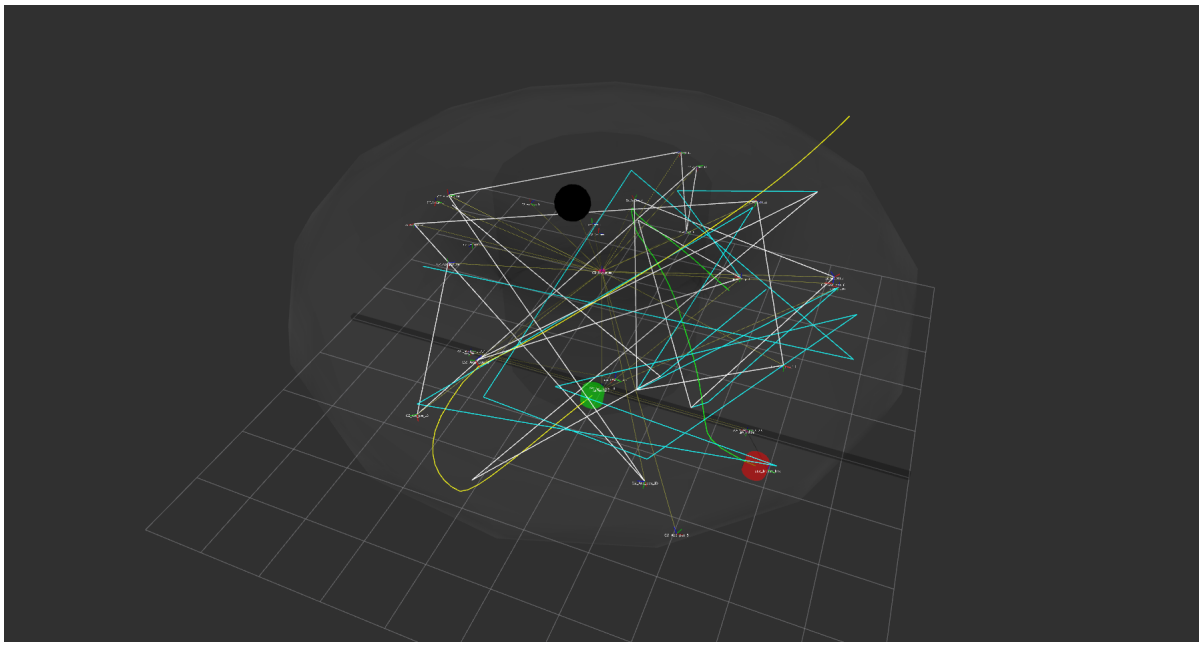

Figure 5.10 Gym world with an obstacle. A black cylinder is placed on the centre within a randomly located obstacle. A concentric white torus limits the possible random waypoints that form the target paths of two UAVs. From the UAVs, represented as coloured cylinders, its followed path depicts the movements accomplished for approaching the targets while avoiding close the close UAV and the obstacle.

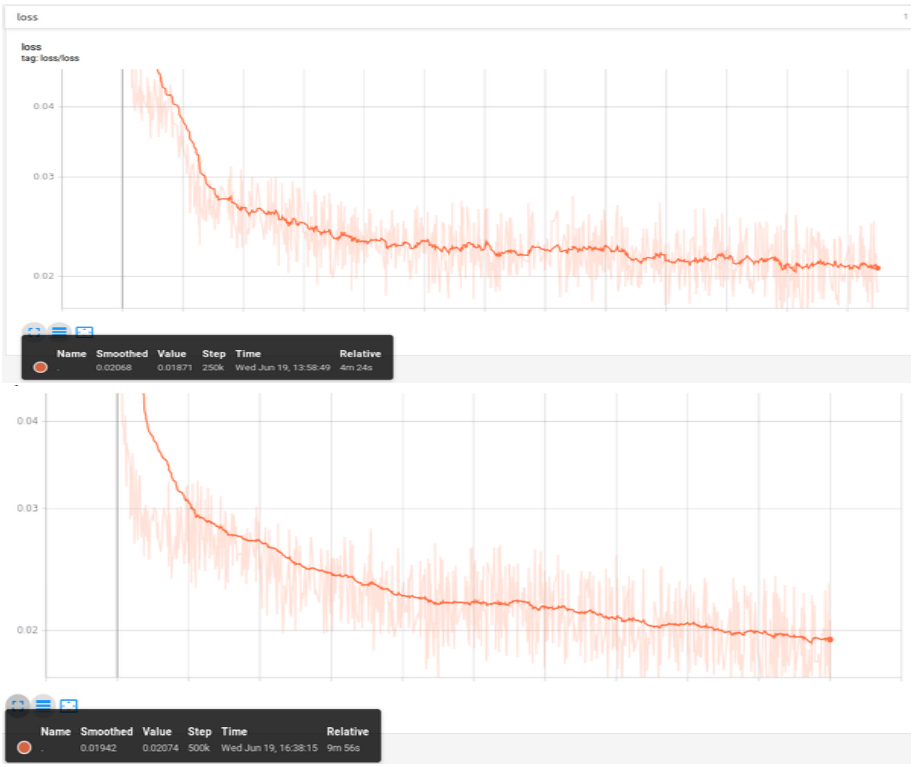

Figure 5.11 Loss functions for different architectures in descendent order inside the lists. LR:0.01, Layers: [10-10,30-30], Steps:[250000,500000], final loss: [0.02068, 0.01942] and test accuracy: [85.2%, 85.6%].

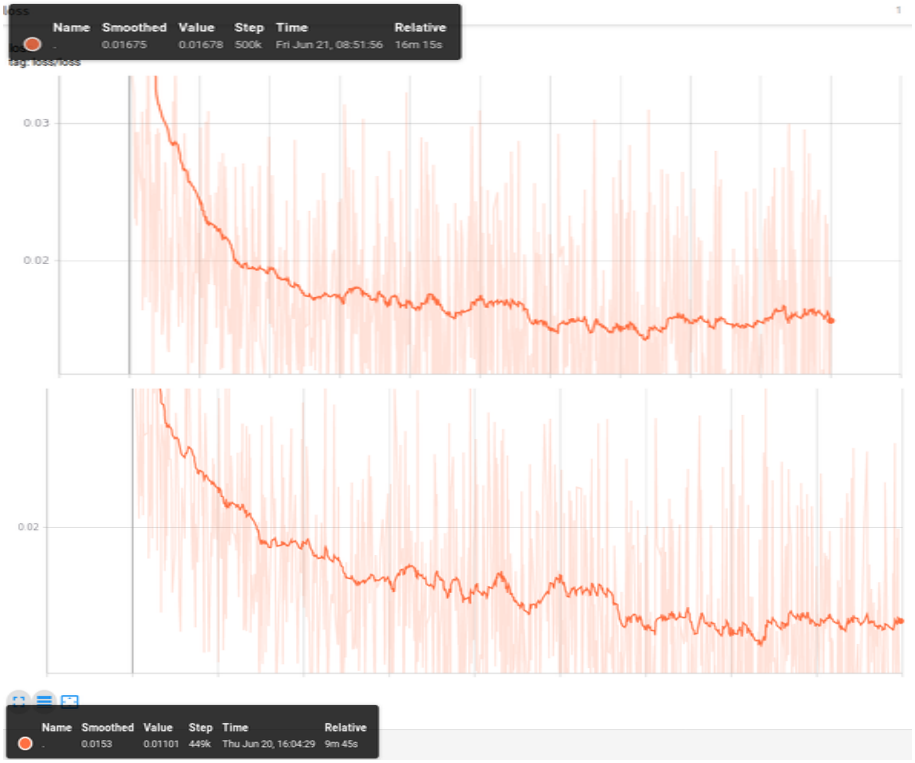

Figure 5.12 Loss functions for different architectures in descendent order inside the lists. LR:0.1, Layers: [10-20-10,20-30-20], Steps:500000, final loss: [0.01678, 0.01101] and test accuracy: [89.0%, 90.1%].

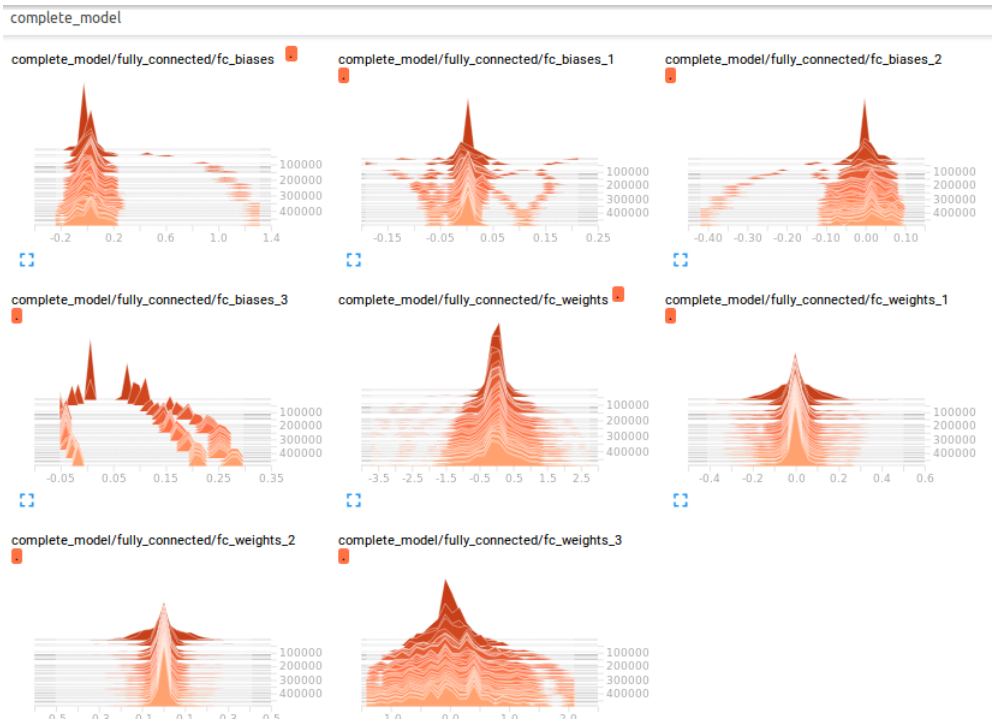

Figure 5.13 Histograms of weights and biases of the optimal found architecture retrieved from Tensorflow.

6 Voice Control Assistant

6.1 Introduction

Sometimes, the control of a mission must be accomplished while the pilot, or more generally, the commander, is executing any other task at the same time. It means a high loss of efficiency if there is a need to be thinking about a complex coded language, and even been typing it on any graphical interface, while a critical situation that requires quickness and lucidity.

With this situation in mind, the goal of this part of the project is to get an abstraction while commanding. A natural language commanding tool has been developed that may be used via the keyboard on a terminal or simply with the voice. This way, the commander may be in charge of the mission by visualizing its current state thanks to the visualization tools, described inside the ROS-MAGNA part, while commanding via oral.

Some challenges must be accomplished to reflect a voice command into a change on the mission as expressed on Figure 6.1. First of all, constant listening must be done to perceive when a sound, i.e. the voice, is spoken and keep listening until some period of silent. That sound must be recorded and saved for its treatment. The speech must be translated into text from which metainformation, containing the main purpose and the details provided. Consequently, that information may be used to configure as a command understandable by ROS-MAGNA. Therefore, a ROS node must be created to send the service.

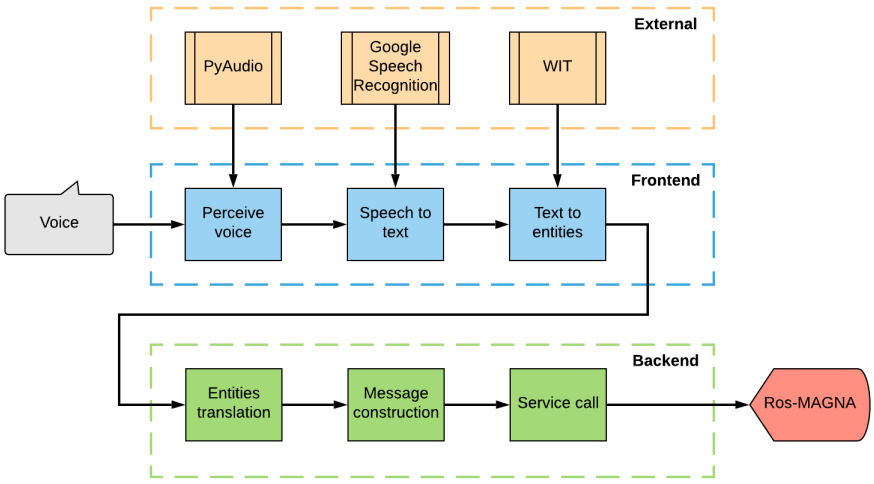

Figure 6.1 General pipeline for the voice control command. The first stage is the FrontEnd that receives the voice command and transforms it employing external libraries. That information is provided to the Backend that sends it to ROS-MAGNA.

6.2 Designed WIT app

The information contained in the voice or text command needs to be extracted to a structure understandable by a machine. Training a neural network for Natural Language Processing would have been a good option since the focus of this project is to dig into Machine Learning knowledge. However, a quicker way has been chosen initially to act as a bridge for this step.

WIT, like some other platforms, offer a pre-trained model for free. This model already extracts general information, but a deeper knowledge of the singular application must be provided by the user. This way, before using their Python API, the online database must be taught with a group of sentences whose main information is extracted so the model learns how to extrapolate the process from those examples. Figure 6.2 presents some of those examples extracted from the WIT page of the application explicitly created for this project.

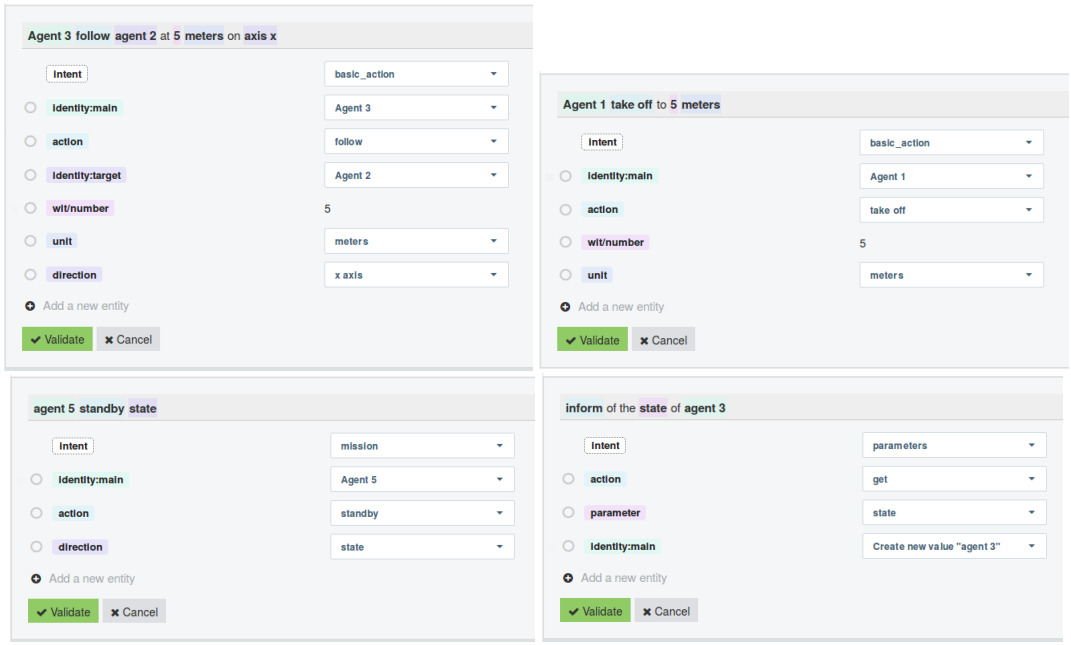

Figure 6.2 Teaching sentences examples. For each one, the top sentence is the introduced one by the user and below a list splitting the different information fields and its component on the sentence.

That information extracted is classified into "entities" whose possible values are already defined by the user. The most important entity is the "intent" and gives an overall intuition of the topic the speech is about. For a better comprehension of how WIT works, Table 6.1 shows all the entities implemented and its possible values.

6.3 Frontend

6.3.1 Introduction

The Frontend is implemented as a Python class with no ROS-related element on it. Its responsibility is to act as the interface with the user, leaving to the backend the ROS part of talking to compound the message and send it to the Ground Station node.

Initially, the backend is initialised and stays awaiting for incoming commands using PyAudio library. Furthermore, the WIT client is spawned, identifying the account that has already been pretrained online for the purpose of this project. The last initialisation is speech recognizer provided by the Google library. As may be appreciated, the Frontend is a chain of external libraries uses more than a new implementation.

Everything is ready now to open the user interface and start interacting with it so, depending on the mode chosen at the execution call, whether an interactive keyboard WIT client is used or the audio must start to be perceived.

Table 6.1 WIT designed entities in which the pieces of information of the sentences are mapped, its intended use and the possible values it may adopt at the time of writing.

Entities	Definition	Values
Intent	Overall topic	Basic action, mission, parameters, algorithms.
Identity	UAV to command	All UAVs, UAV 1, UAV 2...
Action	Type of action to accomplish	Get, set, resume, standby, follow, turn, move, land, take off.
Mission component	Element of mission to alter	State.
Direction	Axis	Up, down, left, right, forward, backward, x axis, y axis, z axis.
wit/number	WIT built-in. Used for quantities	Any
Unit	Defines the unit of a number	Meters, centimetres, degrees.
Parameter	Additional information	Battery level, state.

6.3.2 Speech recognition process

On the case, that voice control was the chosen one, the first assignment to fulfil is to start listening. This step is achieved thanks to the PyAudio library used by a function is not of my authorship but of anonymous user found over the internet. My implementation is not further than a few changes for adaptation. That function receives another callback function that is called whenever a new complete sound has been detected providing the path where it has been stored on the memory, called MessageFromAudio.

The MessageFromAudio function employs the Google Speech Recognition library to create its own audio file object and, sourcing it, directly provides the recognized speech to text result.

Finally, the text is given to the WIT API that splits it into significative information with its online pretrained model. However, that information contains data that is not relevant for this application, so only the entities are used and presented to the Backend

6.4 Backend

6.4.1 Introduction

The Backend is a ROS node created for communications with ROS-MAGNA. Hence, the metainformation received from WIT must be understood, divided into meaningful pieces of data, reconstructed and sent. This is a straight forward process that is performed every time that the Frontend detects a new voice command or a keyboard input.

6.4.2 Entities to command

The function for a new command directly implements the three steps above defined and interconnects their outputs to inputs as a pipeline.

Entities translation

Provided the message entities returned by WIT, the first task is to retrieve the main intent of the command. For that purpose, it is implemented a check of the existence of that information, given the fact that the message would be understandable noise and no practical information was extracted.

The second important data is the agent or UAV that must accomplish the action, contained in the "main" entity. It is imperative that these two fields are known, otherwise, no message may be interpreted. With them, a dictionary defining the message is built and later updated.

The information contained inside the entities is translated into the message dictionary. This dictionary is employed as a standard way to cope with different kinds of commands. For every entity, WIT offers a sort of possible value options with the credibility it gives to each of them. Obviously, this function extracts the one with the highest credibility and if over a minimum thresh.

Ground Station service message construction

The main fields of the service message are the intent, the action and the UAV that must accomplish the command. This information was added to the dictionary at the beginning of the prior task.

The action must be supplemented with several parameters whose value may be an integer or a string. Consequently, the command dictionary information must be decoupled. It is performed straightway by checking the incoming state of the value of each one of the parameters.

Three lists for the parameters are appended to the message, all of them with the same length as indexes relate the parameter. The first one contains the name of the parameters and the rest, for integer and string values. When the parameter corresponds to an integer, its corresponding element on the string list is fulfilled with "as int".

Service call

As it was common in the ROS-MAGNA chapter, ROS services are commonly used to send a request and quick response between nodes. That process is conceived to be executed once. This is the reason why the voice/keyboard control command is implemented with a service: this kind of commands must be sent only one time and the server must remaining accomplishing the task while the user continues controlling the mission.

The fulfilment of the sent is simply waiting for the service to be active, sending the already created message and waiting for the response from the server. Then, the whole process may be restarted, and the Backend remains inactive until another command is required to be sent.

7 Conclusions and future work

7.1 Conclusions

On this project, three work frames for cooperative mission control, machine learning research and voice control have been studied, designed and implemented. Furthermore, they have been fully integrated with each other and are operatively working.

ROS-MAGNA has provided a broad solution for the definition, visualisation command and control of cooperative missions of multiple UAV platforms commanded from a central ground station over ROS. In addition, a world modelling tool is capable of describing the environment with enough precision to make its information accessible for the mission definition but with simplifications for computation lightening. Furthermore, an interface is used to rule the commanded behaviours of the drones in order to optimise its performance while achieving the goals. Finally, a wide range of possible configurations of the drones autopilots, that may be simulated on software-in-the-loop, hardware-in-the-loop and real, making ROS-MAGNA versatile for the different tests that the algorithms need to pass when on development.

A Machine Learning research framework has been successfully developed. It supports all the stages that the process of training a neural network requires. Primarily, data from ROS-MAGNA missions is retrieved, parsed into NN inputs-like vectors and pretreated. Next, the Tensorflow graph is built following predefined rules and using utilities for later data understanding. Finally, the data is fed to the graph and the neural network is trained. The final optimised state is used back in ROS-MAGNA to accomplish missions. The whole process is highly customizable and presents adaptations to the behaviours and scalability of ROS-MAGNA applications.

Finally, the voice control assistant has demonstrated its utility as it lets the user send a set of commands to the ground station or specific drones. Several state-of-the-art online and offline libraries have been used to implement the process that transforms the sound into meaningful pieces of information understandable by a machine. That data is fed to a backend that maps it into ROS-MAGNA-like commands that and sends to the ground station and distributed to its concerning UAV.

7.2 Future work

As has been the main guideline, the next steps to overtake are also split into the different parts of the project, although most of them have a transversal influence.

ROS-MAGNA strongly claims for a graphical user interface that lets him better understand and manipulate the so wide range of parameters required when a complex mission is designed. This GUI should offer different utilities not only for mission and main features selection but also the mission and world graph designers Otherwise, frontend script and JSONs become an arduous environment where the perspective is easily lost even for its designer. Moreover, new states concerning new behaviours and the addition of subparameters for the states for the control, for instance, of sensors, would be very beneficial for the polyvalence of ROS-MAGNA on the projects it is currently working on and for future broadening use. About world modelling, new types of logical set poses would increase the versatility of tasks accomplished on missions. In addition, the World node must get independent of the Ground Station node so the changes and information retrievement may be accomplished at any time of the mission and by any one of the agents on the scene.

Machine Learning research is the most suitable part of the project to be improved. More types of neural networks and libraries, such as Caffe (substitute of Tensorflow) and Keras o Pytorch (over Tensorflow or Caffe) are a very interesting option as they facilitate the implementation of state-of-the-art nets and the acquisition and use of databases. Of course, more behaviours would mean more definitions of inputs and outputs for the nets, but with the current framework, it would be quite easy to implement.

Finally, not only new commands are the potential improvements of the voice control assistance. It would be very interesting the use of offline libraries that make the speech to text and natural language processing much more efficient and application specific. That directly derives in a much shorter answer time, especially important for critical missions.

Bibliography

[1] *Airmap*, https://www.airmap.com/.

[2] *Corus project*, https://www.sesarju.eu/projects/corus.

[3] *Dl applications*, https://paperswithcode.com/sota.

[4] *Gauss project*, https://www.projectgauss.eu/.

[5] *Gazebo web page*, http://gazebosim.org/.

[6] *Google cloudspeech*, https://cloud.google.com/speech-to-text.

[7] *Mission planner*, http://ardupilot.org/planner/.

[8] *Python web page*, http://www.python.org/doc/essays/blurb/.

[9] *Qgroundcontrol*, http://qgroundcontrol.com/.

[10] *Ros web page*, http://www.ros.org/about-ros/.

[11] *Safedrone project*, https://www.sesarju.eu/node/3199.

[12] *Unifly*, https://www.unifly.aero/.

[13] *Wikipedia nlp*, https://en.wikipedia.org/wiki/Natural language processing.

[14] *Wit*, https://wit.ai/.

[15] *Wit web page*, https://wit.ai/.

[16] *Mavros – mavlink extendable communication node for ros with proxy for ground control station*, *http://wiki.ros.org/mavros*, 2019.

[17] *RViz - 3D visualization tool for ROS*, 2019, *http://wiki.ros.org/rviz*, Last accessed on 2019-02-26.

[18] *SMACH - a task-level architecture for rapidly creating complex robot behavior in ROS*, 2019, *http://wiki.ros.org/smach*, Last accessed on 2019-02-26.

[19] *SMACH Viewer - a GUI that shows the state of hierarchical SMACH state machines in ROS*, 2019, *http://wiki.ros.org/smach_viewer*, Last accessed on 2019-02-26.

[20] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015, Software available from tensorflow.org.

[21] David Amos, *Speech recognition guide*, https://realpython.com/python-speech-recognition/.

[22] A. R. Cantieri, A. S. de Oliveira, M. A. Wehrmeister, J. A. Fabro, and M. de Oliveira Vaz, *Environment for the dynamic simulation of ros-based uavs*, Studies in Computational Intelligence, vol. 707, 2017.

[23] J. Dentler, S. Kannan, M. A. Olivares-Mendez, and H. Voos, *Implementation and validation of an event-based real-time nonlinear model predictive control framework with ros interface for single and multi-robot systems*, 1st Annual IEEE Conference on Control Technology and Applications, CCTA 2017, vol. 2017-January, 2017, pp. 1000–1006.

[24] Jeffrey L. Elman, *Finding structure in time*, COGNITIVE SCIENCE **14** (1990), no. 2, 179–211.

[25] V. Grabe, M. Riedel, H. H. Bulthoff, P. R. Giordano, and A. Franchi, *The telekyb framework for a modular and extendible ros-based quadrotor control*, 2013 European Conference on Mobile Robots, ECMR 2013 - Conference Proceedings, 2013, pp. 19–25.

[26] H. Hayakawa, T. Azumi, A. Sakaguchi, and T. Ushio, *Ros-based support system for supervision of multiple uavs by a single operator*, Proceedings - 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, 2018, pp. 341–342.

[27] Wolfgang Hönig and Nora Ayanian, *Flying multiple uavs using ros*, pp. 83–118, Springer International Publishing, Cham, 2017.

[28] E. Kakaletsis, M. Tzelepi, P. I. Kaplanoglou, C. Symeonidis, N. Nikolaidis, A. Tefas, and I. Pitas, *Semantic map annotation through uav video analysis using deep learning models in ros*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 11296 LNCS, 2019.

[29] P. Kremer, J. Dentler, S. Kannan, and H. Voos, *Cooperative localization of unmanned aerial vehicles in ros - the atlas node*, Proceedings - 2017 IEEE 15th International Conference on Industrial Informatics, INDIN 2017, 2017, pp. 319–325.

[30] Dmitriy Kuragin, *apiai*, https://dialogflow.com/.

[31] A. P. Lamping, J. N. Ouwerkerk, N. O. Stockton, K. Cohen, M. Kumar, and D. W. Casbeer, *Flymaster: Multi-uav control and supervision with ros*, 2018 Aviation Technology, Integration, and Operations Conference, 2018.

[32] James Le, *Medium, nn architectures*, https://medium.com/cracking-the-data-science-interview/a-gentle-introduction-to-neural-networks-for-machine-learning-d5f3f8987786.

[33] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE **86** (1998), no. 11, 2278–2324.

[34] B. H. . Lee, J. R. Morrison, and R. Sharma, *Multi-uav control testbed for persistent uav presence: Ros gps waypoint tracking package and centralized task allocation capability*, 2017 International Conference on Unmanned Aircraft Systems, ICUAS 2017, 2017, pp. 1742–1750.

[35] MultiMedia LLC, *MS Windows NT kernel description*, 1999.

[36] R. Mendonça, P. Santana, F. Marques, A. Lourenço, J. Silva, and J. Barata, *Kelpie: A ros-based multi-robot simulator for water surface and aerial vehicles*, Proceedings - 2013 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2013, 2013, pp. 3645–3650.

[37] W. Meng, Y. Hu, J. Lin, F. Lin, and R. Teo, *Ros+unity: An efficient high-fidelity 3d multi-uav navigation and control simulator in gps-denied environments*, IECON 2015 - 41st Annual Conference of the IEEE Industrial Electronics Society, 2015, pp. 2562–2567.

[38] Diego Murray, *Medium, machine learning*, https://medium.com/datadriveninvestor/what-is-machine-learning-55028d8bdd53.

[39] Augustus Odena, Catherine Olsson, David Andersen, and Ian J. Goodfellow, *Tensorfuzz: Debugging neural networks with coverage-guided fuzzing*, Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA, 2019, pp. 4901–4911.

[40] Hubert Pham, *Pyaudio*, https://pypi.org/project/PyAudio/, 2006.

[41] Dmitry Prazdnichnov, *pocketsphinx*, https://pypi.org/project/pocketsphinx/.

[42] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng, *ROS: an open-source Robot Operating System*, ICRA Workshop on Open Source Software, 2009.

[43] Fran Real, Arturo Torres-Gonzalez, Pablo Ramon-Soria, Jesus Capitan, and Anibal Ollero, *UAL: An abstraction layer for unmanned aerial vehicles*, Proceedings of the 2nd International Symposium on Aerial Robotics (Philadelphia, USA), June 2018.

[44] Juan Jesús Roldán, Elena Peña-Tapia, David Garzón-Ramos, Jorge de León, Mario Garzón, Jaime del Cerro, and Antonio Barrientos, *Multi-robot systems, virtual reality and ros: Developing a new generation of operator interfaces*, pp. 29–64, Springer International Publishing, Cham, 2019.

[45] José Andrés Millán Romera, *Ros-magna github page*, https://github.com/JoseAndresMR/ros magna.

[46] F. Rosenblatt, *The perceptron: A probabilistic model for information storage and organization in the brain*, Psychological Review (1958), 65–386.

[47] M. E. Sayed, M. P. Nemitz, S. Aracri, A. C. McConnell, R. M. McKenzie, and A. A. Stokes, *The limpet: A ros-enabled multi-sensing platform for the orca hub*, Sensors (Switzerland) **18** (2018), no. 10.

[48] Jur van den Berg, Stephen J. Guy, Ming Lin, and Dinesh Manocha, *Reciprocal n-body collision avoidance*, 2011, pp. 3–19.

[49] Y. Yu, X. Wang, Z. Zhong, and Y. Zhang, *Ros-based uav control using hand gesture recognition*, Proceedings of the 29th Chinese Control and Decision Conference, CCDC 2017, 2017, pp. 6795–6799.