

UNIVERSIDAD POLITÉCNICA DE MADRID
Escuela Técnica Superior de Ingenieros Industriales



Autonomy in Aerial Robotics: Architecture and Algorithms for Real-World Applications

DOCTORAL THESIS

Submitted for the degree of Doctor by:

Miguel Fernández Cortizas
M.Sc. in Automation and Robotics

Madrid, 2025



UNIVERSIDAD POLITÉCNICA DE MADRID
Escuela Técnica Superior de Ingenieros Industriales

Doctoral Degree in Automatic Control and Robotics

Autonomy in Aerial Robotics: Architecture and Algorithms for Real-World Applications

DOCTORAL THESIS

Submitted for the degree of Doctor by:

Miguel Fernández Cortizas

M.Sc. in Automation and Robotics

Under the supervision of:
Dr. Pascual Campoy Cervera

Madrid, 2025

Title: Autonomy in Aerial Robotics: Architecture and Algorithms for Real-World Applications

Author: Miguel Fernández Cortizas

Doctoral Programme: Automatic Control and Robotics

Thesis Supervision:

Prof. Pascual Campoy Cervera, Dr. in Automation and Robotics (Supervisor)

External Reviewers:

Thesis Defense Committee:

Thesis Defense Date:

This thesis has been partially supported by the European Union under the Horizon Europe Programme through the project CORESENSE: A Hybrid Cognitive Architecture for Deep Understanding (Ref. 101070254); from the Spanish Ministry of Science, Innovation and Universities (MICIU/AEI/FEDER, EU) through the project COMCISE (Ref. RTI2018-100847-B-C21); from the Madrid Government under the R&D Synergic Projects Program through the project COPILOT – Control, Monitoring and Operation of Photovoltaic Solar Power Plants (Ref. Y2020/EMT6368); from MICIU/AEI/10.13039/501100011033 and FEDER, EU through the project INSERTION (Grant Ref. ID2021-127648OBC32); and from the Fonds National de la Recherche of Luxembourg (FNR), under the project C22/IS/17387634/DEUS.

Título:

**Autonomy in Aerial Robotics: Architecture and Algorithms for
Real-World Applications**

Autor:

Miguel Fernández Cortizas
M.Sc. in Automation and Robotics
B.Sc. in Industrial Engineering (Esp. Electronics and Automation)

Tribunal nombrado por el Mgfco. y Excmo. Sr Rector de la Universidad Politécnica de Madrid el día.....de.....de 2025.

Tribunal

Presidente: NOMBRE APELLIDO

Vocal: NOMBRE APELLIDO

Vocal: NOMBRE APELLIDO

Vocal: NOMBRE APELLIDO

Secretario: NOMBRE APELLIDO

Suplente: NOMBRE APELLIDO

Suplente: NOMBRE APELLIDO

Realizado el acto de lectura y defensa de la tesis el día de de 2025.

Calificación de la Tesis.....

El Presidente:

Los Vocales:

El Secretario:

A mis padres, por enseñarme y apoyarme para perseguir todas mis metas.

A mi hermana, por motivarme con su tenacidad y ejemplo de trabajo duro.

A mi esposa Carmen, por su infinito cariño y apoyo.

*A todos los compañeros y amigos que han estado ahí,
ayudándome y apoyándome durante estos largos años.*

– Miguel

Acknowledgments

*If I have seen further it is by standing on the
shoulders of Giants.*

Isaac Newton

The work for my doctoral research was carried out at the Center for Automation and Robotics (CAR) at the Universidad Politécnica de Madrid (UPM), within the Computer Vision and Aerial Robotics (CVAR) research group.

First of all, I would like to thank the director of my doctoral thesis, Prof. Pascual Campoy. I am profoundly grateful to you for giving me the opportunity to explore and discover a truly fascinating field of science, one in which I have found great joy throughout these years. I also wish to express my sincere gratitude for the financial and strategic support you have provided, as well as for granting me the opportunity to participate in key projects and international competitions. These experiences have driven me to grow as a student, a researcher, and an academic. I would like to thank Prof. Ricardo Sanz for the opportunity to broaden my engineering, scientific, and even philosophical thinking—both through our collaborative work and during our informal coffee conversations. I also want to acknowledge the valuable support of Prof. Martín Molina for his advice and recommendations throughout the development of this thesis.

As the proverb says, *“If you want to go far, go together”*. This doctoral thesis would not have been possible without the outstanding collaboration of brilliant people who have contributed to its success. I am profoundly proud to be a member of the CVAR group, which embodies not only academic excellence but also an invaluable, friendly environment where colleagues become true friends. I especially want to thank my research colleagues, with whom I have shared countless hours of work. To David Pérez, who has been there since the very beginning of this journey—learning and growing together. To Rafael Pérez, for his dedication and constant motivation to push us further and strive for more. To Pedro Arias, for his brilliance and his ability to make us think outside the box. I also want to thank Javier Melero, Guillermo García-Patiño, Rodrigo da Silva, Carmen de Rojas, and Francisco J. Anguita—although we only shared the last years of this path, your fresh ideas and energy inspired me to keep working hard. I must also express my deep gratitude to the former members of the group who first motivated me to embark on this adventure: F. Javier Rodríguez, Alejandro Rodríguez, Carlos Sampedro, Ramón Suárez, and Adrián Carrio.

I would also like to mention and thank the members of the Automation and Robotics Research Group (ARG) at the University of Luxembourg, where I conducted my research stay. I am grateful to José Luis Sánchez and Holger Voos for warmly welcoming me and for sharing the outstanding work they have been carrying out over the years. I also want to thank Hriday Bavle for his active guidance throughout my stay, as well as Ali Tourani, Muhammad Shaheer, José Andrés Millán, and the rest of the research group for their kindness and support during my time there.

I am deeply grateful to my family and friends for their unconditional support—especially my parents, Jorge and Luisa; my little sister, Laura; my amazing wife, Carmen; and her parents, Pedro and Manuela.

I want to thank all the institutions that have provided funding for the realization of this thesis and its related works. I am pleased to acknowledge the financial support received from the European Union under the Horizon Europe Programme through the project CORESENSE: A Hybrid Cognitive Architecture for Deep Understanding (Ref. 101070254); from the Spanish Ministry of Science, Innovation and Universities (MICIU/AEI/FEDER, EU) through the project COMCISE (Ref. RTI2018-100847-B-C21); from the Madrid Government under the R&D Synergic Projects Program through the project COPILOT – Control, Monitoring and Operation of Photovoltaic Solar Power Plants (Ref. Y2020/EMT6368); from MICIU/AEI/10.13039/501100011033 and FEDER, EU through the project INSERTION (Grant Ref. ID2021-127648OBC32); and from the Fonds National de la Recherche of Luxembourg (FNR), under the project C22/IS/17387634/DEUS.

Finally, I would also like to thank all those who, although not mentioned by name, have supported me throughout this journey—whether academically, administratively, or simply by being there when it mattered most.

Resumen

Los sistemas robóticos han avanzado significativamente su autonomía en los últimos años gracias a los avances en percepción, control y toma de decisiones. Si bien estos sistemas operan de manera efectiva en entornos conocidos y estructurados, su implementación en escenarios dinámicos y desconocidos continúa siendo un desafío debido a limitaciones en robustez y adaptabilidad.

La motivación de esta tesis surge de la creciente necesidad de desplegar robots autónomos en entornos complejos del mundo real, donde los sistemas tradicionales suelen enfrentar dificultades. En particular, la robótica aérea ha adquirido un papel cada vez más relevante en aplicaciones como la inspección industrial, la búsqueda y rescate, y el mantenimiento de infraestructuras. Sin embargo, las limitaciones en términos de escalabilidad, adaptabilidad y robustez motivan el desarrollo de tecnologías que aumenten la autonomía de estos sistemas.

El avance en sistemas autónomos requiere el desarrollo conjunto de arquitecturas robóticas y algoritmos especializados. Mientras que el diseño de arquitecturas se centra en la versatilidad, la robustez y la escalabilidad, los avances algorítmicos optimizan el rendimiento en tareas específicas como la planificación, la estimación del estado y la toma de decisiones. Esta tesis explora ambos aspectos en paralelo, lo que permite mejorar la eficiencia y el rendimiento de los sistemas robóticos.

En esta tesis, se presenta una arquitectura de referencia en capas, enfocándose en cómo ocurren los procesos de toma de decisiones en distintos sistemas robóticos. Este marco permite analizar sistemas robóticos con distintos niveles de complejidad. Dentro de esta arquitectura, se desarrollan dos conceptos clave para mejorar la autonomía: el **entendimiento**, que se refiere a la capacidad del robot para interpretar y modelar la realidad, y **conciencia**, que define su capacidad para evaluar la calidad de una situación y tomar decisiones. Estos principios pueden extenderse a sistemas multi-robot a través de la **conciencia colectiva**, lo que permite mejorar la comunicación y la coordinación en escenarios complejos.

Sobre esta base arquitectónica, se desarrolla el marco de software de código abierto **Aerostack2** para el desarrollo de sistemas robóticos aéreos. Aerostack2 es una plataforma modular y escalable que admite la interoperabilidad entre diversas plataformas aéreas e integra funcionalidades avanzadas como planificación de misiones, control en tiempo real y coordinación de enjambres. Su diseño está orientado tanto a la investigación como a aplicaciones industriales, facilitando el despliegue de sistemas robóticos aéreos en distintos dominios operacionales.

Sumado a los avances en arquitectura, se presentan dos algoritmos innovadores. El primero es un **algoritmo de modificación dinámica de trayectorias en tiempo real**, que optimiza las rutas de los UAV y ajusta las trayectorias sobre la marcha, garantizando una navegación ágil en entornos que cambian rápidamente y manteniendo un bajo costo computacional. El segundo es un **algoritmo de SLAM colaborativo distribuido**, que aprovecha la información semántico-relacional para mejorar la precisión en la localización y optimizar el intercambio de datos entre múltiples robots, lo que mejora la escalabilidad y el rendimiento general del sistema.

Mediante una validación exhaustiva en simulaciones, experimentos en el mundo real, competencias internacionales y aplicaciones industriales como inspecciones industriales, las contribuciones de esta tesis demuestran **mejoras en la autonomía, adaptabilidad y operación colaborativa de sistemas robóticos**. Al integrar avances en arquitectura de sistemas con métodos algorítmicos

micos del estado del arte, esta investigación establece una base sólida para la próxima generación de sistemas robóticos autónomos capaces de operar de manera confiable en entornos cada vez más complejos.

Abstract

Autonomous robotic systems have made significant progress in recent years due to advancements in perception, control, and decision-making. While these systems operate effectively in structured environments, their deployment in dynamic scenarios remains challenging due to limitations in robustness and adaptability.

The motivation for this thesis stems from the growing need to deploy autonomous robots in complex, real-world settings where traditional systems often struggle. In particular, aerial robotics have become increasingly utilized for applications such as industrial inspection, search and rescue, and infrastructure maintenance. However, the limitations of existing platforms—particularly in scalability, adaptability, and robustness—have become more evident.

Advancing autonomy requires the joint development of robotic architectures and specialized algorithms. While architecture design focuses on versatility, robustness, and scalability, algorithmic advancements optimize performance in specific tasks such as planning, state estimation, and decision-making. This thesis explores both aspects in parallel, providing advancements to the efficiency and performance of robotic systems.

A **novel layered reference architecture** is introduced, focusing on how decision-making processes occur within different robotic systems. This framework enables the analysis of robotic systems with varying levels of complexity. Within this architecture, two key concepts for improving autonomy are elaborated: **understanding**—the robot’s ability to interpret and model reality—and **awareness**—its capacity to evaluate the quality of a situation, leading to more informed decision-making. These principles can be naturally extended to multi-robot systems through **collective awareness**, which enhances communication and coordination in complex scenarios.

Building on this architectural foundation, the **Aerostack2** open-source software framework for aerial systems is developed. Aerostack2 is a modular, scalable platform that supports interoperability among diverse UAV platforms and integrates advanced functionalities such as mission planning, real-time control, and swarm coordination. Its design is tailored for both research and industrial applications, facilitating the deployment of aerial robotic systems across varied operational domains.

Complementing the architectural advances, two innovative algorithms are presented. The first is a **real-time dynamic trajectory modification algorithm**, which optimizes UAV routes and adjusts paths on the fly, ensuring agile navigation in rapidly changing environments while maintaining low computational overhead. The second is a **distributed collaborative SLAM algorithm**, which leverages semantic-relational information to enhance localization accuracy and streamline data exchange among multiple robots, thereby improving overall system scalability and performance.

Through extensive validation in simulations, real-world experiments, international competitions, and industrial applications such as UAV-based inspections, the contributions of this thesis demonstrate significant **improvements in autonomy, adaptability, and collaborative operation**. By integrating advances in system architecture with state-of-the-art algorithmic methods, this research establishes a robust foundation for next-generation autonomous robotic systems capable of reliable operation in increasingly complex environments.

Contents

Acknowledgments	ix
Resumen	xi
Abstract	xiii
List of Figures	xix
List of Tables	xxv

I Introduction and State of the Art	5
1 Introduction	7
1.1 Autonomous Robotic Systems	7
1.2 Thesis Motivation	8
1.3 Methodology	9
1.4 Objectives	11
1.5 Contributions	12
1.6 Outline	13
2 State of the art	15
2.1 Robotic System Architectures	15
2.1.1 Reactive architectures	15
2.1.2 Cognitive Architectures	16
2.2 Aerial Robotics Frameworks	17
2.2.1 Programmable Flight Control Unit	17
2.2.2 High-Level Frameworks for Aerial Robotics	18
2.3 Trajectory Generation for Agile Flights	21
2.4 Collaborative SLAM	22
II Architecture and Frameworks for Robotic Systems	25
3 Reference Architecture for Robotic Systems: An Incremental Approach	27
3.1 Introduction	27
3.2 Tier I: Feedback controlled systems	29
3.2.1 Hardware Interface Subsystem	29

3.2.2	State Estimation Subsystem	31
3.2.3	Motion Control Subsystem	31
3.3	Tier II: Tasks Decomposition	32
3.3.1	Behavior Execution Subsystem	34
3.3.2	Plan Execution Subsystem	34
3.4	Tier III: Cognitive Systems	35
3.4.1	Understanding in a knowledge base subsystem	37
3.4.2	Awareness by Mission Planning	38
3.5	Tier IV: Multi-Agent Systems	40
3.5.1	Communication Subsystem	42
3.5.2	Collective Awareness Subsystem	42
4	Aerostack2: the software framework for Aerial Robotic Systems	47
4.1	Introduction	47
4.2	Stakeholder Needs and Requisites Analysis	48
4.2.1	Functional Requirements Analysis (FR)	49
4.3	Framework Overview	50
4.3.1	Initial approach: Tier II+ framework	51
4.3.2	Extended version: Toward Understanding and Awareness	52
4.4	Sensor-Actuator Interface	54
4.5	Basic Robotic Functions	56
4.5.1	Motion Controller	56
4.5.2	State Estimator	59
4.6	Behaviors	61
4.7	Plan Execution Control	63
4.8	Mission control and supervision	65
4.8.1	System Development tools	65
4.8.2	End Users tools	66
4.9	Knowledge Base	66
4.9.1	Knowledge Server	67
4.9.2	Knowledge Extractors	68
4.10	Communications	68
4.11	Validation Examples	69
4.11.1	Synchronized drone gate crossing.	69
4.11.2	Search and Rescue Operation	71
III	Algorithms proposed	77
5	Dynamic trajectory following for UAVs	79
5.1	Introduction	79
5.2	Methodology	80
5.2.1	Notation	80
5.2.2	Problem formulation	80
5.3	Dynamic Trajectory Generation Algorithm Proposal	81
5.3.1	Generating Baseline Trajectories	82
5.3.2	Fast Trajectory Modification	83

5.4	Experiment and Results	85
5.4.1	Experimental setup	85
5.4.2	Validation of Fast Trajectory Modification	85
5.4.3	Simulation Flights	87
5.4.4	Real Flight Validation	88
5.5	Discussion	91
5.5.1	Validation of Fast Trajectory Modification	91
5.5.2	Simulation Flights	91
5.5.3	Real-World Experiments	91
5.6	Summary	92
6	Distributed Knowledge-Centric Collaborative SLAM	93
6.1	Introduction	93
6.2	System Overview	95
6.3	Local Situational Graph	96
6.4	Situational Graph Distillation	97
6.5	Room-Based Loop Closure	97
6.5.1	Room Descriptor Generator	97
6.5.2	Loop Closure Detection	98
6.6	Collaborative Situational Graph	99
6.7	Experimental Validation	100
6.7.1	Simulated Data	100
6.7.2	Real-world Data.	102
6.7.3	Ablation Study	104
6.8	Results and Discussions	104
6.8.1	Simulated Data	104
6.8.2	Real-world Data	105
6.8.3	Ablation Study	105
6.9	Summary	106
IV	Validation and Field Tests	109
7	Industrial Applications	111
7.1	Photovoltaic Panel Inspection	111
7.1.1	Introduction	111
7.1.2	Hardware Setup	112
7.1.3	Software Architecture	112
7.1.4	Results	113
7.2	Windmill Inspection	116
7.2.1	Introduction	116
7.2.2	Hardware Setup	116
7.2.3	Software Architecture	116
7.2.4	Results	119
8	International Competitions	121
8.1	OpenCV AI Competition 2021	121

8.1.1	Challenge description	121
8.1.2	Proposed Solution	122
8.2	ICUAS 2022: UAV Competition	125
8.2.1	Challenge description	125
8.2.2	Proposed Solution	127
8.2.3	Results	129
8.3	IMAV 2022: Nanocopter AI challenge	133
8.3.1	Challenge Description	133
8.3.2	Proposed Solution	134
8.3.3	Results	135
8.4	ICUAS 2023:	137
8.4.1	Challenge Description	137
8.4.2	Proposed Solution	137
8.4.3	Results	139
8.5	ICUAS 2024: UAVs for Indoor Agriculture Challenge	142
8.5.1	Challenge Description	142
8.5.2	Proposed Solution	143
8.5.3	Results	144
9	Results of Aerostack2 as an Open-Source Project	147
9.1	Open-Source Project Approach	147
9.2	Software Package Structure	148
9.3	System Requirements	149
9.4	Scientific Research Built with Aerostack2	150
9.5	Communication and Dissemination Activities	150
10	Discussion and Conclusions	153
10.1	Discussion of the obtained results	153
10.2	Conclusions	155
10.3	Future Work	157
	Appendices	159
A	Scientific Dissemination	161
A.1	Journal Publications	161
A.2	Peer-reviewed conference publications	162
A.3	Preprints and Under-Review Manuscripts	163
B	Aerial Platforms Used in This Thesis	165
B.1	Crazyflie 2.1	165
B.2	Ryze Tello	167
B.3	F330 Custom quadcopter	168
B.4	DJI Matrice 210 V2 RTK	169
B.5	DJI Matrice 300 RTK	170
B.6	DJI Matrice 350 RTK	171

List of Figures

1.1	The technical processes identified in the ISO/IEC/IEEE 15288:2023 standard are grouped into four blocks. The numbering of each process follows the standard for clarity. Processes shown with a dashed outline have not been considered in this thesis due to the scientific nature of this work. ¹	10
3.1	Tier I Systems Architectural Proposal Schema. In the figure, the boxes represent the three subsystems presented at this tier. Black and white cubes represent the inputs and outputs of each subsystem, respectively. The user is positioned on the left side of the figure—who will provide the references to the controller—while the robotic platform is on the right—who will obey them.	30
3.2	Tier II Systems Architectural Proposal Schema. In the figure, the boxes represent the subsystems of the architecture. The submodules corresponding to different Tiers are separated by blue shades. Black and white cubes represent the inputs and outputs of each subsystem, respectively. The user is positioned on the left side of the figure, while the robotic platform is on the right.	33
3.3	Schema of a Tier III Architecture. The IC blocks represent the Inference Components that interface with the Knowledge Base Subsystem for both, adding or retrieving models from the knowledge base of the system. This components can be used in different subsystems of the architecture to increase the awareness of the overall system.	37
3.4	Tier IV Systems Architectural Proposal Schema. In the figure, multiple agents appear overlapped, with each agent having the same internal architecture as Agent I. Each agent controls a robotic platform, and all of them are interconnected through the <i>Communication Subsystems</i> , as shown on the left. The user interacts with the agents through these communication channels.	41
3.5	Initial situation on both of the emergency landing scenarios. In both scenarios there are a drone (Drone I, in red) that detects a battery decay so that it will not be able to finish its task and follow a trajectory towards the landing area. The issue is that this trajectory collides with another drone trajectory (Drone II, in black) so the system has to take the decision about how to solve this.	45

4.1	Overview of the Tier II+ Aerostack2 framework architecture. Each colorized block represents a hierarchical layer, which can be constituted from multiple ROS 2 nodes or processes, represented as white blocks. All the nodes and agents are communicated through the ROS 2 communication channel, as is represented with grey arrows. The big colorized arrows going left to right, represent the <i>command</i> flows, so that each blocks send commands to the right one. Nevertheless, each module can feed back meaningful information to the left modules through the communication channel, so the communication is bidirectional.	51
4.2	Different users of the Aerostack2 framework, ordered from the user or high-level view on the left to the hardware or low-level view on the right. Each level is accessible to the developer without the necessity of concern for the higher abstraction levels.	52
4.3	Overview of the extended framework architecture. This proposal illustrates two distinct communication channels: one for intra-robot messages and another for inter-agent communications. The ground station communicates with other agents via the inter-agent communication channel. Dashed arrows represent command flows.	53
4.4	Diagram of the Aerial Platform component. Left arrows represent the flow of information towards the rest of the framework. The right arrow represents the communication with the hardware components using the proper protocol.	54
4.5	This FSM diagram represents the platform's state transitions. The DISARMED state is positioned on the left, while LANDED, TAKING_OFF, FLYING, and LANDING form a square structure around the EMERGENCY state in the center. Transitions between states are triggered by specific events such as ARM, DISARM, TAKE_OFF, TOOK_OFF, LAND, LANDED, and EMERGENCY. The EMERGENCY state serves as a critical fallback, accessible from all operational states.	55
4.6	Diagram of the motion controller component. Blue boxes represent the capabilities of it and how interact between them and the other modules. Plugins are represented with green trapezoids.	57
4.7	Sequence diagram describing the interaction between the Motion Controller component, the controller plugin, and the aerial platform to enable following a motion reference in a specific control mode.	59
4.8	Diagram of the State Estimator component. Blue boxes represent the functions of it and how interact between them and the other modules. Plugins are represented with green trapezoids.	60
4.9	Schematic representation of the behavior template architecture. The <i>behavior monitor</i> processes external commands and monitors behavior execution by interfacing with the <i>behavior executor</i> , which handles the underlying implementation. The system outputs include status and feedback, derived from the processed inputs.	62
4.10	Behavior Tree representation of a simple takeoff, go to a position, and wait for a Return to Land (RTL) mission.	64
4.11	Screenshot of the Alphanumeric Viewer tool. All information is displayed within a terminal, minimizing the computational resources required to run it. It has different panes depending on the data to be inspected.	65
4.12	Screenshot of the Keyboard Teleoperation pannel.	66
4.13	Real experiment environment, where the two doors that must be passed through, the Crazyflie, and the Tello can be observed on the left and right, respectively.	70

4.14	Comparison of the trajectories performed by the drones during the gate crossing example.	71
4.15	Plot of the evolution of the position on each axis of both drones during the real experiment.	71
4.16	Forest environment with a flying drone simulated within the Gazebo simulation. .	72
4.17	Status of the Knowledge Graph at startup: both drones are at the home station, disarmed, with high battery levels, and tasked with searching for the lost person. .	75
4.18	Knowledge Graph after one drone locates the person and communicates the position to the other drone, allowing it to approach. This is represented by the fact that the two drones are close to each other, one of them has located the person, and both are still flying.	75
5.1	Drone flying through gates during the real-world flights experiments. The gate position estimation was done using ArUco markers.	79
5.2	Diagram representing the security zone based on $T_{security}$ with respect to the next waypoint w_i , when the UAV is at t_{uav}	81
5.3	New base trajectory (blue) generated from a previous trajectory (red) using swapping waypoints (green circles) and modified waypoints (blue circles). Red crosses represent the previous positions of each waypoint, while the red triangle indicates the UAV's position when the new trajectory is computed.	83
5.4	One-dimensional representation of a generated trajectory (yellow) with a base trajectory (red) and a Local Gaussian Modification (blue) applied to waypoint w with a displacement A along the x-axis. The red triangle represents the UAV's position when the modification is applied, red crosses indicate the original positions of two waypoints, and the blue circle marks the modified position of waypoint w	84
5.5	Axis by axis (x, y, and z-axis, corresponds with top, middle, and bottom subfigures, respectively) plot of several dynamic trajectories position references generated using only LGMs to modify the waypoint positions for different adjustments of some waypoints.	86
5.6	Position, velocity, and acceleration references generated by the trajectory generation during the following of the dynamic trajectory shown in Fig. 5.5.	87
5.7	Aerostack2 components used for the simulation experiments. The gray boxes represent the additional components required for this experiment.	88
5.8	Aerostack2 components used for the real-world experiments. Compared to the simulated experiments, the Aerial Platform has been replaced with the Pixhawk system, the camera Realsense T265 has been integrated into the Sensor-Actuator interface, and the State Estimator plugin has been modified to use the odometry provided by the external sensor.	89
5.9	Trajectory generated (blue) and the trajectory followed by the UAV (orange) during the real flight	90
5.10	Plot per axis of the Trajectory Speed generated during the real flight (blue), and the trajectory speed followed by the UAV (orange), with a continuous increase in speed per lap.	90

6.1	Collaborative graph generated by three robots mapping a building floor (orange, gray, and blue for robots 1, 2, and 3 respectively). The collaborative graph is generated from the point of view of Robot 1. Semantic information gathered by the other robots, such as rooms or walls appears in transparent colors in the graph layers. The origin of the coordinate systems of the other robots appears surrounded by a circle. Below is the result of merging the point clouds obtained from each of the robots. This point cloud is displayed for visualization purposes and is not utilized in inter-robot communication.	94
6.2	<i>Multi S-Graph</i> architecture viewed from <i>Robot_i</i> perspective. Odometry measures and 3D Lidar data are the inputs that the robotic platform provides. Each robot executes its own Local S-Graph system and generates the Room Descriptors and the Distilled S-Graphs that are exchanged with the rest of the robots. Messages exchanged between robots are represented in gray. From the point of view of the <i>Robot_i</i> the rest of the agents provide the Distilled graphs and the Room descriptors that are used to generate the Collaborative S-Graph. Information provided by the rest of the robots appears in green color.	95
6.3	<i>S-Graphs+</i> overview. The 3D LiDAR measurements and robot odometry, are pre-filtered and processed to extract wall planes, rooms, floors, and loop closures. Figure from (Bavle et al., 2023).	96
6.4	Room Descriptor (right) obtained from a Room-Centric Point Cloud (left) using a Scan Context. The value of each cell in the descriptor represents the maximum height of the points within the corresponding bin.	98
6.5	Schema of the different stages of Collaborative Graph generation from the perspective of a single agent. The local graph contains the original S-Graph generated by the robot, while the external graphs correspond to the Distilled Graphs shared by other agents. For clarity, only the Room Layer is represented in this figure. The fixed node indicates that its estimation remains unchanged during optimization and represents the robot's origin.	99
6.6	Simulated experiments, showing the trajectory and the point cloud map generated for each robot.	101
6.7	Real experiment scenarios, showing the trajectory and the point cloud map generated for each robot. In (e) the experiment was run with 3 robots.	102
6.8	Real experiment scenarios, showing the trajectory and the point cloud map generated for each robot. In (e) the experiment was run with 3 robots.	103
6.9	Accumulated data shared between agents, consisting of Walls Nodes (Wall GN) and Room Nodes (Room GN) for the Semantic Graph, and the Room Scan Contexts (Room SC) and Room Centric point clouds (Room PC).	106
7.1	Image captured during the inspection: the left side corresponds to the IR image, while the right side shows the RGB image.	112
7.2	Overview of the system architecture used for the real photovoltaic inspection. Each drone has its own internal communication channel, which it uses to control its behavior. The relevant operational information is sent to the user through an external communication channel. The communication module is responsible for routing the required data between both channels.	113
7.3	Photorealistic simulation of the photovoltaic plant used during the development process.	114
7.4	DJI M300 flying over the photovoltaic panels at the Repsol Technology Lab in Madrid.	115

7.5	Overview of the system architecture used for real windmill inspection. In this setup, the system interfaces directly through the remote controller, which is also connected to the <i>Windmill Observer</i> , the component responsible for updating the nacelle's rotation over time.	117
7.6	Simulated windmill and the predefined route points visualized	118
7.7	Image of a windmill blade obtained during the inspection. All rights reserved by Generadron S.L. ²	119
7.8	Photograph of the drone autonomously flying behind the windmill.	120
8.1	Overview of the system architecture.	122
8.2	<i>Generate Map</i> Behavior pipeline in three steps.	123
8.3	Trajectory generated by the <i>Compute Free Path</i> behavior, visualized along with the depth map in RViz.	124
8.4	Experimental setup used to validate the proposed solution. The drone must navigate in a scenario with a set of columns.	125
8.5	Arrangement of the drone's starting position, the columns, and the drone in two scenarios.	125
8.6	Trajectories followed by the aircraft in both experiments.	126
8.7	Virtual scenario divided into three zones: Zone 1, for taking off, Zone 2, for the exploration task, and Zone 3, for the precision delivery task.	126
8.8	Overview of the Tier II architecture used for the competition. The brown blocks were provided by the organizers.	127
8.9	Plan executor flow diagram.	128
8.10	Occupancy map with the path free of obstacles generated by the path planner. The path calculated is represented in gray, and the waypoints of this path that are sent to the controller are marked in green. Each pixel represents a 0.5 meter square in the real world.	128
8.11	Ball throwing maneuver scheme. The drone will attempt to achieve a constant speed while maintaining a relative height to the target. When the system predicts that the ball will land on the target, it is released, and the drone moves away from the wall.	129
8.12	Ball throwing maneuver showing drone trajectory (blue), ball trajectory (orange), tag actual position (red), target localization (green), and ball contact point (blue).	130
8.13	Timelapse of the drone scoring the ball	132
8.14	ICUAS 2022 Winner Diploma awarded to our team	132
8.15	Simplified schema of the arena. The white drone attempts to traverse the maximum distance, passing through the red gates while avoiding the black obstacles.	133
8.16	Overview of the system architecture used for the IMAV challenge. All computations were performed within the GCS, and the output commands were sent to the drone via radio.	134
8.17	Visual control approach. On the left, the obtained image; on the right, the masked version of the gate. The rectangle encloses the gate, and the green point marks the reference point at the center of the gate. The algorithm tries to center this point in the middle of the image while continuously advancing forward.	135
8.18	Timelapse of the Crazyflie crossing the gate in the IMAV22 competition	135
8.19	Special Achievement Award diploma obtained from our participation in the IMAV 22 Nanocopter AI Challenge.	136

8.20	Schema of the software architecture for the ICUAS23 competition. The square boxes represent processes, while the rounded ones indicate logical components of each process	138
8.21	Image of the detector during the competition. The green rectangle indicates the region of interest, while detections are marked with annotated blue rectangles. . .	139
8.22	Competition arena in the Gazebo simulation environment.	140
8.23	Real environment scenario with the drone flying while searching for defects. . . .	140
8.24	Award of the ICUAS 2023 UAV Competition	141
8.25	Competition Arena in the Gazebo simulated environment	142
8.26	Schema of the software architecture for the ICUAS24 competition. The square boxes represent processes, while the rounded ones indicate logical components of each process.	143
8.27	Visualization of the generated graph (green spheres) and the computed path (green lines).	144
B.1	Crazyflie 2.1	166
B.2	Ryze Tello	167
B.3	Custom F330 based quadrotor equipped with an NVIDIA Jetson Xavier NX, a Pixhawk 4 Mini, an Intel RealSense T265, and a USB fisheye camera.	168
B.4	DJI Matrice 210 V2 Equipped with the Nvidia Jetson AGX XavierSBC	169
B.5	DJI Matrice 300 RTK Equipped with the Nvidia Jetson AGX XavierSBC	170

List of Tables

2.1	Comparison of relevant programmable flight systems.	18
2.2	Comparison of relevant open-sourced software stacks for aerial robotic systems. Industrial use is only evaluated for the frameworks that support real flights. S: Simulation; RL: real experiments in the laboratory; RO: real experiments outside the laboratory.	20
4.1	Functional Requirements obtained from the relevant Stakeholders	50
4.2	Comparison between Aerostack2 modules and plugins used in simulation and real flights of the crossing gates experiment.	69
4.3	Overview of drone-assisted search and rescue tasks and sub-tasks for efficient person localization and coordination.	72
4.4	Key entities and their associated properties in the search and rescue mission. . . .	73
4.5	Relationships between key entities in the search and rescue mission.	73
4.6	Key queries for the search and rescue mission, their return values, and corresponding actions to update the mission plan.	73
5.1	Time spent generating base polynomial trajectories for different numbers of points in a laptop computer and in an onboard computer.	86
5.2	Time spent generating and evaluating the sum of multiple LGMs acting on the same trajectory on a laptop computer and on an onboard computer.	86
5.3	Results for simulated flight with computing time. CTA: computation time added, Ref. speed: reference speed of the trajectory generator, Real speed: actual speed achieved by the drone, Elapsed time: duration of the run, Success rate: proportion of laps completed without collision on each run.	88
6.1	Absolute Trajectory Error (ATE) [cm] for our simulation experiments. The best results are boldfaced.	104
6.2	Data exchanged [MB] between agents in simulation experiments. The best results are boldfaced.	105
6.3	Point cloud RMSE [cm] for our in-house real experiments. The best results are boldfaced.	105
6.4	Data exchanged [MB] between agents in real experiments. The best results are boldfaced.	105
6.5	Ablation study of the ATE and data shared [MB] for simulated datasets with 2 robots, FA means Fine Alignment. The best results are boldfaced.	106
8.1	Experimental results for both scenarios.	124
8.2	Average time from 10 last evaluations during the classification phase.	129

8.3	Average error from 10 last evaluations during the classification phase.	129
8.4	Average time (s) from completed trials during finals in each scenario.	131
8.5	Number of hits on each surface of the throwing completed trials during the finals in each scenario, and estimated hit accuracy error.	131
8.6	Evaluation results comparing different plants to detect in different beds.	145
9.1	Collection of applications and use cases where Aerostack2 is utilized. All simulations were performed in Gazebo. The platform used is followed by the computation method in brackets. The number of drones used in simulation appears in brackets. Both GPS and VIO estimation use an external localization plugin that adapts sensor measurements into Aerostack2.	151
B.1	Specifications of the Bitcraze Crazyflie 2.1	166
B.2	Specifications of the Ryze Tello	167
B.3	Specifications of the Custom F330 Platform	168
B.4	Specifications of the DJI Matrice 210 V2 RTK	169
B.5	Specifications of the DJI Matrice 300 RTK	170
B.6	Specifications of the DJI Matrice 350 RTK	171

Acronyms

AI Artificial Intelligence. 27

API Application Programming Interface. 29, 63

ATE Average Trajectory Error. xxv, 104, 106

CI Continous Integration. 148

CSLAM Collaborative Simultaneous Localization and Mapping. 93, 94, 104, 155

CVAR Computer Vision & Aerial Robotics. 9

DDS Data Distribution Service. 18

FCU Flight Control Unit. 17, 18, 54, 57

GCS Ground Control Station. xxiii, 113, 134

GPS Global Positioning System. 30, 55, 59

IC Inference Components. 36, 38

IMAV International Micro Aerial Vehicles. 133

IMU Inertial Measurment Unit. 30, 55

IR Infrared. xxii, 29, 112

KB Knowledge Base. 36

KG Knowledge Graph. 67

LGM Local Gaussian Modifier. 79, 81, 83–85, 91, 92, 154

MAV Micro Aerial Vehicle. 121, 122

MoCap Motion Capture. 69, 137

MPC Model Predictive Controller. 38

MRS Multi-Robot System. 7, 153

ODE Ordinary Differential Equation. 38

PV Photo-Voltaic. 29, 38–40

R&D Research and Development. 7

RGB Red Green Blue. xxii, 29, 112

ROS Robotics Operating System. 8, 50, 62, 68, 147, 148

SBC Single Board Computer. xxiv, 169, 170

SLAM Simultaneous Localization and Mapping. 8, 15, 93, 94, 122, 155, 156

SotA State-of-the-Art. 104

SSLAM Semantic Simultaneous Localization and Mapping. 122

TRL Technology Readiness Level. 51

TSP Travelling Salesman Problem. 143

UAS Unmanned Aircraft System. 17

UAV Unmanned Aircraft Vehicle. 32, 38–40, 48, 79, 137, 157

Part I

Introduction and State of the Art

Chapter 1

Introduction

1.1 Autonomous Robotic Systems

Autonomous robotic systems are composed of advanced electromechanical machines capable of operating independently in dynamic environments by perceiving their surroundings, processing data, and performing tasks without continuous human intervention. These systems need to sense, decide, and interact in the surrounding environment. In recent years, the research community has been developing and improving a vast amount of algorithms to enhance the "perception", "deliberation", and "control" capabilities of these systems.

The Research and Development (R&D) in these research fields has led to a variety of autonomous robotic systems that can operate autonomously in controlled situations, such as logistic warehouses or factories, where the environment is well known and regulated. However, there is still significant room for improvement when deploying these autonomous systems in real-world scenarios that may be unknown or evolve over time. This is the typical operating condition of mobile robots that function in outdoor or in unstructured environments. In these types of environments, the vast majority of robots are remotely piloted by a human operator, due to the lack of robustness and generality of the existing algorithms.

The complexity of robotic systems increases exponentially when considering Multi-Robot Systems (MRSs). In these systems, capabilities like "communication" or "organization" are needed to take fully advantage of the extended capabilities that these systems can obtain.

Successful examples of cooperative Multi-Robot Systems (MRSs) demonstrate their remarkable potential in real-world applications. A notable instance is drone light shows, where hundreds or even thousands of autonomous drones synchronize their movements with precision, forming intricate aerial displays. These performances are enabled by sophisticated communication protocols and real-time trajectory planning, allowing each drone to coordinate its position relative to the entire fleet. This coordination highlights the benefits of collective decision-making and organization in robotic systems, enabling complex tasks that would be impossible for individual drones.

On top of this, there is a necessity to integrate all of these required capabilities into a sys-

tem architecture that connects and coordinates these functions into an autonomous system. In order to foster the collaboration and the reuse of algorithms between robotic researchers, there is a growing interest in using common frameworks that help with the integration of different capabilities developed from different research centers, allowing to build novel systems based on the state-of-the-art solutions. Robotics Operating System (ROS) framework is one of the best examples of this common framework which has been used in more than 55% of commercial robots¹ in 2024, strongly correlated with high academic adoption rates. These standardizations have helped roboticists quickly adopt the latest algorithms and build on top of them, accelerating global advancements in robotic research.

However, many robotic systems still face significant challenges when operating in open-world or unstructured environments. Such robots often lack situational awareness, making them susceptible to failure when unexpected situations arise or when environmental conditions evolve unpredictably. To address this, knowledge-based systems are increasingly being integrated into robotic architectures, enhancing robustness by enabling robots to reason about their environment and adapt their behaviors dynamically.

At this point, we can distinguish two important study fields in robotics autonomy research: (1) improving the essential capabilities of an autonomous system (e.g. perception, deliberation, or control) and (2) developing system architectures that facilitate the sharing and integration of these capabilities in a seamless and compatible manner.

Although these areas may seem distinct, they are fundamentally interconnected, forming a unified approach to building intelligent robotic systems. Working on both creates a symbiotic relationship that enhances the quality of results obtained in each. First, developing robotic system architectures provides a comprehensive view of the challenges involved in building robotic systems, considering all related fields and their interconnections. However, advancing the state of the art requires narrowing the objective and focusing on a specific aspect of the broader problem of robotic system development. This demands a deep understanding of available technology and the concrete details necessary for practical implementation in real-world scenarios. Combining these two research fields enables the development and refinement of robotic architectures without overlooking critical technical details while also allowing for algorithmic improvements without losing sight of the overall system perspective.

In this doctoral work we will go through this both fields starting from a general pattern architecture for developing robotic system with different capabilities, moving into a software framework for developing aerial robotic system, tested in both scientific and industrial applications. Further we will focus on developing two novel algorithms relevant in the field of dynamic trajectory generation and distributed Collaborative SLAM.

1.2 Thesis Motivation

In recent years, the development of increasingly complex robotic systems has emerged across a wide variety of fields, ranging from general-purpose industrial automation to domain-specific solutions such as agricultural drones and underwater inspection robots (Campilho and Silva, 2023). While robots like articulated robotic arms have been commonly used in factories, such as in the automotive industry, the mobile robotics market did not take off until 2013, when advancements in AI and sensor technologies enabled widespread commercial viability (Raj and Kos, 2022).

One of the key drivers behind the increasing use of mobile robots is the field of aerial robotics, particularly the adoption of UAVs. These vehicles are employed in a wide range of applications,

¹<https://www.businesswire.com/news/home/20190516005135/en/Rise-ROS-55-total-commercial-robots-shipped>

including light shows (Huang et al., 2021), cinematography (Galvane et al., 2017), search and rescue operations (Chen et al., 2024), industrial inspections (Nooralishahi et al., 2021), package delivery (Saunders et al., 2024), and warehouse operations (Awasthi et al., 2023), among others.

The maneuverability, freedom of movement, and ease of use of UAVs have enabled their penetration into a broad market, accounting for 60% of the global professional service robotics market, valued at \$58.5 billion in 2024 (Precedence, 2025)(IndustriyArc, 2024). Despite the widespread deployment of aerial robotic systems, the vast majority of these applications remain teleoperated or rely on simplistic autonomous behaviors with minimal capabilities for autonomous decision-making (Elmokadem and Savkin, 2021).

One of the main focuses of the robotics research community is to enhance the autonomy of robots, striving toward reliable fully autonomous operation across a wide range of applications.

This thesis has been conducted within the Computer Vision & Aerial Robotics (CVAR) Group, a research group dedicated to improving the autonomy of aerial systems by integrating multiple sensory inputs for the decision-making process. The group has specialized in the use of image sensors for this purpose for more than ten years. Over the years, the group has demonstrated its results in various autonomous tasks for both industrial and scientific applications in increasingly complex scenarios. These challenges necessitate the development of novel algorithms and architectures to tackle emerging problems.

Despite significant advancements in the robustness of robotic systems, they are still far from reliably operating in open-world, dynamic, and changing environments due to their “lack of awareness of what is going on.” The CORESENSE European project aims to develop a theory of understanding and awareness for autonomous robots, which will be implemented as a reference architecture and engineering toolbox. This theory and technology will contribute to building autonomous robots with enhanced capabilities for operating with limited or no supervision, as well as improving their cognitive, social, and physical interactions to be more intuitive, safe, and efficient. Additionally, the project seeks to develop this theory at the robotic system level, treating single-agent and multi-agent robotic systems in a unified manner. This approach motivates research into novel methods and techniques for leveraging multi-agent systems in an integrated and transparent way.

1.3 Methodology

The different works presented in this thesis are conceived from a systems engineering perspective, considering that the different algorithms, methods, and architectures will be part of a system, or a system of systems, that has a concrete objective. This perspective allows us to take advantage of the multiple methodologies and standards that exist to provide a standardized and comprehensive description of the different works, while ensuring a detailed and deep analysis of the requirements and constraints existing on each case.

The principal outcomes of this thesis are the development and implementation of software frameworks that are further deployed in actual systems. In order to develop these systems in the more comprehensive way, the methodology used during the development of these thesis are based in several standards widely used in systems engineering.

The standard ISO/IEC/IEEE 15288:2023 (ISO, 2023) is an international standard that establishes a common framework for describing the life cycle processes of systems created by humans. It provides a comprehensive set of processes and associated terminology from an engineering perspective, applicable across various levels of a system’s hierarchy. These processes can be applied at any level within the system’s structure, supporting the definition, control, and improvement of the system’s life cycle processes.

System Life Cycle Processes: Technical Processes

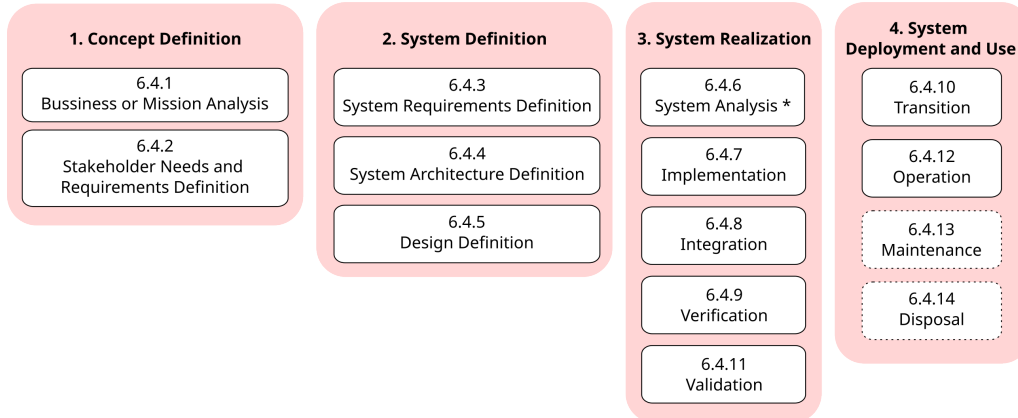


Figure 1.1: The technical processes identified in the ISO/IEC/IEEE 15288:2023 standard are grouped into four blocks. The numbering of each process follows the standard for clarity. Processes shown with a dashed outline have not been considered in this thesis due to the scientific nature of this work.²

This standard provides a comprehensive details about what is the purpose, the outcomes and guidelines about how to carry on the different processes.

Although this standard covers the entire life cycle of a system, product, or service, this thesis focuses on the development of new frameworks and novel systems. Therefore, specific parts of the life cycle processes are emphasized, particularly the technical processes. Figure 1.1 shows the different technical processes outlined in the standard, clustered into four blocks:

1. **Concept Definition:** Focuses on identifying stakeholder needs and exploring potential concepts, including business or mission analysis. It involves defining initial system requirements and evaluating possible solutions to address these needs.
2. **System Definition:** Involves detailed system architecture and design, translating stakeholder needs into specifications. It includes processes for stakeholder requirements, system requirements, architecture, and design definition.
3. **System Realization:** Covers the creation, integration, verification, and transition of the system into a functional operational state. Ensure that the system design is transformed into a concrete, testable, and operational product.
4. **System Deployment and Use:** Focuses on validating system performance in its intended environment and ensuring operational readiness. It includes processes for system operation, maintenance, and ongoing support throughout its use.

Although this systems engineering methodology is the one that best fits the development of system architectures and software frameworks, it lacks some essential aspects that should be taken into consideration when conducting research on the development of novel algorithms.

In addition to the systems engineering perspective, this thesis adopts a **hypothesis-driven methodology** to evaluate the proposed algorithms. This approach is structured around three key steps:

²(ISO, 2023), the System Analysis process is considered separate from these four blocks. However, while the System Analysis process does inform system definition activities, it has been considered more accurately categorized as part of the system realization block. This is because it plays a crucial role in analyzing the realized system components, their interactions, and the overall system behavior, which are key aspects of bringing the system into existence and ensuring it meets its intended purpose.

1. **Hypothesis Formulation:** Develop clear and testable hypotheses based on the overall system requirements and theoretical models. These hypotheses are formulated to capture expected improvements and performance enhancements, ensuring that they are measurable and falsifiable without relying on concrete examples.
2. **Experimental Design and Data Collection:** Design controlled experiments that compare baseline and experimental conditions under similar operational scenarios. This includes defining a consistent set of performance metrics and establishing protocols for data logging in both simulation and real-world settings, ensuring that the data collected is comprehensive and reproducible.
3. **Statistical Validation and Iterative Refinement:** Analyze the collected data using appropriate statistical tests to determine the significance of observed differences. The outcomes of these tests inform whether the hypotheses are validated or require refinement, thus driving iterative improvements in the system design and experimental approach.

In summary, this hypothesis-driven methodology creates a systematic feedback loop within the overall systems engineering framework. By continuously validating and refining the proposed approaches through experimentation and analysis, the methodology ensures that the innovations are robust, scalable, and well-suited to address real-world challenges in robotics.

1.4 Objectives

In this thesis we tackle the interests of both by developing a concrete architecture for developing multi-robot aerial systems and a theoretical architecture abstraction for developing robotic systems from the simplest reactive ones until a complex collective awareness robotic system. The objectives and sub objectives of this doctoral work can be summarized as:

O.1 Development of a reference architecture for autonomous robotic systems.

- O.1.1 Define an incremental layered reference architecture for robotic systems. Clearly establish the concept of incremental layers, explaining how system capabilities can evolve from basic control schemas to complex multi-agent cognitive frameworks.
- O.1.2 Categorize robotic systems according to their decision-making capabilities. Define and characterize the different system tiers based on the complexity and autonomy of their decision-making processes.
- O.1.3 Identify key functionalities and subsystems required at each architectural tier. Analyze and outline the specific components, their interconnections, and information flow necessary to fulfill the objectives of each incremental layer within the architecture.
- O.1.4 Elaborate on the concepts of Understanding and Awareness in robotics and provide structures for taking advantage of them in decision-making processes.

O.2 The development of a comprehensive software framework for developing multi-robot aerial systems. This framework is generic and allow for handling a wide variety of aerial platforms and swarms of heterogeneous drones.

- O.2.1 Design a modular, platform-independent framework for developing UAS: The framework should be adaptable to different aerial platforms, allowing integration with various UAV hardware and software configurations.

- O.2.2 Enable behavior-based mission control for structured task execution: Implement a high-level mission control system that defines UAV behaviors in a modular way, allowing flexible and reusable task execution strategies
 - O.2.3 Ensure scalability and integration across different drones and simulators: Develop a system that supports both single and multi-robot aerial operations, ensuring interoperability between real and simulated environments.
 - O.2.4 Validate the framework through real-world experiments in controlled and industrial environments: Conduct extensive testing in diverse scenarios, including indoor and outdoor settings, to evaluate the robustness and applicability of the framework in real UAV missions.
- O.3 The development of a fast algorithm for dynamically modifying 3D trajectories for resource-constrained environments.
- O.3.1 Develop a computationally efficient method for modifying UAV trajectories in real-time. Design a lightweight algorithm that allows UAVs to dynamically adjust their paths in response to environmental changes without excessive computational overhead.
 - O.3.2 Reduce onboard computational load to enable dynamic trajectory adjustments: Optimize the trajectory generation process to ensure it can run on UAVs with limited computing resources while maintaining high-speed adaptability.
 - O.3.3 Ensure smooth and agile UAV navigation even in constrained environments. Implement a method that preserves trajectory smoothness and minimizes abrupt changes, allowing UAVs to navigate efficiently through cluttered and dynamic spaces.
 - O.3.4 Validate the trajectory generation method through simulations and real-world experiments: Test the approach in both simulated drone racing environments and real-world flight scenarios to demonstrate its effectiveness and reliability.
- O.4 Development of a Collaborative SLAM Approach for Multi-Robot Systems
- O.4.1 Create a distributed SLAM system that reduces inter-robot data exchange. Develop a decentralized SLAM framework that minimizes bandwidth usage by efficiently sharing only the most essential mapping information among robots.
 - O.4.2 Integrate semantic-relational mapping to improve localization accuracy. Utilize high-level semantic features, such as object recognition and environmental context, to enhance mapping precision and robustness in structured environments.
 - O.4.3 Develop robust loop closure detection to enhance map consistency. Implement a novel method for identifying loop closures between different robots' maps, reducing localization drift and improving global map consistency.
 - O.4.4 Validate the SLAM approach in simulated and real-world multi-robot scenarios. Conduct large-scale experiments involving multiple robots collaborating in complex environments to assess the accuracy, efficiency, and scalability of the approach.

1.5 Contributions

The objectives of this thesis are very broad and challenging. This doctoral work has been successfully achieved due to the collaboration of a multidisciplinary team of people belonging to the CVAR Research Group and the partners of the CORESENSE Consortium. These collaborations have resulted in a diverse range of contributions to this thesis, spanning from theoretical advancements to concrete open-source software implementations. We categorize them as follows:

- C.1 A versatile **reference architecture for robotic systems** that structures decision-making processes, elaborating on the concepts of **Understanding** and **Awareness** for both single and multi-robot systems.
- C.2 A comprehensive **software framework** supporting the development of **aerial robotic systems** across multiple operational levels, ranging from low-level flight controllers and state estimators to mission planning, execution, and monitoring. The framework emphasizes component reusability, making it adaptable to various hardware platforms and mission domains.
- C.3 A fast and lightweight algorithm for generating **adaptive trajectories** that dynamically adjust to changes in waypoint configurations, ensuring smooth and agile motion. This approach continuously samples and refines trajectories, enabling real-time adaptability.
- C.4 A distributed **Collaborative SLAM** framework based on a four-layered optimizable hierarchical Situational Graphs structure, leveraging semantic-relational information for map merging while minimizing inter-robot data exchange. It incorporates a hybrid room-based descriptor that combines raw point cloud data with semantic and hierarchical knowledge to enhance inter-robot loop closure detection.
- C.5 **Extensive experimental validation** of contributions C.2, C.3, and C.4 through both simulation and real-world scenarios. The experiments cover a wide range of operational conditions and are analyzed to demonstrate the practical applicability and scalability of the proposed approaches.
- C.6 **Open-sourcing the developed software framework** to foster community adoption and further research contributions. The released software includes comprehensive documentation, modular components, and integration support for widely used robotic middleware.

1.6 Outline

This doctoral work is divided in four parts with a total of eight chapters and two annexes, as follows:

- Part I, formed by two chapters, introduces, motivates, and situates the presented works within the state of the art.
 - Chapter 1, the present one, contextualizes this work within autonomous robotic systems development, motivates the thesis, and summarizes objectives and contributions.
 - Chapter 2 provides a comprehensive review of relevant literature and state-of-the-art approaches closely related to the thesis topics.
- Part II, consisting of two chapters, addresses architectures and software frameworks for robotic systems, providing foundational structures for advanced robot capabilities.
 - Chapter 3 describes a novel reference architecture for robotic systems, presenting an incremental approach divided into four tiers: feedback control, task decomposition, system awareness, and multi-agent capabilities.
 - Chapter 4 introduces Aerostack2, a comprehensive software framework tailored specifically for aerial robotic systems, highlighting its design, implementation, and validation examples such as synchronized drone gate crossing and search and rescue operations.

- Part III includes two chapters that present and validate novel algorithms developed to enhance specific autonomous capabilities in robotic systems.
 - Chapter 5 introduces and evaluates a dynamic trajectory generation algorithm capable of real-time trajectory modifications, essential for agile flight scenarios.
 - Chapter 6 describes a novel distributed knowledge-centric collaborative SLAM algorithm, emphasizing semantic situational graphs, loop closure detection, and collective awareness in multi-agent contexts.
- Part IV, composed of four chapters, synthesizes the achieved results and outlines future research directions.
 - Chapter 7 presents the industrial applications of the developed results, describing the integration of UAV-based inspection systems in two real-world scenarios: photovoltaic panel and windmill inspection.
 - Chapter 8 summarizes participation in international competitions where the proposed methodologies were evaluated. It provides an overview of the challenges, the approaches developed to address them, and the results achieved.
 - Chapter 9 analyzes the proposed software framework as an open-source project, explaining not only software-related metrics but also enumerating other scientific works that use our framework as the foundation of their research.
 - Chapter 10 concludes the thesis by summarizing key findings and presenting potential avenues for future research, further enhancements, and possible expansions of the developed methodologies and algorithms.
- Annexes provide supplementary material relevant to this doctoral thesis:
 - Annex A compiles the scientific publications generated during the thesis development.
 - Annex B details the hardware platforms utilized throughout this research.

Chapter

State of the art

This chapter provides a structured review of the state of the art related to this thesis, aligning with the main research areas explored in subsequent chapters. It begins by examining different robotic system architectures, from reactive to hierarchical and hybrid approaches, highlighting their impact on autonomy, decision-making, and adaptability. Following this, it explores software frameworks for autonomous aerial robotics, discussing their modularity, middleware support, and integration with various hardware and simulation platforms to facilitate the development of aerial robotic systems.

Then, dynamic trajectory generation for quadrotors is reviewed, focusing on state-of-the-art motion planning techniques that enhance flight efficiency, obstacle avoidance, and adaptability in dynamic environments. Finally, collaborative and semantic SLAM techniques are explored, emphasizing their role in multi-robot coordination, long-term autonomy, and environmental understanding.

2.1 Robotic System Architectures

Recent decades have witnessed significant progress in robotic system architectures, evolving from simple, reactive controllers to sophisticated hybrid frameworks that integrate multiple paradigms. Early contributions provided the foundation for modern systems that emphasize real-time performance, modularity, and robust sensor integration while avoiding the computational overhead associated with fully cognitive approaches (Brooks, 1986; Albus, 1991). These advances have enabled robots to operate effectively in dynamic and unstructured environments, balancing immediate responsiveness with long-term planning.

2.1.1 Reactive architectures

Reactive systems are defined by their direct mapping from sensor inputs to actuator commands without relying on an explicit internal world model. Brooks' subsumption architecture (Brooks,

1986) is a landmark in this area, demonstrating how layered control can yield fast and adaptive responses. Early implementations in mobile robots such as Genghis(Angle, 1989), and later in commercial products like the iRobot Roomba, showcased how reactive behaviors such as obstacle avoidance and wall following could be effectively executed with minimal processing delay. Modern reactive systems have been further enhanced through the integration of machine learning techniques, which optimize sensor-to-action mappings and bolster robustness in highly dynamic scenarios.

Building on the principles of reactivity, behavior-based architectures decompose complex tasks into a collection of modular behaviors operating concurrently. The idea that complex, emergent behavior can arise from the interaction of simple modules has been pivotal in advancing robotic control. Contemporary approaches, such as the use of Behavior Trees, formalize the sequencing, prioritization, and reuse of these modules, providing a structured framework that enhances modularity and facilitates debugging (Cao and Lee, 2022). This evolution from rudimentary rule-based systems to sophisticated behavior-based designs has been instrumental in extending the operational capabilities of robots in unpredictable settings.

Hierarchical architectures, often referred to as deliberative systems, follow a sense-plan-act paradigm that integrates sensory data into a comprehensive world model. The pioneering work on Shakey the Robot (Kuipers et al., 2017) established the viability of deliberative planning by incorporating symbolic reasoning and global path planning. This approach has been further developed in the context of autonomous vehicles, where projects such as the DARPA Urban Challenge have successfully demonstrated the use of sensor fusion, real-time planning, and SLAM techniques to navigate complex urban environments (Patz et al., 2008)(Buehler et al., 2009). Despite their computational intensity, modern hierarchical systems have benefited from innovations such as real-time re-planning and incremental world model updates, which significantly improve responsiveness.

Hybrid architectures represent a natural progression by combining the advantages of both reactive and hierarchical approaches. In these systems, a high-level deliberative module formulates strategic plans based on a global environmental model, while a low-level reactive controller executes immediate actions. This division of responsibilities ensures that the system can both plan effectively over longer time horizons and react instantaneously to unforeseen events. Autonomous vehicles developed for the DARPA Grand Challenge are prominent examples of this approach, as they successfully balance strategic decision-making with rapid local responses (Thrun et al., 2006).

The cumulative impact of these advancements is evident across a wide range of applications, including industrial automation, autonomous vehicles, service robots, and mobile platforms. Early reactive systems provided critical insights into real-time control in unpredictable environments, while subsequent deliberative and hybrid approaches have enabled the execution of complex tasks that require both strategic foresight and immediate responsiveness. This progression has not only enhanced operational safety and efficiency but also expanded the range of scenarios in which robots can perform reliably.

2.1.2 Cognitive Architectures

Cognitive architectures constitute theoretical frameworks that model the invariant structural and computational processes underlying human cognition. They provide a blueprint for constructing intelligent agents by specifying how perceptual systems, memory, attention, and decision-making mechanisms interact. In this framework, the architecture's fixed structure is analogous to a building's blueprint, while the variable content—acquired knowledge—resembles the movable furnishings within.

For example, ACT-R (Adaptive Control of Thought—Rational) (Ritter et al., 2019) delineates

the basic cognitive and perceptual operations that underlie human thought. Organized into specialized modules (such as declarative memory, procedural memory, and visual processing) and coordinated by a central procedural system, ACT-R is deeply rooted in cognitive neuroscience. This design has enabled extensive modeling of human performance in tasks ranging from memory recall to complex problem solving.

Similarly, the Soar architecture (Laird et al., 1987) was developed as a unified theory of cognition and a framework for building intelligent agents. Soar operates as a production system, where knowledge is represented by if-then rules in a working memory. It employs mechanisms like goal hierarchies and chunking to continuously enhance learning and problem solving. Although sharing fundamental principles with ACT-R, Soar has focused more on real-time performance and broad artificial intelligence functionality.

In contrast, CLARION (Connectionist Learning with Adaptive Rule Induction On-line) (Sun, 2006) adopts a hybrid approach by integrating both implicit and explicit cognitive processes. It maintains a dual-level structure in which a symbolic top level processes explicit knowledge while a subsymbolic bottom level handles implicit information. This integration models the interplay between consciously articulated reasoning and unconscious intuitive skills, shedding light on how these dual processes work together in cognition.

Extending these models further, the LIDA (Learning Intelligent Decision Agent) architecture draws inspiration from Global Workspace Theory. It conceptualizes cognition as a series of rapid, discrete cognitive cycles—each lasting roughly one-tenth of a second—during which sensory processing, various memory systems (including episodic and declarative memory), and attention mechanisms interact dynamically. This continuous cycle underpins perception, decision-making, and action selection, effectively mirroring the fluid stream of thought observed in biological systems.

Beyond these notable architectures, other models like EPIC (Executive-Process Interactive Control) and Sigma contribute to the field by exploring different aspects of cognition. EPIC focuses on simulating human perceptual-motor performance, while Sigma endeavors to merge symbolic AI with probabilistic graphical models to create a unified framework that encompasses cognitive, perceptual, and motor functions. Together, these diverse architectures provide a comprehensive understanding of human intelligence by illuminating both the constant structural mechanisms and the adaptable nature of knowledge.

2.2 Aerial Robotics Frameworks

During the past few years, Unmanned Aircraft System (UAS) have been controlled by two complementary components: the Flight Control Unit (FCU), which focuses on the low-level control of the aircraft, ensuring stable flight, and high-level control frameworks, which provide further autonomy to the vehicle by making decisions and commanding the FCU to perform specific tasks. Below, we discuss the state of the art in these two areas in more detail.

2.2.1 Programmable Flight Control Unit

These control units can be classified into two blocks: generic or embedded, each presenting distinct advantages and limitations. Generic FCUs offer versatility, accommodating various frames and components for customizable configurations, which is beneficial for developers. However, their integration and calibration require technical expertise and time. Embedded FCUs, integrated into complete aerial platforms, provide a user-friendly experience with pre-calibrated hardware for reliable performance. This simplicity suits users seeking ready-to-fly solutions, but may limit adaptability to new technologies or specific needs.

Flight Controller	Open Source	Simulation	Generic
Pixhawk/PX4	OSH, OSS	SITL, HIL	✓
Ardupilot	OSS	SITL, HIL	✓
Paparazzi	OSH, OSS	SITL, HIL	✓
Crazyflie	OSH, OSS	SITL, HIL	✗
DJI Matrice	Proprietary	HIL	✗
Parrot	Proprietary	SITL	✗
Skydio	Proprietary	-	✗

OSH: Open Source Hardware, OSS: Open Source Software,
SITL: Software In The Loop, HIL: Hardware In The Loop.

Table 2.1: Comparison of relevant programmable flight systems.

In 2018, Ebeid et al. presented a survey of open-source hardware and software, comparing their main features (Ebeid et al., 2018). Table 2.1 lists some relevant flight controller projects. These projects may encompass both the hardware and software development of these controllers. They range from Open Source Hardware (OSH) and Open Source Software (OSS) to proprietary commercial controllers. Furthermore, most of them support simulating their behavior either in a fully simulated environment, in a Software-in-the-Loop (SITL) setup, or in a Hardware-in-the-Loop (HITL) setup, where the autopilot code runs on the actual controller hardware. This capability enhances system validation, leading to more robust and reliable solutions.

Although some flight controllers can be programmed to perform simple autonomous tasks, such as waypoint navigation and basic obstacle avoidance, they lack the capabilities required to handle complex tasks due to limited onboard computing power. For this reason, they are combined with high-level controllers that enhance the robustness and autonomy of these vehicles, enabling them to execute complex autonomous missions.

2.2.2 High-Level Frameworks for Aerial Robotics

Providing autonomy to aerial robotics systems requires complex computing operations that cannot be handled within the FCU. The number of different components that must work together motivates the creation of software frameworks that integrate a set of components and tools to facilitate both the development and implementation of the required functionalities for these systems.

Among general-purpose frameworks, several solutions exist, such as the former Aerostack (Sanchez-Lopez et al., 2017), a ROS-based framework that integrates various computational solutions, including computer vision and motion planning, developed through extensive use in our research lab. The dependency on the former ROS version and the rigidity of defining the entire structure from the outset motivate a reimagining of this framework to improve its usability and capabilities.

The MRS UAV system (Baca et al., 2021) is another ROS-based system designed for onboard use with PX4- and DJI-compatible controllers. It supports both indoor and outdoor applications and is particularly suited for multi-robot experiments. It stands out for its robust state estimation and advanced control methods, making it well-suited for GPS-denied or complex environments. Notably, it implements multi-frame localization, allowing seamless transitions between different sensor modes. Similar to Aerostack, it is built on ROS, which limits the use of DDS-certifiable communications between modules.

The UAV Abstraction Layer (UAL) (Real et al., 2020) further standardizes the interfaces of UAVs across different autopilots, simplifying higher-level algorithm development for both simulated and real platforms. Its key strength lies in providing a consistent and unified high-level API, enhancing algorithm portability and multi-platform compatibility. However, due to its abstraction nature, it does not include essential autonomy functions such as navigation or perception,

necessitating external module integration.

For more specialized applications, Agilicious (Foehn et al., 2022) is a framework optimized for agile quadrotor flight. It features onboard vision sensors, GPU-accelerated hardware, and a versatile software stack for real-time perception and control. Its major advantage is enabling exceptionally aggressive maneuvers using onboard vision sensors. However, its reliance on custom high-performance hardware restricts broad applicability and ease of adoption in less specialized applications.

Similarly, the KumarRobotics flight stack (Mohta et al., 2018) focuses on autonomous navigation in cluttered, GPS-denied environments, performing all sensing and computation onboard without human intervention after launch. Its reliability and robustness in complex scenarios have been validated through extensive field tests. However, it is computationally intensive, and integrating new sensors or algorithms requires advanced expertise.

Also worth mentioning is the LARICS UAV ROS stack (Markovic et al., 2023), which standardizes UAV system design by clearly defining both hardware and software modules, streamlining development through modular templates. Its structured approach greatly simplifies the addition of custom functionalities. Nonetheless, the necessity for careful module compatibility checks and calibration increases initial setup complexity.

In the realm of simulation and experimentation, CrazyChoir (Pichierri et al., 2023) offers a ROS 2 toolbox for working with swarms of Crazyflie nano-quadrotors, enabling complex distributed tasks in both simulations and real-world experiments. It excels with ready-to-use swarm behavior algorithms, easing the implementation of complex group behaviors. However, its operational scope is limited to indoor environments, as it depends heavily on external localization systems such as motion capture.

Similarly, Crazyswarm2 (Preiss et al., 2017) is a toolbox designed for controlling large swarms of Crazyflies flying in dense indoor formations, achieving precise flight for up to 50 drones. It is specifically optimized for accurate swarm control through efficient communication methods. Despite its strong performance, its dependency on external positioning systems and centralized control infrastructure introduces single points of failure.

XTDrone (Xiao et al., 2020) provides a comprehensive simulation platform for various types of UAVs, facilitating the transition from algorithm testing in simulations to deployment in real UAVs. Its primary strength lies in high-fidelity simulations and extensive UAV model support. However, as a simulation-centric platform, it lacks built-in autonomy features, requiring additional development for real-world autonomous behaviors.

Similarly, RotorS (Furrer et al., 2016) is another modular MAV simulation framework designed for quick research integration and seamless transition from simulation to real MAV applications. Its key advantage is its simplicity and ease of integration with physical MAV platforms. However, RotorS primarily focuses on simulation and does not provide built-in autonomy modules, necessitating additional external development for complete autonomous capabilities.

Table 2.2 compares some relevant characteristics of these open-source frameworks. The table shows that while most frameworks are modular, only half are actively maintained. A limited number support ROS 2 as middleware, and even fewer are designed for industrial use. Approximately half of the frameworks support multi-agent operations, but multi-frame and multi-platform capabilities are less common. Very few frameworks feature a plugin-oriented¹ design, which allows for easy addition of new components and supports dynamic system reconfiguration. Furthermore, testing environments vary, with some frameworks validated only in simulations, while others have been tested in real laboratories and field conditions.

¹Plugins are dynamically loadable classes that inherit from a common interface and are loaded from runtime libraries. They enable modularity and flexibility by allowing behavior to be extended or modified without altering the application's source code or requiring explicit linking to the plugin libraries.

Framework	Middleware	Maintained	Modular	Plugin oriented	Multi-frame	Multi-agent	Multi-platform	Industrial use	Tested in
Aerostack (Sanchez-Lopez et al., 2017)	ROS	✗	✓	✗	✗	✓	✓	✓	S,RL,RO
Agilicrus (Foehn et al., 2022)	ROS	✓	✓	✗	✓	✗	✗	✗	S,RL,RO
CrazyChoir (Pichierri et al., 2023)	ROS 2	✓	✗	✗	✗	✓	✗	✗	S,RL
CrazySwarm2 (Preiss et al., 2017)	ROS 2	✓	✓	✓	✗	✓	✗	✗	S,RL
KumarRobotics (Mohata et al., 2018)	ROS	✗	✗	✗	✗	✗	✓	✓	S,RL,RO
LARICS-uavrostack (Markovic et al., 2023)	ROS	✓	✓	✗	✗	✗	✗	✗	S,RL
MRS UAV System (Baca et al., 2021)	ROS	✓	✓	✗	✓	✓	✓	✓	S,RL,RO
RotorS (Furrer et al., 2016)	ROS	✗	✓	✗	✗	✗	✗		S
UAL (Real et al., 2020)	ROS	✗	✗	✗	✓	✗	✓	✓	S,RL,RO
XTDrone (Xiao et al., 2020)	ROS	✓	✓	✗	✗	✗	✗	-	S

Table 2.2: Comparison of relevant open-sourced software stacks for aerial robotic systems. Industrial use is only evaluated for the frameworks that support real flights. S: Simulation; RL: real experiments in the laboratory; RO: real experiments outside the laboratory.

2.3 Trajectory Generation for Agile Flights

Trajectory planning for multirotor UAVs has evolved considerably over the past decade. Early approaches relied on classical methods such as shortest-path algorithms and bang-bang control strategies (Bouabdallah and Siegwart, 2007; Mahony et al., 2012). These techniques typically treat the drone as a point mass and generate collision-free paths through simple waypoint-based planning. However, they neglect the underactuated dynamics of multirotors, leading to trajectories that are often jerky and require conservative maneuvers to ensure safety in flight.

To improve smoothness and dynamic feasibility, optimization-based methods were introduced. In particular, polynomial trajectory planning methods—such as minimum-snap techniques—exploit the differential flatness of quadrotors to generate trajectories that minimize higher-order derivatives (e.g. snap or jerk) while ensuring continuity across waypoints (Mellinger and Kumar, 2011; Richter et al., 2016). These methods solve quadratic programs efficiently, enabling near real-time performance; however, they typically rely on pre-assigned time allocations and may not be inherently time-optimal. Additionally, while the trajectories are smooth, they lack the intrinsic ability to handle unforeseen obstacles or changes in the environment.

Nonlinear optimization methods address these issues by incorporating the full dynamics of the quadrotor into the planning process. Techniques such as the Complementary Progress Constraints (CPC) trajectory generator formulate time-optimal or energy-optimal control problems that fully respect actuator limits and dynamic coupling (Foehn et al., 2021; Penin et al., 2017). Although these approaches can theoretically exploit the complete performance envelope of the UAV, they incur a high computational cost—often requiring minutes or even hours of offline computation—thereby limiting their applicability to real-time scenarios.

Real-time adaptation methods, including Model Predictive Control (MPC) and sampling-based planners (e.g., kinodynamic RRT variants), offer a middle ground by continuously re-planning trajectories in response to dynamic environments (Chen and Chen, 2020; Karaman and Frazzoli, 2011). MPC-based approaches solve a short-horizon optimization problem at high frequencies (typically every 20–50 ms) to adjust control inputs in real time, while sampling-based methods generate motion primitives that respect dynamic constraints and can be updated on the fly. Hybrid strategies like Model Predictive Contouring Control (MPCC) further merge planning and control objectives to allow for aggressive, near time-optimal flight (Romero et al., 2021). Despite their adaptability, these real-time techniques require significant computational resources and carefully tuned solvers to handle the high-frequency updates, especially in cluttered or rapidly changing environments.

The most recent advances leverage data-driven and learning-based methods to mitigate some of the computational challenges. Reinforcement learning (RL) and imitation learning have been applied to enable UAVs to learn optimal or near-optimal trajectories from simulated or demonstrated data (Song et al., 2021). Although training such models is computationally demanding, the inference phase is extremely fast—allowing for trajectory generation or adjustment within milliseconds. End-to-end deep learning approaches further aim to bypass traditional planning pipelines by directly mapping sensory inputs to control commands, though these methods still struggle with safety guarantees and generalization in unstructured environments.

In terms of computational cost, classical methods are very lightweight and rely on simple closed-form solutions or graph searches, making them attractive for systems with limited processing power. Polynomial planners, while more demanding due to the solution of quadratic programs, remain efficient enough for real-time use. In contrast, nonlinear optimization methods—due to their non-convexity and large number of decision variables—are typically reserved for offline computation. Real-time adaptation techniques, such as MPC and sampling-based planning, require high-frequency updates and hence powerful onboard computation to manage rapidly

changing environments. Finally, learning-based methods shift the bulk of the computational expense to the training phase, enabling fast inference that is well-suited to real-time applications (Karaman and Frazzoli, 2011; Penicka et al., 2022).

Overall, the progression from classical to learning-based trajectory generation illustrates a clear trade-off between computational simplicity and dynamic performance. While early methods offer low computational overhead at the expense of smoothness and agility, optimization-based approaches achieve smoother, more efficient trajectories with increased computational demands. Real-time adaptation techniques provide essential responsiveness in dynamic settings, though they require significant processing power, and learning-based methods promise ultra-fast decision-making once trained but still face challenges in robustness and safety. The integration of these approaches continues to be a vibrant area of research, driving advancements toward fully autonomous, agile, and safe multirotor UAV flight.

2.4 Collaborative SLAM

Centralized CSLAM approaches are based on multiple single-robot front-ends with a common back-end. LAMP 2.0 (Chang et al., 2022) is a centralized multi-robot SLAM system that is adaptable to different input odometry sources, featuring a robust and scalable loop closure detection module as well as an outlier-robust back-end based on Graduated Non-Convexity (GNC) for pose graph optimization. It is designed for operation in challenging large-scale underground environments. Maplab 2.0 (Cramariuc et al., 2023) is a large-scale multi-robot multi-session mapping system that integrates non-visual landmarks and incorporates a semantic object-based loop closure module into the mapping framework. Additionally, COVINS-G (Patel et al., 2023) is a generalized back-end for collaborative visual-inertial SLAM built upon the COVINS (Schmuck et al., 2021) framework, enabling compatibility with any arbitrary VIO front-end. However, the drawbacks of centralized architectures lie in their dependence on reliable communication with the common back-end—if that node fails, the entire system collapses—and in their limited scalability when applied to large robot teams.

Within decentralized approaches, most efforts focus on visual-based SLAM. Deutsch et al. (Deutsch et al., 2016) proposed a framework that relies on a Bag of Words (BoW) representation of the keyframes obtained with an RGB-D camera to enable collaborative mapping for existing single-robot SLAM systems. Kimera-Multi (Tian et al., 2022) also employs BoW and requires a *Robust Distributed Initialization* to align all robot poses within a shared global coordinate frame. This system builds a globally consistent metric-semantic 3D mesh model of the environment in real time, remains robust to incorrect loop closures, and relies solely on local communication. Each robot constructs a local trajectory estimate and mesh using Kimera (Rosinol et al., 2020), and when two robots come within communication range, they initiate a distributed place recognition and robust pose graph optimization protocol based on a distributed GNC algorithm. Swarm-SLAM (Lajoie and Beltrame, 2023) further introduces a novel inter-robot loop closure prioritization technique based on algebraic connectivity maximization, which reduces communication overhead and accelerates convergence. It also implements a decentralized approach to neighbor management and pose graph optimization suitable for sporadic inter-robot communication, while supporting various sensor types using descriptors such as CosPlace (Berton et al., 2022) and NetVLAD (Arandjelović et al., 2018) for images, and Scan Context (Kim and Kim, 2018) for LiDAR scans. In the realm of decentralized LiDAR SLAM, solutions like DiSCo-SLAM (Huang et al., 2022) utilize the lightweight Scan Context descriptor and incorporate a two-stage global and local optimization framework for distributed multi-robot SLAM, achieving stable localization even under unknown initial conditions during inter-robot loop closure searches. Similarly, DCL-SLAM (Zhong et al., 2023) leverages a lightweight global descriptor, LiDAR-Iris (Wang et al., 2020), for loop closure

detection, employs a three-stage data-efficient distributed loop closure approach to estimate relative pose transformations between robots, and applies a two-stage distributed Gauss-Seidel (DGS) optimization.

Current decentralized collaborative SLAM systems, however, struggle to effectively use well-mapped areas generated by other agents, leading to map redundancy and costly re-mapping. To address this, Bänninger et al. (Bänninger et al., 2023) propose a strategy to efficiently share areas mapped by different agents in a decentralized, peer-to-peer fashion. Their approach achieves a globally consistent estimate through distributed bundle adjustment using the Alternating Direction Method of Multipliers (ADMM). By leveraging covisibility information between keyframes instantiated by different agents, this method transfers local sub-maps on-the-fly, thereby addressing map redundancy while maintaining consistency and scalability. The incorporation of semantic information from the environment could further reduce the volume of information exchange.

Another important challenge in CSLAM is the combination of local maps into one global representation when the initial pose of each robot is unknown. Single-robot SLAM approaches such as those in (Yu and Shen, 2022), (Tao et al., 2024), and (Dubé et al., 2018) utilize semantic object detection to enhance robustness and accuracy by reducing the loop closure search space. Motivated by these successes, several CSLAM approaches, for example in (Yu et al., 2020) and (Vaščák and Herich, 2023), employ object-level semantics to robustify inter-robot map merging. In terms of map fusion, the framework presented in (Frey et al., 2019) efficiently merges maps in semantic object-based SLAM by detecting duplicate constellations of landmarks. In (Prabhu et al., 2024), semantic place recognition on sparse representations of tree trunks—represented by cylinder models extracted from LiDAR sensor data—was used to calculate relative transformations between multiple robots. However, these works generally do not utilize hierarchical information alongside semantic data to enhance inter-robot map merging and typically require densely populated indoor scenes with a wealth of relevant objects.

Recent research has also aimed to leverage semantic information by integrating scene graphs with SLAM. Hydra-Multi (Chang et al., 2023) is the first multi-robot spatial perception system capable of building a hierarchical 3D scene graph online from various sensor data. Based on the Hydra framework (Hughes et al., 2022), this centralized architecture includes a hierarchical loop closure detection module that utilizes 3D scene graphs, a frame alignment module that estimates the initial poses of the robots, and an alignment-optimize-reconcile framework that uses GNC to optimize multi-robot 3D scene graphs. More recently, CURB-SG (Greve et al., 2023) has been introduced as the first solution to create large-scale 3D scene graphs of dynamic urban scenes for automated driving by leveraging collaborative mapping techniques and panoptic LiDAR data. Despite these advancements, it is important to note that none of these scene graph-based approaches are decentralized, which limits their scalability.

Part II

Architecture and Frameworks for Robotic Systems

Chapter 3

Reference Architecture for Robotic Systems: An Incremental Approach

3.1 Introduction

Developing autonomous robotic systems is a challenging task that requires extensive expertise in many different areas, such as mechanics, control systems, sensor integration, and AI algorithms. In addition to developing these individual components, it is crucial to integrate them effectively, ensuring that all algorithms receive and produce the appropriate inputs and outputs for the robotic system to function correctly. How each component, algorithm, or functionality of the robotic system interacts with the rest is defined by the *system architecture*.

Each system has its own inherent architecture, based on all the decisions that developers make throughout the engineering process. A well-designed architecture can enhance the life cycle of a robotic system by making it easier to create, extend, modify, and maintain. There are multiple aspects of a system architecture, depending on which part of the robotic system is being considered. For example, the hardware architecture defines how each physical component is connected. In this thesis, we will focus on the software architecture of robotic systems, specifying the required software components and how they should be interconnected.

In this chapter, we propose a *reference architecture* for developing autonomous robotic systems. This means that we will not cover the system architecture of a specific robotic system but rather present an abstraction of the types of subsystem and their functions necessary to build a robotic system with a certain level of autonomy.

We propose an incremental layered reference architecture that evolves from the classic closed-loop control schema into a multi-robot cognitive architecture. The term “incremental” means that multiple types of systems can be developed using this reference architecture, and their capabilities can be enhanced by adding layers on top of existing ones.

First, we have categorized systems based on the information they use and how they utilize this

information for decision-making. Using these criteria, we define different levels of systems, which we refer to as Tiers. Each Tier includes all the capabilities of the previous levels and expands upon them. In this thesis, we differentiate them into four Tiers:

Tier I Feedback Controlled Systems: In these systems, the emphasis is on generating the proper control signals to the actuators so that the system evolves into a desired state. To achieve robust results, it is necessary to provide sensory feedback on the state of the robot so that the controller can adapt its response in case of disturbances or inaccuracies in the system model. The decisions of these systems are the commands given to the robot, based on its state.

Tier II Task Decomposition-Based Systems: In these systems, the goal of the system is decomposed into small *tasks*, each with an assigner and an executor. This decomposition allows increasing the complexity of the tasks performed by the system while encapsulating the responsibility for task execution within specific components and delegating task coordination to a separate module that plans and assigns the tasks. In this tier, the system has to decide which task shall be taken next, normally based on a pregenerated plan.

Tier III Cognitive Systems: In these systems, the aim is to improve decision-making capabilities in dynamic and unexpected situations by enhancing the system's cognition. This cognition can be summarized as the ability of the robotic system to *understand* itself and its environment in order to take the best action at each moment. This implies that the system shall have a certain level of *awareness* about the outcomes of its actions and how these affect the pursuit of its goal.

Tier IV Multi-Agent Systems¹: In these systems, multiple agents operate together to achieve a common goal. Each agent should be capable of operating independently with a certain level of autonomy. However, by combining the capabilities of multiple agents, new functionalities emerge within the system, improving its overall performance. In this tier, multiple agents can contribute to deciding what the system will do, increasing the complexity of the decision-making mechanisms.

These are very brief and concise introductions to what we can expect from each type of system. Below, we will delve into a more detailed description of the desired capabilities that each of these systems should have and explain the subsystems required at each Tier to fulfill them.

In this reference architectural proposal, we cover only the functionalities and interfaces of each subsystem and component. The specific components, algorithms, or implementations within each subsystem may be tightly coupled with the particular robotic application. Therefore, in this chapter, we will remain at this level of abstraction.

¹Multi-agent systems can be built on top of Tier I and II systems, as will be further demonstrated in Chapter 4. However, in this thesis, we will focus on multi-robot system interaction based on knowledge exchange (*Understanding*), so we have decided to consider them as Tier IV.

Example: Illustrating the concepts

The proposed architecture is designed to be generally applicable to any robotic system. However, to help clarify some of the concepts presented below, we will cover a concrete robotic application example that will aid in grounding the concepts elaborated in this chapter.

The robotic application showcased is **Photovoltaic panel inspection via aerial imagery**. In this inspection, the goal of the robotic system is to acquire geo-referenced aerial images (typically RGB or IR) covering the full area of a PV power plant. The system must be fault-tolerant and adapt to environmental changes such as wind.

The suitability of this application for exemplifying the proposed architecture lies in the fact that its requirements can be relaxed, allowing all the presented architecture tiers to fulfill the goal at the expense of a less general, robust, or reliable approach.

3.2 Tier I: Feedback controlled systems

Each robotic system must be able to move in some way, following the commands provided by an operator. This is the simplest capability that all robotic systems have in common and was the first challenge in designing these systems. To achieve this, every robotic system must have the ability to interface with hardware components, the ability to sense its state, and the ability to control its actuators to follow a reference. We will formalize this capabilities as follows:

Tier I capabilities

- C1.1 Interface with the hardware actuators and sensors from the robot.
- C1.2 Estimate the state of the robotic system based on the sensory input.
- C1.3 Given an input reference, generate the actuator commands required for follow it precisely, even with external disturbances.

These capabilities are not independent of each other; on the contrary, they are tightly coupled and interconnected. Therefore, to achieve the capability of *governed movement*, all of them must work in synchrony.

In this section, we propose a system architecture pattern for designing a Tier I system, decomposing it into three subsystems, each responsible for one of the previously enumerated capabilities: the *Hardware Interface*, the *State Estimation*, and the *Motion Control* subsystem; see Figure 3.1.

3.2.1 Hardware Interface Subsystem

In order to be able to operate a real robot the architecture has to interface with the collection of drivers, tools, or APIs provided by the robot manufacturers, or the required tools to control the hardware actuators and access to its sensory measurements.

This subsystem is the point of contact for the software components with the physical platform, so if this system fails, the rest of the architecture will lose the control of the robot. This makes this subsystem to require error-handling procedures and well-defined procedures, for avoiding unwanted results.

Logically, as is the interface with the hardware, this subsystem is also the fastest way of commanding actions to the robotic platform, which means that is the part of the architecture in charge

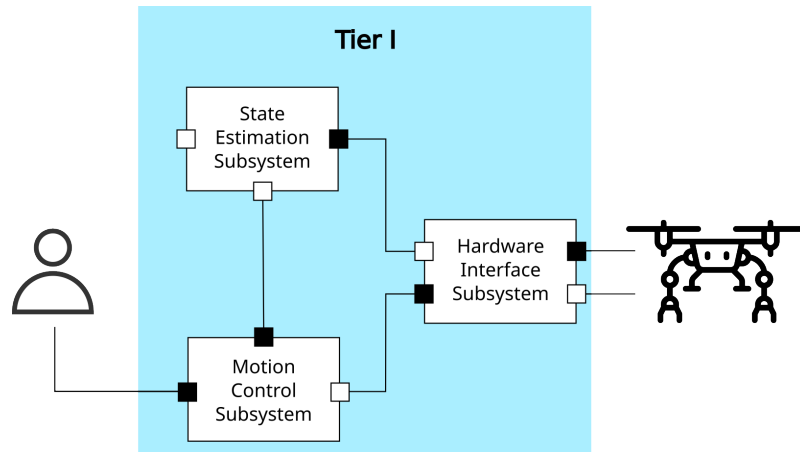


Figure 3.1: Tier I Systems Architectural Proposal Schema. In the figure, the boxes represent the three subsystems presented at this tier. Black and white cubes represent the inputs and outputs of each subsystem, respectively. The user is positioned on the left side of the figure—who will provide the references to the controller—while the robotic platform is on the right—who will obey them.

of handling emergency signals, such as an emergency stop that requires the fastest response possible.

Aside for communicating with the hardware components, this interface has to communicate with the rest of the subsystems, exposing all the information obtained from the different sensors operating and the available commands that can be used for controlling the robot.

Example: Aerial Platform interface

For interfering with the drone, the interface shall provide the data from the IMU, the GPS, the altimeter, the battery level and the camera sensors. On the other hand, the system shall provide the speed references for each rotor of the quadcopter and the motors of the gimbal on which the camera is mounted.

As this subsystem interacts between the physical (hardware) and logical (software) worlds, bridging both, the inputs received from the hardware modules are outputs of the module communicated to the rest of the subsystems, and vice versa. The hardware interface subsystem acts as a bridge that connects both worlds

Taking this into consideration, we will enumerate the inputs and outputs through which this module interfaces with the robotic platform, as well as the corresponding inputs and outputs for the rest of the system.

- Inputs received from the robotic platform / Outputs sent to the rest of the subsystems:
 1. Sensor Measurements: The set of all data retrieved from the sensors.
 2. Status Signals: Information about the functioning of the hardware components, such as the operating mode or configuration details.
 3. Metadata: Information about the sensory input received, such as errors or the timestamp of when the measurement was taken.
- Outputs sent to the robotic platform / Inputs received the rest of the subsystem:
 1. Actuator Commands: Control actions for commanding the hardware actuators.
 2. Alarm signals: Commands like emergency stops that can be handled directly by the hardware components.

3. Configuration Commands: Commands that modify how the hardware components operate.

3.2.2 State Estimation Subsystem

Once the raw information from the sensors is acquired, multiple actions can be taken depending on what information about the state of the system is required. As stated previously, the information required from the system depends on what the system will do and what needs to accomplish that.

In a robotic system, there are two aspects that can be observed and measured: the environment and the robot itself. However, in robotics, one of the most important aspects to determine is where the robot is in relation to the environment the situation. At the end of the day, the objective of robots is always related to interacting with their environment in a specific way. This interaction can be physical—such as carrying a load or touching a surface—but it can also be informative—such as taking photographs of an interesting point or recreating a map of a building. The relative positioning of the robot with respect to the object of interest is crucial to achieving the desired results in both cases. That is why knowing the position and orientation of the robot normally is part of the state of the robotic system.

The state of the system refers to the set of variables that define all the relevant information about the operation of the robotic system, some examples are, where the robot is and whether it has any dynamics that are important for predicting the evolution of the system.

As can be inferred from its name, the state estimation subsystem is the part of the system architecture responsible for estimating the state of the robot based on the available sensory inputs.

Example: Aircraft State

For this application, the basic state information needed includes the GPS coordinates of the UAV, its speed, and the orientation of both the aircraft and the gimbal. To follow a path formed by a set of GPS waypoints at a certain speed, the aircraft must know its location and current speed. Finally, to capture a good image of a panel, the camera should be oriented perpendicular to the panel surface, making it necessary to know the gimbal's orientation for the mission. All components of the state can either be measured directly from sensors or estimated by fusing and integrating sensor inputs.

The *State Estimation Subsystem* has the following inputs and outputs:

- Inputs:
 1. Sensor Measurements: The set of all data retrieved from the sensors.
 2. Metadata: Information about the sensory input received, such as errors or the timestamp of when the measurement was taken.
- Outputs:
 1. State Estimation: The estimated values of the variables that define the state of the robotic system. This estimation also includes the uncertainty associated with each variable.

3.2.3 Motion Control Subsystem

Motion is the most relevant capability that robots have; this allows robotic systems to interact with the environment that surrounds them. This interaction with the real world is bidirectional: just as the robot is capable of producing changes in the environment, the environment can interfere with

the robot. These disturbances can cause the robot to not perform the movement as expected, which can lead to failure in pursuing its goal.

This problem has been covered for scientifics and engineers long time ago when process control systems were invented, and the control theory was created. One of the best ways of achieving a robust and stable behavior when controlling a machine is by continuously measuring the state of the system and adapt the actuator commands to them. This is called feedback control loop and its the base of the majority of machines and systems that want to “control” the dynamic behavior of a system.

Within our architecture, the *Motion Control Subsystem* is in charge of generating the sequence of commands that make the robotic system to move from a current state into a desired one. The generated commands are then fed into the Hardware Interface Subsystem who will translate them conveniently to command the hardware actuators. The current status of the system is being continuously updated by the State Estimator Subsystem, so that the controller can adapt based on it.

For Tier I systems the reference signal can be considered to be sent directly by the user. However, for higher Tier architectures these references other subsystems are the ones that will be in charge of generating the desired references, as will be seen in the next section.

Example: Aircraft Control

To capture images of the power plant, the UAV must be able to navigate to an aerial position suitable for image acquisition. To achieve this, the system must convert the input reference objective position (a GPS coordinate) into motor signals (RPMs) that enable the aircraft to move to the desired position in a stable manner. As a feedback controller, the motion control subsystem receives feedback from the estimated state, allowing it to determine the desired system evolution and adapt to disturbances.

The Motion Control Subsystem has the following inputs and outputs:

- Inputs:
 1. State Estimation: The estimated values of the variables that define the state of the robotic system. This estimation also includes the uncertainty associated with each variable.
 2. Motion Reference: The reference state to which the robotic system is intended to evolve.
- Outputs:
 1. Actuator Commands: Control actions for commanding the hardware actuators.

3.3 Tier II: Tasks Decomposition

From Tier I systems, we fulfill the essential capabilities required to provide motion capabilities to a robotic system. However, expressing a mission in terms of concrete motion references is difficult and can be error-prone. Additionally, it can be challenging for a human to understand the intention behind each movement or, more colloquially, what the robot will do given a concrete motion reference.

As humans, when we face a mission, we decompose it into semantically relevant tasks with a concrete meaning, which helps us express a routine as a sequence of tasks that must be performed in a specific order. This process can lead to an atomization of the list of actions a robotic system

can execute, allowing for the generation of mission plans by properly combining these atomic tasks.

The Tier II System Architecture encompasses systems whose operation can be expressed as a set of activated *tasks*, each with a dedicated component responsible for its execution. To perform a mission, a mission plan must be generated, specifying which *tasks* will be performed and in what order. Furthermore, this mission planning can be *reactive*, meaning the plan can be modified during runtime based on the system's state. Combining this new capabilities we can formulate again what are the required capabilities for a Tier II system as follows:

Tier II capabilities

- C2.1 Include all the capabilities from Tier I systems.
- C2.2 Perform predefined tasks in a repetible and reliable way.
- C2.3 Follow a mission plan formed by a sequence of task executions.

In the proposed system architecture, we recognize that some *functions* of robotic systems are continuous and do not have a specific goal that, once achieved, leads to their conclusion. A simple example is the acquisition and processing of information to generate useful data for other modules. We propose unifying these two types of discrete tasks and continuous functions under the term *Behavior*, ensuring a consistent approach to their management across the subsystems of the architecture.

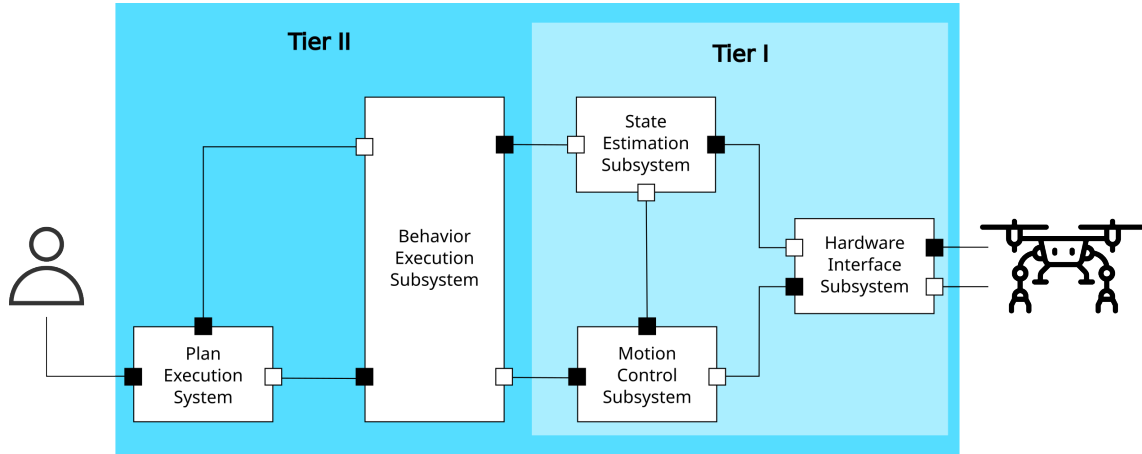


Figure 3.2: Tier II Systems Architectural Proposal Schema. In the figure, the boxes represent the subsystems of the architecture. The submodules corresponding to different Tiers are separated by blue shades. Black and white cubes represent the inputs and outputs of each subsystem, respectively. The user is positioned on the left side of the figure, while the robotic platform is on the right.

To achieve this, we propose the addition of two subsystems to our architecture, designed to allow the system to execute missions specified in terms of required behaviors and their correct execution order. We name these subsystems the *Behavior Execution Subsystem* and the *Plan Execution Subsystem*, see Figure 3.2. In the following subsections, we will elaborate on the details of each of these subsystems.

3.3.1 Behavior Execution Subsystem

The Behavior Execution Subsystem is responsible for managing all the behaviors that the system is capable of performing. Unlike the previous subsystem, there is no specific set of capabilities that this subsystem must provide. Each concrete system implementation will have its own set of *behaviors*, depending on the application.

The proposed subsystem is formed by multiple components—*behaviors*—, each of them responsible to provide the set of functionalities needed for performing a concrete task. Each *behavior* has a set of properties and qualities that are common between them and will rule how these components interact with the rest of the system architecture:

1. **Homogeneous interface:** Every behavior provides a standardized interface for inputs and outputs. This uniformity ensures that behaviors can be integrated, replaced, or extended without the need for customized interfacing, thereby simplifying system maintenance and scalability.
2. **State machine execution:** Every behavior component's execution can be modeled using an identical Finite-State Machine. This common model encapsulates states such as initialization, active execution, pause, and termination, providing a structured framework that governs how behaviors transition and evolve during operation.
3. **Self-supervision:** Each behavior is equipped with internal monitoring mechanisms that continuously assess its performance and operational status. This self-supervision allows behaviors to detect and handle errors or deviations autonomously, enhancing overall robustness and enabling adaptive responses.
4. **Composability:** The architecture adopts a paradigm in which higher-level behaviors can be composed using lower-level behaviors. This hierarchical control mechanism enables the development of complex behaviors.

These properties are designed to allow the scalability of the complexity of the tasks that the robotic system can perform while easing the interaction of them with the rest of the architecture. In the next subsection, how these components are orchestrated to perform a concrete mission.

Example: Behaviors for PV Inspection

The inspection mission can be subdivided into different tasks, primarily categorized into two types: motion-related and mission-related tasks. Motion-related tasks include *takeoff*, *hovering*, *moving to a position*, *following a path*, and *landing*. On the other hand, mission-related tasks include *capturing an image*, *orienting the gimbal*, and *generating paths to cover an area*. Although these behaviors are essential for PV plant inspections, they are also applicable to a wide range of scenarios and have relevance beyond this specific application.

3.3.2 Plan Execution Subsystem

Using *behaviors* as the building blocks for describing the operation of a robotic system—expressed as a sequence of activations and deactivations of behaviors along with their corresponding parameters—provides an expressive and human-readable way of programming robot applications.

However, an autonomous system typically needs to adapt to changes in its environment or choose between multiple options depending on its current state. With the previously presented subsystems, there is no capacity for decision-making or selecting one behavior over another based on the situation.

It could be argued that behaviors themselves can be dynamic and adapt to circumstances. For example, a path planner behavior could continuously recompute the optimal path while the robot moves and detects new obstacles. Additionally, since each behavior can be composed of others, a behavior could decide whether to invoke one behavior or another depending on the situation.

However, to maintain coherence throughout the architecture, the logic implemented within a behavior must remain consistent with the definition of a "behavior" itself. This means that the same logic will always apply to a given behavior, regardless of the mission or application in which it is used. The best way to support the scalability and complexity of longer missions is through a higher-level subsystem that orchestrates how each behavior is invoked.

The Plan Execution Subsystem is responsible for this orchestration. Given a mission plan generated by the user, this subsystem communicates with the Behavior Execution Subsystem to activate or deactivate the required behaviors in the correct order. It can group sets of behavior executions into structured sequences and include logic control statements, allowing the mission plan designer to create adaptable plans.

Example: Mission Plan

The most simple mission plan that can be done for the proposed application is a sequence of blocking activations of the aforementioned behaviors like:

Mission plan 1: Naive Path Following

Input: *path, origin*

takeoff();

foreach *point in path* **do**

 go_to(*point*);

 hover();

 take_photo();

end

go_to(*origin*);

land();

In this case, the mission does not adapt to any changes; it is simply a preprogrammed path. The use of behaviors helps both in the design phase and in providing explainability about what the system will do.

3.4 Tier III: Cognitive Systems

In Tier II systems, the evolution of the system produces changes in the information obtained by the sensors, changes in the environment, or changes in its own state. All these changes affect the internal states of each subsystem; however, there is no explicit and accessible representation of the knowledge the system possesses or its evolution.

Although information flows between subsystems, as evidenced by actuation based on the system's state, this information is typically implicit and lacks a structured representation of the knowledge the system possesses. Tier II systems rely on the interaction of simple *behaviors* without requiring an internal model of the environment.

The absence of explicit representations can lead to challenges in tasks requiring high-level reasoning or planning, as the system may struggle to process abstract concepts or adapt to new situations beyond its predefined behaviors.

In Tier III systems, knowledge representations are used to model the world and the self, informing decision-making processes. Knowledge can be represented within *Models*, which are

stored in what is called a Knowledge Base (KB), allowing any subsystem to consult or update them.

The system's knowledge can be acquired during runtime or provided during the engineering phase, allowing the system to use this knowledge from the start of its operation.

This knowledge can enhance the system's *Understanding* of itself and its environment, providing some level of *Awareness* about how its actions can affect the environment or vice versa. This can improve decision-making processes and enables the system to operate in a more reliable and adaptive manner.

In order to take a decision there are always a set of *values* that are used for knowing how good or bad is taking one decision or other. Being able to judge which is the best action possible means the system is able to predict the "outcome" of taking this action and evaluating which is the value obtained from doing this. At the end of the day, all robotics system are built for providing a *value* to someone, so all the missions can be reduced to the set of values that each robotic system should have.

To sum up, we can enumerate three new important components that shall be added into the Tier II architecture to be able to take advantage of the Awareness and Value capabilities:

In the proposed architecture, this awareness can be achieved by adding a new subsystem—a knowledge base—but also requires the use of specific types of behaviors whose objective is to create and update the models stored in the knowledge base: the *Understanding Behaviors*. Additionally, it necessitates the integration of submodules within some of the existing subsystems to leverage this knowledge: the *Inference Components*.

Tier III capabilities

- C3.1 Include the capabilities of Tier I and II systems.
- C3.2 Build structured models from the perceived signals to enhance comprehension.
- C3.3 Store and update relevant knowledge about the system and the environment to enable sound inference.
- C3.4 Evaluate the convenience of actions based on a predefined value system.

In the proposed architecture, two new subsystems are included: one responsible for knowledge management, the *Knowledge Base Subsystem*, and the *Mission Planning Subsystem*, whose objective is to create and modify the mission plan according to the system's goal and available knowledge.

Additionally, to fully leverage this Tier, some extensions must be made to other subsystems by integrating new Inference Componentss (ICs) across all subsystems that provide knowledge or use the models available in the *Knowledge Base*; see Figure 3.3.

In the following sections, we will provide further details on these new architectural components and subsystems and explore how the presented terms relate to the meaning of *Understanding* and *Awareness* within a robotic system.

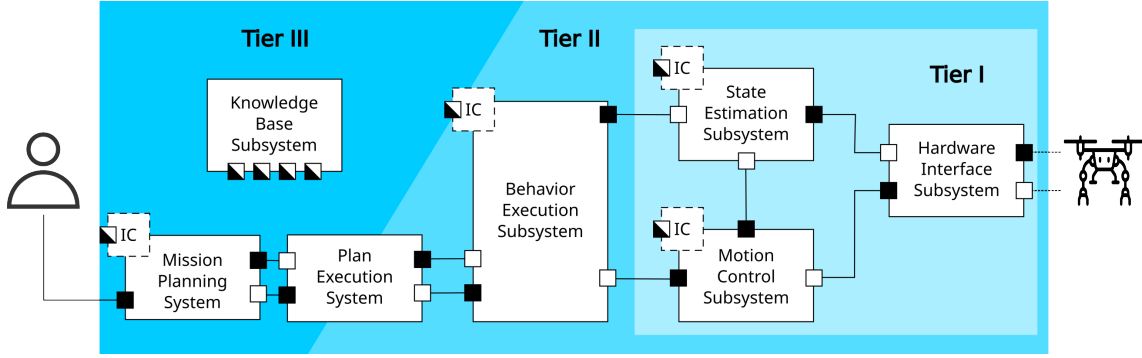


Figure 3.3: Schema of a Tier III Architecture. The IC blocks represent the Inference Components that interface with the Knowledge Base Subsystem for both, adding or retrieving models from the knowledge base of the system. This components can be used in different subsystems of the architecture to increase the awareness of the overall system.

3.4.1 Understanding in a knowledge base subsystem

Awareness and *Understanding*, aside from being two resonant terms, aim to represent two desirable cognitive capabilities of a robotic system. In the following sections, we will elaborate on these concepts.

In this section, we will elaborate on the concept of *Understanding*, starting from the definition developed within the CORESENSE Project (Sanz et al., 2023).

Definition

A subject S understands a phenomenon ϕ if it has a set of models M_ϕ of ϕ and those models can be used to make sound inferences about the phenomenon.

This definition relates a system's capacity to understand something (a phenomenon) to its ability to generate and maintain internal representations (models) of all entities or processes relevant to its functioning.²

In the literature, we can find references to *World Models* (Taniguchi et al., 2023), which enable the robot to anticipate how the world evolves in response to its actions, facilitating informed decision-making and planning. In this type of model, the object of interest is the environment and its evolution, which is called Exteroception. The other relevant object of interest in the robotic system is the robot itself; the perception of the self is also known as Proprioception.

Based on this, if a robotic system is intended to have some degree of *Understanding*, it must include mechanisms that allow it to build, maintain, and collect models of both the world and itself.

In order to fully take advantage of these models, they must be accessible to any other subsystem of the architecture so that they can be used to enhance robustness or performance based on this knowledge. As these systems must be audited, verified, or programmed by humans, these representations must be designed in a way that humans can understand or, at the very least, predict the decision-making processes based on the information available to the system.

The *Knowledge Base Subsystem* is responsible for collecting all the models generated by the different subsystems and integrating them in an accessible manner for the rest of the modules. The set formed by all these models—concerning mission values, the environment, and the system

²Although the definition is not limited to a specific type of phenomenon, in this work, we consider that robotic systems have a concrete objective or operational environment relevant to their operation, and it is what is useful to focus on while developing these systems.

itself—constitutes the World Model, which enables sound inferences about the relevant entities necessary for the successful fulfillment of the mission assigned to the robotic system.

Example: Models for Aerial Inspection

For PV aerial inspection, we can create models for both the UAV and the power plant. Regarding the UAV, a model of its dynamics is typically used within the motion controller. Model Predictive Controllers (MPCs) rely on this model to generate optimal control outputs. This dynamic model can also be combined with a battery discharge model, enabling the computation of an estimated flight time based on motion references. If the system is able to make accurate predictions of this flight time, we can say that the system *understands* the influence of the flight dynamics and the control outputs in the capacity of the battery.

On the environmental side, UAV movement can be affected by obstacles such as power towers or by wind, which influences flight dynamics. A power tower can be modeled as a geometric volume positioned within the map, whereas wind can be represented as a variable force field. In these cases, if the system is able to avoid the power tower even in the presence of wind, it means that the system *understands* where the power tower is, what its dimensions are, and how the wind can affect the aircraft movement.

The way each entity is modeled is closely related to how the model will be utilized by the rest of the subsystems.

Inference Components

The existent models within the *Knowledge Base Subsystem* lacks utility if they are not used for improving themselves, the performance of other subsystems, or for taking decisions based on it. In all cases there is a need of making some type of update or inference on the models.

Inference Components (ICs) are architectural components whose objective is to facilitate the communication of the different subsystems with the knowledge base, allowing them to add new knowledge or perform inferences on the models.

Different types of inference can be performed to different type of models, depending on how the model is expressed and what information include. The inference components have the ability to adapt the inference mechanisms depending on the concrete model of interest.

Example: ODE Solvers vs. Querying

Following the previous example, when considering a dynamic model of the aircraft, the model can be expressed as a set of ODEs. To infer the evolution of this model, a *solver* is required to evaluate the equations and obtain a prediction. This solver performs a specific type of inference that is compatible with this model representation.

On the other hand, some inferences can be simpler, such as merely evaluating the existence of a model. This type of inference requires a different logical approach. ICs provide abstraction mechanisms that help developers perform the correct inference depending on the nature of the model.

3.4.2 Awareness by Mission Planning

In the latest years, robotic researchs have been working in developing systems that exhibit improved *Situational Awareness* and *Self-Awareness*. The main difference between Situational and Self Awareness relays on the subject of the awareness. In the case of the Situational awareness the object of interest is the environment and the evolution of it (Exteroception). Meanwhile, in the Self-Awareness the intetest relays in the status of the agent itself (Propioception).

The “Situational Awareness” definition provided by provided by Endsley in 1988 (Endsley, 1988) (Endsley, 1995):

“Situational Awareness is the perception of the elements in the environment within a volume of time and space, the comprehension of their meaning, and the projection of their status in the near future.”

Self-awareness in robotics refers to the ability of a robot to have an accurate self-model and understand its own body and limitations. Self-modeling is a primitive form of self-awareness, and if a robot has an accurate self-model, it can function better in the world by making better decisions.

In this thesis, we will name *Awareness* for referring the union between Situational Awareness and Self-Awareness of the robotic system. In order to homogenize the Self-Awareness with the Situational Awareness, we adapt the Endsley’s three phases of situational awareness into a self-Awareness way(Endsley, 1995):

1. Perception: This constitutes the measures obtained not only by proprioceptive sensors but also with exteroceptive sensors if the agent can be observed through them. For example, a robot that can see itself through a mirror.
2. Comprehension: In this part is where the modeling part is taken into account, the measures obtained by the robot are combined with the models for creating, validating or refining the model of the system, and evaluate the status of the system with them.
3. Prediction: Using the model obtained in the previous step, accurate predictions of the evolution of the system can be made.

The former definition has similarities with the *Understanding* previously defined. So, what is the difference with the Awareness. The difference is that having *Awareness* means not only *Understand* something, but to know what implications this have for the robotic system.

Definition

A subject S is **aware** of a phenomenon ϕ if it **understands** it and can relate its implications according to a **value** system.

Being able to predict the evolution of the state of the robot or the environment implies a certain level of understanding about the self and the situation. However, a robotic system has to provide a value, has a goal, and each state of the robotic system provides a value.

Example: Understanding vs. Awareness

Consider a UAV flying over a PV power plant. The aircraft is following a path and has models to predict how much battery is required to cover the path. This implies that the system understands the evolution of the battery.

However, awareness is achieved when the system knows that the battery is required not only for covering paths but also for taking images, which is the final goal of the system. Moreover, if the UAV runs out of battery in the air, it can crash, which is highly undesirable. The system possesses values regarding what is good and what is bad and can then use its understanding to predict the best course of action.

This allows not only the modification of its decisions or behavior but also introduces the possibility of Meta-Control Processes, which are responsible for modifying the internal processes within the robotic system or even altering the system architecture. This emerging capability can enhance the adaptability and resilience of robotic systems.

Mission Planner Subsystem

In Tier II systems the mission plan has to be predefined by the user, and then the system will follow it. The plans could have control blocks, like loops or branches, but at the end of the day, are just a set of predefined task sequences in which the system does not really “know” why its doing these tasks.

In this new Tier of architectures, the system has enhanced knowledge about itself, the environment, and the *value* that each of the actions can provide. This *Awareness* about what is the value of the operation of the system, opens the possibility that the system itself can be able to generate the mission plan that maximizes the utility -the value- of the robotic system based on the values that it has. This is a disruptive change from the previous approaches in which the user has to provide **how** the operation will be, with the proposed approach, the user will only be in charge of defining **what** them want to me done by the sytem, and the system will chose how to do it based on the predefined values.

Example: How vs. What in System Operation

In the previous tiers, the mission plan had to be defined in terms of the paths that the UAV must follow to cover the PV plant, specifying waypoints or specific areas to explore. This means that the operator had to decide **how** the inspection should be performed. However, with this mission planner, the system only needs to know **what** plant to inspect and will autonomously determine the necessary paths around all relevant areas to capture images of the panels.

The *Mission Planner Subsystem* is responsible for integrating all values and combining them with the available knowledge in the system to generate the most suitable mission plan.

The user interacts with the system to provide parameters that can affect mission operation, but the concrete mission plan is generated automatically by this subsystem, making the system easier to use for an inexperienced operator.

3.5 Tier IV: Multi-Agent Systems

From this point, the proposed architecture allows the design of cognitive robotic systems with a high level of robustness and adaptability. However, the previous tiers only consider robotic systems composed of a single robotic agent.

In Tier IV, the objective is to provide an architecture capable of facilitating interaction between multiple agents to improve the performance of the robotic system—making it more efficient, more robust, or even enabling emergent functions that could never be achieved by a single-agent robotic system.

Since we are dealing with multi-robot systems, the starting point should be defining what an individual agent is.

Definition

*An **agent** is a system that is capable of autonomous action in an environment to achieve objectives.*

Tier III agents embody a high degree of autonomy since they should be capable of both proactive and reactive behavior, gathering knowledge from the environment they inhabit using their sensors and utilizing that knowledge to make decisions about how to act in order to reach their goals.

Additionally, our context not only implies a situation where multiple agents inhabit the same environment but also involves multiple agents working together toward a common goal. Thus, we need our agents to exhibit what we call social behavior. This entails communicating with each other, perceiving other agents and their actions, and using that knowledge in the decision-making process.

The Tier IV system architecture focuses on extending Tier III systems, considering that all agents work jointly to complete a common mission (goal). In this section, we will not distinguish between centralized and decentralized systems, but we will present a proposal for the architectural components required to effectively develop multi-agent robotic systems.

Definition

*A **multiagent system (MAS)** is a system composed of multiple interacting intelligent agents that interact to solve problems that are beyond the individual capabilities or knowledge of each individual.*

Multi-agent systems require the capability of communicating between them and the capability of using the exchange information to obtain an *emergent* knowledge of the system itself (*Collective Awareness*) or to coordinate the actions between them.

Definition

*A property **emerges** from a system if it can only be attributed to the whole system and not to its parts.*

Taking this into account, the capabilities of the Tier IV systems can be formulated as follows.

Tier IV capabilities

- C4.1 Include the capabilities of Tier I, II, and III systems.
- C4.2 Communicate and exchange information between agents.
- C4.3 Share and integrate knowledge among multiple agents to build a collective representation of the environment and system state.
- C4.4 Coordinate decision-making and actions across agents to achieve common goals efficiently.

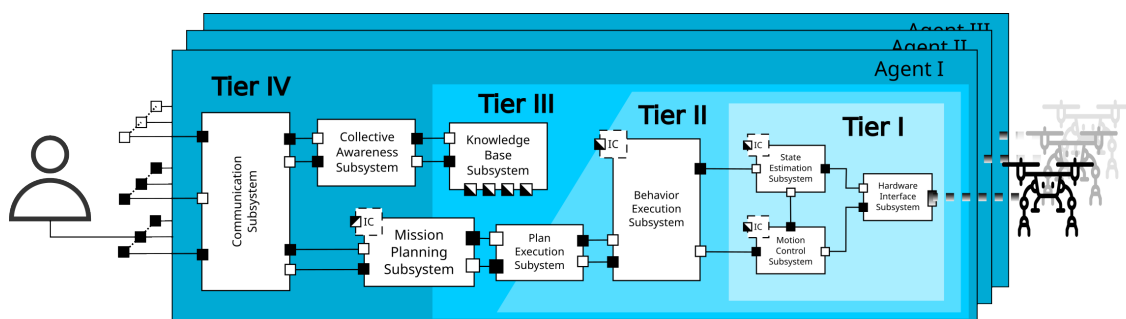


Figure 3.4: Tier IV Systems Architectural Proposal Schema. In the figure, multiple agents appear overlapped, with each agent having the same internal architecture as Agent I. Each agent controls a robotic platform, and all of them are interconnected through the *Communication Subsystems*, as shown on the left. The user interacts with the agents through these communication channels.

To satisfy these capabilities, this work proposes the addition of two new subsystems to the presented architecture: a *Communication Subsystem* and a *Collective Awareness Subsystem*, as illustrated in Figure 3.4. In addition to these new subsystems, the *Mission Planning Subsystem* shall incorporate mechanisms for communicating with other agents and integrating them into the planning processes.

3.5.1 Communication Subsystem

Another important capability that the agents of the system shall have is the ability to communicate with the rest of the agents.

When it comes to the theory of communication, (Shannon, 1948) define communication as the process of reproducing at one point either exactly or approximately a message selected at another point.

Definition

Communication is the act of transmitting information encoded by a sender into one or more receivers for which that information is *intelligible*.

We can categorize the communication between agents in two types:

- **Direct communication:** In this case, information is transmitted directly to different agents through wired or wireless communication technologies. In this communication scheme, information can be exchanged explicitly.
- **Indirect communication:** In this case, agents can communicate with each other by performing actions that other agents can interpret implicitly. A good example of this is stigmergy, a mechanism of indirect coordination between agents or actions through the environment. The principle is that the trace left in the environment by an individual action stimulates the execution of a subsequent action by the same or a different agent. This mechanism produces complex, intelligent structures without the need for any direct communication between agents. Stigmergy is a form of self-organization that enables efficient collaboration between extremely simple agents, who may lack memory or individual awareness of each other (Tang et al., 2019).

Although some systems can work by only exchanging raw data, when the system scales up in complexity, the use of data with high semantic information tends to reduce the bandwidth communications and improve the robustness of these systems. Some examples of these improvements can be seen in state of the art works like (Greve et al., 2023), in which they use semantic information of a city, such as lines or roads, for autonomous driving applications.

The *Communication Subsystem* is in charge of sending and receiving all the information across the different agents of the system. In this work, we have decided to base the communication of the different agents on the knowledge models obtained during operation. This allows reducing the bandwidth required for communication, compared with transmitting raw data.

3.5.2 Collective Awareness Subsystem

In Tier III, we worked on defining *Awareness* and *Understanding* for a single-agent robotic system. Now, we need to extend these definitions to encompass the scope of multi-robot systems.

Definition

*A multi-robot system **understands** a **phenomenon** ϕ if it has models (both individual and collective) that allow it to make sound inferences about ϕ*

The extension of this definition to multi-robot systems raises issues regarding both the meaning of a multi-robot system possessing a model of a phenomenon and the meaning of a multi-robot system producing an inference.

Collective understanding can be considered an emergent property resulting from communication between agents in a multi-robot system. This understanding may arise in different ways through interactions among individual agents. For example, agents may initially have a limited understanding of a phenomenon, which is then expanded through communication with other agents. Alternatively, communication may provide agents with sufficient information about a previously unknown phenomenon, enabling them to incorporate it into their decision-making processes.

Example: Multi-UAV Wind Estimation

Consider a multi-UAV system performing an inspection while estimating the wind. The wind can be modeled as a vector field, applying forces on each aircraft depending on its location. If the vector field is inferred while the system is operating, a more accurate estimation can be achieved when multiple agents work together, sharing their wind estimations until a unified **collective model** of the wind vector field is formed.

When it comes to the issue of owning a model in a multi-robot system, it is possible that these models could be allocated anywhere in the multi-robot system. These models may be allocated to a single agent, who is in charge of performing inferences and sharing them with the rest of the agents, or they may be distributed among the system.

A significant amount of research has been conducted on how these collective models emerge across various fields. Most approaches originate from economics and mathematics, where game theory has been used to formalize these models. However, other disciplines—including biology, computer science, politics, psychology, and philosophy—have developed new concepts beyond game theory to better model these interactions. Some of the most important mechanisms for generating collective models in these fields include:

- **Social choice**, which models collective preferences based on the individual preferences of agents within the collective. The clearest example of this is voting, primarily applied in politics.
- **Fair division**, which models the distribution of resources while ensuring that each participant receives a fair share based on the values they assign to the resource. In computer science and robotics, this is also known as multi-agent resource allocation.
- **Auctions**, which, like fair division models, address resource distribution among agents but assume that agents act in their own self-interest.
- **Negotiation and bargaining**, which represent agreements made between two or more agents by processing their often conflicting individual interests.
- **Coalition formation**, which seeks to partition a group of agents and assign actions to each member in a way that maximizes individual payoffs.
- **Consensus** models, which are used in distributed systems to determine shared values based on individual beliefs.

The function of the *Collective Awareness Submodule* is to generate models that are constructed across all agents, leading to a holistic understanding of the entire system. These collective models can utilize any of the previously mentioned mechanisms to be created, updated, and modified. Since the system as a whole pursues a common goal, its *Awareness* can emerge from these collective models.

Multi-agent mission plan

As in the previous Tier, the goal of the *Awareness* is to use the knowledge of the system to improve decision making processes.

In multi-robot systems, the *Mission Planner Subsystem* must consider not only a single agent's plan but also the mission plans of other agents when determining the overall plan that each agent will follow. There are multiple strategies for assigning tasks to agents and dividing the workload, such as centralized or decentralized approaches. However, in all cases, communication between agents is essential, and the system's overall value must be considered when generating, monitoring, and updating each robot's plan during operation.

These communication and task distribution functionalities have to be integrated into the former Submodule to fully achieve the Tier IV robotic system.

Example: Conflict resolution (I)

For illustrating how Collective Awareness provides planification and conflict resolution based on value we will use an scenerario where, during the inspection of a power plant, there is a drone that detects battery decay and returns to a landing area. In this new route, it will interfere with another drone trajectory, so a decision must be made in order to optimize the mission efficiency while maintaining the safety of the system; see Figure 3.5.

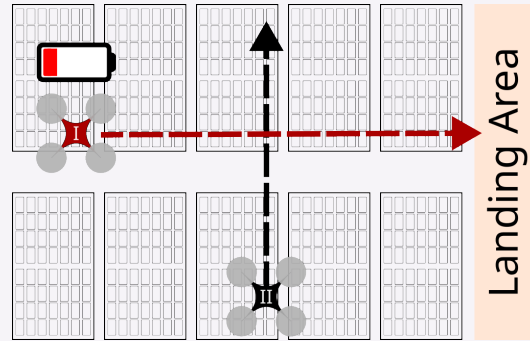
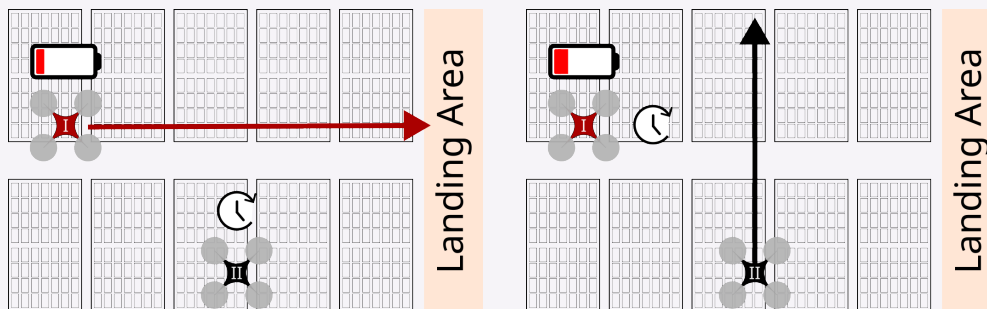


Figure 3.5: Initial situation on both of the emergency landing scenarios. In both scenarios there are a drone (Drone I, in **red**) that detects a battery decay so that it will not be able to finish its task and follow a trajectory towards the landing area. The issue is that this trajectory collides with another drone trajectory (Drone II, in **black**) so the system has to take the decision about how to solve this.

Depending on which drone will maintain its trajectory, which drone will return to the landing area or which is following the mission , we differentiate the two scenarios:



(a) Is considered that the battery is too low for waiting for landing, although it affects the mission performance.

(b) Is considered that the drone has battery enough for waiting without affecting the trajectory of the other drone.

The *Mission Planner Subsystem* is the one in charge of deciding in which of both scenarios the system is and take the best decision based on the status of the whole system.

Chapter 4

Aerostack2: the software framework for Aerial Robotic Systems

4.1 Introduction

In recent years, the robotics community has made significant effort in developing software frameworks for ground and articulated robots, leading to improved control, guidance, and collaboration. Well-established frameworks such as Navigation2 (Macenski et al., 2020) and MoveIt (Coleman et al., 2014) have gained widespread adoption, providing reliable tools for navigation and motion planning. However, the field of aerial robotics has not experienced the same level of standardization and shared development. Existing frameworks for aerial systems often have a narrow focus—such as low-level control or platform-specific implementations—which can limit their adaptability for broader applications. This lack of a unified approach makes it challenging to integrate the strengths of different frameworks, despite the growing need for robust, well-tested solutions that allow developers to prioritize innovation over software engineering complexities.

In this chapter Aerostack2 is presented, a software framework for aerial robotics that brings users and developers together to work toward a common goal: improving the capabilities of aerial robotics systems and enhancing their suitability for increasingly complex applications.

To ensure that the framework meets the needs of its users and effectively supports the development of advanced aerial robotics systems, a stakeholder analysis and a requirement analysis have been conducted in Section 4.2. After this analysis, the proposed framework will be introduced in Section 4.3. Subsequent sections, from Section 4.4 to Section 4.10, will provide a detailed explanation of each software component layer. Section 4.11 will present two practical examples illustrating the application of the proposed framework in different scenarios, reinforcing the previously introduced concepts. Finally, Section 9.4 presents a collection of scientific works that use Aerostack2, demonstrating its suitability for a variety of cases.

4.2 Stakeholder Needs and Requisites Analysis

Although the motivation for creating this framework was to enable the rapid development of both research work and industrial applications within our research lab, we aimed to go further and develop a framework that can be used by the broader robotics community.

The goal of this framework is to serve as a useful open-source tool for different user profiles, who will be considered the stakeholders of the framework. We have categorized them based on two criteria:

- **Profile:** This criterion categorizes individuals or groups involved in various aspects of Unmanned Aircraft Vehicle (UAV) development and operation, focusing on their roles and responsibilities in achieving mission objectives and ensuring system functionality.
 - **Mission Specifiers (MS):** Stakeholders who define the objectives and constraints of UAV missions but may lack technical expertise. Their focus is on outcomes and mission goals.
 - **Use Case Developers (UCD):** Professionals who extend the framework to cover the requirements of specific use cases, ensuring that these systems meet mission needs.
 - **Algorithm Engineers (AE):** Specialists who develop algorithms to introduce new capabilities to UAS or enhance existing ones. Their work spans various fields such as navigation, control, multi-robot coordination, and decision-making.
 - **Hardware Integrators (HI):** Engineers or technicians responsible for assembling and configuring UAV components to ensure proper system functionality.
- **Application:** This criterion highlights the practical uses of UAV technology across different sectors, emphasizing innovation, efficiency, and accessibility for various user groups.
 - **Researchers (RE):** Academic or industrial researchers who explore new UAV technologies, algorithms, or applications. Their focus is on innovation and knowledge expansion.
 - **Industries (IN):** Organizations that utilize UAS for commercial purposes, such as industrial inspection, surveillance, or agriculture, prioritizing efficiency, scalability, and cost-effectiveness.
 - **Hobbyists (HO):** Enthusiasts who use UAVs for recreational purposes or small-scale personal projects, prioritizing ease of use and customization.

4.2.1 Functional Requirements Analysis (FR)

After identifying the *Stakeholders* who may be interested in using this framework, a requirements analysis was conducted. Table 4.1 summarizes the most important requirements obtained during this process.

ID	Name	Description	Rationale
FR-01	Sensor-Actuator Integration	The system shall integrate with various sensors and actuators through a standardized interface.	Facilitates platform independence by allowing seamless integration with different hardware components, ensuring consistent communication between the software framework and the hardware.
FR-02	Platform Independence	The system shall be compatible with various aerial platforms, including DJI platforms, Pixhawk autopilots, and Crazyflie drones.	Ensures that the framework can operate across different hardware setups, providing flexibility and ease of integration.
FR-03	Simulation Support	The system shall support both simulation and real-world operations in a similar manner.	Simplifies the development process by allowing for testing and validation in simulated environments before deployment in real-world scenarios, reducing risks and development time.
FR-04	Ready to fly	The system shall provide essential functionalities such as motion control and state estimation.	Offers foundational capabilities necessary for autonomous flight, ensuring stability and accurate navigation.
FR-05	Indoor and Outdoor Operation	The system shall support flight operations in both indoor and outdoor environments.	Enhances the versatility of the aerial robotic system, allowing it to perform missions in various settings and conditions.
FR-06	Modularity and Flexibility	The system shall allow elements to be changed or interchanged without affecting the rest of the system.	Provides a plugin-based architecture that enables the use of different implementations for every task, enhancing adaptability and long-term maintainability.

FR-07	Task Execution	The system shall perform both motion and cognitive tasks.	Enables the system to handle a variety of tasks, from physical movements to decision-making processes, enhancing its versatility.
FR-08	Task Supervision	The system shall supervise the correct execution of the different task, ensuring a safe and reliable operation	Incorporates checks to detect malfunctions or issues that can be used for ensure the well-being of both the aerial robots and their operating environment.
FR-09	Mission Definition	The system shall perform a mission specified by a user based on a sequence of tasks to be performed.	Ensures that the aerial robotic system can execute user-defined missions, providing flexibility and adaptability to various tasks.
FR-10	Mission Control and Supervision	The system shall provide user interfaces and tools for mission planning, system monitoring, and debugging.	Facilitates user-friendly interaction with the system, catering to both developers and end-users, and ensures effective oversight and control during missions.
FR-11	Multi-Robot Coordination	The system shall support the operation of multiple drones simultaneously, both in indoor and outdoor environments.	Allows for the execution of missions involving multiple aerial robots, enhancing operational efficiency and enabling collaborative tasks.
FR-12	Project-Oriented Development	The system shall allow the installation and use of only the necessary packages and configurations for the application to be developed.	Streamlines the development process by focusing on the specific needs of a project, reducing complexity and resource consumption.

Table 4.1: Functional Requirements obtained from the relevant Stakeholders

4.3 Framework Overview

Based on the previously obtained requirements, we propose a ROS 2-based layered architecture, in which the different components are distributed according to the reference architecture proposed in Chapter 3.

The decision to use ROS 2 as the foundational middleware facilitates the interoperability of the developed models and allows compliance with requirement *FR-06* by distributing the different components across multiple nodes. Moreover, using a well-known robotics middleware lowers the entry barrier for other roboticists to adopt this framework.

Based on the idea of having layers that utilize functions from the layers below them, the proposed architecture follows a five-layered hierarchical structure. Each layer of the system may consist of multiple components that interact with the rest of the system at a similar level of abstraction,

as shown in Fig. 4.1. Components in one layer control those in the layers below, corresponding to more fundamental functionalities.

The Aerostack2 framework has been evolving over the past few years, incorporating additional layers. The initial objective of this framework was to enable the development of Tier II+ systems. These Tier II+ systems can be categorized as Tier II but include enhancements in their ability to generate simple mission plans and distribute them across a fleet of drones using basic centralized mechanisms. This has been the most widely used version of the framework, and its modules have been extensively tested and deployed in various scenarios, including up to TRL-8 systems.

In recent years, we have focused on improving the *Understanding* and *Awareness* of systems developed with this framework, motivating the development of modules and components for Tier IV systems. However, the validation of these new components remains limited to simple use cases.

4.3.1 Initial approach: Tier II+ framework

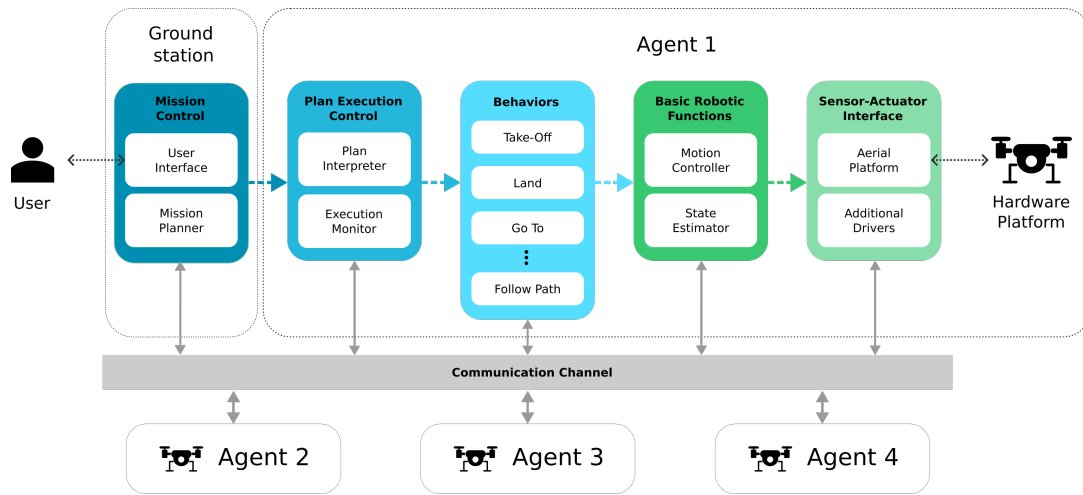


Figure 4.1: Overview of the Tier II+ Aerostack2 framework architecture. Each colored block represents a hierarchical layer, which can be constituted from multiple ROS 2 nodes or processes, represented as white blocks. All the nodes and agents are communicated through the ROS 2 communication channel, as is represented with grey arrows. The big colored arrows going left to right, represent the *command* flows, so that each blocks send commands to the right one. Nevertheless, each module can feed back meaningful information to the left modules through the communication channel, so the communication is bidirectional.

Figure 4.1 illustrates the layered architecture of Aerostack2. This framework encompasses modules enabling user interaction by providing mission specifications. The layers progressively reduce the level of abstraction, translating mission specifications into low-level commands necessary for the aerial platform. The layers constituting the Aerostack2 framework, in descending order of abstraction, are:

- **Sensor-Actuator Interface:** This layer interfaces with the hardware aerial platform and onboard sensors, ensuring platform agnosticism for seamless integration of heterogeneous swarms and facilitating the transition from simulation to real-world applications.
- **Basic Robotic Functions:** This layer consists of essential software components like the motion controller and state estimator, performing fundamental robotic functions crucial for aerial robotics.
- **Behaviors:** Behaviors provide a logical layer for formulating mission plans in a uniform

and simplified manner. Each behavior corresponds to a specific robot skill, encapsulating both control and execution monitoring of the assigned task.

- **Plan Execution Control:** This layer orchestrates and monitors the activation and deactivation of various behaviors, ensuring precise execution of mission plans.
- **Mission Control:** Mission Control provides modules necessary for generating plans based on mission specifications, ensuring that mission objectives are translated into executable tasks.

The layered architecture of Aerostack2 allows users to interact with specific layers of the framework according to their unique requirements, enabling customization and optimization for diverse applications. Figure 4.2 illustrates different types of users interacting with various layers of the framework based on their specific needs. Given its versatility and modularity, Aerostack2 serves as an excellent foundation for the design of our proposed architecture.

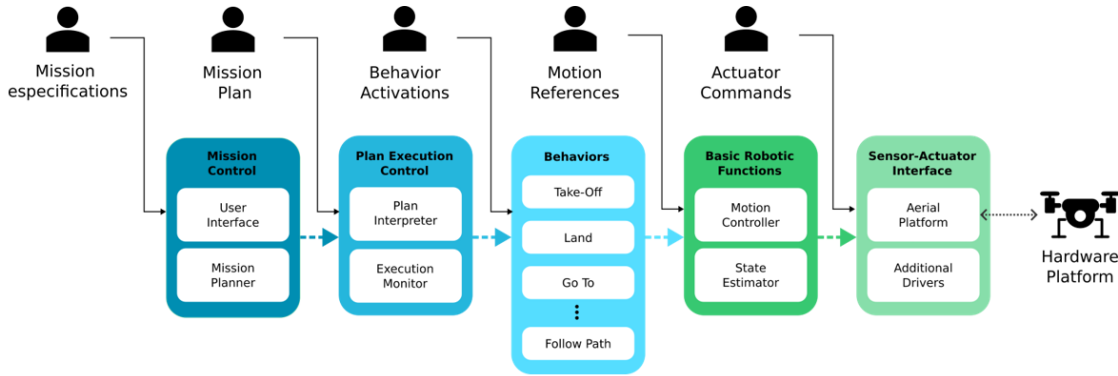


Figure 4.2: Different users of the Aerostack2 framework, ordered from the user or high-level view on the left to the hardware or low-level view on the right. Each level is accessible to the developer without the necessity of concern for the higher abstraction levels.

4.3.2 Extended version: Toward Understanding and Awareness

In complex robotic systems, particularly those involving multi-agent coordination, the ability to process, interpret, and react to dynamic environments is essential for effective operation. To achieve this, two fundamental mechanisms—*Understanding* and *Awareness*—must be integrated into the system architecture. *Understanding* refers to the system’s ability to extract meaningful insights from data, facilitating informed decision-making. *Awareness*, on the other hand, enables the system to maintain a global perspective, ensuring that relevant information is accessible across all agents. The synergy between these mechanisms enhances adaptability, robustness, and efficiency in autonomous operations.

Recognizing these potential areas for improvement, we have focused on two domains within the architecture that can be significantly enhanced through the *Understanding* and *Awareness* mechanisms: planning capabilities and collaborative operation between agents. Enhancing awareness during the planning phase is crucial for optimizing the system’s decision-making processes.

By integrating *values* into the architecture, the planning stage gains access to more comprehensive and accurate information. With this improved understanding, the system can make more informed decisions, ensuring that mission planning is optimized for various scenarios and challenges.

On the other hand, the multi-robot aerial system consists of multiple agents that must collaborate effectively to ensure mission success. Collective awareness capabilities can improve the performance of robotic systems. By creating a coherent knowledge base across the swarm, tasks such as dynamic swarm coordination, intelligent task allocation, and adaptive replanning can be facilitated. Drones within the swarm can share real-time information about their mission objectives, environmental perceptions, and individual statuses. This shared awareness allows the swarm to dynamically adjust its behavior, ensuring efficient task allocation, optimized path planning, and adaptive responses to unforeseen obstacles.

That is why an extension of the Aerostack2 software framework has been proposed, taking into account two new components: the Knowledge Base and a Communication component, paving the way for more complex multi-agent interaction (Figure 4.3):

- **Knowledge Base:** This component responsible for collecting all the models generated by the different subsystems and integrating them in an accessible manner for the rest of the modules. In Aerostack2, in order to help with the modularity and extensibility of the framework we divide this block in the *Knowledge Server*, which is the “data base” responsible of the storage of these models, and the *Knowledge Extractors* which listens to the data flowing over the communication channels for retrieving knowledge without needing to modify or extend the modules previously generated.
- **Communication:** Since there is a division in the inter- and intra- communication channels there is needed a component in charge of deciding what and how the internal information for each agent has to be shared between the rest of agents. This process has also to be performed in the opposite way, by filtering which external information is relevant for each agent and how shall be treated. The user commands shall also be treated through this module.

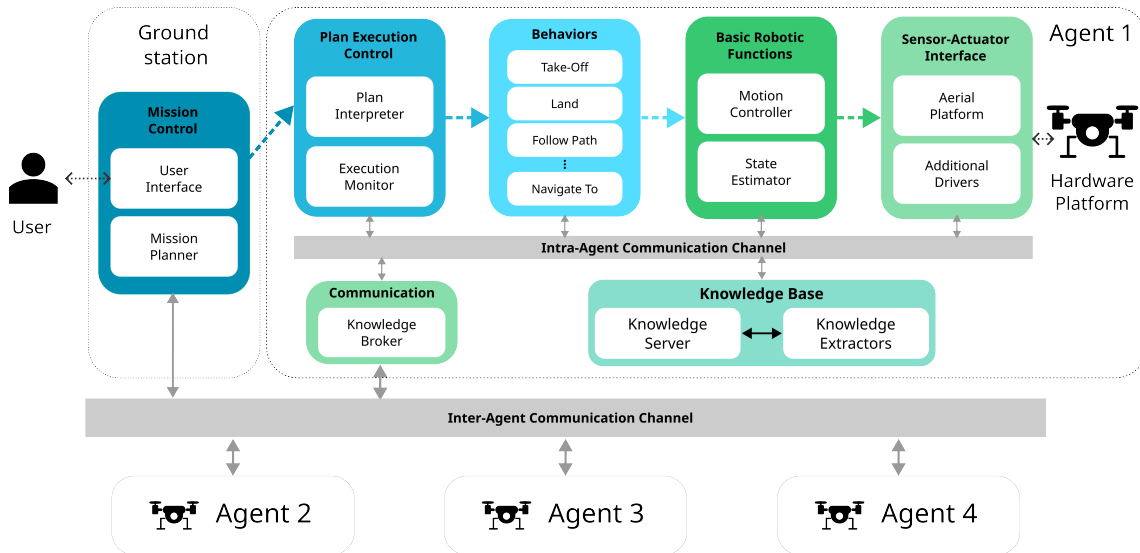


Figure 4.3: Overview of the extended framework architecture. This proposal illustrates two distinct communication channels: one for intra-robot messages and another for inter-agent communications. The ground station communicates with other agents via the inter-agent communication channel. Dashed arrows represent command flows.

In contrast to the previous layered components, these two new components handle information generated across multiple layers, allowing them to have a holistic view of the agent’s knowledge and the information provided by other agents. Another key difference compared to the former version can be found in the division between the inter- and intra-agent communication channels.

In the former version, all components (*ROS 2 Nodes*) of all agents use the same channel for communication. This causes the communication channel to become flooded with messages that are only required for the internal functioning of each agent as the number of agents scales up.

Since this extension is built on top of the existing architecture, the following sections will describe in more detail each layer of the extended software stack. All the proposed layers share the same properties in both approaches.

4.4 Sensor-Actuator Interface

A primary objective of the presented framework is to facilitate the independent implementation of autonomous aerial systems, regardless of the final deployment platform, provided that sensory input and actuators are compatible. This requirement has been identified as *FR-01* and *FR-02*.

To achieve this, it is essential to adopt a higher level of abstraction and establish a general representation of the expected methods for interfacing with the sensors and actuators of different aircraft. In Aerostack2, the primary interface with the hardware platform is handled through the FCU. Each FCU has a distinct communication protocol for executing various commands and retrieving information about sensor measurements and other relevant internal variables. The goal of our framework is to standardize these diverse interfaces into a unified one, allowing the rest of the framework to be developed independently of the specific FCU used in each case.

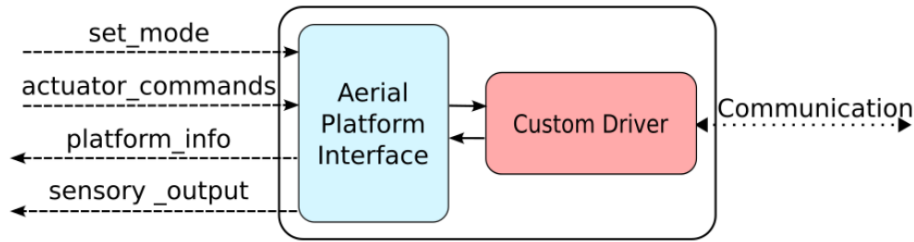


Figure 4.4: Diagram of the Aerial Platform component. Left arrows represent the flow of information towards the rest of the framework. The right arrow represents the communication with the hardware components using the proper protocol.

Additionally, this approach extends to simulation environments by bridging the simulated platform with the broader framework, satisfying requirement *FR-03*. This ensures that algorithms developed in the higher layers remain agnostic to whether they operate within a simulation or on a real-world system.

The primary responsibility of this interface is to gather sensory measurements from the aircraft and transmit them to the rest of the system. Additionally, it is responsible for receiving actuator commands and other requests from various layers of the Aerostack2 framework and relaying them to the aircraft in a platform-specific manner. More specifically, the module must be capable of performing the following tasks:

- **Arm and Disarm:** Enable the activation (arming) and deactivation (disarming) of the aerial platform's motors.
- **Offboard Control:** Provide the capability to send custom commands for direct control of the aerial platform in offboard mode.
- **Set Control Mode:** Allow configuration and modification of the flight control mode based on operational requirements.

- **Send Commands:** Facilitate communication to transmit specific actuation commands to the aerial platform.
- **Handle Internal State Machine:** Manage the platform's internal state machine, ensuring correct transitions during operation, see Figure 4.5.
- **Handle Alert Events:** Detect, process, and respond to alert events generated by the aerial platform.
- **Kill Switch:** Implement an emergency functionality to immediately halt operations in critical situations.
- **Read Sensors:** Acquire data from various sensors on the aerial platform, such as battery status, IMU, magnetometer, GPS, and others.

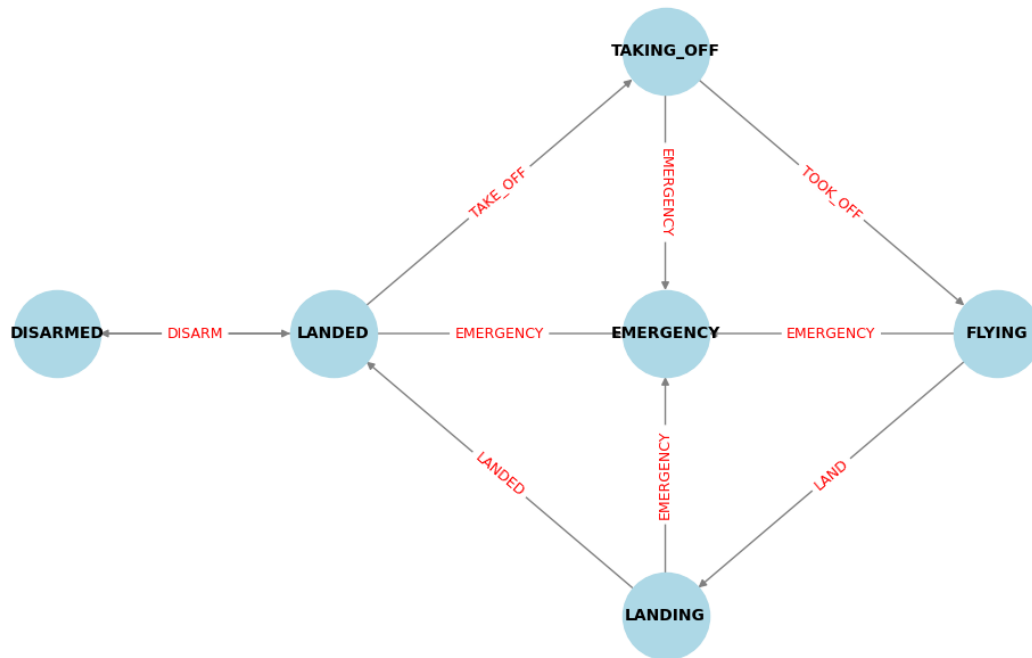


Figure 4.5: This FSM diagram represents the platform's state transitions. The DISARMED state is positioned on the left, while LANDED, TAKING_OFF, FLYING, and LANDING form a square structure around the EMERGENCY state in the center. Transitions between states are triggered by specific events such as ARM, DISARM, TAKE_OFF, TOOK_OFF, LAND, LANDED, and EMERGENCY. The EMERGENCY state serves as a critical fallback, accessible from all operational states.

These platforms facilitate the transition from simulation to real-world deployment by ensuring that logic modules remain agnostic to whether the system is operating on a real platform or in simulation. Additionally, this approach simplifies the implementation of heterogeneous aerial systems across different platforms.

At the time of writing, Aerostack2 supports nine different aerial platforms, ranging from simulated to real systems. In some cases, multiple protocols are supported to better align with the user's preferred workflow. Annex B includes a detailed list of the platforms used in this thesis, highlighting the variety of supported platforms.

4.5 Basic Robotic Functions

In aerial robotics, certain functions are vital for the basic operation of the system, as reflected in requirement *FR-04*. In Aerostack2, we refer to these as Basic Robotic Functions, with each function having a dedicated component responsible for its implementation. Some of these essential functions include motion control and state estimation, both of which support the autonomous operation of every aerial robot.

These functions, combined with the *Sensor-Actuator Interface*, form a Tier I system, as presented in Chapter 3. Given the importance of these components, we have identified several general capabilities that they should implement:

1. The implementation of each basic function must be done in a plugin¹ that inherits from a virtual class, ensuring a common interface and compatibility between plugins at compile time.
2. The plugins must be switchable at run-time.
3. The inputs and outputs should be standardized to maintain a consistent plugin interface.

In this approach, each component is managed by a function manager, which is responsible for loading the plugins with each specific algorithm and managing how they interact with the rest of the framework. The plugin selector can also provide meta-control features, such as plugin replacement, whereas the input and output adapters adjust the input or output of the function plugin to the rest of the Aerostack2 framework. The use of plugins for encapsulating the concrete algorithm implementations satisfy requirement *FR-06*.

In the following, a more detailed explanation of the implementation of the Motion Controller and the State Estimator is provided.

4.5.1 Motion Controller

This component handles the motion references generated from the upper-level layers and convert them into actuator command signals which will be followed by the specific aerial platform. This component performs three tasks:

1. **Plugin Load and Selection:** The module loads the different plugins that are required to operate. In this case, this plugin includes the implementation of different control laws to control the platform. The selection of the plugin can be done by the user or decided in runtime using predefined rules. When the motion reference can be handled directly by the aerial platform, the plugin can be bypassed.
2. **Control Mode Negotiation:** Each plugin implements multiple combinations of input-output control mode. A control mode represents the set of different input signals that a controller plugin can handle. Similarly, each Aerial Platform has a set of control modes that are available to control it. The motion controller component is responsible for finding the combination of modes that satisfies the requirements of the motion reference, the loaded plugin, and the aerial platform.
3. **Input-Output Adaptation:** Based on the motion reference received (e.g. position, velocity, angle, etc.) in a specific reference frame, the module transforms this input into the desired

¹A plugin following the pluginlib C++ approach is a dynamically loadable C++ class that implements a predefined abstract interface and is compiled into a shared library. It registers itself using specific macros, which allows the host application to discover and instantiate it at runtime without needing to link against it at compile time.

coordinated frame of the loaded plugin. Similarly, some reference changes can be needed before sending the actuator commands to the aerial platform.

Figure 4.6 shows how the different functionalities are embedded and connected within the Motion Controller.

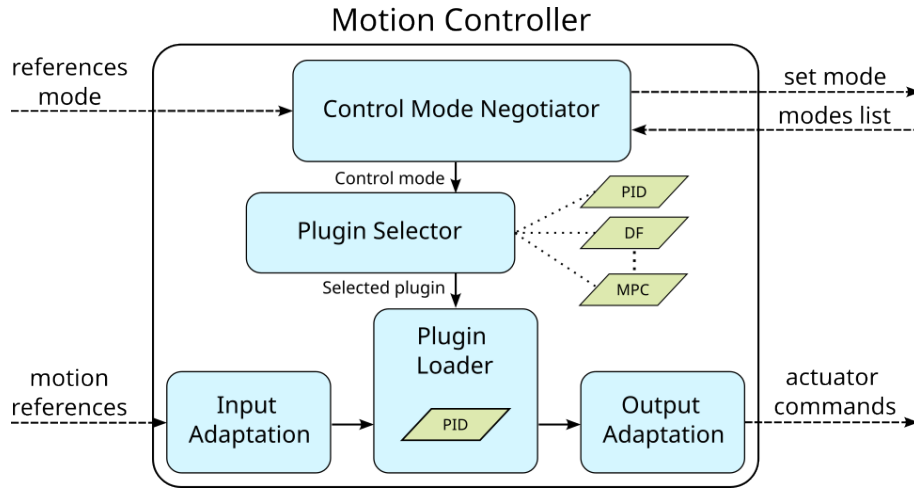


Figure 4.6: Diagram of the motion controller component. Blue boxes represent the capabilities of it and how interact between them and the other modules. Plugins are represented with green trapezoids.

Metacontrol Capabilities: Dynamic Control Mode Negotiation

The control mode refers to the type of references/outputs that a controller manages. In this framework, we differentiate between two controllers that run sequentially: the internal controller running inside the FCU and the outer controller implemented within this motion controller, which runs on the control computer.

Although this setup may introduce computational overhead and add delays to the control loop, it provides important features that make our framework highly configurable depending on the specific use case. This dual-controller setup allows for:

1. Mitigating the influence of platform dynamics and configuration: the FCU is in charge of providing the appropriate signals to each motor, taking into account the configuration—whether it has four, six, or coaxial motors. Sometimes, it also includes a dynamic model of the platform, allowing the outer controller to provide reference signals above the motor control level.
2. Distributing the computational load: By dividing control responsibilities between the FCU and the outer controller, the system can offload low-level motor control tasks to the FCU, reducing the computational load on the main control computer. This distribution improves overall efficiency, allowing real-time response even in computationally demanding scenarios.
3. Handling multiple reference systems: The framework supports different control reference frames, such as body-fixed, inertial, or external positioning systems. This flexibility allows for the use of different navigation strategies, making it possible to adapt to various operational environments and control objectives easily.

Handling the control mode correctly can help reduce the computational load of the framework by delegating low-level control to the autopilot at the expense of decreasing the precision of the aircraft's movement. On the other hand, when highly precise maneuvers are required, the user may opt to use a low-level controller to achieve full control over the UAV's movements.

Based on our experience working with different types of autopilots we have opted for considering the following control mode categories:

- **Motion Control Modes:** Defines the method and objective for controlling the overall movement and positioning of the drone.
 - **Hover:** Maintain a fixed position and altitude, compensating for external disturbances.
 - **Acro:** Direct manual control over angular rates (roll, pitch, yaw), enabling agile maneuvering.
 - **Attitude:** Maintain or achieve specific orientation angles (roll, pitch, yaw).
 - **Speed:** Maintain or achieve a specific linear speed in 3D space.
 - **Speed in a plane:** Maintain horizontal linear speed; altitude control handled separately.
 - **Position:** Move toward or maintain a predefined 3D position.
 - **Trajectory:** Follow a position, velocity and acceleration references.
- **Yaw Modes:** Specifies the strategy for controlling yaw, which is rotation around the vertical axis of the drone.
 - **Angle:** Directly specify and control yaw angle.
 - **Speed:** Control yaw by specifying a desired angular velocity.
- **Frames of Reference:** Defines the coordinate system in which motion commands are expressed and interpreted.
 - **Local Frame FLU (Forward-Left-Up):** Commands relative to drone's body frame.
 - **Global Frame ENU (East-North-Up):** Commands in a fixed global Cartesian frame.
 - **Global Frame LLA (Latitude-Longitude-Altitude):** Commands expressed in geographical coordinates.

Each aerial platform supports a subset of combinations of these modes as an input. Furthermore, the controller plugin implementation can receive inputs for some of these control modes and can provide outputs in one or multiple modes. Ensuring compatibility between the motion reference received by the controller and the output it generates—so that it remains compatible with the platform's supported modes—is crucial for controlling the platform.

Other frameworks fix which control modes are available and reduce this complexity. However, this variety of modes streamlines the development of different tasks, facilitating the development of motion maneuvers.

The motion controller handles all mode negotiation transparently for the user by dynamically adapting to the different motion references received and managing the best configuration of the plugin controller and actuation commands.

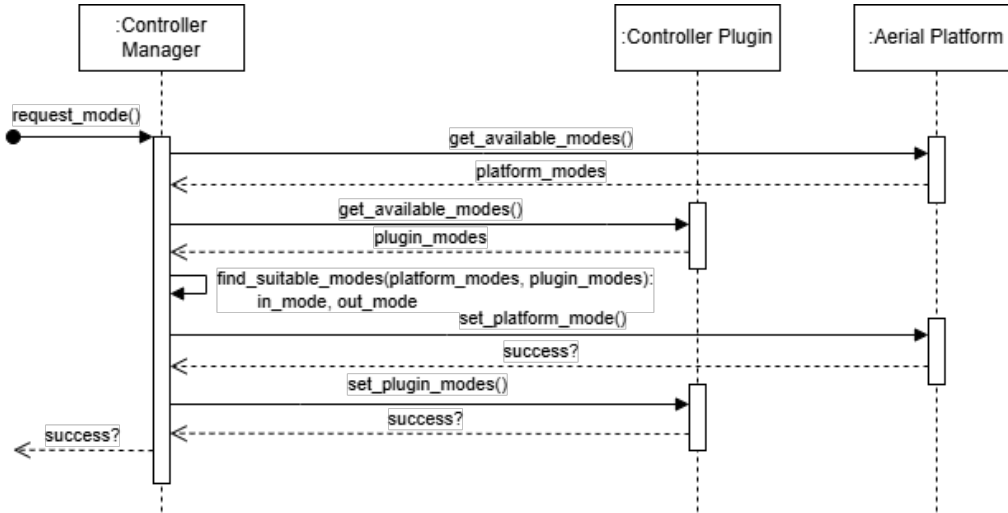


Figure 4.7: Sequence diagram describing the interaction between the Motion Controller component, the controller plugin, and the aerial platform to enable following a motion reference in a specific control mode.

Figure 4.7 presents a sequence diagram of how this process is performed. This metacontrol capability also helps with the interaction between different aerial platforms, which may only allow a specific control mode, but the controller dynamically self-reconfigures to fulfill its interface requirements.

4.5.2 State Estimator

This module fuses the information received from different sensors to estimate the state of the robot, where the state refers to the pose and speeds of each aircraft over time. Similar to the previous one, this module can load multiple state estimation algorithms in the form of plugins. The State Estimator module is responsible for:

1. **Plugin Load and Selection:** The module shall load the different plugins that are required to operate. The selection of the plugin can be done by the user or decided in runtime using predefined rules.
2. **TF tree generation:** The module is in charge of generating the transformation trees (Foote, 2013) that will be used for the rest of the framework, allowing the system to represent information in different coordinate frames. This module is also in charge of managing the origin of the coordinated system in a multi-robot system.
3. **Geodesic to Euclidean Conversion:** When operating outdoors, this component is in charge of storing the geodesic operation origin, allowing the system to be able to work in the Euclidean space and translate geodetic coordinates, helping satisfying requirement *FR-05* when GPS is available.
4. **Algorithm Adaptation:** The module ensures that the most suitable state estimation algorithm is selected and used based on available sensor data and operational conditions. It can dynamically switches between different algorithms to maximize accuracy and reliability, applying predefined rules.

Figure 4.8 shows how these different functionalities are embedded and connected within the State Estimator.

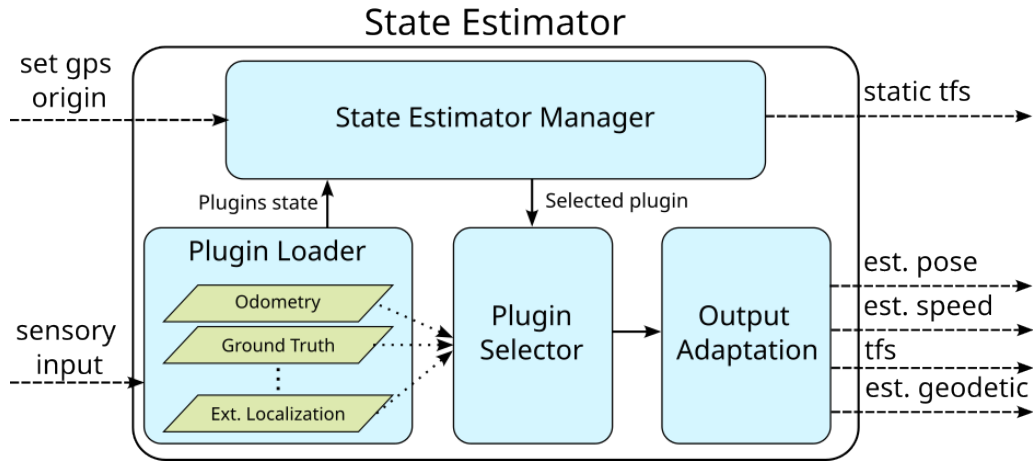


Figure 4.8: Diagram of the State Estimator component. Blue boxes represent the functions of it and how interact between them and the other modules. Plugins are represented with green trapezoids.

Dynamic State Estimator Reconfiguration

Other metacontrol capabilities implemented within the Aerostack2 framework relate to handling state estimation capabilities. This is a fundamental robotic function, whose correct operation is crucial to successfully performing the task and ensuring safe operation. That is why it is a perfect place for enabling metacontrol capabilities to enhance robustness and increase the system's flexibility.

As the state estimator module is built with similar architectural decisions as the motion control module, the metacontrol layer shares some commonalities, such as the main objective: deciding which plugin will be in charge of operating in each situation. However, it has some key differences that prevent the reuse of the same strategy presented above:

- Multiple plugins can operate simultaneously. Multiple algorithms can compute their estimations in parallel over time. However, only one state estimation shall be the output of this module, so a decision must be made about which is the best option at each moment.
- Continuous operation. The state shall be updated continuously as new sensor measurements are received, independently of whether a specific module is requesting it.
- Dependent on sensory input. Each state estimator algorithm may require different sensory inputs—some may use visual information, others satellite data, and others lidar. Depending on what is available, a specific algorithm may or may not be used.
- Application-focused rules. In general, the estimator used will be the one with the lowest covariance, as it should reflect the most precise algorithm. However, in some cases, such as research, specific rules—like having a well-tested backup system—can be applied when testing an in-development algorithm.

In order to apply the metacontrol rules of this module, a multiplexer-like structure was implemented. All the different parts of the estimation pass through the manager, which selects which plugin will be used for each part. Each plugin provides four estimations that, when combined, form the robot's state estimation:

1. **Odometry pose:** The robot's position and orientation estimated relative to a local odometric reference frame. This evolves continuously over time.

2. **Robot velocities:** The robot's linear and angular velocities represented as twist information. These also evolve continuously over time.
3. **Error between the odometry and mapping reference system:** The drift or offset accumulated between local odometry and a global mapping frame, typically corrected through sensor fusion algorithms. This can change abruptly.
4. **Pose of the mapping system in the global frame:** The robot's mapping reference position and orientation within a global frame, obtained through global localization methods such as GPS or motion capture.

Finally, a set of rules is checked and applied sequentially until the final configuration is obtained, and the four estimations are combined and output. These rules are ordered from the most restrictive to the most relaxed. Whenever a restrictive rule is triggered, the selected module is “multiplexed”.

There are two rules that are always applied, after which the developer can append additional rules depending on the application. The most restrictive rule is always the *Existence Rule*: at least one available estimation must be used. If there is only one candidate, it will always be selected.

Then, the *Preference Rule* applies: if the user has specified a preference for any estimation and it is available, it shall be selected.

Finally, an example of an additional rule that can be used is the backup system, which activates when an algorithm is not performing as expected. This is particularly useful when testing algorithms in real-world scenarios with aerial systems.

4.6 Behaviors

At this point, we have developed a Tier I basic framework that enables interfacing with various aerial platforms and provides fundamental capabilities, such as concrete movement and state awareness. However, generating complex plans by directly controlling the drone through control references based on its current state can be tedious, highly repetitive, and challenging for humans to interpret.

According to the terminology of Tier II systems presented in the previous chapter, Aerostack2 employs a specialized component called *behavior*, which introduces a logical layer for formulating mission plans in a uniform and simplified manner compared to directly using state estimators and actuator controllers. These behaviors serve as foundational blocks for specifying missions by chaining them appropriately. Each behavior corresponds to a specific robot skill, such as flight motion (e.g., taking off, landing, hovering, following a path) or other capabilities (e.g., video recording, image detection, generating a trajectory to reach a destination), satisfying requirement *FR-07*.

Each Behavior has its own component and how each behavior is implemented can be different, providing them a certain level of independency with the rest of behaviors.

However, as stated in Section 3.3.1, all behaviors shall have some commonalities between them to ensure the compatibility of them with the rest of the architecture.

In order to ease the creation of these *Behaviors*, Aerostack2 provides a behavior template architecture for programming behaviors that includes two distinct parts (Figure 4.9).

On the one hand, there is a *behavior executor* that performs the execution using the algorithm used for each case. Although is not mandatory, we encourage to use different plugins whenever the same *task* can be performed in different ways and each way corresponds to different algorithms and implementation.

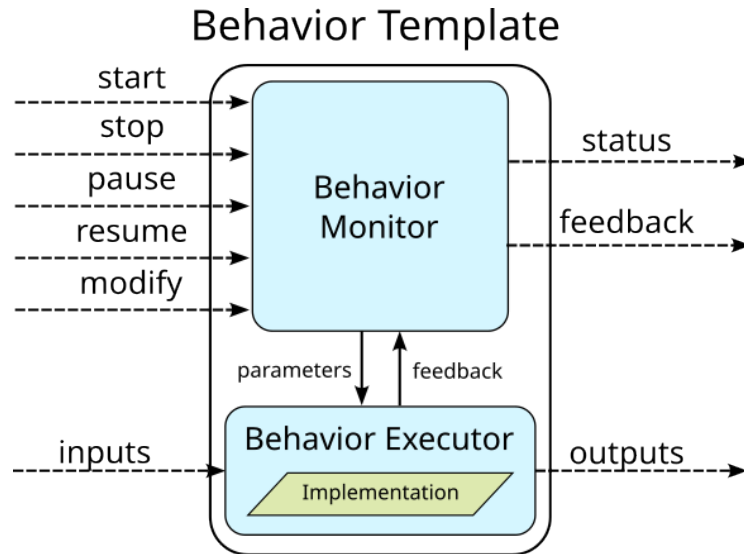


Figure 4.9: Schematic representation of the behavior template architecture. The *behavior monitor* processes external commands and monitors behavior execution by interfacing with the *behavior executor*, which handles the underlying implementation. The system outputs include status and feedback, derived from the processed inputs.

Example: One Behavior and Multiple Implementations

In order to illustrate this, we will consider the following example. A landing task can be done with a simple maneuver by progressively reducing the altitude of the aircraft in a controlled way until it reaches the ground.

However, it can also be performed by using a down-looking camera that corrects the landing position in order to precisely land in a visually marked area.

In both cases, the aircraft lands, but depending on the application, how it is done can be changed.

On the other hand, there is a *behavior monitor* that interacts with the behavior executor to check its operation before, during, and after execution. The monitor can adapt the behavior execution by, for example, considering alternative modes of operation depending on the state of the environment or response measures in the presence of faults. This feature adds adaptability and robustness to the architecture, satisfying requirement *FR-08*. The modular distribution of execution monitoring provided by Aerostack2 with behaviors also provides greater flexibility than architectures with centralized monitoring.

Implementation Details

In order to keep the resemblance of the behaviors with ROS actions, *behaviors* consist of three parts: the *behavior* interface specification, the *behavior* server implementation, and the *behavior* client. The *behavior* interface specification defines the behavior's calling parameters, the feedback provided during its execution, and the expected result. The *behavior* server is the component where the logic is implemented, determining how the task will be executed and monitoring its correct execution. Finally, the *behavior* client is responsible for controlling the execution of the behavior.

In more detail, the uniform interface used to control the execution of the behavior uses the following basic services: *start*, *pause*, *resume* and *stop*. In addition, a service called

`modify` is used to change the parameters of a behavior without the need to stop its execution. The behavior interface informs about its execution state (e.g., idle, running, or paused) and periodic feedback about the concrete details on the running behavior. Aerostack2 behaviors have been implemented in a way that is fully consistent with standard ROS 2 actions.

Each software component that implements a behavior encapsulates the details of the algorithms used, providing a uniform interface that is common to all behaviors. This feature provides simplicity in describing mission plans. Using behaviors, a mission plan is expressed as a controlled sequence of activations (or deactivations) of multiple behaviors that may operate concurrently. The result of each behavior execution is described in terms of success or failure, which is useful to determine the next step to be done during the mission.

4.7 Plan Execution Control

All the previous components of the architecture are built to encapsulate all the functionalities needed to enable the robotic system to perform small tasks autonomously and reliably. However, the designed components were intended to be quite generic and reusable, broadening the number of capabilities that the aerial system has.

The following components take advantage of these functionalities and focus on how the different tasks that drones can perform can be combined to generate increasingly complex behaviors, leading to the completion of a relevant task.

The mission developer can write a plan to specify the set of tasks that a robot must perform in a particular mission. The specification of plans can be formulated in terms of activations and deactivations of different robot behaviors, as expected from *FR-09*. In this way, the user can easily specify complex missions by combining tasks that the robot executes autonomously. The component responsible for executing this plan and handling different situations is the Plan Interpreter.

Aerostack2 provides three main ways to implement this plan interpreter:

1. **Specific plan through Python API.** In this case, plan execution is carried out by directly interpreting a Python script, taking advantage of the Aerostack2 Python API, which helps interact with the Aerostack2 framework by implementing a `DroneInterface` class that contains several methods not only to command different drones but also to retrieve meaningful information about their status. This is a convenient method that provides high flexibility for formulating plans with complex control regimes. Regarding plan control, our API provides a set of functions to activate/deactivate behaviors or to command directly through motion reference messages.
2. **Mission Interpreter.** A mission plan can be described as a sequence of behavior activations and deactivations. These sequences can be specified in JSON format, which can be directly translated into calls to the Python API by the *Mission Interpreter*. This allows each drone to receive the mission plan in a text format and then interpret it. This module also supports conditional sentences that guide mission execution. This provides a comprehensive way of transmitting, loading, or modifying mission plans, which can be very useful in loosely connected communications within a control station, allowing the plan to be generated in a ground station and then loaded into the onboard computer, which will perform the mission even if the connection fails.
3. **Behavior Trees.** One of the most widely used tools for describing plans in robotics is Behavior Trees (Colledanchise and Ögren, 2018), whose philosophy aligns with Aerostack2's approach to decomposing system capabilities into behaviors. The modularity and hierarchy of behavior trees are useful not only during mission plan design but also during mission

execution, thanks to graphical monitoring. Aerostack2 robot behaviors are compatible with the BehaviorTreeCPP² library through their ROS 2 action interface, see Figure 4.10.

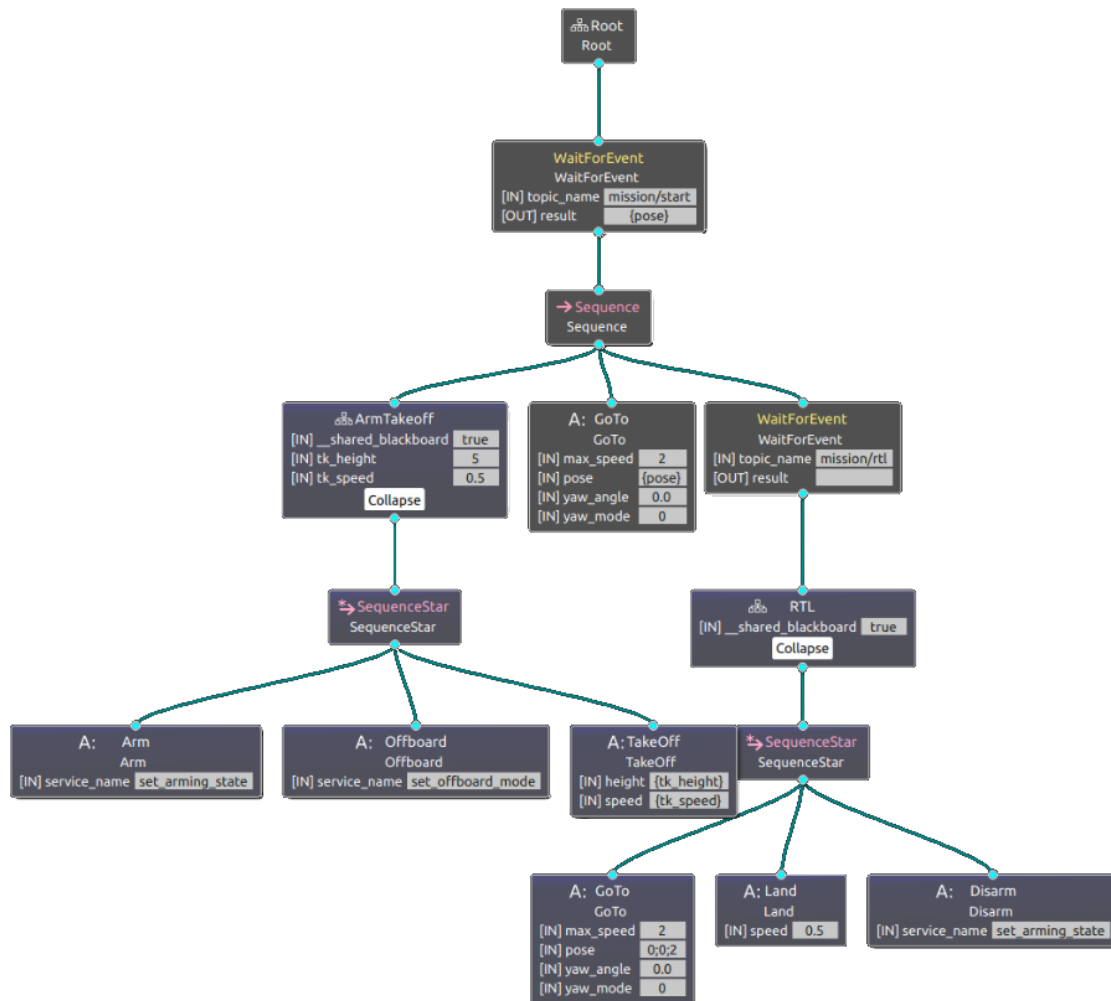


Figure 4.10: Behavior Tree representation of a simple takeoff, go to a position, and wait for a Return to Land (RTL) mission.

All of these methods provide the capability to specify a mission plan in a human-readable way and also include mechanisms for interacting with the rest of the modules in the architecture, both for commanding—activating or deactivating behaviors—and for retrieving information from them, such as the state of the platform to ensure that the next task is feasible.

In Section 4.9, we will further discuss how plan execution and construction can be improved by adding a *Knowledge Base* into the architecture that supports different types of queries.

Multi-Robot Plan Execution

Previously, we stated that the Aerostack2 framework allows for the creation of Tier II+ systems. The reason we consider it beyond a Tier II system is its capability to orchestrate well-defined multi-drone operations in a centralized manner, supporting *FR-11*.

In this layer, we can distinguish two different ways of controlling a multi-robot mission: a central plan executor commands all the agents simultaneously, or the mission plan is divided among

²<https://github.com/BehaviorTree/BehaviorTree.CPP/tree/v3.8>

the agents—avoiding collisions or other disturbances between them—allowing each agent to execute its plan independently.

Each approach has its own pros and cons and depends on the application whether one or the other is more convenient. The central plan executor supports more dynamic and collaborative behaviors, where the plan can be continuously adapted based on the actions of other agents. However, if the central node fails, the entire system fails. In contrast, distributing the mission plan allows agents to cooperate in fulfilling a mission, but collaboration is limited to what is decided during the plan design phase. Once each plan is loaded onto each drone, they can operate independently from the central unit or other agents.

4.8 Mission control and supervision

In the Aerostack2 framework, the top layer is mainly dedicated to the user interface, satisfying *FR-10*. The components of this layer are designed to ease the definition of a mission to a human operator or to help with the supervision of the system. Regarding the level of user expertise, we can differentiate two blocks of tools:

4.8.1 System Development tools

These tools show the internal status of the system, leading to a more detailed explanation of what is happening. The *Alphanumeric Viewer* is a component that monitors the state of specific variables of the system, e.g. sensor measurements, values corresponding to state estimation, references for controllers, etc. The information is distributed in different panes to facilitate the search for a specific variable of the system.

```

- ALPHANUMERIC VIEWER OF AERIAL ROBOTICS DATA -
Key: M (Summary), S (sensors), N (navigation), P (platform)
^
Drone id: /drone_sim_0
Battery charge: 100 %

IMU MEASUREMENTS                                PLATFORM STATUS
Orientation IMU (ypr): 00.00, 00.00, 00.00 rad
Angular speed IMU (ypr): 00.01, 00.01, -00.01 rad/s
Acceleration IMU (xyz): 00.03, -00.09, 09.53 m/s2
Connected: True
Armed: True
Offboard: True

LOCALIZATION
Position (xyz): 001.00, 001.00, 001.10 m
Linear Speed (xyz): 00.00, 00.00, 00.04 m/s
Orientation (ypr): 00.00, 00.00, 00.00 rad
Angular Speed (ypr): 00.00, -00.05, -00.01 rad/s
Status: FLYING

CONTROLLER CONTROL MODE                          PLATFORM CONTROL MODE
Yaw Mode: YAW_ANGLE                               Yaw Mode: YAW_SPEED
Control Mode: POSITION                             Control Mode: SPEED
Frame Mode: LOCAL_ENU_FRAME                       Frame Mode: BODY_FLU_FRAME

```

Figure 4.11: Screenshot of the Alphanumeric Viewer tool. All information is displayed within a terminal, minimizing the computational resources required to run it. It has different panes depending on the data to be inspected.

This component operates in real-time through a Terminal User Interface (TUI), making these tools suitable for onboard debugging in resource-constrained computers. Figure 4.11 shows how this visualizer looks.

On the other hand, tools like *Keyboard Teleoperation* are useful for manipulating the drone swarm in a simple way by sending position and speed commands, allowing the user to check system behavior or take control when autonomous logic fails. This tool enables fast and secure testing and debugging of aerial systems. Figure 4.12 shows the graphical interface of this tool.

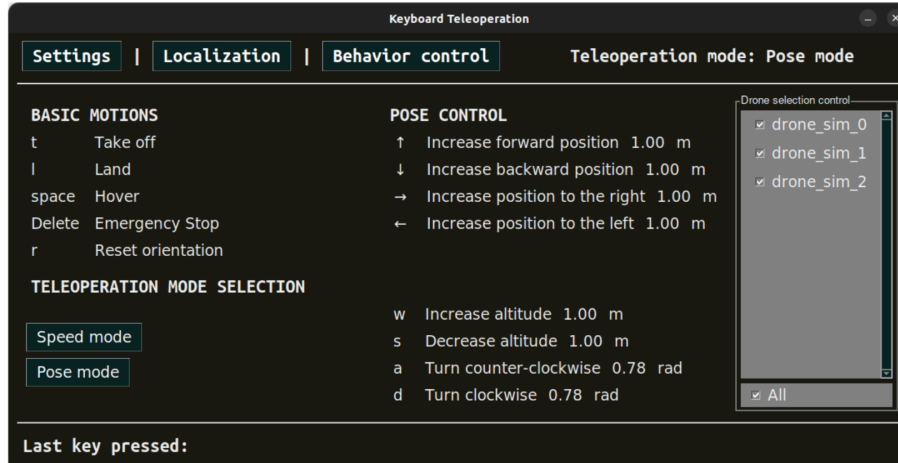


Figure 4.12: Screenshot of the Keyboard Teleoperation pannel.

4.8.2 End Users tools

In this case, the objective is to present all the information in a graphical and simple way so that non-developers can use and interact with the aerial system. In this category of components, Aerostack2 also provides a Graphical User Interface (GUI)³ to use the software framework through a web-based application. This tool is very convenient for rapid planning (e.g. can use an automated planner to generate paths to cover areas) of a mission for one or several drones using graphical resources (e.g., a geographic map and graphical references), which can be done online or offline, allowing repeatability of missions. This user interface is also useful to show graphically details of the mission execution, allowing its monitoring and modification in real-time.

4.9 Knowledge Base

Representing the knowledge possessed by different components of a robotic system in an accessible manner—allowing it to be queried by various modules—facilitates the development of more complex behaviors and planning strategies.

This knowledge can be either *a priori*—known at the engineering or mission definition stage—or *runtime* knowledge, acquired while the robotic system is operating. When this knowledge is represented as a *Model* that enables inference, we can say that the robotic system is building *Understanding*.

The Knowledge Base layer is responsible for two main functionalities: storing and compiling the knowledge that each agent possesses and extracting knowledge from the flows of information within each robotic agent. The *Knowledge Server* component and the Knowledge Extractors are responsible for these tasks, respectively.

Additionally, to handle centralized multi-agent missions, a local Knowledge Server is created for each agent and then merged into a single centralized Knowledge Server, ensuring that there are

³https://github.com/aerostack2/as2_web_gui

no duplicate entities. This strategy allows each drone to perform independent missions, reducing read and reaction times. For example, when two drones share airspace, the shared Knowledge Base enhances mission execution by incorporating entities such as the operator's position or flight status. Furthermore, the drones can infer new knowledge, such as their relative positions (e.g., "close").

4.9.1 Knowledge Server

Mobile robots require comprehensive and precise knowledge about their environment, tasks, actions, and inherent capabilities to make informed decisions and execute tasks successfully across different contexts. Traditionally, this knowledge has been encoded in algorithms and static databases, which often lack the flexibility and scalability needed to handle the dynamic nature of real-world scenarios.

The Knowledge Server centralizes the knowledge that each agent possesses. In Aerostack2, this knowledge is represented using a Knowledge Graph (KG). KGs have emerged as powerful tools for knowledge representation and reasoning. They provide a structured and semantically rich framework for organizing information, enabling both the representation of complex relationships and the implementation of efficient algorithms for knowledge retrieval and reasoning.

A knowledge graph stores information in a structured, graph-based format that represents entities and the relationships between them. Here's a breakdown of the key concepts:

- **Entities as Nodes:** Each real-world object or concept—such as a person, place, organization, or event—is represented as a node. In aerial robotics, we can identify two types of nodes:
 1. **Aerial Robots and Infrastructure:** In this context, nodes represent key physical or digital components. For example, individual drones, ground control stations, sensors, waypoints, and landing zones are all treated as distinct entities. Each node captures the unique attributes of the entity it represents—such as a drone's ID, battery status, or sensor type.
 2. **Environmental Elements:** Nodes can also represent elements of the operating environment, such as geographical landmarks, no-fly zones, or obstacles. This enables the system to integrate situational awareness into the planning and control processes.
- **Relationships as Edges:** The connections between these entities are represented as edges (or links) that connect the nodes.
 1. **Operational Interactions:** Edges in the graph depict interactions between entities. For instance, an edge might connect a drone node to a sensor node, indicating that the sensor is mounted on that particular drone. Similarly, an edge can link a drone node to a waypoint node, representing a planned flight path.
 2. **Spatial and Communication Links:** Relationships can capture spatial associations (e.g., proximity between a drone and an obstacle) as well as communication or control relationships (e.g., the link between a drone and its ground control station). These edges help in understanding dependencies, such as which drones are coordinating for a collaborative task or how data flows between sensors and the central control system.

Additionally, graph nodes can store numerical values as *properties*, which are particularly useful in the robotics domain. The *Knowledge Server* is responsible for enabling the insertion, querying, and deletion of both graph nodes and edges for the other components of the framework.

4.9.2 Knowledge Extractors

Knowledge Extractors consist of a set of nodes responsible for interfacing with the robotic system to extract relevant knowledge that is intended to be included within the *Knowledge Server*. These nodes can subscribe to the various communication flows exposed by different components to generate knowledge without interfering with them. Additionally, *Knowledge Extractors* can also handle existing data in the Knowledge Base to expand the available knowledge by updating it or creating new relationships between entities.

The architectural decision to subscribe to pre-existing information flows was made to avoid interfering with the legacy systems developed in Aerostack2. However, the most effective way to add relevant knowledge is through the integration of *Perception Behaviors*, which aim to model the relevant knowledge of the environment, the robot itself, or other agents.

Knowledge Retrievers

Finally, the stored knowledge can be utilized by different nodes to improve their behavior by reasoning over it. The best example of this is within a mission plan, where decisions can be made based on the results of various queries (inferences) performed on the *Knowledge Server*.

Aerostack2 introduces several functions within the Python API for performing queries over the Knowledge Server, helping with the use of the system knowledge during the mission plan execution:

1. *Node Retrieval*: Functions to retrieve nodes representing entities such as aerial robots, sensors, ground control stations, and environmental elements. These queries can be filtered based on attributes like ID, status, or location.
2. *Edge Retrieval*: Functions to extract edges that denote relationships among nodes. These include interactions (e.g., drone sees human), spatial associations (e.g., proximity to obstacles), and pertinence (e.g., drone has a sensor).
3. *Property-based Querying*: Functions to query specific properties or numerical values associated with nodes and edges. This enables data-driven decisions by accessing parameters such as battery levels, sensor accuracy, or distance measurements.
4. *Subgraph Extraction*: Functions that allow the retrieval of a subset of the knowledge graph based on defined criteria. This capability supports focused analysis and localized reasoning by extracting connected regions of the graph relevant to a particular mission or environment.

4.10 Communications

This part of the system relies on ROS messaging and communication. ROS 2 supports different communication protocols and vendors via the ROS Middleware Interface (RMW), with DDS being one of the most common standards, though alternatives are also used.

This section briefly explains how a *Knowledge Broker* controls agent traffic. The Knowledge Broker plays a central role by abstracting the complexities of heterogeneous communication protocols. Acting as an intermediary, it harmonizes data flow among agents by routing messages based on context and priority while leveraging the protocol-agnostic design of the RMW to integrate various standards. It also implements intelligent filtering to ensure only relevant information is disseminated, enabling scalability, adaptability to changing network conditions, and the easy incorporation of future protocols.

Furthermore, its modular design—featuring a dynamic registry of subscriptions, topics, and communication parameters—enables real-time control of message propagation based on predefined criteria and policies. This enhances system responsiveness and provides a robust framework for fault tolerance and security.

For implementation, Cyclone DDS⁴ manages intra-robot message flow by isolating different ROS networks, while Zenoh Bridges⁵ facilitate inter-agent communication, acting as the *Knowledge Broker*. A set of filtering rules allows developers to determine which messages are broadcast or received and to configure additional parameters, such as network topology and message transfer frequency.

4.11 Validation Examples

This section aims to showcase the capabilities of the Aerostack2 to perform different autonomous drone missions in two different scenarios, trying to ground the concepts previously explained.

4.11.1 Synchronized drone gate crossing.

In this example, two different drones pass through two gates in a synchronized and cyclic fashion. This example was carried out in two phases: The simulation phase, in which the mission is determined and validated using a simulator, and the real flights phase, in which the simulation results are compared with those obtained on real flights.

4.11.1.1 Setup

The simulation was performed using Gazebo simulator and the real experiment was carried out in an area of 60 m^2 equipped with a Motion Capture (MoCap) system that provides the position of drones and gates within the capture area, see Figure 4.13. The UAVs used for this experiment were two: a Bitcraze Crazyflie 2.1 and a Tello Ryze both with IR markers for mocap localization. Due to the limited payload of these micro-UAVs, the computation was performed by the ground station computer.

Module	Simulation	Real
Mission Control	Python API	
Behaviors	Motion, Platform, Trajectory Generator	
State Estimation	Ground Truth	Motion Capture System
Motion Control	DF controller	
Platforms	Gazebo Simulator	Crazyflie / Tello

Table 4.2: Comparison between Aerostack2 modules and plugins used in simulation and real flights of the crossing gates experiment.

4.11.1.2 Modules Used

Table 4.2 shows a comparison between the components used in simulation and real world. For this mission, only basic behaviors are used: motion behaviors for taking off, following a path, and landing. The location of drones and gates is provided by the ground truth of the simulator or the *mocap* system, and for control, a Differential Flatness(DF)-based controller with trajectory references is used.

⁴<https://github.com/eclipse-cyclonedds/cyclonedds>

⁵<https://github.com/eclipse-zenoh/zenoh-plugin-ros2dds>



Figure 4.13: Real experiment environment, where the two doors that must be passed through, the Crazyflie, and the Tello can be observed on the left and right, respectively.

4.11.1.3 Plan generation

The mission was planned using the Aerostack2 Python API. In Listing 4.1 a simplified version of the mission with only one drone is shown. The synchronization mechanism is obtained by waiting for both drones to end their current behavior before activating the follow-up one.

```

1 ...
2 n_laps=3
3 gate_names=["gate1", "gate2"]
4 # Initialize drones interface
5 drone = DroneInterface("tello")
6 drone.arm()
7 drone.offboard()
8 drone.takeoff(height=2.0, speed=1.0)
9 for i in range(n_laps):
10     for gate in gate_names:
11         #The path is on each gate coordinate system
12         drone.follow_path([[ -1, 0, 0], [1, 0, 0]],
13                           speed=1.0,
14                           frame_id=gate)
15 drone.land(speed=0.5)
16 ...

```

Listing 4.1: Simplified mission for Tello drone. The gates poses are provided by the *MoCap* system.

4.11.1.4 Results: Trajectory Comparison

In Fig. 4.14 both simulation and real-world trajectories are plotted, showing the similarities in the trajectory followed in both environments. Moreover, in Fig. 4.15 it can be seen how both UAVs cross the gates in a synchronized and cyclical fashion. This example shows that it is only needed to change the platform and the state estimation component (in this case, a plugin inside of it) to translate the experiment from simulation to the real world, even when the real system is heterogeneous.

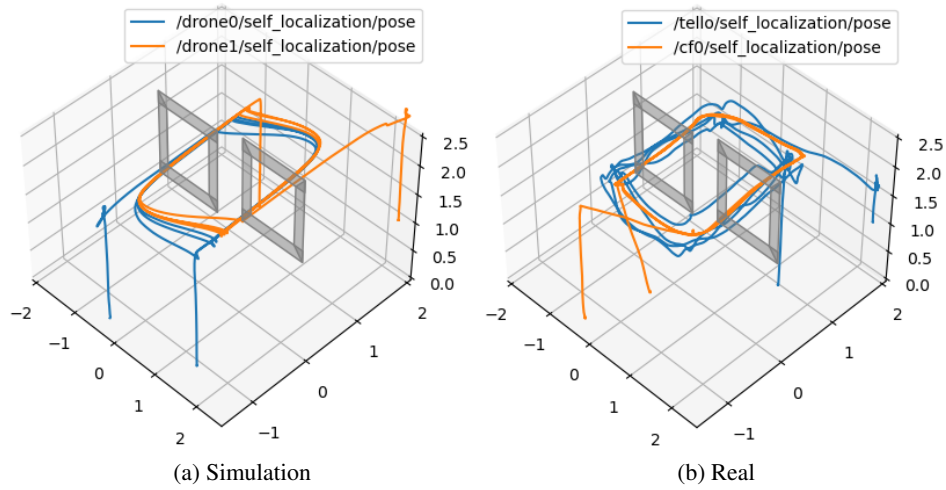


Figure 4.14: Comparison of the trajectories performed by the drones during the gate crossing example.

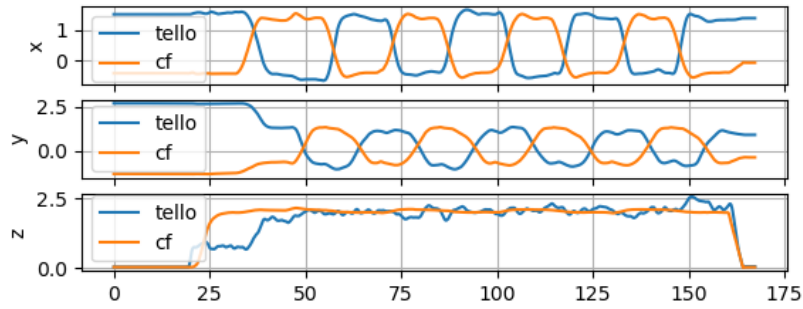


Figure 4.15: Plot of the evolution of the position on each axis of both drones during the real experiment.

4.11.2 Search and Rescue Operation

In this example, two different drones will perform a search and rescue operation in a simulated forest environment. The objective of this example is to illustrate how the knowledge base can be used in a multi-robot operation. In this example, only a simulation case has been considered. The simulation was performed again using the Gazebo simulator, see Figure 4.16.

4.11.2.1 Knowledge Identification

In complex search and rescue operations, understanding the key entities and their interrelationships is crucial for effective mission planning. This analysis identifies the essential components—such as drone positions, sensor inputs, environmental conditions, and operational statuses—and clarifies how they interact. By mapping these elements into the knowledge base, the mission planner can facilitate automated reasoning and decision-making. In this simplified example, Table 4.3 provides an overview of the different tasks and subtasks needed for this mission.



Figure 4.16: Forest environment with a flying drone simulated within the Gazebo simulation.

Task	Sub-task	Input	Output
Traverse the search area	Determine current position of the agent		Current position of each drone
	Plan the route to cover the remaining area	Current position of each drone	Route for complete area coverage
Search and localization of the person	Capture environmental information		Use onboard cameras (visible and thermal)
	Identify the person in the camera image	Camera image	Label as <i>person detected</i> or <i>no person detected</i>
Drone supervision	Monitor battery status		High or low level
	Check relative position between drones	<i>Close</i> or <i>not close</i>	Maintain formation or adjust position
	Verify navigation status		Landed/Flying

Table 4.3: Overview of drone-assisted search and rescue tasks and sub-tasks for efficient person localization and coordination.

From this analysis, the following graph nodes are extracted, representing key entities and their associated properties relevant for the mission completion, see Table 4.4.

In addition to identifying key entities and their properties, the relationships between these entities can also be established. These relationships define interactions that are crucial for modeling the search and rescue mission. Table 4.5 presents the possible relationships between source and target entities, enabling a structured representation for automated reasoning and decision-making.

Entity	Associated Property
Drone	Current pose
Battery	Voltage
Person	Location
Status	Disarmed / Flying / Landed
Home Station	Location

Table 4.4: Key entities and their associated properties in the search and rescue mission.

Source Entity	Possible Relationship	Target Entity
Drone	Looking for Located	Person
Drone	Has status	Status
Drone	At Outside	Home Station
Drone	Battery level: High Battery level: Medium Battery level: Low	Battery
Drone	Close to	Drone

Table 4.5: Relationships between key entities in the search and rescue mission.

By combining the identified entities, properties, and relationships, various queries can be performed to monitor the mission's progress and trigger necessary adjustments in real time. Table 4.6 presents key queries, their possible return values, and the corresponding actions that should be taken to adapt the mission plan dynamically.

Query	Return Value	Action
Battery level	Low	Drone returns to home station
Inspection status	Person located	Send position to home station
Relative position between drones	Close	Drone moves a safe distance away

Table 4.6: Key queries for the search and rescue mission, their return values, and corresponding actions to update the mission plan.

After this identification, multiple *Knowledge Extractors* have been implemented to insert and update the relevant nodes and edges within the *Knowledge Base*, ensuring that the information can be used in the mission planning and plan execution stages.

4.11.2.2 Mission plan definition

Now, we can take advantage of the knowledge retrieved from the system to facilitate mission plan definition, support dynamic monitoring, and make decisions based on it. Below, a simplified search and rescue mission is shown:

```

17 ...
18 if battery_low():
19     print('Battery is low, can not start the mission')
20     return
21 drone_interface.takeoff(TAKEOFF_HEIGHT)
22 set_home()
23 search_path = divide_search_path(limit_coords, n_agents)
24 drone_interface.follow_path(
25     search_path, speed=SPEED, yaw_angle=0.0, wait=False)
26
27 while drone_interface.follow_path.is_running():
28     if not battery_low():
29         drone_interface.follow_path.cancel()
30         break
31     elif person_located():
32         print(f'person located at: {drone_interface.position()}')
33         drone_interface.follow_path.cancel()
34         sleep(0.1)
35
36 if person_located():
37     drone_interface.hover()
38     wait_return_signal()
39
40 print('going home')
41 drone_interface.go_to(home, speed=SPEED, wait=True)
42 drone_interface.land(wait=False)
43 print('Mission finished')
44 ...

```

Listing 4.2: Simplified mission for a single drone. The functions represent calls to the knowledge base.

4.11.2.3 Results: Graphs evolution

The system is capable of performing the whole mission successfully. The status and evolution of the system can easily be observed through the knowledge graph. This helps explain what the system is doing and why a decision is made, which is crucial for monitoring and debugging errors during development phases.

Figure 4.17 represents the initial status of the knowledge graph, while Figure 4.17 shows the state when one drone finds the person, showcasing the utility of the knowledge graph for structuring mission plans and monitoring them.

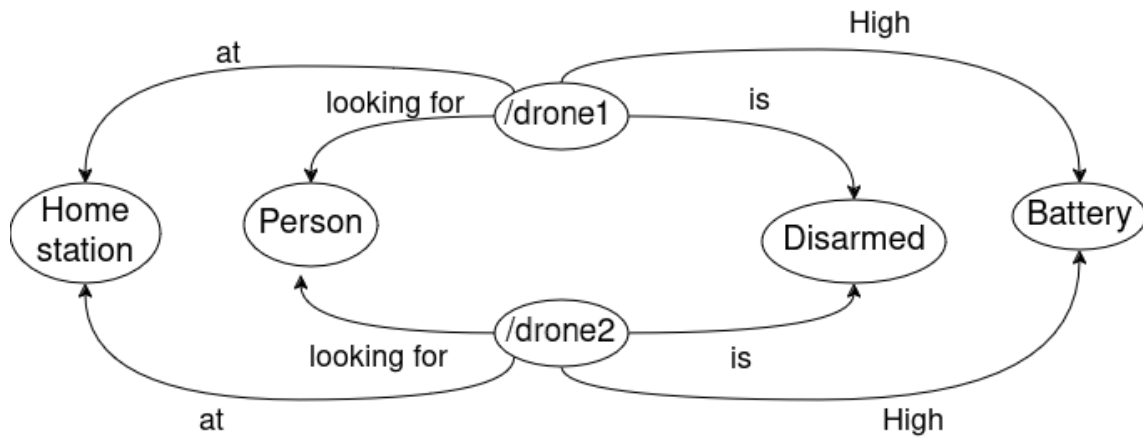


Figure 4.17: Status of the Knowledge Graph at startup: both drones are at the home station, disarmed, with high battery levels, and tasked with searching for the lost person.

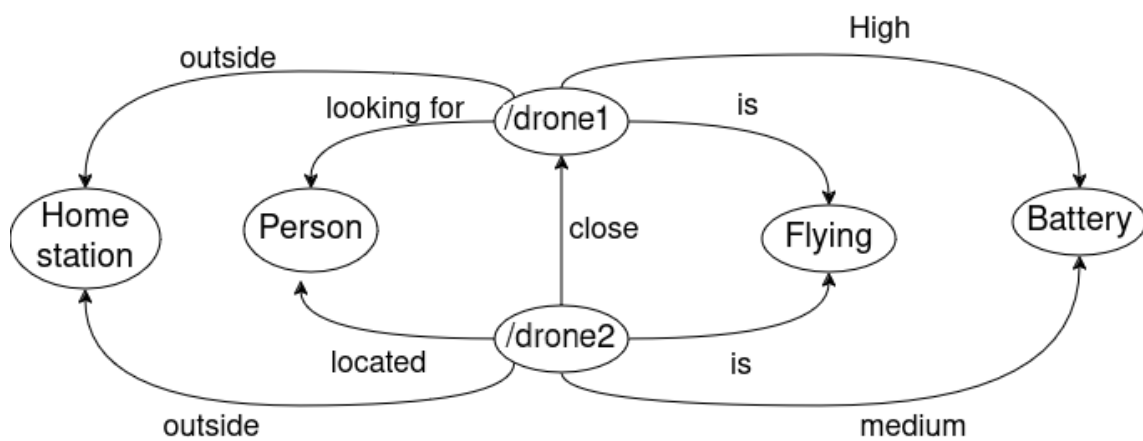


Figure 4.18: Knowledge Graph after one drone locates the person and communicates the position to the other drone, allowing it to approach. This is represented by the fact that the two drones are close to each other, one of them has located the person, and both are still flying.

Part III

Algorithms proposed

Chapter 5

Dynamic trajectory following for UAVs

5.1 Introduction

Multirotors are highly versatile and agile aerial robotic platforms due to their maneuverability and simplicity. When these UAVs operate in real-world autonomous missions, they must handle numerous uncertainties related to self-localization, environment recognition, and dynamic modeling. These challenges are often compounded by environmental changes, emphasizing the critical importance of adaptability to varying conditions (Wu and Low, 2020)(Zollars et al., 2019).

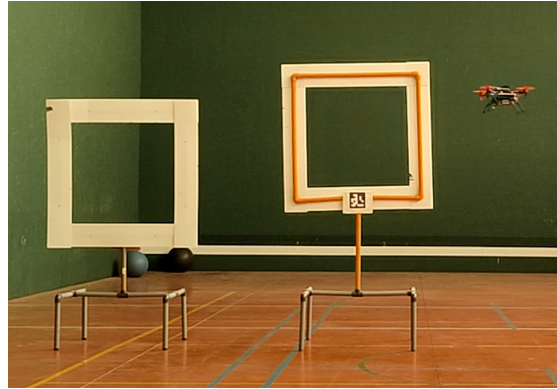


Figure 5.1: Drone flying through gates during the real-world flights experiments. The gate position estimation was done using ArUco markers.

In this chapter, we present an algorithm for generating adaptive trajectories that smoothly and dynamically respond to changes in the set of waypoints. Our approach combines a polynomial trajectory generator, which optimizes the trajectory in terms of snap and serves as the baseline, with Local Gaussian Modifiers (LGMs). These modifiers adjust the baseline trajectory when re-computing the entire trajectory is not feasible. Additionally, we introduce a strategy for smoothly

stitching two trajectories together.

Compared to other UAV trajectory generators, this approach is designed to accommodate the continuous sampling and generation of new trajectories, ensuring smooth transitions in a simple and transparent manner for the user.

5.2 Methodology

5.2.1 Notation

In this chapter, we use the global inertial frame W to plan and generate all trajectories. For vectorial variables, functions, and constants, we use bold letters, such as $\mathbf{x}$. The tilde notation represents the updated version of a variable before it is incorporated into the trajectory. For example, $\tilde{\mathbf{w}}$ denotes the update of the waypoint position before recalculating the new trajectory.

5.2.2 Problem formulation

Given a set of N dynamic waypoints $\mathbf{w}$, our objective is to compute a trajectory $\mathbf{F}(t)$ that passes through all waypoints while minimizing the traversal time.

We define a dynamic waypoint $\mathbf{w} = [x, y, z]^T$ as a 3D point with an ID, whose position can change over time. Each waypoint may have additional constraints, such as velocity $\dot{\mathbf{w}}$ or acceleration $\ddot{\mathbf{w}}$, that the aircraft must satisfy when passing through it.

Mathematically, we denote a trajectory as a function $F(t) : [0, T] \rightarrow \mathbb{R}^3$; $F(t) \in C^3$ where:

$$\mathbf{F}(t) = \mathbf{x}(t) \quad \mathbf{F}(t) \in \mathbb{R}^3 \quad (5.1)$$

$$\dot{\mathbf{F}}(t) = \dot{\mathbf{x}}(t) = \mathbf{v}(t) \quad \dot{\mathbf{F}}(t) \in \mathbb{R}^3 \quad (5.2)$$

$$\ddot{\mathbf{F}}(t) = \ddot{\mathbf{x}}(t) = \mathbf{a}(t) \quad \ddot{\mathbf{F}}(t) \in \mathbb{R}^3 \quad (5.3)$$

subject to:

$$\forall \mathbf{w}_i \exists t_i \mid \mathbf{F}(t_i) = \mathbf{w}_i \quad i \in \{0, N\} \quad (5.4)$$

$$\max(\|\dot{\mathbf{F}}(t)\|) \leq V_{max} \quad \forall t \in [0, T] \quad (5.5)$$

$$\max(\|\ddot{\mathbf{F}}(t)\|) \leq A_{max} \quad \forall t \in [0, T] \quad (5.6)$$

$\mathbf{x}(t) = \{x(t), y(t), z(t)\}$ corresponds to the position of the UAV at time t , while $\dot{\mathbf{x}}$ and $\ddot{\mathbf{x}}$ represent its time derivatives. As stated in the constraints, the trajectory must pass through all waypoints and is subject to maximum speed and acceleration limitations.

The process of computing a trajectory $\mathbf{F}(t)$ requires a significant amount of time, denoted as $T_{\text{computation}}(n)$, which increases with the number of waypoints n . During this computational period, the UAV continues following the previous trajectory and cannot adjust it. Given the limitations of onboard computation, this delay is non-negligible, making it crucial to minimize trajectory correction time for agile flights.

In this work, we assume that upon reaching a waypoint, there is a temporal threshold T_{security} below which there is insufficient time to recompute the trajectory using conventional methods.

$$T_{\text{security}} = C_{\text{security}} \cdot \hat{T}_{\text{computation}}(n); \quad (5.7)$$

where $\hat{T}_{\text{computation}}(n)$ is an estimate of the time required to compute an n -waypoint trajectory, and $C_{\text{security}} > 1$ is a safety constant ensuring that the UAV has sufficient time to react after the trajectory is modified, accounting for latencies in perception, processing, and actuation processes.

Within this threshold, the UAV is essentially flying blindly, making it impossible to correct the trajectory to ensure waypoint traversal. Our goal is to develop a lightweight trajectory modification method that minimizes the UAV's reaction time, thereby increasing the success rate of reaching the waypoints. However, it is important to note that these modifications may lead to following a suboptimal trajectory.

5.3 Dynamic Trajectory Generation Algorithm Proposal

When a UAV is flying at high speeds and the waypoint position changes, fast reactivity is crucial for avoiding collisions. This reactivity is limited by the computational cost involved in generating these trajectories.

In this approach, we propose a holistic pipeline that considers how quickly a new trajectory can be generated safely, ensuring that the modified trajectory will pass through the updated waypoint even when there is not enough time to recompute the optimal trajectory.

The estimation of $\hat{T}_{\text{computation}}(n)$ is calculated online based on the average time taken by previous trajectories with n waypoints. This estimation is continuously updated as new trajectories are computed, accounting for the computational load of the onboard computing system during flight.

Using T_{security} , we can define a security zone as the period during which the UAV is closer to the next waypoint than the safety time T_{security} , as illustrated in Figure 5.2.

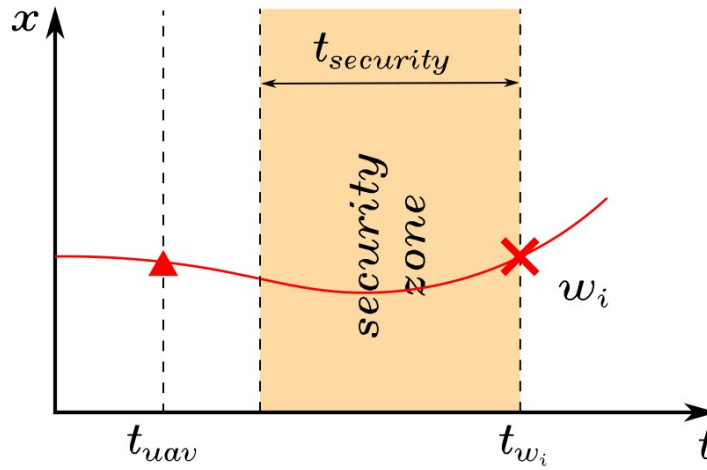


Figure 5.2: Diagram representing the security zone based on T_{security} with respect to the next waypoint w_i , when the UAV is at t_{uav} .

Whenever a trajectory modification is required, depending on the stage of the flight, we consider two main situations:

1. The UAV is **outside** the security zone: In this situation, the UAV is sufficiently far from the next waypoint, allowing it to recompute the optimal trajectory effectively. Here, the new trajectory is computed to be smoothly stitched with the current trajectory.
2. The UAV is **inside** the security zone: In this situation, the UAV is too close to the next waypoint to recompute the trajectory optimally, requiring a rapid modification to ensure waypoint traversal. Here, we employ Local Gaussian Modifiers (LGMs) to efficiently modify the current trajectory in real time.

The following section delves into the detailed approach for addressing each of these challenges.

5.3.1 Generating Baseline Trajectories

The objective of this work is to improve the reactivity of trajectory modifications while striving to maintain the optimality of trajectories computed using optimal control principles. The first part of our work consists of generating polynomial snap-minimum trajectories and proposing a strategy to smoothly stitch two trajectories together. Below, we explain these two parts in detail.

5.3.1.1 Polynomial Trajectory Generation

We use polynomial-based minimum snap multi-waypoint trajectory planning algorithms (Richter et al., 2016; Burri et al., 2015) for trajectory generation, chosen for their simplicity and computational efficiency.

To generate these trajectories, a set of ordered dynamic waypoints is required, where the order indicates the sequence in which the UAV will reach them. When a trajectory is generated from scratch, constraints on maximum speed and acceleration are imposed to ensure the feasibility of the generated trajectory.

For this trajectory generation approach, it is necessary to know the state of the UAV when the process begins, ensuring that trajectories start from the UAV's current position.

To generate a baseline trajectory $\mathbf{P}(t)$, we utilize a polynomial trajectory generator based on the work of Richter et al. (Richter et al., 2016). This approach can be extended to more complex trajectory generators such as CPC (Foehn et al., 2021), but in this work, we opt for a simple and effective method that balances performance and computational cost. The chosen approach generates piecewise polynomial minimum snap trajectories based on the differential flatness property of quadrotor dynamics. This trajectory is expressed as:

$$\mathbf{P}(t) = \begin{cases} \sum_{i=0}^n c_{i,1} t^i & t_0 \leq t < t_1 \\ \sum_{i=0}^n c_{i,2} t^i & t_1 \leq t < t_2 \\ \vdots & \\ \sum_{i=0}^n c_{i,N} t^i & t_{N-2} \leq t < t_{N-1} \end{cases} \quad (5.8)$$

where N represents the number of waypoints, n is the order of the polynomial, and $c_{i,j}$ for $i = 0, \dots, n$ and $j = 1, \dots, N$ are the coefficients of each polynomial segment. More details on computing these trajectories can be found in (Richter et al., 2016; Burri et al., 2015).

5.3.1.2 Trajectory Stitching Constraints

When the aircraft is outside the security zone, modifying the trajectory involves generating a new base trajectory $\tilde{\mathbf{P}}(t)$ from scratch, incorporating updated positions of dynamic waypoints or adding new ones.

While the new trajectory is being generated, the UAV continues following the old trajectory until the new one is computed and replaces it. To maintain smoothness throughout the entire path, the new trajectory is designed to ensure a seamless transition between the old and new trajectories.

To minimize discontinuities during the trajectory swap, the new trajectory is computed using a set of N_{smooth} swapping waypoints $\mathbf{w}_s$, placed between the instant when the new trajectory is generated, t_{gen} , and the estimated time of swapping between the old and new trajectories, $t_{\text{computation}}$. These waypoints facilitate a smooth transition between trajectories.

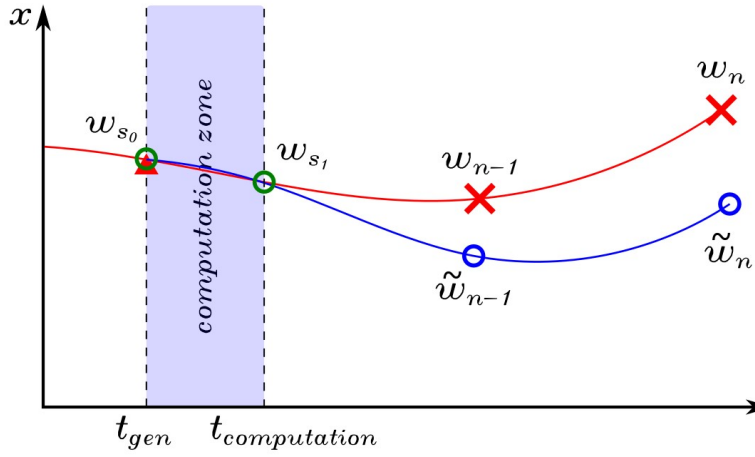


Figure 5.3: New base trajectory (blue) generated from a previous trajectory (red) using swapping waypoints (green circles) and modified waypoints (blue circles). Red crosses represent the previous positions of each waypoint, while the red triangle indicates the UAV's position when the new trajectory is computed.

This set of swapping waypoints is derived from the old trajectory to ensure that the new base trajectory remains similar to the previous one during the computational time $T_{\text{computation}}(n)$:

$$\mathbf{w}_{s_i} = \mathbf{F}(t_{\text{gen}} + i \cdot t_{\text{offset}}), \quad i = 0, \dots, N_{\text{smooth}} \quad (5.9)$$

$$\dot{\mathbf{w}}_{s_i} = \dot{\mathbf{F}}(t_{\text{gen}} + i \cdot t_{\text{offset}}), \quad i = 0, \dots, N_{\text{smooth}} \quad (5.10)$$

$$\ddot{\mathbf{w}}_{s_i} = \ddot{\mathbf{F}}(t_{\text{gen}} + i \cdot t_{\text{offset}}), \quad i = 0, \dots, N_{\text{smooth}} \quad (5.11)$$

where t_{gen} is the time at which the new trajectory begins to be generated, and t_{offset} represents the temporal displacement between each waypoint:

$$t_{\text{offset}} = \hat{T}_{\text{computation}}(n) / N_{\text{smooth}}. \quad (5.12)$$

These swapping waypoints are inserted before the new or modified waypoints through which the trajectory is planned to pass, as depicted in Figure 5.3. Finally, once the new trajectory is computed, it replaces the previous one from that point onward.

5.3.2 Fast Trajectory Modification

When the aircraft is inside the security zone, it is unable to safely recompute a base trajectory, lacking the capability to correct the trajectory in a robust manner. In this scenario, we must use faster but suboptimal approaches to enable real-time modifications.

To make small adjustments to the trajectory, we propose implementing local modifications near the affected waypoint, ensuring that the trajectory smoothly and efficiently passes through it. For these modifications, we have developed **Local Gaussian Modifiers (LGMs)**, which require a computation time more than two orders of magnitude lower than that needed for generating a base trajectory.

We define an **LGM**(t) : $\mathbb{R}^+ \rightarrow \mathbb{R}^3$ as:

$$\mathbf{LGM}(t) = \mathbf{A}e^{-\frac{(t-\mu)^2}{2\sigma^2}} \quad (5.13)$$

where $\mathbf{A} \in \mathbb{R}^3$ represents the magnitude of the modification in the waypoint's position along each axis, and $\mu, \sigma \in \mathbb{R}^+$ are constants computed when each LGM is created. Each dynamic waypoint can have multiple LGMs associated with it.

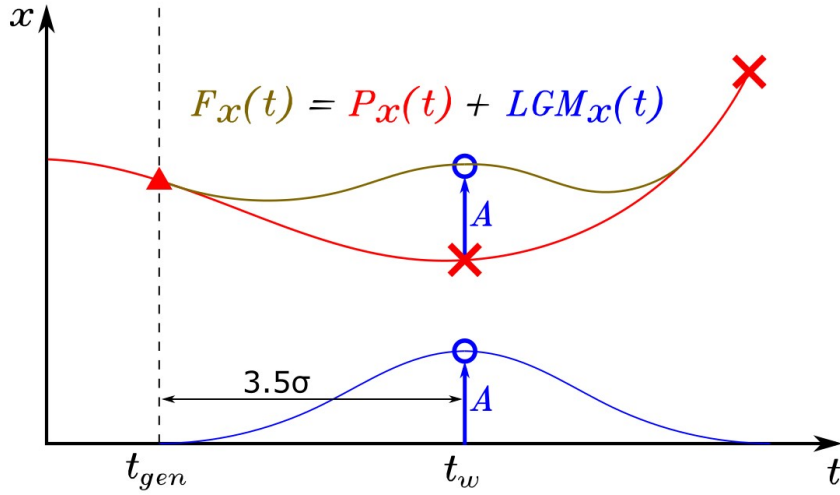


Figure 5.4: One-dimensional representation of a generated trajectory (yellow) with a base trajectory (red) and a Local Gaussian Modification (blue) applied to waypoint w with a displacement A along the x -axis. The red triangle represents the UAV's position when the modification is applied, red crosses indicate the original positions of two waypoints, and the blue circle marks the modified position of waypoint w .

In our approach, we compute an adaptive trajectory that combines a polynomial trajectory $\mathbf{P}(t)$ and a Local Gaussian Modification (LGM). Using these components, we define the time evaluation of our dynamic trajectory $\mathbf{F}(t)$ as:

$$\mathbf{F}(t) = \mathbf{P}(t) + \sum_{i=0}^{N_w} \sum_{j=0}^{N_{m_i}} \mathbf{LGM}_{i,j}(t) \quad (5.14)$$

where N_w represents the number of waypoints in the trajectory, and N_{m_i} is the number of modifications for the i -th waypoint.

Each LGM has three associated constants: $\mathbf{A}$, σ , and μ . To achieve the desired behavior, these constants must be computed before adding the modifier to the trajectory. This process is illustrated in Equation (5.14), where each LGM result is added incrementally to the latest computed trajectory value.

Given a trajectory $\mathbf{F}(t)$ consisting of N waypoints, each waypoint w_i has a position $\mathbf{x}_i$ and a corresponding time t_{w_i} along the trajectory, meaning that $\mathbf{F}(t_{w_i}) = \mathbf{w}_i$. When a modification $\tilde{\mathbf{w}}_i$ occurs at time t_{mod} , a new modifier $\mathbf{LGM}_i(t)$ is generated (see Figure 5.4). If multiple modifications occur for the same waypoint, an LGM is generated for each one. The constants of the modifiers are computed as follows:

$$\mathbf{A} = \begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix} = \tilde{\mathbf{w}}_i - \mathbf{w}_i = \begin{bmatrix} \tilde{w}_{ix} - w_{ix} \\ \tilde{w}_{iy} - w_{iy} \\ \tilde{w}_{iz} - w_{iz} \end{bmatrix} \quad (5.15)$$

$$\mu = t_{w_i} \quad (5.16)$$

$$\sigma = \frac{|t_{mod} - \mu|}{3.5} \quad (5.17)$$

where $\mathbf{A}$ represents the amplitude of the modification, μ denotes the temporal position of the waypoint on the current trajectory, and σ represents the Gaussian variance, which determines the smoothness of the trajectory adaptation.

These parameters are selected to ensure that $\mathbf{LGM}(t_{\text{mod}}) \approx \mathbf{0}$, guaranteeing a smooth transition in the trajectory with each modification. Due to the properties of the Gaussian function (Huber, 2018), approximately 99.98% of the contribution of $\mathbf{LGM}(t)$ falls within the range $t \in [-3.5\sigma, \mu + 3.5\sigma]$. Choosing σ in this way ensures that adding these modifiers to $\mathbf{F}(t)$ preserves the continuity and differentiability conditions of the trajectory.

The low computational cost of generating and evaluating LGMs allows multiple modifiers to be appended to the same trajectory, enabling high reactivity to trajectory changes while maintaining smoothness. This means that when multiple changes occur at the same waypoint, the final trajectory will account for all modifications, as mathematically described in Equation (5.14), due to the cumulative effect of all LGMs at that waypoint.

In this approach, we assume that the UAV's dynamic limits are not reached when computing polynomial trajectories. However, we must still consider speed and acceleration limits when incorporating LGMs.

5.4 Experiment and Results

5.4.1 Experimental setup

All flight experiments have been conducted using ROS 2 Galactic with Aerostack2. The dynamic trajectory generator has been implemented in C++ and is publicly available¹. For the generation of polynomial trajectories, we use a library developed at ETH Zurich², based on the work of Richter et al. (Richter et al., 2016).

This dynamic trajectory generation implementation has been integrated into the Aerostack2 framework as a *Behavior* within the `GeneratePolynomialTrajectory Behavior`³. All flight experiments conducted in this section were performed using Aerostack2 for both simulated and real flights.

Two additional components have been developed for this test: the `DetectArUCo Behavior` and the `Gates2Waypoints` module, which manages plan execution control. The latter is responsible for tracking waypoint modifications measured with ArUCo and triggering waypoint updates within the `GeneratePolynomialTrajectory Behavior`. Figures 5.7 and 5.8 show the system architecture used in both simulated and real flights.

For all the experiments described in this work, we use $C_{\text{security}} = 5$. This value is required for dynamic trajectory generation, as mentioned in Equation (5.7). Similarly, we experimented with different values for N_{smooth} , determining that 1 and 2 are the most suitable choices.

5.4.2 Validation of Fast Trajectory Modification

The first experiment aims to assess the performance of the dynamic trajectories generated only using LGMs to update the trajectory. In this experiment, we generate a trajectory with five waypoints, and as long as the UAV progresses along it, some waypoints increase their distance from the former trajectory. Figure 5.5 illustrates the different modifications made to the main trajectory, while Figure 5.6 shows the references obtained by the quadrotor throughout the track.

¹https://github.com/miferco97/dynamic_trajectory_generator

²https://github.com/ethz-asl/mav_trajectory_generation

³https://github.com/aerostack2/aerostack2/tree/main/as2_behaviors/as2_behaviors_trajectory_generation/generate_polynomial_trajectory_behavior

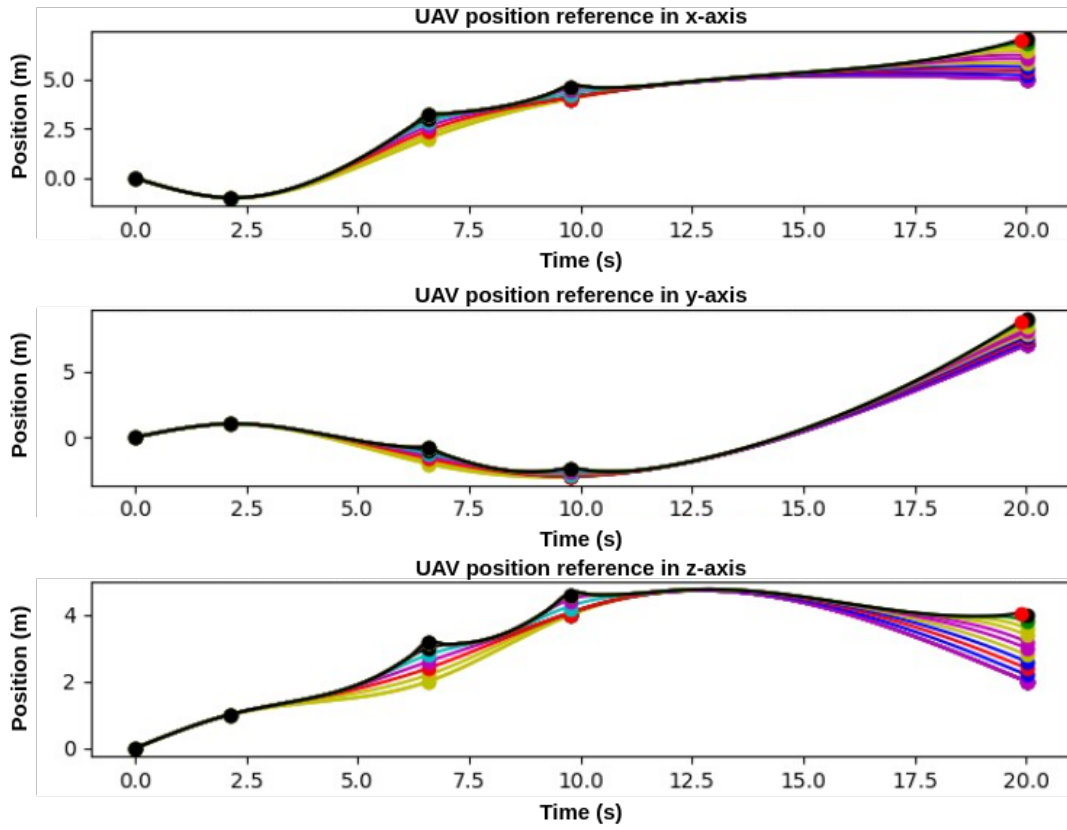


Figure 5.5: Axis by axis (x, y, and z-axis, corresponds with top, middle, and bottom subfigures, respectively) plot of several dynamic trajectories position references generated using only LGMs to modify the waypoint positions for different adjustments of some waypoints.

Some profiling tests have been performed on both the laptop computer and the on-board computer with respect to the time spent computing the base trajectories (see TABLE 5.1) and the time dedicated to generating and evaluating LGMs (see TABLE 5.2).

	Laptop computer	Jetson Xavier NX
6 points	$13.71 \pm 0.14 \text{ ms}$	$51.47 \pm 1.57 \text{ ms}$
14 points	$88.93 \pm 3.11 \text{ ms}$	$321.07 \pm 22.11 \text{ ms}$
26 points	$309.08 \pm 10.20 \text{ ms}$	$1027.76 \pm 19.36 \text{ ms}$

Table 5.1: Time spent generating base polynomial trajectories for different numbers of points in a laptop computer and in an onboard computer.

	Laptop computer	Jetson Xavier NX
1 LGMs	$52.9 \pm 8 \text{ ns}$	$207 \pm 54 \text{ ns}$
8 LGMs	$223 \pm 34 \text{ ns}$	$1123 \pm 112 \text{ ns}$
64 LGMs	$1544 \pm 303 \text{ ns}$	$7694 \pm 442 \text{ ns}$
512 LGMs	$12451 \pm 504 \text{ ns}$	$61374 \pm 829 \text{ ns}$

Table 5.2: Time spent generating and evaluating the sum of multiple LGMs acting on the same trajectory on a laptop computer and on an onboard computer.

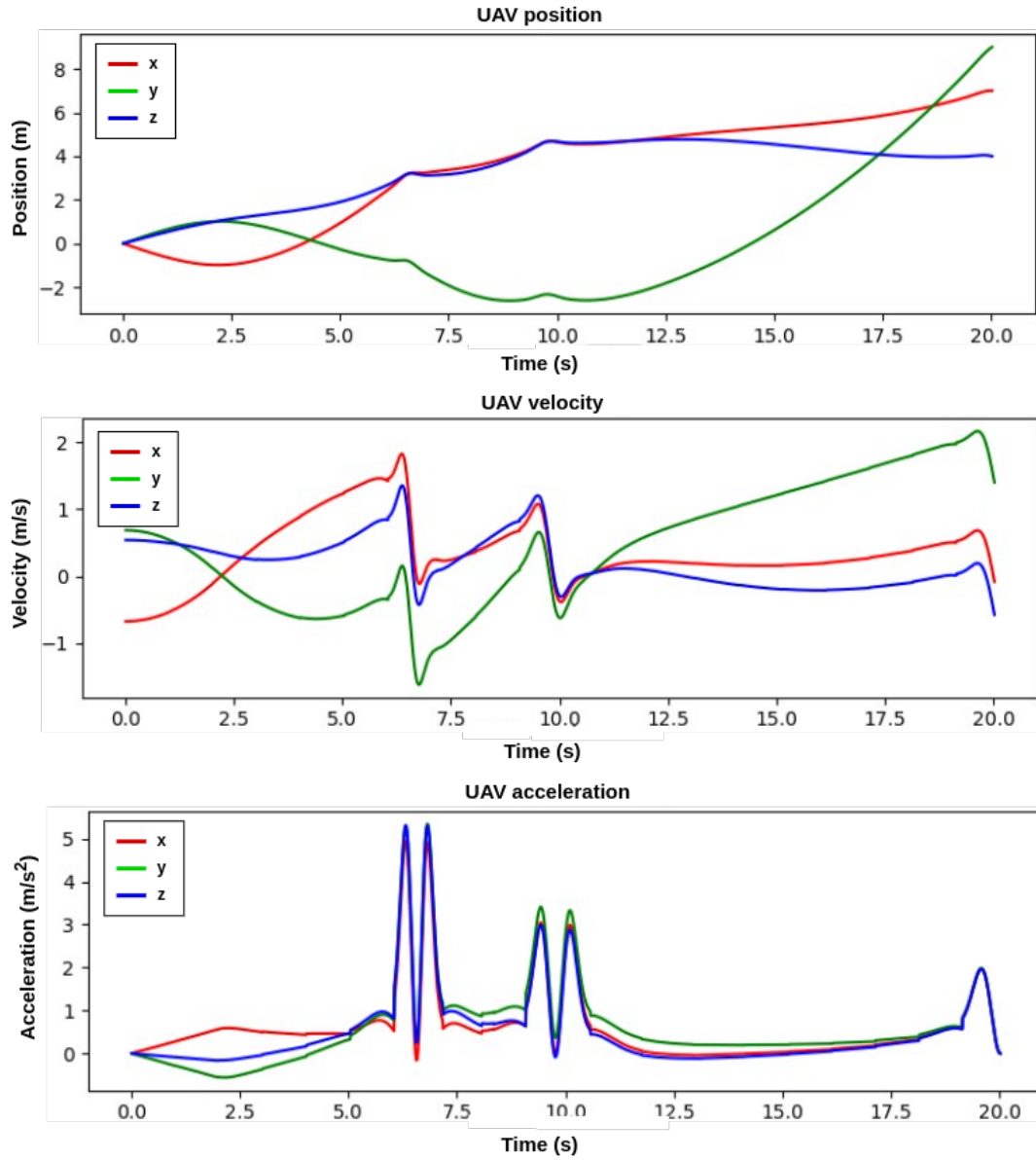


Figure 5.6: Position, velocity, and acceleration references generated by the trajectory generation during the following of the dynamic trajectory shown in Fig. 5.5.

5.4.3 Simulation Flights

The simulation flights were carried out using the Gazebo simulator (Koenig and Howard, 2004). In these experiments, all computations were performed on a laptop running Ubuntu 20.04 with an Intel i7-10870H processor @ 2.2 GHz.

To evaluate the proposed approach in simulation, an experiment was designed with a circuit containing four moving gates. The drone initially knows the positions of the gates to generate a trajectory through the circuit. These gates move back and forth from their original positions, covering a distance of 1 meter at a constant speed of 0.1 m/s. This motion forces the drone to adapt its trajectory to pass through each gate. The positions of both the drone and the gates were obtained from the simulator's ground truth.

Three rounds of experiments were conducted to assess the impact of the system's computation

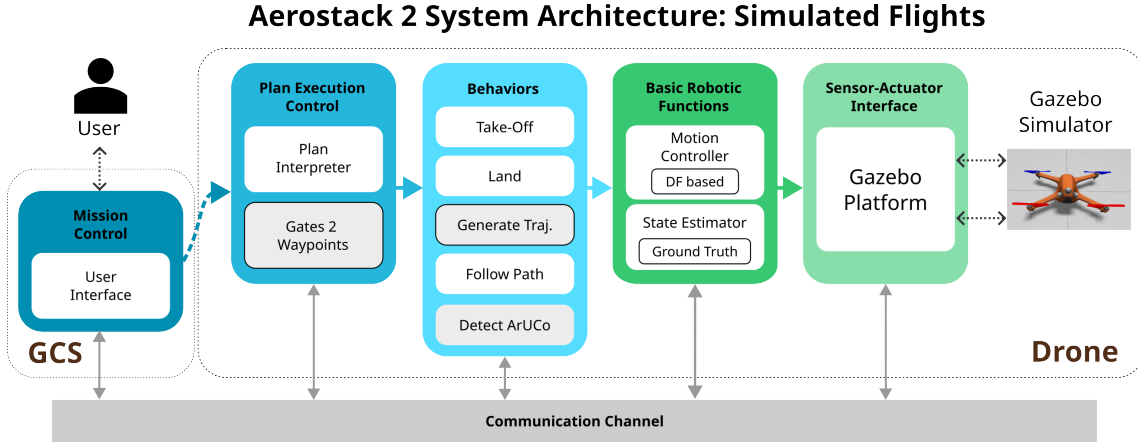


Figure 5.7: Aerostack2 components used for the simulation experiments. The gray boxes represent the additional components required for this experiment.

time on trajectory generation and modification. In the first round, the system operated under normal computation time. In the subsequent rounds, artificial delays of 0.5 seconds and 1 second were introduced to simulate processing on a lower-performance computer, as shown in Table 5.3.

For each of these rounds, the speed limit was set to 5, 10, 15, and 20 m/s, with 10 runs performed for each experimental condition. In each run, the drone had to complete four laps of the circuit. The measured metrics included the maximum and mean speeds of the generated trajectory, the maximum and mean speeds achieved by the drone during the run, the total elapsed time of the runs, and the proportion of laps completed without collision.

CTA (s)	Vmax (m/s)	Ref. speed (m/s)		Real speed (m/s)		Elapsed time (s)	Success rate
		Max	Mean	Max	Mean		
0.0	5	5.15	1.54	5.25	2.23	48.5	0.88
	10	8.59	2.34	9.07	2.94	35.96	0.85
	15	13.05	2.78	11.41	3.55	31.1	0.9
	20	10.28	1.90	11.19	2.45	44.7	0.88
0.5	5	4.47	1.87	5.10	2.39	42.30	0.90
	10	9.50	3.53	9.48	3.61	29.54	0.90
	15	13.63	3.27	13.80	3.35	31.25	0.95
	20	15.64	3.58	16.01	3.70	29.95	0.75
1.0	5	4.71	2.25	4.7	2.29	45.90	0.92
	10	9.63	3.33	11.08	3.46	31.1	0.85
	15	13.61	4.77	13.82	4.87	21.80	0.90
	20	15.58	4.06	16.72	3.24	27.47	0.65

Table 5.3: Results for simulated flight with computing time. CTA: computation time added, Ref. speed: reference speed of the trajectory generator, Real speed: actual speed achieved by the drone, Elapsed time: duration of the run, Success rate: proportion of laps completed without collision on each run.

5.4.4 Real Flight Validation

The aerial platform used for the real experiments was a custom quadrotor based on the DJI F330 frame. This platform was equipped with a Pixhawk 4 Mini as the aircraft autopilot, an Intel RealSense T265 Tracking Module for state estimation, and a USB fisheye camera for gate detection.

Additionally, the quadrotor featured an NVIDIA Jetson Xavier NX single-board computer (SBC) with a 6-core ARM® v8.2 64-bit processor, where all computations were performed.

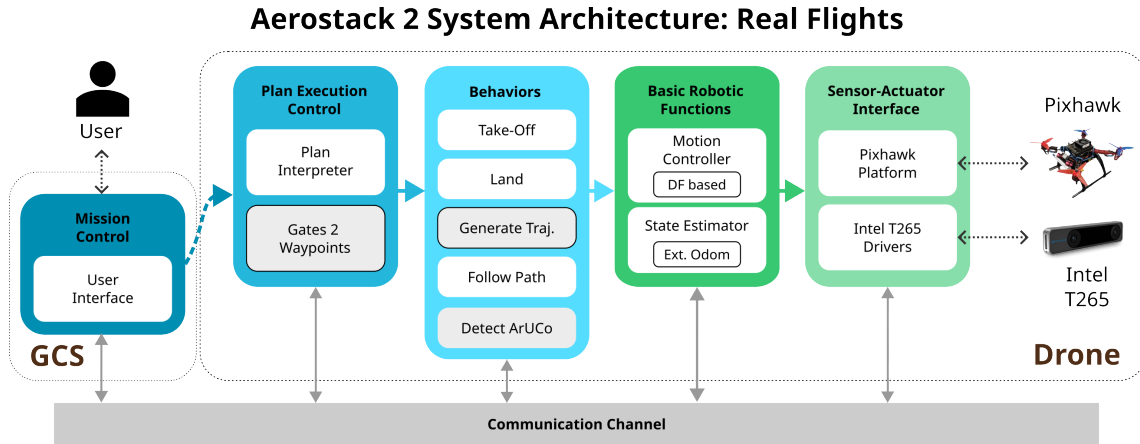


Figure 5.8: Aerostack2 components used for the real-world experiments. Compared to the simulated experiments, the Aerial Platform has been replaced with the Pixhawk system, the camera Realsense T265 has been integrated into the Sensor-Actuator interface, and the State Estimator plugin has been modified to use the odometry provided by the external sensor.

To validate the approach in a real-world environment, an additional experiment was conducted. The experiment involved navigating through a small circuit with two gates per lap at increasing speeds. The initial speed was set to a maximum of 0.5 m/s and was incremented by 0.5 m/s per lap until reaching 3.5 m/s, as illustrated in Fig. 5.9 and Fig. 5.10.

During the flight, the drone visually estimated the gate positions using ArUco markers (Garrido-Jurado et al., 2015; Romero-Ramirez et al., 2018). An approximate position of the gates was provided beforehand to allow trajectory computation even before the gates were detected. Due to the uncertainty in the vision-based estimation of both the gate positions and the drone's state, the trajectory had to be continuously updated to ensure successful passage through the gates.

During real flights we could validate the proposed algorithm in a real world scenario with static gates and modifying the estimated position of each gate for compensate the drift of the Visual Odometry State Estimation provided by the T265. The video ⁴ shows this experiment and can be seen how the trajectory is modified in the Rviz Visualizer.

⁴<https://vimeo.com/683638197>

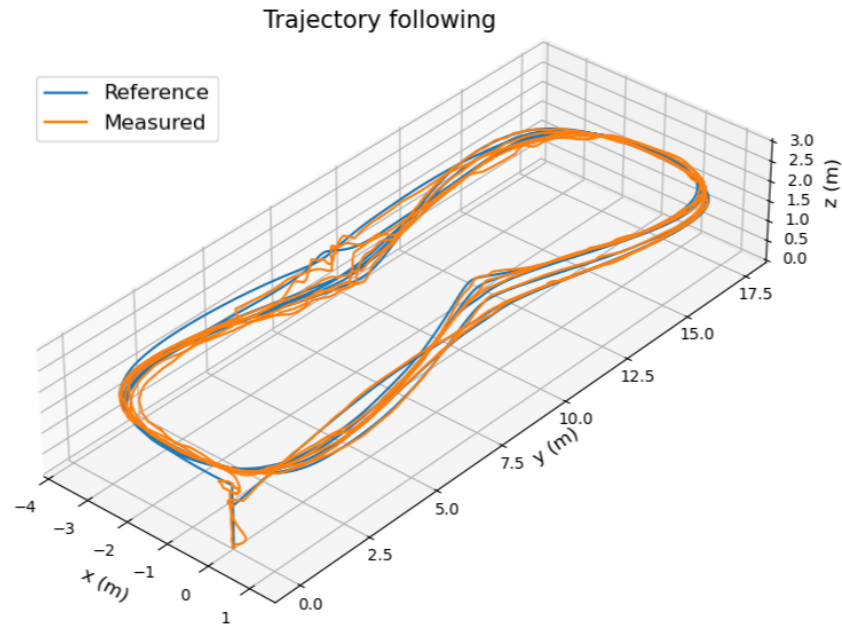


Figure 5.9: Trajectory generated (blue) and the trajectory followed by the UAV (orange) during the real flight

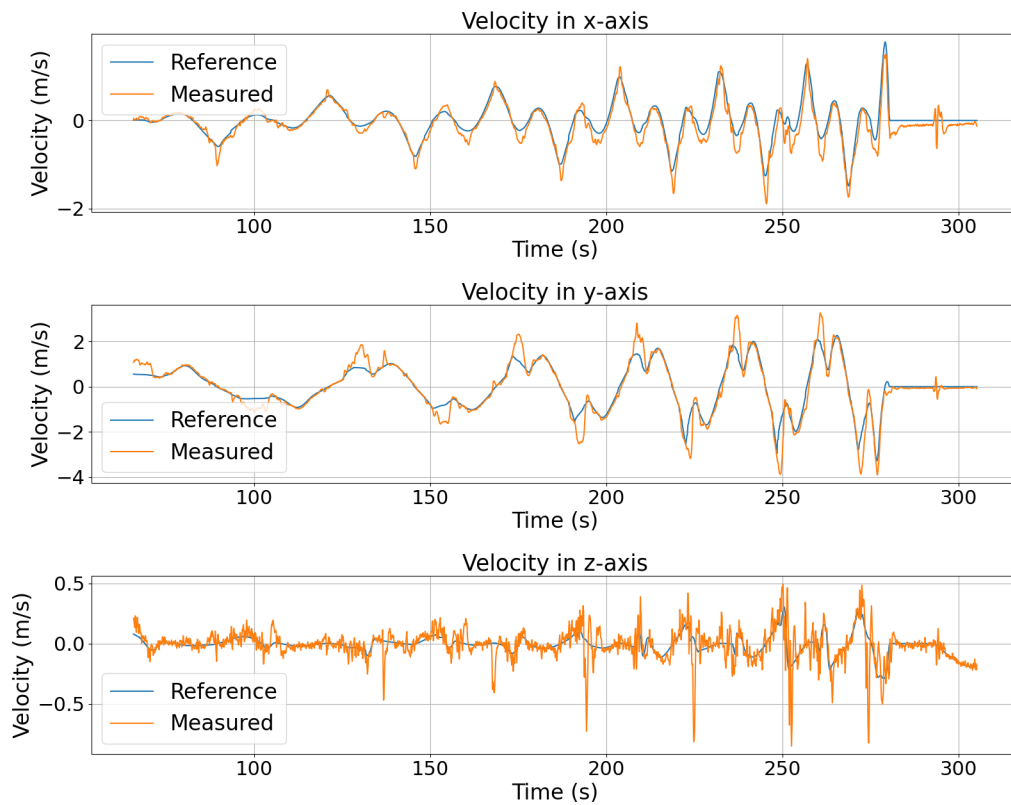


Figure 5.10: Plot per axis of the Trajectory Speed generated during the real flight (blue), and the trajectory speed followed by the UAV (orange), with a continuous increase in speed per lap.

5.5 Discussion

5.5.1 Validation of Fast Trajectory Modification

In the first testbed, the trajectory modified with Local Gaussian Modifier (LGM) successfully reached all waypoints smoothly, generating continuous references for position, velocity, and acceleration, as shown in Figure 5.5 and Figure 5.6. However, modifying the trajectory introduced significant peaks in the drone's acceleration.

Moreover, profiling tests indicated that modifying a trajectory with LGMs is more than three orders of magnitude faster than generating a new polynomial trajectory. Depending on the number of modified points (ranging from 1 to 512), the time reduction is between 10^5 and 10^3 times faster for modifying a short trajectory with six points and between 10^6 and 10^4 times faster for a longer trajectory with 26 points. This difference remains consistent across both a laptop and an onboard computer.

5.5.2 Simulation Flights

In the simulated experiments, it was demonstrated that with sufficiently short computation times—such as those achieved on a laptop—the system is capable of continuously regenerating the trajectory each time it detects a change in the position of a waypoint. However, when the computation time was increased to simulate lower-performance computers, the system was no longer able to generate new trajectories in real time and instead had to rely on LGMs for modifications. This effect became more pronounced at higher speeds, allowing the drone to exceed speeds of 14 m/s while consistently passing through moving gates.

When employing LGMs, the drone successfully navigated through the gates whenever the speed limit was within its actuation capabilities, demonstrating similar performance to that of generating a new trajectory. This similarity was observed up to a speed limit of 15 m/s, where the drone's actual speed closely matched the reference speed. However, when approaching the drone's maximum actuation limits, the reference speeds provided by the LGMs could exceed the maximum acceleration constraints, making them unattainable for the drone. This led to a significant decrease in the success rate.

5.5.3 Real-World Experiments

In real-world experiments, the system successfully employed LGMs to navigate through the gates, reaching peak speeds of up to 4 m/s, with an average speed of 1 m/s throughout the experiment. However, the use of visual perception and localization techniques introduced inaccuracies in estimating both the gate and drone positions, which increased with higher drone speeds.

Unlike in the simulation scenario, the flight area in the real-world experiment was smaller, reducing the available time for the drone to accurately estimate gate positions. This resulted in less precise measurements. Additionally, localization shifts occurred due to fluctuations in the odometry device's state estimation. Specifically, the RealSense T265, which was mounted facing backward, experienced abrupt changes in image capture as the drone passed through the gates.

The inability to achieve higher speeds in real-world experiments was primarily attributed to the accumulation of these inaccuracies, which introduced noise into trajectory generation and tracking, ultimately leading to crashes.

5.6 Summary

In this chapter, we introduced a novel lightweight method for modifying precomputed trajectories, particularly in scenarios with limited reaction time and computational resources. The system demonstrated the capability to achieve high speeds of up to 16 m/s in a simulated dynamic environment, utilizing a polynomial trajectory generator combined with a local modifier. It also exhibited robustness to variations in base trajectory computation time.

This approach holds potential utility for devices with very low computational power as a primary method for generating dynamic trajectories, as well as for high-speed platforms where rapid reaction time is critical.

Although the Local Gaussian Modifier demonstrated advantages in terms of increased reactivity in onboard computing scenarios, they also exhibited certain limitations. For example, the unbounded influence of the Gaussian function could cause nearby points to be affected, unintentionally modifying the trajectory beyond the intended point. Additionally, adapting the generated trajectory to adhere to the physical limits of the platform could result in exceeding speed and acceleration constraints. Addressing these challenges may involve exploring alternative modifier functions that account for these limitations and adjust the trajectory accordingly.

Chapter 6

Distributed Knowledge-Centric Collaborative SLAM

6.1 Introduction

The development of Tier IV systems requires that each agent in the system generates its own *Understanding* of the situation while also ensuring that the knowledge generated by different agents is fused into a *World Model*, which exploits this collective knowledge to make optimal decisions. In this chapter, we focus on how the knowledge of a single agent can be combined with that obtained by the rest of the agents to generate *Collaborative Understanding* in a Collaborative Simultaneous Localization and Mapping algorithm. Here, we do not cover how this emergent *Collaborative Awareness* can be leveraged to enhance decision-making processes.

Collaborative Simultaneous Localization and Mapping (CSLAM) is a technology that enables multi-robot systems to collaboratively create a map of their environment while simultaneously determining their own positions within it. CSLAM exemplifies how *Collective Understanding* enhances the knowledge of a multi-robot system. In SLAM, each agent maintains models of the landmarks in the environment, which are mapped in real time, generating knowledge about its surroundings.

As stated in Section 2.4, most CSLAM techniques, such as (Lajoie et al., 2020), (Zhong et al., 2023), and (Huang et al., 2022), rely on transmitting low-level features like keyframe descriptors between robots to construct and merge maps. However, this often leads to incorrect loop closures (Azzam et al., 2020), as these methods lack a deeper understanding of the environment.

To address this, recent single-robot SLAM approaches incorporate metric-semantic-relational information. *Hydra* (Hughes et al., 2022) leverages 3D scene graphs for loop closure, while *S-Graphs+* (Bavle et al., 2022, 2023) integrates a situational graph with the underlying SLAM graph, improving accuracy and robustness.

Building on this, *Hydra-Multi* (Chang et al., 2023) and *CURB-SG* (Greve et al., 2023) ap-

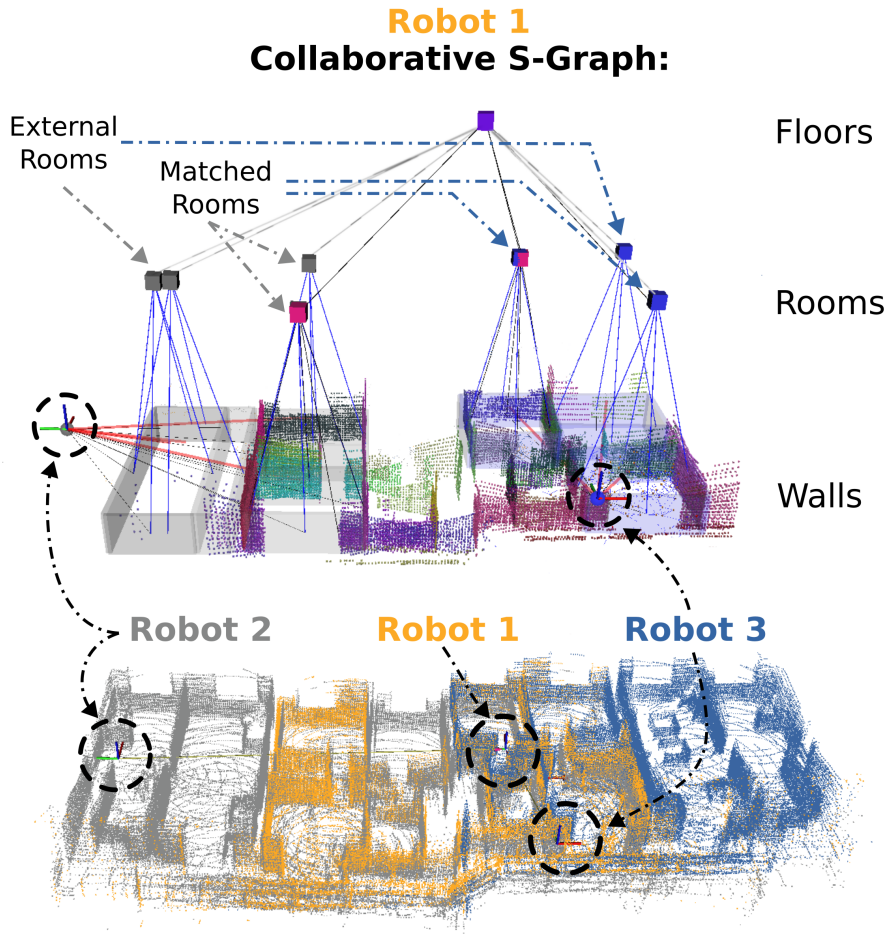


Figure 6.1: Collaborative graph generated by three robots mapping a building floor (orange, gray, and blue for robots 1, 2, and 3 respectively). The collaborative graph is generated from the point of view of Robot 1. Semantic information gathered by the other robots, such as rooms or walls appears in transparent colors in the graph layers. The origin of the coordinate systems of the other robots appears surrounded by a circle. Below is the result of merging the point clouds obtained from each of the robots. This point cloud is displayed for visualization purposes and is not utilized in inter-robot communication.

ply 3D scene graphs to CSLAM in a centralized manner, using semantic-relational knowledge to enhance inter-robot loop closures. However, these methods remain centralized and explicitly constrain robot poses rather than the semantic entities detected by multiple agents.

In this chapter, we present *Multi S-Graphs*, a LiDAR-based distributed SLAM system designed for structured environments. It leverages multi-layered Situational Graphs (S-Graphs), a single optimizable factor graph that integrates a pose graph (i.e., keyframes) with a 3D scene graph containing semantic-relational concepts. This approach enables the generation of precise environmental maps and enhances robot localization while minimizing inter-robot data exchange.

Although the system has been tested and exemplified with legged robots, the algorithm is agnostic to the type of platform, so it could also be deployed with aerial vehicles equipped with a LiDAR.

6.2 System Overview

Multi S-Graphs is a fully decentralized system in which each robot independently runs the *Multi S-Graphs* framework, as illustrated in Fig. 6.2. At the core of our system architecture is the Situational Graph (S-Graph), a multi-layered optimizable factor graph that unifies a pose graph with a scene graph enriched with semantic-relational concepts. This graph encodes essential knowledge for extracting key information required for inter-robot communication and integrates data from other agents, forming a Multi-Robot Situational Graph.

Robots share information via a common communication channel, enabling efficient data exchange with minimal overhead. Our system also addresses the multi-robot kidnapped problem, where each robot initially lacks knowledge of its own position or the locations of other robots. To overcome this, each robot constructs an environmental map relative to its starting position and localizes other agents accordingly.

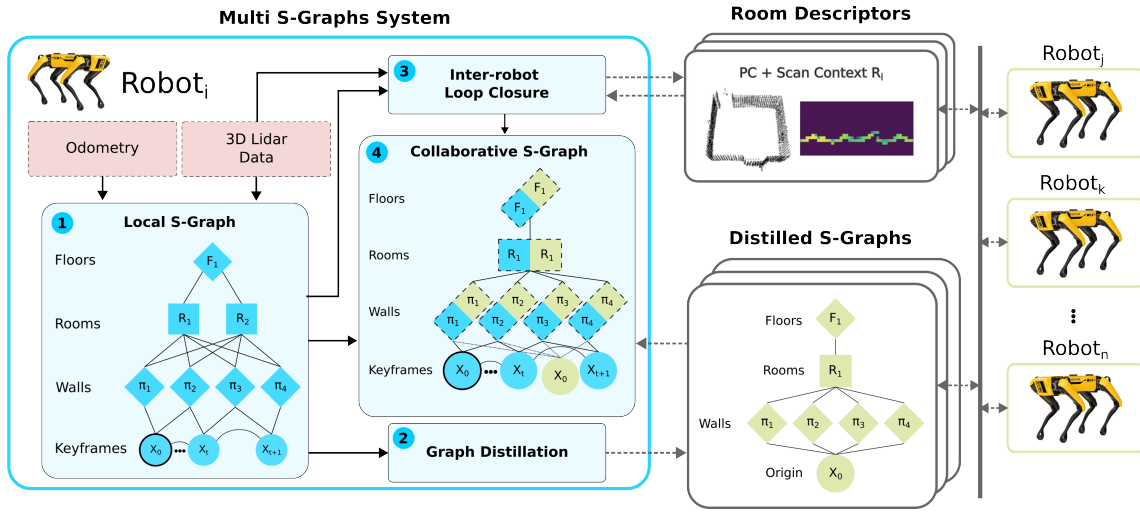


Figure 6.2: *Multi S-Graph* architecture viewed from $Robot_i$ perspective. Odometry measures and 3D Lidar data are the inputs that the robotic platform provides. Each robot executes its own Local S-Graph system and generates the Room Descriptors and the Distilled S-Graphs that are exchanged with the rest of the robots. Messages exchanged between robots are represented in gray. From the point of view of the $Robot_i$ the rest of the agents provide the Distilled graphs and the Room descriptors that are used to generate the Collaborative S-Graph. Information provided by the rest of the robots appears in green color.

Our proposed system is composed of four main modules:

1. Local S-Graph Generation (Section 6.3). To generate the local S-Graph for each robot, we use *S-Graphs+* (Bavle et al., 2023) which runs onboard each one using the odometry and 3D LiDAR measurements of the robot. It also estimates the position and trajectory followed by the robot while generating it. Like *S-Graphs+*, our system can work with any external odometry source.

2. S-Graph Distillation (Section 6.4). We marginalize the local S-graph generated by each robot to only transmit semantic-relational information and not the low-level information stored on each keyframe. Each distilled graph includes a reference node that represents the origin of the local coordinated system of each robot. This distilled S-graph is transmitted to the rest of the robots.

3. Room-Based Inter-Robot Loop Closure (Section 6.5). Relying only on the generated semantic graph for loop closure can generate alignment errors in the presence of symmetry in the environment. In order to combine the raw information of the point clouds and take advantage of the high-level semantic information that each room contains, we generate a hybrid descriptor for

each room, a *Room Descriptor*. This module is in charge of two tasks: generating the descriptors and finding appropriate room matches.

4. Collaborative S-Graph Generation and Optimization (Section 6.6). This module integrates the Distilled S-Graphs received from the other agents with its own local S-Graph. The different graphs are connected together with the loop closures generated by the previous module. The connected graphs are globally optimized to generate the final collaborative situational graph.

6.3 Local Situational Graph

In this approach, the knowledge of each robot is encapsulated within a situational graph whose entities have been designed to represent the different elements that make up the layout of a building. In this work, we rely on the *S-Graphs+* (Bavle et al., 2023) algorithm for generating this Situational Graph, based on the LiDAR and odometry measurements of the robots and estimating the localization of the robot. Figure 6.3, shows the *S-Graphs+* modules and the hierarchical structure of the Situational Graphs generated.

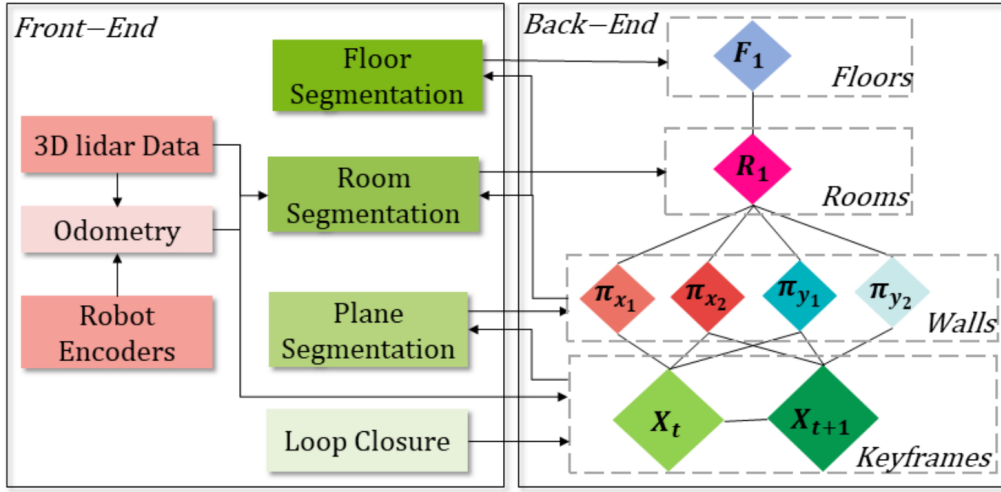


Figure 6.3: *S-Graphs+* overview. The 3D LiDAR measurements and robot odometry, are pre-filtered and processed to extract wall planes, rooms, floors, and loop closures. Figure from (Bavle et al., 2023).

These S-Graphs are four-layered, optimizable hierarchical graphs that contain relevant information about the structure of the building:

1. **Keyframes Layer.** This layer consists of robot poses represented as SE(3) nodes in the agent map frame A_i , with pairwise odometry measurements constraining them.
2. **Walls Layer.** This layer consists of planar wall surfaces extracted from 3D LiDAR measurements, represented using minimal plane parameterization. The planes observed by their respective keyframes are constrained using pose-plane constraints.
3. **Rooms Layer.** This layer consists of two-wall rooms (corridors) or four-wall rooms, each constraining either two or four detected walls, respectively.
4. **Floors Layer.** This layer consists of a floor node positioned at the center of the current floor.

We use four-wall rooms for S-Graph distillation (Section 6.4) and room-based loop closure (Section 6.5) because their boundaries and dimensions can be easily extracted from the walls that constitute them.

6.4 Situational Graph Distillation

To reduce the amount of data that must be exchanged between robots, we extract the most meaningful information from the local S-Graph. This graph encodes higher-level layers of the S-Graph and locates them with respect to the origin node of each robot, enabling a minimal yet meaningful exchange of information about the structure of the building.

The Distilled S-Graph is organized into the following layers:

1. **Origin Layer.** The origin node of each Distilled S-Graph represents the origin of the coordinate system of each robot. All other elements in the graph are positioned relative to this node.
2. **Walls Layer.** This layer contains the estimated positions and covariances of walls extracted from the S-Graph of each agent. The edges connecting the planes to their corresponding keyframe nodes are replaced by an edge to the origin vertex.
3. **Rooms Layer.** This layer contains the estimated positions of four-wall room vertices detected by each agent, with edges connecting the rooms to their four respective wall nodes.
4. **Floors Layer.** This layer consists of a floor node positioned at the center of the current floor.

In this distillation, we extract all relevant information from the different keyframes while preserving the higher layers of the S-Graph as they are. These nodes contain significantly more meaningful information while requiring less memory for storage.

6.5 Room-Based Loop Closure

In order to connect the Local S-Graph with the external Distilled S-Graphs, a loop closure method is required. To prevent errors when aligning robot positions in highly symmetric environments, such as a corridor with multiple rooms on one side, we cannot rely solely on the structural information stored in the Distilled S-Graphs. More fine-grained information may be needed to break the symmetry and determine whether two rooms are the same or not.

6.5.1 Room Descriptor Generator

We present a hybrid descriptor, the Room Descriptor, which integrates the semantic information of each room with a *Room-Centric Point Cloud*-based descriptor. This approach enhances the robustness and accuracy of the loop closure algorithm with only a minimal increase in data transfer. An example of a Room Descriptor is shown in Fig. 6.4.

To generate these descriptors, we rely on the Scan Context descriptor (Kim and Kim, 2018). This is an egocentric, yaw-invariant point cloud descriptor that has demonstrated state-of-the-art performance due to its simplicity and fast convergence time. However, its main drawback is its sensitivity to linear motion.

To address this issue, we propose generating Scan Contexts based on a Room-Centric Point Cloud, which reduces sensitivity to linear motion and improves descriptor matching.

In this work, we exploit the semantic and hierarchical information from the Local S-Graph to determine the room center and its boundaries. This allows us to extract all keyframes obtained within the room and generate the corresponding Room-Centric Point Cloud.

Each room keyframe R_{k_i} can be expressed as:

$$R_{k_i} = U\{^{R_i}K_j\} \quad ; \quad \forall j \mid K_j \in R_i \quad (6.1)$$

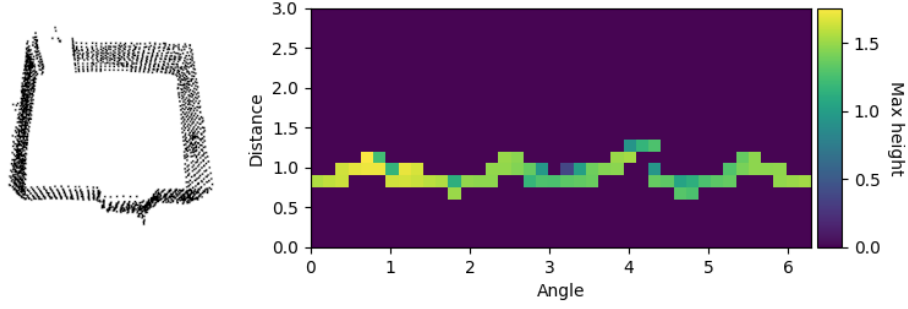


Figure 6.4: Room Descriptor (right) obtained from a Room-Centric Point Cloud (left) using a Scan Context. The value of each cell in the descriptor represents the maximum height of the points within the corresponding bin.

where ${}^{R_i}K_j$ represents the point cloud associated with the keyframe K_j in the R_i frame (a frame located in the center of the room i).

To obtain the room descriptor R_{d_i} from a room keyframe R_{k_i} we first down-sample R_{k_i} with a fixed voxel size to homogenize the number of points for each room. Finally, the Scan Context descriptor of each room keyframe R_{k_i} is computed to create the room descriptor R_{d_i} :

$$R_{d_i} = SC(\phi(R_{k_i})) \quad (6.2)$$

where $\phi(R_{k_i})$ represents the down-sampled room keyframe cloud, and $SC(\vec{X}) : R^{n \times 3} \rightarrow R^{n_s \times n_r}$ is the Scan Context function obtaining the descriptor for a given point cloud.

6.5.2 Loop Closure Detection

To generate suitable candidates for alignment, we leverage room descriptors to match rooms that serve as connection points between the local S-Graph and external distilled graphs, thereby producing a global alignment of each robot's coordinate system.

Room matching is performed in a two-step process:

1. **Scan Context Matching:** Each robot receives and stores the room descriptors of the other agents. Whenever a new descriptor is created, a search is performed to find a suitable match using the Scan Context distance (Kim and Kim, 2018).
2. **Fine Alignment:** Whenever a Room Scan context match is found between the robot and another agent's room, a prior transformation between both is obtained. However, the precision is quite poor, so the second step consists of obtaining an improved transformation by requesting the Room-Centric point cloud that generated the Scan Context and using a VGICP (Koide et al., 2021) registration algorithm.

If a match is found, the room match with the relative transformation between rooms is sent into the following module (Section 6.6) to connect the different graphs.

Scan Context and ICP match validity depend on the SC_{th} and ICP_{th} thresholds respectively. The first one regulates the volume of point clouds sent over the network, while the second adjusts the match precision. It is crucial for Room Descriptors to remain unique and constant over time for accurate matches, as identical rooms or changing environments can lead to erroneous matches.

6.6 Collaborative Situational Graph

The final module integrates the Local S-Graph with the Distilled S-Graphs, using the loop closures provided by the previous module, generating a Collaborative S-Graph. During this Collaborative S-Graph generation, three different steps can be found:

1. **Disconnected Graphs:** At startup, this module stores information from the Local S-Graph and the Distilled S-Graphs received from other robots. However, since the robots do not know each other's starting positions, they cannot combine the different graphs. At this stage, each external Distilled S-Graph has its origin layer connected to the walls and rooms layer, but this origin layer remains "free" because there are no connections to the robot's Local S-Graph, resulting in multiple disconnected graphs. As new information is received, these graphs are updated accordingly.

2. **Room Matching:** If a room match is found using the room matching step (Section 6.5), the factors between the Local S-Graph and the Distilled S-Graphs are formulated in two steps.

In the first step, a room-to-room factor is created for the matched rooms. For these matched rooms, lower-level wall-to-wall factors are established to enhance the robustness of graph alignment. As shown in Fig. 6.2, graphs are merged at the room, wall, and floor levels. However, since floor nodes connect all rooms at the same height, explicit floor-to-floor factors are not required between the robots' floor nodes.

3. **Collaborative Graph Optimization:** Whenever Local S-Graphs and external Distilled S-Graphs are connected with appropriate factors, a common graph optimization is performed similarly to (Bavle et al., 2022), resulting in an optimized Collaborative S-Graph and, consequently, an improved state estimation of the robot.

After this optimization, the external graphs are aligned with the robot's local coordinate frame. The new information provided by other robots can be integrated into the Collaborative Graph using the same coordinate frame as the local robot to represent the different entities of the external Distilled Graphs.

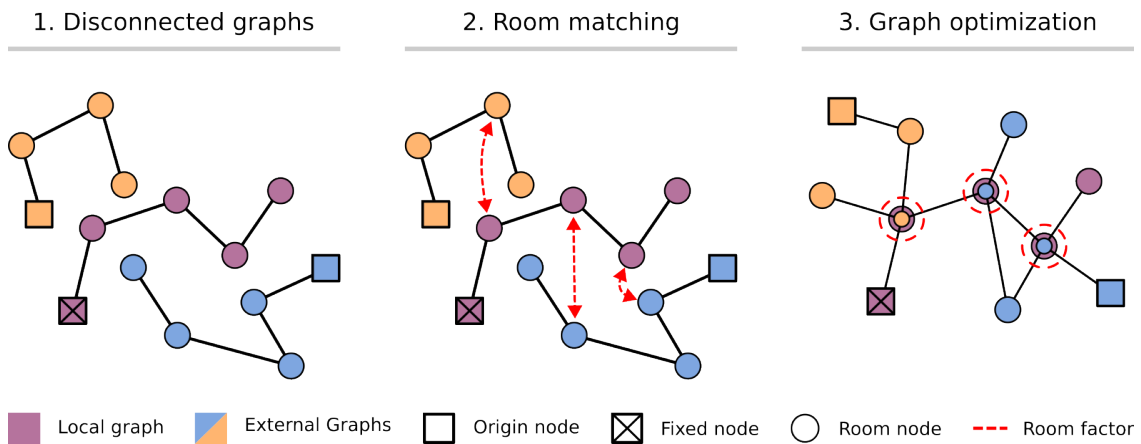


Figure 6.5: Schema of the different stages of Collaborative Graph generation from the perspective of a single agent. The local graph contains the original S-Graph generated by the robot, while the external graphs correspond to the Distilled Graphs shared by other agents. For clarity, only the Room Layer is represented in this figure. The fixed node indicates that its estimation remains unchanged during optimization and represents the robot's origin.

Figure 6.5 represents this process schematically. During the execution of this pipeline, the Local S-Graph generated by *S-Graphs+* is continuously updated, adding new nodes to the graph and incorporating intra-robot loop closures at the four hierarchical levels, as described in (Bavle et al., 2022). Since the Collaborative Graph includes this local graph, these updates are also integrated into it. If a new room match is subsequently found, the two-step factor formulation—enumerated in the second step—is applied, increasing the number of factors between graphs and further refining the optimized graph.

6.7 Experimental Validation

To validate and compare the performance of the proposed solution against state-of-the-art techniques, we have selected the following lidar-based distributed CSLAM algorithms for comparison due to the availability of their code: DCL-SLAM (Zhong et al., 2023), Swarm-SLAM (Lajoie and Beltrame, 2023), and Disco-SLAM (Huang et al., 2022). Additionally, we challenge our algorithm against the centralized CSLAM approach, LAMP 2.0 (Chang et al., 2022), since centralized algorithms typically outperform distributed ones when the number of robots is small.

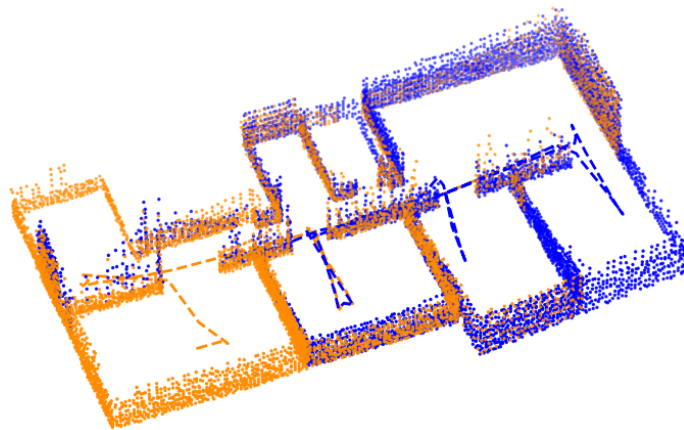
The experiments were conducted using quadruped robots equipped with LiDAR in structured environments within the context of the “multi-robot kidnapped problem,” where the initial positions of the robots were unknown. The only exception was LAMP 2.0, which requires prior knowledge of the robots’ initial positions within the environment.

We conducted experiments in both simulated and real-world scenarios. In simulation, we present results in terms of the Absolute Trajectory Error (ATE) compared to the provided ground truth. In real-world experiments, we report the Root Mean Square Error (RMSE) of the estimated 3D maps relative to the actual 3D map derived from architectural plans. Additionally, we analyzed the data exchanged between agents in each scenario. Furthermore, we performed an ablation study to evaluate the contribution of different hierarchical factors between the local S-Graph and the distilled S-Graphs in terms of accuracy and data transmission.

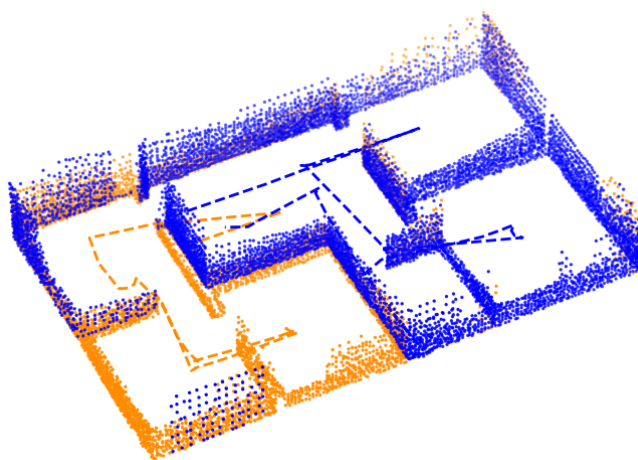
Experimental Setup The experiments were carried out using a laptop computer with an Intel i7-10870H (8 cores, 2.2 GHz) with 32 GB of RAM memory. *Multi S-Graphs* is developed under *S-Graphs+* (Bavle et al., 2023) using ROS 2 for implementing the algorithms. The room matching parameters have been tuned to give a robust performance in the experimental environments, obtaining values of $SC_{th} = 0.35$ and $ICP_{th} = 0.07$.

6.7.1 Simulated Data

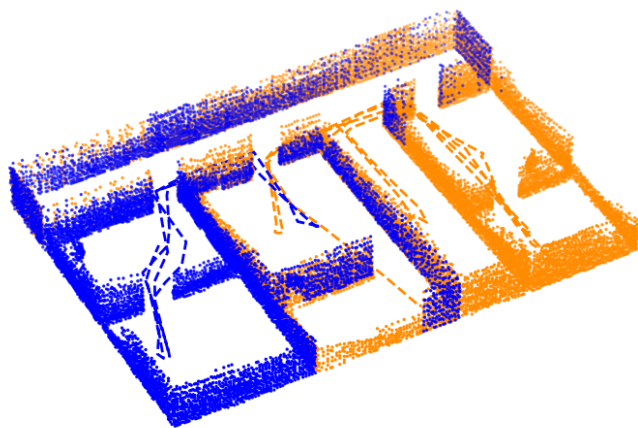
We employed the Gazebo physics simulator to replicate indoor settings for our experiments. In all these simulated scenarios, we relied on LiDAR data for odometry estimation. Our study encompassed a total of three simulated experiments, out of which C1F0 and C1F2 were constructed using 3D mesh representations of real architectural plans, and SE1 is a simulated environment mimicking common indoor layouts with varying room configurations.



(a) C1F0s



(b) C1F2s

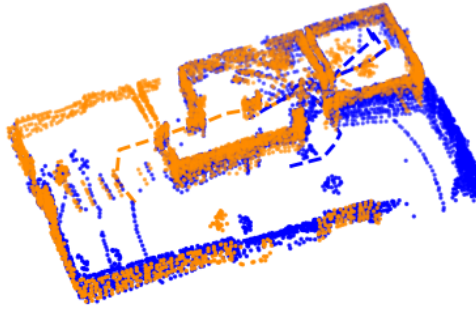


(c) SE1

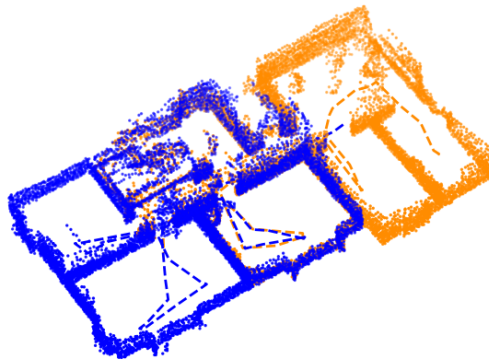
Figure 6.6: Simulated experiments, showing the trajectory and the point cloud map generated for each robot.

6.7.2 Real-world Data.

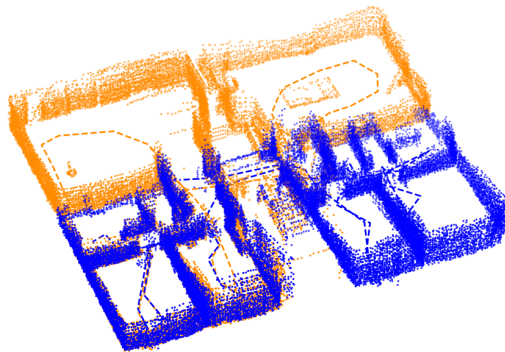
The datasets utilized in our experiments consist of a collection of in-house datasets acquired from a construction site: C1F1, C1F2, C3F1, C3F2, and C2F2. In addition, an experiment with 3 robots was performed on the C2F2 stage enabled by the size of the scenario. The robot used for generating the datasets is a Boston Dynamics Spot with a Velodyne VLP-16 onboard. For these experiments we relied on the robot encoders to generate the odometry data. Unfortunately, our real-world datasets do not include IMU data and consequently, algorithms that rely heavily on IMU, such as DCL-SLAM and Disco-SLAM, could not be evaluated.



(a) C1F1

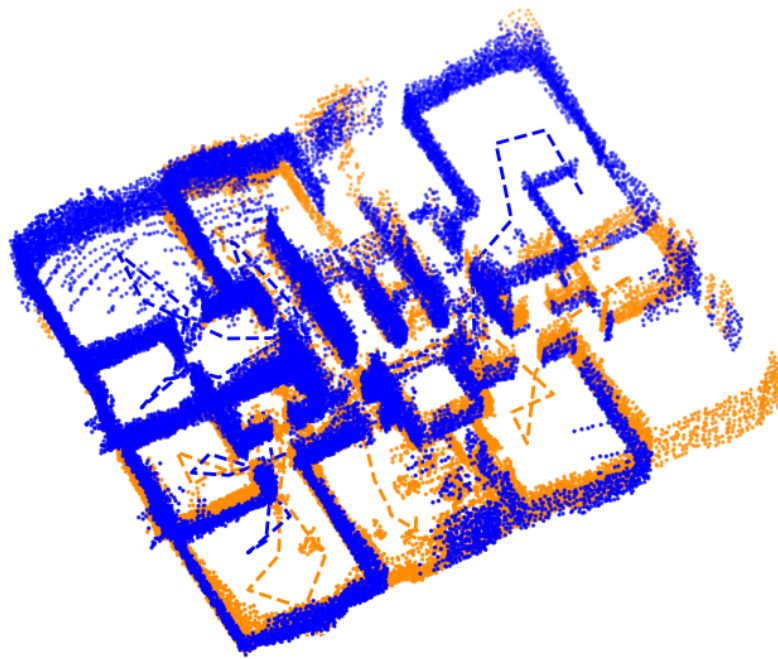


(b) C1F2

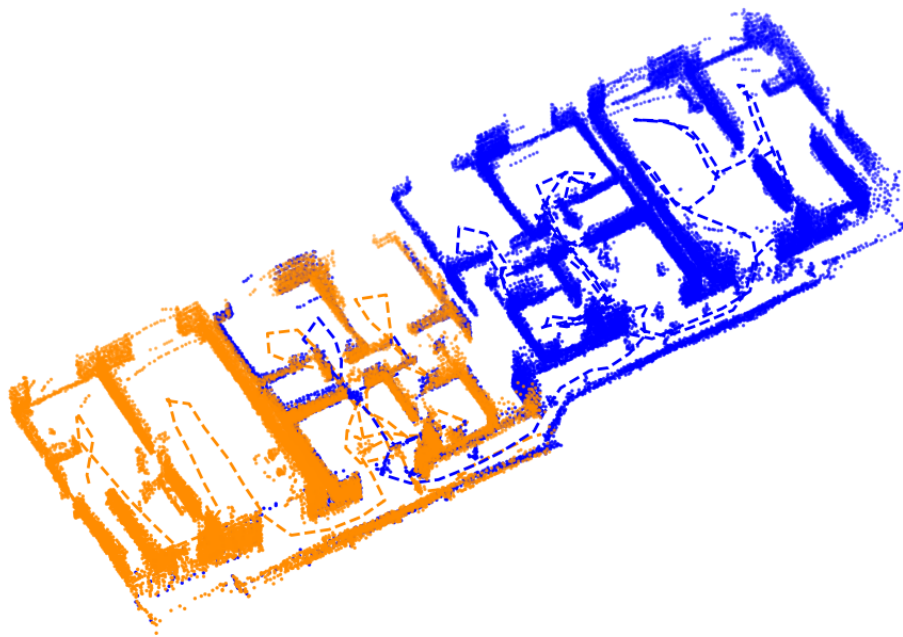


(c) C3F1

Figure 6.7: Real experiment scenarios, showing the trajectory and the point cloud map generated for each robot. In (e) the experiment was run with 3 robots.



(a) C3F2



(b) C2F2

Figure 6.8: Real experiment scenarios, showing the trajectory and the point cloud map generated for each robot. In (e) the experiment was run with 3 robots.

6.7.3 Ablation Study

In this study, we conducted a thorough analysis of the contribution of hierarchical graph factors (Section 6.6) and the influence of Room-Centric point clouds (Section 6.5) on graph alignment precision and the exchanged bandwidth. The ablated stages are described below.

- **Rooms:** Matched room entities between robots are connected using room-to-room edges without fine alignment.
- **Rooms with Fine Alignment:** Room-to-room matching is refined through fine alignment between room point clouds.
- **Rooms and Walls:** Walls are incorporated into the room-to-room connection using wall-to-wall edges without fine alignment.
- **Rooms and Walls with Fine Alignment (Full Algorithm):** Room-to-room matching with wall-to-wall edges and fine alignment.

Notably, when *Fine Alignment* (Subsection 6.5.2) is not performed during loop closure, no point cloud information of the room is exchanged between robots. This significantly reduces the required bandwidth.

6.8 Results and Discussions

6.8.1 Simulated Data

Figure 6.6 presents the qualitative results of *Multi S-Graphs* obtained during the simulated experiments. The outcomes shown in Table 6.1 indicate that the *Multi S-Graph* algorithm achieves a lower Average Trajectory Error (ATE) for the aggregated robot trajectory compared to the selected SotA CSLAM algorithms. Our approach outperforms all other decentralized algorithms, reducing the accuracy error by more than 90% (93.7%, 94.5%, and 98.4% for DCL-SLAM, Disco-SLAM, and Swarm-SLAM, respectively). It also surpasses LAMP 2.0, achieving 72.9% less error, despite LAMP 2.0 requiring precise initial poses of the robots. The lack of features in the simulation environment causes measurement errors to negatively impact optimization. However, our algorithm mitigates this issue by extracting semantic information specific to this type of environment.

Additionally, our approach significantly reduces the exchange of information between agents by 94.50%, 97.20%, and 97.64% for Swarm-SLAM, DCL-SLAM, and Disco-SLAM, respectively, and by 81.3% for LAMP 2.0 (see Table 6.2). This reduction can be attributed to the effective use of semantic information, which minimizes the amount of exchanged data.

ATE [cm]	2 Robots			
Algorithm	C1F0s	C1F2s	SE1	Avg
LAMP 2.0	9.66	8.10	6.27	8.01 ± 1.69
DCL-SLAM	31.15	33.81	38.43	34.46 ± 3.68
Disco-SLAM	35.81	38.45	45.03	39.76 ± 4.75
Swarm-SLAM	119.95	78.08	213.09	137.04 ± 69.11
<i>Multi S-Graphs (ours)</i>	3.64	1.39	1.50	2.17 ± 1.26

Table 6.1: Absolute Trajectory Error (ATE) [cm] for our simulation experiments. The best results are boldfaced.

Data Exchange [MB]	2 Robots		
Algorithm	C1F0s	C1F2s	SE1
LAMP 2.0	3.23	3.69	7.68
DCL-SLAM	25.20	24.89	41.54
Disco-SLAM	30.28	32.36	45.69
Swarm-SLAM	11.31	14.40	20.80
<i>Multi S-Graphs (ours)</i>	0.53	0.95	1.08

Table 6.2: Data exchanged [MB] between agents in simulation experiments. The best results are boldfaced.

6.8.2 Real-world Data

Table 6.3 presents the outcomes of the real-world experiments. The results from real datasets closely resemble those from the simulations, with our algorithm achieving a 78.81% improvement in map matching accuracy over Swarm-SLAM and a 45.3% improvement over LAMP 2.0. In this case, although the accuracy of the extracted semantic features slightly decreases due to additional elements present in real environments, the use of semantic-relational information from room-wall structures and their point clouds enables our approach to outperform the other algorithms. Our method achieves an average reconstruction error of 20.5 cm, with slight variations across different scenarios.

Point Cloud RMSE [cm]	2 Robots						3 Robots
Algorithm	C1F1	C1F2	C2F2	C3F1	C3F2	Avg	C2F2
LAMP 2.0	61.49	23.61	31.89	30.46	40.05	37.50 ± 14.63	35.34
Swarm-SLAM	75.45	94.60	147.22	40.36	36.42	78.81 ± 45.32	158.60
<i>Multi S-Graphs (ours)</i>	27.03	15.12	17.73	21.04	21.58	20.18 ± 4.49	20.86

Table 6.3: Point cloud RMSE [cm] for our in-house real experiments. The best results are boldfaced.

Data Exchange [MB]	2 Robots					3 Robots
Algorithm	C1F1	C1F2	C2F2	C3F1	C3F2	C2F2
LAMP 2.0	1.89	3.72	19.03	8.18	8.59	20.48
Swarm-SLAM	4.31	8.80	44.84	17.53	17.65	48.79
<i>Multi S-Graphs (ours)</i>	0.26	0.82	3.50	1.12	0.66	5.73

Table 6.4: Data exchanged [MB] between agents in real experiments. The best results are boldfaced.

6.8.3 Ablation Study

The results of this study are collected in Table 6.5. We can see that the utilization of hierarchical optimization, which establishes relationships between the walls of two rooms among agents (*Rooms and Planes*), means a significant improvement of accuracy, reducing the ATE by 25.6% with an increase in the amount of data by 77%. On the other hand, the use of Fine alignment improves the performance by 73.6% and 25.6% over the *Rooms* and *Rooms and Walls* configurations, respectively, with a 40 times higher data exchange requirement. In general, fine alignment techniques perform better compared to their non-fine alignment counterparts, primarily due to the limited angular resolution of the Scan Context, which is insufficient for precise alignment (see Section 6.5). Furthermore, the use of wall-to-wall factors contributes to alignment improvement,

consequently significantly enhancing overall performance. Consequently, the best performance is obtained by combining the semantic information extracted from the walls with fine alignment using the Room Centric point clouds, *Rooms and Walls with Fine alignment*, improving the performance by up to 74.7%. However, due to the larger amount of data exchanged required, when communication channels are limited, *Rooms and Walls*, which only use semantic information and avoid sending point cloud messages, may be an interesting solution without a significant decrease in accuracy.

	C1F0s		C1F2s		SE1	
Algorithm	ATE [cm]	Data Shared [MB]	ATE [cm]	Data Shared [MB]	ATE [cm]	Data Shared [MB]
<i>Rooms</i>	9.78	12.88	1.99	18.90	8.06	26.46
<i>Rooms with FA</i>	3.78	524.32	1.42	946.91	1.62	1069.03
<i>Rooms & Walls</i>	5.33	22.81	1.72	27.38	1.73	37.85
<i>Rooms & Walls with FA</i>	3.64	537.53	1.39	955.39	1.50	1080.41

Table 6.5: Ablation study of the ATE and data shared [MB] for simulated datasets with 2 robots, FA means Fine Alignment. The best results are boldfaced.

In Fig. 6.9 we can see on a logarithmic scale how the graph information (composed mainly of walls and room graph nodes) is negligible compared to the room point clouds. This observation highlights that point clouds lead to a substantial increase in transmitted data. Although the growth in the graph data during map exploration does not result in a significant increase in data volume, the cost associated with point clouds significantly escalates.

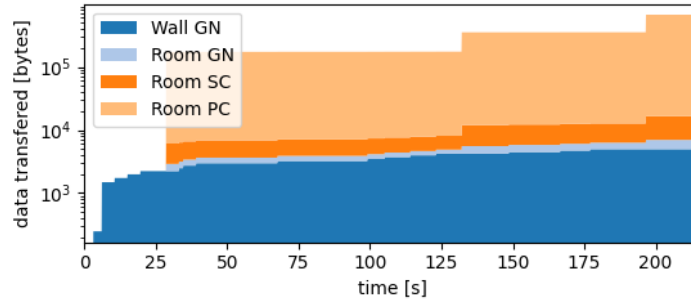


Figure 6.9: Accumulated data shared between agents, consisting of Walls Nodes (Wall GN) and Room Nodes (Room GN) for the Semantic Graph, and the Room Scan Contexts (Room SC) and Room Centric point clouds (Room PC).

6.9 Summary

In this chapter, a distributed CSLAM system is presented, leveraging the semantic-relational features extracted by the S-Graphs (Bavle et al., 2022), (Bavle et al., 2023) to filter and reduce the amount of data that must be transmitted between robots while outperforming state-of-the-art mapping and positioning accuracy in structured environments.

We propose a novel Room Descriptor that fuses room structural elements with fine-grained point cloud information, reducing the number of candidates for loop closure without requiring object identification in the scene or the exchange of keyframes between agents.

We tested our algorithm for a map generation task in multiple scenarios, both simulated and real, outperforming other state-of-the-art LiDAR-based CSLAM algorithms in terms of precision and bandwidth. Our approach reduced trajectory and map reconstruction errors by more than 90% while saving 95% of the data exchanged between robots.

The main drawback is the dependence on semantic-relational structures, defined in our case as four-walled rooms in building-like environments, which are not always easy to extract. Room matching relies on these structures and the detection of different static features within them, so in feature-sparse or dynamic environments, room matching may not function properly.

For future work, a more generic method for detecting different types of rooms and aligning Situational Graphs should be developed to increase the variety of scenarios in which *Multi S-Graphs* can be used.

