UNIVERSIDAD POLITÉCNICA DE MADRID

Autonomous Navigation Behaviors for an Aerial Robotics Software Framework

by

Guillermo Echegoyen Blanco

A master thesis submitted in partial fulfillment for the masters degree of Artificial Intelligence

in the
Escuela Técnica Superior de Ingenieros Informáticos
Departamento de Inteligencia Artificial

July 2018

Declaration of Authorship

I, AUTHOR NAME, declare that this thesis titled, 'THESIS TITLE' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Write a funny quote here."

If the quote is taken from someone, their name goes here

UNIVERSIDAD POLITÉCNICA DE MADRID

Abstract

Escuela Técnica Superior de Ingenieros Informáticos
Departamento de Inteligencia Artificial

Master Degree in Artificial Intelligence

by Guillermo Echegoyen Blanco

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor…

Contents

Declaration of Authorship i

Abstract iii

Acknowledgements iv

List of Figures vii

List of Tables viii

1 Introduction 1
1.1 Context . . . 1
1.2 Motivation . . . 2
1.3 General Objective . . . 2
1.4 Specific Objectives . . . 2
1.5 Overview . . . 3

2 Problem Description 4
2.1 The Aerostack Framework . . . 4
2.2 Requirements . . . 6
2.3 Details of the New Features . . . 6

3 State of the Art 7
3.1 Localization . . . 8
3.1.1 Outdoor localization . . . 8
3.1.2 Indoor localization . . . 9
3.1.3 SLAM . . . 10
3.1.3.1 Extended Kalman Filters SLAM . . . 10
3.1.3.2 Particle Filters . . . 12
3.1.3.3 Graph Optimization SLAM . . . 13
3.1.3.4 Lidar SLAM . . . 13
3.1.3.5 Visual SLAM . . . 14
3.2 Planning . . . 14

4 Implemented Modules 16

4.1 Requirements . . . 16
4.1.1 Specification . . . 17
4.1.2 Integration . . . 18
4.2 Behavior Self Localize and Map by Lidar . . . 20
4.3 Navigation Interface . . . 22
4.3.1 Behavior Go to Point in Occupancy Grid . . . 24
4.3.2 Behavior Follow Path in Occupancy Grid . . . 26
4.3.3 Behavior Generate Path in Occupancy Grid . . . 27
4.3.4 Path Planner in Occupancy Grid . . . 28

5 Validation **29**
5.1 Validation Tests . . . 29
5.2 Simulation . . . 30
5.3 Real Flight . . . 31
5.4 Testing Mission . . . 31
5.5 Experiments . . . 32
5.5.1 Behavior Generate Path in Occupancy Grid . . . 32
5.5.2 Behavior Follow Path in Occupancy Grid . . . 33
5.5.3 Behavior Go To Point in Occupancy Grid . . . 33
5.5.4 Simulation . . . 34
5.5.5 Real Flight . . . 35
5.6 Experimental Results . . . 36
5.7 Discussion . . . 39

6 Conclusions & Further Research **41**
6.1 Conclusions . . . 41
6.2 Further Research . . . 42

Bibliography **43**

A Appendix A **1**
A.1 Go to Point test mission . . . 1
A.2 Generate Path test mission . . . 3
A.3 Follow Path test mission . . . 5

List of Figures

2.1 The Aerostack architecture 5

3.1 The area enclosed by the three intersections ca be used to localize univocally. The three simple intersections formed by pairs of circumferences are always outside of the Earth. As more satellites are included, precision increases. Taken from [23]. 9
3.2 EKF applied to the SLAM problem. Dotted line: robot's path. Shaded ellipses: position estimates. Small dots: Unkown location landmarks. White ellipses: Landmarks' position estimates. In (d) the robot senses the first landmark, anchoring the rest of the estimates, reducing uncertainty. Taken from [18] 11

4.1 Behavior self localize and map by lidar architecture. *Hector mapping* is the SLAM module from [8]. *Drone Robot Localization* does the EKF. *Self Localization Selector* gives the localization based on the selected technique. 21
4.2 Self Localize and Map behavior catalog specification. Required processes and timeout values are specified. 22
4.3 Behavior Go to Point architecture. Move base is the planner 24
4.4 Go to Point behavior catalog specification. Required processes and timeout values are specified. 25
4.5 Behavior follow path in occupancy grid architecture 26
4.6 Follow Path in Occupancy Grid behavior catalog specification. Required processes and timeout values are specified. 27
4.7 Behavior generate path in occupancy grid architecture 27
4.8 Generate Path behavior catalog specification. Required processes and timeout values are specified. 28

5.1 Simulated boiler. 57 meters tall by 16 meters (square section). The UAV is enclosed in the yellow circle. 34
5.2 Gazebo and Rviz. Simulation visualization during the execution of the mission. Left: Gazebo World, top-right: Rviz lidar measures, bottom-right: Hummingbird front camera 35

List of Tables

5.1 Simulation computer specifications 34

5.2 Simulation computer specifications 34

5.3 Dimensions of the sports centre used for real flight tests. School of Industrial Engineers 35

5.4 Onboard computer specification 35

5.5 Results from *generate path* behavior, run accross 10 tests. Correct executions, averaged execution time for all 6 points and total execution of the mission. Measured in minutes 36

5.6 Results from *Follow Path* behavior, run accross 10 tests. Correct executions, averaged execution time for all 6 points and total execution of the mission. Measured in minutes 37

5.7 Results from *Go to Point* behavior, run accross 10 tests. Correct executions, averaged execution time for all 6 points and total execution of the mission. Measured in minutes 38

5.8 Results from *Go to Point* behavior in real flight, run accross 3 tests. Correct executions, averaged execution time for all 6 points and total execution of the mission. Measured in minutes 39

For/Dedicated to/To my…

Chapter 1

Introduction

The following work proposes a navigation system based on state of the art algorithms. As a testbed for these algorithms we will employ the Aerostack platform, which is an extensively tested general purpose aerial robotics software framework already available in the research community. We propose the implementation of new behaviors, which are the high level abstraction modules embraced by the framework.

We propose theoretic hypotheses for a modular navigation system and provide empiric proofs through simulated and real flight missions to support them. Accross this document, the reader can grasp an idea of the state of planning and localization in aerial robotic applications and watch in action an experimentaly tested solution.

1.1 Context

Aerostack is a framework for aerial robots, aimed at giving flight autonomy to some extent. It features a modular approach for the construction of behaviors that can be used to develop complex flights and automatic handling for certain situations such as battery level or hardware conditions. It is the frame for the following work, which adds more autonomy through the integration of a lidar based localization and mapping system and a global planner, composing a navigation interface for UAVs. This provides both a novel localization and planning techniques for the framework, extending the current system.

1.2 Motivation

So far, there exists only one simple geometry planner and Aruco or odometry based localization techniques inside the Aerostack. For indoor environments, this system compels the need for environment preparation, the Arucos must be placed beforehand in well known localizations that must be hardcoded in the robot map. In this sense, there exists a need for a more robust, preparation-free localization system and accompanying planner. This work provides such improvements with the introduction of a lidar-based navigation system and a global planner.

1.3 General Objective

Through this thesis we provide a way for the Aerostack enabled UAVs to map an indoor environment and localize inside it using lidar sensors and then outline a plan to traverse it in a secure manner. We believe that lidar sensors in conjunction with planning algorithms can be used to build a real time navigation system that can be used with an existent platform to enlarge the capabilities of aerial robotics. Therefore, the general objectives of this work can be described as:

1. Test whether state of the art algorithms can be employed to construct a real time, autonomous navigation interface that can be used with the existent robotic platforms.

1.4 Specific Objectives

Using the available literature as the basement for the developed modules, we aim at the following specific objectives:

1. Contribute to the research community with our work through the implementation and integration of the proposed algorithms inside the Aerostack framework.
2. Measure whether the proposed algorithms comply with the proposed constraints and are suitable for real time robotic applications.
3. Test the implemented modules both in real flight and simulation to ensure that the proposed approach meets the imposed requirements.

1.5 Overview

[ToDo := Review when everything is finished]

This dissertation is organized as follows: ...

Chapter 2

Problem Description

In the present chapter the problem is presented, along with the proposal of new features to implement. It is structured as follows: Section 2.1 introduces the Aerostack framework, section 2.2 presents the context of the problem and the requirements a replacement should have and section 2.3 closes the chapter giving an overview on the improvements presented.

2.1 The Aerostack Framework

Aerostack is a software framework that helps developers design and build the control architecture of aerial robotic systems, integrating multiple heterogeneous computational solutions (i.e.: computer vision algorithms, motion controllers, self localization and mapping methods, planning algorithms, etc.).

Aerostack is useful for building autonomous aerial systems in complex and dynamic environments and it is also a useful research tool for aerial robotics to test new algorithms and architectures.

It was created to be available for communities of researchers and developers and it is currently an active open-source project. It provides some low level components as well as coordination processes and some planners. Figure 2.1 shows the general architecture of the framework. It is fully described in [16] and [17] and publicly avaiable in [22].

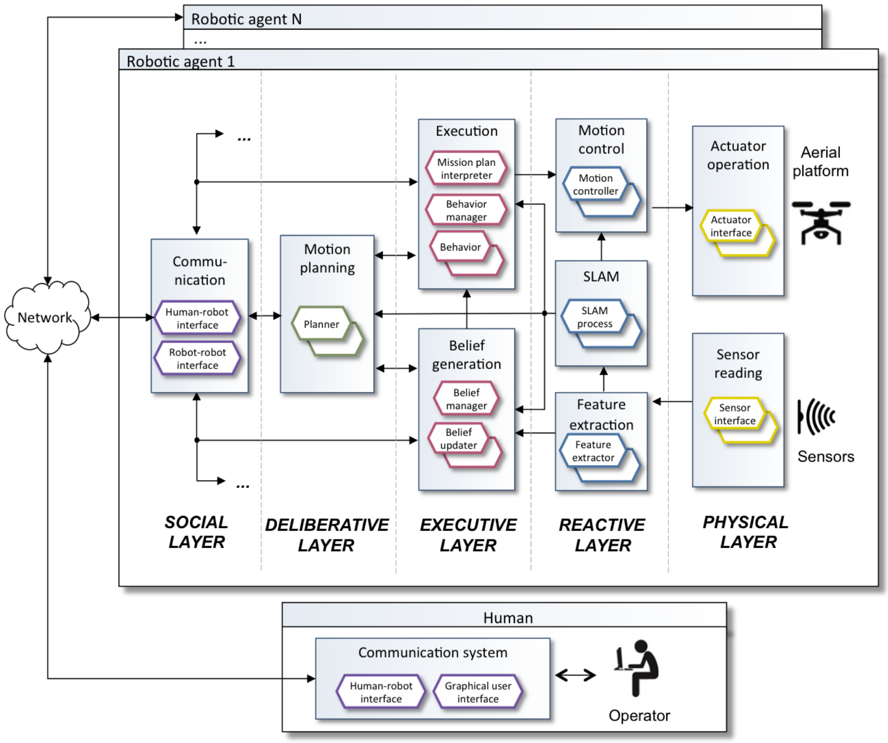

FIGURE 2.1: The Aerostack architecture

This work will be specifically focused on the executive layer, the one pertaining the highest level of abstraction. The executive layer comprises the following components:

- Behavior coordinator: The process in charge of managing and monitoring the correct execution of behaviors, checks the necessary constraints and conditions to be met for a behavior to be executed. It is the central key to the whole behavioral system.
- Belief generation: This is the subsystem in charge of the store and retrieval of symbolic information, usually gathered from high level processors of sensory input, such as localization modules and behaviors.

When a behavior activation request arrives at the coordinator, it first checks the constraints of the requested behavior, if it conflicts with any other behavior, that behavior is deactivated. The next step is check the necessary processes, those are processes needed for the correct execution of the given behavior (more on this in the implementation chapter 4). The coordinator activates those processes and then activates the behavior.

This a flexible enough approach to specify pre and post conditions for a behavior to be executed, providing with the ability to check, at runtime, that the desired environment for a high level algorithm is ensured. This approach opens the door for programming and testing algorithms that enrich the capabilities of a robot in a robust and expressive manner.

2.2 Requirements

As of the second version of Aerostack, the only localization techniques available are: visual localization based on the recognition of a special type of marker called Aruco ([15]) and odometry based localization. Aruco markers were first used for augmented reality applications, it is a fast and reliable technique to estimate the pose of the camera capturing the image. Although this system works fine for many applications, it imposes the need of preparing the environment, placing these markers in a very precise way and annotating it's exact position before the experiments. While this might not be a problem in an augmented reality like scenario, when it comes to live localization in unknown environment it becomes useless. Hence, a new system for localization is required.

Along with the aforementioned localization technique comes the navigator which coordinates with a 2D geometric planner to accomplish the mission at hand. As the localization technique is to be changed, leading to a new way to perceive the environment, a new navigator and planner will also be necessary.

2.3 Details of the New Features

A lidar is going to be used as the main source for localization. The module in charge of this part is already present in the Aerostack framework, but it is not being used. Based on lidar input an occupancy grid is built. This occupancy grid is then used by the navigator, along with the planner instructions to build a motion plan to execute. Therefore, we will build:

- A module for localization and mapping based on lidar
- A module to do planning based on occupancy grids.
- A module capable of use the map and planning information to move the robot.

In the next chapter, the state of the arts algorithms chosen will be explained in depth.

Chapter 3

State of the Art

This chapter will review various methods used for localization, mapping and planning, which are the core for any navigation technique. It starts by describing the problems that arise both in localization and mapping to continue with some of the most prolific solutions found to these problems. Mapping techniques will be described briefly at the end of the chapter.

Localization is referred to all the techniques used to find the position in coordinates of an agent or object inside the world, relative to a reference frame. As far as its absolute coordinates are known, anything can be used as reference frame, it is used as the coordinates system's centre. In an outdoors environment, the Earth could be the reference frame and the robot's coordinates can be acquired with satellite systems, giving an absolute point inside the three dimensional space. It is desirable for these coordinates to be in a format that a computer can handle efficiently, tipically as two or three floating point numbers (although integer numbers are used sometimes too), depending on the number of dimensions used to represent the space. To save computational effort, the z axis (height) could be unused in a wheeled robot.

Moving a robot avoiding possible obstacles through the space is tricky in itself, obstacles must be detected and handled correctly, moving objects can appear in the way, and so on. This alone does not provide any intelligence nor it helps planning, to aid in planning and moving smartly in the space, a map can be constructed while the localization is happening. The term mapping covers all the algorithms used to construct a map combining the data acquired from the many input sources a robot can have. Mapping opens the door for smart planning, along with many more advantages. A classic example is finding cycles in planned paths.

In the current field of study, planning refers to the set of algorithms dedicated to map a target goal (be it a coordinate in a map or a robot pose) to the necessary set of motions required to reach the goal. In our case study it will consist of composing a valid trajectory to reach from the current robot position to the target position given the map and the robot's location.

3.1 Localization

Localization techniques are divided into two groups: Outdoor and indoor techniques. The distinction comes from the fact that satellite systems signals cannot go through walls. This fact has led to a whole new set of technologies and techniques that are able to localize in environments without an absolute reference of the world.

This section is organized as follows: First outdoor localization will be analyzed, with a brief review of it's core components, then various indoor localization techniques will be exposed, focusing on lidar based ones.

3.1.1 Outdoor localization

By now, the most robust, reliable solution for outdoor localization is based on combining different sources of data. The mobile platform has drawn great attention over the past few years, pushing some of the largest companies in a shared effort to improve localization services while minimizing the impact over the battery's performance.

Satellite systems localization in mobile applications have certain drawbacks: The most obvious one is that acquiring and processing the signal wastes power, but also that in the case of civilian systems (such as GPS) the localization has a precision of 10 meters for security reasons. Figure 3.1 shows the localization dissambiguation through the trilateration technique with three sources of signal employed in any satellite system.

To aid both problems two new localization techniques were implemented:

1. GSM Localization. As every GSM antenna has a well known location stored in a database, one can localize trilaterating the near GSM antennas. Obviosly, this method is subject to GSM signal coverage.

2. WiFi Localization. This powerful method can serve both as an indoor and outdoor localization technique (see sect. 3.1.2). When a smarphone detects a WiFi hotspot, it sends it's BSSID along with the GPS coordinates if available to a centralized

server. As this database grows the localization precision improves and every user can take advantage from it. This is scpecially useful in urban areas where stallite signal is poor or intermitent and improves as more and more users log information.

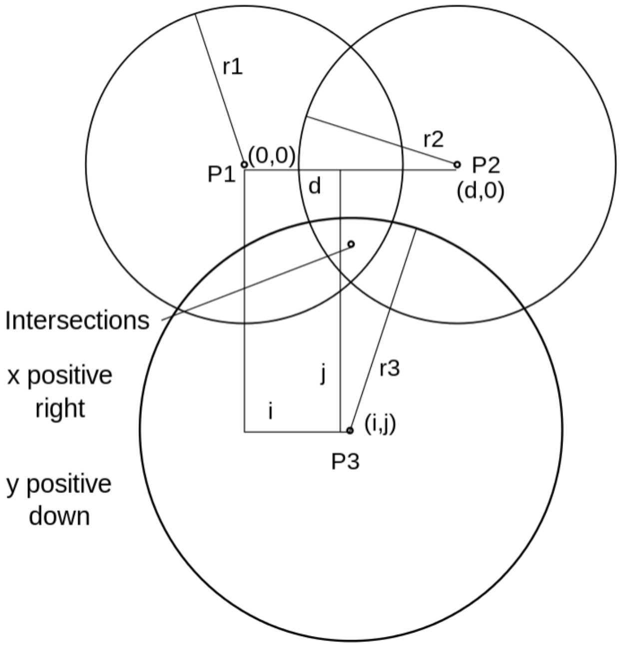

FIGURE 3.1: The area enclosed by the three intersections ca be used to localize univocally. The three simple intersections formed by pairs of circumferences are always outside of the Earth. As more satellites are included, precision increases. Taken from [23].

The best localization services can be provided with the conjunction of these three main techniques, which can be easily merged with an *Extended Kalman Filter*.

Although these techniques are widely used they are limited to mobile platforms, which are restricted both in sensorization and in processing capabilities. For robotic applications more sophisticated sensors are used, this includes depth cameras and lidars for the most part. Again, all the sensors output is merged to get the best estimates.

In robotics, it is usual that the localization we are interested in is relative instead of absolute. This is done to aid in locating near objects in the space relative to the robot but also to construct a map, the process of localization and mapping simultaneously is called *SLAM - Simultaneous Localization And Mapping* (see sect. 3.1.3).

3.1.2 Indoor localization

To localize in indoor environments many strategies can be followed, the general trend is to place different markers (active or passive) beforehand in well known locations.

This markers are then recognized by the localizing device to know it's location. This *recognizable markers* can be anything, a Bluetooth beacon, a WiFi hotspot, etc.

Bluetooth beacons are specially crafted for this purpose as they can provide much more information. It has been extensively used in congresses and hotels to provide hosts with more information beyond localization, as services and timetables based on location.

In the case of WiFi hotspots localization is usually done by analyzing the signal strength and incoming angle. This method only works when the hotspots' location is known beforehand and is very prone to errors because the device must remain static in a certain angle.

From the computer vision perspective, visual markers can be placed too and processed by the device localizing and again, this requires preparation beforehand. One example of this setup are the well known Arucos [15].

In many robotic applications like swarm robotics there is a necessity to track each member of the swarm in a closed, contained environment, for this purpose an OptiTrack system [21] can be used, it is a highly precise camera set that can track various markers (marked swarm members) and serve it's location through the network in real time. This setup is specially useful to monitor the swarm, enabling each member to access it's location.

3.1.3 SLAM

SLAM is the process of mapping and, at the same time localizing inside that map. This is particularly useful in environments that are not prepared like the ones exposed previously, enabling the robot to work on an unknown place without getting lost nor entering cyclic paths. SLAM techniques are crucial in any robot with some degree of autonomy, it makes the navigation possible.

There are three main variants here, the ones based on Extended Kalman Filters, the probabilistic ones and the ones based on Graph Optimization.

3.1.3.1 Extended Kalman Filters SLAM

Extended Kalman Filters (EKF) SLAM is the earliest technique developed for SLAM. EKF is a general technique to find the best estimate for the measuring variable based on the mean and covariance. In this case the inputs are the odometry used to estimate the robot position, features of the environment that anchor the odometry measures and the robot motor system sensors (wheel decoders, etc) to estimate the change on the position. Then the objective is to find the best estimate for the current robot's position.

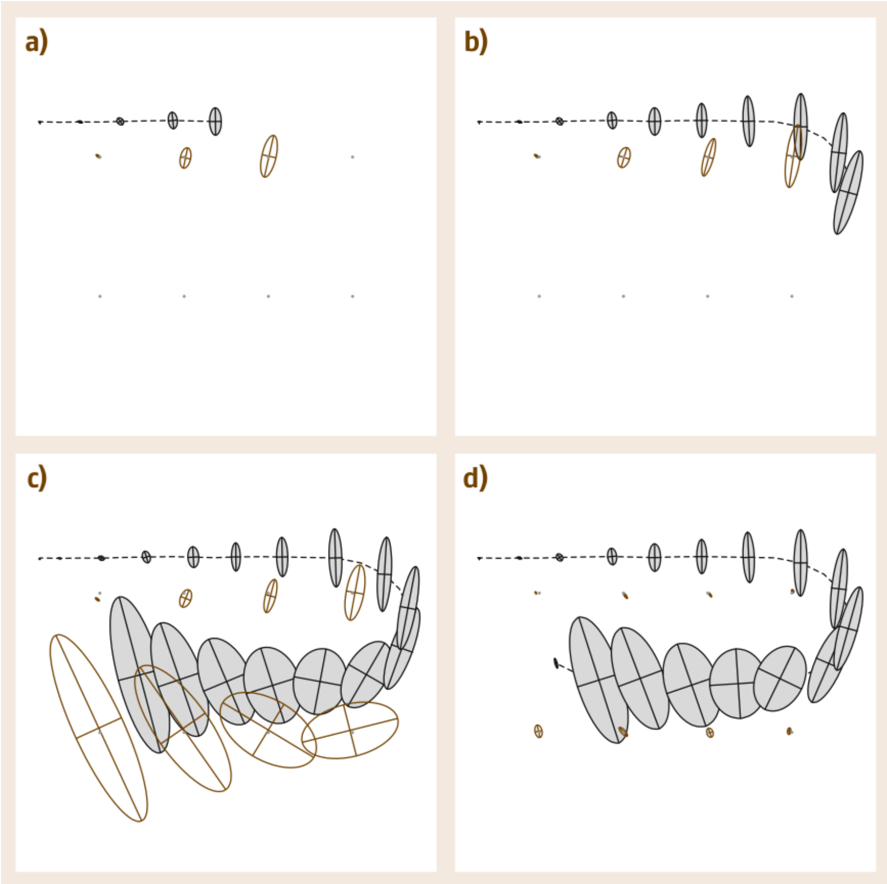

FIGURE 3.2: EKF applied to the SLAM problem. Dotted line: robot's path. Shaded ellipses: position estimates. Small dots: Unkown location landmarks. White ellipses: Landmarks' position estimates. In (d) the robot senses the first landmark, anchoring the rest of the estimates, reducing uncertainty. Taken from [18]

At the start of the process, the system's (0, 0) coordinate is established where the robot is, this is the most confident measure about the robot position. As the robot navigates the environment, succesive measures are taken and paired with the known landmarks, when a previously seen landmark is witnessed again, the position estimate is corrected with the covariance matrix, and the error correction is propagated along the previous estimates. In this way a long as the robot is navigating an sensing the landmarks, the position estimate improves. Figure 3.2 shows the full process, in (a) the process is started, there is a lot of uncertainty, as process continues (a-c) the uncertainty increases, until the first landmark is sensed again (d), reducing the uncertainty on the current position's estimate and the subsequent estimates.

More formally, the EKF algorithm represents the robot estimate by a multivariate Gaussian (eq. 3.1)

$$p(x_t, m|Z_t, U_t) = N(\mu_t, \Sigma_t) \tag{3.1}$$

Where μ_t contains the robot's best estimate of its current location x_t and the locations of all the landmarks, its size is $3 + 2N$, 3 points for the robot location and 2 for each of the N landmarks. The matrix Σ_t is the covariance of the expected error in the guess μ_t assessed by the robot, a square, dense matrix of $(3 + 2N) \times (3 + 2N)$.

Although this technique works well on small maps, it renders unusable for large maps, this happens because the covariance matrix Σ_t used to correlate the position estimates grows cuadratically with the measures, making the memory footprint wildly large and the overall processing time very high. Some researchers have proposed an improvement over the EKF SLAM algorithm through submap decomposition [6, 10].

3.1.3.2 Particle Filters

Particle Filters are a probabilistic approach to position estimation, usually called Fast SLAM [12]. It uses various *particles* that represent the posterior probability of the true distribution of maps and possible paths. To do so it stands over a method called Rao-Blackwellization, that aids in dimensioning the number of particles needed to represent the map. Also, as conditional indepence is assumed between the observed landmarks, every landmark can be represented as N small Gaussians, which is linear, instead of exponential on the number of landmarks.

At any point a set of K particles is retained, where each particle has the form exposed in equation 3.2.

$$X_t^{[k]}, \mu_{t,1}^{[k]}, \cdots, \mu_{t,N}^{[k]}, \Sigma_{t,1}^{[k]}, \cdots, \Sigma_{t,N}^{[k]} \tag{3.2}$$

Where k is the index of the path sample and n the number of the landmark. This implies that every particle contains a sample path $X_t^{[k]}$ and a set of N Gaussians $\approx (\mu_{t,n}^{[k]}, \Sigma_{t,n}^{[k]})$, one for each landmark.

When a new odometry measure is received, it is combined with the previous knowledge through probabilistic sampling. A new location is generated for each particle, following a distribution based on the robot motion model and the previous measure of that particle $(x_{t-1}^{[k]})$. More specifically:

$$x_t^{[k]} p(x_t|x_{t-1}^{[k]}, u_t) \tag{3.3}$$

Then, when a new measure z_t is received, each particle's importance is weighted, assigning how important is that particle to that measure, this is the probability of that measure based on the particle's knowledge, defined in equaton 3.4. Let n be the observed landmark's index:

$$w_t^{[k]} = N(z_t | x_t^{[k]}, \mu_{t,n}^{[k]}, \Sigma_{t,n}^{[k]}) \tag{3.4}$$

After equation 3.4 is applied for each particle, all the weights are normalized to sum up to 1, then a set new particles is drawn with replacement, where the probability of being picked is each particle's weigth. Intuitively, this means that only the particles that fit the most with the current measures survive for next rounds. The final step of FastSLAM updates the mean $\mu_t^{[k]}$ and covariance $\Sigma_{t,n}^{[k]}$ based on the new measure z_t, which is similar to the EKF updates, but with much smaller filters.

Although this method is easy to implement, fast enough for real time applications with not very high demanding software and yields good results on small to medium maps, it suffers from the fact that lots of particles are needed to represent big maps, specially with multiple nested loops. Therefore, many improvements have been proposed, [5] for example uses occupancy grids instead of Gaussians.

3.1.3.3 Graph Optimization SLAM

The Graph Optimization SLAM techniques try to optimize a graphical model representing the landmarks and robot locations. In this representation, each location is viewed as a node in the graph, and the edge (called soft constraint) between two consecutive nodes is the captured odometry. The key intuition behind these methods is that at the end, the graph is sparse, because, each node will have just a few connections to other nodes. Also, at worst, the number of entries in the graph is linear in the time elapsed and in the number of nodes.

This is the most widely used approach because sparse linear optimization is in a very advanced stage, allowing for scalable, yet efficient implementations of the algorithm.

3.1.3.4 Lidar SLAM

Lidar is a well established laser range sensor that can be used for depth estimation. By doing fast sweeps in 360 degrees it can compose a depth map which can be used for SLAM.

Hector SLAM [8] is a technique developed in the Darmstadt University. It uses a lidar sensor to do a fast SLAM by matching rays along sweeps.

Along this work, the *hector_slam* ROS module will be integrated into the Aerostack framework, providing a robust SLAM technique ready to use for the drones equiped with lidar.

3.1.3.5 Visual SLAM

Using computer vision for SLAM have been a challange since it's conception, it raises the difficulty, specially for monocular cameras. Although many features can be extracted from images, it is not clear how to process nor store the data taking into account the full 6 degrees of freedom in a camera. All the parameters of the camera must be known beforehand, depth cameras include a lot of noise and monocular cameras do not have scale.

In [13], ORB-SLAM is proposed. Intuitively, it creates ORB features from a visual input and stores it in a sparse matrix, then a matching process is launched to localize every feature, improving the localization along the way. It can work on Monocular (no scale), Stereo and Depth Cameras, giving extraordinary results.

3.2 Planning

As planning algorithms have been around since the conception of robots, we will just go through the employed method and it's integration with Aerostack.

Once the occupancy grid map is constructed and target position is given, a plan has to be made in order for the robot to get to that target. Also, obstacles should be avoided. The module in charge of this task is the planner.

There exists many planning techniques, amongst the most common, we find *probabilistic road maps* [7], which builds a unidirected graph that connects random points for later exploration, or *rapidly exploring random trees* [9], that grows a tree in the search space from the starting position until the goal is reached.

In the current work we will employ a planner based on *elastic bands* [14]. The elastic band framework comprises a gloabl planner that makes a valid, global path based on the current world model and a local planner that deforms that path in real time to accomodate the perceptions of the robot. This way, obstacles can be avoided in real time by applying forces (internal o external) to the path as if it where atracted or repulsed. Note that this planning is agnostic of the controllers or the internal map representations of the robot.

More specifically, we will integrate an already made ROS module called *move_base* [11] for the planning that uses a local planner (*eband_local_planner* [1]) implemented after the elastic band framework.

This chapter reviewed the main adversities inherent to the SLAM problem and shed some light over the current state of the art. Aslo, a brief introcuction to planning was made. In the rest of this document, we will focus solely on lidar SLAM, as the drones used with Aerostack are bound to lidar sensors and elastic band planning.

Chapter 4

Implemented Modules

In this chapter, the technical goals of the project are introduced. Each developed module will be explained as well as it's requirements and specification details. For each module, the following points will be addressed:

- Technical goals
- Problem to which the module provides a solution
- Some of it's properties (reusability, scalability ...)
- Integration with the current version of Aerostack

The chapter is structured as follows: Section 4.1 presents the requirements imposed for the implemented modules, explaining the specifications for Aerostack modules 4.1.1 and its integration 4.1.2. Sections 4.2 and 4.3 explain the slam behavior and the navigation API respectively and Sections 4.3.1 to 4.3.4 go through the rest of behaviors, explaining its particularities.

4.1 Requirements

In order for the Aerostack framework to localize with a different technique rather than visual markers, a lidar sensor will be used. In this case, the *Hokuyo Eye* range sensor will be employed, which is the *defacto* range sensor in this context. Also, the low level implentation provides a nice ros API that can be used to fetch data. To wrap all this functionallity we propose the implementation of a new high level behavior that coordinates all the framework with the lidar interface, providing a high level, standarized API for lidar-based localization.

As of the current version of Aerostack, navigation is done with a 2D probabilistic roadmap planner, the input for the planner is a predefined map, done by hand in the Graphical User Interface that Aerostack provides. This is a static map and goes against the nature of the any dynamically acquired mapping signal. To tackle this problem several new navigation behaviors are proposed. These behaviors will abstract the planner used for each localization mode, providing a high level standarized API that can be used independently of the localization technique, replacing the old one.

4.1.1 Specification

Each implemented module should follow the specification imposed by the Aerostack framework. In Aerostack there are different types of processes providing structure and added functionallity. When a new process is created it should be decided whether to implement it as a plain, simple ros node, a robot process or a behavior process. Their differences are as follows:

- ROS Node: This is the standard way of adding modules in a ROS oriented architecture. A ros node is simply a process programmed in any of the programming languages supported by ros (C, C++, Python . . .) that implements a task and is interfaced through the ros master server with named topics, services or both. A ros node can subscribe or publish topics and optionally, provide services, as many topics or services as it wants. These topics and services are nothing more than binded ports to the ROS master server, that works over TCP (normally) or UDP to distribute traffic. This is the implementation to follow when adding very low level modules, like platform drivers.
- Robot Process: A Robot Process is an abstraction provided by Aerostack, it serves mainly as a standarization layer, providing an interface for the rest of the architecture to be used. It provides three services to manage the process, one for stopping it, one for starting it and another one to check whether it is running or not, aditionally it emits an alive signal every second or an error signal when the thread crashes. It runs the inheritors' code inside a separate thread in order to monitor it. When adding a module that abstracts some low level APIs, like a visual marker processor, this is the class to inherit from.
- Behavior Process: This is the highest level of the hierarchy, inside the Aerostack framework there exists a process that coordinates all the behaviors, to do so, every behavior exposes an interface similar to the Robot Process and a configuration file that especifies the mid and low level processes the behavior depends upon (it's

capabilities and incompabilities), amongst other parameters, in this sense, the behavior that provides localization based on visual markers depends on the visual marker processor to work. Formally, a behavior is just a high level process that monitors an algorithm: it runs the algorithm in a separate thread and emits the state and error signals, listening to *start/stop* events and acting accordingly over the algorithm. When adding a high level functionallity, this is the class that should be inherited.

In a similar fashion to the visual marker localization behavior, the lidar localization behavior proposed will require three more processes: the slam process, an ekf that combines various signals and a localization technique selector, this is explained in detail in the corresponding behavior section (sect. 4.2).

The navigation interface is slightly more complicated, it will be composed of various behaviors that provide different functionallity, abstracting away the logic needed to navigate at different levels. We propose three new behaviors and the inclusion of a new planner, efficiently designed to work with lidar signal. More details will be provided in the corresponding section 4.3

4.1.2 Integration

To integrate each behavior, we will follow a bottom up procedure. This way, we will ensure that the processes the behavior depends on are working correctly inside the Aerostack and the error doesn't get masked with the behavior integration.

When integrating a new behavior some steps should be followed:

- Add the necessary mid and low level processes to the Aerostack and ensure they can be started automatically.
- Add the technical specification of the behavior to the behavior catalog. These are the capabilities and incompabilities of the behavior and should include the mid and low level processes previously mentioned so that they can be started automatically. In this step, the behaviors that are incompatible with the new one should be identified.
- Add the implementation of the behavior and test it with the Aerostack to ensure it can be started and that no incompatibilities arise.

The lidar-based localization behavior will provide a new localization mode, so it is reasonable to mark the rest localization behaviors as incompatible, also, a new localization method selector process will be added, this will ensure that when various localization techniques are to be used in the same mission, they can be easily toggled on and off. This will be detailed in the corresponding behavior section 4.2.

We can define the requirements for these modules more formally, in a global sense, the new functionality added should meet the following requirements.

1. It should be able to handle lidar sensor input and use it both to localize and map.
2. It should be able to do planning with the available data to get to a desired goal avoiding obstacles.
3. It should be able to adapt in environments with moving obstacles.
4. It should be efficient enough to run both onboard and on ground control stations.
5. It should be possuble to be interfaced at different levels of abstraction.
6. It should provide an standarized API access to navigate the robot using lidar sensorization.

4.2 Behavior Self Localize and Map by Lidar

The lidar range sensor outputs raytraces reflected over the near objects, in a way, it resembles a sonar sensor (that's way it's called lidar). Each raytrace, measures the distance from a concrete angle to a point at a certain distance, these measures then have to be converted in some way that can be used to map the environment and use this mapped environment to localize inside it (see section 3.1.3 for an in-depth explanation of SLAM). Refer to figure 4.1 for a visual representation of the behavior and it's subprocesses.

We will use a ros module developed at the Darmstadt University that provides a SLAM node (*hector_mapping* [8]). Taking the lidar's cloud of points it is able to construct an occupancy grid map and then localize inside it (SLAM). The output of this module can be used to create a plan avoiding obstacles to reach a target point. It will output the estimated localization along with the mapped environment. This localization will be merged with the measures from the rest of the sensors (namely odometry, IMU ...) using an extended kalman filter (see section 3.1.3.1) to output a robust estimate of the robot's position inside the mapped world.

The process in charge of the EKF is called *droneRobotLocalization* and inherits from *Robot Process* class. It will listen for updates on the robot's pose (*hector_pose* topic) and both the IMU and the odometry topics and output the estimated pose.

The estimated pose is then fed to the selector, which will toggle the localization technique. This implementation opens the door to new localization behaviors, a GPS based

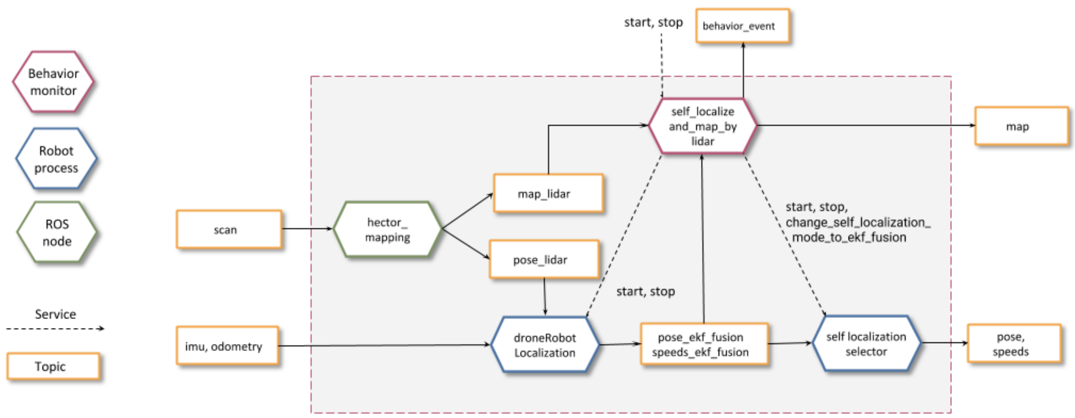

FIGURE 4.1: Behavior self localize and map by lidar architecture. *Hector mapping* is the SLAM module from [8]. *Drone Robot Localization* does the EKF. *Self Localization Selector* gives the localization based on the selected technique.

one for instance and also makes compatible the previous ones (visual markers or odometry based). It is easy to think of an scenario that requires both indoor and outdoor localization, in such a mission, the localization technique in use should be toggled in order for the navigator to work properly. As can be seen in Fig. 4.1, this is a *Robot Process* with an added service to change the localization technique used.

Bellow there is a list of the inputs and outputs of this behavior:

- **scan**: This is the output of the lidar node, it is directly fed into the *hector mapping* node
- **imu**: This topic contains the measures from the *inertial measurement unit*
- **odometry**: This is a general topic with the measures from odometry
- **map lidar**: Map output from *hector mapping*, used as internal representation for the behavior.
- **pose lidar**: Pose output from *hector mapping*, used as internal representation for the behavior and input to EKF fusion module.
- **map**: This is the map as processed by the *hector mapping* node
- **pose ekf fusion**: This is the estimated speed from the whole behavioor process, fed to the selector.
- **speed ekf fusion**: Estimated pose from the whole behavioor process, fed to the selector.

```yaml
#-----------------------------------------------------
# SELF_LOCALIZE_AND_MAP_BY_LIDAR
#-----------------------------------------------------
  - behavior: SELF_LOCALIZE_AND_MAP_BY_LIDAR
    timeout: 120
    processes:
      - hector_mapping
      - droneRobotLocalization
      - self_localization_selector
#-----------------------------------------------------
# Self-localization behaviors are mutually exclusive
#-----------------------------------------------------
- mutually_exclusive:
  - SELF_LOCALIZE_BY_ODOMETRY
  - SELF_LOCALIZE_BY_VISUAL_MARKERS
  - SELF_LOCALIZE_AND_MAP_BY_LIDAR
```

FIGURE 4.2: Self Localize and Map behavior catalog specification. Required processes and timeout values are specified.

This behavior monitors the correct working of the algorithm (*hector mapping*) by listening on the *map* topic, when it outputs strange or simply wrong data, an error is emitted.

In the configuration file of this behavior the localization by visual markers behavior will appear as incompatible. As for the capabilities, all of *hector mapping*, *drone robot localization* and *self localization selector* will figurate as capabilities, indicating that those processes should be started before this behavior. Furthermore, when it's activation conditions are tested (*checkOwnActivationConditions*, native method each behavior should implement), it will check that the robot actually counts with a lidar interface. All these configuration parameters are depicted in figure 4.2.

The monitor algorithm will consist in checking the consistency of the output map as well as it's frequency, when a drift is detected a warning will be printed, indicating that the slam module is having problems. In the worst case, when the node does not output a map at all, an error will be emitted and the behavior will terminate.

4.3 Navigation Interface

We will consider the navigation interface as the minimum set of behaviors necessary to provide a robust, flexible API to do navigation tasks related to lidar-based localization and mapping techniques. It should be able to generate obstacle-free trajectories to any given point (when there exists one) and be able to move the robot along those trajectories.

The identified tasks for this API are as follows:

1. Given a point (or goal), execute the necessary motions to get the robot to that goal.
2. Given a path, execute the necessary motions to follow it until the path is finished.
3. Given a point (or goal), generate an obstacle-free path from the current robot's position to that goal.

This tasks can be directly mapped with processes. However, we will implement them as separate behaviors to provide more modularity and reusability. Also, as each process will provide abstraction at a certain level of granularity, it makes sense to implement it as separate, independent behaviors (although some code will be duplicated)

As of the current version of Aerostack, there exists a behavior that executes the motion of going to a given point in the 2D map representation used by Aerostack. However, this behavior is not general enough to be used with a different map representation, so we will implement a new one capable of executing the motion in the new map format: occupancy grids (which is the format used in *hector mapping*). Also, in a dynamic environment, obstacles can arise in the path, this behavior will ensure that no collisions happen when executing the motion. More details to follow in section 4.3.1.

The task of following a path or trajectory consists in instructing the previously available trajectory controller to follow a set of points (that conform the trajectory), given in a specific reference frame (world coordinates in this case). Contrary to the previously defined behavior, this one executes the motion blindly, providing a lower level of control to the user. Please refer to 4.3.2 for more details.

For the last functionality, generating obstacle-free paths, another behavior will be implemented. It will consist mainly in a wrapper around the new planner, providing the lowest level of control in our navigation interface. For the planning we will employ the previously mentioned planner *move base*, provided as a ros package, which is specially crafted for lidar interfaces. It accepts an occupancy grid map and the raytraces from the lidar and implements the planning algorithm. Under the hood, it uses the elastic band algorithm for path optimization, as mentioned in section 3.2.

The proposed names for each behavior are: *behavior go to point in occupancy grid*, *behavior follow path in occupancy grid* and *behavior generate path in occupancy grid*. Figs. 4.3, 4.5 and 4.7 ilustrate the architecture followed by each of these behaviors. The following subsubsections explain each behavior in detail.

4.3.1 Behavior Go to Point in Occupancy Grid

This behavior provides the highest abstraction level of all the navigation interface, provided a target point, it will generate an obstacle-free trajectory to follow and send it to the trajectory controller, which executes the necessary motions to follow that trajectory. During the motion, this behavior will also ensure that no dynamic object gets in the way, recalculating the trajectory if necessary. In order to plan the trajectories, the new planner, *move base* will be used, and as it will be a goal based behavior, the taget goal will be given as an argument to the start service. Figure 4.3 illustrates the general architecture of this behavior.

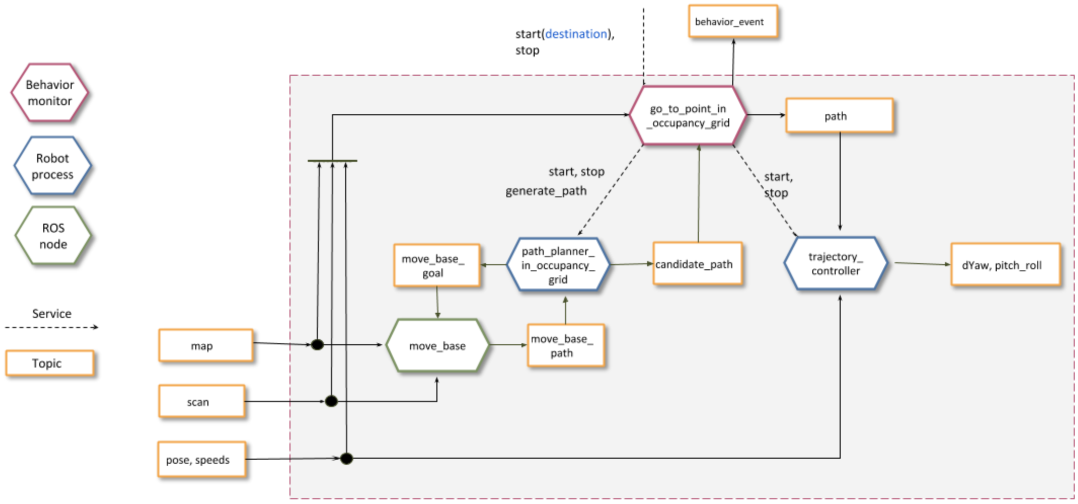

FIGURE 4.3: Behavior Go to Point architecture. Move base is the planner

A condition for this behavior to operate correctly is that no other behavior is instructing the trajectory controller. To ensure this condition is met, the list of motion behaviors is stated as incompatible. The capabilities should list that it is a set-point-based flight behavior, instructing the behavior coordinator to setup the trajectory controller accordingly, the new planner (move base) should be explicitly declared as well so it is started automatically. Figure 4.4 depicts the part of the global configuration file pertaining this behavior. Using four parameters as input enables the behavior to accept orientation (yaw) commands as part of the goal, making the behavior more flexible.

The proposed topics will be:

- **scan**: Scan data from the lidar, direct input to the planner.
- **map**: Occupancy grid map from the self localize and map by lidar behavior, direct input to the planner.

- **pose, speed**: Estimates of the current robot's position and speed.
- **move base goal**: Topic to instruct the planner to calculate an obstacle-free trajectory. Used by the path planner module.
- **move base path**: Topic with the planned trajectory from the move base module. Grabbed from the path planner module.
- **candidate path**: Output topic with the candidate trajectory, output from the path planner, grabbed from the behavior to instruct the trajectory controller.

```
#-----------------------------------------------------
# GO_TO_POINT_IN_OCCUPANCY_GRID
#-----------------------------------------------------
  - behavior: GO_TO_POINT_IN_OCCUPANCY_GRID
    timeout: 240
    processes:
      - move_base
      - path_planning_in_occupancy_grid
      - droneTrajectoryController
    arguments:
      - argument: coordinates
        allowed_values: [-100,100]
        dimensions: 4
```

```
#-----------------------------------------------------
# Motion behaviors are mutually exclusive
#-----------------------------------------------------
- mutually_exclusive:
  - TAKE_OFF
  - LAND
  - KEEP_HOVERING
  - GO_TO_POINT
  - ROTATE
  - KEEP_MOVING
  - FOLLOW_OBJECT_IMAGE
  - GO_TO_POINT_IN_OCCUPANCY_GRID
  - FOLLOW_PATH_IN_OCCUPANCY_GRID
- mutually_exclusive:
  - GENERATE_PATH_IN_OCCUPANCY_GRID
  - FOLLOW_PATH_IN_OCCUPANCY_GRID
  - GO_TO_POINT_IN_OCCUPANCY_GRID
```

FIGURE 4.4: Go to Point behavior catalog specification. Required processes and timeout values are specified.

The process *path planning in occupancy grid* is in charge of monitoring the *move base* planner, it's the intermediary between any behavior requesting a trajectory plan and the planner itself. It is explained in detail in section 4.3.4.

When the behavior is started it receives an absolute or relative coordinate to go, it waits for the current position estimate and starts planning. If the robot's orientation has to be changed it first instructs the trajectory controller to do so, until the target orientation is achieved. Once the robot is correctly oriented, the path planner is asked for a candidate path to follow and the trajectory controller is instructed to follow it. Also, while the trajectory is being executed, the behavior still listens to new paths, this way, when the planner outputs an empty trajectory we know that an obstacle is obstructing the path, stop the controller and replan again.

This strategy for detecting obstacles comes from the fact that the newly integrated planner outputs an empty path when an obstacle is found on the way.

As for the activation conditions, this behavior requires that the battery is not in a *LOW* state and the aircraft is in state *FLYING*. To check these conditions, it will make use of the *belief manager* that stores important state information from all the Aerostack in the so called *beliefs*. Beliefs are the symbolic representation of the known environment and gathered information so far in the mission, they are the other part of the executive layer, providing persistance accross the whole mission. For example, when the drone starts flying it will toggle its state from *LANDED* to *FLYING* and the belief manager will gather this information for easy retrieval, this goes for the battery state too.

4.3.2 Behavior Follow Path in Occupancy Grid

This behavior will communicate with the trajectory controller, instructing it to move following a given path received as an input argument. For this behavior to work correctly, the motion behaviors should be disabled too. In this case, no other low level process is needed. The following figure illustrates the general concept (4.5). Following a path can be a useful feature when developing fine grain controlled missions, like the ones enabled with Python.

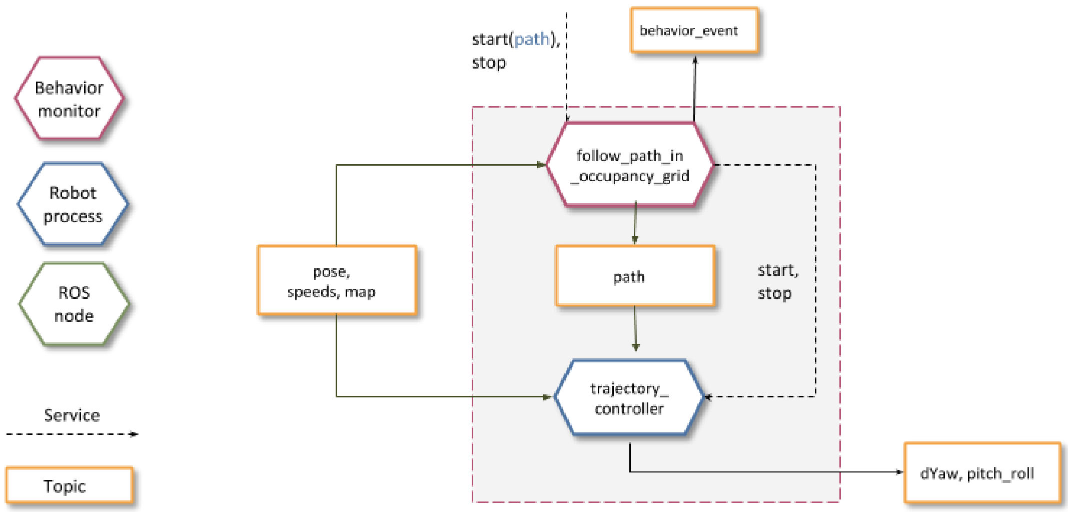

FIGURE 4.5: Behavior follow path in occupancy grid architecture

The important topics are explained below:

- **pose, speed, map**: These are the input topics both for the trajectory controller and the behavior.

Again, the correct settings should be ensured in the configuration file, marking motion behaviors to avoid trajectories interferences. This settings can be observed in figure 4.6.

This behavior has similar constraints to the previous one, if the aircraft battery is low or is landed, it cannot be executed. In this case, the monitor is much simpler, given a trajectory it instructs the trajectory controller to follow it, not even listening to

```yaml
#-----------------------------------------------------------
# FOLLOW_PATH_IN_OCCUPANCY_GRID
#-----------------------------------------------------------
  - behavior: FOLLOW_PATH_IN_OCCUPANCY_GRID
    timeout: 240
    processes:
      - path_planning_in_occupancy_grid
      - move_base
      - droneTrajectoryController
    arguments:
      - argument: path
        dimensions: 1
```

```yaml
#-----------------------------------------------------------
# Motion behaviors are mutually exclusive
#-----------------------------------------------------------
- mutually_exclusive:
  - TAKE_OFF
  - LAND
  - KEEP_HOVERING
  - GO_TO_POINT
  - ROTATE
  - KEEP_MOVING
  - FOLLOW_OBJECT_IMAGE
  - GO_TO_POINT_IN_OCCUPANCY_GRID
  - FOLLOW_PATH_IN_OCCUPANCY_GRID
- mutually_exclusive:
  - GENERATE_PATH_IN_OCCUPANCY_GRID
  - FOLLOW_PATH_IN_OCCUPANCY_GRID
  - GO_TO_POINT_IN_OCCUPANCY_GRID
```

FIGURE 4.6: Follow Path in Occupancy Grid behavior catalog specification. Required processes and timeout values are specified.

path events. Not avoiding obstacles was a design decision taken to promote developers independence and granularity control. When the behavior starts, the provided path will be parsed into a known data structure, if no path is given or it is malformed an error will be thrown and the behavior will finish.

4.3.3 Behavior Generate Path in Occupancy Grid

In this case we will only implement a wrapper around the path planner module. This is the lowest abstraction level behavior provided by the navigation interface API. Being able to generate paths can be useful for fine grain controlled missions and debugging. The only communication way for this behavior will be dropping the planned path on the belief memory, this way the amount of traffic and topics is reduced, saving computing resources and avoiding polluting more topics. It's architecture is depicted in figure 4.7.

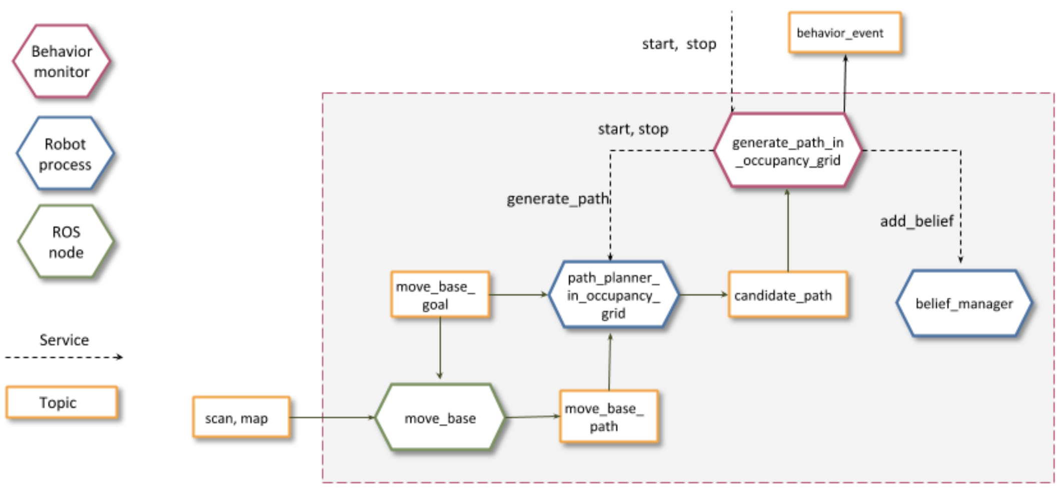

FIGURE 4.7: Behavior generate path in occupancy grid architecture

```yaml
#----------------------------------------------------
# GENERATE_PATH_IN_OCCUPANCY_GRID
#----------------------------------------------------
  - behavior: GENERATE_PATH_IN_OCCUPANCY_GRID
    timeout: 120
    processes:
      - move_base
      - path_planning_in_occupancy_grid
    arguments:
      - argument: coordinates
        allowed_values: [-100,100]
        dimensions: 3
```

FIGURE 4.8: Generate Path behavior catalog specification. Required processes and timeout values are specified.

Contrary to the rest of the presented behaviors, as this one does perform any motion, it does not have any restriction, nor it conflicts with any behavior requiring the trajectory controller. This behavior will store any given path in the belief memory with a different unique identifier, ensuring that the computed paths are not smashed, this is a desirable feature for many reasons, the most obvious being able to avoid the overhead of planning again, reusing precalculated paths. This allows for a developer to execute a trajectory by parts for example. Figure 4.8 overviews the part of the catalog dedicated to this behavior.

4.3.4 Path Planner in Occupancy Grid

This module was developed as part of the Navigation Interface to provide a unified API access to the move base planner, reducing the overhead of changing the planner in the future and monitoring the correct funcioning of the move base planner.

It implements a queue of target goals and dispatches them one by one to the planner. This decision was taken for two reasons, the main one is that *move base* contains a bug, when the computation power is lowering, sometimes the planner fails to generate a path. To monitor this strange behavior and correctly handle it we implemented this intermediate module. The second reason is that in the future, the underlying planner could be changed, unifying a path planning API will smooth the transition, making it transparent to the API consumers.

This chapter has reviewed the most important aspects and specification of the implemented modules. In the next one, the validation and testing for these behaviors will be introduced.

Chapter 5

Validation

This chapter will go through all the validation and tests implemented to measure the performance and the correct functioning of the new behaviors.

To validate each implemented behavior we propose using them both in simulation and in real flight missions. The validation tests are presented in detail in section 5.1, explaining what the tests measure and how they do it as well as the intuition behind each test. Afterwards, in section 5.5 we explain in detail the mission implemented for each behavior along with further information of the simulator employed in section 5.5.4. The simulation integration with the current version of Aerostack and the specific details of the real flight aircraft are detailed in section 5.5.5. Section 5.6 reports on the results obtained for each mission both in sumulation and in real flight and 5.7 closes the chapter with the discussion of the results.

5.1 Validation Tests

A validation test should ensure that the behavior complies with the following constraints:

1. Functioning: It should do what it is supposed to do. No more, no less.
2. Requirements: It should meet the imposed requirements (explained in 4.1)

The first constraint is the most obvious, a behavior is designed to do a concrete task. It should do only what is designed to do. The idea behind a behavior is to encapsulate a concrete algorithm, it can be the layer that encapsulates the algorithm, providing a standard API access, but it can only be one functionality.

To test this constraint we will provide both a simulation and a real mission and put to work each behavior independently, testing that each one works as expected. We will measure the precision of each behavior based on the amount of the task that it is able to complete without errors.

One of the hardest requirements to meet is that of the efficiency, any implemented behavior should be efficient enough to run both onboard and in ground control stations in real time. Although this requirement could not be a problem for a PC it surely can be a problem on onborad computers. Also, it is difficult to measure for real flight missions because a cpu tracker also takes computing resources and the onboard computers mounted on the UAVs are not very powerful. Nevertheless, the cpu usage will be measured both in sumulation and in real flight and the corresponding data will be provided for analysis.

5.2 Simulation

The chosen simulator for these tests is Gazebo Sim ([19]), an open source, multiplatform, robot simulator. It was chosen because of it's open source nature, ease of use and great integration with ROS. Gazebo features an open, modular, plugin based architecture, which makes it perfect to integrate new components and opens the door for modules being simulated inside it and the outside world. In our case, to communicate the simulated UAV with ROS and the Aerostack framework.

In order to convey the simulator with ROS we will employ an already made plugin called RotorS [4]. RotorS provides some UAV models and a plugin that translates gazebo topics to ROS ones, unifying the access to the data. This architecture is perfect for any robotic environment as it minimizes the overhead of changing from simulation to real flight, as long as the topics are called the same, the Aerostack framework does not even notice it.

As all the simulation is launched with configuration files and was done with flexibility in mind, we could adapt the already available configurations to our needs, minimizing the overhead of naming topics to our conventions.

To choose a UAV model from the available ones we looked up for one that matches our requirements, namely a lidar sensor (hokuyo if possible), a front camera and an altitude sensor. RotorS provides a UAV modeled after the AscTec Hummingbird [20] drone, which meets all these requirements.

5.3 Real Flight

Our requirements came mainly because of the implemented behaviors, which in turn came from the available hardware in the research group. Currently, the aircraft used for the most important missions is a Matrice 100 from DJI ([3]) with all the sensorization cited above.

This is used because of it's extendability (any sensor platform can be plugged in), open API and powerful motors. It's verstaile enough to provide a testbed for many research experiments and the battery lasts enough for medium duration missions.

5.4 Testing Mission

To provide the most realistic environment possible, the mission used to conduct all the experiments will be based in a real assignment requested to the research group a few months ago. The mission consists in inspecting the internal facade of a plant's boiler. At the time of request this new navigation interface was not implemented, so the flight was made almost by hand. In fact, this inteface was proposed after the need of an autonomous flight navigator whith the available hardware.

The idea behind the mission is to fly a drone along the facade filming all the breathers, after the mission is completed, the film is extracted and given to the plant experts for their analysis. Additionally, a handmade mission could be very costly for the gas company as they would have had to install a portable crane and put a human to do the inspection of the 40 meters long facade.

A human commanded drone accomplished the whole inspection in less than one hour. Furthermore, even in that case it was challenging for the human operator as he had to fly near the wall, which causes the drone to destabilize due to the air flows. This is the point where autonomous navigation comes in. In an autonomous mission, the control loop can be closed with any parameter that can be measured, in this case, the go to point behavior could have been employed to make the aircraft fly upwards until the whole boiler is inspected and then commanded back to land at a certain point.

As the job is already done it is not possible to replicate it, hence we propose to simulate a boiler in Gazebo that resembles to the real mission and the inspection of an internal facade in a building for the real flight.

The next sections will deepen in all the details of the conduted experiments and the results obtained during the tests.

5.5 Experiments

As mentioned before, for each behavior we will ensure it complies with our constraints: correct functioning and requirements (efficiency mostly).

Once the whole Aerostack system is deployed, the first test for each behavior consists in checking it's activation conditions, i.e.: the *behavior self localize and map by lidar* cannot work without a lidar, deploying the whole Aerostack in a lidar-less UAV should cause the behavior to be deactivated instantly.

This way we tested all conditions and capabilities of all the proposed behaviors. As this is all software related, we easily corrected all the bugs found. One example of this was the access to the path planner module: The new planner (*move base*) can only plan to one goal, this makes all the behaviors requiring this module mutually incompatible, namely, the behavior that generates paths cannot work simultaneously with that of the go to point . In the same fashion the behaviors that make use of the trajectory planner cannot work together or a collision could occur.

For the functioning constraint we setup both the real and simulated missions with a python script that commands the Aerostack, if after doing a certain amount of missions, the behaviors work as expected we consider that they work correctly.

The following subsections explain the implemented mission for each behavior. Note that, the first two missions where only tested on the simulated environment to minimize the human operator time needed for the tests. For all missions both the simulation and the real environments used are the same, nothing was changed in the environment.

5.5.1 Behavior Generate Path in Occupancy Grid

This mission was only tested in simulation, it consisted in generating a path for every point in the go to point mission (sect. 5.5.3). After the mission is finished, 6 paths should be present in the belief memory. Therefore the mission can be described as follows:

1. Generate path for point: [0, 0, 1.5]
2. Generate path for point: [1, 0, 1.5]
3. Generate path for point: [1, 0, 10]
4. Generate path for point: [1, -5, 10]
5. Generate path for point: [0, -5, 1.5]

6. Generate path for point: [0, -5, 1]

The full python source code can be found in the appendix A.2

5.5.2 Behavior Follow Path in Occupancy Grid

In this mission we employ the previous behavior to generate paths from the current position to the target point and fed it as input parameter to this behavior. Working in tandem, these behaviors do a similar job to the go to point one (without the obstacle avoidance). The generated mission is outlined as:

1. Follow path for point: [0, 0, 1.5]
2. Follow path for point: [1, 0, 1.5]
3. Follow path for point: [1, 0, 10]
4. Follow path for point: [1, -5, 10]
5. Follow path for point: [0, -5, 1.5]
6. Follow path for point: [0, -5, 1]

The python code for this mission can be found in the appendix A.3

5.5.3 Behavior Go To Point in Occupancy Grid

This is the centre key of the navigation system, providing the most rich, autonomous behavior. The mission is as simplified version of the one made for the real boiler inspection, but fitted to the real flight scenario at hand. It's outline is presented below:

1. Take Off
2. Go to 1.5 meters height ([0, 0, 1.5])
3. Go to 1 meter to the front, maintaining the altitude. ([1, 0, 1.5])
4. Go to 10 meters height, maintaining the same distance to the wall. ([1, 0, 10])
5. Go to 5 meters to the right, keeping the same distance and altitude. ([1, -5, 10])
6. Go to 1 meter away from the wall, maintaining the same altitude. ([0, -5, 1.5])

7. Go to 1 meter height, maintaining the same distance to the wall. ([0, -5, 1])

8. Land

Please, find all the python code for this mission attached in the appendix A.1.

5.5.4 Simulation

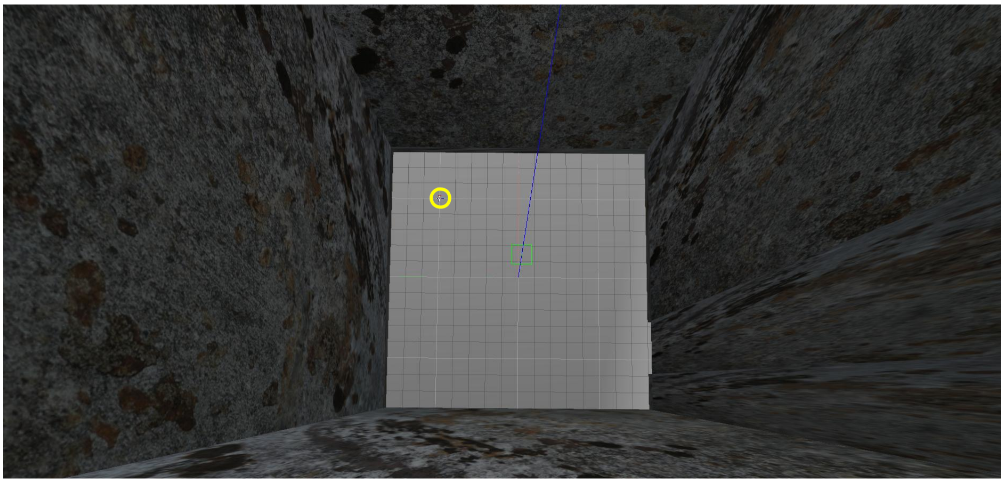

FIGURE 5.1: Simulated boiler. 57 meters tall by 16 meters (square section). The UAV is enclosed in the yellow circle.

For the simulation we created a blender model of a boiler and exported it to Gazebo, then we created a RotorS enabled Gazebo world with the hummingbird drone model. The dimensions of the simulated boiler are listed in table 5.1. To get an idea of the proportions of the boiler with the aircraft see figure 5.1. The drone has been outlined in yellow, it almost unseeable.

Simulated boiler dimensions	
Width $\times$ Depth $\times$ Height	$16 \times 16 \times 57$

TABLE 5.1: Simulation computer specifications

The specifications of the computer employed for all the simulations are depicted in the following table 5.2:

component	*value*
Ram	DDR 4 32 GB
Processor	3.4 Ghz 8 cores
GPU	GTX 1050 Ti

TABLE 5.2: Simulation computer specifications

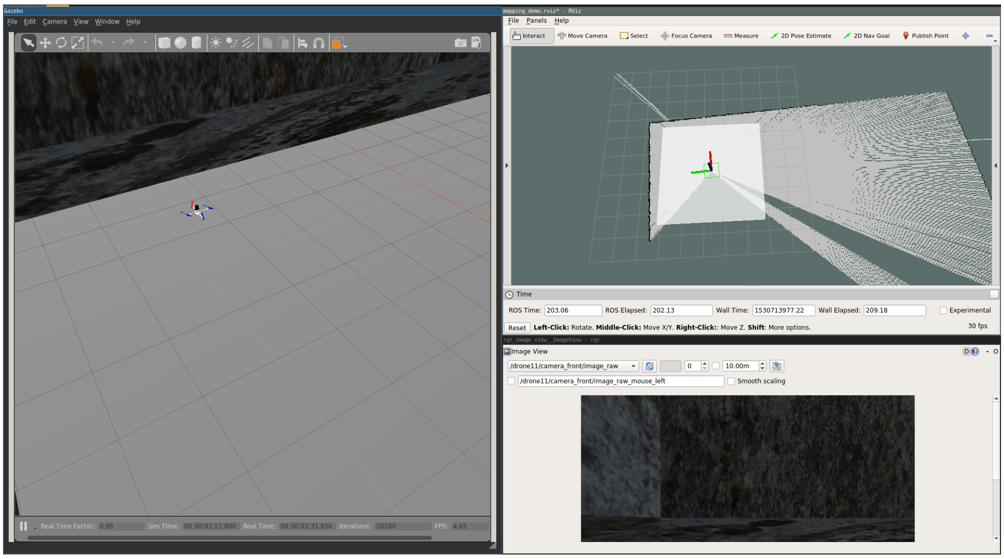

FIGURE 5.2: Gazebo and Rviz. Simulation visualization during the execution of the mission. Left: Gazebo World, top-right: Rviz lidar measures, bottom-right: Hummingbird front camera

In this scenario, performance does not comprise a problem because with enough GPU, the simulation can run smoothly and all processes can run with good memory support.

Figure 5.2 shows the simulated world running inside gazebo, along with the camera output and a visualization of the current lidar map.

5.5.5 Real Flight

For the real flight tests we used the sports centre in the School of Industrial Engineers, which is a closed space that can serve for our purposes, it's dimensions are listed in table 5.3

Sports Centre Dimensions	
Width × Depth × Height	$10 \times 25 \times 14$

TABLE 5.3: Dimensions of the sports centre used for real flight tests. School of Industrial Engineers

The chosen drone ships a DJI Manifold micro computer for onboard computation ([2]). Its technical details are contained in table 5.4

Onboard computation is the better option in this case because there is no *auto pilot* (fallback driver controller, shipped in many drones to do automatic hover when no orders are received) and given the distances it can travel and the altitude, ensuring WiFi coverage is difficult. Hence, the most secure option is to load all the necessary

component	*value*
Ram	DDR 3 2 GB
Processor	2.5 Ghz 4 cores
GPU	NVIDIA Kepler GeForce

Table 5.4: Onboard computer specification

software inside the onboard computer and send just a few orders from the ground control station. More specifically, launch the Aerostack and the python mission.

The manifold computer is not very powerful so special attention must be put on performance , if computing power drains it can be catastrophic. To aid in this situation a human pilot was prepared to take control during all the tests, although at the end it was not necessary.

5.6 Experimental Results

This section will explain the results obtained, we will first go through the simulation results and continue with the real flight ones. For each behavior, tables [ref tables] show: the correct execution of the behavior, the mean and standard deviation execution time for all points and the total time of the mission, also, the last row shows the averaged scores and times. Everything is measured in minutes and each experiment is run ten times, also, the timeout counter for each behavior is set to 4 minutes.

Generate Path in Occupancy Grid			
Test Number	*Correct*	*Point time*	*Total time*
1	6/6	0.21 (± 0.01)	1.25
2	6/6	0.20 (± 0.00)	1.20
3	6/6	0.20 (± 0.00)	1.20
4	6/6	0.20 (± 0.00)	1.20
5	6/6	0.20 (± 0.00)	1.20
6	6/6	0.20 (± 0.00)	1.20
7	6/6	0.20 (± 0.00)	1.20
8	6/6	0.20 (± 0.00)	1.20
9	6/6	0.20 (± 0.00)	1.20
10	6/6	0.20 (± 0.00)	1.20
#	*Total Correct*	*Avg. Total Point time*	*Avg. Total time*
-	60/60 (100%)	0.20 (± 0.00)	1.20 (± 0.01)

Table 5.5: Results from *generate path* behavior, run accross 10 tests. Correct executions, averaged execution time for all 6 points and total execution of the mission. Measured in minutes

Table 5.5 shows the results for the behavior *generate path in occupancy grid*. Since this behavior does not perform any motion, it is just planning, a 100% hits seems reasonable, it means that the planner does its job correctly, both the move base planner and the path planner modules work correctly, also, the behavior works as expected, generating all the requested paths. Furthermore, the times employed to generate the trajectories are very stable, which meets our requirements.

Follow Path in Occupancy Grid			
Test Number	*Correct*	*Point time*	*Total time*
1	4/6	1.76 (± 1.60)	11.24
2	2/6	2.78 (± 1.84)	17.37
3	4/6	1.93 (± 1.66)	12.24
4	3/6	2.32 (± 1.57)	14.40
5	2/6	2.69 (± 1.66)	16.76
6	4/6	1.95 (± 1.65)	10.86
7	6/6	0.95 (± 0.77)	6.26
8	6/6	1.10 (± 0.90)	7.16
9	3/6	2.47 (± 1.67)	15.28
10	5/6	1.70 (± 1.34)	10.48
#	*Total Correct*	*Avg. Total Point time*	*Avg. Total time*
-	39/60 (0.65%)	1.97 (± 0.59)	12.21 (± 3.58)

Table 5.6: Results from *Follow Path* behavior, run accross 10 tests. Correct executions, averaged execution time for all 6 points and total execution of the mission. Measured in minutes

This behavior had the worst performance of all, although it scores for 65% of the points, it has a lot of variability, this may be due to the fact that it executes the motions blindly, no logic is applied to recalculate paths when needed. It just executes the given trajectory. This behavior is slighty more complicated than the previous one in the sense that it has to execute motions, which increases the difficulty.

We acknowledged various fault causes, the very first one is estimation, although some points where not perfectly matched, the end positions where very close to the goal. This drift in localization arises from the lack of fine tuning, the localization EKF is not completely tuned for simulatation or the UAV. Also, during the tests we observed that the trajectory controller has some weird behaviors. There were various cases where the orders took no effect. Taking a look over the memory consumption clarified the assumptions that there are some conditions that cause the controller to hang in an intensive loop. As of the time of writing, the controller is being remade from the ground up.

Not matching the target points means that the behavior does not finish and it only stops when the timeout is reached. This faulty finish condition, in turn adds up to the execution time, which explains some of the large times of execution. Tests 2, 4, 5 and 9 accounts for this fact, they score the least but also last the most. Interestingly enough, this test has more fully completed missions that the one of *go to point*, 2 out of 10 versus 1 out time, but in general has more faulty points, rising the global time of execution for the whole test.

Go to Point in Occupancy Grid			
Test Number	*Correct*	*Point time*	*Total time*
1	4/6	1.78 (± 1.74)	11.21
2	3/6	2.42 (± 1.64)	15.07
3	3/6	2.43 (± 1.66)	15.13
4	4/6	1.72 (± 1.69)	10.85
5	3/6	2.40 (± 1.66)	14.91
6	5/6	1.36 (± 1.29)	8.68
7	4/6	1.71 (± 1.69)	10.77
8	5/6	1.37 (± 1.33)	8.75
9	6/6	0.79 (± 0.77)	5.28
10	4/6	2.19 (± 1.57)	13.68
#	*Total Correct*	*Avg. Total Point time*	*Avg. Total time*
-	41/60 (0.68%)	1.82 (± 0.52)	11.43 (± 3.12)

TABLE 5.7: Results from *Go to Point* behavior, run accross 10 tests. Correct executions, averaged execution time for all 6 points and total execution of the mission. Measured in minutes

This behavior is most complex, but also more stable than the previous one, it contains some logic to handle obstacles and track the drone position. Nevertheless, the bad matching of points is a problem, this is the case of the 2nd, 3rd and 5th tests, which fail to reach half of the points, rising the execution time to 15 minutes at worst, the correlation is clear. We strengthen our assumption with tests 8th and 9th, that amounts for 5 out of 6 and 6 out 6 goals reached, respectively, with the lowest times of all the tests. The implemented behavior works correctly in 68% of the cases, which is not bad given the complexity in coordination needed to accomplish the task and the short time employed implementing it.

The last test in our experiments are run in real flight, with the previously explained drone and environment. Due to time constraints and both hardware and pilot availability, we ran only 3 tests, the results are reported in table 5.8

It is worth noting that these tests are much more robust and fast, the intuition behind these results is that the localization module is widely tested and tuned for this specific aircraft which makes sense given that it is used for industrial level real flight missions. Also, the speed of this drone is comparably higher.

Real Flight - Go to Point behavior			
Test Number	*Correct*	*Point time*	*Total time*
1	4/6	1.26 (± 0.92)	7.91
2	6/6	0.71 (± 0.37)	4.78
3	6/6	0.84 (± 0.31)	5.24
#	*Total Correct*	*Avg. Total Point time*	*Avg. Total time*
-	16/18 (0.88%)	0.94 (± 0.23)	5.97 (± 1.38)

Table 5.8: Results from *Go to Point* behavior in real flight, run accross 3 tests. Correct executions, averaged execution time for all 6 points and total execution of the mission. Measured in minutes

Although these tests were run with a human operator fallback, just in case something went wrong, none of the tests required it. Even in the first test, where two points failed to match, the system recovered and successfully finished the mission. Furthermore, the two failed points where encountered when reaching the highest altitude and when moving away from the wall at highest altitude, points for and six, respectively (refer to 5.5.3 for detailed coordinates). Our best guess for this performance is based on the environment: the sports centre walls are made of various materials, it has about 8-9 meters of solid wall and then a grid gate with holes to the roof. We believe that the holes in the grid gate tricked the lidar based localization, which in turn caused the go to point not to match the target point. Also, as the localization is made out of the fusion of various sensor inputs, this phenomenon does not happen always, which is another proof of the robustness of the system.

[ToDo := Add speeds of each drone?]

5.7 Discussion

The results obtained shed some light over the problems that arise in an autonomous navigation system, clarifying the need for more tests and a better prepared simulation environment.

First, the environment used for simulation is very well suited for our needs, but lacks some tuning. The fact that there exists such difference between the real flight mission and the simulated one is not good, although clearly improvable through extensing tuning and testing. Even though, the proposed system works reasonably well for the time invested to implementing and also highlights that we are on the correct track for future experiments.

This chapter presented all the validation missions and environment employed for testing all the implemented behaviors, also the results were presented and analyzed for discussion. The next chapter will conclude the thesis adding more in depth thoughts about the results obtained and further research.

Chapter 6

Conclusions & Further Research

This chapter will conclude the thesis with our conclusions on the conducted experiments and the proposed system and expose the open lines of study we would like to target our research to.

6.1 Conclusions

At the sight of the results obtained it is clear for us that we are on the right track towards a fully autonomous navigation system. We would like to stress out that, as far as the navigation system is supported by an aerial architecture, it depends upon its correct functioning. In fact, in our experiments, the biggest source of failure was the slam module, although the employed technique is very advanced and stands over the state of the art papers, it fails in some situations, propagating the error above until higher levels of abstraction: the Behavioral system, where we implemented the whole navigation system.

In this thesis we delved into different high level abstractions for the construction of a fully autonomous navigation system, we proposed a system based on the most up to date theory in planning and localization, we prepared some experiments to test our research hypotheses and implemented all the behaviors it is composed of. Also, we counted on the necessary resources both in hardware and in time to conduct all the tests, and at the end, we proved that the Aerostack system is the perfect testbed for artificial intelligence algorithms aimed at aerial robotics.

6.2 Further Research

It is only through extensive and stressful tests that a system like this can prove useful and bullet proof. In the future, we would like explore the robustness and reliability of the proposed system under the conditions of more difficult environments, as well as to test it on a real life, business oriented mission, which is where the inspiration for this whole work.

After all the tests conducted, it is clear that the localization subsystem is the key to more reliable, successful intelligent behaviors, is the basement for higher levels of abstraction. We would like to strengthen these lower level algorithms first in order to improve the base system and open the door for more intelligent, autonomous behaviors, maybe in the future more complex missions can be achieved with this system based solely on a few sources of data.

Bibliography

[1] Piyush Khandelwal Christian Connette Bhaskara Marthi. *Elastic Band Non holonomic planner*. [Online; accessed 25-June-2018]. 2018. URL: http://wiki.ros.org/eband_local_planner.

[2] DJI. *DJI Manifold High-performance embedded computer*. [Online; accessed 25-June-2018]. 2018. URL: https://www.dji.com/es/manifold.

[3] DJI. *DJI Matrice 100 Quadcopter for developers*. [Online; accessed 25-June-2018]. 2018. URL: https://www.dji.com/es/matrice100.

[4] Fadri Furrer et al. "Robot Operating System (ROS)". In: *Studies Comp.Intelligence Volume Number:625* The Comple.978-3-319-26052-5 (2016), Chapter 23.

[5] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. "Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters". In: *IEEE Transactions on Robotics* 23.1 (Feb. 2007), pp. 34–46. ISSN: 1552-3098. DOI: 10.1109/TRO.2006.889486. URL: http://ieeexplore.ieee.org/document/4084563/.

[6] J.E. Guivant and E.M. Nebot. "Optimization of the simultaneous localization and map-building algorithm for real-time implementation". In: *IEEE Transactions on Robotics and Automation* 17.3 (June 2001), pp. 242–257. ISSN: 1042296X. DOI: 10.1109/70.938382. URL: http://ieeexplore.ieee.org/document/938382/.

[7] L E Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580. ISSN: 1042-296X. DOI: 10.1109/70.508439.

[8] S Kohlbrecher et al. "A Flexible and Scalable SLAM System with Full 3D Motion Estimation". In: *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE. Nov. 2011.

[9] Steven M Lavalle. *Rapidly-Exploring Random Trees: A New Tool for Path Planning*. Tech. rep. 1998.

[10] John J. Leonard, John J. Leonard, and Hans Jacob S. Feder. "A computationally efficient method for large-scale concurrent mapping and localization". In: (2000), pp. 169–176. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.122.4705.

[11] Eitan Marder-Eppstein. *Move Base Robotic planner*. [Online; accessed 25-June-2018]. 2018. URL: http://wiki.ros.org/move_base.

[12] Michael Montemerlo et al. "FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem". In: *IN PROCEEDINGS OF THE AAAI NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE* (2002), pp. 593–598. URL: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.2153.

[13] Montiel J M M Mur-Artal Raúl and Juan D Tardós. "ORB-SLAM: a Versatile and Accurate Monocular SLAM System". In: *IEEE Transactions on Robotics* 31.5 (2015), pp. 1147–1163. DOI: 10.1109/TRO.2015.2463671.

[14] S Quinlan and O Khatib. "Elastic bands: connecting path planning and control". In: *[1993] Proceedings IEEE International Conference on Robotics and Automation*. May 1993, 802–807 vol.2. DOI: 10.1109/ROBOT.1993.291936.

[15] Francisco J. Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. "Speeded up detection of squared fiducial markers". In: *Image and Vision Computing* 76 (2018), pp. 38–47. ISSN: 0262-8856. DOI: https://doi.org/10.1016/j.imavis.2018.05.004. URL: http://www.sciencedirect.com/science/article/pii/S0262885618300799.

[16] J L Sanchez-Lopez et al. "AEROSTACK: An architecture and open-source software framework for aerial robotics". In: *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*. 2016, pp. 332–341. DOI: 10.1109/ICUAS.2016.7502591.

[17] Jose Luis Sanchez-Lopez et al. "A Multi-Layered Component-Based Approach for the Development of Aerial Robotic Systems: The Aerostack Framework". In: *Journal of Intelligent & Robotic Systems* 88.2 (2017), pp. 683–709. ISSN: 1573-0409. DOI: 10.1007/s10846-017-0551-4. URL: https://doi.org/10.1007/s10846-017-0551-4.

[18] Bruno Siciliano and Oussama Khatib. "Springer Handbook of Robotics". In: 2nd. Springer Publishing Company, Incorporated, 2016. Chap. 46, pp. 1153–1175. ISBN: 3319325507, 9783319325507.

[19] Gazebo Sim. *Gazebo Simulator*. [Online; accessed 25-June-2018]. 2018. URL: http://gazebosim.org/.

[20] Ascending Technologies. *AscTec Hummingbird*. [Online; accessed 25-June-2018]. 2018. URL: http://www.asctec.de/en/uav-uas-drones-rpas-roav/asctec-hummingbird/.

[21] Opti Track. *Opti Track Product*. [Online; accessed 25-June-2018]. 2018. URL: http://www.optitrack.com/.

[22] Vision4UAV. *Aerostack Wiki*. [Online; accessed 25-June-2018]. 2018. URL: https://github.com/Vision4UAV/Aerostack/wiki.

[23] Wikipedia contributors. *Trilateration — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-June-2018]. 2018. URL: https://en.wikipedia.org/w/index.php?title=Trilateration&oldid=847430880.

Appendix A

Appendix A

A.1 Go to Point test mission

```
#!/usr/bin/env python2

import executive_engine_api as api
import os.path as path
import time
import rospy

from std_msgs.msg import String
from std_srvs.srv import Empty

# reset sequence is:
# 1. LAND
# 2. Reset world: /gazebo/reset_world ""
# 3. Reset planner: /drone11/move_base/syscommand "reset"
# 4. SLAM
def reset():
  # rospy.init_node('reset')
  print('Reset sequence started...')
  uid = api.executeBehavior('LAND')
  print('-> Landed')
  gazebo_pub = rospy.ServiceProxy('/gazebo/reset_world', Empty)
  gazebo_pub.call()
  print('-> Gazebo worl reset')
  planner_pub = rospy.Publisher('/drone11/move_base/syscommand', String, queue_size=1)
  planner_pub.publish(String('reset'))
  print('-> Planner reset')
  activated, uid = api.activateBehavior('SELF_LOCALIZE_AND_MAP_BY_LIDAR')
  print('-> SLAM')
  rospy.sleep(0.5)
  print('... done')

behavior_uids = {}
behavior_names = {
  'go_to_point' : 'GO_TO_POINT_IN_OCCUPANCY_GRID',
```

```
  'slam'        : 'SELF_LOCALIZE_AND_MAP_BY_LIDAR',
  'take_off'    : 'TAKE_OFF',
  'land'        : 'LAND'
}

def dump(test_no, dir_path, prefix, store):
  file_path = path.abspath('{}/{}_{}'.format(dir_path, prefix, test_no))
  print('dumping {} to {}_{}'.format(prefix, prefix, test_no))
  f = open(file_path, 'w+')
  f.writelines(list(map(lambda x: str(x), store)))
  f.close()

def run_mission(coordinates='absolute', points=[]):
  print('Starting mission...')
  activated, uid = api.activateBehavior(behavior_names['slam'])
  if not activated:
    raise Exception('Unable to continue without SLAM')
  rospy.sleep(0.2)
  print('-> take off')
  result = api.executeBehavior(behavior_names['take_off'])
  print('-> result {}'.format(result))
  times = []
  fails = 0
  for point in points:
    print('-> go to point {}'.format(str(point)))
    start = time.time()
    result = api.executeBehavior(behavior_names['go_to_point'], coordinates=point)
    end = time.time()
    elapsed = end - start
    times.append(elapsed)
    print('-> result {}'.format(result))
    if result != 'GOAL_ACHIEVED':
      fails += 1

  print('-> land')
  result = api.executeBehavior(behavior_names['land'])
  print('-> result {}'.format(result))
  print('Finish mission...')
  return times, fails

def runMission():
  data = {
    "coordinates": "absolute",
    "points": [
      [0, 0, 1.5],
      [1, 0, 1.5],
      [1, 0, 10],
      [1, -5, 10],
      [0, -5, 10],
      [0, -5, 1.5]
    ]
  }
  tests = 10
  mission_times = []
  dir_path = '/root/workspace/ros/aerostack_catkin_ws/src/aerostack_stack/launchers/tfm_guillerm
```

```
  print('Start dump dir: {}'.format(dir_path))
  for test_no in range(tests):
    print('######## TEST {} ########'.format(test_no))
    # start measuring time
    start = time.time()
    point_times, fails = run_mission(**data)
    # end measuring time
    end = time.time()
    elapsed = end - start
    print('Completed mission in {} secods with {} fails'.format(elapsed, fails))
    mission_times.append(elapsed)
    dump(test_no=test_no, dir_path=dir_path, prefix='point_fails', store=[fails])
    dump(test_no=test_no, dir_path=dir_path, prefix='point_times', store=point_times)
    reset()
    print('########  DONE  ########'.format(test_no))
  dump(test_no=0, dir_path=dir_path, prefix='global_times', store=mission_times)
```

A.2 Generate Path test mission

```
#!/usr/bin/env python2

import executive_engine_api as api
import os.path as path
import time
import rospy

from std_msgs.msg import String
from std_srvs.srv import Empty

# reset sequence is:
# 1. LAND
# 2. Reset world: /gazebo/reset_world ""
# 3. Reset planner: /drone11/move_base/syscommand "reset"
# 4. SLAM
def reset():
  # rospy.init_node('reset')
  print('Reset sequence started...')
  uid = api.executeBehavior('LAND')
  print('-> Landed')
  gazebo_pub = rospy.ServiceProxy('/gazebo/reset_world', Empty)
  gazebo_pub.call()
  print('-> Gazebo worl reset')
  planner_pub = rospy.Publisher('/drone11/move_base/syscommand', String, queue_size=1)
  planner_pub.publish(String('reset'))
  print('-> Planner reset')
  activated, uid = api.activateBehavior('SELF_LOCALIZE_AND_MAP_BY_LIDAR')
  print('-> SLAM')
  rospy.sleep(0.5)
  print('... done')

behavior_uids = {}
behavior_names = {
```

```
  'generate' : 'GENERATE_PATH_IN_OCCUPANCY_GRID',
  'slam'     : 'SELF_LOCALIZE_AND_MAP_BY_LIDAR',
}

def get_paths(round_n, n_tests, path_store):
  for test_no in range(1, n_tests + 1):
    path_no = round_n * 10 + test_no
    print('ask for path', path_no)
    query = 'path({},?y)'.format(path_no)
    success, unification = api.consultBelief(query)
    path = '(FAIL)' if not success else unification['y']
    path_store.append(path + '\n')

def dump(test_no, dir_path, prefix, store):
  file_path = path.abspath('{}/{}_{}'.format(dir_path, prefix, test_no))
  print('dumping {} to {}_{}'.format(prefix, prefix, test_no))
  f = open(file_path, 'w+')
  f.writelines(list(map(lambda x: str(x) + '\n', store)))
  f.close()

def run_mission(points=[]):
  print('Starting mission...')
  activated, uid = activated, uid = api.activateBehavior(behavior_names['slam'])
  if not activated:
    raise Exception('Unable to continue without SLAM')
  rospy.sleep(0.2)
  times = []
  fails = 0
  for point in points:
    print('-> generate path to point {}'.format(str(point)))
    start = time.time()
    result = api.executeBehavior(behavior_names['generate'], coordinates=point)
    end = time.time()
    elapsed = end - start
    times.append(elapsed)
    print('-> result {}'.format(result))
    if result != 'GOAL_ACHIEVED':
      fails += 1

  print('Finish mission...')
  return times, fails

def runMission():
  data = {
    "points": [
      [0, 0, 1.5],
      [1, 0, 1.5],
      [1, 0, 10],
      [1, -5, 10],
      [0, -5, 10],
      [0, -5, 1.5]
    ]
  }
  tests = 10
  mission_times = []
```

```
  dir_path = '/root/workspace/ros/aerostack_catkin_ws/src/aerostack_stack/launchers/tfm_guillerm
  print('Start dump dir: {}'.format(dir_path))
  for test_no in range(tests):
    print('######## TEST {} ########'.format(test_no))
    # start measuring time
    start = time.time()
    point_times, fails = run_mission(**data)
    # end measuring time
    end = time.time()
    elapsed = end - start
    print('Completed mission in {} secods with {} fails'.format(elapsed, fails))
    mission_times.append(elapsed)
    path_store = []
    # get_paths(test_no, tests, path_store)
    # dump(test_no=test_no, dir_path=dir_path, prefix='paths', store=path_store)
    dump(test_no=test_no, dir_path=dir_path, prefix='point_fails', store=[fails])
    dump(test_no=test_no, dir_path=dir_path, prefix='point_times', store=point_times)
    print('########  DONE  ########'.format(test_no))
  dump(test_no=0, dir_path=dir_path, prefix='global_times', store=mission_times)
```

A.3 Follow Path test mission

```
#!/usr/bin/env python2

import executive_engine_api as api
import sys
import json
import os.path as path
import time
import rospy

from std_msgs.msg import String
from std_srvs.srv import Empty

# reset sequence is:
# 1. LAND
# 2. Reset world: /gazebo/reset_world ""
# 3. Reset planner: /drone11/move_base/syscommand "reset"
# 4. SLAM
def reset():
  # rospy.init_node('reset')
  print('Reset sequence started...')
  uid = api.executeBehavior('LAND')
  print('-> Landed')
  gazebo_pub = rospy.ServiceProxy('/gazebo/reset_world', Empty)
  gazebo_pub.call()
  print('-> Gazebo worl reset')
  planner_pub = rospy.Publisher('/drone11/move_base/syscommand', String, queue_size=1)
  planner_pub.publish(String('reset'))
  print('-> Planner reset')
  activated, uid = api.activateBehavior('SELF_LOCALIZE_AND_MAP_BY_LIDAR')
  print('-> SLAM')
```

```
  rospy.sleep(0.5)
  print('... done')

behavior_uids = {}
behavior_names = {
  'take_off': 'TAKE_OFF',
  'follow'  : 'FOLLOW_PATH_IN_OCCUPANCY_GRID',
  'slam'    : 'SELF_LOCALIZE_AND_MAP_BY_LIDAR',
  'generate': 'GENERATE_PATH_IN_OCCUPANCY_GRID'
}

def get_path():
  query = 'path(0,?y)'
  success, unification = api.consultBelief(query)
  path = 'no_path' if not success else unification['y']
  if path == 'no_path':
    print('gen failed, ask again')
    return None
  api.assertBelief('path(0, no_path)', multivalued=False)
  return path


def dump(test_no, dir_path, prefix, store):
  file_path = path.abspath('{}/{}_{}'.format(dir_path, prefix, test_no))
  print('dumping {} to {}_{}'.format(prefix, prefix, test_no))
  f = open(file_path, 'w+')
  f.writelines(list(map(lambda x: str(x) + '\n', store)))
  f.close()

def run_mission(paths=[]):
  print('Starting mission...')
  activated, uid = api.activateBehavior(behavior_names['slam'])
  if not activated:
    raise Exception('Unable to continue without SLAM')
  rospy.sleep(0.2)
  result = api.executeBehavior(behavior_names['take_off'])
  rospy.sleep(1)
  max_tries = 5
  times = []
  full_test_elapsed = 0
  fails = 0
  for path in paths:
    str_path = None
    tries = 0
    print('-> follow path to point {}'.format(str(path)))
    while str_path is None and tries < max_tries:
      print('-> get a valid path tries {}'.format(str(tries)))
      result = api.executeBehavior(behavior_names['generate'], coordinates=path)
      str_path = get_path()
      print('-> got path back')
      tries += 1
    start = time.time()
    result = api.executeBehavior(behavior_names['follow'], path=str_path)
    end = time.time()
    elapsed = end - start
```

```
    times.append(elapsed)
    full_test_elapsed += elapsed
    print('-> result {}'.format(result))
    if result != 'GOAL_ACHIEVED':
      fails += 1

  print('Finish mission...')
  return full_test_elapsed, times, fails

def runMission():
  data = {
    "paths": [
      [0, 0, 1.5],
      [1, 0, 1.5],
      [1, 0, 10],
      [1, -5, 10],
      [0, -5, 10],
      [0, -5, 1.5]
    ]
  }
  tests = 5
  mission_times = []
  dir_path = '/root/workspace/ros/aerostack_catkin_ws/src/aerostack_stack/launchers/tfm_guillerm
  print('Start dump dir: {}'.format(dir_path))
  for test_no in range(tests):
    print('######## TEST {} ########'.format(test_no))
    # start measuring time
    elapsed, path_times, fails = run_mission(**data)
    # end measuring time
    print('Completed mission in {} secods with {} fails'.format(elapsed, fails))
    mission_times.append(elapsed)
    dump(test_no=test_no, dir_path=dir_path, prefix='point_fails', store=[fails])
    dump(test_no=test_no, dir_path=dir_path, prefix='path_times', store=path_times)
    reset()
    print('########  DONE  ########'.format(test_no))
  dump(test_no=0, dir_path=dir_path, prefix='global_times', store=mission_times)
```