

# Designing Fuzzy Logic Controllers for ROS-Based Multirotors

Emanoel Koslosky, André Schneider de Oliveira,  
Marco Aurélio Wehrmeister and João Alberto Fabro

**Abstract** This chapter presents a tutorial on using an open-source ROS package for implementing control systems based on Fuzzy Logic. Such a package has been created to facilitate the development of fuzzy control systems along with ROS technology and infrastructure. A step-by-step tutorial discusses how to develop a set of distributed and interconnected fuzzy controllers using the proposed ROS package. A fuzzy control system that controls the movement of an unmanned multirotor (specifically a hexacopter) is presented as case study. The behavior of this control system is demonstrated by means of a commercial robotics simulation environment named V-REP. One scenario is used to illustrate the fuzzy control system steering the movement of a virtual hexacopter carrying an attached loose payload, i.e. such a loose payload forms a pendulum. In this case study, one can see the hexacopter flight after receiving commands to fly to distinct positions within the scenario. It is important to highlight that, in order to be able to perform this tutorial, the reader must use ROS Indigo Igloo and V-REP PRO EDU version V3.3.0 both running on Ubuntu 14.04.4 LTS.

**Keywords** ROS · Multirotor · Fuzzy logic · Simulation

---

The source code and examples discussed in this chapter are available as a catkin package published in [1]

---

E. Koslosky (✉) · A.S. de Oliveira · M.A. Wehrmeister · J.A. Fabro  
Advanced Laboratory of Embedded Systems and Robotics (LASER), Federal University of Technology—Parana (UTFPR), Av. Sete de Setembro 3165, Curitiba 80230-901, Brazil  
e-mail: [ekosky@gmail.com](mailto:ekosky@gmail.com)

A.S. de Oliveira  
e-mail: [andreoliveira@utfpr.edu.br](mailto:andreoliveira@utfpr.edu.br)

M.A. Wehrmeister  
e-mail: [wehrmeister@utfpr.edu.br](mailto:wehrmeister@utfpr.edu.br)

J.A. Fabro  
e-mail: [fabro@utfpr.edu.br](mailto:fabro@utfpr.edu.br)

# 1 Introduction

Recent technology advances have led to a cost reduction in electronic and electromechanical components, providing new capabilities to small electromechanical aircrafts such as multirotor helicopters (also known as drones). Such devices are being applied in many distinct application fields, such as video recording, plantation inspections, search-and-rescue assistance, military and civil surveillance applications, among others. Some of these new applications demand multirotor helicopters that fly autonomously as presented in [2, 3]. For that, additional computing systems must be embedded into an autonomous multirotor helicopter, in addition to movement and stabilization control systems, in order to provide higher level capabilities to support the mission accomplishment. Unmanned Aerial Vehicles (UAV) are the preferred choice for these applications due to cost reductions obtained from eliminating the need of high-skilled and trained pilots. It is important to highlight that, in this text, the term “*multirotor*” is used as a synonym for “*multirotor helicopter*”.

The multirotor rotors can be organized in different ways, varying in the amount of rotors, as well as their positions onto the aircraft frame. The so-called quadcopter is a multirotor equipped with four rotors. It is the most common multirotor. However, its characteristic may limit some applications, e.g. a payload transportation from one point to another. Recently, other multirotor topologies have become popular such the hexacopter [4] that is equipped with six rotors.

In order to provide a stable flight for UAVs, hybrid control approaches (parallel, cascade) with multiple PID controllers are commonly used [5, 6]. Although these methods perform the system control in a proper way, they require a precise mathematical formulation as well as the identification of UAV dynamics, in order to stabilize the system while minimizing disturbances [7].

Adaptive algorithms can be applied to control multivariable systems (such as UAV flying control system) more efficiently than classical strategies. In [8], an approach based on artificial neural networks is presented to control the trajectory of UAV flight. A genetic algorithm is applied to control the flight of a hexacopter in [9], where as a fuzzy logic method has been proposed to control the position of a hexacopter in [10]. The main focus of these previous works is on the UAV stabilization in the presence of linear disturbances. However, these works do not consider nonlinear disturbances, such as the ones introduced when the UAV carries a variable or loose payload. In a previous work [11], we created a fuzzy logic controller to control the movements of a hexacopter and also to deal with nonlinear disturbances. Such a fuzzy control system was created to provide a robust and flexible controller that is able to keep the hexacopter stability when moving or hovering, even when it carries a free or loose payload that changes its center of gravity. It is important to highlight that, due to space constraints, this chapter does not provide a in-depth discussion on this fuzzy control system. Interested readers are referred to [11] in order to obtain details on the design of such a control system.

This chapter discusses a ROS-based implementation of our fuzzy control system in terms of an open-source ROS-based fuzzy logic library designed to control

multirotors. Specifically, the proposed fuzzy library has been implemented to be used within a `roscpp` Node. The main goal is to present a step-by-step tutorial on designing fuzzy based controllers for mobile robots (focusing on UAVs) using ROS features [12]. This tutorial is intended to be followed by beginner level ROS users. It discusses how ROS is used to receive signals from sensors and also to send commands to actuators by means of the publisher/subscriber mechanism. Moreover, the tutorial shows how to integrate a different robot simulator named V-REP [13] with the fuzzy control system implemented with the proposed library. Thus, the engineers may perform a round-trip engineering process<sup>1</sup> by integrating the developed fuzzy control system with a virtual environment or the real hardware seamlessly.

It is important to highlight that, as already mentioned, the tutorial described in this chapter is aimed to beginner level users of ROS. Thus, in order to correctly understand its content, the reader should be familiar with Linux and C/C++ programming language, as well as he/she should have some basic ROS understanding. The beginner level tutorials [12] should be sufficient to understand the presented approach (tutorial #16 presents the concepts about nodes, messages, publishers and subscribers). Although it might be a good idea to follow the V-REP BubbleRob tutorial [14], the short introduction given here would be enough for allowing the experimentation. Moreover, a tutorial about V-REP and ROS integration is presented in [15]. The versions of the software used in this tutorial are: (i) Operating System: Ubuntu 14.04.4 LTS; (ii) ROS Indigo Igloo; (iii) V-REP PRO EDU version V3.3.0.

The remainder of this chapter is organized as follows. Section 2 presents a brief overview of multirotors features and movements. Section 3 summarizes our fuzzy control system for a hexacopter. Section 4 presents the open-source ROS library that provides the services supplied within our fuzzy logic library. Section 5 discusses how to use a different robotics simulation environment along with ROS and our library to perform experiments on fuzzy control system, including the discussion of a case study that illustrates the concepts and technologies discussed in this tutorial. Finally, Sect. 6 concludes this chapter by discussing some final remarks.

## 2 Brief Overview of Multirotors

This section provides a description on how multirotors perform their movements. An empirical discussion is presented rather than a formal modeling of multirotor dynamics, in order to provide a practical view similar to a human pilot controlling the multirotor by means of a radio control system (i.e. RC controller). Formal models of multirotor dynamics can be obtained, for instance, in [5–7].

Initially, it is worth mentioning that a multirotor can present various distinct configurations, i.e. multirotors may present various topologies. In summary, multirotor topology varies in rotors number as well as the rotors position. Regarding the number of rotors, a multirotor may have from three, four, six or eight rotors, namely, tricopter,

---

<sup>1</sup>This chapter does not intend to propose or discuss any concrete round-trip engineering process.

quadcopter, hexacopter and octocopter, respectively. The most common multirotor is the quadcopter, although the hexacopter is recently becoming popular due to its good trade-off among cost, flight robustness, fault-tolerance, and capacity of flying with heavier payloads.

On the other hand, these rotor may be positioned in distinct topologies regarding the front/rear of the multirotor.

- “X” topology presents two rotors on both front and rear. One rotor is positioned on the right-hand side and the other one on the left-hand side of front/rear. This topology can be used with quad-, hexa-, and octocopters.
- “I” or “+” topology presents one rotor positioned on the front and one rotor position on the rear. The remainder rotors are distributed evenly on the right-hand and left-hand sides of the multirotor. This topology can be used with quad-, hexa-, and octocopters.
- “H” topology is similar to the “X” topology, i.e. two rotors on both the front and rear. However, the arms of the aircraft frame form an “H” rather than an “X”. This topology can be used with quad- or hexacopters.
- “Y” topology presents two rotors on either front or rear, and one rotor on the opposite side. This topology can be used with tri- or hexacopters. In hexacopters, the counter rotating propellers are placed one on top of another.

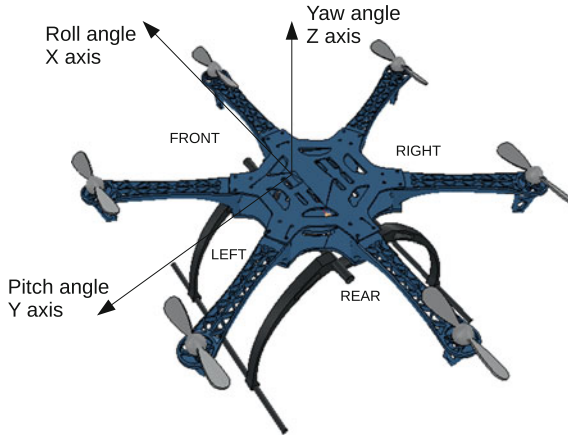
It is important to highlight that both the amount of rotors and their positioning onto the aircraft frame influence how a multirotor performs its movements, and hence, how the movement control system is designed. In the remainder of this section, the hexacopter “+” topology is used to illustrate the multirotor movements.

A multirotor moves on three dimensions along X, Y and Z axes as shown in Fig. 1. In summary, the rotors create thrust that allows for pitch, roll, yaw, uplift and downfall movements. Thus, by activating the rotors accordingly, it is possible to control the hexacopter movement along X, Y and Z axes. In other words, by speeding up or slowing down some rotors, it is possible to move the multirotor towards the desired direction on each axis.

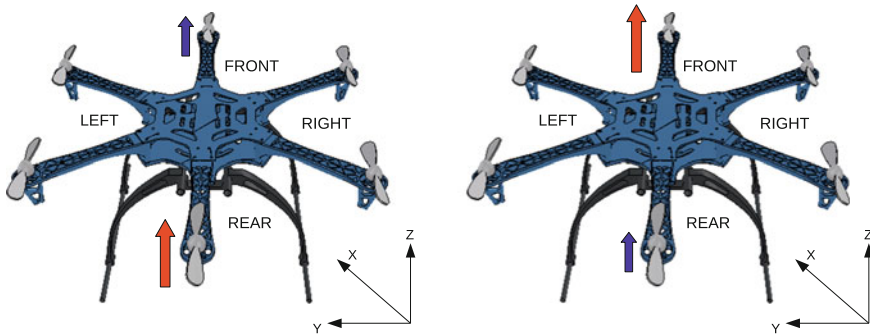
Pitch is the multirotor movement towards either forward or backward. Controlling pitch angle implies on the control of multirotor rotation<sup>2</sup> on Y-axis as shown in Fig. 2. The multirotor moves forward when the rear rotor spins faster than the front rotor; similarly, it moves backward when front rotor spins faster than the rear rotor. The difference in these rotors spinning produces an unbalanced thrust on each rotor, rotating the multirotor around its Y-axis leading to an horizontal movement along the X-axis. The multirotor slows down the movement when the rotor positioned on the movement direction spins faster than the other rotor, i.e. this decreases the pitch angle.

Roll is the multirotor movement towards right-hand or left-hand side. Controlling roll angle implies on the control of multirotor rotation (see footnote 2). on X-axis as shown in Fig. 3. The multirotor moves sideways when the rotor on one side spins faster

<sup>2</sup>Figures 2, 3, 4 and 5 depict the inertial frame at the right-bottom corner of the figures. It is important to note that this inertial frame is used in both V-REP environment and ROS representation.



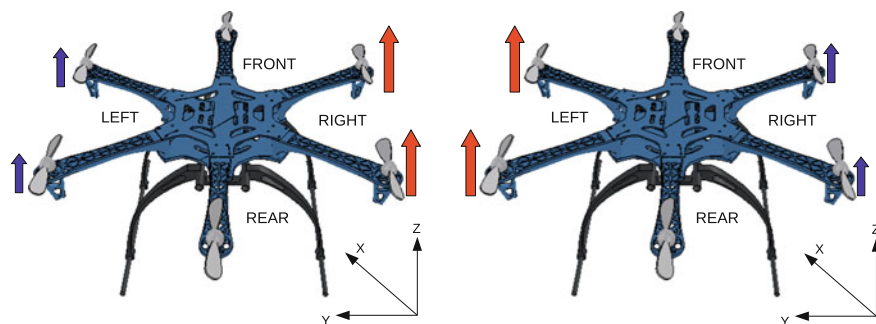
**Fig. 1** Hexacopter movements: (i) roll is the rotation on X axis; (ii) pitch is the rotation on Y axis; and (iii) yaw is the rotation on Z axis. The *arrow* indicates positive direction



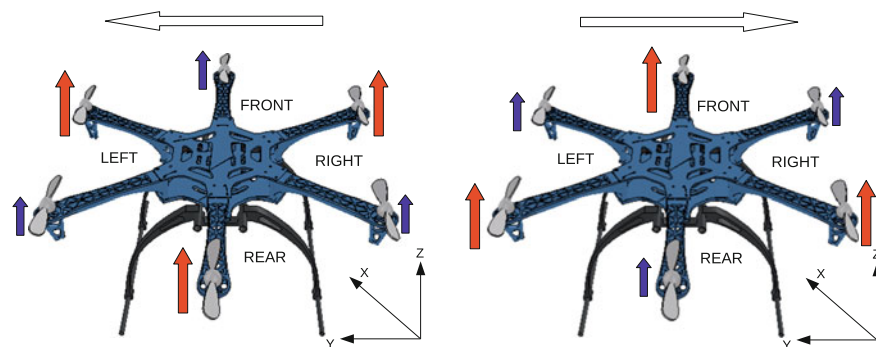
**Fig. 2** Hexacopter pitch movement: rotation around Y-axis

than the ones on the other side. The difference in these rotors spinning produces an unbalanced thrust on right-hand or left-hand side, rotating the multirotor around its X-axis leading to a horizontal movement along the Y-axis. Likewise forward/backward movement, the multirotor slows down the movement when the rotors positioned on the movement direction spins faster than the other ones, i.e. this decreases the roll angle.

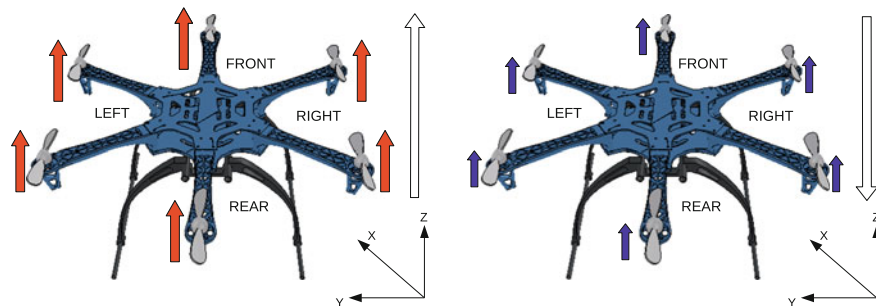
Yaw is the deviation of the multirotor head (i.e. its front orientation) towards either right or left. Controlling yaw angle implies on the control of multirotor rotation on Z-axis as shown in Fig. 4. For that, interleaved rotors must spin faster than the other ones leading to a gyroscopic effect on the multirotor frame. It is important to note that the propellers attached to interleaved motors rotate either clockwise or counterclockwise. Therefore, in order to turn the multirotor to right-hand side direction, the clockwise rotors must spin faster. Similarly, for turning to the left-hand side direction, the



**Fig. 3** Hexacopter roll movement: rotation around X-axis



**Fig. 4** Hexacopter yaw movement: rotation around Z-axis



**Fig. 5** Hexacopter uplift and downfall movements

counterclockwise rotors must spin faster. The multirotor does not move along X- or Y-axis; it only rotates around Z-axis.

Uplift and downfall are the multirotor movements related to the flight altitude. These movement are achieved by spinning all rotors on the same pace. The multirotor flies upwards when the lift force produced by the rotors is higher than the multicopter weight. On the other hand, the multirotor flies downwards when the lift

force produced by the rotors is lower than the multicopter weight. When lift force is equal to the weight, the multicopter hovers in the air.

Furthermore, in many application fields, a multirotor carries a payload in order to accomplish its mission. A payload attached to the multirotor body frame changes the gravity center of the whole aircraft thus affecting the way the multirotor performs its movements [16]. When the payload has a constant mass and is fixed to the body frame, the multirotor gravity center is modified but it remains on the same position. On the other hand, when the payload has a varying mass (e.g. leaking bag of sand) or it is loosely attached to the multirotor, the center of gravity changes during the flight, introducing nonlinear disturbances. A loosely attached payload forms a moving pendulum when the multirotor flies. Hence, while the pendulum is moving, center of gravity of the entire aircraft changes as well [17]. Fuzzy logic controllers have been used to deal with the moving center of gravity created by a moving pendulum [17–19].

Finally, it is worth mentioning that a multirotor is equipped with a set of sensors and actuators in order to run high- and low-level control systems. An Inertial Measurement Unit (IMU) provides a combination of gyroscope, accelerometer and compass (magnetometer) sensors. Multirotors demand three dimensional IMUs. The accelerometer detects the current acceleration rate along with X, Y and Z axes, whereas changes in rotational attributes, e.g. roll, pitch and yaw, are measured by the gyroscope. The gyroscope provides the body frame angles known as Euler angles. The magnetometer measures the magnetic field in order to assist the calibration against orientation drift. In order to obtain the absolute position of the multirotor within the environment, the Global Positioning System (GPS) sensor are used. GPS is also applied to decrease errors in position and velocity produced within an inertial navigation system. Stereo cameras or laser scanners can also be used to obtain the distance to obstacles in the environment, and hence, allowing the multirotor to avoid collisions.

### 3 Fuzzy Control System for Hexacopters

#### 3.1 *Brief Overview of Fuzzy Logic*

This section provides an overview of the key concepts of Fuzzy Logic in order to improve the reader's understanding on our ROS-based fuzzy logic library. Interested readers must refer to [17, 20] in order to get a deeper discussion on Fuzzy Logic.

Fuzzy logic is a way to model a system using basic human interpretations, providing a method to describe both the system model and the computation of its outputs. For instance, when someone says that “I am close to a car” and “you are not close to it”, the meaning of these phrases can be diverse. How can a computer calculate how close is something? A human could answer “yes”, “not” or “almost”. However, in order to provide such an answer, it is important to consider the context to realize

the meaning of this answer. Fuzzy logic provides a way to cope with the intrinsic imprecision of these answers by means of representing imprecisions and a common reference to the meaning of concepts such as “close” and “distant”.

Let us use the altitude control of the hexacopter as an example. For instance, the operator sets the altitude to ten meters. If the hexacopter is on the ground, at altitude zero, some amount of power must be applied to all rotors until it reaches the target altitude. How much power must be applied? If the target altitude is far away from the starting position, the maximum power might be applied, and hence, the hexacopter reaches the target altitude faster but it might overshoot the desired position. On the other hand, if the applied power is minimal, the hexacopter moves slower in order to minimize the overshooting, but it takes too much time to reach the desired position. The process of mapping and adjusting the numerical values to represent human linguistic values is key to design fuzzy systems.

A Fuzzy controller comprises a set of artifacts that enables the translation from human linguistic terms into elements that are processed by computers. The following artifacts compose a fuzzy control system:

- **Linguistic Variables** are used to represent the meaning of terms that are related to input or output signals. The concept of “distance” is an example. It may have the following values: “far”, “near”, “very close” and “on”. Linguistic variables can define output values as well, e.g. the “power” to be applied on the rotors could have the absolute values, e.g. “minimum”, “middle”, “maximum”, or relative values, e.g. “much lower”, “lower”, “maintain”, “higher” and “much higher”. Therefore any system input or output can be modeled as a set of linguistic values that have meaning for an human expert.
- **Membership Function** defines the mapping linguistic variable and its linguistic values. Since linguistic variable defines a set of concepts that are understood by human experts, its linguistic values must be defined so that the computer can process them. The expert defines numeric values to represent each concept. Each membership function defines the interpretation of any numeric value, incorporating the subjective imprecision. This is done by using a geometric representation of the concept, such as a Gaussian function, or triangular/trapezoidal function. Examples of some membership functions defined in the proposed fuzzy controllers are presented in the following sections.
- **Rules** define the relationship between input and output linguistic variables. A fuzzy system comprises a set of rules that relates premises and consequences in the form: IF premises THEN consequences. Premises represent comparisons between an input linguistic variable and values of its membership function. During the inference process, the relative strength of each premise is obtained by means of a process called *fuzzyfication* and thereafter propagated to the consequences. Once all the rules are evaluated, the consequences are evaluated altogether. The quantitative output values are produced within the *defuzzyfication* process and then such values are applied to the controlled process.



It is worth pointing out that, besides defining the fuzzy set, the engineers are responsible for tuning the membership function values, in order to obtain the desired behavior for each situation.

Furthermore, the execution of fuzzy logic systems comprises the following three processes:

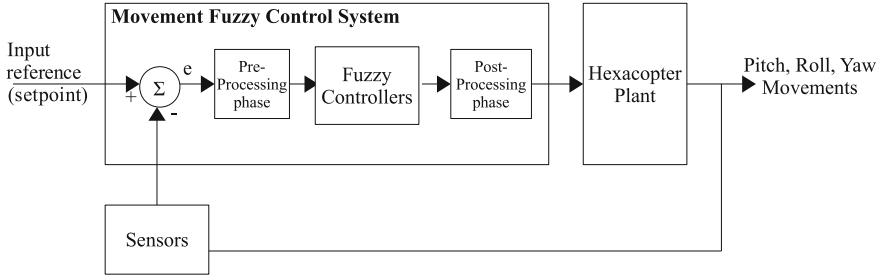
- **Fuzzyfication** is the process in which raw values obtained from input signal are compared with values of each member function, in order to find out the corresponding activation level. Usually the input signal value comes directly from a sensor reading. However, it can also be derived from some kind of calculation, such as the velocity obtained from the difference of two consecutive position readings provided by a GPS. If the raw value intercepts more than one membership function, all concepts are considered, and thus, each one presents a different activation level.
- **Rule inference** (also known as **rule evaluation**) is a process that evaluates all rules of the fuzzy system. It takes the fuzzyfied input values and evaluates the activation value of each premise, calculating the output value for each rule. Partially activated premises lead to partial activation of consequences, allowing for a “fuzzy” inference procedure.
- **Defuzzification** process obtains an exact output value (e.g. numerical value) that can be directly applied onto an actuator, e.g. the power to be applied onto the hexacopter rotors. The output strength points are used to calculate an average value. For that, the area formed by the union of each output membership function is used. There are several methods to obtain the output value. For instance, Center of Gravity (COG) defuzzification method takes into account the relative position over the horizontal axis plus the weight of the combined area.

### 3.2 Overview of Hexacopter Movement Control System

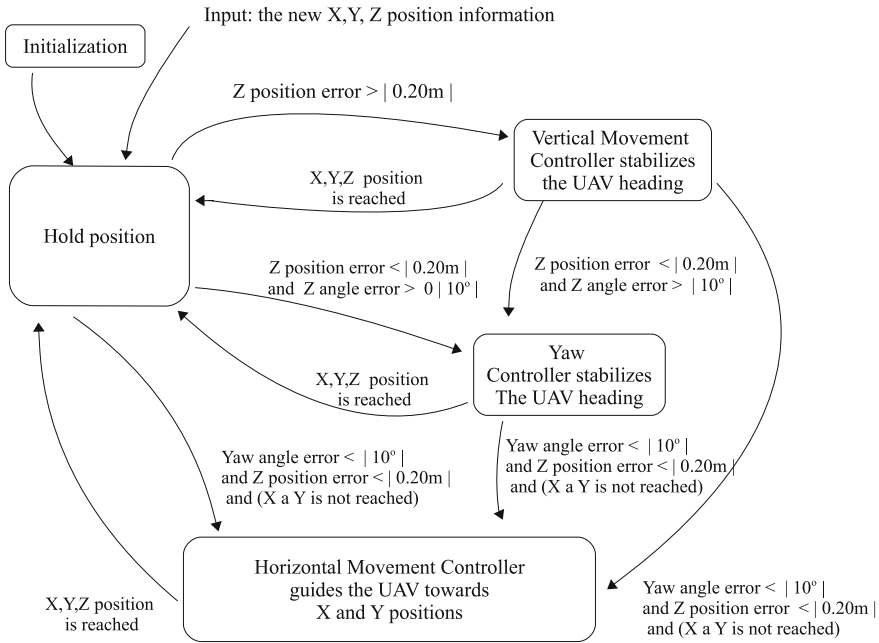
This section provides an overview of the fuzzy movement controller for a hexacopter, in order to explain the ROS-based open-source package presented in this chapter. Interested readers must refer to [11] in order to obtain details on the design of such a control system.

The proposed controller implements a closed loop that comprises three layers. Data produced as output of one layer is passed as input to the next layer. The proposed multi-layer fuzzy controller is based on [17] and is depicted in Fig. 6. The *Movement Fuzzy Control System box* is composed by a pre-processing phase (first layer), a set of fuzzy controllers (second layer), and post-processing phase (third layer).

As one can observe, after the post-processing phase, the control outputs are applied onto the plant by means of the hexacopter rotors that actuate on the hexacopter movement, i.e. pitch, roll, yaw. The sequence of maneuvering is depicted in Fig. 7. The sensors perceive the changes on the plant controlled variables, and hence, provide the feedback to the controller. The controller, in turn, compares these input values



**Fig. 6** Overview of the hexacopter movement fuzzy control system



**Fig. 7** FSM for sequencing the hexacopter maneuvering process. (X, Y, Z) inputs represent the new target position of the hexacopter

with the reference values established as setpoints thereby closing the control loop [17, 20].

The pre-processing phase (first layer) is responsible for acquiring data from the input sensors, processing the input movement commands, as well as calculating the controlled data used as input to the fuzzy controllers in the second layer. Before the multi-layer controller starts its execution, there is an initialization phase that is performed within the first layer. The target position is set as the current position, so that the hexacopter does not move before receiving any command. Gyroscope and accelerometer sensors are calibrated and the GPS sensor is initialized by gathering

at least four satellites. During the execution phase, the first layer is responsible to calculate the input variables to the fuzzy controllers: (i) the angular and linear distance (delta error) for X, Y, and Z axes between the current hexacopter position and the target position; (ii) the rotation and translation movement matrices in order to translate movement along X, Y and Z axes into the speed related to the ground (i.e. X and Y axis). In addition, the pre-processing phase is responsible to convert the input movement commands into setpoints for X-, Y- and Z-axis. Movements commands are composed of three values representing the positive or negative movement along X, Y and Z axes regarding the current positions, i.e. a command indicates a target relative position. Thus, when a new command is received, the first layer converts such a command to an absolute position. Then, once the control system is running, this layer uses the current GPS position to determine the error in the position of the hexacopter concerning the target position. These calculated errors in position are the inputs to the fuzzy controllers (Euler X, Euler Y and Euler Z errors).

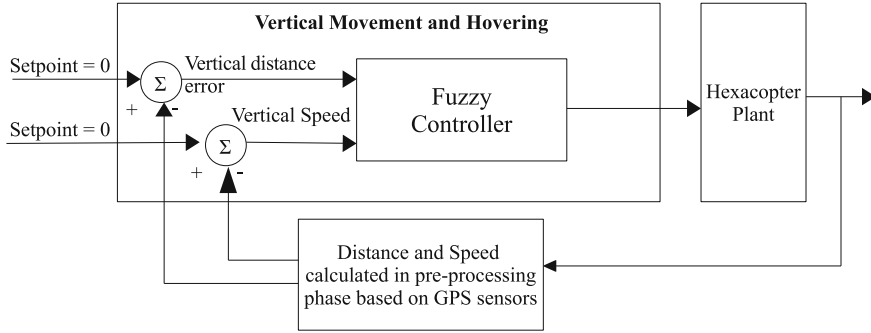
The second layer contains five fuzzy controllers, which act on issues regarding the hexacopter movement, namely hovering, vertical and horizontal movement and heading. As mentioned, these controllers take as input the data produced in the first layer and generate output for the third layer. The generated outputs represent the actuation on the six rotors for performing pitch, roll, yaw movements for all maneuvers necessary to reach the target position. In order to provide an illustrative example, one fuzzy controller is discussed in details in the next section.

The post-processing phase (third layer) is responsible for coordinating the fuzzy controllers outputs. In order to perform a proper maneuver, the proposed multi-layer controller establishes a priority on movements needed to complete a maneuver. When a new command is received, i.e. a new target point is set, the hexacopter must firstly reach the target altitude. Then, the hexacopter must turn until its front aims the target position. Finally, the hexacopter moves horizontally towards the target position. This layers controls the position thresholds by means of output values saturation, in order to keep the hexacopter stable while flying or hovering.

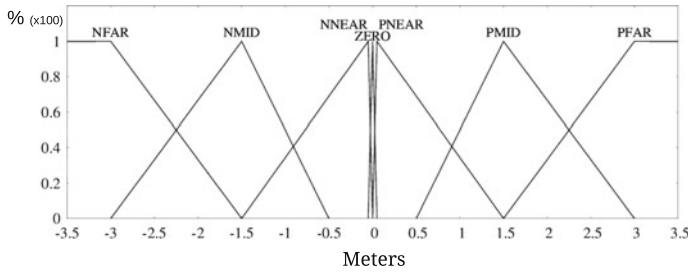
### ***3.3 Example: Design of Vertical Movement and Hovering Controller***

Vertical movement and hovering fuzzy controller controls the movement on the Z-axis, i.e. it controls uplift, downfall and hovering movements. Figure 8 shows the block diagram of this controller.

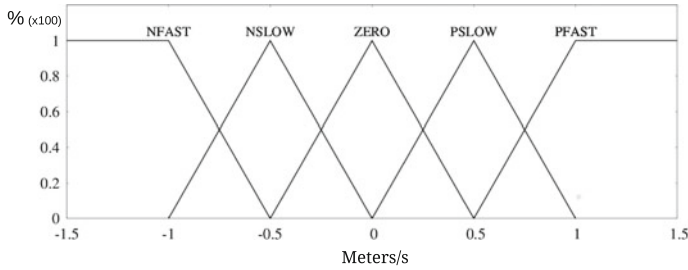
This controller takes as input the vertical distance to the target position, as well as the vertical speed. The first one is the error in vertical distance (altitude error), i.e. the difference between target position and actual hexacopter position on Z-axis. The second input is the current speed calculated as a derivative information of hexacopter displacement over time.



**Fig. 8** Vertical movement and hovering fuzzy controller



**Fig. 9** Input linguistic variables and their membership functions for vertical distance



**Fig. 10** Input linguistic variables and their membership functions for vertical speed

The linguistic variables for vertical distance and vertical speed are shown in Figs. 9 and 10, respectively. Letters “N” and “P” at the beginning of each membership function mean negative and positive values, respectively. In Figs. 9 and 10, values in the X-axis represent the distance from the setpoint in meters, whereas values in the Y axis indicate the activation of each membership function, varying from 0.0 (minimum) to 1.0 (maximum activation) indicating a percentage. The shape of these membership functions is the triangle, since the algorithm for calculating its area presents a low computing cost, and hence, it may be used on embedded system platform.

**Table 1** Control rules of vertical navigation

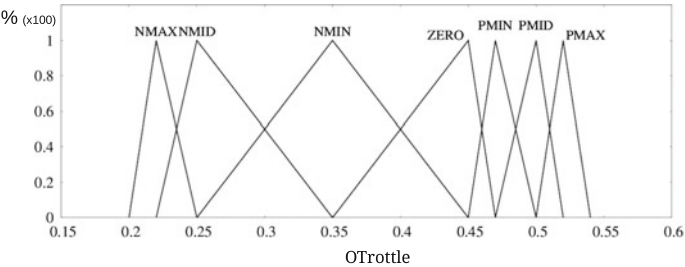
V. Dist. V. Speed	NFAR	NMID	NNEAR	ZERO	PNEAR	PMID	PFAR
NFAST	NMAX	NMID	NMIN	PMIN	PMID	PMAX	PMAX
NSLOW	NMAX	NMID	NMIN	PMIN	PMID	PMAX	PMAX
ZERO	NMAX	NMID	NMIN	ZERO	PMIN	PMID	PMAX
PSLOW	NMAX	NMAX	NMID	NMIN	PMIN	PMID	PMAX
PFAST	NMAX	NMAX	NMID	NMIN	PMIN	PMID	PMAX

Moreover, one can see the intersections of membership variables values, i.e. adjacent variables share a given range of values. This is an important characteristic of fuzzy systems and allows the modeling of smooth transitions among adjacent concepts. If there are gaps between transitions, i.e. no membership function is activated, the fuzzy control system may stop working. Another important issue is that the input membership functions must cover the complete range of input values, so that the fuzzy process can work with all possible sensor readings.

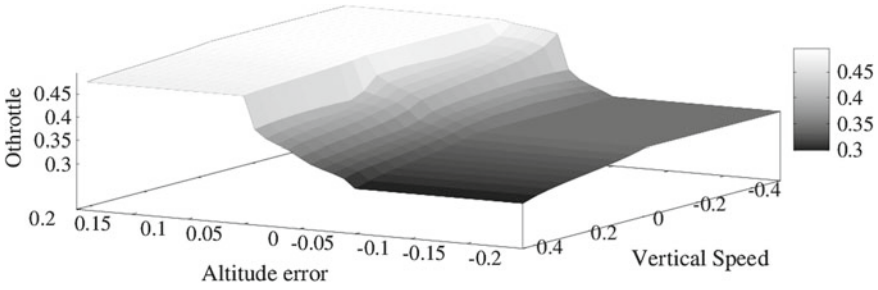
Fuzzy rules can be specified by means of clauses (e.g. if premise then consequence) or a table. The Table 1 shows the set of rules that composes the vertical movement and hovering controller. It is important to highlight that the number of rules increases as more linguistic variables and membership functions are added to the fuzzy control system.

This controller sets the output throttle variable `OThrottle` as result (see Fig. 11) Each value presented in Table 1 is decomposed into an amount of power applied on all rotors, increasing or decreasing the overall lift force making the hexacopter fly on higher or lower altitude. It is worth noting that the power applied on the rotors decreases along with vertical speed when the hexacopter comes closer to a target altitude.

The fuzzy control surface graphic shown in Fig. 12 provides the visualization of the input and output values of the vertical movement and hovering fuzzy controller. The altitude is maintained by means of controlling the throttle applied on all rotors



**Fig. 11** Output linguistic variables and their membership functions for `OThrottle`



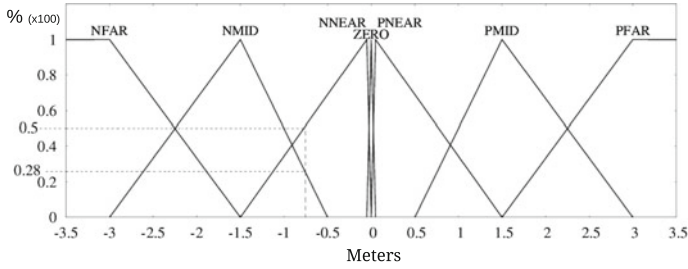
**Fig. 12** Surface of fuzzy controller for vertical navigation and hovering control

simultaneously. The GPS sensor provides the current altitude information. On the other hand, the vertical speed is used to decrease the oscillation when the hexacopter reaches the desired altitude. In order to illustrate the relationship between vertical distance error (altitude error) and vertical speed, let us suppose some situations. In the first one, the altitude error is zero (i.e. the hexacopter reaches the target vertical position) and the vertical speed is positive (e.g. 0.4 or higher). This situation means that the hexacopter has reached the target altitude but it will fly beyond that position because the speed indicates the hexacopter is still flying upwards. The hexacopter vertical speed must be slowed down before reaching the target altitude, and hence, the controller must set an output value lower than ZERO. In the second situation, the altitude error is zero and the vertical speed is negative. In this situation the hexacopter is falling down, and hence, the controller must set an output value higher than ZERO in order to stop the fall. It is worth noting that the ZERO output value for OThrottle does not mean that any power is applied on the rotors; instead, it represents a minimal power value that keeps the hexacopter hovering at the current altitude.

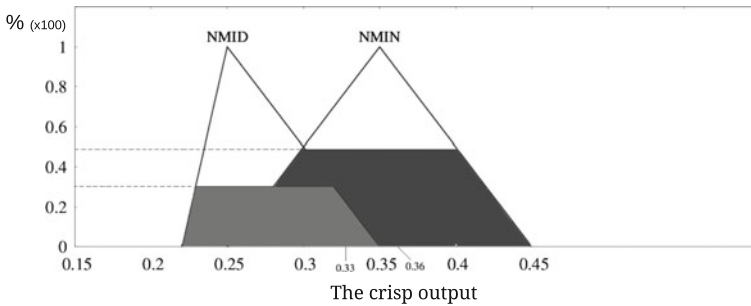
In order to illustrate the fuzzyfication and defuzzyfication process, let us assume that the current vertical distance error (altitude error) is  $-0.75$  m. This value is compared to the level of membership functions during the fuzzification process. After fuzzyfication, the value represents 0.28 (28%) of the NMID membership function and 0.5 (50%) of the NNEAR as shown in Fig. 13. During rules inference process, the activation of NMID and NNEAR membership function enables ten rules (see third and fourth columns of Table 1).

As one can see in Table 1, these rules combine two linguistic variables (vertical distance error and vertical speed), and hence, they define two premises that are connected with an “AND” operator (the minimum operator). “AND” operator selects a lower number of rules that have received high activation values. The resulting consequence depends on both linguistic variables. Let us suppose that the current value of speed\_uavZ is ZERO. The rule with the highest value is the intersection of NNEAR column and ZERO line of Table 1.

However, it is important to highlight that NMID variable has some influence on the result of the fuzzy rules inference. Therefore, Othrottle output linguistic variable is defined as a combination of NMID and NMID values. This situation forms points of



**Fig. 13** An example of input at  $-0.75$  m. After fuzzification the distance means 0.28 of negative middle, NMID, and 0.5 of NNEAR



**Fig. 14** Obtaining *Othrottle* raw value after defuzzification process

different activations between two output membership functions, creating two areas. The defuzzification process produces the raw value that represents the influence of both NMID and NMIN values. The values provided to each linguistic value depends on the activation of each rule. Figure 14 shows two possible output values. The first output value is 0.33, and it was obtained by calculating the arithmetic average of the two areas. The second output value is 0.36, and it was calculated using the center of gravity (COG) method that represents a weighted average between the two areas.

## 4 Open-Source Package of ROS-Based Fuzzy Logic Control Systems

### 4.1 Package Overview

This section describes our open-Source Package of ROS-based Fuzzy Logic Controllers [1]. This package provides the following artefacts: (i) the proposed fuzzy logic library; (ii) examples of fuzzy set files for the hexacopter movement fuzzy control system; (iii) the fuzzy control system main software implemented using ROS;

(iv) a software to send commands (i.e. the desired target position) to the virtual hexacopter; (v) a telemetry software that displays data from the hexacopter as well as the fuzzy controllers.

The package structure is composed by the following directories<sup>3</sup>:

- **fz** directory stores the text files that specify the fuzzy set. One fuzzy set is formed by three files: (i) input linguistic variables, e.g. “HexaPlus\_i\_stabZ.fz” defines the inputs for the vertical controller and hovering fuzzy controller; (ii) fuzzy rules, e.g. HexaPlus\_r\_stabZ.fz specifies the rules for the vertical controller and hovering fuzzy controller; (iii) output linguistic variables, e.g. “HexaPlus\_o\_stabZ.fz” defines the output linguistic variables.
- **include** directory provides the C/C++ header files. This directory presents two subdirectories: one subdirectory provides the header files for the tutorial, while the other one provides the header files for the fuzzy library, so that the library can be reused in other projects.
- **src** directory stores the source code. The code files of fuzzy logic library are stored within the subdirectory “fuzzy”, whereas the tutorial source code files are located directly in the `src` directory.
- **scenes** directory provides the V-REP scenarios that are used in this tutorial.

The next sessions discuss how to install, use and test our package. For that, the reader must use Ubuntu 14.04.4 LTS Operating System with the following software installed: ROS Indigo Igloo, catkin, cmake, V-REP PRO EDU version V3.3.0.<sup>4</sup>

## 4.2 *Configuring ROS Environment and Installing the Package*

First of all, a workspace is created in order to compile the shared library files so that the V-REP can be integrated with ROS. The V-REP provides a set of ROS packages to generate the share libraries. The workspace will also host the fuzzy package, as well as some libraries will be copied to the V-REP directory.

1. Create a directory under your home directory and initialize the catkin work-space using `catkin_init_workspace`.

```

1 $ cd ~
2 $ $ mkdir -p catkin_hexacopter/src
3 $ cd catkin_hexacopter/src
4 $ catkin_init_workspace
5 $ cd ..
6 $ catkin_make
7 $ source devel/setup.bash

```

<sup>3</sup>Files and subdirectories created automatically by catkin/make commands are ignored.

<sup>4</sup>This tutorial assumes that V-REP has been installed in `/opt/V-REP/` directory.



2. Copy the ROS package from the V-REP directory and generate the libraries with `catkin_make`.

```
1 $ cd ~/catkin_hexacopter/src
2 $ cp -rp /opt/V-REP/V-REP_PRO_EDU_V3_3_0_64_Linux/programming
  /ros_packages/* .
3 $ cd ..
4 $ catkin_make
```

3. Once two shared libraries have been created, copy these libraries to the V-REP directory, enabling the V-REP to work with the `roscore`.

```
1 $ cd ~/catkin_hexacopter
2 $ cp devel/lib/libv_repExtRos.so /opt/V-REP/V-
  REP_PRO_EDU_V3_3_0_64_Linux/
3 $ cp devel/lib/libv_repExtRosSkeleton.so /opt/V-REP/V-
  REP_PRO_EDU_V3_3_0_64_Linux/
```

These libraries enable V-REP to look for an instance of `roscore` at startup. If `roscore` is not running and the simulation scenario has some call to ROS, the simulation fails and the user is warned. V-REP acts as a ROS node, and hence, `roscore` must be running before starting V-REP. The integration between ROS and V-REP succeeded whether the ROS plugins are loaded during the V-REP startup, as shown below.

```
1 $ /opt/V-REP/V-REP_PRO_EDU_V3_3_0_64_Linux/vrep.sh &
2 ...
3 Plugin 'Ros': loading...
4 Plugin 'Ros': load succeeded.
5 ...
```

4. After performing this tutorial, the reader may want to delete V-REP packages, and thus, the following commands must be executed:

```
1 $ cd ~/catkin_hexacopter
2 $ rm -fr ros_bubble_rob vrep_joy vrep_*
```

Once `catkin` workspace identified as `catkin_hexacopter` has been created and configured, the reader can download our ROS-based fuzzy logic package from [1]. In order to compile and run such a package, the package zip file must be uncompressed inside the `catkin_hexacopter` workspace source directory.

```
1 $ unzip hexaplustutorial.zip -d ~/catkin_hexacopter/src
2 $ cd ~/catkin_hexacopter
3 $ catkin_make
```

If the fuzzy package has been uncompressed into a workspace with a lot of others packages, use the option “`pkg`” to compile only the fuzzy package.

```
1 $ catkin_make --pkg hexaplustutorial
```

These commands compile the package and generate the objects listed below.

```
1 [ 14%] [ 28%] Built target FuzzySet
2 Built target Linguistic
3 [ 28%] [ 28%] Built target FuzzyLoader
4 Built target HexaPlus
```

```

5 [ 28%] Built target LinguisticSet
6 [ 42%] [ 42%] Built target MembershipFunction
7 Built target Rule
8 [ 57%] Built target RuleSet
9 [ 71%] Built target RuleElement
10 [ 71%] [ 71%] Built target navigation

```

Thereafter, it is important to check whether the package was successfully compiled and it is working properly. For that, the roscore must be started, and then three package applications can be executed: `rosvrep_controller`, `rosvrep_panel`, `rosvrep_telemetry`. In addition, check the created topics by using `rostopic` and `rqt_graph`. However, before executing the package applications, the user must run the source command on the `setup.sh` file (line 2) at least once in the shell session, as well as open five terminals to run each application separately.

```

1 $ roscore &
2 $ source ~/catkin_hexacopter/devel/setup.sh
3 $ xterm & xterm & xterm & xterm & xterm &

```

The following commands should be executed in each terminal.

```

1 $ roslaunch hexaplayer_hexaplayer rosvrep_controller
2 $ roslaunch hexaplayer_hexaplayer rosvrep_panel
3 $ roslaunch hexaplayer_hexaplayer rosvrep_telemetry
4 $ rostopic list
5 $ rqt_graph

```

The `rostopic` command lists the topics beginning with `/vrep/`. These names can be easily modified through the `remap` argument of `roslaunch` command; for details see [21].

### 4.3 Fuzzy Set Files

As discussed in Sect. 3.1, the fuzzy set is composed of linguistic variables for inputs and outputs, membership functions and rules. Such information can be hard-coded into the source code files. This way, the fuzzy set is stored in memory by using arrays or lists, and thus, the fuzzy inference engine is able to produce the expected outputs. However, a flexible fuzzy inference engine is able to load the fuzzy set from a file stored in a storage drive (e.g. disk). In the proposed fuzzy library, we implemented a flexible engine that loads the fuzzy sets from plain text files. As described below, we defined a simple format to describe linguistic variables, membership functions and rules, in order to facilitate the specification process and also the system maintenance.

The fuzzy set is located in “*fz*” subdirectory. In order to illustrate how to specify a fuzzy set, the vertical movement and hovering controller (see Sect. 3.3) is used as a case study. The files whose name ends with “*\_stabZ.fz*” are related to this controller. Listing 1.1 describes the input linguistic variable depicted in Fig. 9 and specified in `HexaPlus_i_stabZ.fz` file. Likewise, Listing 1.2 describes the output linguistic variable depicted in Fig. 10 and specified in `HexaPlus_o_stabZ.fz` file.

**Listing 1.1** HexaPlus\_i\_stabZ.fz: Input linguistic variables and membership functions of vertical movement and hovering controller

```

1 uavZ_error NFAR -1001 -3.00 -3.00 -1.50
2 uavZ_error NMID -3.00 -1.50 -1.50 -0.50
3 uavZ_error NNEAR -1.50 -0.05 -0.05 0
4 uavZ_error ZERO -0.05 0 0 0.05
5 uavZ_error PNEAR 0 0.05 0.05 1.50
6 uavZ_error PMID 0.50 1.50 1.50 3.00
7 uavZ_error PFAR 1.50 3.00 3.00 1001
8
9 speed_uavZ NFAST -1000000 -1.00000 -1.00000 -0.50000
10 speed_uavZ NSLOW -1.00000 -0.50000 -0.50000 0.00000
11 speed_uavZ ZERO -0.50000 0.00000 0.00000 0.50000
12 speed_uavZ PSLOW 0.00000 0.50000 0.50000 1.00000
13 speed_uavZ PFAST 0.50000 1.00000 1.00000 1000000

```

**Listing 1.2** HexaPlus\_o\_stabZ.fz: Output linguistic variables and membership functions of vertical movement and hovering controller

```

1 uavZ_error NFAR -1001 -3.00 -3.00 -1.50
2 uavZ_error NMID -3.00 -1.50 -1.50 -0.50
3 uavZ_error NNEAR -1.50 -0.05 -0.05 0
4 uavZ_error ZERO -0.05 0 0 0.05
5 uavZ_error PNEAR 0 0.05 0.05 1.50
6 uavZ_error PMID 0.50 1.50 1.50 3.00
7 uavZ_error PFAR 1.50 3.00 3.00 1001
8
9 speed_uavZ NFAST -1000000 -1.00000 -1.00000 -0.50000
10 speed_uavZ NSLOW -1.00000 -0.50000 -0.50000 0.00000
11 speed_uavZ ZERO -0.50000 0.00000 0.00000 0.50000
12 speed_uavZ PSLOW 0.00000 0.50000 0.50000 1.00000
13 speed_uavZ PFAST 0.50000 1.00000 1.00000 1000000

```

Linguistic variables files follow the same format. Thus, these output linguistic variable files can be reused as input from different fuzzy controllers. As one can see in Listings 1.1 and 1.2, each line describes one linguistic variable. The first token is the name of linguistic variable, e.g. the vertical distance error `uavZ_error` and the vertical speed `speed_uavZ`. The second token is the name of the membership function, and the next four fields describe the range of values. There are four values in order to create membership function with triangular or trapezoidal shape. A triangle is formed when the second and third values are the same. These four points is referenced in the code implementation by variables “a”, “b”, “c” and “d”, respectively, defined in the *MembershipFunction* class (see *MembershipFunction.h* *MembershipFunction* in the *include* directory).

The vertical movement and hovering controller defines 35 fuzzy rules as depicted in Table 1 in Sect. 3.3. These rules are specified in `HexaPlus_r_stabZ.fz` file that is shown in Listing 1.3).

**Listing 1.3** HexaPlus\_r\_stabZ.fz: fuzzy rules of vertical movement and hovering controller

```

1 STABZ_NFAR_01 if uavZ_error is NFAR and speed_uavZ is PFAST then Othrottle is NMAX
2 STABZ_NFAR_02 if uavZ_error is NFAR and speed_uavZ is PSLOW then Othrottle is NMAX
3 STABZ_NFAR_03 if uavZ_error is NFAR and speed_uavZ is ZERO then Othrottle is NMAX
4 STABZ_NFAR_04 if uavZ_error is NFAR and speed_uavZ is NSLOW then Othrottle is NMAX
5 STABZ_NFAR_05 if uavZ_error is NFAR and speed_uavZ is NFAST then Othrottle is NMAX
6
7 STABZ_NMID_01 if uavZ_error is NMID and speed_uavZ is PFAST then Othrottle is NMAX
8 STABZ_NMID_02 if uavZ_error is NMID and speed_uavZ is PSLOW then Othrottle is NMAX
9 STABZ_NMID_03 if uavZ_error is NMID and speed_uavZ is ZERO then Othrottle is NMID
10 STABZ_NMID_04 if uavZ_error is NMID and speed_uavZ is NSLOW then Othrottle is NMID

```

```

11 STABZ_NMID_05 if uavZ_error is NMID and speed_uavZ is NFAST then Othrottle is NMID
12
13 STABZ_NNEAR_01 if uavZ_error is NNEAR and speed_uavZ is PFAST then Othrottle is NMID
14 STABZ_NNEAR_02 if uavZ_error is NNEAR and speed_uavZ is PSLOW then Othrottle is NMID
15 STABZ_NNEAR_03 if uavZ_error is NNEAR and speed_uavZ is ZERO then Othrottle is NMID
16 STABZ_NNEAR_04 if uavZ_error is NNEAR and speed_uavZ is NSLOW then Othrottle is NMID
17 STABZ_NNEAR_05 if uavZ_error is NNEAR and speed_uavZ is NFAST then Othrottle is NMID
18
19 STABZ_ZERO_01 if uavZ_error is ZERO and speed_uavZ is PFAST then Othrottle is NMID
20 STABZ_ZERO_02 if uavZ_error is ZERO and speed_uavZ is PSLOW then Othrottle is NMID
21 STABZ_ZERO_03 if uavZ_error is ZERO and speed_uavZ is ZERO then Othrottle is ZERO
22 STABZ_ZERO_04 if uavZ_error is ZERO and speed_uavZ is NSLOW then Othrottle is PMIN
23 STABZ_ZERO_05 if uavZ_error is ZERO and speed_uavZ is NFAST then Othrottle is PMIN
24
25 STABZ_PNEAR_01 if uavZ_error is PNEAR and speed_uavZ is NFAST then Othrottle is PMID
26 STABZ_PNEAR_02 if uavZ_error is PNEAR and speed_uavZ is NSLOW then Othrottle is PMID
27 STABZ_PNEAR_03 if uavZ_error is PNEAR and speed_uavZ is ZERO then Othrottle is PMID
28 STABZ_PNEAR_04 if uavZ_error is PNEAR and speed_uavZ is PSLOW then Othrottle is PMIN
29 STABZ_PNEAR_05 if uavZ_error is PNEAR and speed_uavZ is PFAST then Othrottle is PMIN
30
31 STABZ_PMID_01 if uavZ_error is PMID and speed_uavZ is NFAST then Othrottle is PMAX
32 STABZ_PMID_02 if uavZ_error is PMID and speed_uavZ is NSLOW then Othrottle is PMAX
33 STABZ_PMID_03 if uavZ_error is PMID and speed_uavZ is ZERO then Othrottle is PMID
34 STABZ_PMID_04 if uavZ_error is PMID and speed_uavZ is PSLOW then Othrottle is PMID
35 STABZ_PMID_05 if uavZ_error is PMID and speed_uavZ is PFAST then Othrottle is PMID
36
37 STABZ_PFAR_01 if uavZ_error is PFAR and speed_uavZ is NFAST then Othrottle is PMAX
38 STABZ_PFAR_02 if uavZ_error is PFAR and speed_uavZ is NSLOW then Othrottle is PMAX
39 STABZ_PFAR_03 if uavZ_error is PFAR and speed_uavZ is ZERO then Othrottle is PMAX
40 STABZ_PFAR_04 if uavZ_error is PFAR and speed_uavZ is PSLOW then Othrottle is PMAX
41 STABZ_PFAR_05 if uavZ_error is PFAR and speed_uavZ is PFAST then Othrottle is PMAX

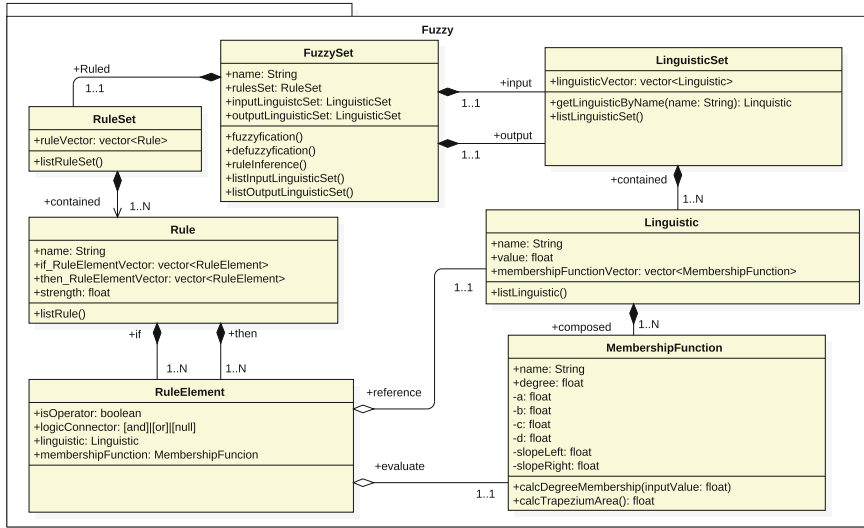
```

The first token is the rule name. Such information is used to identify each rule and it is not used for processing. The token “if” indicates that the next tokens are related to the rule premises. Each premise must specify one or more logical expressions following the format *name of linguistic variable* “is” *name of a membership function value*. The token “is” indicates that the activation level for each membership function must be calculated. A premise may indicate multiple expressions that are related by means of “AND” or “OR” operators. If “AND” operator is used, the rule activation (consequence) is equal to the minimum activation of all the premises of this rule. If “OR” operator is used, the rule activation is equal to the maximum activation of all premises.

The token “then” indicates that the next tokens are related to the consequence. During the rule inference process, an activation is assigned to the rule according to its premises activation. During the defuzzification, the activation level of each output linguistic variable is obtained by using the maximum operator, forming an area under all the membership functions that are activated in the rule (see Sect. 3.1). There may be one or more membership function values whose area value is greater than zero. The Center of Gravity (COG) approach is used to determine the weighted average of these areas which is used to determine the raw output value. Section 3.3 provides an example of this process.

## 4.4 Fuzzy Library Implementation

The fuzzy library provides the set of classes depicted in class diagram shown in Fig. 15. Our package provides a C++ implementation of the fuzzy library. A *Fuzzy Set* object is composed of two sets of *Linguistic Variables* objects: one for



**Fig. 15** The UML class diagram of the fuzzy logic library

representing input and another representing output linguistic variables. In addition, *Fuzzy Set* object owns a set of fuzzy *Rules*. Each *Rules* is composed of one or more premises and one or more consequences. Premises and consequences are represented as *Rules Element* objects. One premise describes a logic expression that includes a *Linguistic Variable* and *Membership Function* value. One consequence defines a value for a output *Linguistic Variable*. Each *Linguistic Variable* has a set of *Membership Functions*, which, in turn, represents triangular or trapezoidal shapes as depicted in Figs. 9, 10, and 11 in Sect. 3.3. The fuzzy library provides an additional class named *FuzzyLoader*. Such a class is responsible to load the information contained in the fuzzy set files, i.e. linguistic variables, membership functions, and rules (see Sect. 4.3).

In order to process the fuzzy control system implemented using the proposed library, the sequence of steps must be performed. First of all, the objects representing fuzzy set must be instantiated. For that, a *FuzzyLoader* object is created. It loads the information from the fuzzy set files (\*.fz) and instantiates the library objects accordingly. In the hexacopter movement control system, the *HexaPlus* class implements a method called *HexaPlus::initFZ()* that is responsible for loading the fuzzy set. Listing 1.4 shows a code fragment of this method. This code shows how to load the fuzzy set files of the vertical movement and hovering controller.

**Listing 1.4** Source code fragment of *HexaPlus::initFZ()* method

```

1 void HexaPlus::initFZ() {
2     // Getting the package path
3     std::string path = ros::package::getPath("hexaplus_tutorial");
4     // String objects declared
5     std::stringstream ssi, sso, ssr;
6 }

```

```

7 // Using the FuzzyLoader to upload the fuzzy artifacts
8 fuzzy::FuzzyLoader fzLoader;
9 ... // some lines are omitted
10
11 // Loading inputs and outputs linguistics variable
12 // with their membership functions
13 // and the rules of vertical controller
14 ssi.str(""); ssi << path << "/fz/HexaPlus_i_stabZ.fz";
15 sso.str(""); sso << path << "/fz/HexaPlus_o_stabZ.fz";
16 ssr.str(""); ssr << path << "/fz/HexaPlus_r_stabZ.fz";
17
18 fzLoader.loadFromFile(fS_stabZ,
19                      ssi.str().c_str(),
20                      sso.str().c_str(),
21                      ssr.str().c_str());
22 ... // remainder lines are omitted

```

Once the fuzzy objects are instantiated, the system controller can be initialized and executed. The *HexaPlus::initHexaPlus()* method initializes all variables related to the hexacopter movement control system. On the other hand, the source code file named *rosvrep\_controller.cpp* contains the *main()* function. In addition to the invocation of *HexaPlus::initHexaPlus()* method, the *main()* function defines a ROS node and configures the data publishers and subscribers. Thereafter the main control loop is executed. Details are provided in Sect. 4.5.

The fuzzy system is processed within the *FuzzySet::fuzzifying()* method. As presented in Sect. 3.2, there are five fuzzy controllers, and hence, there are *FuzzySet* objects to control hovering, vertical and horizontal movements and heading. Listing 1.5 shows the implementation of *FuzzySet::fuzzifying()* method. As one can see three main steps are executed.

**Listing 1.5** Source code of *FuzzySet::fuzzifying()* method

```

1 void FuzzySet::fuzzifying()
2 {
3     fuzzification();
4     ruleInference();
5     defuzzification();
6 }

```

The first step is called *fuzzification* and consists of converting the raw value of a sensor reading into a value of linguistic variable. For that, the raw value is compared to the range of values defined in a membership function, in order to calculate the degree of membership of the raw values. Therefore, once the degree of membership is calculated, activation level for the raw value is identified. Listing 1.6 shows *FuzzySet::fuzzification()* method implementation. Listing 1.7 shows *MembershipFunction::calcDegreeMembership()* method implementation. The *delta1* and *delta2* variables indicate the distance of *inputValue* parameter<sup>5</sup> to the points *a* and *d* that represent the base of the trapezium or triangle. The membership degree is calculated by comparing the inclination of two segments (*slopeLeft* and *slopeRight*) multiplied by the corresponding delta values.

<sup>5</sup>I.e. the raw value read from a sensor.

**Listing 1.6** Source code of `FuzzySet::fuzzyfication()` method

```

1 void FuzzySet::fuzzyfication()
2 {
3     typedef std::vector<Linguistic>::iterator
4         linguisticIterator_t;
5     typedef std::vector<MembershipFunction>::iterator
6         membershipFunctionIterator_t;
7     linguisticIterator_t it_Ling;
8     membershipFunctionIterator_t it_Mem;
9
10    for (it_Ling=inputLinguisticSet->linguisticVector.begin();
11         it_Ling !=inputLinguisticSet->linguisticVector.end();
12         it_Ling++) {
13        for (it_Mem=it_Ling->membershipFunctionVector.begin();
14             it_Mem!=it_Ling->membershipFunctionVector.end();
15             it_Mem++) {
16            it_Mem->calcDegreeMembership(it_Ling->value);
17        }
18    }
19 }

```

**Listing 1.7** Source code of `MembershipFunction::calcDegreeMembership()` method

```

1 void MembershipFunction::calcDegreeMembership(float inputValue)
2 {
3     float delta1, delta2;
4     delta1 = inputValue - a;
5     delta2 = d - inputValue;
6     if ((delta1 <= 0) || (delta2 <= 0)) {
7         degree = 0;
8     } else {
9         degree= minimum((slopeLeft*delta1),(slopeRight*delta2));
10    }
11    degree = minimum(degree, FZ_MAX_LIMIT);
12 }
13 // Slopes are calculated during the loading of fuzzy set files
14 // slopeLeft  = (float) FZ_MAX_LIMIT/(b-a);
15 // slopeRight = (float) FZ_MAX_LIMIT/(d-c);

```

The second step of fuzzifying process is to perform the rules inference process. Such a process is implemented in `FuzzySet::ruleInference()` method as shown in Listing 1.8. As one can see, the inference process has two main steps. In the first step, the inference values previously calculated are dismissed. On the other hand, in the second step, the strength of all rules is calculated. The degree of membership for each linguistic variable is used to define the rule strength which, in turn, is used to define the rule activation.

**Listing 1.8** Source code of `FuzzySet::ruleInference()` method

```

1 void FuzzySet::ruleInference()
2 {
3     typedef std::vector<Rule>::iterator RuleIterator_t;
4     typedef std::vector<RuleElement>::iterator
5         RuleElementIterator_t;
6
7     RuleIterator_t it_r;
8     RuleElementIterator_t it_e;
9
10    //Clean up IF elements for new round
11    for (it_r=ruleSet->ruleVector.begin(); it_r != ruleSet->
12         ruleVector.end(); it_r++) {
13        it_r->strength=0;
14    }
15 }

```

```

12     for (it_e=it_r->then_RuleElementVector.begin(); it_e !=
13           it_r->then_RuleElementVector.end(); it_e++) {
14         it_e->linguistic->value= 0;
15         it_e->membershipFunction->degree = 0;
16     }
17
18     // Inference
19     float strength_tmp;
20     for (it_r=ruleSet->ruleVector.begin(); it_r != ruleSet->
21           ruleVector.end(); it_r++) {
22         // Calculate the strength of the premises
23         strength_tmp = FZ_MAX_LIMIT;
24         for (it_e=it_r->if_RuleElementVector.begin(); it_e != it_r
25               ->if_RuleElementVector.end(); it_e++) {
26             strength_tmp = minimum(strength_tmp, it_e->
27                                   membershipFunction->degree);
28             if (!it_e->isOperator)
29                 strength_tmp = 1-strength_tmp;
30         }
31         // Calculate the strength of the consequences
32         for (it_e=it_r->then_RuleElementVector.begin(); it_e !=
33               it_r->then_RuleElementVector.end(); it_e++) {
34             it_e->membershipFunction->degree = maximum(strength_tmp
35                                                         ,it_e->membershipFunction->degree);
36             if (!it_e->isOperator)
37                 it_e->membershipFunction->degree = 1 - it_e->
38                   membershipFunction->degree;
39         }
40         it_r->strength = strength_tmp;
41     }
42 }

```

Once all elements of rules are evaluated, the third step of fuzzifying process is the defuzzification process. As discussed in Sect. 3.2, during the defuzzification process, the triangle/trapezoid area of the output linguistic variables is calculated taking into account the membership function and the rule activation. Then the linguistic value chosen as output is converted into a raw value that may be applied to the rotors. Such a raw value is obtained by means of calculating an arithmetic average or a weighted average (Center of Gravity method) of two areas. Listing 1.9 shows the implementation of `FuzzySet::defuzzification()` method.

**Listing 1.9** Source code of `FuzzySet::defuzzification()` method

```

1 void FuzzySet::defuzzification()
2 {
3     typedef std::vector<Linguistic>::iterator
4         LinguisticIterator_t;
5     typedef std::vector<MembershipFunction>::iterator
6         MembershipFunctionIterator_t;
7
8     LinguisticIterator_t it_l;
9     MembershipFunctionIterator_t it_m;
10
11     float sum_prod;
12     float sum_area;
13     float area, centroide;
14
15     for (it_l=outputLinguisticSet->linguisticVector.begin(); it_l
16           != outputLinguisticSet->linguisticVector.end(); it_l++)
17     {
18         sum_prod=sum_area=0;

```



```

16     for (it_m = it_l->membershipFunctionVector.begin(); it_m
17         != it_l->membershipFunctionVector.end(); it_m++)
18     {
19         area = it_m->calcTrapeziumArea();
20         centroide = it_m->a + ((it_m->d - it_m->a) / 2.0);
21         sum_prod += area * centroide;
22         sum_area += area;
23     }
24     if (sum_area==0)
25         it_l->value = FZ_MAX_OUTPUT;
26     else
27     {
28         it_l->value = sum_prod/sum_area;
29     }
30 }
31 }

```

## 4.5 Main Controller Implementation

The hexacopter movement fuzzy control system has been implemented in some distinct source code files. The source code file named `rosvrep_controller.cpp` implements the main control loop, i.e. the system `main()` function. The file `HexaPlus.cpp` contains the implementation of the *HexaPlus* class that is responsible for initializing the fuzzy library objects (see Sect. 4.4).

The `main()` function is divided in two parts. The first one performs all necessary initialization, i.e. it instantiates the *HexaPlus* object, loads the fuzzy set files, creates a ROS node, and and configures the data publishers (i.e. callback functions) and subscribers. Listing 1.11 depicts fragments of the initialization part of the `main()` function.

**Listing 1.10** Fragments of `main()` function in `rosvrep_controller.cpp`

```

1  //////////////// THE CALLBACK FUNCTION
2  // Subscriber callback functions for euler angles
3  ...
4  void callback_eulerZ(const std_msgs::Float32 f)
5  { eulerZ = f.data; }
6  // Subscriber callback functions for GPS position
7  ...
8  void callback_gpsZ(const std_msgs::Float32 f)
9  { gpsZ = f.data; }
10 // Subscriber callback functions for accelerometer sensor
11 ...
12 void callback_accelZ(const std_msgs::Float32 f)
13 { accelZ = f.data; }
14 // Subscriber callback functions for operator setpoints
15 ...
16 void callback_gpsZ_setpoint(const std_msgs::Float32 f)
17 { gpsZ_setpoint = f.data; }
18 ...
19 //////////////// END CALLBACK FUNCTION
20 // Subscriber callback functions for euler angles
21 int main(int argc, char* argv[])
22 {

```

```

23     unsigned long int time_delay=0;
24
25 // Initialize the ros subscribers
26 ros::init(argc, argv, "rosvrep_controller");
27 ros::NodeHandle n;
28
29 // the rosSignal is used to send signal to uav via Publisher.
30 std_msgs::Float32 rosSignal;
31 // Hexacopter sensor subscribers
32 ... // some lines are omitted
33
34 // Initialize the ROS Publishers
35 // Rotors publishers
36 ros::Publisher rosAdv_propFRONT =
37     n.advertise<std_msgs::Float32>("/vrep/propFRONT",1);
38 ros::Publisher rosAdv_propLEFT_FRONT =
39     n.advertise<std_msgs::Float32>("/vrep/propLEFT_FRONT",1)
40     ;
41 ros::Publisher rosAdv_propLEFT_REAR =
42     n.advertise<std_msgs::Float32>("/vrep/propLEFT_REAR",1);
43 ros::Publisher rosAdv_propREAR =
44     n.advertise<std_msgs::Float32>("/vrep/propREAR",1);
45 ros::Publisher rosAdv_propRIGHT_FRONT =
46     n.advertise<std_msgs::Float32>("/vrep/propRIGHT_FRONT"
47     ,1);
48 ros::Publisher rosAdv_propRIGHT_REAR =
49     n.advertise<std_msgs::Float32>("/vrep/propRIGHT_REAR",1)
50     ;
51 ros::Publisher rosAdv_propYaw =
52     n.advertise<std_msgs::Float32>("/vrep/Yaw",1);
53 ... // remainder lines are omitted

```

The second part is the main control loop of the hexacopter movement fuzzy control system. Such a loop performs three main activities: (i) pre-processing phase, (ii) processing of five distinct fuzzy controllers, (iii) post-processing phase. These activities are discussed in Sect. 3.2. Moreover, the execution frequency of loop iterations is 10 Hz. Such an execution frequency is obtained by using the commands `loop_rate.sleep()` and `ros::spinOnce()` at the end of the loop. The 10 Hz timing requirement has been arbitrarily defined and has been demonstrated to be enough to control a simulated hexacopter as discussed in Sect. 5. However, it is important to highlight that a more careful and sound timing analysis is required in order to define the execution frequency of the main control loop for a real hexacopter. A discussion on such an issue is out of this chapter scope. Interested reader should refer to [19, 22–27].

The pre-processing phase is responsible for acquiring data from the input sensors, processing the input movement commands, as well as for calculating the controlled data used as input to the five fuzzy controllers. Two examples of data calculated in this phase are: (i) vertical and horizontal speed calculated using the hexacopter displacement over time; and (ii) the drift of new heading angle in comparison with the actual heading. Listing 1.11 presents some fragments of the code related to the pre-processing phase.

**Listing 1.11** Fragments of pre-processing phase in `rosvrep_controller.cpp`

```

1 ... // some lines are omitted
2 // Determine the delta as errors.
3 // It means the difference between setpoint and current
  information

```

```

4
5      // GPS error
6      gpsX_error = (float) gpsX_setpoint - gpsX;
7      gpsY_error = (float) gpsY_setpoint - gpsY;
8      gpsZ_error = (float) gpsZ_setpoint - gpsZ;
9
10     // View position error (yaw or heading of the hexacopter)
11     viewX_error = (float) viewX_setpoint - gpsX;
12     viewY_error = (float) viewY_setpoint - gpsY;
13
14     // Calculate the drift_angle
15     // This angle is the difference between direction
16     //to navigate and direction of view (yaw).
17     drift_angle = (float) eulerZ - uav_goal_angle;
18     ... // remainder lines are omitted

```

Once the pre-processing phase is executed, the second activity is responsible to execute the five fuzzy controllers. This occurs by means of invoking the `fuzzifying()` method of each controller `FuzzySet` object. The “fuzzifying” process includes “fuzzification”, rules inference, and “defuzzification” (see Sect. 4.4). Listing 1.12 depicts the code fragment that processes the five fuzzy controllers.

**Listing 1.12** Fragment depicting the processing five fuzzy controllers in `rosvrep_controller.cpp`

```

1      ... // previous lines are omitted
2      hexaplus.fS_stabX->fuzzifying();
3      hexaplus.fS_stabY->fuzzifying();
4      hexaplus.fS_stabZ->fuzzifying();
5      hexaplus.fS_yaw->fuzzifying();
6      hexaplus.fS_hnav->fuzzifying();
7      ... // remainder lines are omitted

```

The last activity is the post-processing phase. In this phase the output linguistic variables are transformed in raw values that are applied on the rotors in order to control the hexacopter movements. Listing 1.13 presents a fragment of post-processing phase code.

**Listing 1.13** Fragment depicting the post-processing phase in `rosvrep_controller.cpp`

```

1      // fuzzifying is finished, applying the outputs
2      Opitch = hexaplus.fz_Opitch->value;
3      Oroll = hexaplus.fz_Oroll->value;
4      Othrottle = hexaplus.fz_Othrottle->value;
5      Oyaw = hexaplus.fz_Oyaw->value;
6      Opitch_nav = hexaplus.fz_Opitch_nav->value;
7
8      propForceFRONT = (float) Othrottle - 0.45*zOth*cos(
9          angleOth);
10     propForceRIGHT_FRONT = (float) Othrottle - (0.45*zOth*(sin(
11         angleOth)/2 ));
12     propForceRIGHT_REAR = (float) Othrottle - (0.45*zOth*(sin(
13         angleOth)/2 ));
14     propForceREAR = (float) Othrottle + 0.45*zOth*cos(
15         angleOth);
16     propForceLEFT_REAR = (float) Othrottle + (0.45*zOth*(sin(
17         angleOth)/2 ));
18     propForceLEFT_FRONT = (float) Othrottle + (0.45*zOth*(sin(
19         angleOth)/2 ));
20
21     // Sending signals to the rotors
22     rosSignal.data = propForceFRONT;

```

```

17   rosAdv_propFRONT.publish(rosSignal);
18   rosSignal.data = propForceRIGHT_FRONT;
19   rosAdv_propRIGHT_FRONT.publish(rosSignal);
20   rosSignal.data = propForceRIGHT_REAR;
21   rosAdv_propRIGHT_REAR.publish(rosSignal);
22   rosSignal.data = propForceREAR;
23   rosAdv_propREAR.publish(rosSignal);
24   rosSignal.data = propForceLEFT_REAR;
25   rosAdv_propLEFT_REAR.publish(rosSignal);
26   rosSignal.data = propForceLEFT_FRONT;
27   rosAdv_propLEFT_FRONT.publish(rosSignal);
28   rosSignal.data = Oyaw;
29   rosAdv_propYaw.publish(rosSignal);

```

Finally, it is worth mentioning that the main controller interacts with other two applications. A command interface application named *Panel* sends commands to determine a new position, as well as new heading direction, towards which the hexacopter must fly. Moreover, some data produced in the main controller are published so that these telemetry data can be seen within an application named *Telemetry*. Listing 1.14 shows the code in `rosvrep_controller.cpp` that configures ROS publishers for the telemetry data. Next sections provide detail on these two applications.

**Listing 1.14** Configuring ROS publisher for publishing telemetry data in `rosvrep_controller.cpp`

```

1  // Telemetry
2  ros::Publisher rosAdv_gpsX_error = n.advertise<std_msgs::
    Float32>("/hexapalus_tutorial/gpsX_error",1);
3  ros::Publisher rosAdv_gpsY_error = n.advertise<std_msgs::
    Float32>("/hexapalus_tutorial/gpsY_error",1);
4  ros::Publisher rosAdv_gpsZ_error = n.advertise<std_msgs::
    Float32>("/hexapalus_tutorial/gpsZ_error",1);
5  ros::Publisher rosAdv_drift_angle = n.advertise<std_msgs::
    Float32>("/hexapalus_tutorial/drift_angle",1);
6  ros::Publisher rosAdv_uav_goal_angle = n.advertise<std_msgs::
    Float32>("/hexapalus_tutorial/uav_goal_angle",1);
7  ros::Publisher rosAdv_uav_goal_dist = n.advertise<std_msgs::
    Float32>("/hexapalus_tutorial/uav_goal_dist",1);
8  ros::Publisher rosAdv_speed_uavZ = n.advertise<std_msgs::
    Float32>("/hexapalus_tutorial/speed_uavZ",1);
9  ros::Publisher rosAdv_speed_goal = n.advertise<std_msgs::
    Float32>("/hexapalus_tutorial/speed_goal",1);

```

## 4.6 Command Interface Implementation

The command interface application named *Panel* is a ROS node that allows a user to send commands to modify hexacopter pose and position. The implementation of such an application is provided in `rosvrep_panel.cpp` file. Two types of commands are allowed: (i) the user can set a new (X, Y, Z) position, and hence, the hexacopter will fly towards this target position; (ii) the user can set a new heading direction by setting a new (X, Y) position, and hence, the hexacopter will perform a yaw movement in order to aim the target position.

The *Panel* application is very simple: it publishes a setpoint position and a view direction, as well as provides means for user input. Listing 1.15 shows the code fragment that configures the ROS publisher for the new 3D position (i.e. setpoint) and new heading (i.e. view direction).

**Listing 1.15** Configuring ROS publisher for telemetry data in `rosvrep_panel.cpp`

```

1 // Operator setpoint Publishers
2 ros::Publisher rosAdv_gpsX_setpoint = n.advertise<std_msgs::
   Float32>("/hexaplug_tutorial/gpsX_setpoint",1);
3 ros::Publisher rosAdv_gpsY_setpoint = n.advertise<std_msgs::
   Float32>("/hexaplug_tutorial/gpsY_setpoint",1);
4 ros::Publisher rosAdv_gpsZ_setpoint = n.advertise<std_msgs::
   Float32>("/hexaplug_tutorial/gpsZ_setpoint",1);
5
6 ros::Publisher rosAdv_viewX_setpoint = n.advertise<std_msgs::
   Float32>("/hexaplug_tutorial/viewX_setpoint",1);
7 ros::Publisher rosAdv_viewY_setpoint = n.advertise<std_msgs::
   Float32>("/hexaplug_tutorial/viewY_setpoint",1);

```

The *Panel* application must be executed with the `roslaunch` command as depicted in line 01 from Listing 1.16. When the user presses “s”, he/she is asked to inform new position setpoint in terms of X, Y, Z coordinates. When the user presses “y”, he/she is asked to inform the new heading direction new in terms of X, Y coordinates. In the example presented in lines 09–12 from Listing 1.16, the user sent (5, 3, 7) as new (X, Y, Z) target position. It is important to mention that the values for (X, Y, Z) coordinates are measured in meters. After sending the new setpoints, the hexacopter starts moving. If the user press CTRL-C and the ENTER keys, the program is finished and the hexacopter continues until it reaches the target position.

**Listing 1.16** Panel application

```

1 $ roslaunch hexaplug_tutorial rosvrep_panel
2
3 =====
4
5 Setpoints for position [s]
6 Setpoints for View heading [y]
7 Or CTRL-C to exit.
8
9 Enter the option: s
10 Enter X value: 5
11 Enter Y value: 3
12 Enter Z value: 7

```

## 4.7 Telemetry Implementation

The *Telemetry* application is also a very simple program. It receives the signals from the hexacopter sensors and from some data calculated during the execution of the control program. Likewise the *Panel* application, the *Telemetry* application is executed with the `roslaunch` command as depicted in line 01 from Listing 1.18. The telemetry data is shown in lines 03–23.

**Listing 1.17** Panel application

```

1 $ rosrund hexapplus_tutorial rosvrep_telemetry
2
3 ----- Telemetry -----
4 gpsX .....: 0.000000
5 gpsY .....: 0.000000
6 gpsZ .....: 0.000000
7 gpsX_error .....: 0.000000
8 gpsY_error .....: 0.000000
9 gpsZ_error .....: 0.000000
10 drift_angle .....: 0.000000 (0.000000
    degrees)
11 uav_goal_angle .....: 0.000000 (0.000000
    degrees)
12 uav_goal_dist .....: 0.000000
13 speed_uavZ .....: 0.000000
14 speed_goal .....: 0.000000
15 ----- Operator Command -----
16 gpsX_setpoint .....: 0.000000
17 gpsY_setpoint .....: 0.000000
18 gpsZ_setpoint .....: 0.000000
19
20 viewX_setpoint .....: 0.000000
21 viewY_setpoint .....: 0.000000
22 -----
23 Press CTRL-C to exit

```

*Telemetry* application is a very simple program. It subscribes some ROS topics and displays them on the terminal. The program terminates when CTRL-C is pressed. The `rosvrep_telemetry.cpp` file implements this application. The main part of the code is the declaration of ROS subscribers and callback functions. Listing 1.18 shows these declarations. Callback functions declaration is depicted in line 02–04, while ROS subscribers in line 09–11. One can notice that some topics start with “/vrep”, others with “/hexapplus\_tutorial”; this means that some topics came from V-REP and other ones from the control program.

**Listing 1.18** Panel application

```

1 ... // previous line omitted
2 void callback_gpsX_error(const std_msgs::Float32 f) { gpsX_error
    = f.data; }
3 void callback_gpsY_error(const std_msgs::Float32 f) { gpsY_error
    = f.data; }
4 void callback_gpsZ_error(const std_msgs::Float32 f) { gpsZ_error
    = f.data; }
5
6 ... // some lines omitted
7
8 ros::Subscriber sub_gpsX_error = n.subscribe("/hexapplus_tutorial
    /gpsX_error",1, callback_gpsX_error);
9 ros::Subscriber sub_gpsY_error = n.subscribe("/hexapplus_tutorial
    /gpsY_error",1, callback_gpsY_error);
10 ros::Subscriber sub_gpsZ_error = n.subscribe("/hexapplus_tutorial
    /gpsZ_error",1, callback_gpsZ_error);
11 ... // remaining lines omitted

```

## 5 Virtual Experimentation Platform

### 5.1 Introduction

A common tool used during the design of control systems is the simulator. There is a number of different simulators available for using, e.g. Simulink, Gazebo and Stage. In special, for robotics control systems design, a virtual environment for simulation must allow the creation of objects and also the specification of some of the physical parameters for both objects and the environment. The virtual environment should also provide a programming interface to control not only the simulation, but also the objects behavior and the time elapsed in simulation.

Although there are some robotics simulators supported in ROS such as Gazebo and Stage, this tutorial discusses the use of a different robotics simulator named V-REP. The main goal is to show the feasibility of using other (non-standard) simulators, opening room for the engineer to choose the tools he/she finds suitable for his/her project. The hexacopter movement fuzzy control system is used to illustrate how to integrate V-REP with ROS. An overview on V-REP virtual simulation environment is presented, so that the reader can understand how a virtual hexacopter was created. In addition, the reader will learn how the V-REP acts as a ROS publisher/subscriber to exchange messages with `roscore`.

V-REP uses the Lua language [28] to implement scripts that access and control the simulator. Lua is quite easy to learn, and hence, only a few necessary instructions are presented herein. Although V-REP uses Lua for its internal scripts, there are many external interfaces to other languages, such as C/C++, Java, Python, Matlab and ROS. V-REP documentation is extensive, and hence, the interested reader should refer to [29].

The installation of the V-REP simulator on Linux is simple: the reader must download the compressed installation file from Coppelia Robotics' website [30] and expand it on a directory using the UNIX `tar` command. It is interesting to mention some subdirectories within V-REP directory:

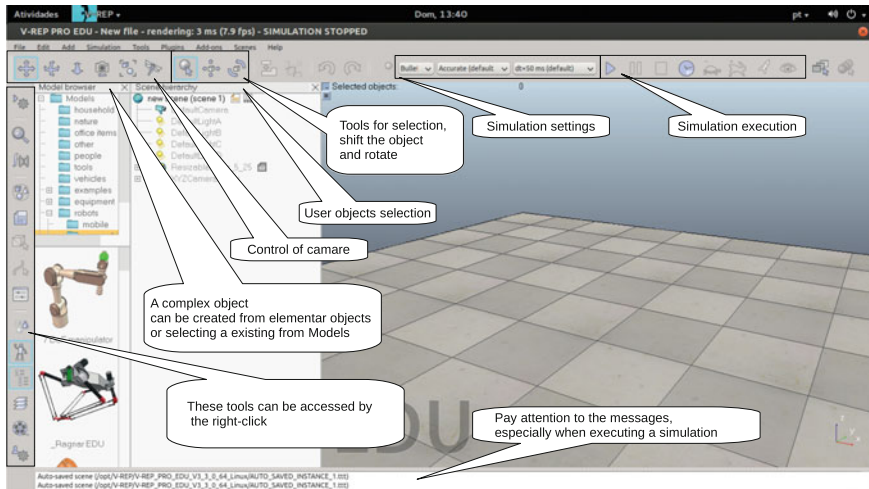
- **scenes:** V-REP provides a number of scenes as examples. The scene files extension is “`ttt`”.
- **tutorial:** This directory provides all scenes used in the tutorials presented in the V-REP site [29].
- **programming:** This directory provides examples written in C/C++, Java, Lua and Python. In addition, it provides the `ros_packages` interface that are in this tutorial.

## 5.2 V-REP Basics

When V-REP is started, a blank scenario is open automatically for using. The user can start developing a new scenario, or open a scenario created previously, or open a scenario from `scenes` directory. Figure 16 shows a screenshot.

In order to illustrate the use of V-REP, select the menus `File` → `Open scene` and choose the scene `Hexacopter.ttt` provided in the directory `~/catkin_hexacopter/src/hexaplayer_tutorial/scenes`. A complex object like a hexacopter is built by putting objects under a hierarchic structure. For instance, the sensors such as GPS, gyroscope and accelerometer are under the *HexacopterPlus* object. During the simulation execution, if the *HexacopterPlus* or any subpart, is moved, all parts are moved as if they are a single object. Any primitive object, e.g. cuboids, cylinders and spheres, can be inserted into a scene. There are some especial objects such as joints, video sensors, force sensors, and other. The special objects have some specific attributes used during simulation, e.g. position information, angle measurements, force data, etc. The sensors and rotors are made from these kinds of objects. The user can get some already available devices from “Model Browser”. For instance, there are several sensors available in the `Model Browser` → `components` → `sensors`, e.g. laser scanners, the Kinect sensor, Velodyne, GPS, Gyrosensor and Accelerometer. The last three sensors were used in the hexacopter model.

It is important to mention that when a new robot is created, one must pay attention to the orientation between the robot body and its subparts, especially sensors. Sensors will not work properly whether there are inconsistencies in the parts orientation. The



**Fig. 16** Screenshot of the V-REP initial screen



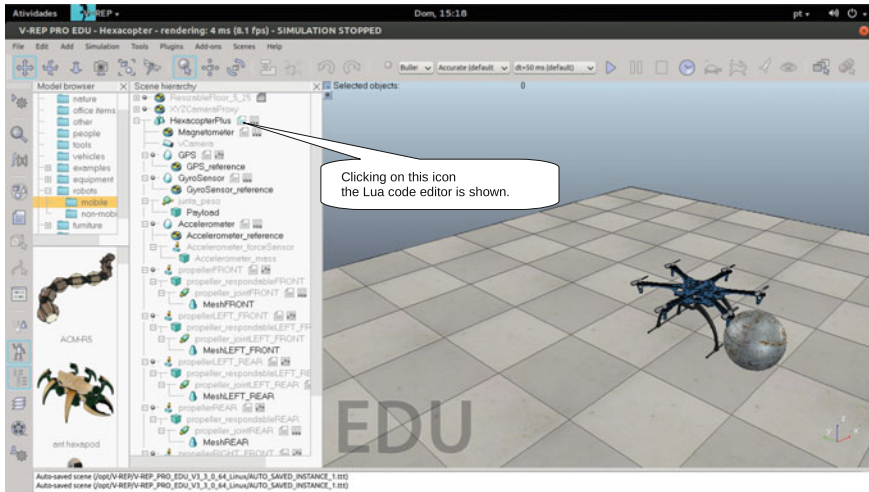


Fig. 17 Open the Lua script code

inertial frame 3D orientation is shown on the bottom right corner. When one clicks on any object, the 3D axes of the selected object body orientation is depicted.

The camera sensor is an exception. The camera orientation has a rotation of  $+90^\circ$  over the Z-axis and  $-90^\circ$  over the X-axis in relation to the axes of the robot body. Such a situation leads to an issue: Z-axis of the camera matches with the X-axis of the robot. Thus, the camera X-axis matches the robot Y-axis, and the camera Y-axis matches the robot Z-axis. Such a difference can be seen by clicking on vCamera and hexacopter object while pressing the shift key at the same time.

In addition, one can observe that some objects have an icon to edit its Lua script code, as shown at Fig. 17. If the object does not have a piece of code, it is possible to add one by the right-clicking on the object and choosing Add  $\rightarrow$  Associated child script  $\rightarrow$  Non Threaded (or Threaded).

While a simulation is running, V-REP executes the scripts associated to each object throughout the main internal loop. Script execution can be run in separate thread whether the associated script is indicated as threaded. V-REP controls the simulation elapsing time by means of time parameters. In order to execute the simulation of this tutorial, set the time configuration as “Bullet”, “Fast” at “dt = 10.0 ms”. This will ensure a suitable simulation speed.

### 5.3 Publishing ROS Topics

V-REP provides a plugin infrastructure that allows the engineer customize the simulation tool. *RosPlugin services* in V-REP is an interface to support general ROS

functionality. The V-REP has several mechanisms to communicate with the user code: (i) tubes are similar to the UNIX pipe mechanism; (ii) signals are similar to global variables; (iii) wireless communication simulation; (iv) persistent data blocks; (v) custom Lua functions; (vi) serial port; (vii) LuaSocket; (viii) custom libraries, etc. An easy way to communicate with ROS is creating a V-REP signal and publishing or subscribing its topic. *RosPlugin publishers* offer an API to setup and publish data within ROS topics.

An example on how the V-REP publishes messages to `roscore` can be found in the Lua child object script of *HexacopterPlus*. Let us consider the GPS as an example. Before publishing GPS data, it is necessary to check if the ROS module has been loaded. Listing 1.19 depicts the Lua script defined in the *HexacopterPlus* element as shown in Fig. 17.

**Listing 1.19** Lua script to check whether ROS module is loaded

```

1  ... -- previous lines are omitted
2  -- Check if the required remote Api plugin is there:
3  moduleName=0
4  moduleVersion=0
5  index=0
6  pluginNotFound=true
7  while moduleName do
8      moduleName,moduleVersion=simGetModuleName(index)
9      if (moduleName=='Ros') then
10         pluginNotFound=false
11     end
12     index=index+1
13 end
14 if (pluginNotFound) then
15     -- Display an error message if the plugin was not found:
16     simDisplayDialog('Error',
17         'ROS plugin was not found.&&nSimulation will not run
18         properly',
19         sim_dlgstyle_ok,false,nil,{0.8,0,0,0,0,0},{0.5,0,0,1,1,1})
20 else
21     -- Ok go on.
22 ... -- remainder lines are omitted

```

All plugins are loaded by executing the `simLoadModule` function, however, ROS plugins are loaded automatically during V-REP startup, i.e. the library `libv_repExtRos.so` is loaded automatically. This is achieved because the shared ROS libraries were generated and copied to the V-REP directory in Sect. 4.2.

The scripts in V-REP are divided into sections. At simulation time, all scripts are executed within the internal main loop. Some sections are executed once, whereas others are performed on each loop iteration. The script fragment presented in Listing 1.20 executes once on each simulation. For publishing the GPS data as topic to `roscore`, a special Lua function is called. There is a variety of Lua functions provided by V-REP team in order to work with ROS and others communication channels.

**Listing 1.20** preencher

```

1  ... // previous lines are omitted
2  -- Publish the Euler angle as ROS topic
3  topicName=simExtROS_enablePublisher('eulerX' 1,
    simros_strcmd_get_float_signal,-1,-1,'eulerX',0)

```

```

4 topicName=simExtROS_enablePublisher('eulerY',1,
    simros_strmcmd_get_float_signal,-1,-1,'eulerY',0)
5 topicName=simExtROS_enablePublisher('eulerZ',1,
    simros_strmcmd_get_float_signal,-1,-1,'eulerZ',0)
6 ... // next lines are omitted

```

The `simExtROS_enablePublisher` function is used to enable a publisher on V-REP. The parameters are similar to the function used for publishing data by means of `ros::Publisher.advertise` method as follows:

1. The name of the target topic to which data is published, e.g. “eulerX”.
2. The queue size has the same meaning of the original queue size of ROS publisher.
3. The stream data type parameter is used define how to process the two following parameters, e.g. the user can use the `simros_strmcmd_get_float_signal` signal to publish floating-point data. There is a variety of predefined data types.
4. The meaning of `auxiInt1` parameter depends on the data type. When this parameter is not in use, the value is `-1`.
5. The `auxiInt2` parameter semantics is similar to `auxiInt1`.
6. The `auxString` parameter. The type `simros_strmcmd_get_float_signal` means that a float type from a V-REP signal is being published. This parameter must match with the signal name. Listing 1.21 depicts the GPS script code that is used to explain how a V-REP signal is declared within a Lua script.
7. The `publishCnt` parameter indicates the number of times a signal is published before it goes to sleep. The `-1` value lead to start the sleep mode, whereas values greater than zero indicates that data are published exactly `publishCnt` times. The publisher wakes up when `simExtROS_wakePublisher` is executed. All published data never sleep by setting this parameter to zero.

Some published or subscribed data types use the parameters `auxiInt1` or `auxiInt2`. For example, the `simros_strmcmd_get_joint_state` type was used to get the joint state. It uses the `auxiInt1` to indicate the joint handle. Other type is `simros_strmcmd_get_object_pose` which is used to enable data streaming from the the object pose. This type uses the `auxiInt1` to identify V-REP object handle and the `auxiInt2` indicates the reference frame from which the pose is obtained. For more information please see [31].

Listing 1.21 presents a code fragment of the virtual GPS script. These lines create three distinct signals related to the object position information. The `objectAbsolutePosition` variable is a Lua vector with values calculated before in this fragment execution.

**Listing 1.21** Fragment of Lua script of the virtual GPS

```

1 ... // previous lines are omitted
2 simSetFloatSignal('gpsX',objectAbsolutePosition[1])
3 simSetFloatSignal('gpsY',objectAbsolutePosition[2])
4 simSetFloatSignal('gpsZ',objectAbsolutePosition[3])
5 ... // next lines are omitted

```

## 5.4 Subscribing to ROS Topics

A ROS node (e.g. the hexacopter main controller) may subscribe to ROS topics in order to receive data published by other nodes, e.g. the sensor in the V-REP. Thus, a virtual object can be controlled during simulation by means of subscribing ROS topics within V-REP scripts. For instance, the rotor of the virtual hexacopter must receive throttle signals published by the ROS node created in the `main()` function in `rosvrep_controller.cpp` file (see Sect. 4.5). Listing 1.22 shows the fragment of *HexacopterPlus* object script that enables V-REP to subscribe topics and receive the ROS messages.

**Listing 1.22** Fragment of Lua script of the *HexacopterPlus* object

```

1 ... // previous lines are omitted
2 -- Rotors Subscribers
3 simExtROS_enableSubscriber('propFRONT', 1,
   simros_strmcmd_set_float_signal, -1,-1, 'propFRONT')
4 simExtROS_enableSubscriber('Yaw', 1,
   simros_strmcmd_set_float_signal, -1,-1, 'Yaw')
5 ... // next lines are omitted

```

The parameters of `simExtROS_enableSubscriber` function are similar to the `simExtROS_enablePublisher` function (see Sect. 5.3), however, there is a difference in the specification on how data are handled. The `simros_strmcmd_set_float_signal` parameter indicates that V-REP subscribes to the topic, while `simros_strmcmd_get_float_signal` indicates that V-REP publishes in the topic. The last parameter is a signal that is used in a Lua script associated with any objects from V-REP virtual environment. For instance, `propFRONT` signal is used in the script of *propeller\_jointFRONT* object by means of calling `simGetFloatSignal` function in the parameters list of `simSetJointTargetVelocity` function as shown in Listing 1.23. A V-REP signal is a global variable. When the `simExtROS_enableSubscriber` is executed, a value is assigned to that global variable. If such a global variable does not exist, the `simExtROS_enableSubscriber` creates it.

**Listing 1.23** Fragment of Lua script of the *HexacopterPlus* object

```

1 ... // previous lines are omitted
2 simSetJointTargetVelocity(simGetObjectHandle('
   propeller_jointFRONT'),
3                               simGetFloatSignal('propFRONT')*-200)
4 ... // next lines are omitted

```

## 5.5 Publishing Images from V-REP

Many robotic applications usually demand some sort of video processing in order to perform advanced tasks. Cameras are commonly used in computational vision tasks, e.g. for collision avoidance while the robot is moving or for mapping and navigating towards the environment [32]. Thus, it is important to provide means

for video processing during the simulation phase of a robot design. This sections presents how to setup a virtual video camera in V-REP and how to stream the capture video to a ROS node.

It is possible to many distinct data types within topics published or subscribe between ROS and V-REP, including images from the virtual video sensor. A ROS node receives an image that can be processed using the OpenCV API [33]. Although the OpenCV library is not a part of ROS, `vision_opencv` package [34] provides an interface between ROS and the OpenCV library. This package was used in the camera application<sup>6</sup> implemented in `rosvrep_camera.cpp`. Although this tutorial does not discuss video processing, we show how to setup a ROS topic and stream the video captured within the V-REP simulated environment. For that, the *HexacopterPlus* has an object named `vCamera` attached onto its frame. `vCamera` object is a video sensor that streams images captured during simulation. Using the `simros_strmcmd_get_vision_sensor_image` type, V-REP is able to send images to a ROS node. Listing 1.24 depicts a fragment of the Lua script from *HexacopterPlus* object. The video streamed from the virtual camera can be seen in the camera application.

**Listing 1.24** Fragment of Lua script from the *HexacopterPlus* object

```

1 ... // previous lines are omitted
2 vCameraHandle=simGetObjectHandle('vCamera')
3 topicName=simExtROS_enablePublisher('vCamera',1,
4         simros_strmcmd_get_vision_sensor_image,
5         vCameraHandle,0,'')
6 ...

```

## 5.6 Running the Sample Scenarios

Our package provides two scenes that are located in the `scenes` subdirectory. The first scene was modeled in `Hexacopter.ttt` file. It was created to illustrate how the hexacopter was created in V-REP. The second scene was modeled in `rosHexaPlus_scene.ttt`. This is a more elaborated scene whose environment presents trees and textures. The aim is to illustrate the hexacopter movements, and hence, it is the scene used in the rest of this section.

Before starting the scene execution, the reader must ensure that `roscore` and V-REP are running (see Sect. 4.2). In the V-REP, open the `rosHexaPlus_scene.ttt` file using the menu command `File → Open scene`. The reader must start the simulation by either choosing the menu option `Simulation → Start Simulation` or by clicking on `Start/resume simulation` in the toolbar.

Go to the terminal that is executing the `rosvrep_panel` application as shown in Fig. 18. The first set of coordinates must be inserted in the order presented in in Table 2, aiming to command the hexacopter to fly around the environment. It is

<sup>6</sup>This application was created for debugging purposes and it is not discussed in this chapter.





**Fig. 19** The V-REP screenshot of short distance execution

**Table 3** First test: hexacopter flying around the environment

Commands sequence	Target coordinates			Heading	
	X	Y	Z	X	Y
1	-20	18	3	-18	23
2	-18	18	3	-10	18
3	-23	22	5	-10	18

## 6 Final Remarks

This chapter describes a tutorial on how to implement a control system based on Fuzzy Logic. The movement control system of an hexacopter is used as a case study. A ROS package that includes a fuzzy library was presented. By using such a package, we discussed how to integrate the commercial robotics simulation environment named V-REP with a fuzzy control system implemented using ROS infrastructure. Instructions on how to perform a simulation with V-REP were presented. Therefore, this tutorial provides additional knowledge on using different tools for designing ROS-based systems.

This tutorial can be used as a starting point to make more experiences. The reader can modify or improve the proposed fuzzy control system by changing the “.fz” files. There is no need to modify the controller main controller implementation in `rosvrep_controller.cpp` file. As a suggestion to further improve the skills on using the propose fuzzy package and V-REP, the reader could create another ROS node which may act as a mission controller by sending automatically a set of target positions.



## References

1. Koslosky, E., et al. Hexacopter tutorial package. <https://github.com/ekosky/hexaplus-ros-tutorial.git>. Accessed Nov 2016.
2. Bipin, K., V. Duggal, and K.M. Krishna. 2015. Autonomous navigation of generic monocular quadcopter in natural environment. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 1063–1070.
3. Haque, M.R., M. Muhammad, D. Swarnaker, and M. Arifuzzaman. 2014. Autonomous quadcopter for product home delivery. In *2014 International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*, 1–5.
4. Leishman, R., J. Macdonald, T. McLain, and R. Beard. 2012. Relative navigation and control of a hexacopter. In *2012 IEEE International Conference on Robotics and Automation (ICRA)*, 4937–4942.
5. Ahmed, O.A., M. Latief, M.A. Ali, and R. Akmeliawati. 2015. Stabilization and control of autonomous hexacopter via visual-servoing and cascaded-proportional and derivative (PD) controllers. In *2015 6th International Conference on Automation, Robotics and Applications (ICARA)*, 542–549.
6. Alaimo, A., V. Artale, C.L.R. Milazzo, and A. Ricciardello. 2014. PID controller applied to hexacopter flight. *Journal of Intelligent & Robotic Systems* 73 (1–4): 261–270.
7. Oldziej, D., and Z. Gosiewski. 2013. Modelling of dynamic and control of six-rotor autonomous unmanned aerial vehicle. *Solid State Phenomena* 198: 220–225.
8. Collotta, M., G. Pau, and R. Caponetto. 2014. A real-time system based on a neural network model to control hexacopter trajectories. In *2014 International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM)*, 222–227.
9. Artale, V., C.L. Milazzo, C. Orlando, and A. Ricciardello. 2015. Genetic algorithm applied to the stabilization control of a hexarotor. In *Proceedings of the International Conference on Numerical Analysis and Applied Mathematics 2014 (ICNAAM-2014)*, 222–227.
10. Bacik, J., D. Perdukova, and P. Fedor. 2015. Design of fuzzy controller for hexacopter position control. *Artificial Intelligence Perspectives and Applications*, 193–202. Berlin: Springer.
11. Koslosky, E., M.A. Wehrmeister, J.A. Fabro, and A.S. Oliveira. 2016. On using fuzzy logic to control a simulated hexacopter carrying an attached pendulum. In *Designing with Computational Intelligence*, vol. 664, ed. N. Nedjah, H.S. Lopes, and L.M. Mourelle. Studies in Computational Intelligence. Berlin: Springer. 01–32 Expected publication on Dec. 2016.
12. Open Source Robotics Foundation: ROS basic tutorials. <http://wiki.ros.org/ROS/Tutorials>. Accessed March 2016.
13. Coppel Robotics: V-REP: Virtual robot experimentation platform. <http://www.coppeliarobotics.com>. Accessed March 2016.
14. Coppel Robotics: V-REP bubblerob tutorial. <http://www.coppeliarobotics.com/helpFiles/en/bubbleRobTutorial.htm>. Accessed March 2016.
15. Coppel Robotics: V-REP tutorial for ROS indigo integration. <http://www.coppeliarobotics.com/helpFiles/en/rosTutorialIndigo.htm>. Accessed March 2016.
16. Yoshida, K., I. Kawanishi, and H. Kawabe. 1997. Stabilizing control for a single pendulum by moving the center of gravity: theory and experiment. In *American Control Conference, 1997. Proceedings of the 1997*, vol. 5, 3405–3410.
17. Passino, K.M., and S. Yurkovich. 1998. *Fuzzy Control*. Reading: Addison-Wesley.
18. Hwang, G.C., and S.C. Lin. 1992. A stability approach to fuzzy control design for nonlinear systems. *Fuzzy Sets and Systems* 48 (3): 279–287.
19. Pedro, J.O., and C. Mathe. 2015. Nonlinear direct adaptive control of quadrotor UAV using fuzzy logic technique. In *2015 10th Asian Control Conference (ASCC)*, 1–6.
20. Pedrycz, W., and F. Gomide. 2007. *RuleBased Fuzzy Models*, 276–334. New York: Wiley-IEEE Press.
21. Open Source Robotics Foundation: ROS remapping. <http://wiki.ros.org/Remapping%20Arguments>. Accessed March 2016.



22. Chak, Y.C., and R. Varatharajoo. 2014. A heuristic cascading fuzzy logic approach to reactive navigation for UAV. *IJUM Engineering Journal, Selangor - Malaysia* 15 (2).
23. Sureshkumar, V., and K. Cohen. Autonomous control of a quadrotor UAV using fuzzy logic. *Unisys Digita - Journal of Unmanned System Technology, Cincinnati, Ohio*.
24. Eusebiu Marcu, C.B. UAV fuzzy logic control system stability analysis in the sense of Lyapunov. *UPB Scientific Bulletin, Series D* 76 (2).
25. Abeywardena, D.M.W., L.A.K. Amaratunga, S.A.A. Shakoor, and S.R. Munasinghe. 2009. A velocity feedback fuzzy logic controller for stable hovering of a quad rotor UAV. In *2009 International Conference on Industrial and Information Systems (ICIIS)*, 558–562.
26. Gomez, J.F., and M. Jamshidi. 2011. Fuzzy adaptive control for a UAV. *Journal of Intelligent & Robotic Systems* 62 (2): 271–293.
27. Limnaios, G., and N. Tsourveloudis. 2012. Fuzzy logic controller for a mini coaxial indoor helicopter. *Journal of Intelligent & Robotic Systems* 65 (1): 187–201.
28. Ierusalimsky, R., W. Celes, and L.H. de Figueiredo. 2016. Lua documentation. <https://www.lua.org/>. Accessed March 2016.
29. Coppelia Robotics: V-REP help. <http://www.coppeliarobotics.com/helpFiles/>. Accessed March 2016.
30. Coppelia Robotics: V-REP download page. <http://www.coppeliarobotics.com/downloads.html>. Accessed March 2016.
31. Coppelia Robotics: ROS publisher typer for V-REP. <http://www.coppeliarobotics.com/helpFiles/en/rosPublisherTypes.htm>. Accessed March 2016.
32. Steder, B., G. Grisetti, C. Stachniss, and W. Burgard. 2008. Visual SLAM for flying vehicles. *IEEE Transactions on Robotics* 24 (5): 1088–1093.
33. Itseez: *OpenCV* - Open Source Computer Vision Library. <http://opencv.org/>. Accessed Nov 2016.
34. Mihelich, P., and J. Bowman. 2016. *vision\_openCV* documentation. Accessed March 2016.

## Author Biographies

**Emanuel Koslosky** is Master's degree student in the applied computing and embedded systems. As a student, he took classes about Mobile Robotics, Image Processing, Hardware Architecture for Embedded Systems, Operating Systems in Real Time. As a professional, he received Certifications of Oracle Real Application Clusters 11g Certified Implementation - Specialist, Oracle Database 10g Administrator Certified Professional - OCP, Oracle8i Database Administrator Certified Professional - OCP. He has professionally worked as programmer and developer since 1988 using languages such as C/C++, Oracle Pro\*C/C++, Pro\*COBOL, Java and Oracle Tools like Oracle Designer, Oracle Developer, as a Database Administrator worked with high availability and scalability environment, also as a System Administrator of Oracle e-Business Suite - EBS.

**Marco Aurélio Wehrmeister** received the Ph.D. degree in Computer Science from the Federal University of Rio Grande do Sul (Brazil) and the University of Paderborn (Germany) in 2009 (double-degree). In 2009, he worked as Lecturer and Postdoctoral Researcher for the Federal University of Santa Catarina (Brazil). From 2010 to 2013, he worked as tenure track Professor with the Department of Computer Science from the Santa Catarina State University (UDESC, Brazil). Since 2013, he has been working as a tenure track Professor with the Department of Informatics from the Federal University of Technology - Paraná (UTFPR, Brazil). From 2014 to 2016, he was the Head of the M.Sc. course on Applied Computing of UTFPR. In 2015, Prof. Dr. Wehrmeister was a Visiting Fellow (short stay) with School of Electronic, Electrical and Systems Engineering from the University of Birmingham (UK). Prof. Dr. Wehrmeister's thesis was selected by the Brazilian Computer Society as one of the six best theses on Computer Science in 2009. He is member of the special commission on Computing Systems Engineering of the Brazilian Computer Society. Since 2015, he is a member of the IFIP Working Group 10.2 on Embedded Systems.

His research interests are in the areas of embedded and real-time systems, aerial robots, model-driven engineering, and hardware/software engineering for embedded systems and robotics. Prof. Dr. Wehrmeister has co-authored more than 70 papers in international peer-reviewed journals and conference proceedings. He has been involved in various research projects funded by Brazilian R&D agencies.

**Andre Schneider de Oliveira** holds a degree in Computer Engineering from the University of Vale do Itajaí (2004), master's degree in Mechanical Engineering from the Federal University of Santa Catarina (2007) and Doctorate in Automation and Systems Engineering from the Federal University of Santa Catarina (2011). He is currently Assistant Professor at the Federal Technological University of Paran - Curitiba campus. He has carried out research in Electrical Engineering with emphasis on Robotics, Mechatronics and Automation, working mainly with the following topics: navigation and positioning of mobile robots; autonomous and intelligent systems; perception and environmental identification; and control systems for navigation.

**João Alberto Fabro** is an Associate Professor at Federal University of Technology - Parana (UTFPR), where he has been working since 2008. From 1998 to 2007, he was with the State University of West-Parana (UNIOESTE). He has an undergraduate degree in Informatics, from Federal University of Paran (UFPR 1994), a Master's Degree in Computing and Electric Engineering, from Campinas State University (UNICAMP 1996), a Ph.D. degree in Electric Engineering and Industrial Informatics(CPGEI) from UTFPR (2003) and recently actuated as a Post-Doc at the Faculty of Engineering, University of Porto, Portugal (FEUP, 2014). He has experience in Computer Science, specially Computational Intelligence, actively researching on the following subjects: Computational Intelligence (neural networks, evolutionary computing and fuzzy systems), and Autonomous Mobile Robotics. Since 2009, he has participated in several Robotics Competitions, in Brazil, Latin America and World Robocup, both with soccer robots and service robots.

Robot Operating System (ROS)

The Complete Reference (Volume 2)

Koubaa, A. (Ed.)

2017, XVII, 655 p. 344 illus., 256 illus. in color.,

Hardcover

ISBN: 978-3-319-54926-2