Part D

Simulation

D.1 Introduction

Simulation in Unmanned Aerial Vehicles is a very broad topic. It mainly depends on the application which is under testing. There are different simulators if the operators need to learn how to fly a UAV compared to the ones that are useful for the engineers to help them design an effective chassis. In my project's case, I need a simulation in order to investigate the behaviour of developed applications in certain circumstances. The key features that I needed were the following:

- A realistic UAV simulation in terms of the physics and dynamics that applied.
- Ability to utilize different types of sensors and customise their specifications.
- Ability to build custom landscapes and test the behaviour of the algorithms for various different cases.
- Be as close as possible to the structure of a real UAV in terms of the flight controller, to the input commands for navigation and to the individual devices from which it is built, like the global positioning system (GPS) or the inertial measurement unit (IMU).
- The ability to apply the developed algorithm to a real UAV with minor modifications.

After an extended research of many available approaches, all the aforementioned requirements lead us to a specific solution, which is the combination of ROS and Gazebo. Some of the solutions that were explored but were rejected were the following:

- Building a custom simulation from scratch using Matlab. During the second semester there was a related unit that gave us the opportunity

to build a simple UAV simulation. However, it was assumed that the UAV moves in 2D and it did not simulate the flight dynamics of the UAV. Generally speaking, the simulation was intended to test UAV swarm robotics algorithms. Consequently, the effort to build a simulation in this way from scratch was a time consuming procedure with uncertainty about the effective fulfilment of the requirements.

- Available simulation found on Internet based on Matlab and Simulink. There are several simulations of this type and a possible approach could be to adapt one of those and customise it. Most of them have been built as master or part of PhD theses. The disadvantages of using them are that they are primarily focused on simulating the dynamics and they do not include sensors. Moreover, they do not offer a build in feature of designing realistic landscapes. In general, depending on the quality, most of them have poor comments on the code and most importantly they do not offer support. Consequently, this solution was ineffective since it may have meant debugging others' code rather than building creative tasks.
- There was the possibility to use Webots (Cyberbotics.com, 2015). This is a development environment used to model, program and simulate mobile robots. It offers the choice of using a wide range of simulated sensors and actuators. The robot behavior can be tested in physically realistic worlds. It is a popular tool for research, well documented and continuously maintained for over 18 years. The reason why it was not adopted in my research was that it is not free of charge. It offers a 30-day trial version but after this period a valid license should be acquired.

In short, various approaches were considered for simulating a UAV executing different behavioral algorithms. The most promising and professional way seemed to be using the combination of ROS and Gazebo. Following this method in order to carry out my experiments finally proved a successful decision considering the outcomes of my research.

In the "Simulation Topics" appendix there is a section related with a general description of ROS, Gazebo as well as a terminology section, which is extensively used in the remaining of this chapter.

D.2 UAV package

When using ROS and Gazebo in robotics simulation what the roboticist want to avoid is the reinvention of the wheel. Therefore having these tools installed and working, the users could utilise packages that implement UAVs equipped with all the required features. Depending on the application, there are UAV implementations available which have many common characteristics but also differences that help us when decision making. For my project, I explored the following three packages:

- Tum simulator

This package (goo.gl/ZBjaAY) contains the implementation of a Gazebo simulation for the Ardrone 2.0 (fig.D.1). It is a nice simple quad-rotor simulator developed at the Technical University of Munich. The simulator supports both Ardrone 1.0 and 2.0 versions but it is optimised for the latter. It has a built-in support for joystick manipulation, which means that the users can use it to fly the real drone as well as the simulator. The Tum-simulator is equipped with 2 cameras, one front and one bottom and a height sonar sensor.

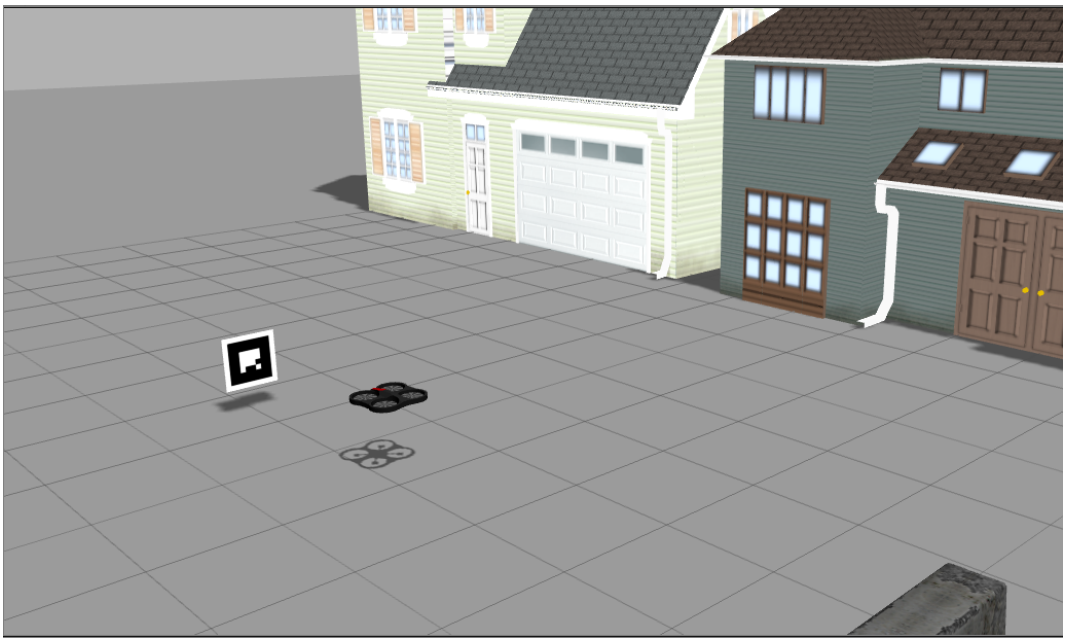

Fig.D.1 Tum simulator in action (image from answers.ros.org)

There are two reasons why this simulator was not adapted in this project. Firstly, it is executable in ROS Fuerte released in 2012, four versions older compared to the current. After this version, ROS changed completely the low-level built system macros and infrastructure (catkin). Consequently, I wanted to avoid working with an outdated framework, which would downgrade all the other tools to ensure compatibility. Secondly, the provided sensors are not suitable for my project. This simulator is more focused on computer vision applications where the users capture video from the cameras and process data, using for example OpenCV, to make navigation decisions. There is always the choice of adding or developing custom sensors but it is safer to find a simulator that already integrates them.

- PX4 SITL

This simulator developed by PX4 Autopilot Project for the software in the loop simulation (Pixhawk.org, 2015). The ROS SITL setup allows the users to run certain PX4 Firmware modules, which use the multiplatform wrappers within ROS. These applications publish data in the related ROS topics. The PX4 simulator is compatible with ROS Indigo, which is a very recent ROS version. Likewise the previous, it allows joystick manipulation. It is

noteworthy that includes two UAV simulators, one based on Ardrone and the other on Iris (fig.D.2).

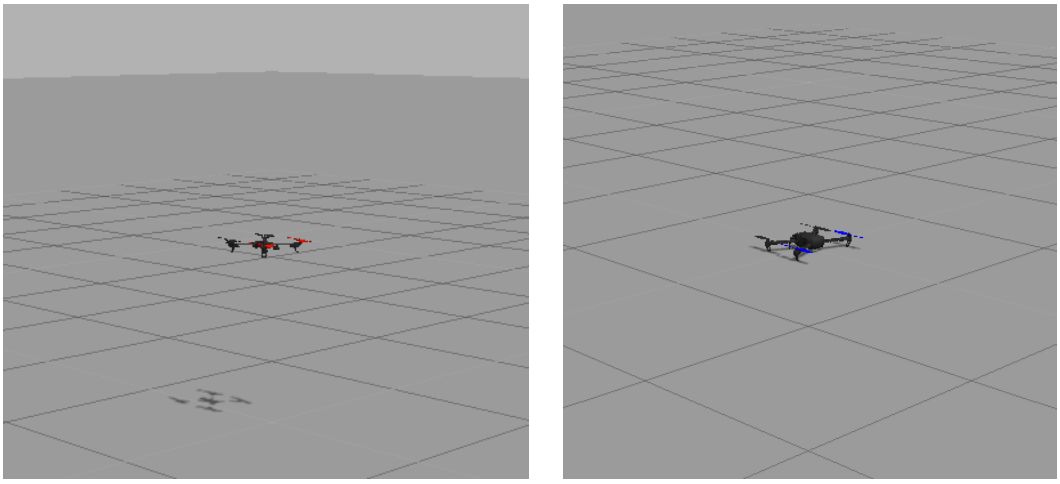

Fig.D.2 PX4 SITL simulation using Ardrone and Iris UAVs in action

The additional characteristic that this simulator includes, compared to the others, is the MAVROS (Micro Air Vehicle ROS) package. This package provides a communication driver for various autopilots with MAVLink communication protocol. In addition, it provides UDP MAVLink bridge for the ground control station. Using this protocol ROS can send specific commands to the UAV, which is essentially what this simulator is specialised for. To illustrate this, using ROS in a companion computer, navigation commands can be sent directly to the flight controller produced by a robotics algorithm via MAVROS. This is called off-board control. This simulation is intended for those cases where the users want to test the MAVROS commands before applying them to the real UAV.

Since it is actually a port of the PX4 flight stack in ROS, it is a high fidelity simulation coming with plenty of capabilities like real UAV. For example, in order to begin it is required to arm. Some of the available features provided are:

- Inertial measurement unit (IMU) which provides linear acceleration, angular velocity, atmospheric pressure and altitude
- Compass which provides heading

- GPS which provides longitude, latitude and altitude
- PX4flow optical flow sensor which is integrated to provide ground distance
- Manual set of the motor speed for each individual motor

There are reasons why I did not select this simulator for my task. Firstly, it has some problems when I used it in combination with the RViz, the ROS visualization tool. This is the all-seeing ROS visualisation utility. This tool becomes very important when there is a need for customisation of the sensors since it visualises the laser scans and the ultrasonic lobes. Secondly, it did not include laser or ultrasonic sensors and my preference was to have them integrated to avoid misconfigurations because of my short experience. Thirdly, that fact that it is equipped with MAVROS and includes highly detailed UAV configuration, it introduces an unwanted complexity for performing behavioural analysis of navigation algorithms. As I already explained, from my perspective the PX4 SITL is focused on different tasks compared to my requirements.

- Hector quad-rotor

The Hector quad-rotor simulator is a framework that implements the simulation of quad-rotor UAVs employing ROS and the Gazebo simulator (fig.D.3). It is developed by the Research Training Group in Darmstadt Technical University (Meyer et al., 2012). The dynamics model of the quad-rotor has been parameterized using wind tunnel tests and validated by a comparison of simulated and real flight data. It is available for all the latest ROS versions from 2012 and on (Fuerte, Groovy, Hydro and Indigo).

Hector quad-rotor comes with a wide range of available sensors. It is equipped with an Inertial Measurement Unit (IMU) measures the angular velocities and accelerations of the vehicle's body in the inertial frame. It provides ultrasonic sensor to measure the distance from the ground in a specific range. In addition, it has a Magnetic Field sensor to measure the heading of the UAV. A GPS receiver is also installed to provide pseudo range

measurements. Aside from the standard devices, it also offers additional and more sophisticated ones. More precisely, it provides the Hokuyo UTM-30LX as laser scanner, a depth camera like Kinect and a normal camera. Furthermore, like all the other simulators, it is possible to manipulate it using joystick (like x-box controller) as well as keyboard.

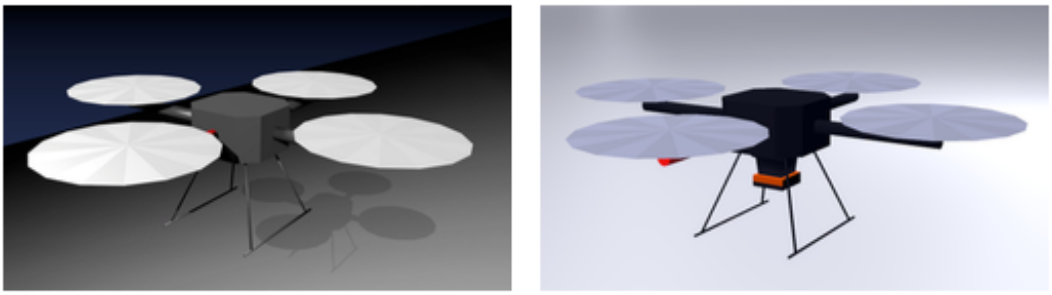

Fig.D.3 The Hector quad rotor model with a Hokuyo UTM-30LX laser scanner mounted in the second picture (image from (Meyer et al., 2012))

The Hector quad-rotor is delivered as a ROS stack, which means that is a collection of packages and not only a single one. So, aside from the packages that are responsible for the modelling, the control and the Gazebo simulation of the UAV, it also includes a visual SLAM (Simultaneous Localisation and Mapping) package to demonstrate the great potentials to build robotics applications.

Hector quad-rotor combines all the types of the sensors, which I wanted to employ in my simulation in a well-structured package, as complex as it is required for a comprehensive realistic simulation. Everything is located in the place where I expected to find it. This enabled me to customise the configuration and move it closer to my real setup. It is also well documented with many related questions in the ROS support forums. My working system is configured in ROS Hydro version with all the related tools working flawlessly like RViz. The same implies in the ROS Indigo as well. All in all, from my experience with working with it, it is a reliable and stable tool that helps to focus on the details of the developing robotics applications leaving away all the other distracting points.

D.3 UAV configuration in simulator

The rich features of Hector quad-rotor package were utilized for simulation in the current project. In order to build a UAV close to my working scenario, I needed to refine the sensors positions and operation. The selection and the settings of the sensors were based on the available equipment I had for the real UAV.

From the available sensors provided in the hector UAV I utilized the following:

- IMU

  The IMU was exploited to report the heading of the UAV (yaw angle). An alternative method to obtain this value was the Magnetic Field Sensor (the compass).
- Sonar

  The ultrasonic sensor transmits short ultrasound impulses. Sonar reports the distance corresponding to the first echo returned from the ground or any other object within its FOV (Field Of View). The comprehensive way that the simulator was built allows modifying all the related parameters of the ultrasonic sensor. More precisely, in the file (goo.gl/RCZ1u8) the users can change the following characteristics:
  - Update rate (in Hz)
  - Minimum and maximum range (in m)
  - The name of the published topic
  - The position, the orientation and the parent frame link in which is attached

  If the users want to experiment with more advanced characteristics, using the parameters inside the file (goo.gl/daIwSo), they can alter parameters such as resolution, Gaussian noise, moments of inertia and many more. In this project the sonar was used twice. Firstly, an ultrasonic sensor

installed under the central hub, facing the ground, to report the distance from the ground –that is the altitude. Secondly, an ultrasonic sensor installed in the frame arm oriented to the back in order to measure the distance between the UAV and any object in the back. To add more devices in the UAV configuration, the users should edit this file (goo.gl/XJU7TU) accordingly.

- Laser scanner

  The Hokuyo UTM-30LX Scanning Laser Rangefinder is a great device having features like 270° area scanning range with 0.25° angular resolution at a speed of 25msec/scan. However, it is very expensive with a price approximately 5.000$. In my project the principle of using relatively cheap COTS technology did not justify the use of this device. Therefore, I did not use this device as it is. The simulator structure allowed me to alter the characteristics and degrade it in a simple laser sensor. Using this (goo.gl/ZzL6HI) file I could change:

  - The minimum and maximum angle from 270^{o} to 180^{o}.
  - The ray count from 1800 (in the 270^{o} range) to 180 (one scan per degree for my 180^{o} range)
  - And the update rate.

  Likewise, the users that want to experiment with advanced features of the laser can alter the related parameters in this file (goo.gl/FhyLzY).

Aside from the devices that I tested and utilised, I additionally experimented with the following:

- GPS

  The GPS reports longitude, latitude and altitude measures. There are two useful ways to exploit these data. The first is using an approximation, to estimate the distance between two points. This accomplished by converting the (lat, lon) coordinates to meters. The second is to use the altitude. However, in my case it was not accurate, as it is expected. Furthermore, I couldn't rely on that value in order to hover constantly in low height.

- Barometric sensor

  This sensor measures the air pressure as the primary means for determining altitude, also called Pressure Altitude. Like with the sonar sensor, the reported height is utilised to lock the altitude during the flight. However, if the air pressure is changing in the flight area due to extreme weather, the UAV will follow the air pressure change rather than actual altitude. In reality when the altitude is less than 8 meters, the sonar is the main device of obtaining the altitude and not the barometer. In the simulation, the measurement of the barometer in 1m above the ground was even a negative number. Consequently, this sensor was not used in the simulation.
- Front camera

  In the default configuration of the Hector simulator a front camera is enabled. I did not experiment in depth with this device because I did not have any intention to use computer vision as part of my application.

The complete setup of the simulated UAV is shown in the following figure (fig.D.4). In this screenshot, a hector quad-rotor UAV equipped with a laser scanner and two sonars is visualised in rviz.

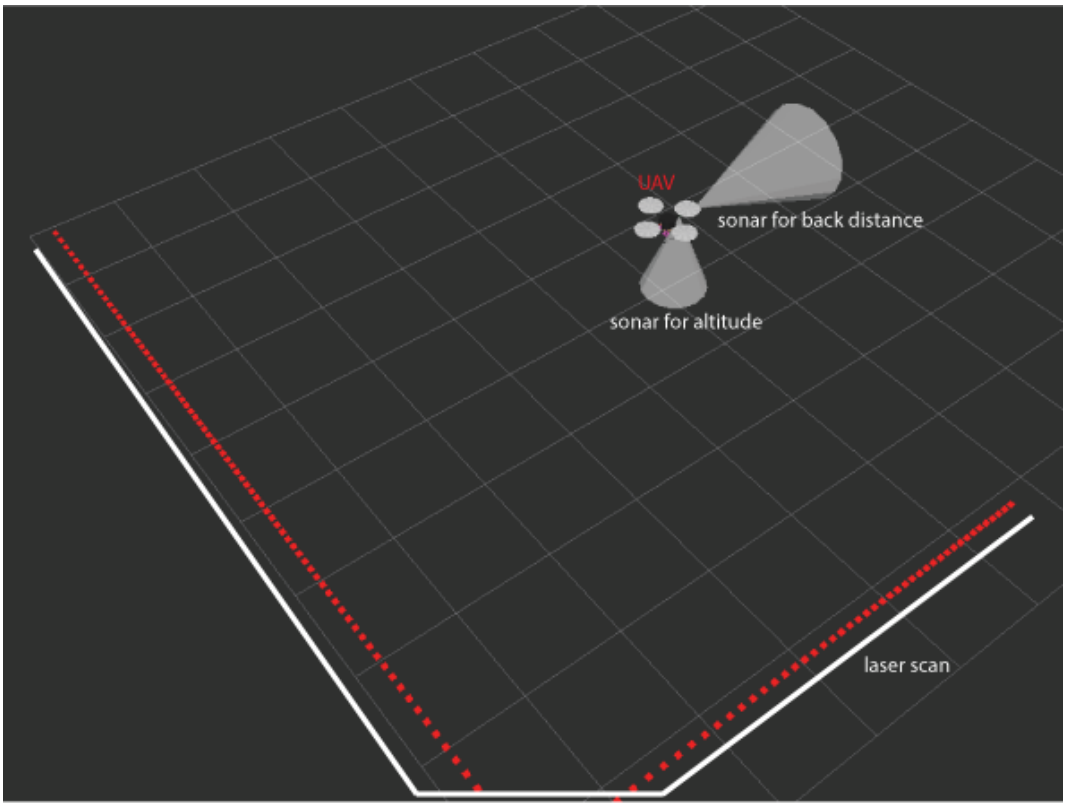

Fig.D.4 Visualisation of a simulated UAV using the ROS rviz tool. This screenshot illustrates the two sonar lobes and the laser scanning of a corner (two perpendicular walls)

D.4 The radiation mapping and simulation requirements

It is important to explain clearly the task of UAV taking radiation measurements so as to gain better understanding of the simulation. In a real UAV for radiation measurements, a gamma spectrometer is also installed in addition to the proximity sensors. Based on the radiation mapping techniques, to obtain high quality measurements, the UAV should fly close to the inspected surface and in a constant distance. This means that the UAV should fly in low height above the surface and this distance should be locked. In the real UAVs there is flight mode for this case called Altitude Hold Mode.

A desired feature during the radiation mapping is the obstacle avoidance capability. To illustrate this, as the operators control manually the UAV, if

this gets near an obstacle, in the direction of its movement, an assisted behaviour will be performed. Thus, if the UAV finds an obstacle in front of it, it will assess the surrounding environment and accordingly it will accomplish an avoidance maneuver. At the end of this intervention the position of the UAV should be pointing to the open space and in the predefined locked altitude. After this point the control returns to the operators who are free to continue without worries till the next one. Simultaneously, if for any reason the UAV sense an object around it that is closer than one meter, it will proceed to emergency landing.

The described obstacle avoidance algorithm was implemented in the simulation environment to test its behaviour and get an assessment of its effectiveness. The technical details of the implementation are explained in the following sections. All the source files created for the obstacle avoidance teleoperation algorithm has been grouped in a ROS package called my_hector (goo.gl/DpgSre). The required changes in the existing files of the Hector quad-rotor ROS stack are explicitly cited in the related chapters with direct links.

D.5 Topics nodes

The key feature of the radiation mapping is the capability to maintain a constant distance from the floor, even if this is the ground or the terrace of a building. The sensor that was employed for this task was the sonar, which is mounted under the central hub looking downwards. This device in the simulation publishes continuously, with a rate of 10Hz, the measured distance. It has a range from 3cm to 3m.

In order to implement the required behaviour, I designed the hover node (goo.gl/vTajQO). Essentially, this node gets input from the sonar to access the actual altitude of the UAV. In addition, it takes input from a custom topic *hover_height* with messages of type *geometry_msgs* (fig.D.5). This is the

interface of the node with the ROS ecosystem. Every node that wants to adjust the altitude hold of the UAV can set up a publisher for this topic and send the appropriate message to the hover node. Inside this message the z component commands the node to lock the altitude in this specific value.

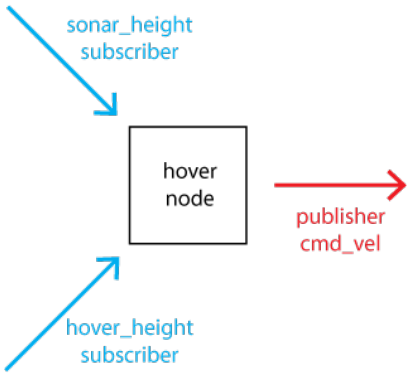

Fig.D.5 The messages' flow for the hover node

Since the node has all the data needed for its operation, it proceeds to the requested behaviour. Technically, both subscribed topics are connected with callback functions. The *heightCallBack*, related to *hover_height* receives the desired altitude hold and checks if it is inside the range of the ultrasonic sensor. In the second callback function *heightCallBack*, which corresponds to the *sonar_height* topic, the real work of this node takes place. Inside this function there is a simple P-controller (fig.D.6). This controller calculates the difference of the desired and the current height and outputs a value adjusted according to a gain. This outcome is basically the linear velocity in the z direction. Consequently, using this value a *geometry_msgs/Twist* message is prepared and sent to the *cmd_vel* topic. As long as the altitude of the UAV is stable and equal to the desired, the node does not need to act. It is worth noting that this node is active throughout the simulation since the altitude hold is a requirement in the flight mode. To achieve this automatically, the relevant command has been included inside the simulation launch file, for example in (goo.gl/jBtQsi).

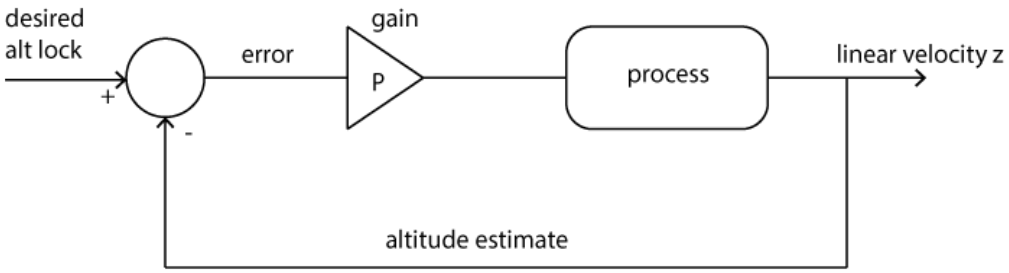

Fig.D.6 The schematic of the P-controller inside the hover node

Besides the hover node, a node responsible to change the heading (the yaw angle) is also developed called turn node. The operation principal is the identical to the hover node. The difference is that it gets the reading from the IMU sensor and publishes an angular velocity in the z axis instead of a linear velocity. A figure that summarizes the subscribed and published inputs is depicted in (fig.D.7). This node, based on topics, was replaced at the final obstacle avoidance application from a node, based an actionlib, for reasons that will be explained further on.

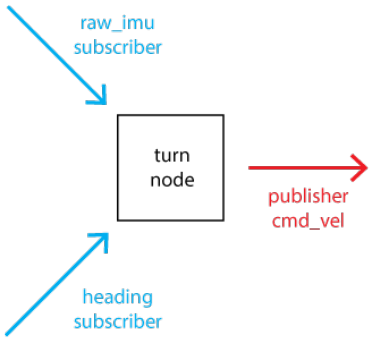

Fig.D.7 The messages' flow for the turn node

D.6 Actions nodes

In the introduction of ROS, it was described that there are many ways for the nodes to communicate with each other. Aside from the topics and the services, actionlibs is a powerful tool to design communication channels

between nodes. In particular cases, when the following conditions exist, there is the only way.

- When the task is required to take time and feedback of the progress is needed, as well as a sign of completion or failure at the end. In this way, the application knows exactly the state of the ordered task and when the correct time to continue has come. This cannot be implemented using topics or services.
- When the application needs a way to cancel the ordered command.

For these reasons I decided to build action servers nodes in order to change UAV's heading and altitude lock. There is a predefined way that an actionlib server is built. In this section I will explain how the change-heading server was designed. First, in order for the client and server to communicate, there is the need to define three different types of messages on which they communicate. More precisely, the Goal, Feedback, and Result messages with which clients and servers communicate should be defined.

- Goal: To accomplish tasks using actions, the notion of a goal is introduced that can be sent to an action server by an action client. In my case the desired heading (in radians) is sent.
- Feedback: This message provides server implementers a way to tell an action client about the incremental progress of a goal. In my case, the difference between the desired heading and the current heading is published.
- Result: A result is sent from the action server to the action client upon completion of the goal. The difference with the feedback is that it is sent only once. This is also useful when the purpose of the action is to provide some sort of information at the end. In my case the result message means the successful completion of the ordered task and it is the accomplished heading.

Technically, to create new type of messages there is a need to define them in a specific action file. Then this file should be properly declared in the

CMakeList.txt file of the package. The next important step is to execute the *catkin_make* process. Upon the completion of this, the required C++ header files containing the messages definitions have been created in a specific folder in the devel section of the ROS project. Now, the developers can use these messages merely by including the required headers.

The moment the infrastructure is complete, the design of the server can be implemented. For this particular server there is a need to know the actual heading. An easy and reliable way to obtain the heading is to employ the IMU sensor. Consequently the heading-change server has to subscribe to the *raw_imu* topic. In this topic *raw_imu* transmits quaternions. Using a couple of convenient methods it is possible to convert a quaternion to roll-pitch-yaw and access the UAV heading.

Following the creation of the server, three required function callbacks should be declared (fig.D.9). The first one is responsible for setting the goal in the action server. In my case, the *goalCB* callback function sets the desired heading. The second one is the preempt callback function. This action is event driven. It only executes when the callbacks occur, therefore a preempt callback is created to ensure that the action responds to a cancel request. In my case this callback function called *preemptCB*. The third callback function is the IMU callback function. Inside this function the change of heading takes place. At the beginning the current heading is obtained, as I described before. Then, the difference between the actual and the goal is calculated. At this point the feedback message is published. After that, this difference combined with a constant gain, form the input of a P-controller (fig.D.8). The outcome of this procedure is the angular velocity in the z axis. As it is expected at this point *geometry_msgs/Twist* message is created and published to the *cmd_vel* topic. At the end of this method, a termination condition is evaluated. More precisely, when the absolute difference between the desired and the actual heading is less than 3 degrees, the procedure terminates successfully. At this point the result message is published, only once. The reason why I introduce a 3-degree angular tolerance is that it is very easy to overshoot the rotation

and even a slight angular error can ruin the operation of the entire application. Empirically it was found that a tolerance of about 2.5 degrees gives acceptable results. The complete source file of the change-heading server can be found in this location (goo.gl/qbZDVR).

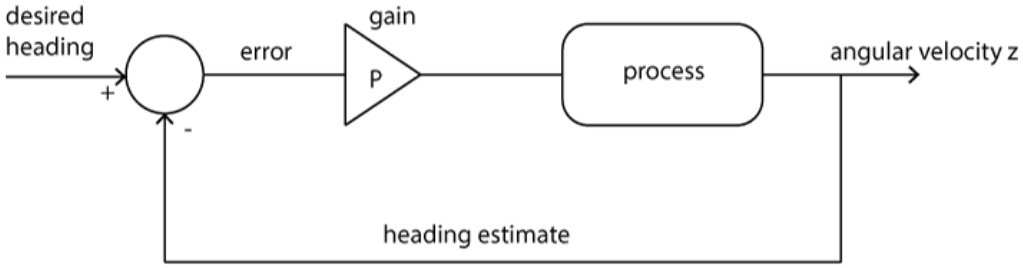

Fig.D.8 The schematic of the P-controller inside the change-heading server

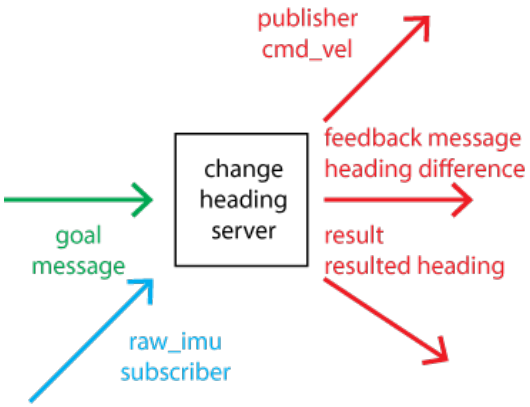

Fig.D.9 The messages flow for the change-heading action server

Before the action server was utilized as part of the application, the debugging and testing stage could be performed using a simple action client. In this client the goal heading is sent to the server. After that, the timeout duration is defined. During this period the client receives and reports the feedback messages. If the procedure ends within the time limit, it is considered successful otherwise the action has failed and an appropriate message is printed out. Since the actions are more complex than the other approaches, for every individual action server I have developed a suitable action client.

An additional important action server of the application is the hover-height server. This server is responsible to change the altitude to the new desired

value, which has been set as goal. The underlying logic and the structure of source file are similar the change-heading server that I have already described. The few differences that exist are the following as depicted in the figure (fig.D.10).

- The device that has undertaken the task to inform the current height is the sonar.
- The action server does not implement any controller. Instead of this, it takes advantage of the hover node and sends the required message to this mechanism in order to move the UAV vertically.

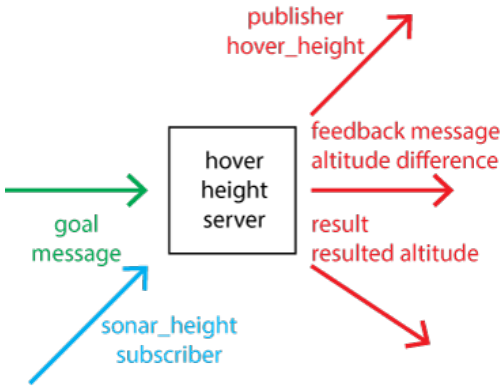

Fig.D.10 The messages' flow for the hover-height action server

The importance of both action servers in the obstacle avoidance algorithm will become apparent when the application will be explained. Nevertheless, before settling on the final version I had experimented with other implementations as well, which I included in my source code as available tools in the toolkit. The first one is a server (goo.gl/k6OjVX) which -instead of the desired altitude- accepts a vertical distance as its goal. For instance, if the goal is 1 the hover-distance action server will move the UAV 1m upwards. The second one is a server (goo.gl/KhdlHm) that performs the same action as the hover-distance but the measured height depends on the GPS altitude instead of the sonar's distance from the ground.

D.7 Obstacle Avoidance

Having developed a complete toolkit of useful tools, it is time to present the design of the obstacle avoidance algorithm. The heart of algorithm is the finite state machine (FSM). This allows adding intelligence to the UAV since it is able to switch between different behaviors according to its sensors readings.

It is important to describe the scenario of the simulation. In this simulation, as in the real UAV as well, the operator controls the flight manually using a joystick or the keyboard (fig.D.11). If the UAV encounters an obstacle in front of it or senses something closer than 1m around it, then the control passes to the obstacle avoidance algorithm. From this point, the UAV based on the FSM will perform a single or a series of maneuvers. When it assess that there is no obstacle to avoid, it will pass the control back to the operator. After the avoidance procedure, the algorithm will try to deliver the UAV in the initial heading that this had before the algorithm took the control. In complicated cases, when the UAV performs a sequence of maneuvers this could not be possible. This description corresponds to an assistive teleoperation solution where the application helps the operator by performing the avoidance maneuvers.

The aforementioned approach can be translated technically using the following techniques. The control input devices, described extensively in "Simulation Topics" appendix, send traditionally the *geometry_msgs/Twist* commands in the *cmd_vel* topic. To enable the obstacle avoidance application to alter the predefined flow of the commands, I structured it as a message getaway or a filter. To illustrate this, the input control devices have been remapped to publish the navigation commands to the *obs_cmd_vel* topic instead of the initial *cmd_vel*. The obs_cmd_vel is the one that the application listens to. If there is no need for intervention, the node is transparent. Whatever navigation message comes from the inputs, it is published automatically in the *cmd_vel* topic. If there is need to perform avoidance

maneuver, the node overrides the control and sends the algorithm's messages as input to *cmd_vel* topic.

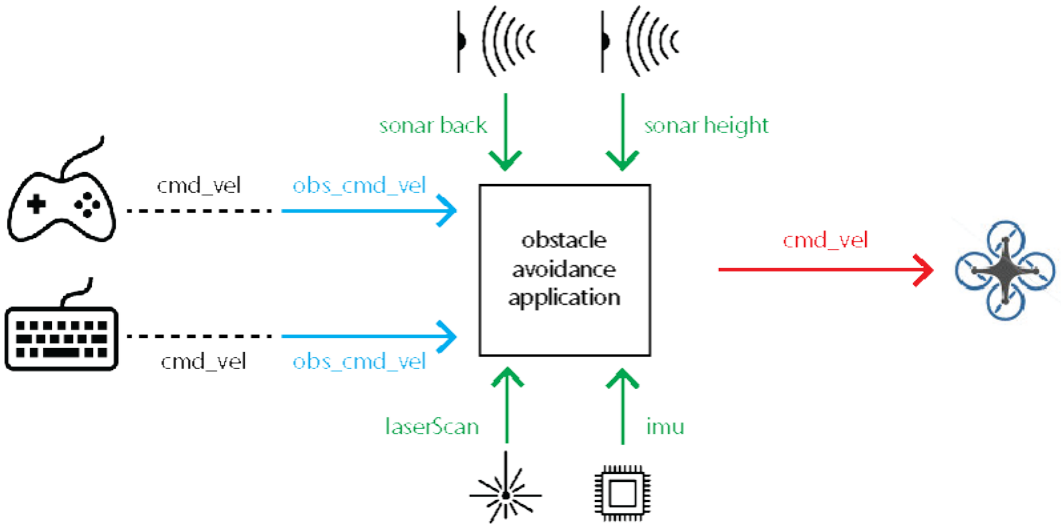

Fig.D.11 The messages flow for the obstacle avoidance application

At this point I should clarify the way that the UAV measures the distances that were used in the algorithm. The height and the back distance are measured using two ultrasonic sensors pointing towards the ground and back accordingly. For the distances in front and on both sides the laser scanner is used (fig.D.12). More precisely, having configured the laser to take measurements every 1-degree in 180^{o} range, it is possible to choose these directions from the related positions of the distances in the reported scanning. Consequently, the front distance is given by getting the minimum distance in the range from 70^{o} to 110^{o} heading. The left is the minimum distance in the range from 0 to 20^{o} and the right from 160^{o} to 180^{o}. This approach is closer to the real UAV since I do not have laser scanner sweeping of 180^{o}. Instead of this, I have an individual sensor on each side or a single laser taking measurements for every perpendicular direction.

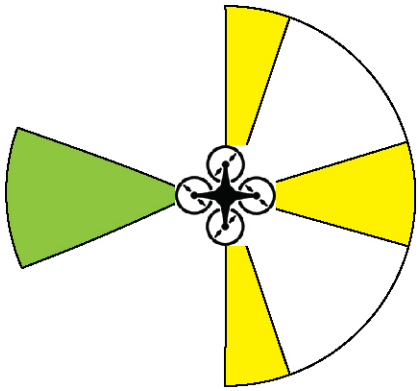

Fig.D.12 UAV proximity sensors. The sonar lobe in the back (green) and the segmented laser scanning in front and on the both sides (yellow)

Having talked about the finite state machine (FSM), a detailed explanation includes the description of the individual states and conditions that should hold to move between them (fig.D.13). The loop that executes the FSM is repeated at a constant rate of 20Hz.

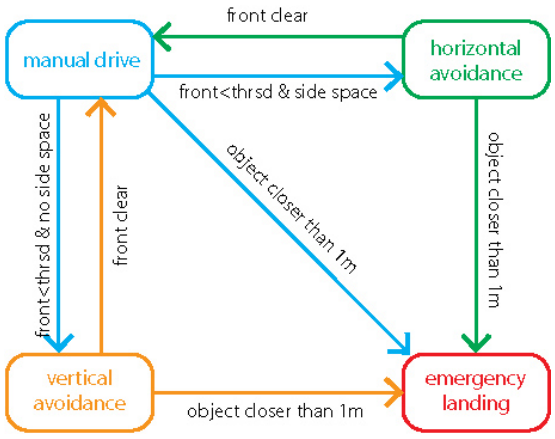

Fig.D.13 The UAV's finite state machine

State 0 – manual drive

In this state, input devices such as a keyboard or joystick control the UAV. Every time this state is executed, a safety check is performed. If the distance in front is less than a defined threshold a transition should occur. The destination of the transition depends on the distances on the left and the right of the UAV. There are the following possible cases:

- The distance on both sides is less than a threshold. This means that the UAV is at the end of a corridor, like a dead end. If this is the case then it changes to state 2.
- The distance on the left is greater than the one the right or vise versa. In situation like this, the UAV seems that flies near a corner. In this case the UAV will change to state 1 by rotating 90^o to the side where the open space is sensed.
- The distance on the left is equal to the distance on the left. This happened when there is no obstacle on both sides and the laser reports its highest range, or when there are obstacles further to the maximum range. In this case the UAV will randomly turn on one side or the other side and change to state 1.

Finally, if the UAV senses an object in distance closer to 1m at any direction (front, back, sides) then it performs emergency landing in state 3.

State 1 – Horizontal avoidance

In this state the UAV has already sensed that there is space to move in the one or the other side. Before entering this state a 90^o change of heading has been performed during the transition from state 0 to the side that has the greatest open space. The UAV starts to move forward and continuously measures the distance on the side where the obstacle exists. If at one point it is noted that the distance gets higher than a threshold, this means that there is no obstacle any more. This signs the end of the obstacle but not the end of the maneuver. After that the UAV will move a bit more and it will correct the heading back to the initial one which it had before the override. During the aforementioned procedure two simultaneous safety checks are executed at the same time.

- Since the UAV moves till it finds the end of the obstacle, it always measures the distance in front of it. If it finds that there is another obstacle ahead of it, it repeats the avoidance maneuver again and continues, maintaining the state to 1.
- If at any point for any reason it senses an object closer to 1m on the proximity directions (back, front, sides), it changes to state 3.

At the end of the procedure, if the avoidance maneuver completed successfully, the UAV moves back to state 0.

State 2 – Vertical avoidance

In this state the UAV has already discovered that it is at the end of a corridor. There is no point to explore the sides for horizontal escape. At this point, it will attempt a vertical overcome of the obstacle. Depending on the height of the obstacle in front of the UAV there are two available options:

- The first case implies if the obstacle's height is lower than the sonar's range. In this situation, the UAV will start to elevate and simultaneously measure the front distance. At a point before it reaches the upper limit, the UAV will sense that there is space ahead. Consequently, at this point it will stop the vertical translation and it will move forward to overcome the obstacle. After a certain predefined distance, the UAV will move back to the altitude hold position and the control will return to the operator in state 0.
- The second case happens when the UAV reaches the limit of the sonar range. At this point I do not want to move further up and obtain the height altitude measurement using the barometer or the GPS. These devices are not so accurate and the altitude hold mode becomes unstable. So if this is the case, the UAV will stop the vertical translation and it will change heading by 180^{o}. After that, it will move back to the predefined altitude hold position. Eventually, the UAV control will return to the manual operation in state 0. Similarly to the previous states, it is possible to change towards the emergency landing state 3 if any object is sensed closer than 1m.

State 3 – Emergency landing

During the movement of the UAV in manual mode or even when it performs avoidance maneuvers a safety check is concurrently run. In this check the UAV gathers the measurements in every perpendicular direction (0, 90^{o}, 180^{o}, 270^{o}) and if it encounters an object closer to 1m it will proceed to emergency landing. Since the altitude is locked to a low height this solution is

workable to enhance the safety of the UAV and the surrounding environment. The landing was preferred because it is difficult to trust a subtle maneuver with a hazard so close. However, if there is time for exhaustive testing, a more complicated behavior could be developed. In this emergency situation the control doesn't return back to the operator and it requires the physical intervention in the location of landing.

The source file in which the obstacle avoidance application has been implemented is located in (goo.gl/eG2mXn). It is a ROS node that acts as a central hub since it is subscriber to all the sensor and input control devices, publisher to the velocity control topic and client to action servers related to the heading and altitude.

D.8 Testing

During the development of the code the proper debugging took place in order to produce a functional application that will allow me to experiment with the behavior of the model. At this point, I exploited the dynamic features that Gazebo 3D simulator offers to the users. There is online an available library of objects which can easily be utilized to construct a landscape. These objects are declared in an XML-like file by defining the exact positions and orientations. In my case, the wall object was an ideal choice since I could place it in various combinations and explore how the UAV reacts. The specific file in which the landscape is defined can be found in this location (goo.gl/vfi9NH).

In the following figures the scenarios of some of the experiments, which have been conducted, are illustrated (fig.D.14), (fig.D.15), (fig.D.16) and (fig.D.17). The experiments were successful and the behavior of the UAV was stable and as it is expected to be. There are available online videos of the experiments

in this address (goo.gl/JyTzGp). The code of the project can be found online at the following git repository (goo.gl/BU1ydu).

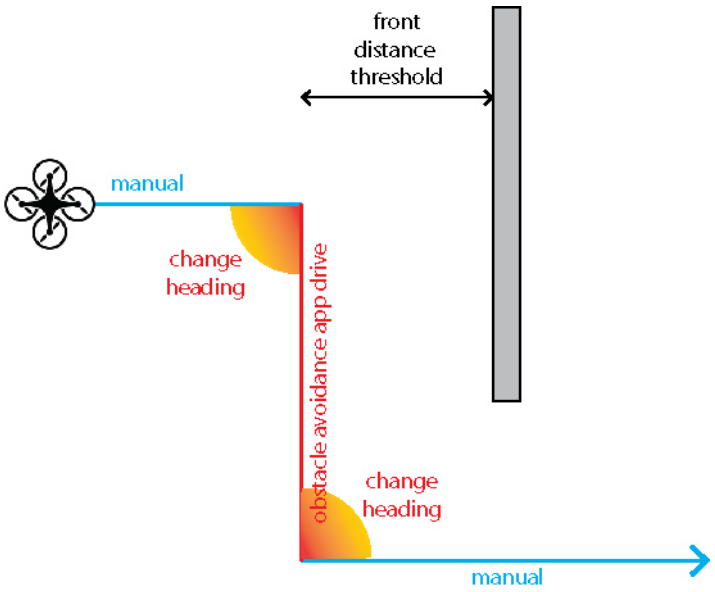

Fig.D.14 The avoidance maneuver when there is no obstacle in both sides of the wall

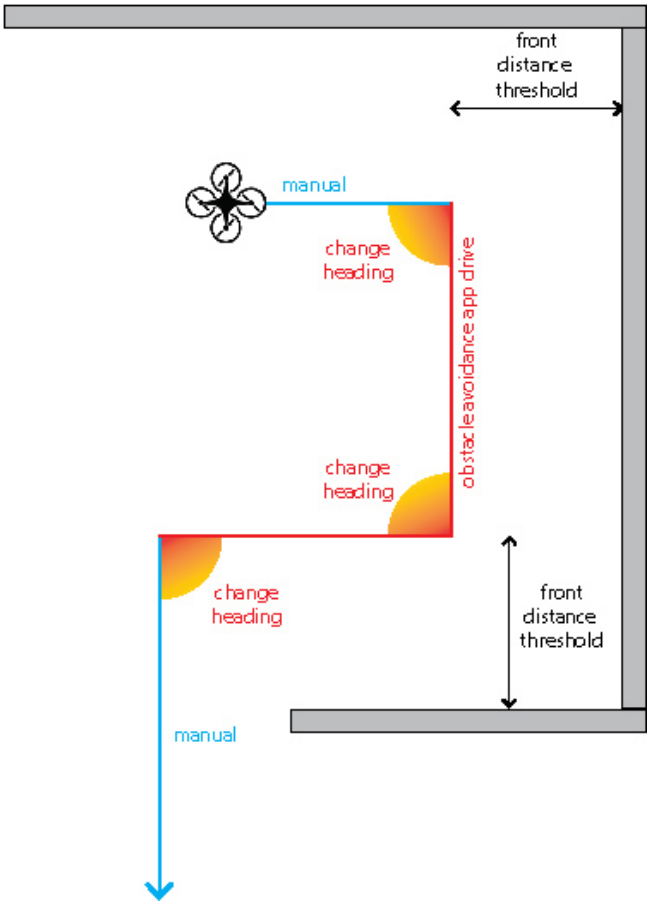

Fig.D.15 The avoidance maneuver when there are multiple obstacles in the path of the

UAV. Two avoidance maneuvers in sequence.

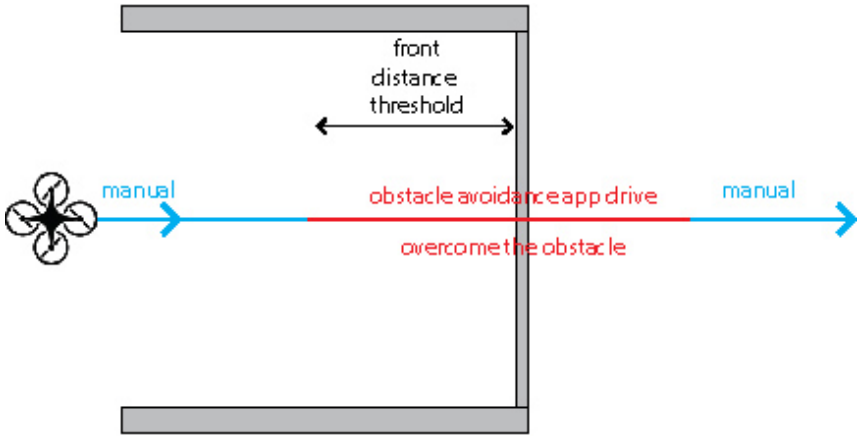

Fig.D.16 The UAV moves towards a dead end. The obstacle in front is lower than the maximum range of the altitude sensor. The UAV overcomes it and returns to alt hold.

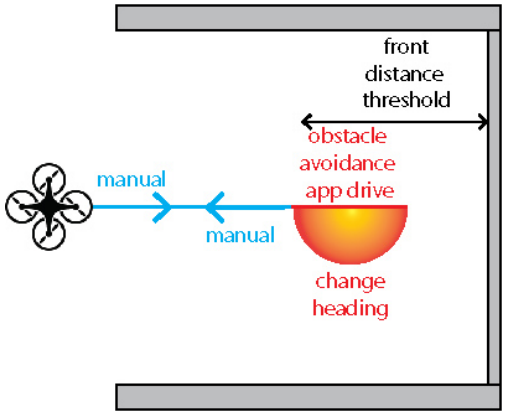

Fig.D.17 The UAV moves towards a dead end. The obstacle in front is higher than the maximum range of the altitude sensor. The UAV changes heading by 180^{o} and returns to alt hold.