Ministry of Higher Education and Scientific Research

Ecole Nationale Polytechnique

Electrical Power Engineering department

Final year's project thesis

To obtain the State Engineer Diploma in Electrical Power Engineering

Entitled

DESIGN OF AN UNMANNED AERIAL VEHICLE FOR POWER GRID MONITORING

Presented by:

MOKRANI Aymen and KHELALEF Yasser

Defended on the 9th of July 2020

Under the supervision of
Dr. Rabie BELKACEMI

ENP 2020

Ministry of Higher Education and Scientific Research

Ecole Nationale Polytechnique

Electrical Power Engineering department

Final year's project thesis

To obtain the State Engineer Diploma in Electrical Power Engineering

Entitled

DESIGN OF AN UNMANNED AERIAL VEHICLE FOR POWER GRID MONITORING

Presented by:

MOKRANI Aymen and KHELALEF Yasser

Defended on the 9th of July 2020

Under the supervision of

Dr. Rabie BELKACEMI

Before the jury composed of:

Pr. Taher ZEBADJI	President	Ecole Nationale Polytechnique
Dr. Rabie BELKACEMI	Supervisor	Ecole Nationale Polytechnique
Pr. Abdelhafid HELLAL	Examinor	Ecole Nationale Polytechnique

ENP 2020

ملخص:

ويمثل العمل الذي تم تنفيذه في هذه الأطروحة تصميم مركبة جوية بدون طيار لرصد وتتبع شبكة الطاقة الكهربائية بشكل ذاتي القيادة

يتكون التصميم من عدة أجزاء، وهذا سيسمح لنا إما بالتحكم في النموذج الأصلي يدويا أو ارسال مهمات آلية ذاتية القيادة، وهذه الأجزاء هي الأجهزة والمعدات المستخدمة لبناء النموذج وكذلك البرمجيات اللازمة لتشغيل هذه الأجهزة.

الكلمات المفتاحية: مركبة جوية بدون طيار، مركبة ذاتية القيادة، طيار آلي، نظام تشغيل الآليات، الذكاء الاصطناعي، الشبكة العصبية، تصنيف الصور.

Résumé :

Le travail réalisé dans cette thèse représente la conception d’un véhicule aérien sans pilote pour la surveillance autonome du réseau électrique.

La conception se compose de plusieurs parties, ce qui nous permettra soit de contrôler le prototype manuellement ou de planifier des missions autonomes, ces parties sont, essentiellement, le matériel et les équipements utilisés pour construire le prototype UAV, ainsi que les logiciels et programmes qui doivent être installer sur le matériel pour le contrôler.

Mots-clés : Véhicule aérien sans pilote, Véhicule autonome, Auto pilote, Système d’exploitation robotique, Intelligence artificielle, Réseau neuronal, Classification des images.

Abstract:

The work carried out in this thesis represents the design of an unmanned aerial vehicle for autonomous power grid monitoring.
The design consists of several parts, which will allow us to either control the prototype manually or plan autonomous missions, these parts are the hardware and the equipment used to build the UAV prototype, as well as the software needed to be implemented on the hardware to control it.
Keywords: Unmanned aerial vehicle, Autonomous vehicle, Auto pilot, Robotics operating system, Artificial intelligence, Neural network, images classification.

ACKNOWLEGEMENTS

We thank ALLAH the Almighty for giving us health and the willingness to start and finish this thesis.

Our sincere thanks to Dr. Rabie BELKACEMI for accepting to tutor, supervise and guide us in the development of this modest work.

We also thank the members of the jury, Pr. Taher ZEBBADJI, the president of the juries, and Pr. Abdelhafid HELLAL, the examiner, for agreeing to evaluate this work and for all their comments and criticisms, as well as all the teachers of the Electrical Power engineering department of the Ecole Nationale Polytechnique.

Our deep thanks also go to our families, for supporting, advising and encouraging us throughout all our educational cursus.

And finally, a deep gratitude to all the people who have helped and supported us from near and far.

Table of contents

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

GENERAL INTRODUCTION 10

Chapter 1 : Overall Concepts 12

1. Introduction: 13

2. Quadcopter (Drone): 13

3. Hardware: 13

3.1. Flight controller: 13

3.2. Raspberry Pi: 17

4. **Software:** 22

4.1. PX4 Flight stack: 22

4.2. MAVLink Communication Protocol: 23

4.3. Robotics Operating System ROS: 24

5. Conclusion: 33

Chapter 2 : QUADCOPTER MODELING 34

1. Introduction: 35

2. Theoretical operation of the UAV: 35

3. Change of reference between the inertial mark and the UAV body marker: 36

4. Angular velocities: 38

5. Relationship between Euler angles and quaternions: 39

6. Conclusion: 41

Chapter 3 : Software in The Loop Simulation SITL 42

1. Introduction: 43

2. Software in the loop simulation environment: 43

2.1. Gazebo simulator: 44

2.2 PX4 Firmware: 46

2.3 QGroundControl: 47

2.4. MAVROS: 48

3. Results and interpretation: 49

3.1. Codes to build the ROS package: 49

3.2. ROS with SITL Gazebo simulation: 53

3.3. Images classification: 64

4. Conclusion: 69

Chapter 4 : Launching the prototype 70

1 Introduction: 71

2 Identifying the components: 71

2.1 The Electronic speed controller (ESC): 71

2.2 Battery: 74

2.3 Motors: 76

2.2 Airframe: 80

3. Setting up the Raspberry Pi: 81

4. Configuring the PIXHAWK: 81

5. Power up the drone: 83

5.1 Connecting to the raspberry pi: 83

5.2 Launching the programs: 84

6. Conclusion: 85

BIBLIOGRAPHY 89

APPENDIX 90

LIST OF TABLES

Table 1.1 Table showing different Raspberry Pi models 20

Table 1.2 Table showing different pins functionalities 22

LIST OF FIGURES

Figure 1.1 PIXHAWK Hardware 17

Figure 1.2 Block Diagram for Raspberry Pi 18

Figure 1.3 Raspberry Pi Components 20

Figure 1.4 MAVLink’s role 23

Figure 1.5 A typical ROS Package structure 26

Figure 1.6 A simple ROS Program 27

Figure 1.7 roscore command output 28

Figure 1.8 rosnode list command output 29

Figure 1.9 catkin_init_workspace command output 30

Figure 1.10 Typical folders in a ROS workspace 31

Figure 1.11 The .bashrc file after editing 32

Figure 2.1 Main UAV landmarks 35

Figure 2.2 UAV auxiliary markers 37

Figure 2.3 Representation of a quaternion 40

Figure 3.1 Cycle SITL simulation enivrement 44

Figure 3.2 Gazebo simulator interface 45

Figure 3.3 Gazebo toolbox 46

Figure 3.4 QGROUNDCONTROL user interface 47

Figure 3.5 Adding dependencies to the CMaleLists.txt file 50

Figure 3.6 Adding the libraries to the CMakeLists.txt file 50

Figure 3.7 workspace folder 51

Figure 3.8 ROS package folder 51

Figure 3.9 Inside the ROS package 51

Figure 3.10 The nodes (programs) inside the src folder of the ROS package 51

Figure 3.11 Active nodes 52

Figure 3.12 Active nodes and topics 53

Figure 3.13 PX4 SITL gazebo simulation 54

Figure 3.14 MAVROS node output..55

Figure 3.15 The output of the drone_control.py node 56

Figure 3.16 The output of the input_node.py node...56

Figure 3.17 Takeoff mode source code..57

Figure 3.18 The position and orientation coordinates in takeoff mode 58

Figure 3.19 Takeoff flight mode..58

Figure 3.20 Landing flight mode 59

Figure 3.21 Landing mode source code..59

Figure 3.22 Forward and back source code 60

Figure 3.23 Slide right and left source code 60

Figure 3.24 Go up and down source code...61

Figure 3.25 Hovering mode source code 62

Figure 3.26 Position and orientation coordinates in hover mode 63

Figure 3.27 Linear and angular velocities in hover mode 63

Figure 3.28 Image classification output...66

Figure 3.29 The axes systems 67

Figure 3.30 functions to calculate the velocity according to the fixed axis xyz 68

Figure 3.31 source code for left and right directions...68

Figure 3.32 source code for neutre direction 69

Figure 4.1 Connection of the ESC 72

Figure 4.2 Block diagram of the different components inside the ESC 73

Figure 4.3 Inside a LiPo battery...74

Figure 4.4 brushless DC motor 76

Figure 4.5 Brushless motor sizing 77

Figure 4.6 The threaded bolt holes 78

Figure 4.7 Flashing the PIXHAWK with the chosen PX4 version..........................82

Figure 4.8 Airframe selection 82

Figure 4.9 Sensors calibration..83

ABBREVIATIONS

UAV	UNMANNED AERIAL VEHICLE
ESC	ELECTRONIC SPEED CONTROL
I2C	INTER-INTEGRATED CIRCUIT
ROS	ROBOT OPERATING SYSTEM
MAVLINK	MICRO AIR VEHICLE LINK
GPS	GLOBAL POSITIONING SYSTEM
FMU	FLIGHT MANAGEMENT UNIT
GPIO	GENERAL PURPOSE INPUT-OUTPUT
SITL	SOFTWARE IN THE LOOP
API	APPLICATION PROGRAMMING INTERFACE
UDP	USER DATAGRAM PROTOCOL
GCS	GROUND CONTROL STATION
MCU	MICRO CONTROLLER UNIT
LIPO	LITHIUM POLYMER
BLDC	BRUSHLESS DIRECT CURRENT

GENERAL INTRODUCTION

The inspection of high voltage power transmission lines is mainly carried out by manned aerial vehicles or foot patrol. However, these maintenance methodologies for inspection are somehow inefficient and expensive. Moreover, helicopter assisted inspection endangers the human life. Recently, unmanned aerial vehicles have been under development in several research centers all over the world due to its potential applications. In this work, we are going to talk about these methodologies and focus more on the unmanned aerial system based on the quadrotor helicopter for high voltage power line inspection. Our interest is to equip the quadrotor helicopter with the necessary payload in order to be able to carry out a qualitative inspection, therefore the hardware architecture of the aerial robotic system is presented.

In this project, we are interested in building a drone prototype where a camera is integrated in it that does image classification. The type of images processed by the onboard computer in the drone are high voltage power lines, and depending of the direction of these lines (left, right, direct) the computer will send commands to the flight controller of the drone to adjust its direction and keep tracking the power lines in order for it to be able to perform its main objective that is: a qualitative power line inspection.

In the first chapter, we will discuss all the concepts used in building the model, starting from the hardware equipment, to the software that need to be implemented on the hardware and the programs and platforms used to build to command and control the prototype.

The chapter that follows contains the mathematical and mechanical models that describe the UAV motion, which will help us eventually determine the necessary variables to control in order to command the UAV.

In chapter three, all the hardware and software elements discussed will be put into test in a simulation environment, where we will observe the drone’s behavior to various commands and codes before attempting to try it on the field.

Finally, on chapter four, we will discuss the methodology to follow in order to launch the UAV model, as well as the components needed and chosen to build the prototype.

Chapter 1 : Overall Concepts

1. Introduction:

UAVs have evolved throughout this century to be used today in several fields. The UAV is a flying device that does not require a human pilot. These autonomous aircraft were used for the first time in the military field during the First World War. The evolution of technology has now reached the level of electronics, allowing the performance of the UAV to be improved in a way that is significative. UAVs are now used in the medical audiovisual and engineering fields.

This chapter will discuss a category of drones called quadcopters. We will note, in a non-exhaustive way, the different technologies behind these devices, their applications, the different controllers used to pilot the UAV.

2. Quadcopter (Drone):

The quadcopter and rotary wing UAVs differ from fixed wing UAVs in their ability to move with a greater degree of freedom. The quadcopter has the advantage of being easy to control and inexpensive. Its ability to hover and fly at low speeds makes the quadcopter an interesting element in the use of UAVs in research. These UAVs can be several different sizes, from the size of a coin to several meters long. Smaller quadcopters are often categorized as micro UAVs and have special properties. The quadcopter belongs to the category of multi-rotor wing UAVs. It consists of four main components: the four engines, the chassis, the electronic board and the radio receiver.

3. Hardware:

3.1. Flight controller:

3.1.1. Introduction:

An autopilot is an embedded card has as a roll to perform on-board operations during the unmanned tasks of a vehicle, such as the flight of an aircraft, the journey of an autonomous car, the immersion of a submarine robot, or any other type of mobile robot.

Unlike a development card, the autopilot usually has a greater capacity for processing and data transfer. This is because:

- Orientation and position sensors are read.
- Signals are read from the remote control.

- Other sensors coupled to the system are read, either through analog ports or digital or serial transmission protocols.
- Flight data is stored for later statistical or graphical use.
- The unmanned vehicle is intercommunicated with other vehicles or a base on the ground, using wireless networks.
- The battery is measured.
- Visual and sound alerts are sent.
- The control is processed.
- The data obtained is filtered.
- The control is written to the motors.
- Selected processes are executed in real-time modules.
- Demanding mathematical operations are performed in very short times, such as multiplication of large dimension matrices, calculation of trajectories, and estimation of speeds and accelerations.

With the demand for resources, a development card tends to collapse or simply can't achieve such performance. For example, the Arduino development board, in its mega model, cannot operate more than a brushless motor at 490hz since in principle its clock barely manages 300hz for a single motor, compromising the operation of the rest of the ports and systems.

Now, if we compare it against another type of development cards or even more sophisticated and specialized processors such as a Raspberry Pi, the autopilot only contains the minimum equipment necessary and is only optimized for the teleoperation of a vehicle; that is, writing to an adequate number of motors (from 4 to 12, for example), writing to auxiliary motors (servos, for example), reading of positioning and orientation data, data feedback and control by the remote user, storage of flight data, and additional reading of on-board equipment (distance sensors, GPS redundant modules, etc.). Therefore, space, weight, and power consumption are optimized for the task of driving a vehicle.

Among the best-known autopilots are the Pixhawk, the Naza, the ArduPilot, the Crazyflie, and the CC3D.

3.1.2. PIXHAWK:

The Pixhawk dates back to 2008. It was initially developed as a student project in the ETH of Switzerland by Lorenz Meier, and it was marketed in mid-2012 by the company 3DR. Throughout this text, you will see that the ETH is an important part of the history of drone design. [1]

The Pixhawk has in its FMUv2 version (which will be used for this text) the following features (note that they change a little bit between manufacturers and clones):

Processor:

- 32-bit STM32F427
- 168Mhz RAM 256Kb
- 2MB flash memory

Integrated Sensors:

- 3-axis gyro with 16-bit resolution ST Micro L3GD20
- Accelerometer with 3-axis magnetometer and 14 bits of resolution ST Micro LSM303D
- Accelerometer with redundant 3-axis gyroscope Invensense MPU 6000
- Barometer MS5611
- Some versions have GPS

Weigh and dimensions:

- 33-40 grams depending on the model and manufacturer 80x45x15mm approximately
- Power consumption: 3.3V and 6.6V ADC inputs

Communication ports:

- I2C
- Analog inputs 3.3V and 6V
- SPI
- MicroUSB
- Futaba and Spektrum radio ports
- Power port
- CAN
- 5 UART
- PPM port
- microSD

Throughout this text, we will use Pixhawk version 1 or its clones 2.4.8 or 2.4.6 (note that they are only names, since the real version 2 of the autopilot dates from 2017). However, the use of the libraries is extensible to other autopilots of the Pixhawk family and even other families of autopilots and drones.

As mentioned, although version 1 or its clones contain more ports, the most used are as follows (and as shown in Figure 1.1):

- Serial communication port (wired): With this port, it’s possible to connect an Arduino or any other development card with the intention of external processing of data and receiving only simplified information. For example, one use is image processing with the Raspberry Pi to identify positions of objects and send these positions to the Pixhawk by standard serial protocol. See FRONT 3 in Figure 1.1.
- Serial communication ports (wireless): With this port, it’s possible to connect an intercom in order to transfer data wirelessly between autopilots (altitudes, angles, sequences of operation). This shouldn’t be confused with the radio control port; this interface operates at 915Hz, except in Europe and countries with the European standards. See FRONT 2 in Figure 1.1.
- Analog interface ports: With these ports, it’s possible to connect analog sensors such as potentiometers, ultrasonic position sensors, temperature sensors, or pressure sensors. The Pixhawk has three analog ports; one at 6.6V and two shared at 3.3V. See FRONT 13 and FRONT 14 in Figure 1.1.
- Ports of digital interface: It is possible to use these ports as GPIO ports (generic digital input and output ports). With them it’s possible to use push buttons, LEDs, or any other device that works with binary logic (on and off). They are shared with the Auxiliary PWM ports. See AUXILIARY SLOTS PWM in Figure 1.1.
- Fast PWM ports for brushless motors: These ports are used to connect the main motors of the system and operate at 419hz. See MAIN SLOTS PWM in Figure 1.1.
- PWM ports of slow writing or auxiliary ports: These ports are for servos and motors aimed at secondary operation of the system (fins, robotic arms of support, stabilizers of cameras, etc.). They operate at 50hz. See AUXILIARY SLOTS PWM in Figure 1.1.
- Radio interface ports: The most commonly used is the PPM port, not to be confused with the serial ports of wireless communication. They work in a way that allows the user to have manual control of the vehicle. This can serve as an emergency stop or to activate a sequence of operations in a semi-automatic way (takeoff, trajectory following, rotation

and anchoring, descent). This interface works at 3Mhz, except in countries with European standards. See RC Input Port in Figure 1.1.

- LED signaling: This is a Toshiba device incorporated in order to indicate visual alerts. See FRONT 15 in Figure 1.1.
- SD memory port: This stores the flight data to use later for statistics or graphics. See SIDES 2 in Figure 1.1.
- Emergency or auxiliary buzzer: Used to activate a variety of sound alerts. See FRONT 8 in Figure 1.1.
- Security switch: If it is not activated, the motors simply will not turn. The security switch is a button to avoid cutting or injuring anyone with propellers or motors due to unwanted behavior. See FRONT 7 in Figure 1.1.

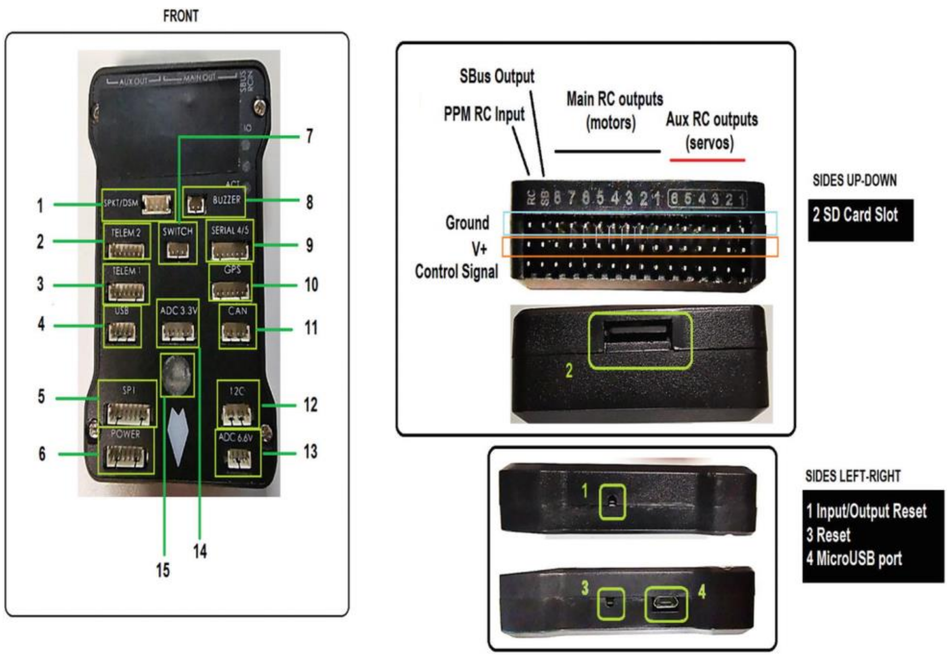


Figure 1.1 PIXHAWK Hardware

3.2. Raspberry Pi:

3.2.1. Introduction:

The Raspberry Pi is a series of small single-board computers developed in the United Kingdom by the Raspberry Pi Foundation to promote teaching of basic computer science in schools and in developing countries. The original model became far more popular than anticipated, selling outside its target market for uses such as robotics. It now is widely used even in research projects, such as for weather monitoring because of its low cost and portability. It does not include peripherals (such as keyboards and mice) or cases. However, some accessories have been included in several official and unofficial bundles.

The Raspberry Pi hardware has evolved through several versions that feature variations in the type of the central processing unit, amount of memory capacity, networking support, and peripheral-device support.

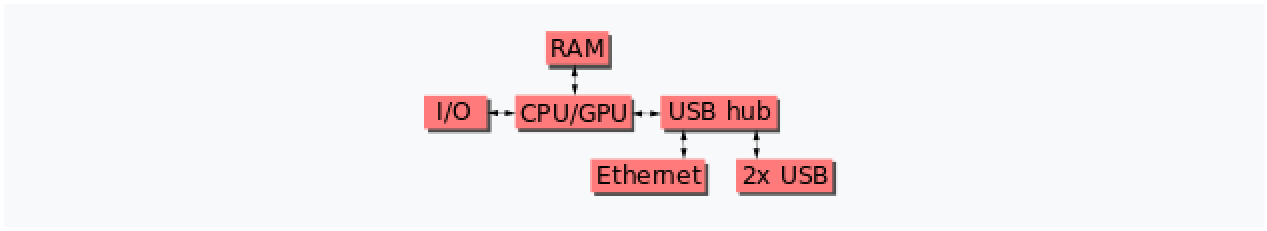


Figure 1.2 Block Diagram for Raspberry Pi

This block diagram describes Model B and B+; Model A, A+, and the Pi Zero are similar, but lack the Ethernet and USB hub components. The Ethernet adapter is internally connected to an additional USB port. In Model A, A+, and the Pi Zero, the USB port is connected directly to the system on a chip (SoC). On the Pi 1 Model B+ and later models the USB/Ethernet chip contains a five-port USB hub, of which four ports are available, while the Pi 1 Model B only provides two. On the Pi Zero, the USB port is also connected directly to the SoC, but it uses a micro USB (OTG) port. Unlike all other Pi models, the 40 pin GPIO connectors are omitted on the Pi Zero with solderable through holes only in the pin locations.[2]. The Pi Zero WH remedies this, as shown in this table:

Family	Model	Form Factor	Ethernet	Wireless	GPIO	Released	Discontinued
Raspberry Pi Zero	W/WH	Zero	No	Yes	40-pin	2017	
Raspberry Pi Zero	Zero	Zero	No	No	40-pin	2015	
Raspberry Pi 4	B (1 GiB)	Standard	Yes	Yes	40-pin	2019	Yes
Raspberry Pi 4	B (2 GiB)	Standard	Yes	Yes	40-pin	2019[31]	
Raspberry Pi 4	B (4 GiB)	Standard	Yes	Yes	40-pin	2019[31]	
Raspberry Pi 4	B (8 GiB)	Standard	Yes	Yes	40-pin	2020	
Raspberry Pi 3	B	Standard	Yes	Yes	40-pin	2016	
Raspberry Pi 3	A+	Compact	No	Yes	40-pin	2018	
Raspberry Pi 3	B+	Standard	Yes	Yes	40-pin	2018	
Raspberry Pi 2	B	Standard	Yes	No	40-pin	2015	
Raspberry Pi	B	Standard	Yes	No	26-pin	2012	Yes

Raspberry Pi	A	Standard	No	No	26-pin	2013	Yes
Raspberry Pi	B+	Standard	Yes	No	40-pin	2014	
Raspberry Pi	A+	Compact	No	No	40-pin	2014	

Table 1.1 Table showing different Raspberry Pi models

3.2.2. Components:

The components of a Raspberry Pi are similar to those you will find in any modern device (phone, tablet, laptop, desktop...) as shown in figure 1.3:

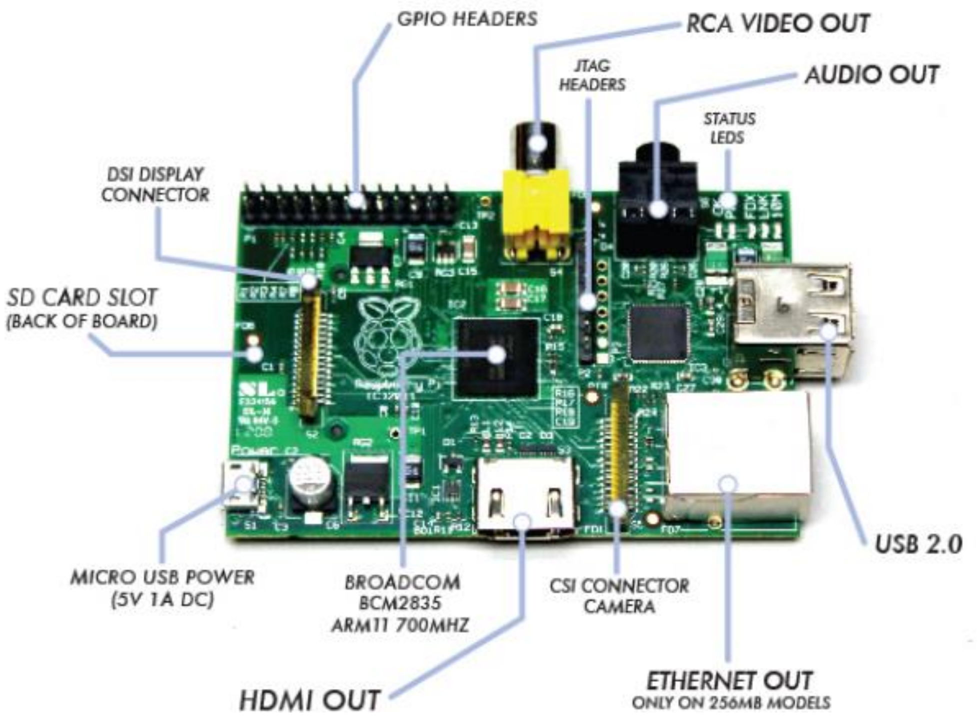

GPIO HEADERS
RCA VIDEO OUT
JTAG HEADERS
AUDIO OUT
STATUS LEDS
DSI DISPLAY CONNECTOR
SD CARD SLOT (BACK OF BOARD)
USB 2.0
MICRO USB POWER (5V 1A DC)
BROADCOM BCM2835 ARM11 700MHZ
CSI CONNECTOR CAMERA
HDMI OUT
ETHERNET OUT ONLY ON 256MB MODELS

Figure 1.3 Raspberry Pi Components

3.2.3. General purpose input-output (GPIO) connector:

Raspberry Pi 1 Models A+ and B+, Pi 2 Model B, Pi 3 Models A+, B and B+, Pi 4, and Pi Zero, Zero W, and Zero WH GPIO J8 have a 40-pin pinout. Raspberry Pi 1 Models A and B have only the first 26 pins.

In the Pi Zero and Zero W the 40 GPIO pins are unpopulated, having the through-holes exposed for soldering instead. The Zero WH (Wireless + Header) has the header pins preinstalled, as shown in table 1.2:

GPIO#	2nd func.	Pin#		Pin#	2nd func.	GPIO#
	+3.3 V	1		2	+5 V	
2	SDA1 (I²C)	3		4	+5 V	
3	SCL1 (I²C)	5		6	GND	
4	GCLK	7		8	TXD0 (UART)	14
	GND	9		10	RXD0 (UART)	15
17	GEN0	11		12	GEN1	18
27	GEN2	13		14	GND	
22	GEN3	15		16	GEN4	23
	+3.3 V	17		18	GEN5	24
10	MOSI (SPI)	19		20	GND	

9	MISO (SPI)	21	22	GEN6	25
11	SCLK (SPI)	23	24	CE0_N (SPI)	8
	GND	25	26	CE1_N (SPI)	7
(Pi 1 Models A and B stop here)					
0	ID_SD (I²C)	27	28	ID_SC (I²C)	1
5	N/A	29	30	GND	
6	N/A	31	32		12
13	N/A	33	34	GND	
19	N/A	35	36	N/A	16
26	N/A	37	38	Digital IN	20
	GND	39	40	Digital OUT	21

Table 1.2 Table showing different pins functionalities

4. **Software:**

4.1. PX4 Flight stack:

The Pixhawk autopilot is hardware-compatible with the ArduPilot, PX4, Dronekit, MAVROS libraries, and even with Parrot Bebop drone. On the other hand, the ArduPilot libraries are compatible with the Pixhawk autopilot, the APM, Snapdragon, ErleBrain NAVio, and Parrot

Bebop drone. The complete lists of compatibilities as well as versions currently not supported are on their respective web pages.

Similar SDK projects are:

- PX4
- Paparazzi
- Crazyflie
- Dronekit

PX4 is the Professional Autopilot. Developed by world-class developers from industry and academia, and supported by an active worldwide community, it powers all kinds of vehicles from racing and cargo drones through to ground vehicles and submersibles. It is the flight controller software that will be used during this project.[3]

4.2. MAVLink Communication Protocol:

MAVLink is a very lightweight messaging protocol for communicating with drones (and between onboard drone components). MAVLink follows a modern hybrid publish-subscribe and point-to-point design pattern: Data streams are sent / published as topics while configuration sub-protocols such as the mission protocol or parameter protocol are point-to-point with retransmission.

Messages are defined within XML files. Each XML file defines the message set supported by a particular MAVLink system, also referred to as a "dialect". The reference message set that is implemented by most ground control stations and autopilots is defined in common.xml (most dialects build on top of this definition).

The MAVLink toolchain uses the XML message definitions to generate MAVLink libraries for each of the supported programming languages. Drones, ground control stations, and other MAVLink systems use the generated libraries to communicate. These are typically MIT-licensed, and can therefore be used without limits in any closed-source application without publishing the source code of the closed-source application.[3]

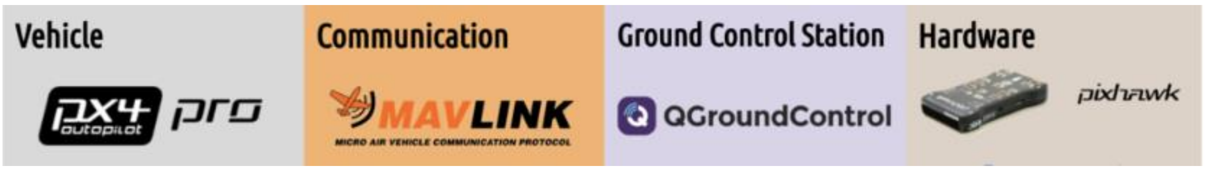


Figure 1.4 MAVLink's role

4.3. Robotics Operating System ROS:

4.3.1. Introduction:

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.[4]

4.3.2. ROS Goal:

The quality that makes ROS better, unique, and powerful is not because it is a framework with the most features. Instead, ROS was mainly and primarily built to support code reuse in robotics research and development as it is a distributed framework of processes (also known as Nodes) that enables executables to be individually designed and easily coupled at runtime.

In support of this primary goal of sharing and collaboration, there are several other goals of the ROS framework [4]:

- Thin: ROS is designed to be as thin as possible, so that code written for ROS can be used with other robot software frameworks. A corollary to this is that ROS is easy to integrate with other robot software frameworks: ROS has already been integrated with OpenRAVE, Orocos, and Player.
- Language independence: the ROS framework is easy to implement in any modern programming language (C++, Python, JAVA…).
- Easy testing
- Scaling: ROS is appropriate for large runtime systems and for large development processes.

ROS currently only runs on Unix-based platforms. Software for ROS is primarily tested on Ubuntu and Mac OS X systems. Even though a port to Microsoft Windows for ROS is possible, it has not yet been fully explored.

4.2.3. ROS Concepts:

ROS has three levels of concepts: The Filesystem level, the Computation Graph level, and the Community level. These levels and concepts. First, ROS Filesystem Level which covers ROS resources that you encounter on disk. Second is Computational Graph level, this latter is the network of ROS processes that are processing data together. Third and finally, The ROS

Community Level concepts which are ROS resources that enable separate communities to exchange software and knowledge.

4.2.3.1. ROS Filesystem Level:

- Packages: Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together. Packages are the most atomic build item and release item in ROS. Meaning that the most granular thing you can build and release is a package.
- Metapackages: Metapackages are specialized Packages which only serve to represent a group of related other packages. Most commonly metapackages are used as a backwards compatible place holder for converted rosbuild Stacks.
- Package Manifests: Manifests (package.xml) provide metadata about a package, including its name, version, description, license information, dependencies, and other meta information like exported packages.
- Repositories: A collection of packages which share a common VCS system. Packages which share a VCS share the same version and can be released together using the catkin release automation tool bloom. Often these repositories will map to converted rosbuild Stacks. Repositories can also contain only one package.
- Message (msg) types: Message descriptions, stored in my_package/msg/MyMessageType.msg, define the data structures for messages sent in ROS.
- Service (srv) types: Service descriptions, stored in my_package/srv/MyServiceType.srv, define the request and response data structures for services in ROS.

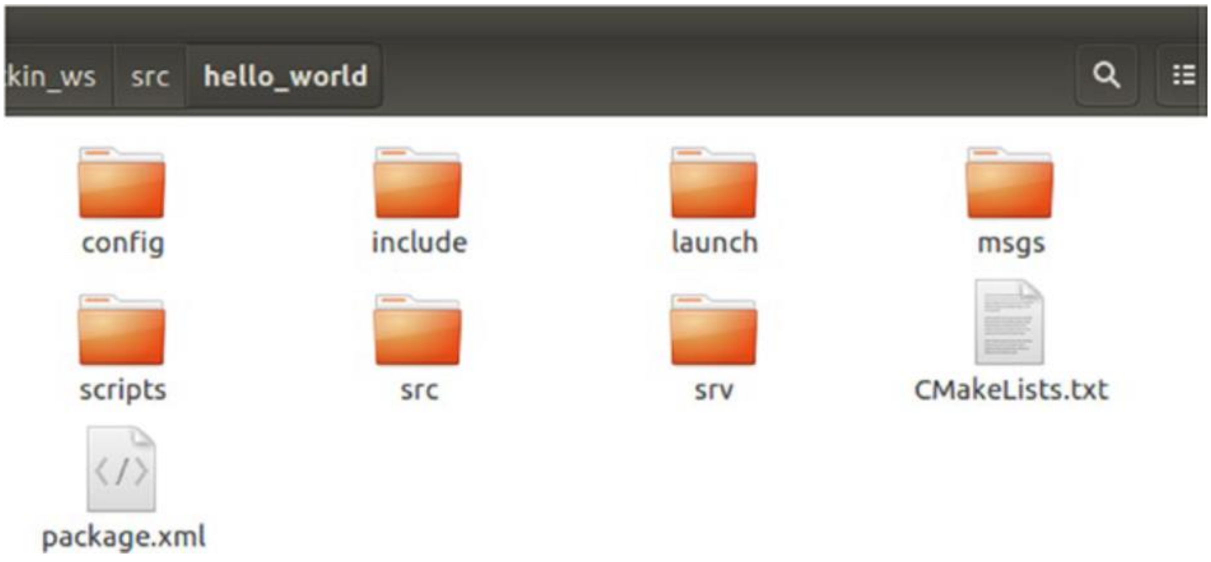


Figure 1.5 A typical ROS Package structure

4.2.3.2. ROS Computation Graph Level:

The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways.

- Nodes: Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one Node provides a graphical view of the system, and so on. A ROS node is written with the use of a ROS client library, such as roscpp or rospy.
- Master: The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.
- Parameter Server: The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.
- Messages: Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields. Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types. Messages can include arbitrarily nested structures and arrays (much like C structs).
- Topics: Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each other’s' existence. The idea is to decouple the production of information from its consumption. Logically, one can think of a topic as a strongly typed message bus. Each bus has a name, and anyone can connect to the bus to send or receive messages as long as they are the right type.
- Services: The publish / subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions, which are often required in a distributed system. Request / reply is done via services, which are defined by a pair of message structures: one for the request

and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply. ROS client libraries generally present this interaction to the programmer as if it were a remote procedure call.

- Bags: Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

The ROS Master acts as a nameservice in the ROS Computation Graph. It stores topics and services registration information for ROS nodes. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run. [4]

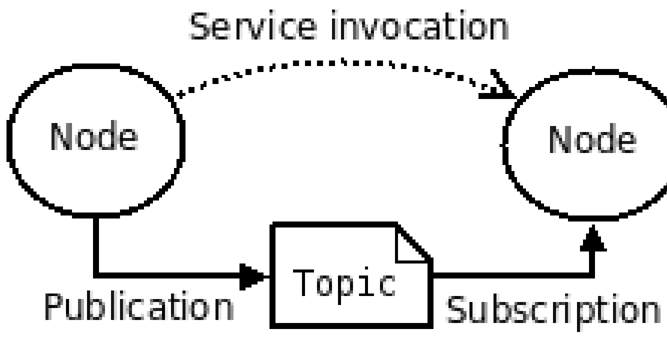


Figure 1.6 A simple ROS Program

4.2.3.3. ROS Community Level:

These resources include:

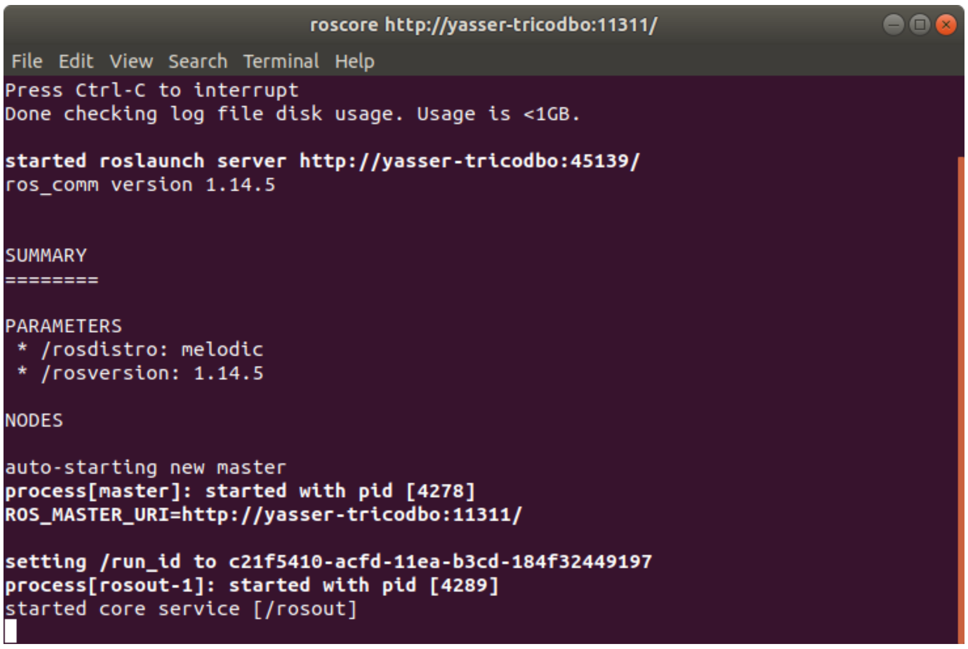
- Distributions: ROS Distributions are collections of versioned stacks that you can install. Distributions play a similar role to Linux distributions: they make it easier to install a collection of software, and they also maintain consistent versions across a set of software.
- Repositories: ROS relies on a federated network of code repositories, where different institutions can develop and release their own robot software components.

- The ROS Wiki: The ROS community Wiki is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.

4.2.4. ROS Command Tools:

The use of ROS requires a set of command-line tools. They are used to explore various aspects of ROS. We can implement almost all the capabilities of ROS using these tools. The command-line tools are executed in the Linux terminal; like the other commands in Linux. [5]

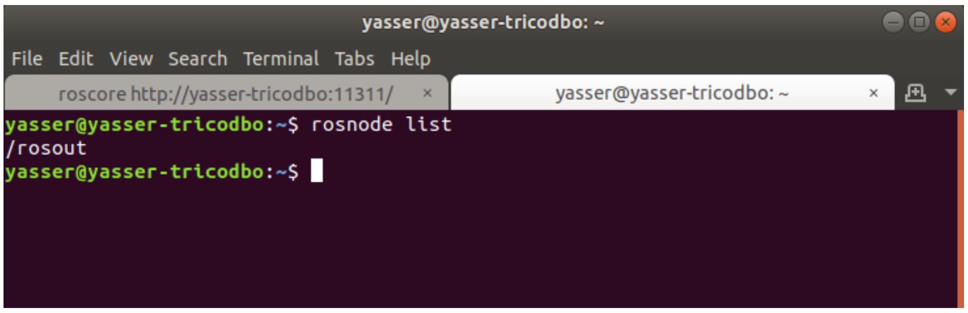
- The roscore command: $ roscore
  is a very important tool in ROS. When we run this command in the terminal, it starts the ROS master, the parameter server, and a logging node. We can run any other ROS program/node after running this command. So, run roscore on one terminal window, and use another terminal window to enter the next command to run a ROS node.


roscore http://yasser-tricodbo:11311/
File Edit View Search Terminal Help

```
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://yasser-tricodbo:45139/
ros_comm version 1.14.5


SUMMARY
========

PARAMETERS
 * /rosdistro: melodic
 * /rosversion: 1.14.5

NODES

auto-starting new master
process[master]: started with pid [4278]
ROS_MASTER_URI=http://yasser-tricodbo:11311/

setting /run_id to c21f5410-acfd-11ea-b3cd-184f32449197
process[rosout-1]: started with pid [4289]
started core service [/rosout]
```

Figure 1.7 roscore command output

- The rosnode command: explores all the aspects of a ROS node.
  For example, we can list the number of ROS nodes running on our system. If you type any of the commands, you get complete help for the tool.
  $ rosnode list


yasser@yasser-tricodbo: ~
File Edit View Search Terminal Tabs Help
roscore http://yasser-tricodbo:11311/
yasser@yasser-tricodbo: ~

```
yasser@yasser-tricodbo:~$ rosnode list
/rosout
yasser@yasser-tricodbo:~$ 
```

Figure 1.8 rosnode list command output

- The rostopic command: provides information about the topics publishing/subscribing in the system. This command is very useful for listing topics, printing topic data, and publishing data.
  `$ rostopic list`
  If there is a topic called /chatter, we can print/echo the topic data using the following command.
  `$ rostopic echo /<topic_name>`
  If we want to publish a topic with data, we can easily do this command.
  `$ rostopic pub topic_name msg_type data`
- The roslaunch command is also useful in ROS. If you want to run more than ten ROS nodes at time, it is very difficult to launch them one by one.
  In this situation, we can use roslaunch files to avoid this difficulty. ROS launch files are XML files in which you can insert each node that you want to run. Another advantage of the roslaunch command is that the roscore command executes with it, so we don’t need to run an additional roscore command for running the nodes.
  The following is the syntax for running a roslaunch file. The 'roslaunch' is the command to run a launch file, along with that we have to mention package name and name of launch file.
  `$ roslaunch ros_pkg_name launch_file_name`
- To run a ROS node, you have to use the rosrun node. Its usage is very simple.
  `$ rosrun ros_pkg_name node_name`

4.2.5. Programming with ROS:

4.2.5.1. Creating a ROS workspace:

As any framework, a workspace is needed to be able to start working with the framework. For ROS, a workspace is where ROS packages are kept. We can create new packages, install existing packages, build and create new executables. To create a ROS workspace folder, its name and location does not matter. To do so, we enter the following command in a new terminal.

```
$ mkdir -p ~/catkin_ws/src
```

This creates a folder called catkin_ws, inside of which is another folder called src. The ROS workspace is also called the catkin workspace.

What is important and mandatory though is that the src folder shouldn't be changed, yet we can change the workspace folder name.

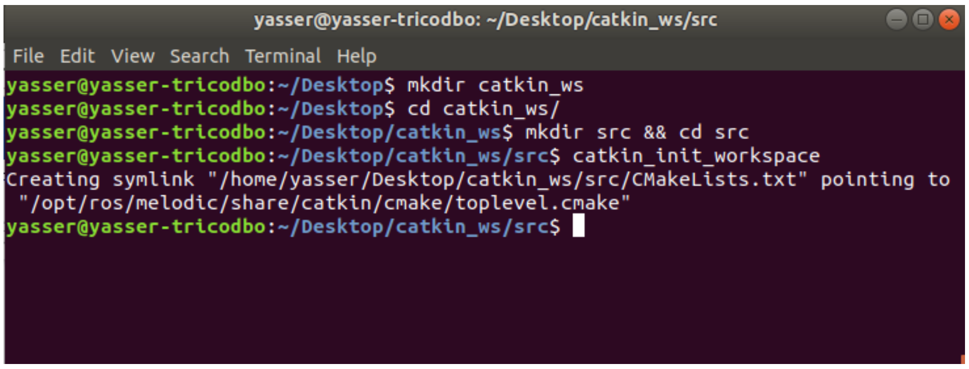
After entering the command, we can switch to the src folder by using the cd command.

```
$ cd catkin_ws/src
```

The following command initializes a new ROS workspace. If we are not initializing a workspace, we cannot create and build the packages properly.

```
$ catkin_init_workspace
```

After this command, we see the message in Figure 1.9:


yasser@yasser-tricodbo: ~/Desktop/catkin_ws/src
File Edit View Search Terminal Help

```
yasser@yasser-tricodbo:~/Desktop$ mkdir catkin_ws
yasser@yasser-tricodbo:~/Desktop$ cd catkin_ws/
yasser@yasser-tricodbo:~/Desktop/catkin_ws$ mkdir src && cd src
yasser@yasser-tricodbo:~/Desktop/catkin_ws/src$ catkin_init_workspace
Creating symlink "/home/yasser/Desktop/catkin_ws/src/CMakeLists.txt" pointing to
 "/opt/ros/melodic/share/catkin/cmake/toplevel.cmake"
yasser@yasser-tricodbo:~/Desktop/catkin_ws/src$ 
```

Figure 1.9 catkin_init_workspace command output

There is a CMakeLists.txt inside the src folder. After initializing the catkin workspace, we can build the workspace. We can able it to build the workspace without any packages. To build the workspace, we can switch from the catkin_ws/src folder to the catkin_ws folder.

```
$ ~/catkin_ws/src$ cd ..
```

The command to build the catkin workspace is catkin_make.

```
$ ~/catkin_ws$ catkin_make
```

We get the following output after entering this command, now we can see a few folders in addition to the src folder:

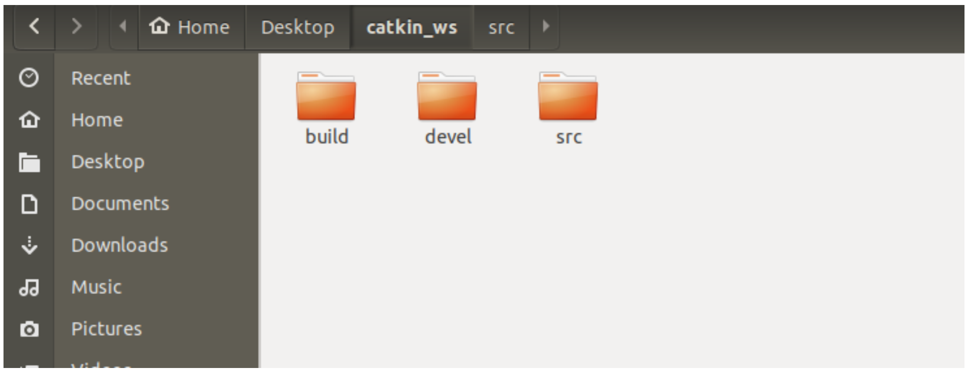


Figure 1.10 Typical folders in a ROS workspace

- src Folder:
  The src folder inside the catkin workspace folder is the place where it is possible to create, or clone, new packages from repositories. ROS packages only build and create an executable when it is in the src folder. When we execute the catkin_make command from the workspace folder, it checks inside the src folder and build each package.
- Build Folder:
  When we run the catkin_make command from the ROS workspace, the catkin tool creates some build files and intermediate cache CMake files inside the build folder. These cache files help prevent from rebuilding all the packages when running the catkin_make command; for example, if you build five packages, and then add a new package to the src folder, only the new package builds during the next catkin_make command. This is because of those cache files inside the build folder. If you delete the build folder, all the packages build again.
- Devel Folder:
  When we run the catkin_make command, each package is built, and if the build process is successful, the target executable is created. The executable is stored inside the devel folder, which has shell script files to add the current workspace to the ROS workspace path. We can access the current workspace packages only if we run this script. Generally, the following command is used to do this.

  ```
  source ~/<workspace_name>/devel/setup.bash
  ```

It is important now to add the workspace environment. This means we have to set the workspace path so that the packages inside the workspace become accessible and visible.

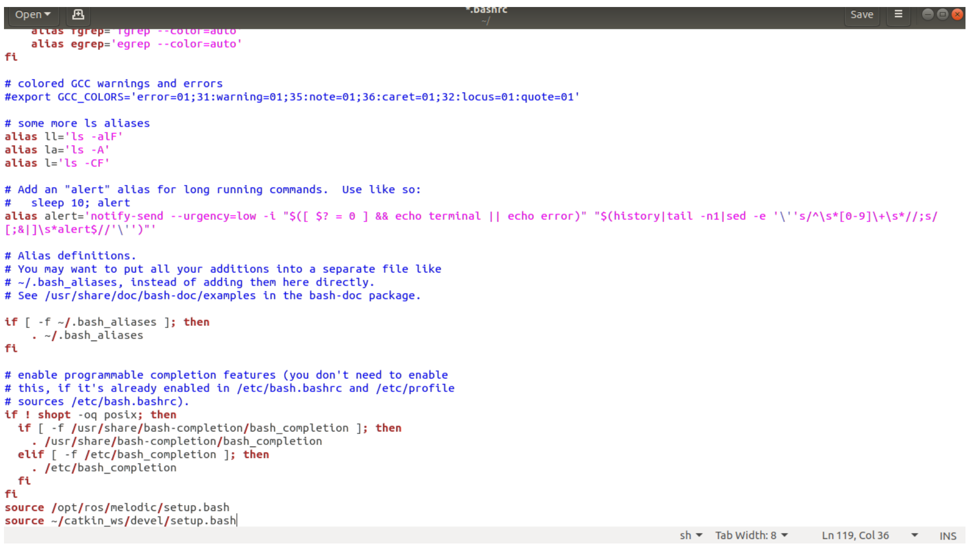
To do this, you have to do the following steps.

- We open the .bashrc file in the home folder and add the following line at the end of the file.
- At a terminal, we switch to the home folder and select the .bashrc file.

```
$ gedit .bashrc
```

- Add the following line at the end of .bashrc.

```
source ~/catkin_ws/devel/setup.bash
```



```
    alias egrep='egrep --color=auto'
fi

# colored GCC warnings and errors
#export GCC_COLORS='error=01;31:warning=01;35:note=01;36:caret=01;32:locus=01:quote=01'

# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'

# Add an "alert" alias for long running commands.  Use like so:
#   sleep 10; alert
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)" "$(history|tail -n1|sed -e '\''s/^\s*[0-9]\+\s*//;s/
[;&|]\s*alert$//'\'')"'

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi
source /opt/ros/melodic/setup.bash
source ~/catkin_ws/devel/setup.bash
```

Figure 1.11 The .bashrc file after editing

As we already know, the .bashrc script in the home folder executes when a new terminal session starts. So, the command inserted in the .bashrc file also executes. setup.bash in the following command has variables to add to the Linux environment.

```
source ~/catkin_ws/devel/setup.bash
```

When we source this file, the workspace path is added in the current terminal session. Now when we use any terminal, we can access the packages inside this workspace.

4.2.5.2. ROS Build system:

In ROS, there is a build system for compiling ROS packages. The name of the build system that we are using is catkin. catkin is a custom build system made from the CMake build system and Python scripting. CMake is not directly used because building a set of ROS packages is complicated. The complexity increases with the number of packages and package dependencies. The catkin build system takes cares of all these things.

4.2.5.3 Creating a ROS package:

After creating the necessary workspace, it is possible for us now to create a catkin ROS package by using the following command.

```
$ catkin_create_pkg ros_package_name package_dependencies
```

The command that we use to create the package is catkin_create_ pkg. The first parameter for this command is the package name, and the dependencies of the package follow it. You have to execute the command from the src folder in the catkin workspace.[5]

Inside the package is the src folder, package.xml, CMakeLists.txt, and the include folder.

- CMakeLists.txt: This file has all the commands to build the ROS source code inside the package and create the executable.
- package.xml: This is basically an XML file. It mainly contains the package dependencies, information, and so forth.
- src: The source code of ROS packages is kept in this folder. Normally, C++ files are kept in the src folder. If you want to keep Python scripts, you can create another folder called scripts inside the package folder.
- include: This folder contains the package header files. It can be automatically generated, or third-party library files go in it.

5. Conclusion:

This chapter has listed the different technologies used to control a quadcopter, where we use the flight controller PIXHAWK with the PX4 flight stack installed in it, and an onboard computer called the Raspberry Pi where ROS will be implemented in it.

Chapter 2 : QUADCOPTER MODELING

1. Introduction:

In order to control a drone, we need first to understand how it works, in some other words, the mathematical and mechanical models behind its movements. Hence in this chapter, we will elaborate the physical movements of the drone and as well as the different models and equations that describe that.

2. Theoretical operation of the UAV:

The quadcopter is made up of 4 engines allowing to orient the UAV (Figure 2.1). The angular velocity ω_1 and ω_1 of the *M1* and *M3* engines go counter-clockwise and the velocities ω_2 and ω_4 of the *M2* and *M4* engines go clockwise. The quadcopter can be oriented in different ways: the so-called extra configuration, where the UAV's nose is in front of the *M1* engine, and the *cross* configuration, where the nose is between the *M1* and *M2 engines*. A positive rotation of the yaw angle is achieved by increasing the velocities of the *M2* and *M4 engines* relative to the velocities of the *M1* and *M3 engines*. An increase in the velocities of the *M1* and *M3 motors relative to the* velocities of the *M2* and *M4 motors* will produce negative yaw rotation.

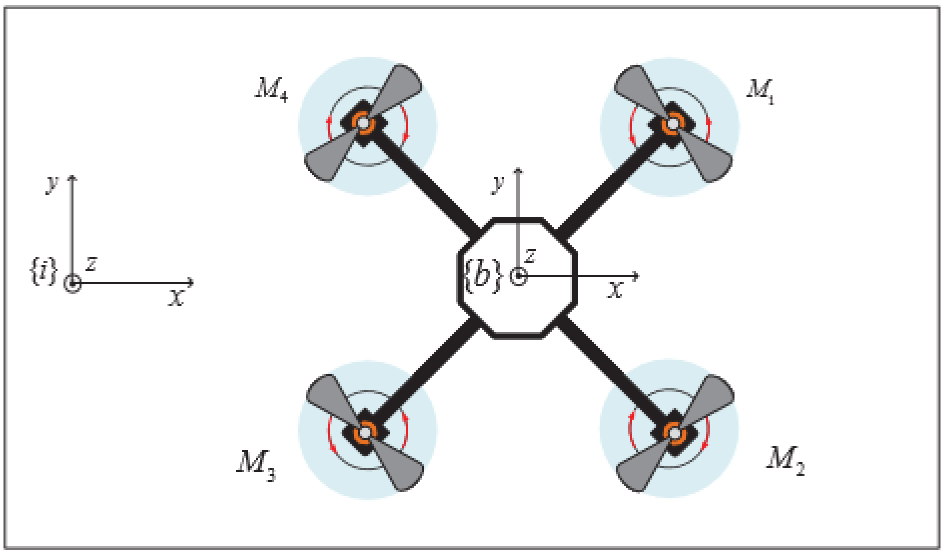


Figure 2.1 Main UAV landmarks

When the UAV is in a cross configuration, the speed difference between the four engines is used to perform a roll or pitch rotation. Indeed, an increase of the velocities of motors *M1* and *M4* in relation to the velocities of motors *M2* and *M3* allows for positive pitch rotation and vice versa for negative rotation. An increase in the velocities of the *M1* and *M2* motors in relation to the speeds of the *M3* and *M4 motors* achieves a positive rotation in roll and vice versa for a negative rotation.[6]

When the UAV is in an extra configuration, the difference in velocities between the two engines is used to make the roll and pitch rotations. An increase in velocity of engine *M1* compared to *M3* will make a positive rotation in pitch. An increase in the velocity of motor *M4 with* respect to *M2 will make a* positive rotation in roll.

Two main markers are used to describe the movement of the UAV. Mark {i} represents the inertial marker, it's fixed in relation to the Earth. Marker {b} represents the marker on the drone's body. The *z-axis of* frame *{b}* is always normal to the UAV body. The {b} marker has been taken to be in a cross configuration.

The orientation of the UAV can be represented in different ways. One can, indeed, use quaternions or Euler angles. Contrary to Euler angles, quaternions do not need to have auxiliary markers to be properly described. They can also avoid the phenomenon known as *gimbal lock*, which takes away two degrees of freedom from the UAV. This phenomenon as well as the quaternions will be explained in more detail in the rest of this chapter.

3. Change of reference between the inertial mark and the UAV body marker:

Three auxiliary markers {ri}, {rψ} and {rθ} must be used to describe Euler angles, the yaw-pitch-roll convention will be used (Figure 2.2):

- The origin of the {ri} marker {ri} is in the center of the drone's body. Its orientation is the same as the orientation of the inertial marker {i};
- The {rψ} marker follows an angular rotation ψ on the *z* axis of the *{i}* marker;
- The {rθ} marker follows an angular rotation θ on the *y-axis of the* {rψ} marker;
- The marker {rθ} follows an angle rotation φ on the x-axis of the marker {rθ} and is confused with the mark {b}.

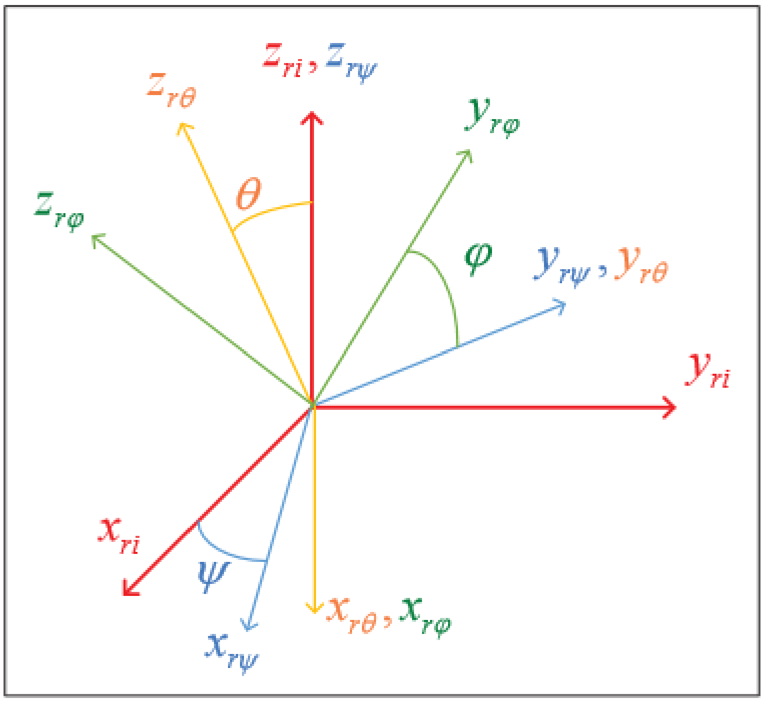


Figure 2.2 UAV auxiliary markers

Rotation matrices can be used to describe changes in the benchmark. We will call ${}_{y}^{x}R$ the rotation matrix allowing to go from the {y} to the {x} marker. We then have the following rotation matrices:

$$ {}_{r\psi}^{r\theta}\boldsymbol{R} = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} $$ *Equation 2.1*

$$ {}_{r\theta}^{b}\boldsymbol{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{bmatrix} $$ *Equation 2.2*

$$ {}_{r_i}^{r\psi}\boldsymbol{R} = \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} $$ *Equation 2.3*

From equations (2.1), (2.2) and (2.3), we can find the rotation matrix ${}_{ri}^{b}R$ allowing to pass from the inertial reference mark {ri} to the body reference mark {b}:

$$ {}_{r_i}^{b}\boldsymbol{R} = {}_{r_\theta}^{b}\boldsymbol{R}{}_{r_\psi}^{r_\theta}\boldsymbol{R}{}_{r_i}^{r_\psi}\boldsymbol{R} $$ *Equation 2.4*

$$ {}_{r_i}^{b}\boldsymbol{R} = \begin{bmatrix} \cos(\theta)\cos(\psi) & \cos(\theta)\sin(\psi) & -\sin(\theta) \\ \sin(\phi)\sin(\theta)\cos(\psi)-\cos(\phi)\sin(\psi) & \sin(\phi)\sin(\theta)\sin(\psi)+\cos(\phi)\cos(\psi) & \sin(\phi)\cos(\theta) \\ \cos(\phi)\sin(\theta)\cos(\psi)+\sin(\phi)\sin(\psi) & \cos(\phi)\sin(\theta)\sin(\psi)-\sin(\phi)\cos(\psi) & \cos(\phi)\cos(\theta) \end{bmatrix} $$ *Equation 2.5*

In other words, an ${}^{i}X$ vector expressed in the {i} marker can be expressed in the {b} marker by a ${}^{b}X$ vector using the following expression:

$$ {}^{b}\boldsymbol{X} = {}_{i}^{b}\boldsymbol{R}{}^{i}\boldsymbol{X} = {}_{r_i}^{b}\boldsymbol{R}{}^{i}\boldsymbol{X} $$ *Equation 2.6*

4. Angular velocities:

The quadcopter has a gyroscope to record the angular velocities of the UAV with respect to the {i} marker and expressed in the {b} marker. These speeds can be symbolized by a vector ${}^{b}\omega$. This vector is composed of three coordinates *p,q,r* describing respectively the angular velocities around the *x, y* and *z* axes of frame *{b}:*

Equation 2.1

$$ {}^{b}\omega = \begin{bmatrix} p \\ q \\ r \end{bmatrix} $$

In order to be able to implement a command to control the UAV orientation, the Euler angle velocities must be expressed. It is therefore important to find a relation between the data from the gyroscope and the angles. It can be shown that the change in angular rate from one benchmark to another is:

$$ {}^{i+1}\boldsymbol{\omega}_{i+1} = {}_{i}^{i+1}\boldsymbol{R}{}^{i}\boldsymbol{\omega}_{i} + \dot{\boldsymbol{\theta}}_{i+1}{}^{i+1}\hat{\boldsymbol{Z}}_{i+1} $$ *Equation 2.8*

With ${}^{i+1}\omega_{i+1}$ the angular velocity of the marker {*i*+1} with respect to the inertial marker expressed in the marker {i+1}, ${}_{i}^{i+1}R^{i}$ the rotation matrix passing from the old marker to the new one, θ_{i+1} The angular velocity of the angle of rotation and ${}^{i+1}Z_{i+1}$ the collinear unit vector with the axis of rotation in the new marker. From (2.8) and by recurrence, we can deduce:

$$^{b}\boldsymbol{\omega} = \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + {}^{b}_{r\theta}\boldsymbol{R} \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + {}^{b}_{r\theta}\boldsymbol{R}^{r\theta}_{r\psi}\boldsymbol{R} \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix}$$ *Equation 2.9*

$$^{b}\boldsymbol{\omega} = \begin{bmatrix} 1 & 0 & -\sin(\theta) \\ 0 & \cos(\phi) & \sin(\phi)\cos(\theta) \\ 0 & -\sin(\phi) & \cos(\phi)\cos(\theta) \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix}$$ *Equation 2.10*

$$^{b}\boldsymbol{\omega} = \boldsymbol{J}\dot{\boldsymbol{\eta}}$$ *Equation 2.11*

5. Relationship between Euler angles and quaternions:

In order to see the limits of modelling with Euler angles, the J matrix must be inverted. We end up with this result:

$$\boldsymbol{J}^{-1} = \begin{bmatrix} 1 & \sin(\phi)\tan(\theta) & \cos(\phi)\tan(\theta) \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi)\sec(\theta) & \cos(\phi)\sec(\theta) \end{bmatrix}$$ *Equation 2.12*

The relation (2.12) allows to see the limits of UAV modeling with Euler angles. $\ddot{\varphi}$ and $\ddot{\psi}$ are undefined when θ is at 90 degrees. The UAV then loses two degrees of freedom. This phenomenon is called dial blocking. Using Euler angles forbids the UAV to make more complicated trajectories (to make acrobatic trajectories for example). This project being limited to make simple trajectories, the orientation of the UAV will never reach these singularities. However, these singularities can pose a problem when trying to recover orientation angles using sensors (gyroscope, accelerometer and magnetometer). The UAVs used in this project use quaternions to record orientations, so it is important to understand the fundamentals and the relationship between Euler angles and quaternions.[7] Quaternions are used to describe the orientation of a benchmark using a unit vector and an angle θ_{ref} (see Figure 2.3).

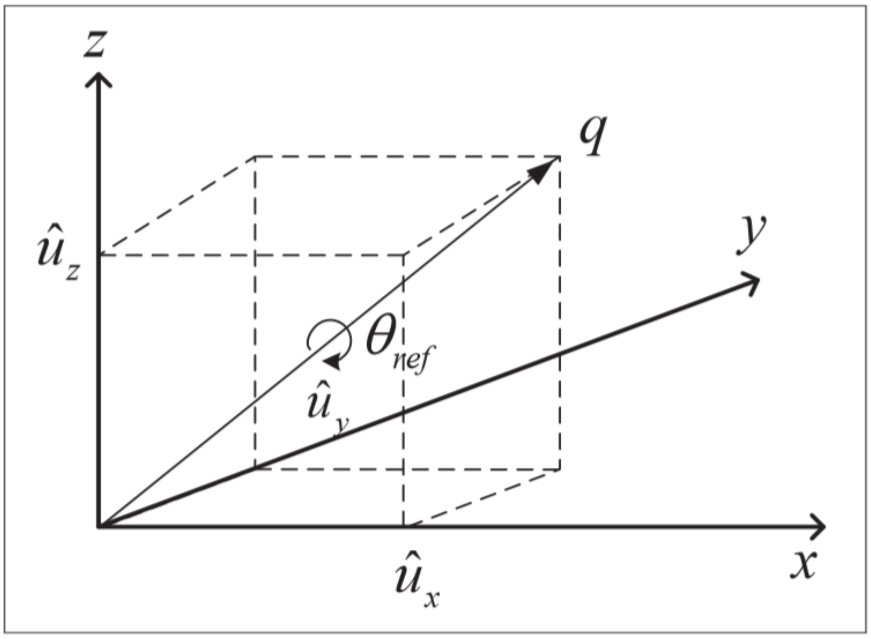


Figure 2.3 Representation of a quaternion

One way to define a quaternion q is to define its elements as follows:

$$\boldsymbol{q} = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix} = \begin{bmatrix} \cos(\frac{\theta_{ref}}{2}) & \hat{u}_x \sin(\frac{\theta_{ref}}{2}) & \hat{u}_y \sin(\frac{\theta_{ref}}{2}) & \hat{u}_z \sin(\frac{\theta_{ref}}{2}) \end{bmatrix} \quad \textit{Equation 2.13}$$

With $\hat{u}_x$, $\hat{u}_y$ and $\hat{u}_z$ the elements of the unit vector. The conjugate of the quaternion q is then represented as follows:

$$\boldsymbol{q}_{conj} = \begin{bmatrix} q_1 & -q_2 & -q_3 & -q_4 \end{bmatrix} \quad \textit{Equation 2.14}$$

If we define ${}^a{}_b q$ as the quaternion to describe the orientation of a marker *{b} comparing to* a marker *{a}* and *goes* a given vector v_a in the marker *{a}*, the relation describing the vector v_a into the marker *{b}* is:

$$\begin{bmatrix} 0 & {}^b\boldsymbol{v} \end{bmatrix} = {}^b_a\boldsymbol{q} \otimes \begin{bmatrix} 0 & {}^a\boldsymbol{v} \end{bmatrix} \otimes {}^b_a\boldsymbol{q}_{conj} \quad \textit{Equation 2.15}$$

The symbol $\otimes$ being the product of quaternions using Hamilton's principle. We can then, from relations (2.13), (2.14) and (2.15), establish the rotation matrix of the UAV orientation according to the elements of the quaternion:

$$ {}_b^i\boldsymbol{R} = \begin{bmatrix} 2q_1^2 - 1 + 2q_2^2 & 2(q_2q_3 - q_1q_4) & 2(q_2q_4 + q_1q_3) \\ 2(q_2q_3 + q_1q_4) & 2q_1^2 - 1 + 2q_3^2 & 2(q_3q_4 - q_1q_2) \\ 2(q_2q_4 - q_1q_3) & 2(q_3q_4 + q_1q_2) & 2q_1^2 - 1 + 2q_4^2 \end{bmatrix} $$

Equation 2.16

From the inverse rotation matrix of equation (2.5) and by identification, we find the following expressions:

$$ \psi = \arctan 2(2q_2q_3 + 2q_1q_4, 2q_1^2 - 1 + 2q_2^2) $$

Equation 2.17

$$ \theta = -\arcsin(2q_2q_4 - 2q_1q_3) $$

Equation 2.18

$$ \phi = \arctan 2(2q_3q_4 + 2q_1q_2, 2q_1^2 - 1 + 2q_4^2) $$

Equation 2.19

6. Conclusion:

In this chapter, we highlighted the mathematical model for the mechanical movements of the drone. This model will help us understand how the drone actually work by knowing its different variables that can be adjusted and controlled which will eventually help us build the ROS package to control the drone.

Chapter 3 : Software in The Loop Simulation SITL

1. Introduction:

Simulators allow PX4 flight code to control a computer modeled vehicle in a simulated "world". You can interact with this vehicle just as you might with a real vehicle, using QGroundControl, an offboard API, or a radio controller/gamepad.

PX4 supports both Software In The Loop (SITL) simulation, where the flight stack runs on computer (either the same computer or another computer on the same network) and Hardware In the Loop (HITL) simulation using a simulation firmware on a real flight controller board.

2. Software in the loop simulation environment:

The diagram below shows a typical SITL simulation environment for any of the supported simulators. The different parts of the system connect via UDP, and can be run on either the same computer or another computer on the same network.[3]

- PX4 uses a simulation-specific module to connect to the simulator's local TCP port 4560. Simulators then exchange information with PX4 using the Simulator MAVLink API described above. PX4 on SITL and the simulator can run on either the same computer or different computers on the same network.
- PX4 uses the normal MAVLink module to connect to ground stations (which listen on port 14550) and external developer APIs like MAVSDK or ROS (which listen on port 14540).
- A serial connection is used to connect Joystick/Gamepad hardware via QGroundControl.

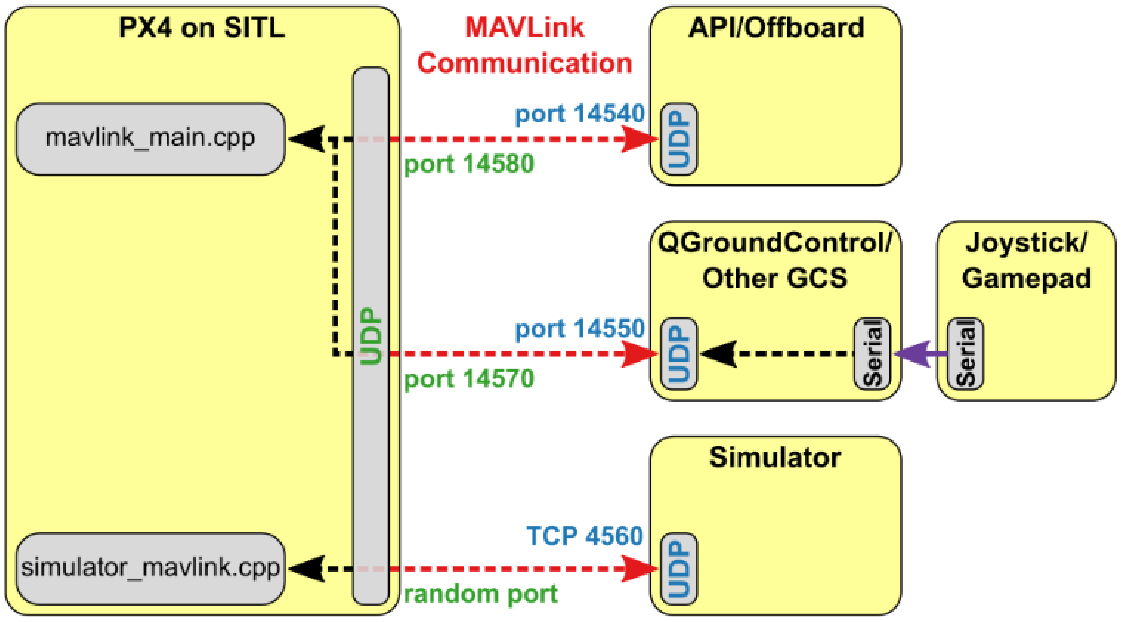


Figure 3.1 Cycle SITL simulation enivrement

2.1. Gazebo simulator:

Robot simulation is an essential tool in every roboticist's toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments.

It is a 3D dynamic simulator with the ability to simulate various robots in indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces.

Gazebo interface consists of multiple sections. First of them - scene is the main part of the simulator. This is where the simulated objects are animated and interact with the environment. Second section are panels.[7]

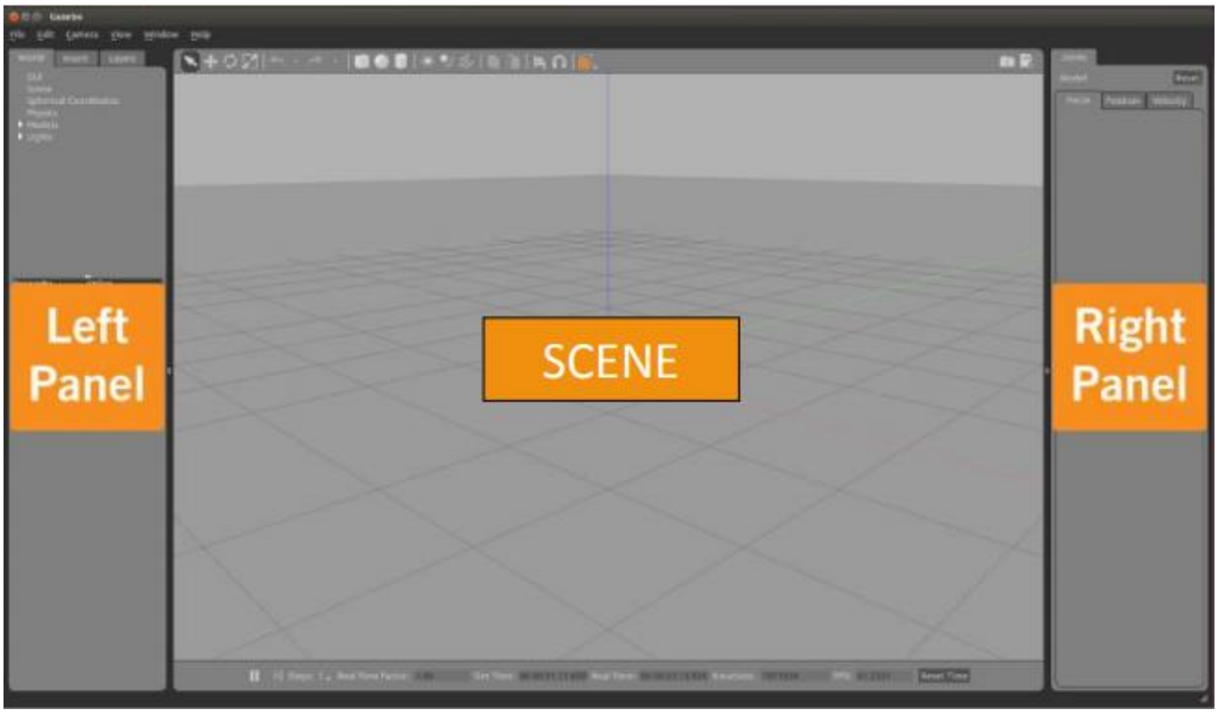


Figure 3.2 Gazebo simulator interface

The left panel appears by default when you launch Gazebo. There are three tabs in the panel:

- WORLD: displays the models that are currently in the scene, and allows you to view and modify model parameters, like their pose
- INSERT: The Insert tab is where you add new objects (models) to the simulation.
- LAYERS: The Layers tab organizes and displays the different visualization groups that are available in the simulation.

The right panel is hidden by default. It is used to interact with the mobile parts of a selected model (the joints). If there are no models selected in the Scene, the panel does not display any information.

Third section are two toolbars. One of them is located above the Scene and one below. The upper toolbar is a main one and includes most-used options: select, move, rotate, scale, create shape, copy and paste, as shown in this figure:

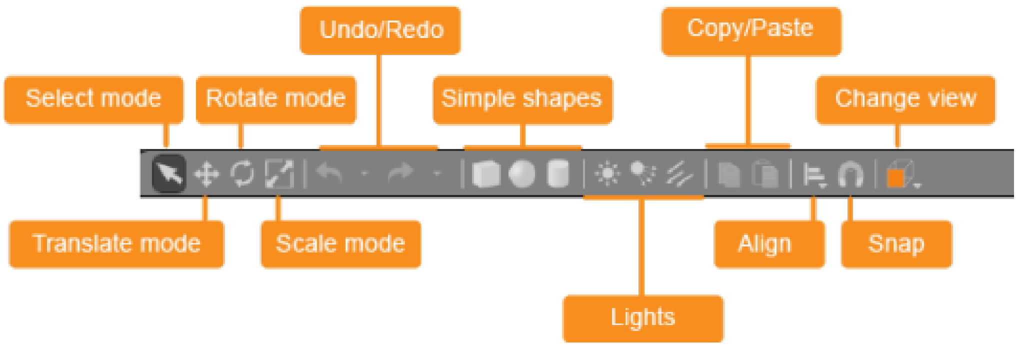


Figure 3.3 Gazebo toolbox

The Bottom Toolbar is useful during the simulation. It displays simulation time, real time and Real Time Factor, which is a relationship between two previous ones. The state of the world in Gazebo is calculated once per iteration. You can see the number of iterations on the right side of the bottom toolbar. Each iteration advances simulation by a fixed number of seconds, called the step size.

2.2 PX4 Firmware:

It is an open source repository holds the PX4 flight control solution for drones, with the main applications located in the src/modules directory. It also contains the PX4 Drone Middleware Platform, which provides drivers and middleware to run drones.

This repository can be found on GitHub website via this link:

https://github.com/PX4/Firmware

To clone this repository into our system, a git command is used:

```
git clone https://github.com/PX4/Firmware.git
```

This repository will play the role of the flight controller, instead of using the PIXHAWK hardware itself, it contains different worlds and models, and support many types of airframes that can be loaded into Gazebo to be simulated, as well as its compatibility with the QGROUNDCONTROL that is the ground control station which we'll be discussing in what follows.

2.3 QGroundControl:

QGroundControl provides full flight control and mission planning for any MAVLink enabled drone. Its primary goal is ease of use for professional users and developers. All the code is open-source source, so anyone can contribute and evolve it as they want.[3]

The key features of QGROUNDCONROL are:

- Full setup/configuration of ArduPilot and PX4 powered vehicles.
- Flight support for vehicles running PX4 and ArduPilot (or any other autopilot that communicates using the MAVLink protocol).
- Mission planning for autonomous flight.
- Flight map display showing vehicle position, flight track, waypoints and vehicle instruments.
- Video streaming with instrument display overlays.
- Support for managing multiple vehicles.
- QGC runs on Windows, OS X, Linux platforms, iOS and Android devices.

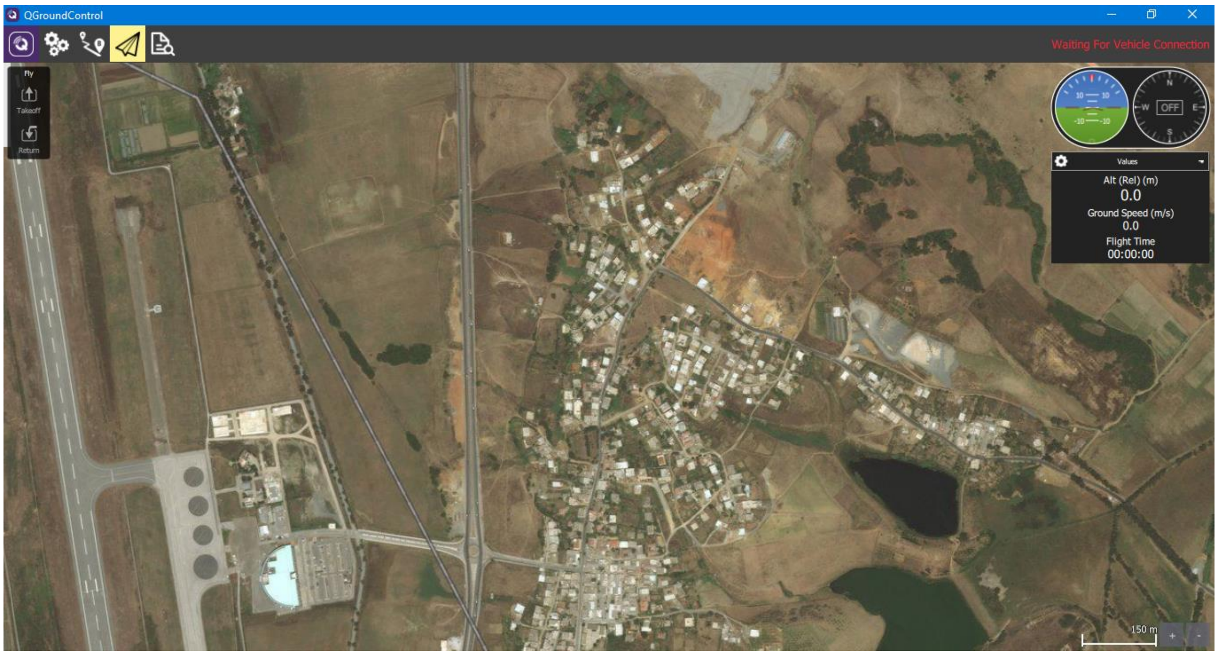


Figure 3.4 QGROUNDCONTROL user interface

2.4. MAVROS:

2.4.1. Definition:

MAVROS is the official supported bridge between ROS and the MAVLink protocol that we discussed earlier. It is a ready ROS package that enables MAVLink extendable communication between computers running ROS, MAVLink enabled autopilots, and MAVLink enabled GCS.

2.4.2. Installation:

First step is to create a ROS workspace where the MAVROS package will be installed in, following these commands:

```
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws
catkin init
wstool init src
```

Next step is to install ROS Python tools: wstool (for retrieving sources), rosinstall, and catkin_tools, through the following command:

```
sudo apt-get install python-catkin-tools python-rosinstall-generator -y
init ~/catkin_ws/src
```

The following step is to install MAVLINK and MAVROS:

- MAVLink:

```
rosinstall_generator --rosdistro kinetic mavlink | tee /tmp/mavros.rosinstall
```

- MAVROS:

```
rosinstall_generator --upstream-development mavros | tee -a /tmp/mavros.rosinstall
```

Now creating workspace and dependencies:

```
wstool merge -t src /tmp/mavros.rosinstall

wstool update -t src -j4

rosdep install --from-paths src --ignore-src -y
```

Next is installing GeofraphicLib datasets:

```
./src/mavros/mavros/scripts/install_geographiclib_datasets.sh
```

Build source:

```
catkin build
```

And finally, sourcing the setup.bash file:

```
source devel/setup.bash
```

Launching and using MAVROS protocol will discussed in the following.

3. Results and interpretation:

3.1. Codes to build the ROS package:

3.1.1. Creation of the catkin workspace:

The first step in this process is to create the workspace we will be working in (as mentioned in chapter 1), following these commands:

```
mkdir -p /project_ws/src
cd project_ws/src
catkin_init_workspace
cd ..
catkin_make
```

Finally add this line “source /project_ws/devel.bash” to the .bashrc file.

3.1.2. Creation of the ROS package:

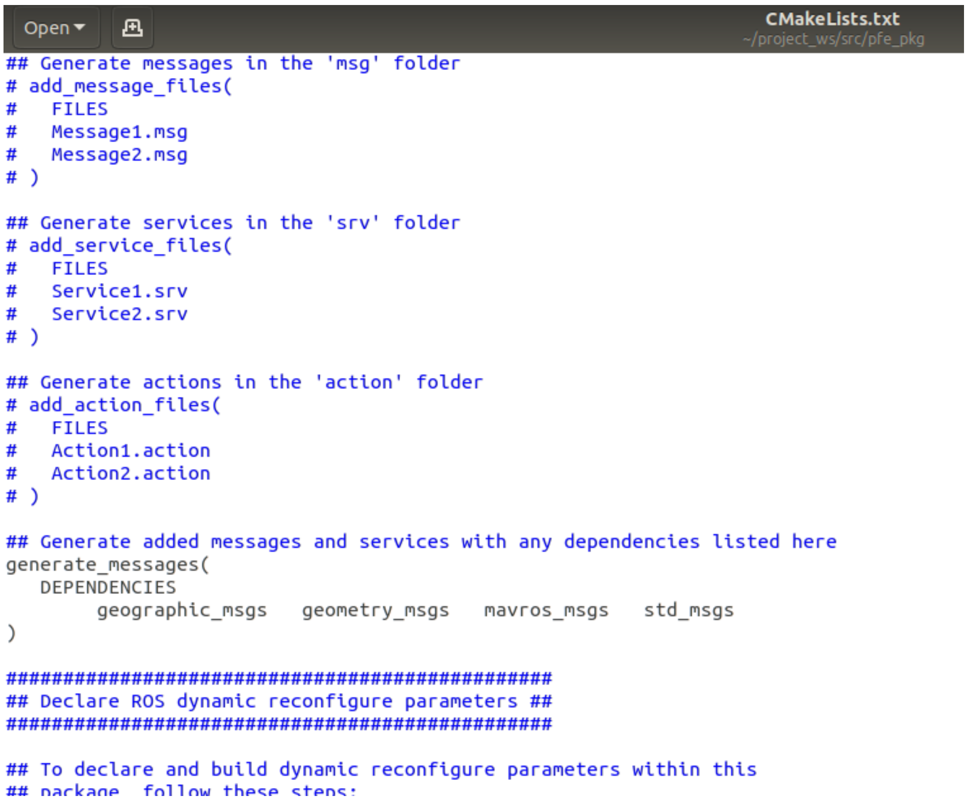
The step that follows is to create our ROS package which will contain the nodes (programs) that will run our drone:

```
cd project_ws/src
catkin_create_pkg pfe_pkg std_msgs rospy mavros mavros_msgs geometry_msgs
geographic_msgs tf sensor_msgs
```

Whereas:

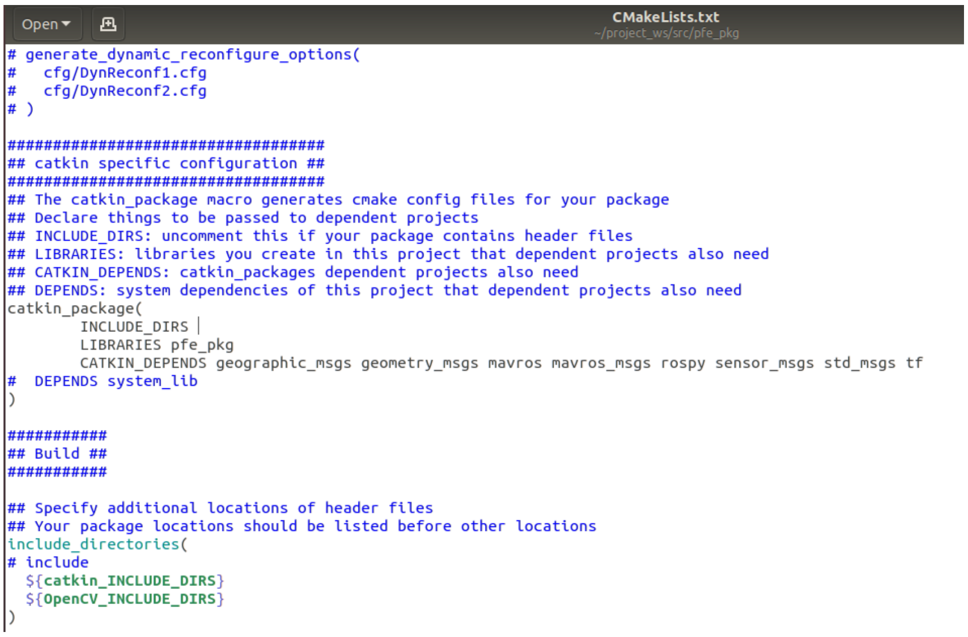
- pkg_pfe is the name we gave to our package
- “std_msgs rospy mavros mavros_msgs geometry_msgs geographic_msgs tf sensor_msgs” are the dependencies that our programs need to be executed

The next step is to go inside the ROS package we created, open the CMakeList.txt file and do few modifications (adding libraries and dependencies…) as shown in figure 3.5 and 3.6



```
## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )

## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )

## Generate actions in the 'action' folder
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

## Generate added messages and services with any dependencies listed here
generate_messages(
   DEPENDENCIES
        geographic_msgs   geometry_msgs   mavros_msgs   std_msgs
)

################################################
## Declare ROS dynamic reconfigure parameters ##
################################################

## To declare and build dynamic reconfigure parameters within this
## package, follow these steps:
```

Figure 3.5 Adding dependencies to the CMaleLists.txt file



```
# generate_dynamic_reconfigure_options(
#   cfg/DynReconf1.cfg
#   cfg/DynReconf2.cfg
# )

###################################
## catkin specific configuration ##
###################################
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if your package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
        INCLUDE_DIRS
        LIBRARIES pfe_pkg
        CATKIN_DEPENDS geographic_msgs geometry_msgs mavros mavros_msgs rospy sensor_msgs std_msgs tf
#  DEPENDS system_lib
)

###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
# include
  ${catkin_INCLUDE_DIRS}
  ${OpenCV_INCLUDE_DIRS}
)
```

Figure 3.6 Adding the libraries to the CMakeLists.txt file

Next is to cd into the src folder of the ROS package, and put there the python nodes(programs) that we will use to run the drone.

The final step is to go outside the ROS package folder and run the following command:

```
catkin_make
```

The result is shown in figure 3.7, 3.8, 3.9 and 3.10:

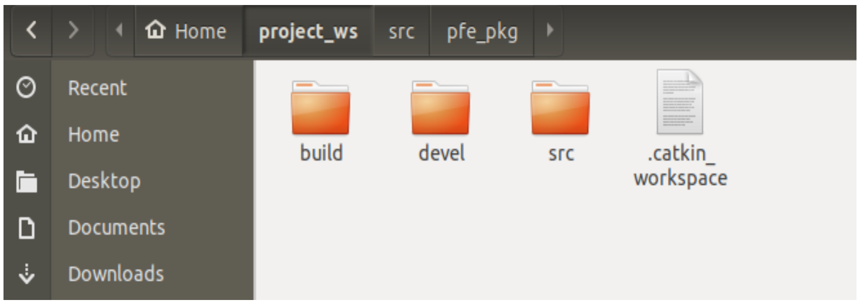


Figure 3.7 workspace folder

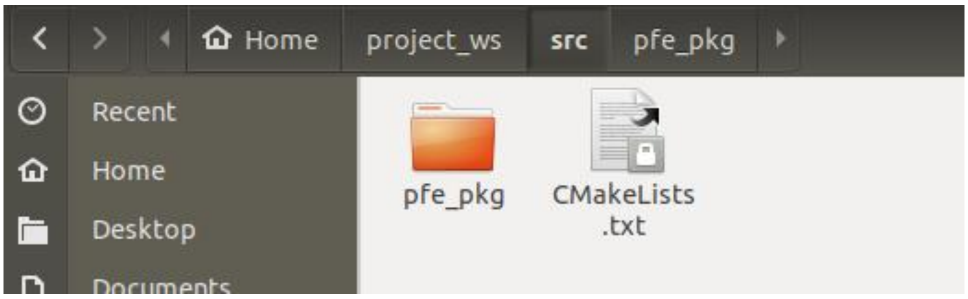


Figure 3.8 ROS package folder

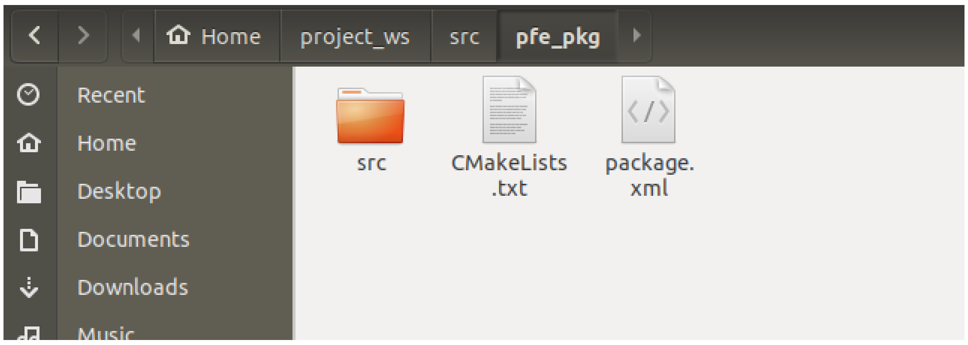


Figure 3.9 Inside the ROS package

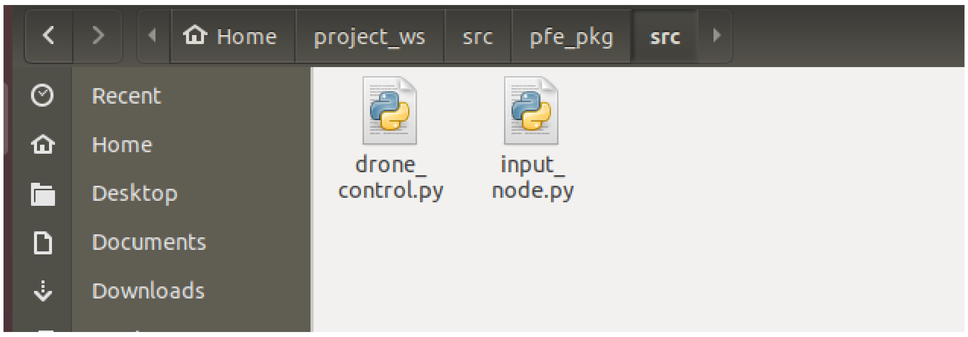


Figure 3.10 The nodes (programs) inside the src folder of the ROS package

3.1.3. Nodes:

In this project we will use two main nodes that we created (in addition to the MAVROS node that we discussed earlier). The First node is the “drone_control.py” node, and the second one is “input_node.py” node.

User input node will take the flight mode requested by the user, publish to the user input topic, then the drone control node will subscribe to that topic and send data to the MAVROS node according to the flight mode message.

Using rqt_graph command, we can see all the nodes that our system consists of, as well as the capacity to see active topics as well.

Figure 3.11, shows the active nodes in our program:

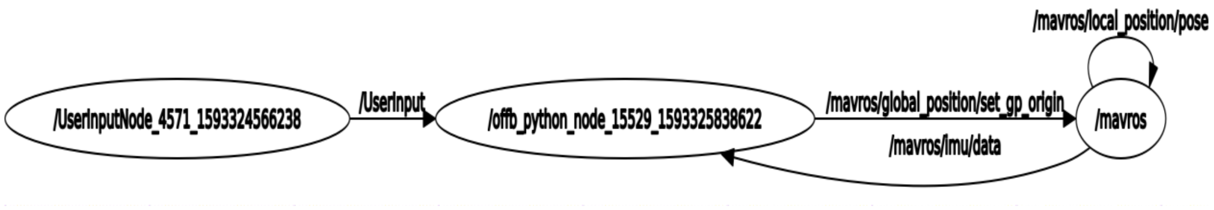

/UserInputNode_4571_1593324566238
/UserInput
/offb_python_node_15529_1593325838622
/mavros/global_position/set_gp_origin
/mavros/imu/data
/mavros
/mavros/local_position/pose

Figure 3.11 Active nodes

While figure 3.12 shows all the nodes as well as the active topics of this system:

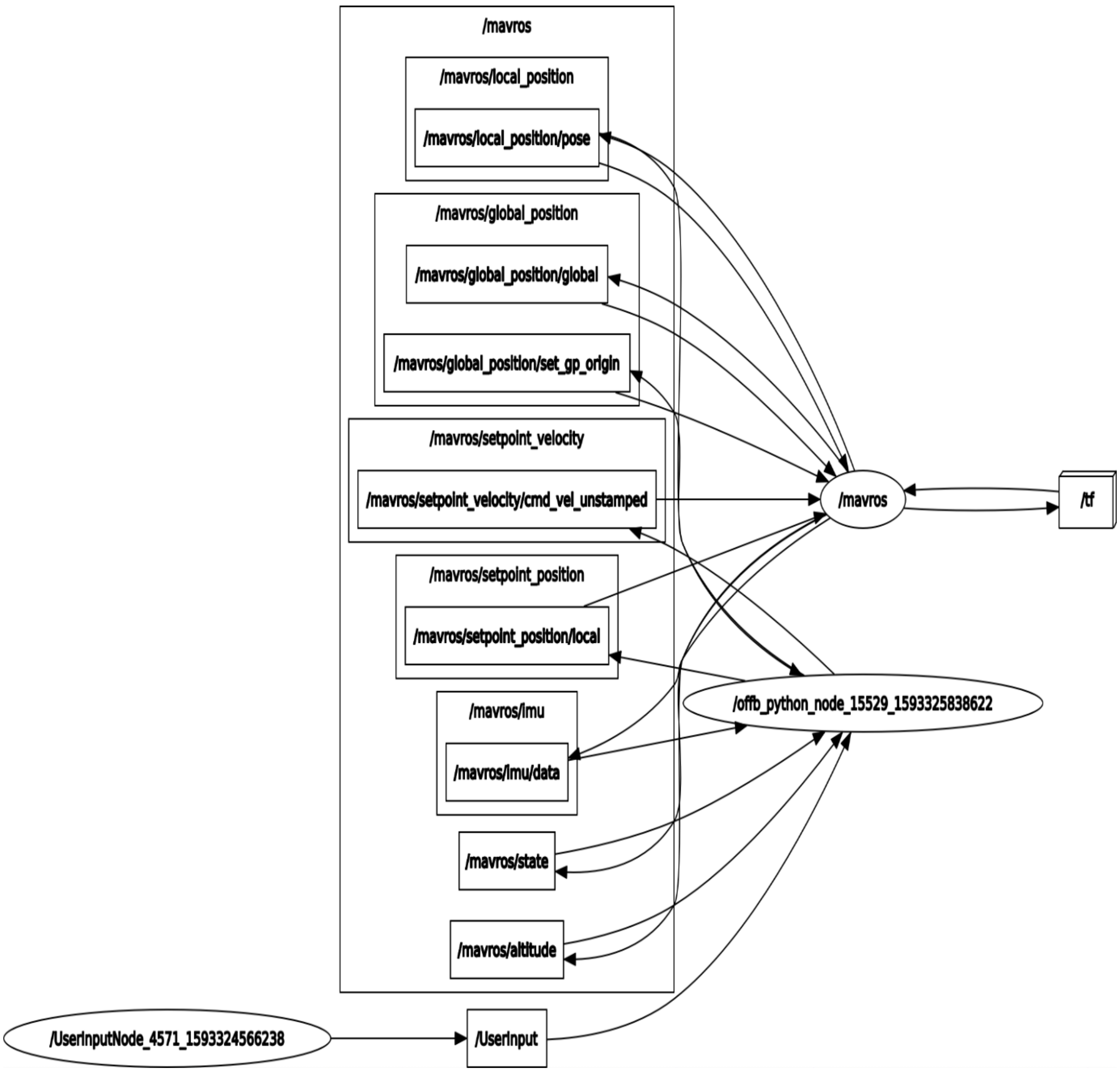

/mavros
/mavros/local_position
/mavros/local_position/pose
/mavros/global_position
/mavros/global_position/global
/mavros/global_position/set_gp_origin
/mavros/setpoint_velocity
/mavros/setpoint_velocity/cmd_vel_unstamped
/mavros/setpoint_position
/mavros/setpoint_position/local
/mavros/imu
/mavros/imu/data
/mavros/state
/mavros/altitude
/mavros
/tf
/offb_python_node_15529_1593325838622
/UserInputNode_4571_1593324566238
/UserInput

Figure 3.12 Active nodes and topics

N.B:

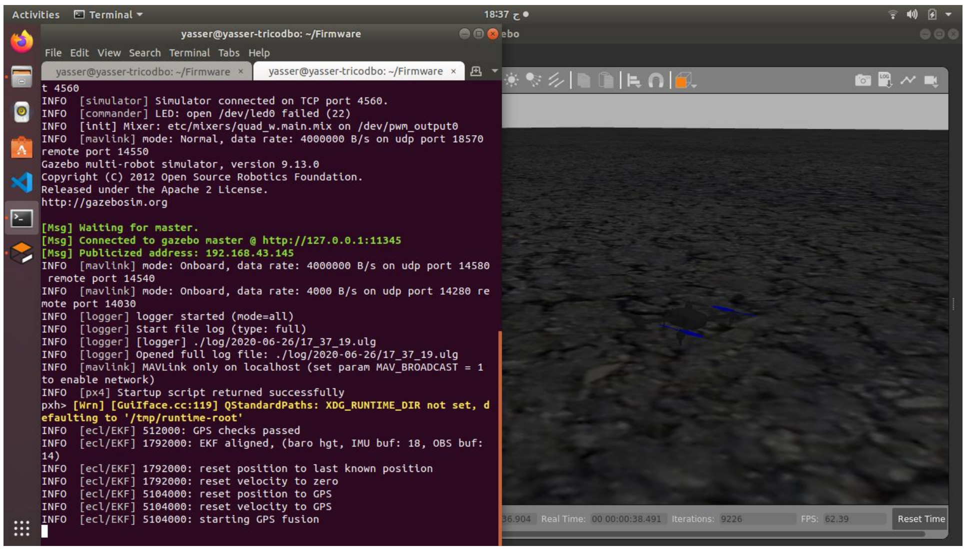
The source code of the nodes was built using python programming language, and the scripts can be found in the appendix A and B document.

3.2. ROS with SITL Gazebo simulation:

To run the simulation, we have to follow few steps, the first one is to launch the PX4 SITL Gazebo simulation, then we launch the MAVROS node, that will enable the communication between our ROS package and the PX4 software through the MAVLink protocol, then finally running the drone control node followed by the user input node. More details are in what follows:

- As mentioned earlier, the first thing to do is launch the PX4 SITL Gazebo simulation, this will load a drone model called "iris" in the Gazebo simulation scene, as shown in figure 3.13.

Command: sudo make px4_sitl gazebo


Activities
Terminal
18:37
yasser@yasser-tricodbo: ~/Firmware
File Edit View Search Terminal Tabs Help
yasser@yasser-tricodbo: ~/Firmware
yasser@yasser-tricodbo: ~/Firmware
Reset Time

```
t 4560
INFO  [simulator] Simulator connected on TCP port 4560.
INFO  [commander] LED: open /dev/led0 failed (22)
INFO  [init] Mixer: etc/mixers/quad_w.main.mix on /dev/pwm_output0
INFO  [mavlink] mode: Normal, data rate: 4000000 B/s on udp port 18570
remote port 14550
Gazebo multi-robot simulator, version 9.13.0
Copyright (C) 2012 Open Source Robotics Foundation.
Released under the Apache 2 License.
http://gazebosim.org

[Msg] Waiting for master.
[Msg] Connected to gazebo master @ http://127.0.0.1:11345
[Msg] Publicized address: 192.168.43.145
INFO  [mavlink] mode: Onboard, data rate: 4000000 B/s on udp port 14580
 remote port 14540
INFO  [mavlink] mode: Onboard, data rate: 4000 B/s on udp port 14280 re
mote port 14030
INFO  [logger] logger started (mode=all)
INFO  [logger] Start file log (type: full)
INFO  [logger] [logger] ./log/2020-06-26/17_37_19.ulg
INFO  [logger] Opened full log file: ./log/2020-06-26/17_37_19.ulg
INFO  [mavlink] MAVLink only on localhost (set param MAV_BROADCAST = 1
to enable network)
INFO  [px4] Startup script returned successfully
pxh> [Wrn] [GuiIface.cc:119] QStandardPaths: XDG_RUNTIME_DIR not set, d
efaulting to '/tmp/runtime-root'
INFO  [ecl/EKF] 512000: GPS checks passed
INFO  [ecl/EKF] 1792000: EKF aligned, (baro hgt, IMU buf: 18, OBS buf:
14)
INFO  [ecl/EKF] 1792000: reset position to last known position
INFO  [ecl/EKF] 1792000: reset velocity to zero
INFO  [ecl/EKF] 5104000: reset position to GPS
INFO  [ecl/EKF] 5104000: reset velocity to GPS
INFO  [ecl/EKF] 5104000: starting GPS fusion
```

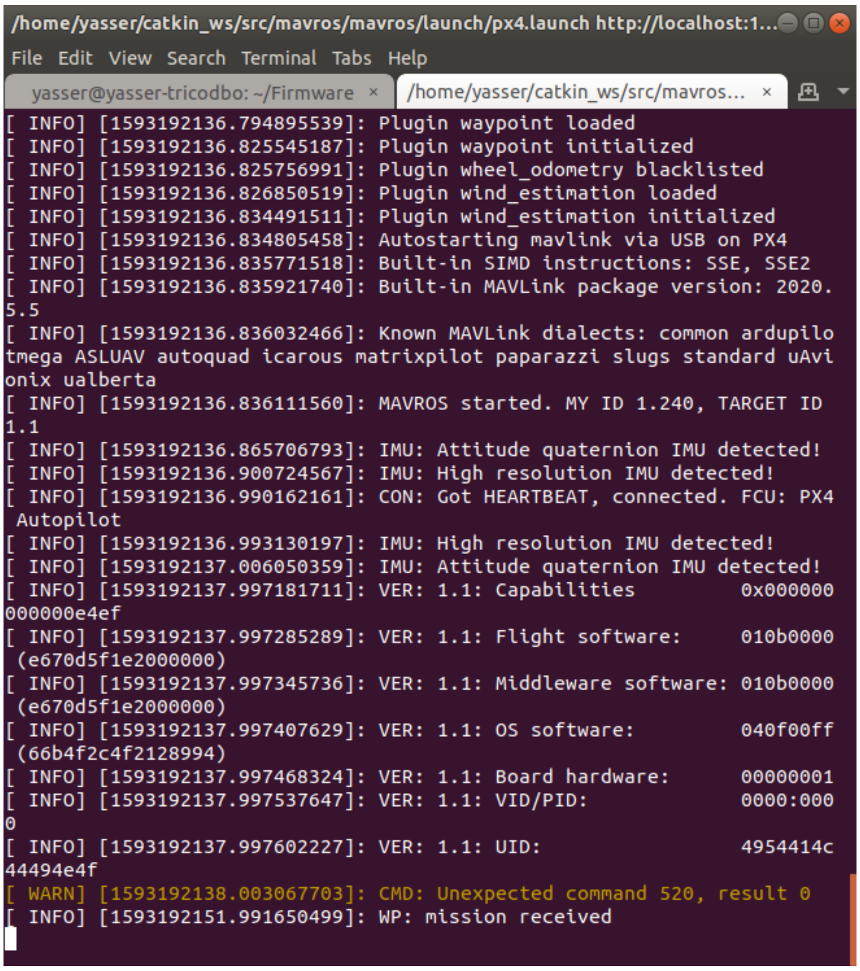
```
36.904  Real Time: 00 00:00:38.491  Iterations: 9226  FPS: 62.39
```

Figure 3.13 PX4 SITL gazebo simulation

- Next thing to do is to launch the MAVROS node to be able to connect to the PX4 firmware through the MAVLink protocol, as shown in figure 3.14:

In a new tab type:

```
roslaunch mavros px4.launch fcu_url:="udp://:14540@127.0.0.1:14557"
```


/home/yasser/catkin_ws/src/mavros/mavros/launch/px4.launch http://localhost:1...
File Edit View Search Terminal Tabs Help
yasser@yasser-tricodbo: ~/Firmware
/home/yasser/catkin_ws/src/mavros...

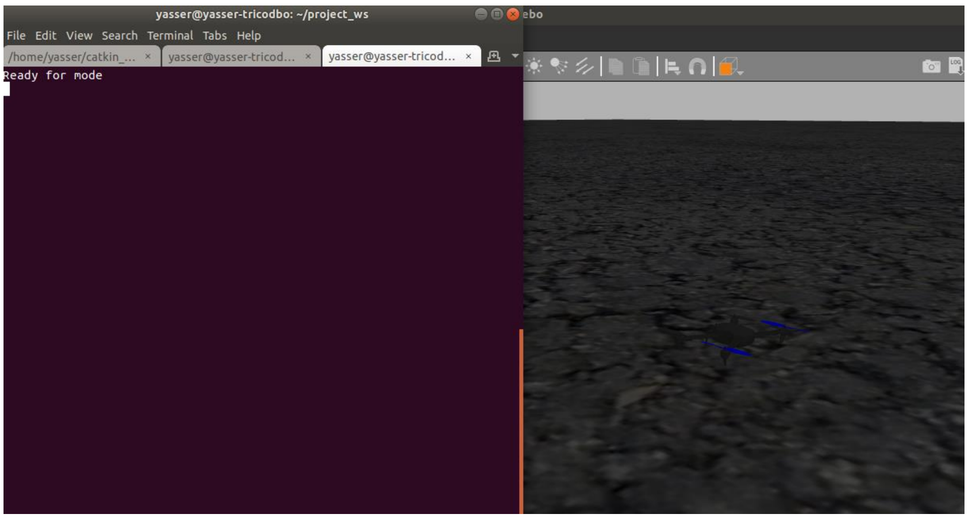
```
[ INFO] [1593192136.794895539]: Plugin waypoint loaded
[ INFO] [1593192136.825545187]: Plugin waypoint initialized
[ INFO] [1593192136.825756991]: Plugin wheel_odometry blacklisted
[ INFO] [1593192136.826850519]: Plugin wind_estimation loaded
[ INFO] [1593192136.834491511]: Plugin wind_estimation initialized
[ INFO] [1593192136.834805458]: Autostarting mavlink via USB on PX4
[ INFO] [1593192136.835771518]: Built-in SIMD instructions: SSE, SSE2
[ INFO] [1593192136.835921740]: Built-in MAVLink package version: 2020.
5.5
[ INFO] [1593192136.836032466]: Known MAVLink dialects: common ardupilo
tmega ASLUAV autoquad icarous matrixpilot paparazzi slugs standard uAvi
onix ualberta
[ INFO] [1593192136.836111560]: MAVROS started. MY ID 1.240, TARGET ID
1.1
[ INFO] [1593192136.865706793]: IMU: Attitude quaternion IMU detected!
[ INFO] [1593192136.900724567]: IMU: High resolution IMU detected!
[ INFO] [1593192136.990162161]: CON: Got HEARTBEAT, connected. FCU: PX4
 Autopilot
[ INFO] [1593192136.993130197]: IMU: High resolution IMU detected!
[ INFO] [1593192137.006050359]: IMU: Attitude quaternion IMU detected!
[ INFO] [1593192137.997181711]: VER: 1.1: Capabilities         0x000000
000000e4ef
[ INFO] [1593192137.997285289]: VER: 1.1: Flight software:     010b0000
 (e670d5f1e2000000)
[ INFO] [1593192137.997345736]: VER: 1.1: Middleware software: 010b0000
 (e670d5f1e2000000)
[ INFO] [1593192137.997407629]: VER: 1.1: OS software:         040f00ff
 (66b4f2c4f2128994)
[ INFO] [1593192137.997468324]: VER: 1.1: Board hardware:      00000001
[ INFO] [1593192137.997537647]: VER: 1.1: VID/PID:             0000:000
0
[ INFO] [1593192137.997602227]: VER: 1.1: UID:                 4954414c
44494e4f
[ WARN] [1593192138.003067703]: CMD: Unexpected command 520, result 0
[ INFO] [1593192151.991650499]: WP: mission received
```

Figure 3.14 MAVROS node output

When “mission received” message appears, it means the MAVROS node is ready.

- Now in a new tab we launch the drone control node:

Command: `cd project_ws && rosrun pfe_pkg drone_control.py`

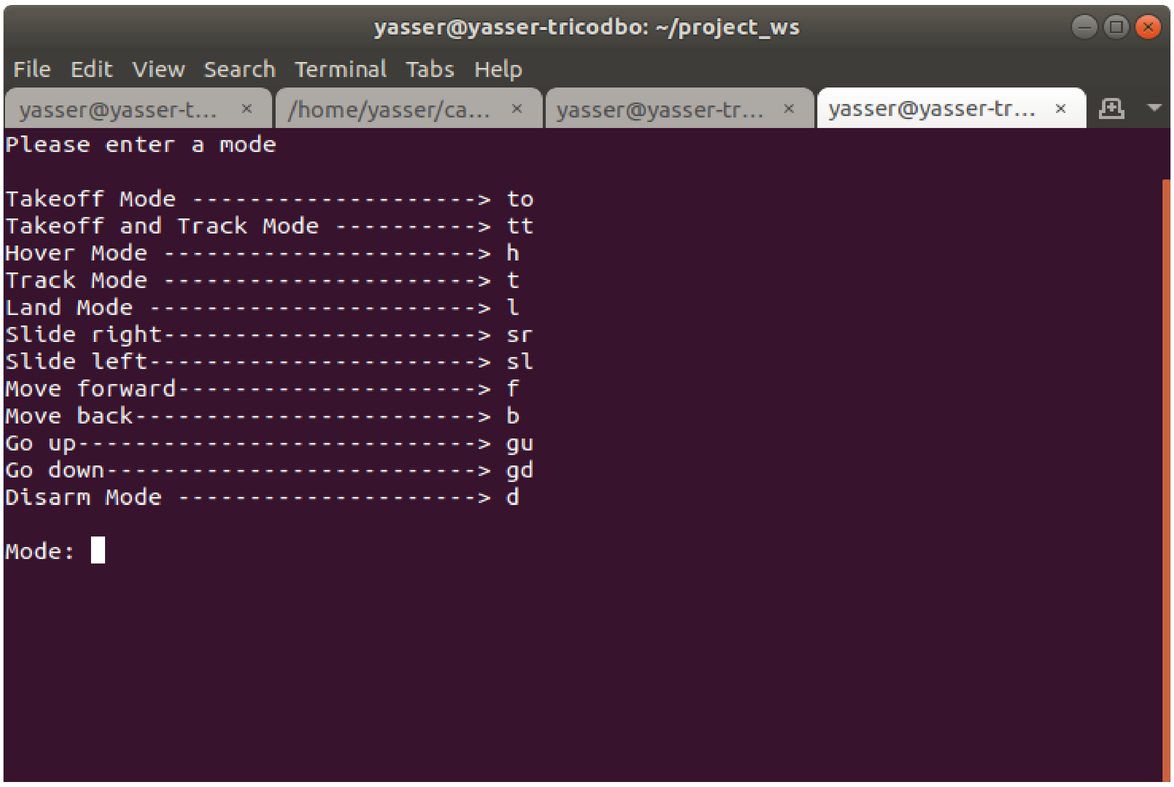


```
Ready for mode
```

Figure 3.15 The output of the drone_control.py node

- Finally, in another tab, we launch the user input node:

Command: `cd project_ws && rosrun pfe_pkg input_node.py`



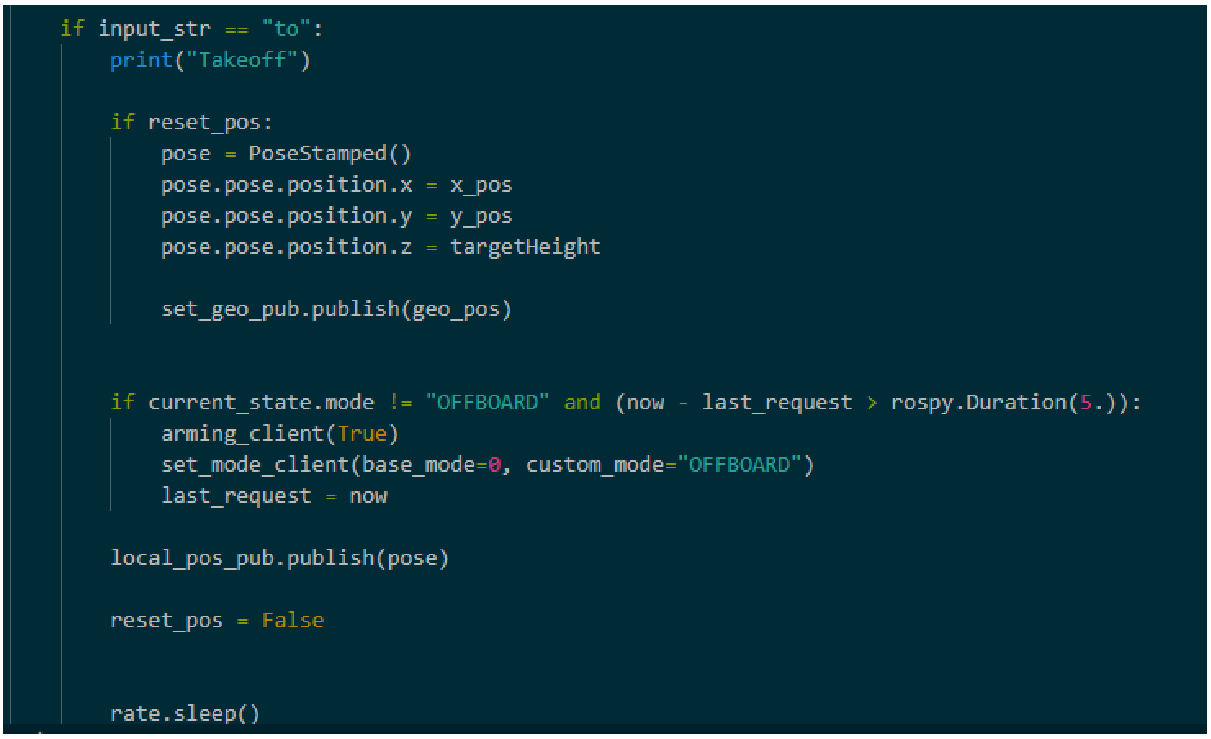
```
Please enter a mode

Takeoff Mode --------------------> to
Takeoff and Track Mode ----------> tt
Hover Mode ----------------------> h
Track Mode ----------------------> t
Land Mode -----------------------> l
Slide right----------------------> sr
Slide left-----------------------> sl
Move forward---------------------> f
Move back------------------------> b
Go up----------------------------> gu
Go down--------------------------> gd
Disarm Mode ---------------------> d

Mode:
```

Figure 3.16 The output of the input_node.py node

Testing the codes:

- Takeoff flight mode: ‘to' as user input, the drone will fly to the target height that is set in the program.

The part of the code that does that is in figure 3.17:



```
if input_str == "to":
    print("Takeoff")

    if reset_pos:
        pose = PoseStamped()
        pose.pose.position.x = x_pos
        pose.pose.position.y = y_pos
        pose.pose.position.z = targetHeight

        set_geo_pub.publish(geo_pos)


    if current_state.mode != "OFFBOARD" and (now - last_request > rospy.Duration(5.)):
        arming_client(True)
        set_mode_client(base_mode=0, custom_mode="OFFBOARD")
        last_request = now

    local_pos_pub.publish(pose)

    reset_pos = False


    rate.sleep()
```

Figure 3.17 Takeoff mode source code

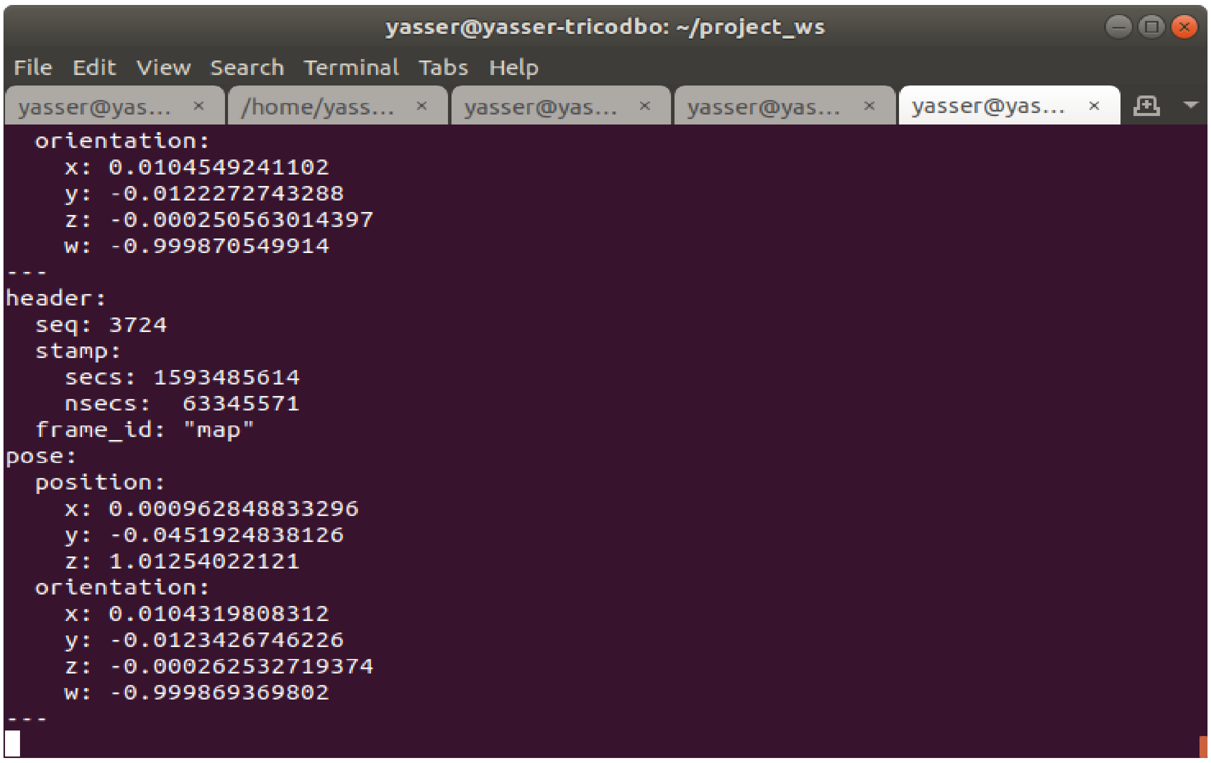
In this mode, after setting the drone to offboard mode that will allow us to command the different variables as we want, we use the /mavros/setpoint_position/pose topic, the messages in this topic are simply the coordinates of the drone comparing to a fixed xyz axes where the (0.0.0) point is the takeoff position.

By setting

pose.pose.position.x = x_pos
pose.pose.position.y = y_pos
pose.pose.position.z = targetHeight

whereas x_pos, y_pos are initialized with 0 value and targetHeight is initialized with 1 meter, the drone will only go according to the z axis with a distance of 1 meter.

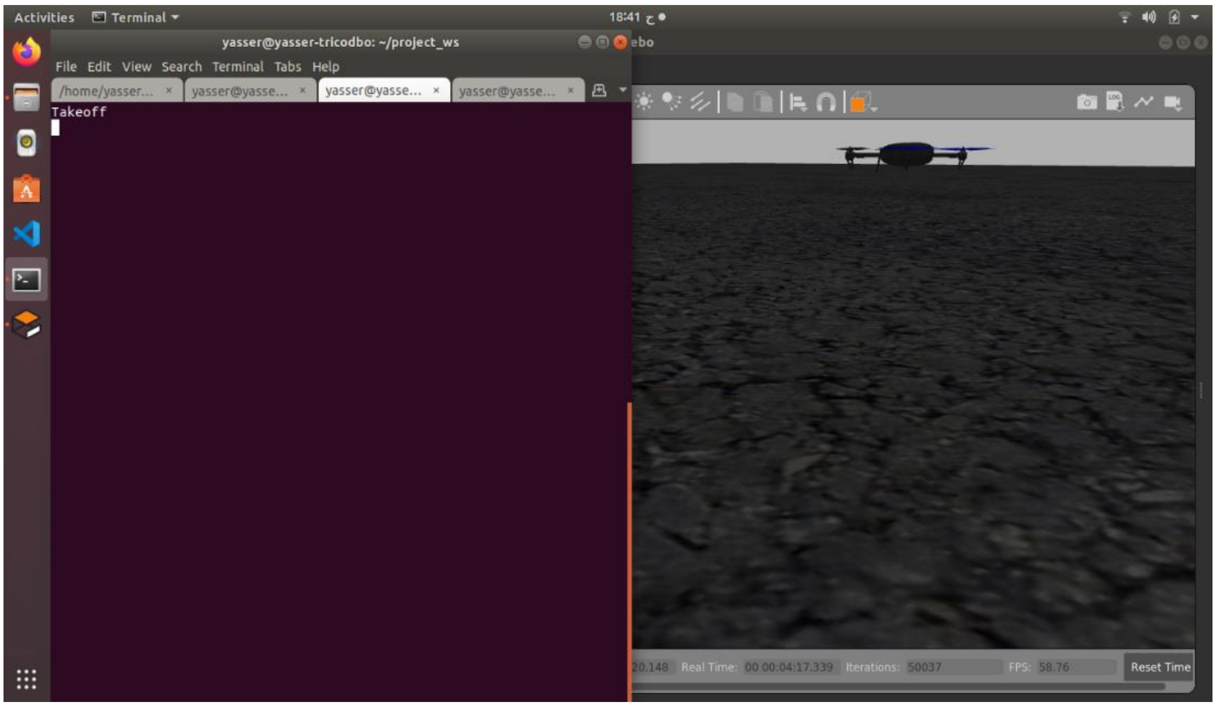
Figure 3.18 shows the different value of the /mavros/setpoint_position/pose topic (using the command rostopic echo /mavros/setpoint_position/pose):



```
  orientation:
    x: 0.0104549241102
    y: -0.0122272743288
    z: -0.000250563014397
    w: -0.999870549914
---
header:
  seq: 3724
  stamp:
    secs: 1593485614
    nsecs:  63345571
  frame_id: "map"
pose:
  position:
    x: 0.000962848833296
    y: -0.0451924838126
    z: 1.01254022121
  orientation:
    x: 0.0104319808312
    y: -0.0123426746226
    z: -0.000262532719374
    w: -0.999869369802
---
```

Figure 3.18 The position and orientation coordinates in takeoff mode

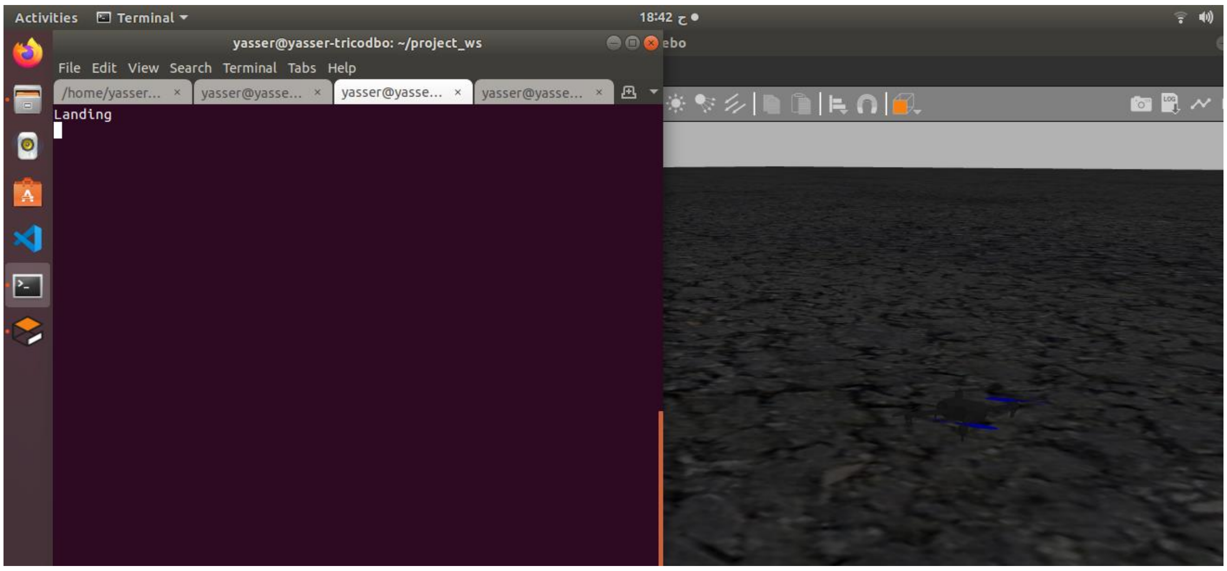
Figure 3.19 shows the drone in the simulation environment



```
Takeoff
```

Figure 3.19 Takeoff flight mode

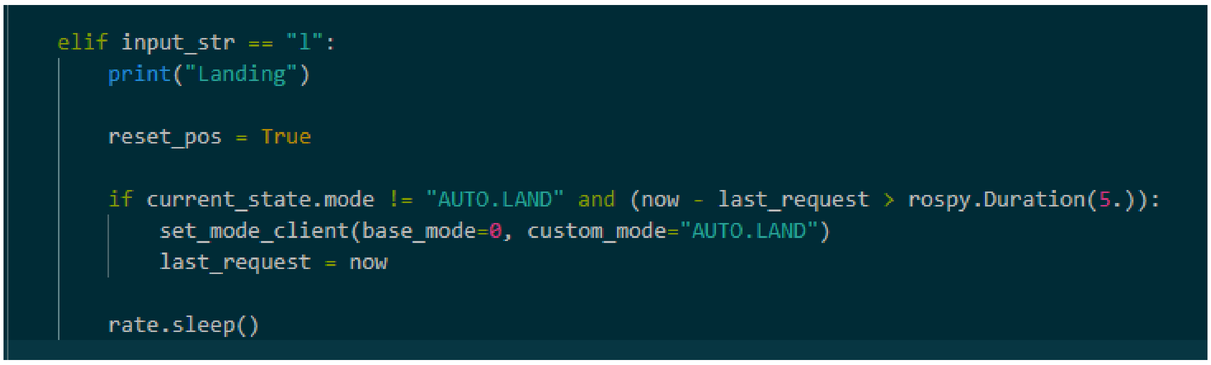
- Land flight mode: 'l' is the user input; the drone will automatically land as shown in figure 3.20:



```
Landing
```

Figure 3.20 Landing flight mode

The part of the code that does that is in figure 3.21:



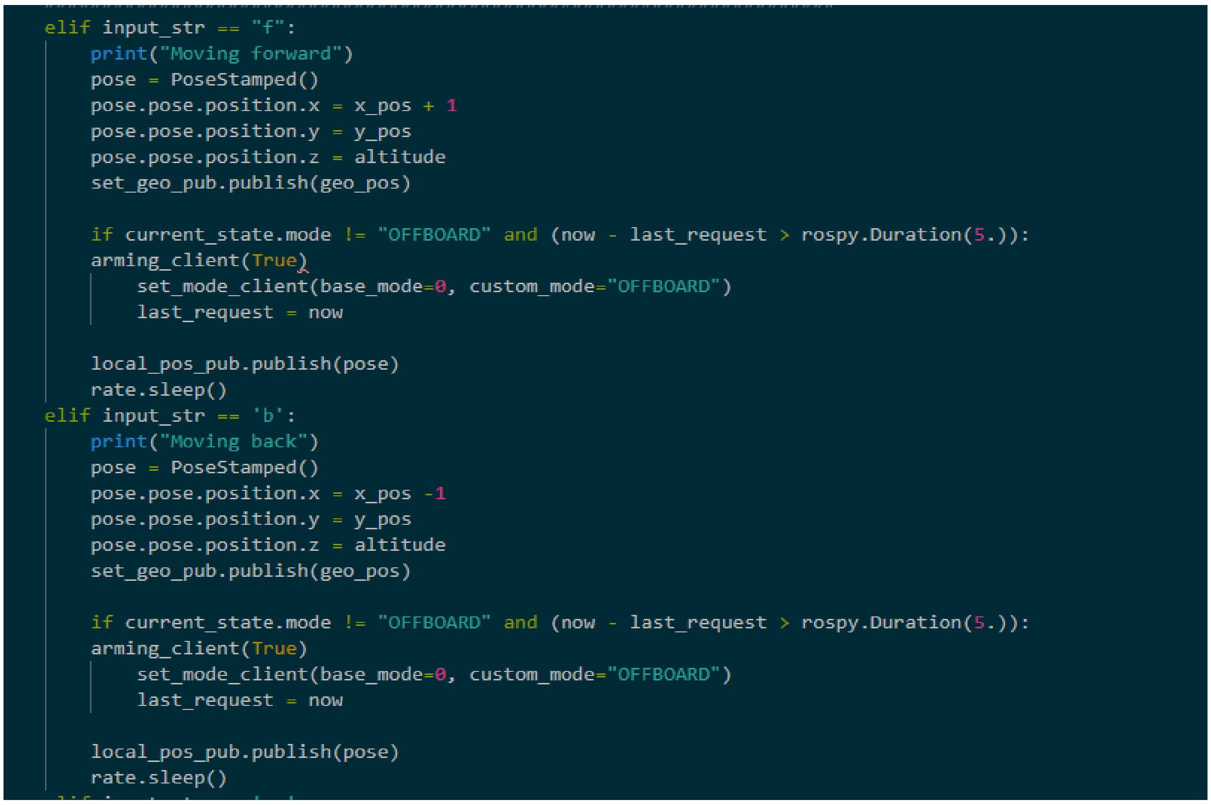
```
elif input_str == "l":
    print("Landing")

    reset_pos = True

    if current_state.mode != "AUTO.LAND" and (now - last_request > rospy.Duration(5.)):
        set_mode_client(base_mode=0, custom_mode="AUTO.LAND")
        last_request = now

    rate.sleep()
```

Figure 3.21 Landing mode source code

as we can see, we did not command the position variables of the drone, but instead we used an existing mode within the MAVROS node called "AUTO.LAND" which will make the drone land automatically.

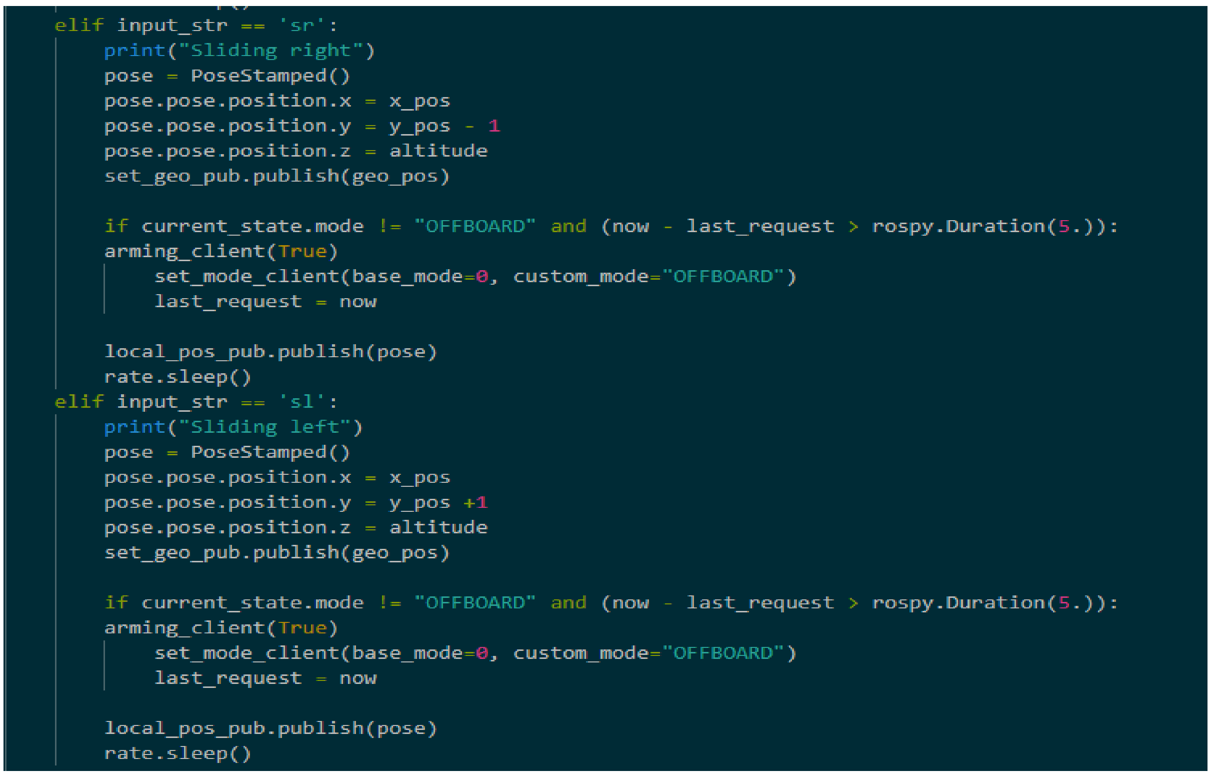
- As we saw the in the user input_node output, there are the forward, back, slide right, slide left, go up and go down modes, by choosing one of these modes, the drone will move accordingly.

The part of code for this matter is in figures 3.22, 3.23, 3.24:



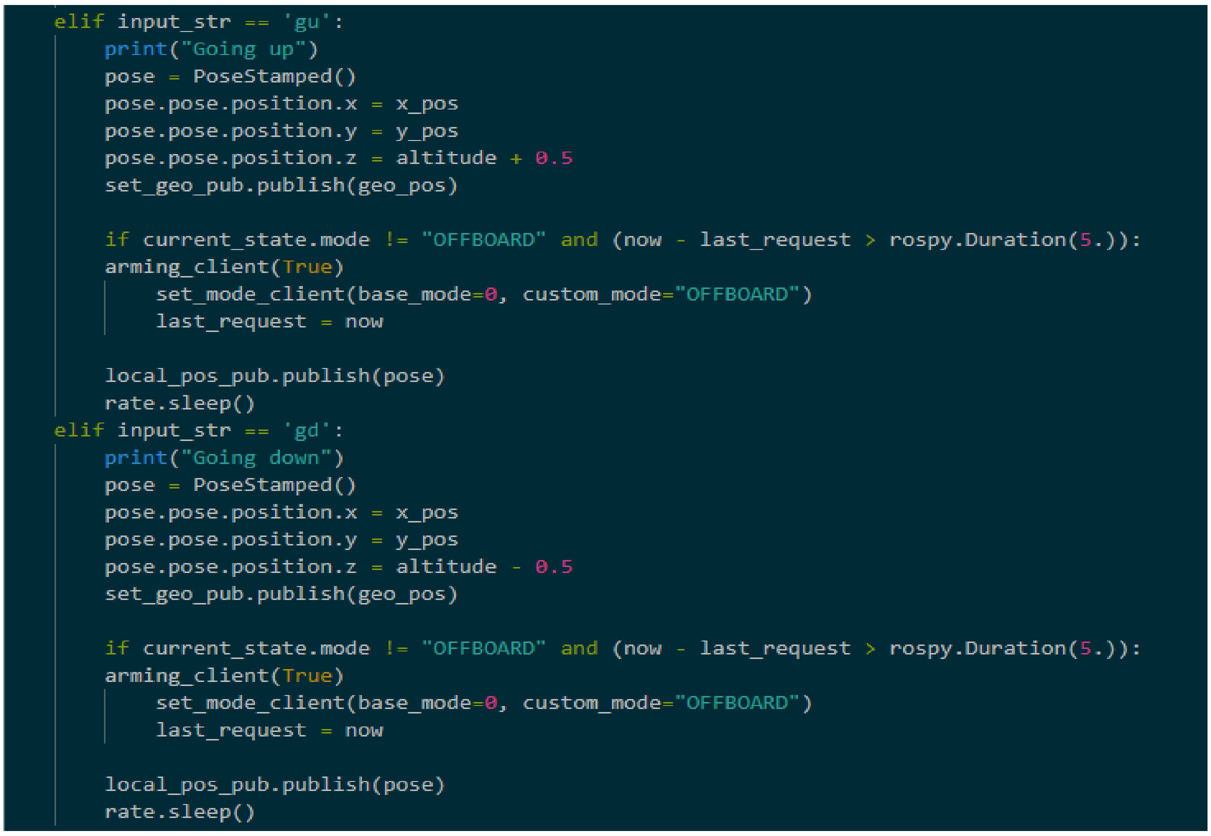
```
elif input_str == "f":
    print("Moving forward")
    pose = PoseStamped()
    pose.pose.position.x = x_pos + 1
    pose.pose.position.y = y_pos
    pose.pose.position.z = altitude
    set_geo_pub.publish(geo_pos)

    if current_state.mode != "OFFBOARD" and (now - last_request > rospy.Duration(5.)):
    arming_client(True)
        set_mode_client(base_mode=0, custom_mode="OFFBOARD")
        last_request = now

    local_pos_pub.publish(pose)
    rate.sleep()
elif input_str == 'b':
    print("Moving back")
    pose = PoseStamped()
    pose.pose.position.x = x_pos -1
    pose.pose.position.y = y_pos
    pose.pose.position.z = altitude
    set_geo_pub.publish(geo_pos)

    if current_state.mode != "OFFBOARD" and (now - last_request > rospy.Duration(5.)):
    arming_client(True)
        set_mode_client(base_mode=0, custom_mode="OFFBOARD")
        last_request = now

    local_pos_pub.publish(pose)
    rate.sleep()
```

Figure 3.22 Forward and back source code



```
elif input_str == 'sr':
    print("Sliding right")
    pose = PoseStamped()
    pose.pose.position.x = x_pos
    pose.pose.position.y = y_pos - 1
    pose.pose.position.z = altitude
    set_geo_pub.publish(geo_pos)

    if current_state.mode != "OFFBOARD" and (now - last_request > rospy.Duration(5.)):
    arming_client(True)
        set_mode_client(base_mode=0, custom_mode="OFFBOARD")
        last_request = now

    local_pos_pub.publish(pose)
    rate.sleep()
elif input_str == 'sl':
    print("Sliding left")
    pose = PoseStamped()
    pose.pose.position.x = x_pos
    pose.pose.position.y = y_pos +1
    pose.pose.position.z = altitude
    set_geo_pub.publish(geo_pos)

    if current_state.mode != "OFFBOARD" and (now - last_request > rospy.Duration(5.)):
    arming_client(True)
        set_mode_client(base_mode=0, custom_mode="OFFBOARD")
        last_request = now

    local_pos_pub.publish(pose)
    rate.sleep()
```

Figure 3.23 Slide right and left source code



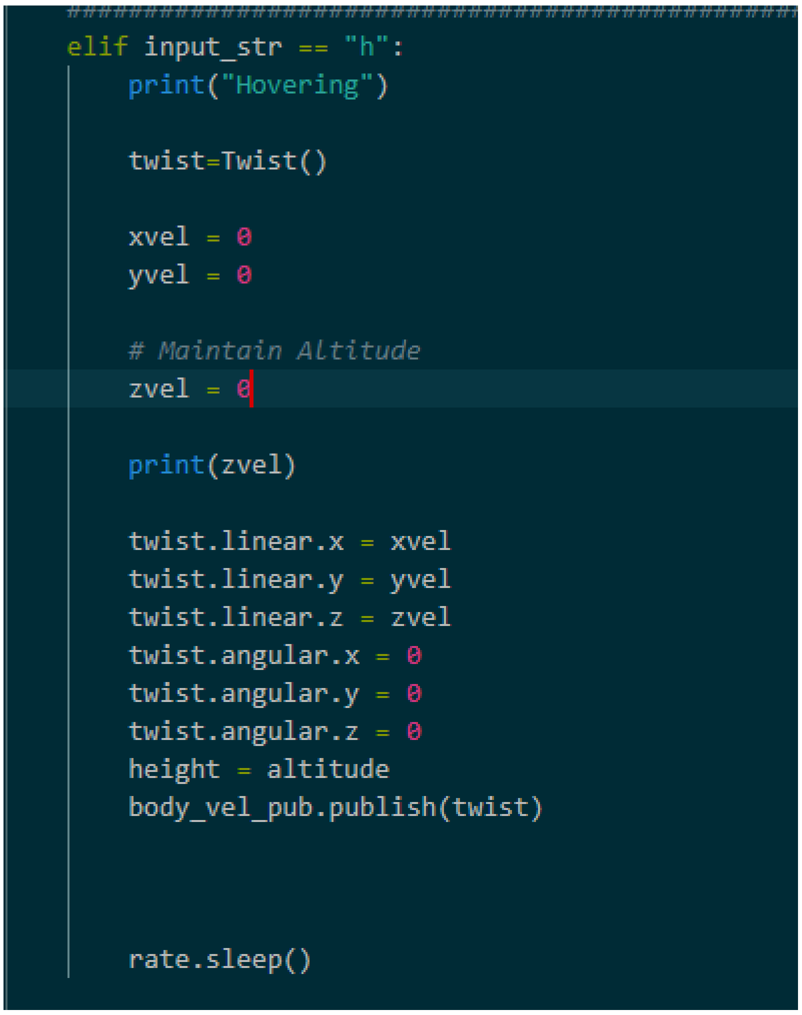
```python
    elif input_str == 'gu':
        print("Going up")
        pose = PoseStamped()
        pose.pose.position.x = x_pos
        pose.pose.position.y = y_pos
        pose.pose.position.z = altitude + 0.5
        set_geo_pub.publish(geo_pos)

        if current_state.mode != "OFFBOARD" and (now - last_request > rospy.Duration(5.)):
        arming_client(True)
            set_mode_client(base_mode=0, custom_mode="OFFBOARD")
            last_request = now

        local_pos_pub.publish(pose)
        rate.sleep()
    elif input_str == 'gd':
        print("Going down")
        pose = PoseStamped()
        pose.pose.position.x = x_pos
        pose.pose.position.y = y_pos
        pose.pose.position.z = altitude - 0.5
        set_geo_pub.publish(geo_pos)

        if current_state.mode != "OFFBOARD" and (now - last_request > rospy.Duration(5.)):
        arming_client(True)
            set_mode_client(base_mode=0, custom_mode="OFFBOARD")
            last_request = now

        local_pos_pub.publish(pose)
        rate.sleep()
```

Figure 3.24 Go up and down source code

As we can see, we once again used the /mavros/setpoint_position/pose topic variables. Hence if we want to slide the drone right or left we need to make the drone move according the y axis by changing the pose.pose.position.y value, similarly if we want to move the drone back and forth we need to make it move according to the x axis by changing pose.pose.position.x value and finally to move the drone up and down we move it according to the z axis by changing the pose.pose.position.z value.

- Once we moved the drone up/down slide left/right and forward/back, we need to stop it at that desired position, to do that we use the Hovering flight mode, where the user input is ‘h’, and the drone will hover in the air in the desired position.

The source code for the hover mode is in figure 3.25:



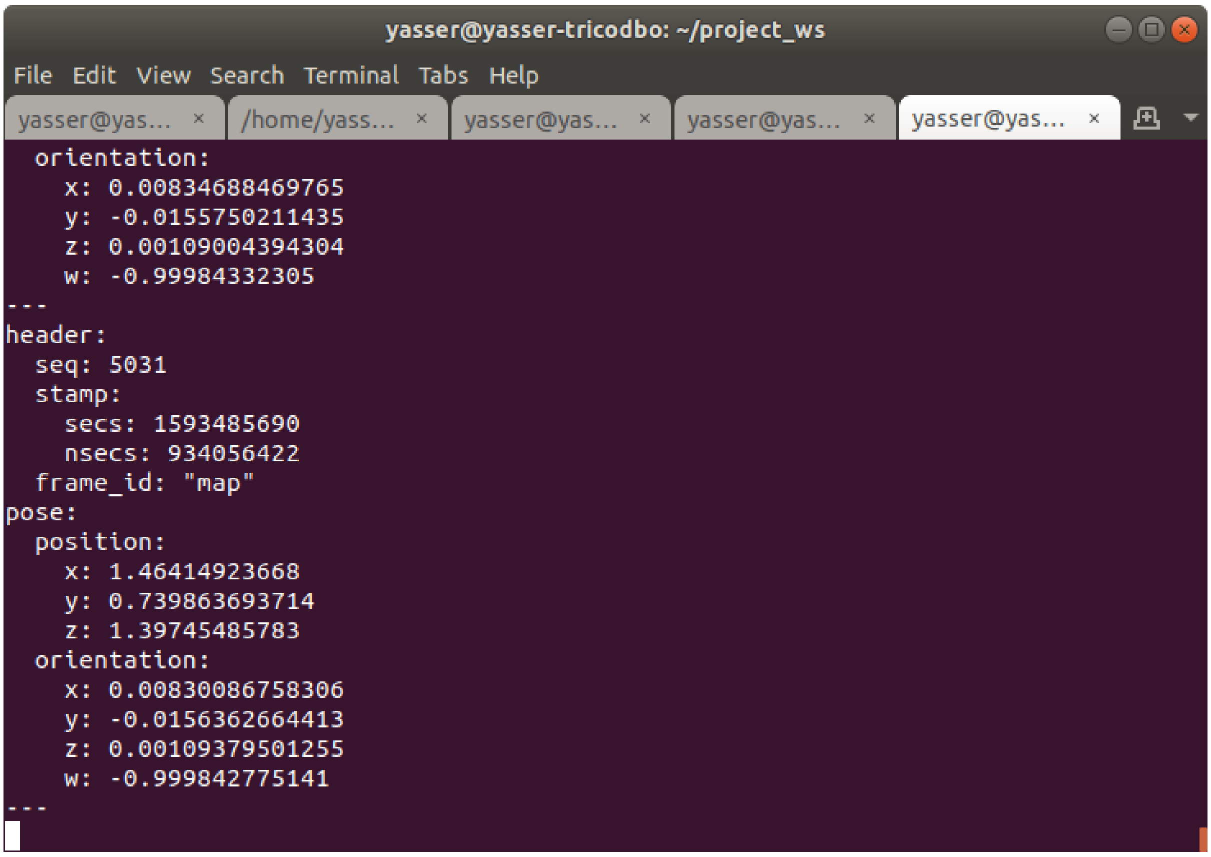
```
###########################################################
elif input_str == "h":
    print("Hovering")

    twist=Twist()

    xvel = 0
    yvel = 0

    # Maintain Altitude
    zvel = 0

    print(zvel)

    twist.linear.x = xvel
    twist.linear.y = yvel
    twist.linear.z = zvel
    twist.angular.x = 0
    twist.angular.y = 0
    twist.angular.z = 0
    height = altitude
    body_vel_pub.publish(twist)



    rate.sleep()
```

Figure 3.25 Hovering mode source code

In this mode, we use the velocity variables in the /mavros/setpoint_velocity topic, this topic has six variables: twist.linear.x, twist.linear.y, twist.linear.z, whereas these variables will give the drone velocity values to move the drone according the specified axis, and twist.angular.x, twist.angular.y, twist.angular.z which will make the drone rotate around itself according to the specified axis.

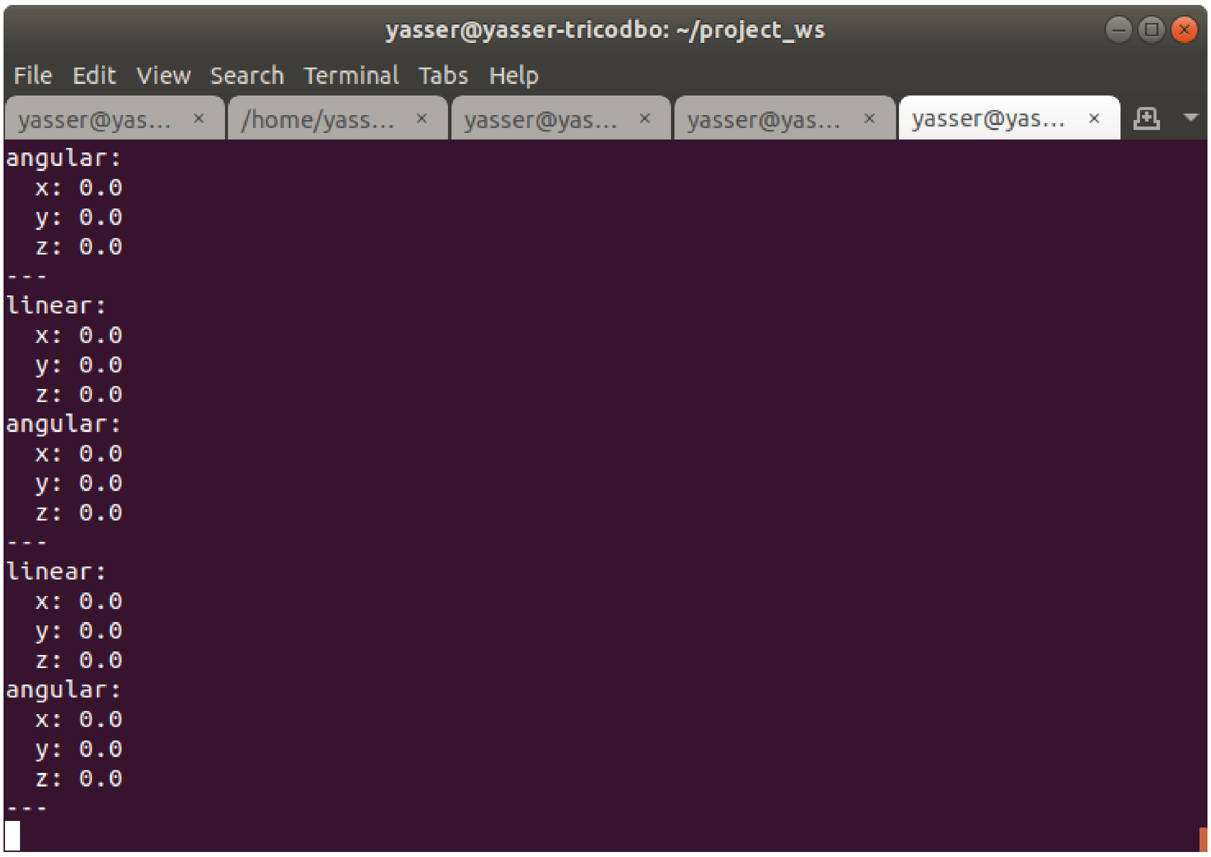
By setting these variables to 0, the drone will stop moving while maintaining its current altitude.

The values of these different variables are shown in figure 3.21, using the command: rostopic echo /mavros/setpoint_velocity and rostopic echo /mavros/setpoint_position/pose.


yasser@yasser-tricodbo: ~/project_ws
File Edit View Search Terminal Tabs Help
yasser@yas... /home/yass... yasser@yas... yasser@yas... yasser@yas...

```
  orientation:
    x: 0.00834688469765
    y: -0.0155750211435
    z: 0.00109004394304
    w: -0.99984332305
---
header:
  seq: 5031
  stamp:
    secs: 1593485690
    nsecs: 934056422
  frame_id: "map"
pose:
  position:
    x: 1.46414923668
    y: 0.739863693714
    z: 1.39745485783
  orientation:
    x: 0.00830086758306
    y: -0.0156362664413
    z: 0.00109379501255
    w: -0.999842775141
---
```

Figure 3.26 Position and orientation coordinates in hover mode


yasser@yasser-tricodbo: ~/project_ws
File Edit View Search Terminal Tabs Help
yasser@yas... /home/yass... yasser@yas... yasser@yas... yasser@yas...

```
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
---
```

Figure 3.27 Linear and angular velocities in hover mode

For the tracking flight modes, we will be discussing them in what follows.

3.3. Images classification:

3.3.1. Introduction:

As mentioned earlier, a tracking flight mode exists (‘t’ as a user input), the camera of the drone will spot the cable from above, and changes its directions accordingly. The system that does image classification is a trained neural network built using TensorFlow and Keras platforms. The advantage of these platform is to avoid building a complicated powerful neural network from scratch, for the reason that they provide useful and powerful modules for that matter.

3.3.2. TensorFlow:

TensorFlow, in the most general terms, is a software framework for numerical computations based on dataflow graphs. It is designed primarily, however, as an interface for expressing and implementing machine learning algorithms, chief among the deep neural networks.

TensorFlow was designed with portability in mind, enabling these computation graphs to be executed across a wide variety of environments and hardware platforms. With essentially identical code, the same TensorFlow neural net could, for instance, be trained in the cloud, distributed over a cluster of many machines or on a single laptop. It can be deployed for serving predictions on a dedicated server or on mobile device platforms such as Android or iOS, or Raspberry Pi single-board computers.[8]

TensorFlow is also compatible, of course, with Linux, macOS, and Windows operating systems.

The core of TensorFlow is in C++, and it has two primary high-level frontend languages and interfaces for expressing and executing the computation graphs. The most developed frontend is in Python, used by most researchers and data scientists. The C++ frontend provides quite a low-level API, useful for efficient execution in embedded systems and other scenarios.

3.3.3. Keras:

Keras is one of the most popular and powerful TensorFlow extension libraries. Among the extensions we survey in this chapter, Keras is the only one that sup ports both Theano—upon which it was originally built—and TensorFlow. This is possible because of Keras’s complete abstraction of its backend; Keras has its own graph data structure for handling computational graphs and communicating with TensorFlow.

In fact, because of that it could even be possible to define a Keras model with either TensorFlow or Theano and then switch to the other.

Keras has two main types of models to work with: sequential and functional. The sequential type is designed for simple architectures, where we just want to stack layers in a linear fashion. The functional API can support more-general models with a diverse layer structure, such as multioutput models.[9]

3.3.4. Code:

The source codes to train the neural network system, as well as the source code to start image classification can be found in the appendix C document.

To start the image classification program, we use the following command:

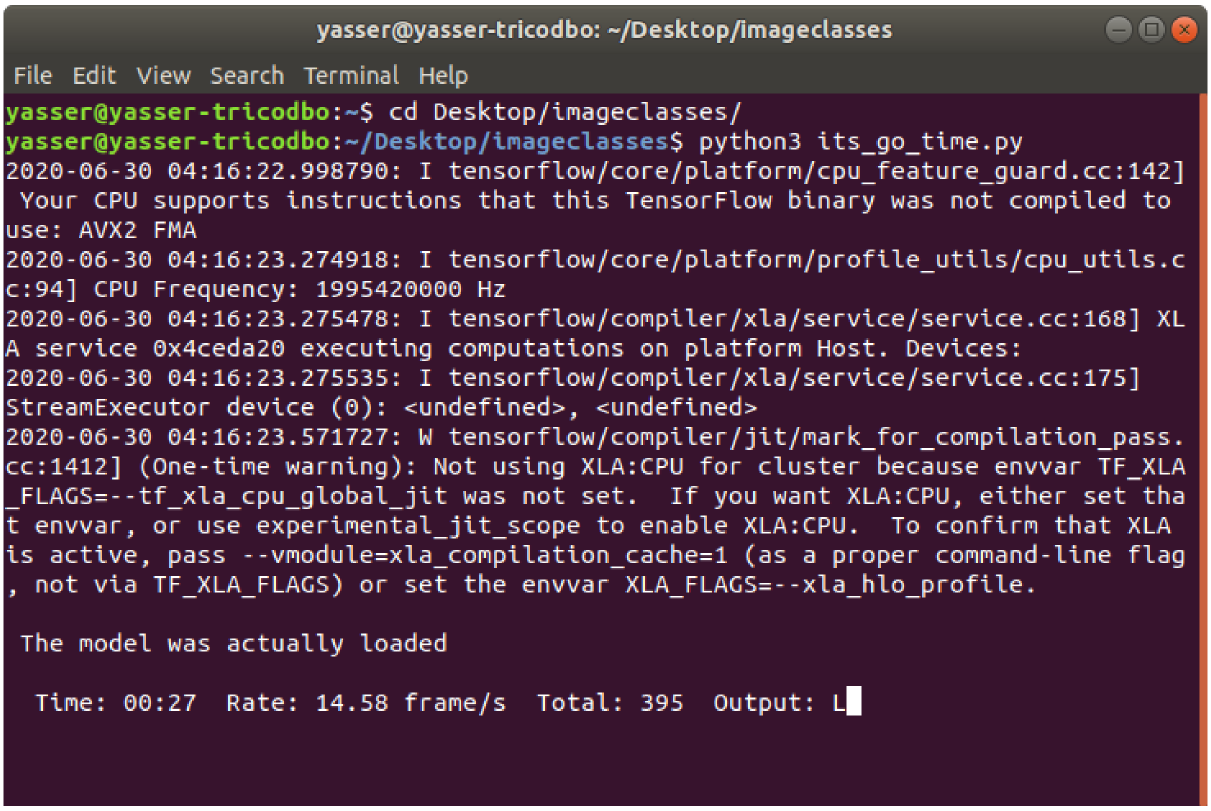
```
python3 imageclassification.py
```

The program will open the camera of the computer running the simulation, the frames captured by the camera will be scaled and filtered according to the resolution and the dataset used when training the model.

The output of the program is a string (R for right, L for left, S for straight, and U for unknown image), this output will be sent to the drone control node through pyperclip module which is a python module to copy and paste data between different platform.

The drone control node will receive the data from the image classification program, and when choosing tracking flight mode, the node will send message to MAVROS topics to control the drone accordingly.

Figure 3.28 shows the output of the program:


yasser@yasser-tricodbo: ~/Desktop/imageclasses
File Edit View Search Terminal Help

```
yasser@yasser-tricodbo:~$ cd Desktop/imageclasses/
yasser@yasser-tricodbo:~/Desktop/imageclasses$ python3 its_go_time.py
2020-06-30 04:16:22.998790: I tensorflow/core/platform/cpu_feature_guard.cc:142]
 Your CPU supports instructions that this TensorFlow binary was not compiled to
use: AVX2 FMA
2020-06-30 04:16:23.274918: I tensorflow/core/platform/profile_utils/cpu_utils.c
c:94] CPU Frequency: 1995420000 Hz
2020-06-30 04:16:23.275478: I tensorflow/compiler/xla/service/service.cc:168] XL
A service 0x4ceda20 executing computations on platform Host. Devices:
2020-06-30 04:16:23.275535: I tensorflow/compiler/xla/service/service.cc:175]
StreamExecutor device (0): <undefined>, <undefined>
2020-06-30 04:16:23.571727: W tensorflow/compiler/jit/mark_for_compilation_pass.
cc:1412] (One-time warning): Not using XLA:CPU for cluster because envvar TF_XLA
_FLAGS=--tf_xla_cpu_global_jit was not set.  If you want XLA:CPU, either set tha
t envvar, or use experimental_jit_scope to enable XLA:CPU.  To confirm that XLA
is active, pass --vmodule=xla_compilation_cache=1 (as a proper command-line flag
, not via TF_XLA_FLAGS) or set the envvar XLA_FLAGS=--xla_hlo_profile.

 The model was actually loaded

  Time: 00:27  Rate: 14.58 frame/s  Total: 395  Output: L
```

Figure 3.28 Image classification output

3.3.4.1. Mechanical equations to adjust the drone's speed:

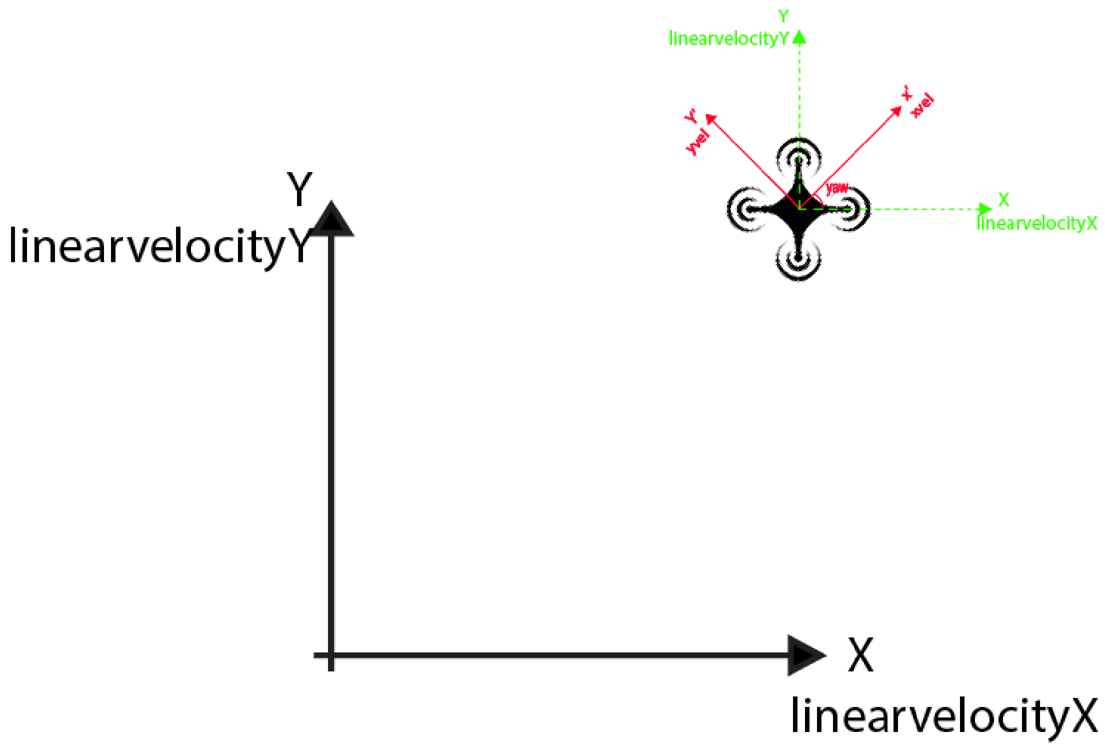


Figure 3.29 The axes systems

By a simple projection we can say:

$linearvelocityX1 = Xvel.\cos(yaw)$ Equation 3.1

$linearvelocityY1 = Xvel.\sin(yaw)$ Equation 3.2

And

$linearvelocityX2 = Yvel.\sin(yaw)$ Equation 3.3

$linearvelocityY2 = Yvel.\cos(yaw)$ Equation 3.4

So

$linearvelocityX = linearvelocityX1 + linearvelocityX2$ Equation 3.5

$linearvelocityY = linearvelocityY1 + linearvelocityY2$ Equation 3.6

Yaw value can be obtained by using the equation 2.17.

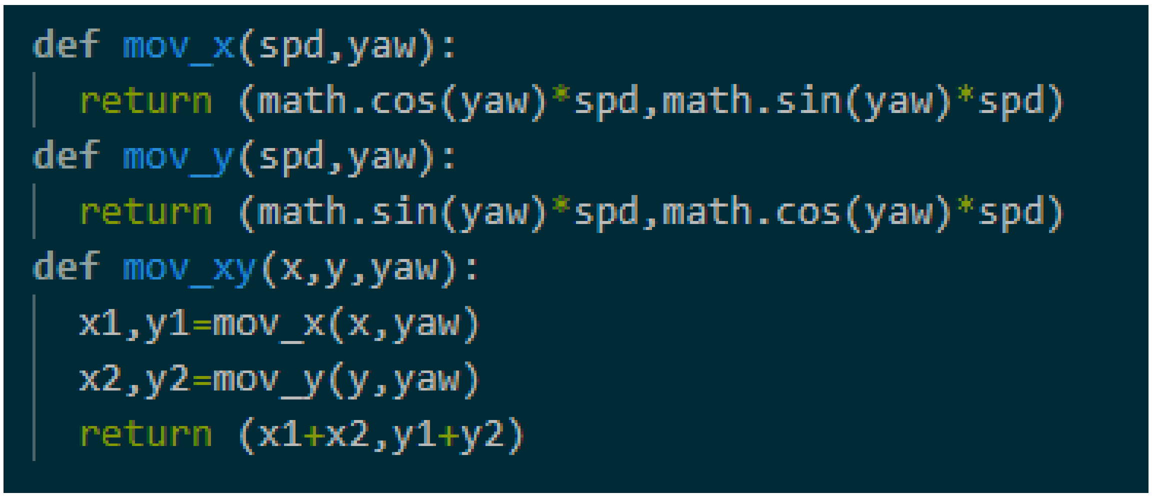
3.3.4.2. Source Code:

To adjust the drone’s movement for image classification program output we will use the /mavros/setpoint_velocity topic, this latter, as we mentioned earlier, has six variables: linear velocities (xyz) and angular velocities (xyz). In our case we only need the linear velocities xy and the angular velocity z.

The linear velocities xy go with the fixed axis xy, while the angular velocity z goes with the perpendicular axis z’ on the drone’s body.

Hence, to adjust the drone’s velocity according to the relative axis x’y’z’, we need to convert them to the fixed axis xyz using the equation 3.5 and 3.6.

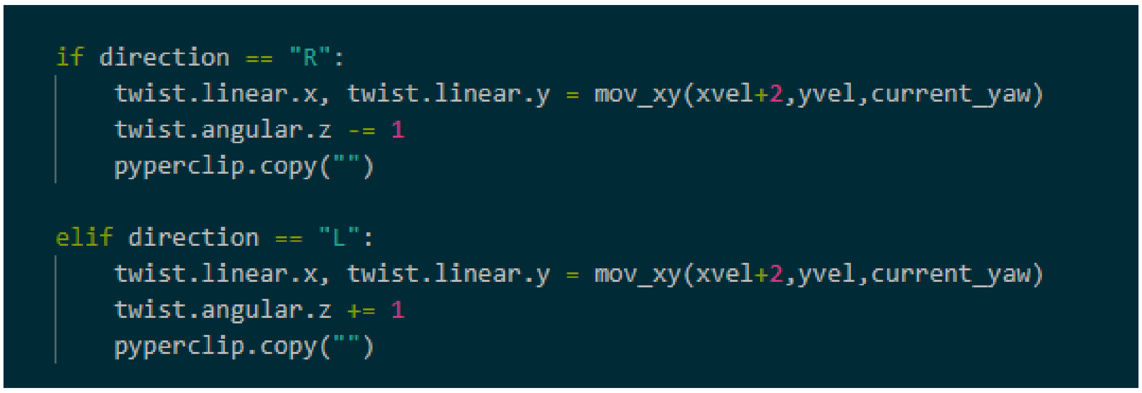
The part of the code that does that is in figure 3.30:



```python
def mov_x(spd,yaw):
    return (math.cos(yaw)*spd,math.sin(yaw)*spd)
def mov_y(spd,yaw):
    return (math.sin(yaw)*spd,math.cos(yaw)*spd)
def mov_xy(x,y,yaw):
    x1,y1=mov_x(x,yaw)
    x2,y2=mov_y(y,yaw)
    return (x1+x2,y1+y2)
```

Figure 3.30 functions to calculate the velocity according to the fixed axis xyz

- The image classification output is ‘R’ or ‘L:

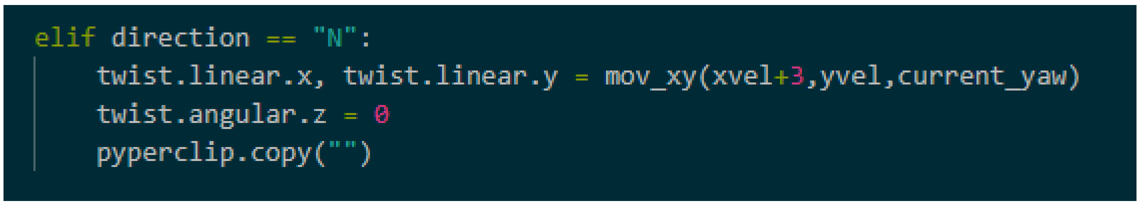


```python
if direction == "R":
    twist.linear.x, twist.linear.y = mov_xy(xvel+2,yvel,current_yaw)
    twist.angular.z -= 1
    pyperclip.copy("")

elif direction == "L":
    twist.linear.x, twist.linear.y = mov_xy(xvel+2,yvel,current_yaw)
    twist.angular.z += 1
    pyperclip.copy("")
```

Figure 3.31 source code for left and right directions

As we can see, we're adjusting the xvel (to keep the drone moving forward, and the twist.angular.z variable to make the drone rotate according to the direction.

- The image classification output is 'N':



```
elif direction == "N":
    twist.linear.x, twist.linear.y = mov_xy(xvel+3,yvel,current_yaw)
    twist.angular.z = 0
    pyperclip.copy("")
```

Figure 3.32 source code for neutre direction

In this case we only adjusted the xvel value to keep the drone moving forward only.

4. Conclusion:

Throughout this chapter, we could see the benefits of using the simulation Gazebo to run the PX4 firmware, for the reason that it allowed us to access the different variables of the drone, as well as being able to test the codes inside the simulation environments before attempting to test in reality.

This chapter also showed the power of the ROS-MAVLink (MAVROS) combination, for that it provides us with a powerful and reliable communication protocol between the PX4 firmware, the simulator and the API used to control the drone (ROS).

Chapter 4 : Launching the prototype

1 Introduction:

After the promising result of the SITL simulation, it is now time to test the programs on a real drone, hence in this final chapter we will discuss the different components that were used to build up the drone (the body airframe, controllers…) as well the necessary configuration and tools to launch the drone which includes calibrating the Pixhawk sensors as well us setting the Raspberry Pi to be the onboard computer.

2 Identifying the components:

The four types of components we're going to be interfacing with in the drone are
the Electronic Speed Controller (ESC), the motors, airframe, battery.[1]

2.1 The Electronic speed controller (ESC):

The ESC is also easily identified as it is where the main power lead (plug for the battery) is attached. It regulates the power from the battery to the motors, and thus controls the speed of the motor. Using good quality ESC's means you should have a reliable and smooth flight experience, though of course, there are other factors to consider which are:

- Current Rating – Amperage

The first thing to look at when choosing ESC is the current rating, which is measured in Amps. Motors draw current when they spin, if we draw more Amps than your ESC can handle, it will start to overheat and eventually fail.[12] There are three things that tend to increase our current draw and put more stress on our ESC:

  - Higher motor KV
  - Larger motor size (stator width and height)
  - Heavier propellers (length and pitch)

- Battery limitations

When you draw current from LiPo battery, the voltage sags due to internal resistance. When you reach the discharge limit of the battery, the voltage would sag so much, it can no longer sustain the high current draw.

It's completely okay to use larger ESC's than required, downsides are the extra weight, size and cost. In fact, there are advantages with higher current ESC's, which are the lower chance of overheat and higher efficiency.

On an ESC, there are MOSFET (or FET) that basically do all the hard work handling high current. The FET's are bigger and beefier on higher current ESC's, and they don't generate as much heat as the smaller ones, therefore they can be more energy efficient.

2.1.1 Connection of the ESC:

ESC is powered directly by LiPo battery, and motor speed is controlled by a signal from the flight controller.

The motors are connected to the ESC through 3 wires. The wire order doesn't actually matter. If the motor spins the wrong direction, simply swap any two wires.

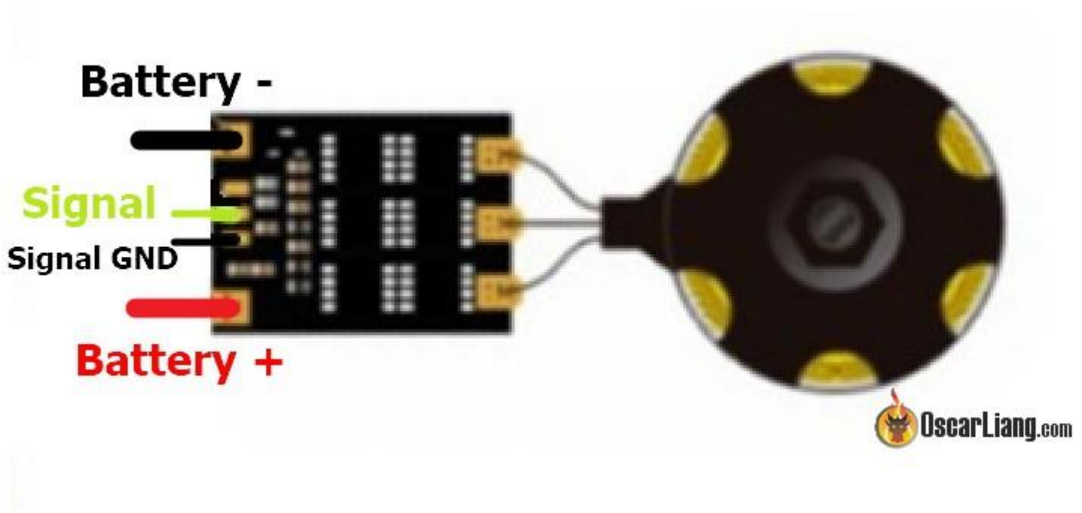


Figure 4.1 Connection of the ESC

2.1.2 Anatomy of the ESC:

An ESC is made up of the following components:

- Micro Controller
- Gate Drivers
- MOSFET
- LDO

- Arrays of filtering capacitors
- Optional: Current Sensor
- Optional: LED

Whereas:

- LDO

These are voltage regulators for converting voltage down to power the micro controller and other components.

- Micro Controller

Micro controller, or MCU, is the brain of an ESC.

- Gate Driver

Gate drivers are used to drive the MOSFET's in our ESC, and actually bring benefits to the performance. It's connected to the gate of a MOSFET hence the name "gate driver".

- MOSFET

MOSFET are like switches, it switches the power on and off thousands of times per second, this is how the motors are driven.

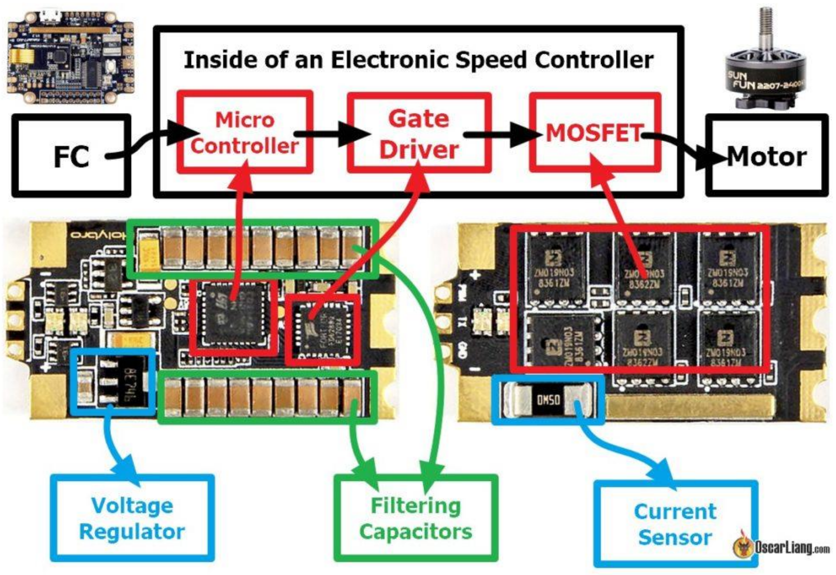


Figure 4.2 Block diagram of the different components inside the ESC

2.2 Battery:

A LiPo battery powers all of the electronics, and motors on your drone. The difference with LiPo batteries and the ones you would use in your TV remote is the chemicals used in the battery. LiPo batteries are based on Lithium Polymer chemistry (hence the name LiPo) which allow these batteries to have a very high energy density compared to other types of batteries. A battery with a higher energy density will be able to hold more energy compared to another battery of the same weight which is why LiPo batteries are commonly used with R/C aircraft and drones.

A LiPo battery is constructed from individual cells, where each cell consists of some metal and chemicals packaged together to generate an electrical charge. By connecting these cells together in various ways, we are able to make different LiPo batteries with various voltages, and capacities.[11]

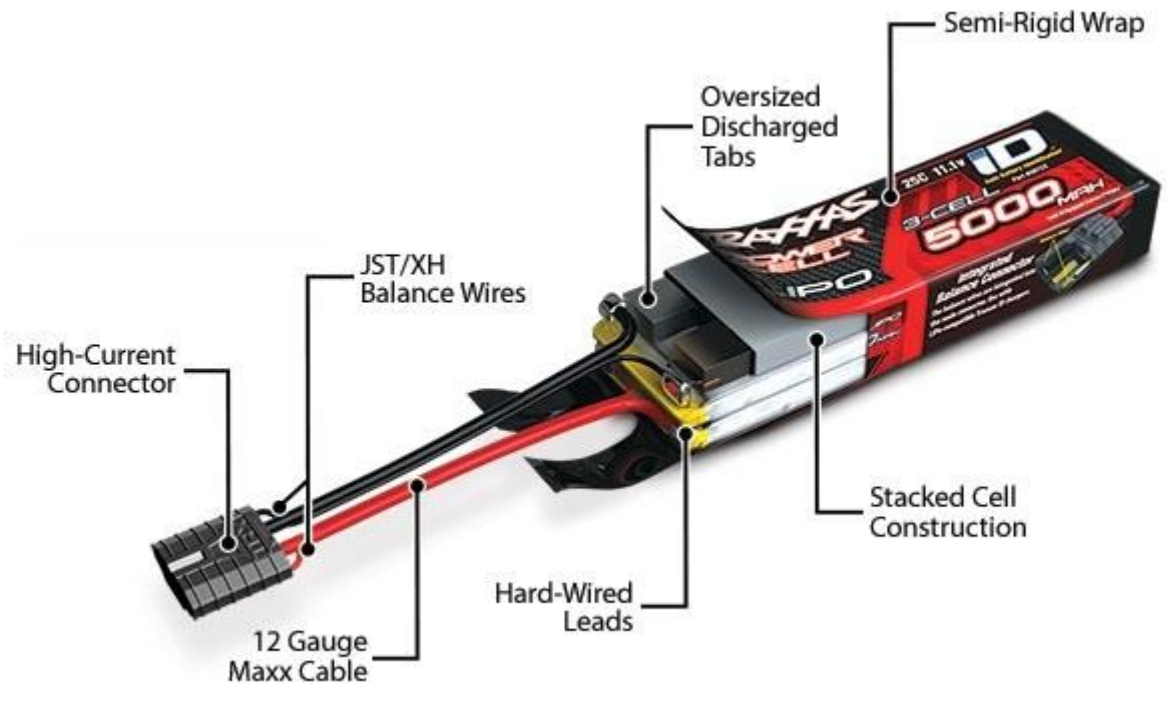


Figure 4.3 Inside a LiPo battery

In what follows we find the meaning of the numbers in a LiPo Battery:

2.2.1 Cells:

A battery is constructed from rectangular cells which are connected together to form the battery. A cell which can be considered a battery in itself, holds a nominal voltage of 3.6V. By connecting more of these in series, the voltage can increase to 7.2V for a 2-cell battery, 11.1V for a 3-cell battery and so on. By connecting more batteries in parallel the capacity can be increased. Often we will see numbers like 3S2P, which mean the battery as 3 cells (3S) connected in series, and there are 2 cell sets connected in parallel (2P) , giving a total number of 6 individual cells in the battery.

So, the number of cells is what defines the voltage of the battery. Having a higher voltage means the battery can provide more power to drive bigger motors, however more power does not necessarily mean the battery will provide energy for longer, that is defined by the battery capacity.

2.2.2 Capacity:

The capacity of a battery is a representation of how long it can provide energy for, often quoted in mili Amp hours, (mAH). The bigger this number, the more capacity the battery has, so it can run your motors for longer. However, the higher the capacity of a battery, the heavier it is so there is always a tradeoff between the battery capacity and weight for your aircraft.

2.2.3 Discharge Rate:

The discharge rate is a very important specification to check when buying a battery. This number, also known as the battery C rating (or continuous C rating) defined how fast we can extract the energy from your battery. If your motors draw more energy that what battery we can provide, we can possibly damage your battery which can result in we crashing our drone.

To work out the actual current in Amps we simply multiply the capacity by the C value. So, a 2200mAh battery with a C rating of 25C would have a continuous current output of 25 x 2.2 = 55A continuous current output.

In this project, we used a 5200 mAh, with a rating of 50C, with a 3 cells battery (3.7 V output per cell) which an output of 11.1 V

2.3 Motors:

2.3.1 Introduction:

There are a few types of motors that are used to build drones. But as the drone needs to be thrust in the air to float, we should use some powerful motors. The cheap, lightweight, small, and powerful motors used in drones are Brushless DC motors **(**BLDC). As the name implies, a brushless drone motor lacks the brushes. The brushless motor can be effectively divided into two separate components; the rotor and the stator. The stator is the central unit into which the rotor is mounted. The stator is made up of a network of radial electromagnets that alternatively power on and off to produce a temporary magnetic field when a current is passed through the windings. The rotor holds a collection of permanent magnets which are positioned in close proximity to the semi-permanent stator electromagnets. Attractive and repulsive interaction of the stator and rotor magnets is translated into rotational movement. When assembled, the shaft of the rotor is inserted into a pair of ball bearings located in the stator that maintain linear, smooth revolution of the rotor.[10]

Figure 4.4 brushless DC motor

2.3.2 Motor Sizing and Identification:

The size of a brushless motor is identified by a four-digit code that details the dimensions of the stator in millimeters, for example: 2206. The first two numbers in the series determine the diameter of the stator, in this case, 22mm. The final two describe the height of the stator, the last two numbers in this series are “06” therefore the stator unit is 6mm tall. It is important to remember that these numbers do not describe the external dimensions of the brushless motor itself.

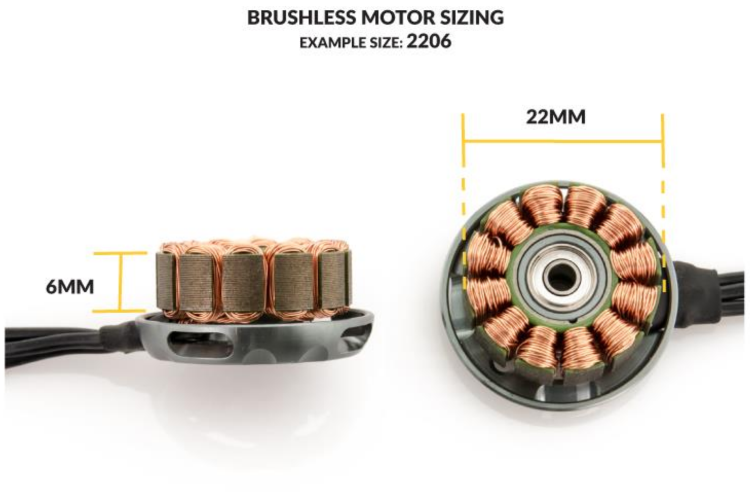


Figure 4.5 Brushless motor sizing

2.3.3 Mounting Patterns and Thread Size:

Mounting patterns and thread sizing is dependent on the type of motor and its application. The mounting pattern defines the positioning of the threaded bolt holes on the base of the motor. Each number describes the diameter of a circle with its center placed in the middle of the motor shaft. Usually, four holes are placed along the circumference of the circle, if two numbers are given, two holes are placed on each circle. For example, a 2205 with 16×19 spacing will have four M3 size threaded holes distributed evenly on both the circumference of the 16mm circle and 19mm circle. The dimensions of the threaded shaft are given by an ISO screw thread rating, which describes the outer diameter of the shaft.

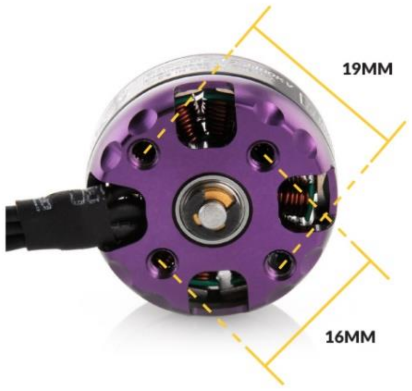


Figure 4.6 The threaded bolt holes

2.3.4 Velocity Constant — How fast a Motor Spins:

kV=RPM per 1 Volt

k = The kV rating of the motor e.g. 2300

V = Voltage input e.g. 16.8v

Example: 2300(kV rating) X 16.8(Voltage) = 38,640(Revolutions Per Minute)

The velocity constant (kV) determines how many rotations a motor can make within a minute without a load (no propeller) and at a constant current of 1 Volt. Simply, kV is a representation of how fast the motor can potentially spin. The kV of a motor is defined by the strength of the magnetic field at the stator and the amount of turns in the windings. A motor with a lower kV is best suited for efficiently driving heavy propellers. A high kV motor is optimized for lightweight propellers.

2.3.5 Thrust:

Thrust is one of the key factors to consider when choosing a motor. The thrust output of a motor is usually measured in grams and varies depending on how fast the motor is spinning and the propeller that it is rotating. Before a multicopter can begin to accelerate, a certain amount of thrust is required to overcome drag, as well as the pull of gravity.

2.3.6 Weight and Drone Motor Momentum:

When selecting a motor, it’s not all about thrust numbers. The weight of the motor should also be considered, as it has a significant impact on the flight characteristics of the multicopter. Due to the moment of inertia, a heavier motor will be more resistant to changes in acceleration than a lighter motor. The primary issue with a heavy multicopter motor being resistant of acceleration is that it will provide inaccurate flight characteristics and poor responsiveness once in the air. If maneuverability is a priority, a lightweight motor is an exemplary choice. On the other hand, an application in which maximum all-out speed is a must; larger motors will be able to provide the higher thrust numbers that are required.

2.3.7 Drone Motor Response Time:

Torque is a measurement of how quickly a motor can reach a certain RPM, directly affecting the responsiveness of a motor. Torque allows a multicopter to briskly maneuver through flips and rolls, additionally improving the accuracy of these movements. The amount of torque a motor can output also influences propeller selection. Heavier props will require more torque to accelerate than lighter props. The best gauge for motor torque is the dimensions of the stator. Larger stators tend to be capable of producing greater torque. Although, a larger stator will increase the total weight of the motor.

2.3.8 Drone Motor Efficiency:

Motor efficiency is a balancing act, requiring an equilibrium to be struck between the electrical power entering the motor and the mechanical power being produced by the motor as it spins. The importance of motor efficiency varies based on the situation. If high speed is prioritized, short flight times are often seen to be acceptable; quadcopter races may only last for two minutes! In the contrary, long-range multicopters require maximum efficiency to achieve longer flight times, increasing the distance that can be travelled.

In this project we used an A2212/19T 1000KV model.

2.2 Airframe:

The airframe consists of two parts which are:

- **The booms:** Shorter booms increase maneuverability, while longer booms increase stability. Booms must be tough to hold up in a crash, while interfering with prop downdraft as little as possible. In many drones, the boom is part of the main body. Other drones have a definite boom as a separate part.
- **Main Drone Body Part:** This is the central hub from which booms radiate like spokes on a wheel. It houses battery, main boards, processors avionics, cameras, and sensors.
- **Landing Gear:** Drones, which need high ground clearance may adopt helicopter style skids mounted directly to the body, while other drones which have no hanging payload may omit landing gear altogether.

In this project we used the F450 Drone model which is presented in following figure:

F450 drone kit model

3. Setting up the Raspberry Pi:

As we saw in the previous chapter, with used a computer to run ROS programs to control the drone, now that we are running practical tests, we cannot use a big computer, hence using the Raspberry Pi as a companion computer (will be mounted on the drone) will be a good choice.

But before that, the Pi needs configuration, which will be mentioned in the following steps:

The Raspberry Pi is best characterized by the funky and useful Raspbian operating system. As the name suggests, Raspbian is designed specifically for the Raspberry Pi and is suitable for most applications. However, if you want to use your Raspberry Pi for desktop computing, you might be disappointed. Raspbian is based on Debian, a Linux operating system designed for stability. Updates take place only once every few years meaning you won't have the latest version of programs and their features. Fortunately, there's a way to bring the popular Ubuntu desktop operating system to your Raspberry Pi. Ubuntu MATE uses an extremely lightweight desktop environment and there's even a version designed specifically for your Raspberry Pi's ARM architecture. we need to install UBUNTU linux distribution, because ROS needs this distribution so that it will be able to work[11]. To install it, we can find instructions in the following website:

https://www.techradar.com/how-to/how-to-install-ubuntu-on-the-raspberry-pi

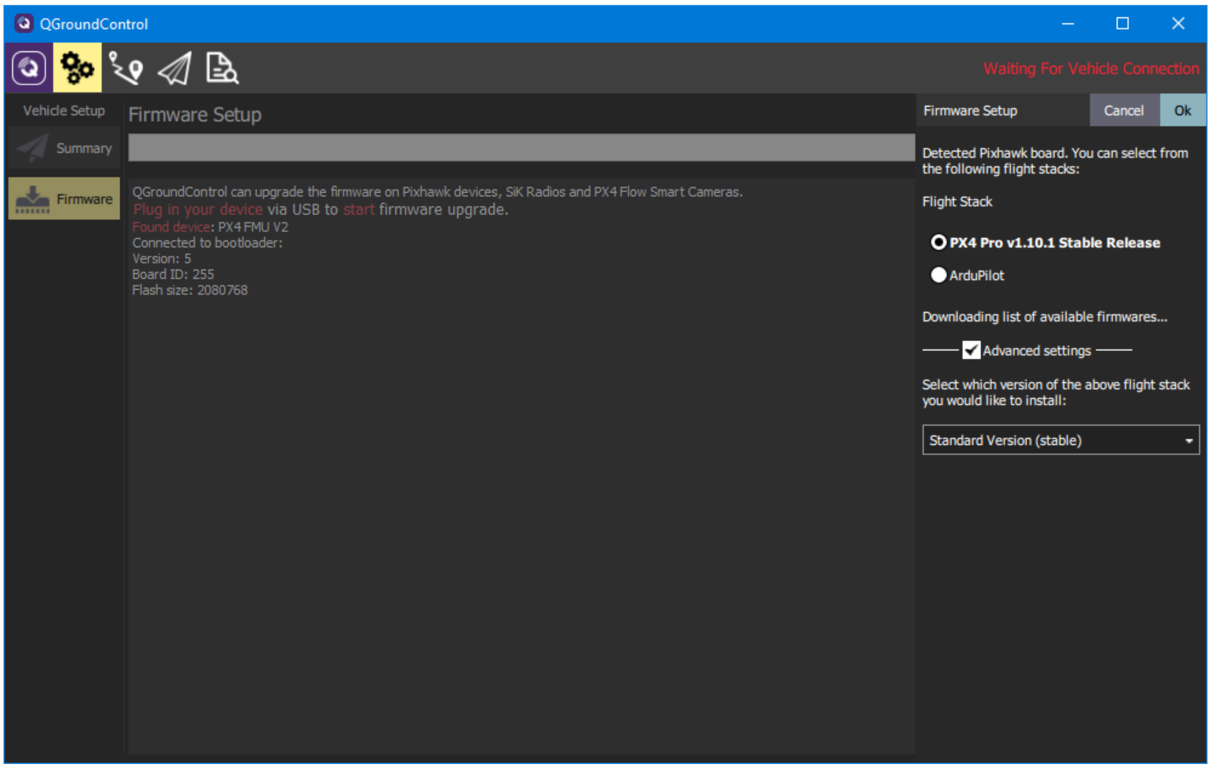
The next step is to install Robotics Operating System ROS on the UBUNTU mate distribution, to do this we can find instructions in the official ROS website:

http://wiki.ros.org/ROSberryPi/Installing%20ROS%20Kinetic%20on%20the%20Raspberry%20Pi

The last and final step is to create a ROS package for our programs, the same way we did in chapter3 (SITL Simulation).

4. Configuring the PIXHAWK:

We open the QGROUNDCONTROL and plug in the PIXHAWK through USB cable. Installing the compatible version of PX4 (in our case PX4 Pro v1.10 stable release).



QGroundControl can upgrade the firmware on Pixhawk devices, SiK Radios and PX4 Flow Smart Cameras.
Plug in your device via USB to start firmware upgrade.
Found device: PX4 FMU V2
Connected to bootloader:
Version: 5
Board ID: 255
Flash size: 2080768

Detected Pixhawk board. You can select from the following flight stacks:

Flight Stack

[✓ PX4 Pro v1.10.1 Stable Release]

ArduPilot

Downloading list of available firmwares...

[✓ Advanced settings]

Select which version of the above flight stack you would like to install:

Standard Version (stable)

Figure 4.7 Flashing the PIXHAWK with the chosen PX4 version

Picking the compatible airframe according to the drone's model where the PIXHAWK will be mounted. In our case it is a Generic Quadcopter.

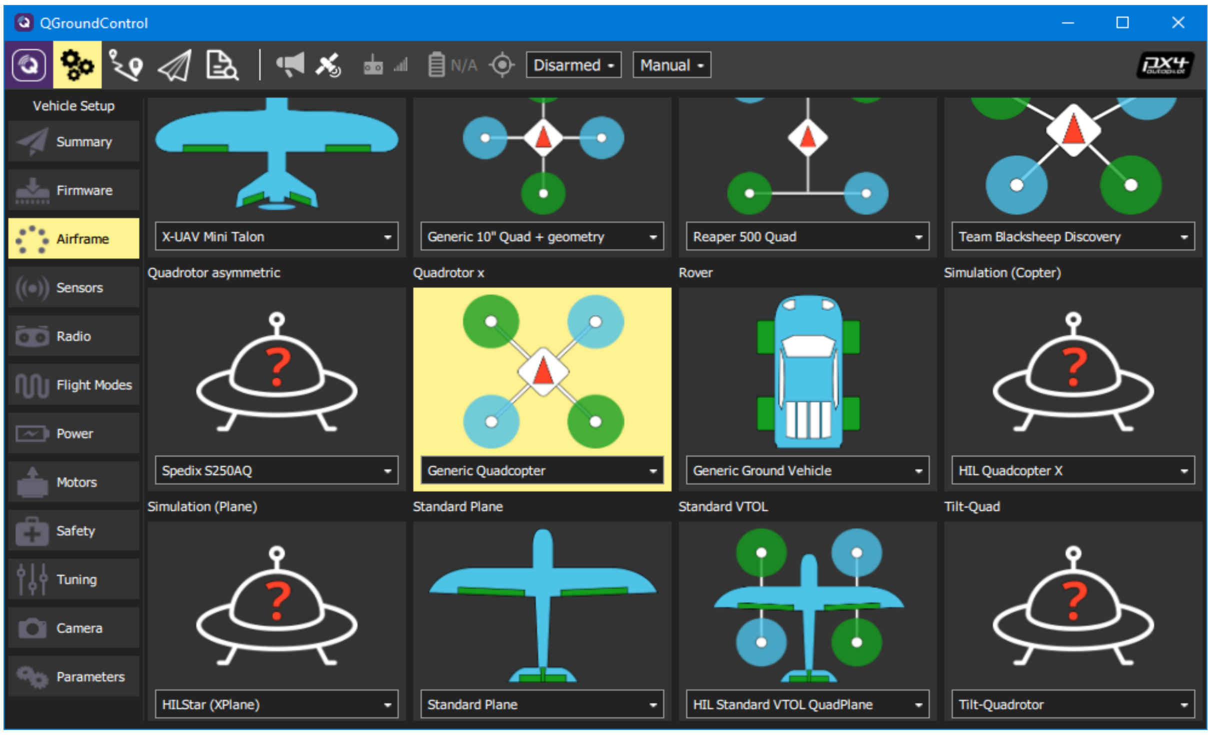
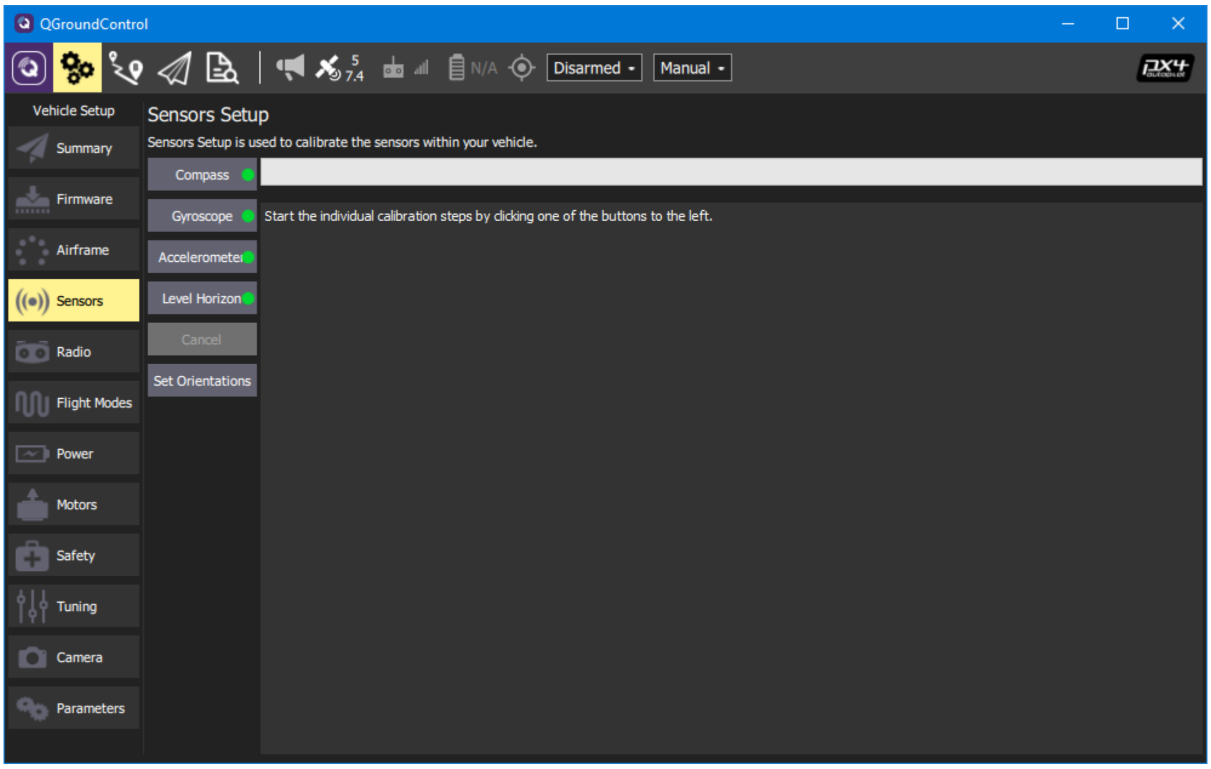


Figure 4.8 Airframe selection

Next step is to calibrate the sensors, there are different types of sensors to be calibrated (Compass, Gyroscope, Accelerometer, and level horizon), by clicking on each sensor, the different movements needed to calibrate the PIXHAWK will appear.



Sensors Setup is used to calibrate the sensors within your vehicle.

Start the individual calibration steps by clicking one of the buttons to the left.

Figure 4.9 Sensors calibration

5. Power up the drone:

By plugging the power cable of the battery, the PIXHAWK will be turned on and ready to receive the mission commands. The Raspberry Pi will be powered through its +5V and GND pins and the telemetry module port on the Pixhawk.

5.1 Connecting to the raspberry pi:

In order to do that we use the remote-control application that is implemented on the Windows operating system. A WIFI network is needed, so that when the Raspberry Pi powers up, it connects to it, and we on the other hand will connect to the same network, so that we will be able to access the Raspberry Pi through its IP address.

5.2 Launching the programs:

Similarly, to the simulation part, we will proceed with the same command, except this time, we do not need to run the PX4 SITL gazebo simulation, because the programs will be executed with PIXHAWK hardware itself.

Hence, first thing to do is to launch the MAVROS node to be able to communicate with the PIXHAWK

And second, we launch the drone control node followed by the user input node in the packages we created previously

Unfortunately, due to the reason that we did not have access to a Raspberry Pi camera, we cannot perform tracking mode and image classification in reality. In order for us to gain some time when launching the different programs, we created some abbreviation files, where series of commands are written in the file, and when we want to launch them, we just execute those files.

```
./aa file
```

```
cd ~/catkin_ws && source devel/setup.bash && roslaunch mavros px4.launch
fcu_url:=serial:///dev/ttyACM0:57600 gcs_url:=udp://@127.0.1.1
```

this file launches the MAVROS node to connect to the PIXHAWK.

```
./bb
```

```
cd ~/ project_ws && source devel/setup.bash && catkin_make && source devel/setup.bash
&& chmod +x src/pfe_pkg/src/scripts/drone_control.py && rosrun pfe_pfe drone_control.py
```

This file launches the drone control node.

```
./cc
```

```
cd ~/project_ws && source devel/setup.bash && catkin_make && source devel/setup.bash
&& chmod +x src/pfe_pkg/src/scripts/input_node.py && rosrun pfe_pkg input_node.py
```

To launch these files, we just need to type ./aa (or ./bb ./cc), each in a different terminal window.

6. Conclusion:

In this chapter we discussed the protocol to power up a drone, set its different components up and start driving it. We also could see the importance of the onboard computer (Raspberry Pi) for the reason of its small size, its ability to perform the ROS package's programs, and allowing us to access the drone from distance, control it and set autonomous missions. The only inconvenient that should be highlighted here is the Wi-Fi connection range, because if we lose the connection, we can't control the drone.

General Conclusion

In this project, we were able to build up an unmanned aerial vehicle UAV prototype for power lines monitoring. We can either control the drone manually or set the tracking mode where the camera implemented on the drone will analyze the images captured, classifies them and send the appropriate direction to the drone's program to move it accordingly.

We can say that this project can be divided into two major parts:

The first part concerns the autopilot used (PIXHAWK as a hardware and PX4 as a software installed in the PIXHAWK), where we could see its accuracy and the freedom it gives to the user to control the drone as needed, without forgetting the MAVLink communication protocol that provides us with a powerful communication link between the PX4 and other parts of the system.

The second major part is the Robotics Operating System, this API provided us with a great flexibility and powerful tools to be able to build up codes depending on the missions and the services needed by the drone.

Such a combination gives a great advantage to the drone, which is autonomy. The drone can be set up and put manually above a line then start the tracking mode so that it starts following the power lines for a certain specified period of time, while of course saving the mission's video to be analyzed by power lines inspectors to detect different faults in power lines, and when the mission's time is over, the drone will be automatically set to "RETURN TO LAUNCH" mode, which, as the name suggests, makes the drone go back to its launching point. All this without needing to be connected to it.

The major inconvenient that this prototype has, is the battery's duration. Running long period missions will be impossible with one battery charge, but researches are being carried out now to build capacitive power wireless transfer, so that when the drone's battery is low, it simply approaches the line and charges itself again to keep the mission going.

Implementing such a system will provide engineers and especially power lines inspectors with such a great tool to perform their job more effectively and avoid the different difficulties they face nowadays.

At the end, we can say that this project allowed us to discover, even a little bit, the world of robotics and autonomous vehicles, as well as using our skills and courses acquired during our educational cursus as future engineers.

BIBLIOGRAPHY

[1] : Julio Alberto Mendoza-Mendoza, Victor Javier Gonzalez-Villela, Gabriel Sepulveda-Cervantes, Mauricio Mendez-Martinez, Humberto Sossa-Azuela, "Advanced Robotic Vehicles Programming: An Ardupilot and Pixhawk Approach"[online], 2020 [visited on 09/06/2020], PDF format. Available on: https://www.Oreailly.com

[2] : Greg Loyse. Raspberry Pi documentation, [visited on 21/06/2020] available on: https://RaspberryPi.com

[3] : Dronecode, Open source for drones. [visited on 23/04/2020] available on: https://px4.io/

[4] : Documentation ROS Wiki. [visited on 20/05/2020]: https://wiki.ros.org/

[5] : Lentin Joseph, Robot Operating System for Absolute beginners. 2018, pp152-159

[6] : Hernandez Brice, 'Commande par linéarisation entrée-sortie d'un drone de type quadcopter à l'aide de la Kinect One', Mémoire : L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE, MONREAL LE 11 OCTOBRE 2017, pp17-24

[7] : Artur Banach. 'Visual control of the Parrot drone with OpenCV, ROS and Gazebo Simulator'. [12/06/2016], pp06-10

[8] : TensorFlow website.[visited on 25/06/2020]: https://www.tensorflow.org/

[9] : Tom Hope, Yehezkel S.Resheff and Itay Lieder, 'Learning TensorFlow: A guide to building deep learning systems', pp06-07 and pp136-137

[10] : Drone nodes-Explore. [visited on 26/06/2020]: https://dronenodes.com/

[11] : Drone trest. [visited on 26/06/2020]: https://www.dronetrest.com

[12] : Oscar Liang. sharing knowledge and ideas. [visited 25/06/2020]: https://oscarliang.com/

[13] : Raspberry Pi - Wikipedia [visited on 26/06/2020]: https://en.wikipedia.org/wiki/Raspberry_Pi

APPENDIX

APPENDIX A : drone control node full source code

```
import rospy, mavros, time, os
import math
from geometry_msgs.msg import PoseStamped, Twist
from geographic_msgs.msg import GeoPointStamped
from mavros_msgs.msg import State, Altitude
from mavros_msgs.srv import CommandBool, SetMode
from std_msgs.msg import Float64, String
from tf.transformations import *
from sensor_msgs.msg import LaserScan, Imu
import keyboard
import pyperclip

# callback method for state sub
current_state = State()
offb_set_mode = SetMode
def state_cb(state):
    global current_state
    current_state = state

#callback method for position subscriber
def position_cb(get_pose):
   global altitude
   altitude = get_pose.pose.position.z
   global x_pos
   x_pos = get_pose.pose.position.x
   global y_pos
   y_pos = get_pose.pose.position.y
   global current_yaw
   qx=get_pose.pose.orientation.x
   qy=get_pose.pose.orientation.y
   qz=get_pose.pose.orientation.z
   qw=get_pose.pose.orientation.w
   current_yaw = math.atan2( 2.0*(qw*qz + qx*qy),  1.0 - 2.0*(qy**2 + qz**2) )

#callback method for altitude subscriber
def alt_cb(data):
    global rel_alt
    rel_alt = data.relative

#callback method for imu data (accelerometer)
def imu_cb(data):
    global x_accel
    x_accel = data.linear_acceleration.x
    global y_accel
    y_accel = data.linear_acceleration.y
    global z_accel
    z_accel = data.linear_acceleration.z

#callback method for user input subscriber
def input_cb(user_input):
    global input_str
    input_str = user_input.data

#callback method for rplidar subscriber
def rplidar_cb(data):
    global range_min
    range_min = data.range_min
    global range_max
    range_max = data.range_max
    global ranges
    ranges = data.ranges


mavros.set_namespace()

########## Define Publishers
local_pos_pub = rospy.Publisher(mavros.get_topic('setpoint_position', 'local'), PoseStamped,
queue_size=10)
body_vel_pub = rospy.Publisher(mavros.get_topic('setpoint_velocity', 'cmd_vel_unstamped'),
Twist, queue_size=10)
set_geo_pub = rospy.Publisher(mavros.get_topic('global_position', 'set_gp_origin'),
GeoPointStamped, queue_size=10)

########## Define Subscribers
state_sub = rospy.Subscriber(mavros.get_topic('state'), State, state_cb)
local_pos_sub = rospy.Subscriber(mavros.get_topic('local_position', 'pose'), PoseStamped,
position_cb)
alt_sub = rospy.Subscriber(mavros.get_topic('altitude'), Altitude, alt_cb)
imu_sub = rospy.Subscriber(mavros.get_topic('imu', 'data'), Imu, imu_cb)
input_sub = rospy.Subscriber('UserInput', String, input_cb)
laser_sub = rospy.Subscriber('scan', LaserScan, rplidar_cb)

########## Define Services
arming_client = rospy.ServiceProxy(mavros.get_topic('cmd', 'arming'), CommandBool)
set_mode_client = rospy.ServiceProxy(mavros.get_topic('set_mode'), SetMode)

targetHeight = 1.0  # Fly 1 meter high

geo_pos = GeoPointStamped()
geo_pos.position.latitude = 0
geo_pos.position.longitude = 0
geo_pos.position.altitude = 0
```

```
initial_pose = PoseStamped()
initial_pose.pose.position.x = 0
initial_pose.pose.position.y = 0
initial_pose.pose.position.z = 0

input_str = "ready" # Initialize input command

os.system('clear')  # Clear screen

print('Connecting...')

def position_control():
    rospy.init_node('offb_python_node', anonymous=True)
    prev_state = current_state
    freq = 20
    rate = rospy.Rate(freq) # MUST be more then 2Hz

    recent_angles = [0]
    recent_displacements = [0]
    global angle
    angle = 0
    global displacement
    displacement = 0
    global xvel
    xvel = 0
    global yvel
    yvel = 0
    global zvel
    zvel = 0

    global x_pos
    x_pos = 0
    global y_pos
    y_pos = 0
    global current_yaw
    current_yaw = 0
    global range_min
    range_min = 0
    cnt = 0
    global Height
    Height = 0
    zyaw = 0

    last_direction = "U"

    pose = PoseStamped()
    pose.pose.position.x = x_pos
    pose.pose.position.y = y_pos
    pose.pose.position.z = targetHeight
    # send a few setpoints before starting

    for i in range(100):
        local_pos_pub.publish(pose)
        rate.sleep()


    # wait for FCU connection
    while not current_state.connected:
        rate.sleep()

    last_request = rospy.get_rostime()
    while not rospy.is_shutdown():
        now = rospy.get_rostime()
        if current_state.mode != "OFFBOARD" and (now - last_request > rospy.Duration(5.)):
            set_mode_client(base_mode=0, custom_mode="OFFBOARD")
            last_request = now
        else:
            if not current_state.armed and (now - last_request > rospy.Duration(5.)):
                arming_client(True)
                last_request = now

        # older versions of PX4 always return success==True, so better to check Status instead
        if prev_state.armed != current_state.armed:
            rospy.loginfo("Vehicle armed: %r" % current_state.armed)
        if prev_state.mode != current_state.mode:
            rospy.loginfo("Current mode: %s" % current_state.mode)
            offboard_started_time = rospy.get_rostime()
        prev_state = current_state

        now = rospy.get_rostime()

        os.system('clear')


################################################################################
        ############# READY MODE
################################################################################
        if input_str == "ready":
            print("Ready for mode")

            reset_pos = True
            first_run = True


            rate.sleep()
```

```
##############################################################################
    ############# SAFETY BUTTON MODE
##############################################################################
    if input_str == "sb":
      print("Safety Button mode")

      reset_pos = True
      first_run = True

      rate.sleep()



##############################################################################
    ############# TAKEOFF MODE
##############################################################################

    if input_str == "to":
      print("Takeoff")

      if reset_pos:
        pose = PoseStamped()
        pose.pose.position.x = x_pos
        pose.pose.position.y = y_pos
        pose.pose.position.z = targetHeight

        set_geo_pub.publish(geo_pos)


      if current_state.mode != "OFFBOARD" and (now - last_request >
rospy.Duration(5.)):
        arming_client(True)
        set_mode_client(base_mode=0, custom_mode="OFFBOARD")
        last_request = now

      local_pos_pub.publish(pose)

      reset_pos = False


      rate.sleep()
    ############################################################################
    elif input_str == "f":
      print("Moving forward")
      pose = PoseStamped()
      pose.pose.position.x = x_pos + 1
      pose.pose.position.y = y_pos
      pose.pose.position.z = altitude
      set_geo_pub.publish(geo_pos)
```

```
            if current_state.mode != "OFFBOARD" and (now - last_request >
rospy.Duration(5.)):
                        arming_client(True)
                    set_mode_client(base_mode=0, custom_mode="OFFBOARD")
                    last_request = now

            local_pos_pub.publish(pose)
            rate.sleep()
        elif input_str == 'b':
            print("Moving back")
            pose = PoseStamped()
            pose.pose.position.x = x_pos -1
            pose.pose.position.y = y_pos
            pose.pose.position.z = altitude
            set_geo_pub.publish(geo_pos)

            if current_state.mode != "OFFBOARD" and (now - last_request >
rospy.Duration(5.)):
                        arming_client(True)
                    set_mode_client(base_mode=0, custom_mode="OFFBOARD")
                    last_request = now

            local_pos_pub.publish(pose)
            rate.sleep()
        elif input_str == 'sr':
            print("Sliding right")
            pose = PoseStamped()
            pose.pose.position.x = x_pos
            pose.pose.position.y = y_pos - 1
            pose.pose.position.z = altitude
            set_geo_pub.publish(geo_pos)

            if current_state.mode != "OFFBOARD" and (now - last_request >
rospy.Duration(5.)):
                        arming_client(True)
                    set_mode_client(base_mode=0, custom_mode="OFFBOARD")
                    last_request = now

            local_pos_pub.publish(pose)
            rate.sleep()
        elif input_str == 'sl':
            print("Sliding left")
            pose = PoseStamped()
            pose.pose.position.x = x_pos
            pose.pose.position.y = y_pos +1
            pose.pose.position.z = altitude
            set_geo_pub.publish(geo_pos)
```

```
          if current_state.mode != "OFFBOARD" and (now - last_request >
rospy.Duration(5.)):
                      arming_client(True)
                   set_mode_client(base_mode=0, custom_mode="OFFBOARD")
                   last_request = now

          local_pos_pub.publish(pose)
          rate.sleep()
       elif input_str == 'gu':
          print("Going up")
          pose = PoseStamped()
          pose.pose.position.x = x_pos
          pose.pose.position.y = y_pos
          pose.pose.position.z = altitude + 0.5
          set_geo_pub.publish(geo_pos)

          if current_state.mode != "OFFBOARD" and (now - last_request >
rospy.Duration(5.)):
                      arming_client(True)
                   set_mode_client(base_mode=0, custom_mode="OFFBOARD")
                   last_request = now

          local_pos_pub.publish(pose)
          rate.sleep()
       elif input_str == 'gd':
          print("Going down")
          pose = PoseStamped()
          pose.pose.position.x = x_pos
          pose.pose.position.y = y_pos
          pose.pose.position.z = altitude - 0.5
          set_geo_pub.publish(geo_pos)

          if current_state.mode != "OFFBOARD" and (now - last_request >
rospy.Duration(5.)):
                      arming_client(True)
                   set_mode_client(base_mode=0, custom_mode="OFFBOARD")
                   last_request = now

          local_pos_pub.publish(pose)
          rate.sleep()

##############################################################################
       ############# LAND MODE
##############################################################################

       elif input_str == "l":
          print("Landing")

          reset_pos = True
```

```
        if current_state.mode != "AUTO.LAND" and (now - last_request >
rospy.Duration(5.)):
            set_mode_client(base_mode=0, custom_mode="AUTO.LAND")
            last_request = now

        rate.sleep()


############################################################################
 ############# RETURN TO LAUNCH MODE
############################################################################

      elif input_str == "return to launch":
        print("Returning to Launch Location")

        #reset_pos = True

        if current_state.mode != "AUTO.RTL" and (now - last_request > rospy.Duration(5.)):
            set_mode_client(base_mode=0, custom_mode="AUTO.RTL")
            last_request = now

        rate.sleep()


      ############################################################
        ######### DISARM  ##########################################
      ############################################################
      elif input_str == "d":
            print("Disarming")

            arming_client(False)

            rate.sleep()

      #############################################################
      ############# HOVER MODE ###################################
      #############################################################

      elif input_str == "h":
        print("Hovering")

        twist=Twist()

        xvel = 0
        yvel = 0

        # Maintain Altitude
        zvel = 0

        print(zvel)
```

```
    twist.linear.x = xvel
    twist.linear.y = yvel
    twist.linear.z = zvel
    twist.angular.x = 0
    twist.angular.y = 0
    twist.angular.z = 0
    height = altitude
    body_vel_pub.publish(twist)



    rate.sleep()

############################################################################
############## LINE FOLLOW MODE #######################################
############################################################################

elif input_str == "t":
    print("Following Line\n")

    twist=Twist()

    direction = pyperclip.paste()
    print(pyperclip.paste())

    if direction == "R":
        twist.linear.x, twist.linear.y = mov_xy(xvel+2,yvel,current_yaw)
        twist.angular.z -= 1
        pyperclip.copy("")

    elif direction == "L":
        twist.linear.x, twist.linear.y = mov_xy(xvel+2,yvel,current_yaw)
        twist.angular.z += 1
        pyperclip.copy("")

    elif direction == "N":
        twist.linear.x, twist.linear.y = mov_xy(xvel+3,yvel,current_yaw)
        twist.angular.z = 0
        pyperclip.copy("")

    elif direction == "U":
        twist.linear.x, twist.linear.y = mov_xy(xvel, yvel, current_yaw)
        twist.angular.z = 0
        print("I got U info")
        pyperclip.copy("")

    else:
        # Clipboard is empty or has undefined value so do nothing
        print("I got nothin man")
```

```
            pass

          zvel = (Height - rel_alt)*2
          twist.linear.z = 0

          print("z: " + str(twist.angular.z))

          body_vel_pub.publish(twist)
          rate.sleep()

def mov_x(spd,yaw):
  return (math.cos(yaw)*spd,math.sin(yaw)*spd)
def mov_y(spd,yaw):
  return (math.sin(yaw)*spd,math.cos(yaw)*spd)
def mov_xy(x,y,yaw):
  x1,y1=mov_x(x,yaw)
  x2,y2=mov_y(y,yaw)
  return (x1+x2,y1+y2)


if __name__ == '__main__':
    try:
        position_control()
    except rospy.ROSInterruptException:
        pass
```

APPENDIX B : User input node source code

```
#!/usr/bin/env python

import rospy
import os
from std_msgs.msg import String

os.system('clear')

def input_pub():
   pub = rospy.Publisher('UserInput', String, queue_size=2)
   rospy.init_node('UserInputNode', anonymous=True)
   rate = rospy.Rate(10) # 10hz
   input_str = ""
   while not rospy.is_shutdown():
      if input_str == "":
         print("Please enter a mode \n")
      elif input_str == "to" or input_str == "tt" or input_str == "h" or input_str == "t" or 
input_str == "l" or input_str == "d" or input_str == "sr" or input_str == "sl" or input_str == 
"f" or input_str == "b" or input_str == "gu" or input_str == "gd":
            print("Current mode: %s \n " % (input_str))
      else:
            print("'%s' is not a valid mode. Please try again \n" % (input_str))
      print("Takeoff Mode --------------------> to ")
      print("Takeoff and Track Mode ----------> tt ")
      print("Hover Mode ----------------------> h ")
      print("Track Mode ----------------------> t ")
      print("Land Mode -----------------------> l ")
      print("Slide right----------------------> sr")
      print("Slide left-----------------------> sl")
      print("Move forward---------------------> f")
      print("Move back------------------------> b")
      print("Go up----------------------------> gu")
      print("Go down--------------------------> gd")
      print("Disarm Mode ---------------------> d \n")

      input_str = raw_input("Mode: ")

         os.system('clear')

         pub.publish(input_str)

         rate.sleep()

if __name__ == '__main__':
   try:
      input_pub()
   except rospy.ROSInterruptException:
      pass
```

APPENDIX C: Image classification source code

```
import  os, sys
from keras.models import load_model
import numpy as np
import cv2
import pyperclip
import time
import signal
from datetime import datetime
from threading import Thread, Event
#sys.stderr = stderr
from tqdm import tqdm

#These are the only things that should need to change
res=32                          #The resolution that we want the images to be
freq=25                                 #The number of frames you want to try to process per second

runtime=3600 #The number of seconds the program will run
#Basic number crunching, initialization, and directory walking to get everything prepared to
launch
default_dir = 'D:\RaspberryPiCode\keras_ws'
for (_,_,files) in os.walk(default_dir):
        break
for file in files:
        if file.endswith('.h5'):
                model=load_model(default_dir+'/'+file)
                i=np.zeros([1, res, res, 1])*255
                model.predict_classes(i)
                print('\n The model was actually loaded \n')
                break
cap=cv2.VideoCapture(0)
cap.set(3,res)
cap.set(4,res)
try:
        ready,img=cap.read()
except:
        pass
translator=np.array(['L','N','R','U'])
pbar=tqdm(total=0, bar_format='  Time: {elapsed}  Rate: {rate_fmt}  Total: {n}  Output:
{desc}', unit=" frame",mininterval=0.001,maxinterval=1.)
w=1/freq/workers
period=1/freq
threads = []
if not cap.isOpened():
        print('\n The VideoCapture object is closed \n')
finish_time=time.time()+runtime

#The function everything is built around
def img_process():
        ready,img=cap.read()
        if ready:
                img = cv2.resize(img,(res,res),interpolation = cv2.INTER_AREA)
                img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
                img = np.expand_dims(img,axis=0)
                img = np.expand_dims(img,axis=-1)
                p = translator[model.predict_classes(img/255)[0]]
                pyperclip.copy(p)
                pbar.update()
                pbar.set_description_str("%s" % p)

img_process()
pbar.close()
cap.release()
```