Politecnico di Torino

Master's degree course in Mechatronic Engineering

Master's Degree Thesis

Development of a ROS2 flight software framework & Attitude Determination application for nanosatellites

Candidate:

Marzani Nicolò

Supervisor:

Prof. Corpino Sabrina

Tutor:

Eng. Zanotti Andrea

Academic year 2020/2021

Ad A. e alla mia famiglia

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

1.1. Nanosatellites overview

In recent years, space exploration has attracted a lot of social interest and economic investments by both public and private entities. The development of new technologies, that can be useful and applied in many fields, has allowed the foundation of many realities that are now leaders of the space industry. In this context a particular implementation of these new technologies, that is becoming a very important part of the space exploration sector, is made up by the *CubeSats.*

The first CubeSat was developed by the California Polytechnic State University and Stanford University in 1999 for educational purposes, then due to their low costs they have been adopted in space industry for many types of missions. These artificial satellites are small and light, normally with a mass below 500 kg, and they are instrumented with particular devices called *payloads* used for collecting data and in general for performing an assigned mission (data collection, science experiments, …). Depending on their masses, they can be classified in minisatellites (100~500Kg), microsatellites (10~100Kg) or nanosatellites (1~10Kg). In general the standard for CubeSats is the 1U (one unit) that has dimensions 10x10x10 cm, 1 dm^3 volume and a weight not more than 1.33 Kg; is also possible to have bigger ones with other configurations like 3U CubeSat with dimensions of 10x10x30 cm or 6U CubeSat 10x20x30cm and so on.

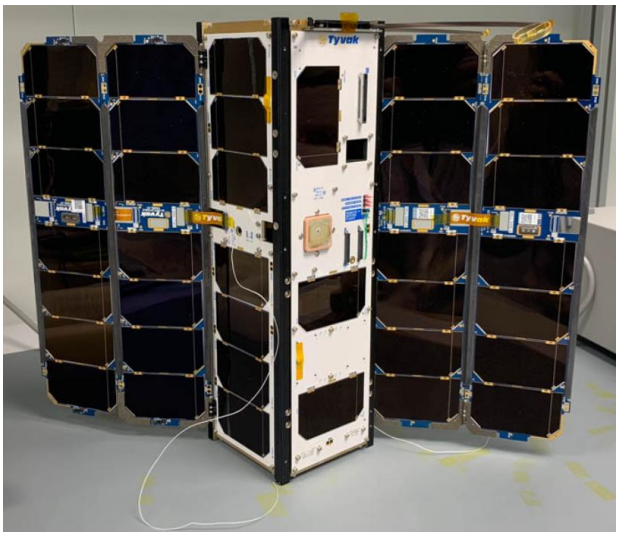

Figure 1. 1: Tyvak's Commtrail Satellite (3U)

They are widely employed because their production and launch costs are cheaper compared to a bigger standard satellite: in general the bigger is the satellite the bigger the rocket must be for reaching the desired orbit and, in addition, it is also possible to deploy more satellites with a single launch. Nowadays nanosatellites can be applied in many different fields that range from earth observation to space exploration and, in the near future, in planetary defence too with the

ESA Hera mission. Due to their small dimensions they can be easily employed in swarm for performing missions that could not be possible for single satellites: data collection about the same phenomenon from different positions, in-orbit inspection of bigger satellites and many others. Even if their concept is very simple since the body of these satellite is made up by cubes, they involve very complex technologies from both electronic/mechatronics (sensors, actuators, …) and software side for implementing all the required subsystems that the satellite needs.

Among these subsystems there is the ***ADCS*** (attitude, determination and control system), intended for monitoring the attitude of the satellite and to autonomously perform control actions on the actuators for accomplishing several duties, for example the "detumbling" of the satellite when it is deployed in the orbit. This system in particular requires a software framework able to collect data from several sensors and to send the right control action to the mounted actuators, at a fixed rate (that can be very high). In order to simplify the software implementation and management, a framework like ROS2 (second version of the *Robot Operating System*) can take advantage for its simplicity and modularity. It's strongly supported by the community and provides native functions that ranges from navigation services to graphical visualization for simulation and debugging. ROS2 it's widely used in the robotic industry, but it can be easily applied to different fields due to the advantages listed before.

1.2. Thesis objective and context

This thesis work is an R&D (*research and design*) project which context takes place in the aerospace industry, particularly in the field of software engineering for nanosatellites. The design and the validation of a software framework is one of the most critical phases in the realization of a complex system like nanosatellites and it must follow a precise life cycle dictated by software engineering rules. The steps to achieve a good software realization can be described with a V-shaped process flow, presented in Figure 1. 2:

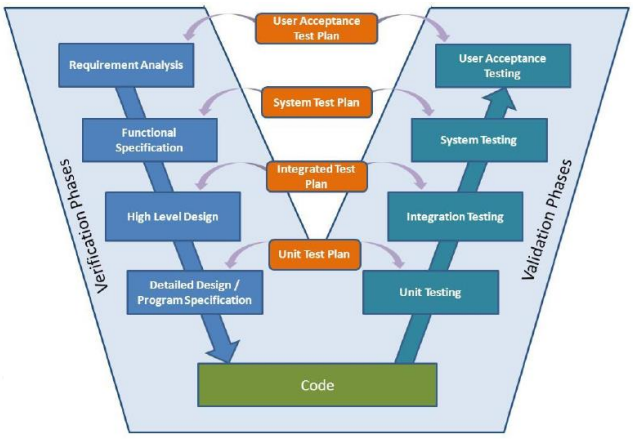

Figure 1. 2: V-Shaped Software flow

The left part of the V-shaped flow includes the verification and design process of the system while the right part includes the validation process:

- The first phase is the analysis of the system in terms of requirements. Based on the functionality of the system, the requirements can be classified in functional requirements (to describe how the system must respond to specific input and the list of the operations that the system must perform) and domain requirements (to specify the domain of interest of the system). This phase also incorporates the prediction of the cost of the system.

- The second phase is the system design and it includes a first part concerning the architectural design, which defines which are the applications that must be implemented and how they communicate with each other. The second part is the detailed design and program specification, to define the deadlines for the development of the applications and how to implement them.

- After that, the drawing up of the code can start and it results to be the core phase of the software development.

- Once all the applications of the software are developed, the software needs to be validated. To do this, different kind of tests are performed to check that the system works properly. The first test to be performed is the unit testing which consists to test the single applications developed to check if bugs are present and if they realize the proper functionality. After that, the applications modules are integrated into subsystems and they are tested together as a group (integration testing). If in these two phases, all the functionalities are satisfied and the subsystems work properly, the whole system is integrated and tested (system testing) to check that all the functionalities are implemented and cooperate properly.

- Finally, the software framework design can be considered completed and it is delivered to the clients, but it always needs to be maintained.

The maintaining phase includes also the so-called “evolution” of the system, which incorporate bugs to be fixed, changes in the requirements, new updates and releases or new features to be added. All these operations are considered critical since they increase the cost of the development.
To simplify these processes, new approaches to software engineering are considered. A first and widely adopted solution is ***MBSD*** (Model Based Software Design) which consists in realizing a model of the system in a simulation environment like Matlab/Simulink and auto-generate the C/C++ code, with provided toolboxes, for implementing control systems in suitable embedded systems. Considering nanosatellites as example, this solution can be a good choice for the development of the ADCS since the control laws are designed in Simulink and, once the simulations results are evaluated, the code can be directly obtained from these models.
Another possible solution is to design the software framework with tools and libraries like ROS

(*Robot operating system*) or ROS2. These have taken hold mainly in robotics applications but they can be easily employed in the design of any kind of complex system, even nanosatellites, by providing a lot of APIs (*Application programming interfaces*) to implement common features for mechatronic systems.
In this scenario takes place this thesis work, linked to a new R&D project started by Tyvak International and intended to demonstrate and realize a first implementation of an Italian flight software framework for nanosatellites using ROS2 and apply it to an attitude determination application to illustrate the proper functioning of the whole system.
The main reason that convinced the software team of Tyvak International to start this new R&D project (named *Phoenix*) is related to the fact that Tyvak International is a start-up born by the American counterpart Tyvak Nanosatellites that provides technologies for the their satellites, including the software framework.
For this reason Tyvak International does not hold its own flight software framework, and that could cause problems in managing the software, find bugs and realize patches to correct them. This means that, if there is an intention of implementing a new feature, a reverse engineering process has to be done to understand how to integrate that feature on the provided framework realized by Tyvak Nanosatellites. The flight software framework developed by Tyvak Nanosatellites (*MK-2*) is taken as starting point to understand what are the main applications that are needed for a real satellite to allow it to perform in-orbit operations. After that, the fundamental applications to realize a first implementation of the system to achieve an attitude determination (watchdog, reading sensors and telemetry) are implemented into ROS2 nodes and their structure will be described in the following chapters of the thesis.

1.3. Structure of the thesis

The thesis is intended to explain the development process of some applications enabling the ROS2 flight software framework, by explaining the concept of each node and why the selected solution can be better compared to another one. Finally, an application related to the Attitude Control system is studied and tested in MATLAB/Simulink. The thesis is structured as following:

- Chapter 2: a brief explanation of both the hardware and software used for the project, starting from the sensor module to the Raspberry Pi 3 B+ embedded system, describing their usage and reporting the circuit diagram used as reference for building the final electronic circuit. Finally, an overview of ROS2 is presented, listing some peculiarities and advantages.

- Chapter 3: description of the implemented nodes in ROS2, explaining the concept of each one and some architectural choices. Finally their functioning and the practical implementation in python.

- Chapter 4: the general problem of attitude determination is presented within the TRIAD algorithm solution. Finally, the attitude determination node is explained in detail.

- Chapter 5: a MATLAB real-time animation is performed to test the system designed in ROS2. Particularly, a 3U satellite is simulated to visualize its attitude determination. The simulation uses ROS Toolbox that allows the integration of ROS / ROS2 with Matlab / Simulink.

- Chapter 6: some personal conclusions about the project and suggestions for future improvements and developments.

- Appendix A: general description of Buildroot and description of the first attempt to realize an image for a Tyvak custom board

2. SYSTEM ARCHITECTURE

2.1. Project overview

Since the objective of the thesis is to realize a first version of a new flight software framework, based on ROS2, and to study in details a possible application, a preliminary selection of fundamentals applications needed in a flight software is performed. To this aim the MK-2 flight software developed by Tyvak Nanosatellites is taken as example, for understand how a flight software is designed and which applications are needed for realizing a first implementation.
Among the applications implemented in the MK-2 flight software, this combination of them has been preferred:

- Watchdog: to check the status of other important applications.
- Sensors reader: for enabling the sensor data reading over I2C/SPI buses.
- Sensors telemetry: to store the collected data.

The selection of these applications (detailed in the following sections) is not casual; indeed they can ensure the enabling of a first draft of a flight software framework that will be able to collect data from sensors, store them and to autonomously react to sudden crashes affecting its processes. Moreover this first version of flight software can be used for a simple ADCS application.
In order to test the developed flight software the reference embedded system selected is the Raspberry Pi 3 B+.

2.2. Hardware configuration

This section is devoted to broadly introduce all the hardware needed for the thesis project, paying attention to the connections between all the components rather than describing in detail each one of them; this job will be performed in the following sections.
The components used are:

- Raspberry Pi 3 B+ as embedded system, used for managing the collected sensors data and executing all the ROS2 processes.
- A custom sensor module, provided by Tyvak International, generally used for attitude determination purposes. It mounts an AD7415 temperature sensor, an HMC5883L magnetometer and a E910.86 sun sensor.

- A custom connector for interfacing with the sensor module.
- A TXB0108 level shifter for properly connecting the sun sensor to the Raspberry.

A level shifter is a very simple device that rescales a certain voltage, in this case the 5V voltage coming from the MISO output line of E910.86, to another desired voltage, in this case the 3.3V accepted by the Raspberry GPIO pins.
The TXB0108 level shifter is mandatory for connecting the E910.86 sun sensor to the Raspberry Pi 3 B+ without damaging the board because, as can be seen in Figure 2. 1, the MISO output signal that would go from the sun sensor to the Raspberry pins works at voltages that are greater than the voltage tolerated by the Raspberry GPIO pins, that is 3.3V.

No.	Parameter	Condition	Symbol	Min.	Typ.	Max.	Unit
SPI DC Characteristics, output terminal MISO							
1	Output voltage low	I = 0.5mA	V_{MISOL}			0.4	V
2	Output voltage high	I = -0.2mA	V_{MISOH}	V_{DD} - 0.4			V

Figure 2. 1: E910.86 MISO output voltages. Vdd=4.5V to 5.5V

The connections between all the components are schematized in this circuit diagram:

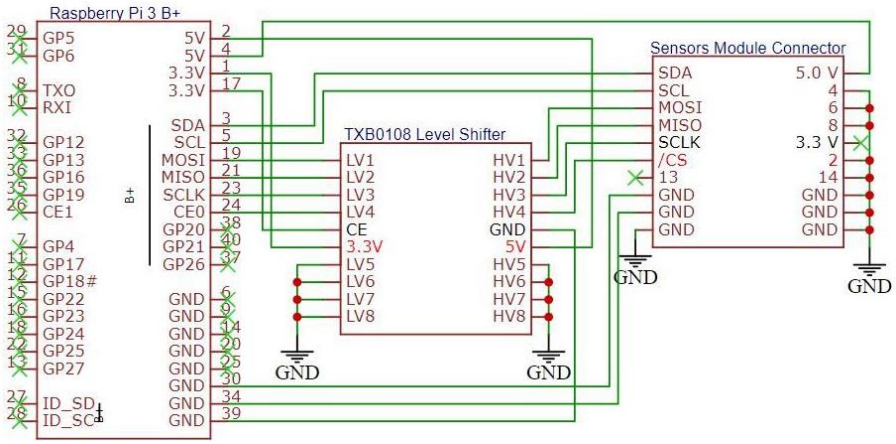

Figure 2. 2: Components connections circuit diagram

2.2.1. Temperature sensor AD7415

The AD7415 sensor is a standalone digital temperature sensor, widely used in several fields of applications, that is mounted in the provided sensor module. The serial interface is *I2C* and *SMBus* compatible, due to this the sensor can be easily interfaced with "*smbus2"* python library. The sensor requires a 2.7V to 5.5V power supply and so it can be used without any problems with a Raspberry PI 3 B+. A schematic representation of the sensor register structure is portrayed in the following figure:

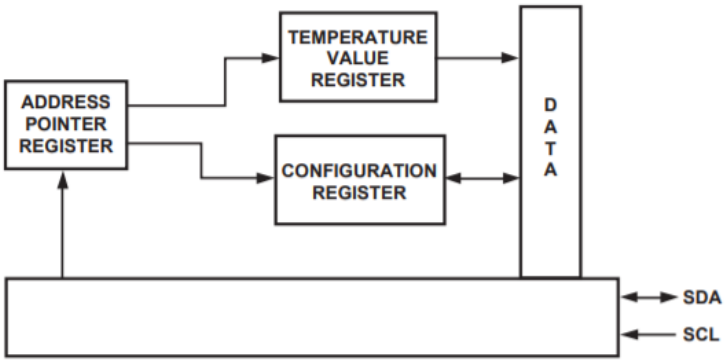

Figure 2. 3: AD7415 Register structure

To correctly initialize the AD7415 we must configure it by writing a particular byte in its configuration register at *0x01* address.

[1] Default settings at power-up.

D7	D6	D5	D4	D3	D2	D1	D0
PD	FLTR	TEST MODE			ONE SHOT	TEST MODE	
0[1]	1[1]	0s[1]			0s[1]	0s[1]	

Figure 2. 4: AD7415 Configuration register bits definition

For the thesis purposes a very simple configuration has been selected by writing a "1" in the **ONE SHOT** bit of the configuration register. In this way the AD7415 is expected to power-up, perform a single conversion and then power down again automatically.
Finally, the sensor is able to perform the temperature sensing and to store the result on the temperature register at *0x00* address. The temperature value register is a 10-bit, read-only register that stores the temperature reading from the ADC in twos complement format.
Two reads are necessary to read the actual data from this register:

Temperature Value Register (First Read)

D15	D14	D13	D12	D11	D10	D9	D8
MSB	B8	B7	B6	B5	B4	B3	B2

Temperature Value Register (Second Read)

D7	D6	D5	D4	D3	D2	D1	D0
B1	LSB	N/A	N/A	N/A	N/A	N/A	N/A

Figure 2. 5: AD7415 Temperature value register readings output

As written in Figure 2. 5. above, by reading the temperature value register twice, we will obtain two bytes containing the actual 10-bit data needed and other N/A bites that are neglectable. After extracting the raw digital value of the temperature in the 10-bit form from this row of bits (from D6 bit to D15 bit), is easy to retrieve the actual value of the temperature in **°C** since the

temperature resolution of the ADC is 0.25 °C, which corresponds to 1 LSB of the ADC; so by applying the following function:

$$Temperature[°C] = \frac{Raw\_digital\_temperature\,_{[decimal]}}{4}$$

the value of the temperature in °C is obtained.
The circuit diagram of the sensor is reported in Figure 2. 6

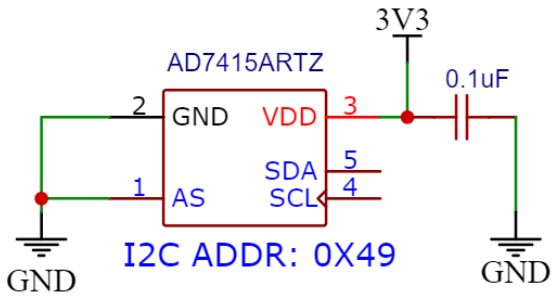

Figure 2. 6: AD7415 circuit diagram

2.2.2 AD7415 Testing

The AD7415 sensor gives as output the temperature expressed in °C. In order to check the output results, a first test is to let the sensor measure a temperature and see if the value read is always the same. The resulting output (measured with a frequency of 0.5 Hz) can be seen in Figure 2. 7:

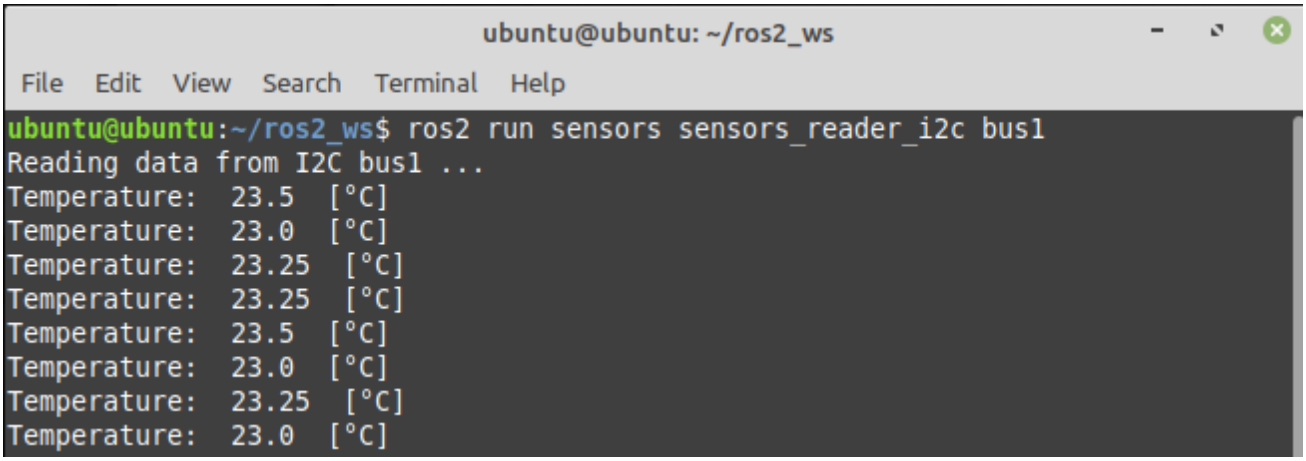

Figure 2. 7: Constant output temperature

As it can be seen, the temperature is correctly measured with an approximately error of 0.5 °C.

After that, the second test that is performed is to heat up the sensor module to check if the measured temperature increases with respect to the one measured above. The output obtained can be seen in Figure 2. 8:

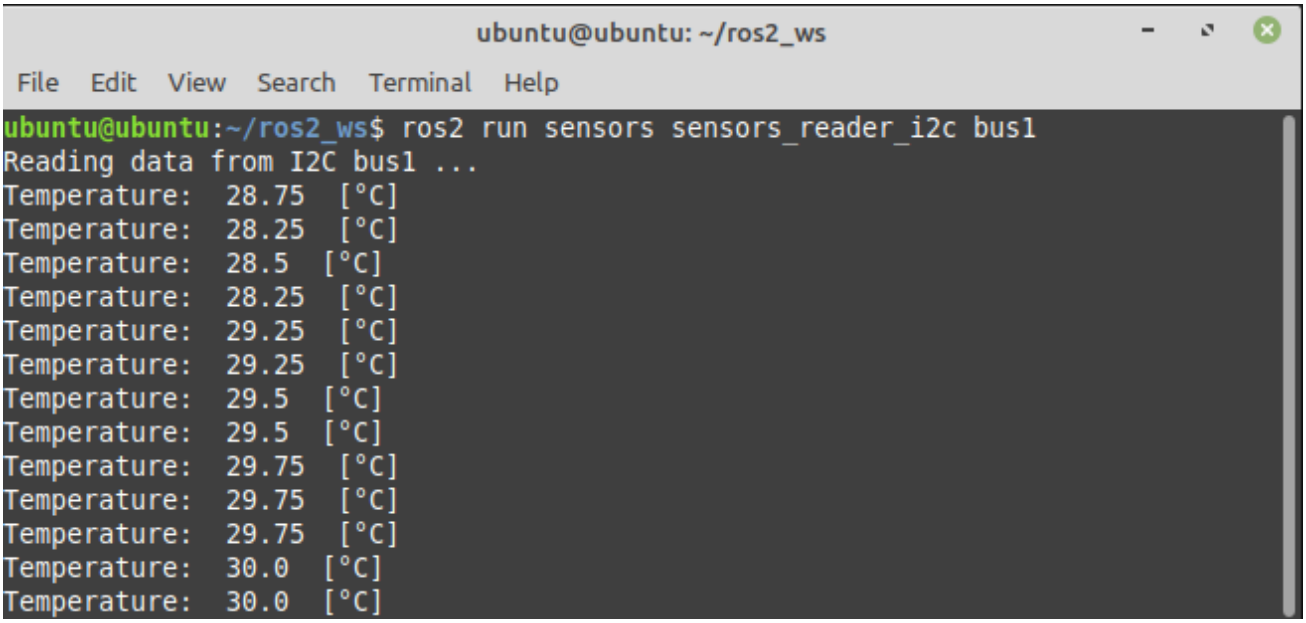

Figure 2. 8: Variable output temperature

As it can be seen, considering an initial temperature of 23 °C as the one shown in Figure 2. 8, the measured temperature is correctly greater due to the heating of the sensor module.

2.2.3 Magnetometer HMC5883L

The HMC5883L sensor is a 3-axis magnetometer supported by a 12-bit ADC coupled with a Low noise AMR sensor that achieves a 5 milli-Gauss resolution in ± 8 Gauss fields. This enables a 1° to 2° compass heading accuracy that makes this sensor suitable for mobile phones and auto-navigation systems. This magnetometer provides an I2C serial bus interface, just like the AD7415, and can be supplied with a voltage up to 3.6V.
The device is controlled and configured via several on-chip registers, described in Figure 2. 9

Address Location	Name	Access
00	Configuration Register A	Read/Write
01	Configuration Register B	Read/Write
02	Mode Register	Read/Write
03	Data Output X MSB Register	Read
04	Data Output X LSB Register	Read
05	Data Output Z MSB Register	Read
06	Data Output Z LSB Register	Read
07	Data Output Y MSB Register	Read
08	Data Output Y LSB Register	Read
09	Status Register	Read
10	Identification Register A	Read
11	Identification Register B	Read
12	Identification Register C	Read

Figure 2. 9: HMC5883L register list

So in order to use the sensor we need to properly set the bits of the configuration register A and B, and the mode register. This can be easily done with a *write* operation on the proper address location. For our purposes a continuous-measurement mode is selected by writing all zeroes in the mode register: in continuous-measurement mode, the device continuously performs measurements and places the result in the data register.
The result is stored in 3 channels (one for each axis): X, Y and Z channels; and each one of them is made up by two 8-bit output registers (A and B) where we can find the desired measurement.

DXRA7	DXRA6	DXRA5	DXRA4	DXRA3	DXRA2	DXRA1	DXRA0
(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
DXRB7	**DXRB6**	**DXRB5**	**DXRB4**	**DXRB3**	**DXRB2**	**DXRB1**	**DXRB0**
(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

Figure 2. 10: HMC5883L channel X data output registers A and B

Taking for example the A and B output registers of the X channel in Figure 2. 10 is possible to see that each register contains 8-bit (the number in parenthesis indicates the default value of that bit), and in the specific: the A output register will contain the MSB of the measurement result while the B output register will contain the LSB.
The value stored in these two registers is a 16-bit value in 2's complement form, whose range is 0xF800 to 0x07FF.
The circuit diagram of the sensor is reported in Figure 2. 11

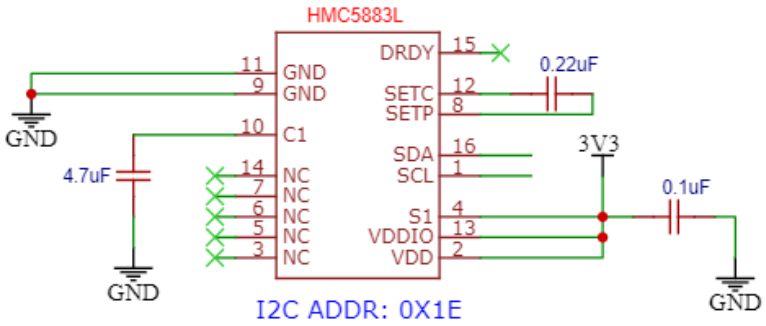

Figure 2. 11: HMC5883L circuit diagram

2.2.4 HMC5883L Testing

The output values of the magnetic field measured by the magnetometer are expressed in Gauss (G). To test it, the values measured on the three axes are printed with a frequency of 0.5 Hz. The results are shown in Figure 2. 12:

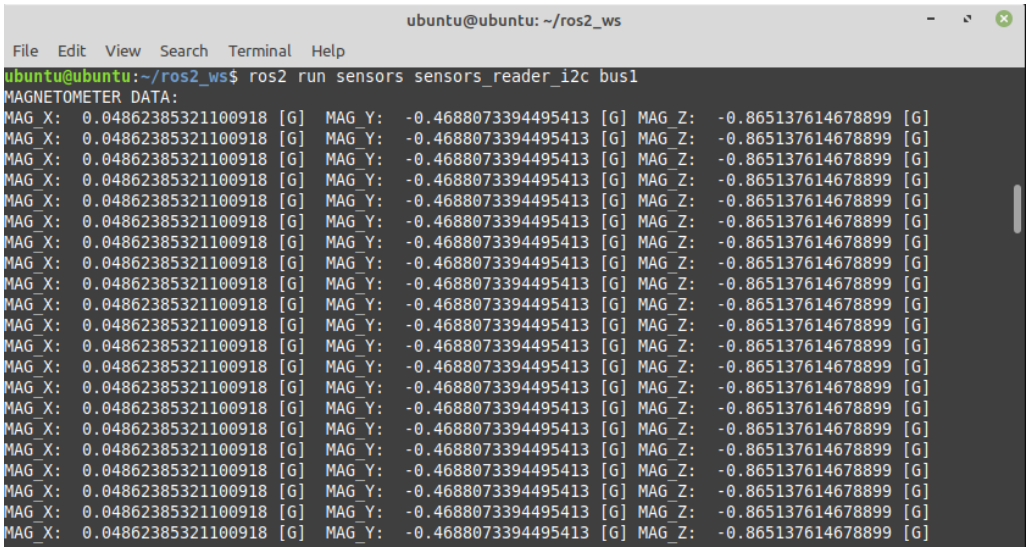

Figure 2. 12: Output values of the magnetometer

These values are expressed in the reference frame provided by the magnetometer with X axis pointing down, Z axis pointing out of the sensor and Y axis to complete a right-handed reference frame.

2.2.5 Sun sensor E910.86

The E910.86 is a two-axis digital sun sensor, manufactured by Elmos, that provides three sensing functions:

- The angle of light incidence in both xz (*Xn*) and yz (*Yn*) plane
- The light intensity for each of two different spectral range
- The chip temperature

The only output used for the purpose of this thesis is the first one. The physical representation of the Xn angle, with respect to the magnetometer reference frame, can be seen in Figure 2. 13

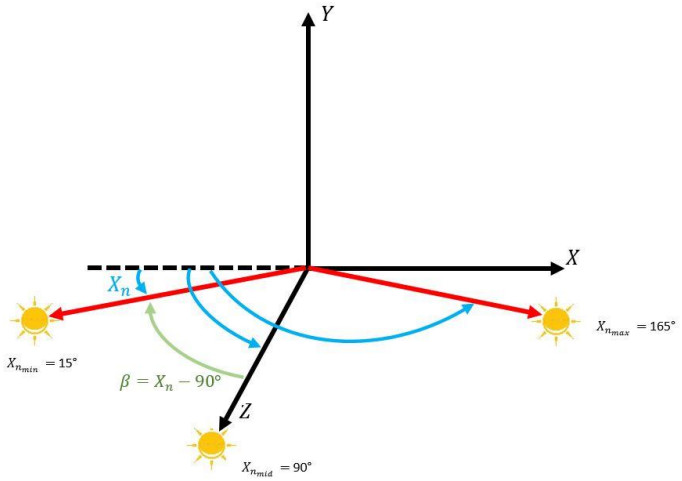

Figure 2. 13: Physical model of Xn angles

The same model can be used for the Yn angle on the yz plane.
These output values are accessible through the SPI protocol that uses a 16 bit word to communicate, composed by an address and a data section. *Read* commands start with a '00' while *write* commands start with '10'. The SPI response always starts with '01'.
According to the sensor datasheet, the commands used in order to initialize the sensor and read its result are:

Command	Operation	SPI response	Data
10x100XXYYPSZDDD	Write E910.86 and analog output status	011100XXYYPSZDDD	E910.86 and analog output status
X0x000xxxxxxxxxx	Read X and Y sensor angle data	0100$X_5X_4X_3X_2X_1X_0Y_5Y_4Y_3Y_2Y_1Y_0$	X and Y sensor data Yn= angle yz-plane Xn=angle xz-plane

Figure 2. 14: E910.86 write and read commands used

The data section of the word is used to configure the pull diodes (XX and YY operating mode (Z and DDD bits).
In order to communicate with the sensor using the SPI protocol, the Python SPIdev library is used. Once the initialization command is sent through the *xfer2* SPIdev function, and the SPI mode and frequency are set, the sensor is ready to be read.
Once the byte word (16 bits) is read, we can extract the bits referred to Xn and Yn data obtaining the following digital value: $\mathbf{X_5X_4X_3X_2X_1X_0Y_5Y_4Y_3Y_2Y_1Y_0}$.
The float value of the angles can be easily retrieved by using the following linear relation contained in the sensor datasheet and represented in Figure 2. 15.

$$X_{n_{deg}} = \frac{75 * X_{n_{byte\ word}}}{27} + 15 \qquad Y_{n_{deg}} = \frac{75 * Y_{n_{byte\ word}}}{27} + 15$$

The angles value can span from 15° to 165° with a resolution of 2.7°.

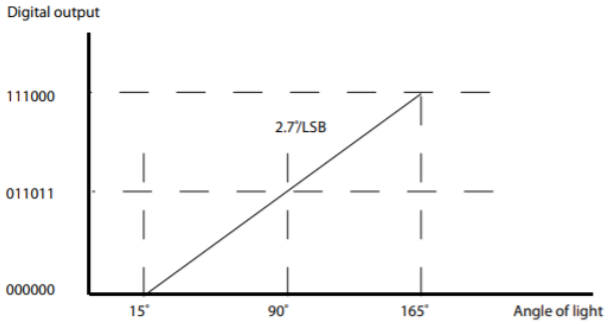

Figure 2. 15: Digital output – angles relation

Once the conversions are computed, the resulting values are the Xn and Yn angles (in radians). Now that these angles are known, referring to Figure 2. 16, the sun vector can be computed.

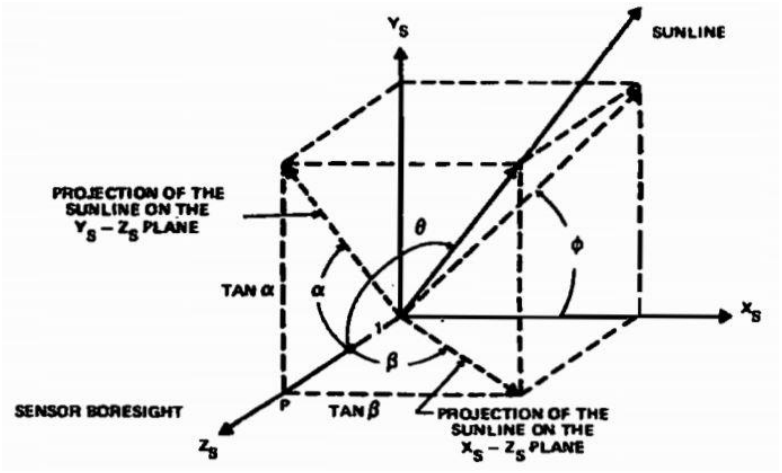

Figure 2. 16: Sun vector model

Referring to the Figure 2. 13, the angle β can be computed as Xn-90°. In this way, when Xn is ranging from 15° (the minimum value that can be obtained from the sensor) to 90° the value of β is negative; instead, when Xn is ranging from 90° to 165° (the maximum value that can be obtained from the sensor), β is positive. The same model is used for the angle Yn, using angle α instead of β.

In this way, using the values of α and β, and referring to the picture in Figure 2. 16, we can easily compute X, Y, Z coordinates of the sun vector, expressed in the sensor frame, by applying the following formula:

$$\begin{bmatrix} X_{Sb} \\ Y_{Sb} \\ Z_{Sb} \end{bmatrix} = \begin{bmatrix} tan\beta \\ tan\alpha \\ 1 \end{bmatrix}$$

The vector obtained from this computation is then normalized. The circuit diagram of the sensor is reported in Figure 2. 17

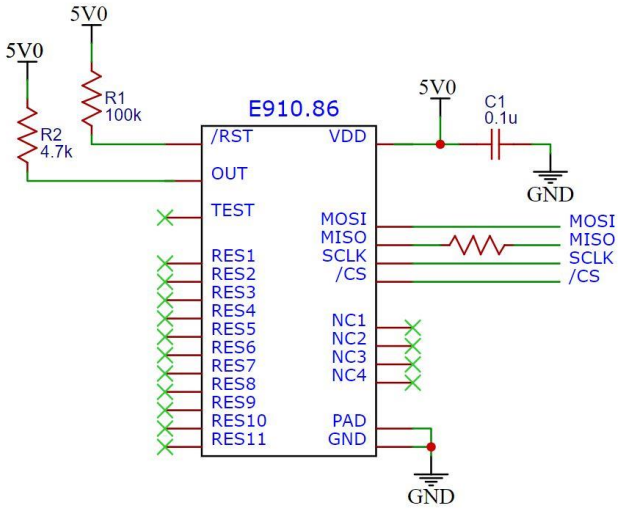

Figure 2. 17: E910.86 circuit diagram

2.2.6 E910.86 Testing

The output values provided by the Sun sensor are the XZ and YZ angles (respectively named Xn and Yn). A first test is performed in order to check if the sensor correctly measure an angle of 90° on both XZ and YZ plane when a light is positioned in front of the sensor as it is shown in Figure 2. 18:

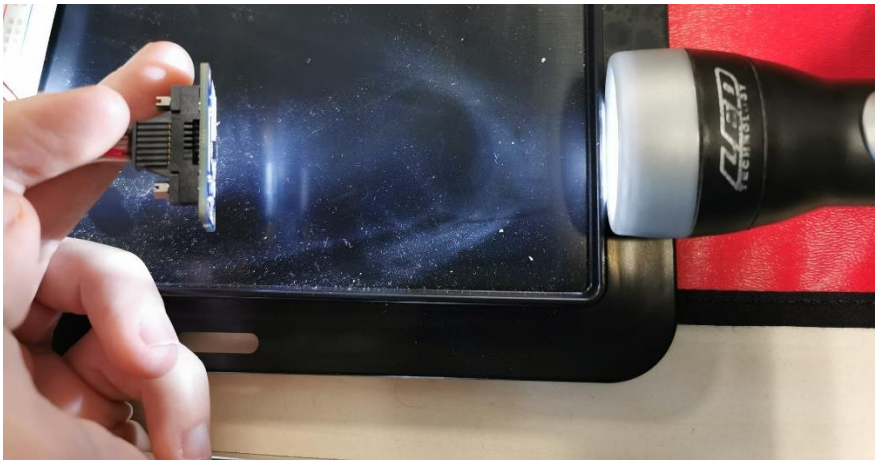

Figure 2. 18: Setting of the Sun sensor testing

The outputs obtained from this experiment are presented in Figure 2. 19:

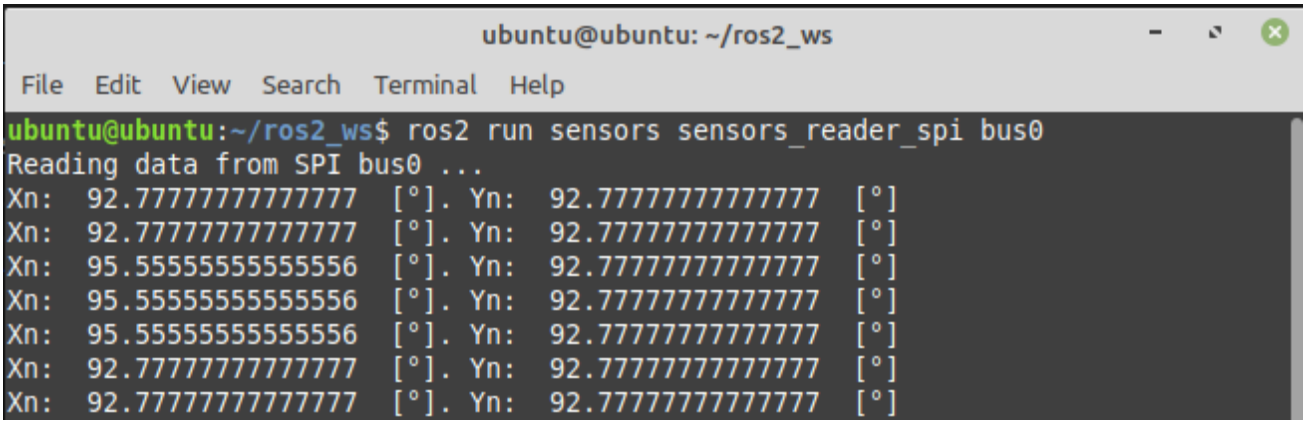

Figure 2. 19: Xn and Yn equal to 90° test

As it can be seen, the angles are correctly measured with a precision of 2.7° (due to the resolution of the sensor).
A second test is performed by moving the light along the X axis of the Sun sensor reference frame and check that the Xn angle changes. The results are presented in Figure 2. 20:

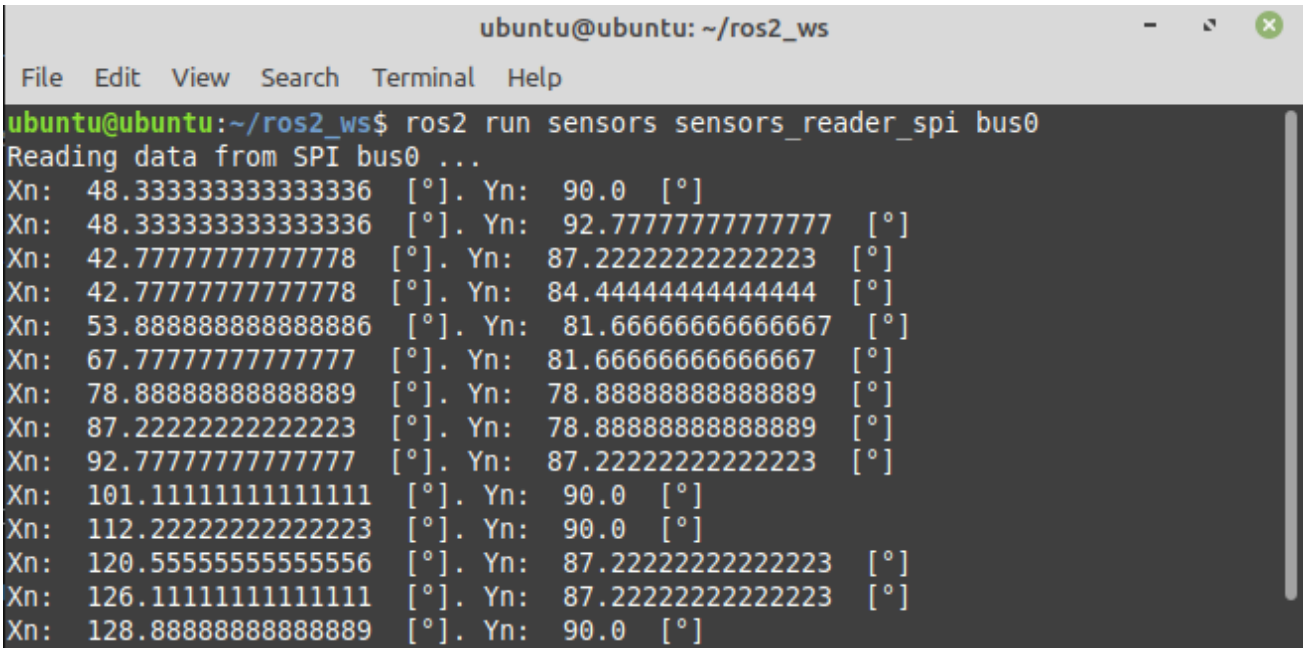

Figure 2. 20: Xn changing test

The results obtained are correct since Xn values are changing. Yn angle it is correctly maintained to a value of approximately 90°. The precision is about 10° since the light is moved by hand and some errors occur during the movement.
The same test is performed by moving the light along the Y axis to check if the Yn values change. The results are presented in Figure 2. 21:

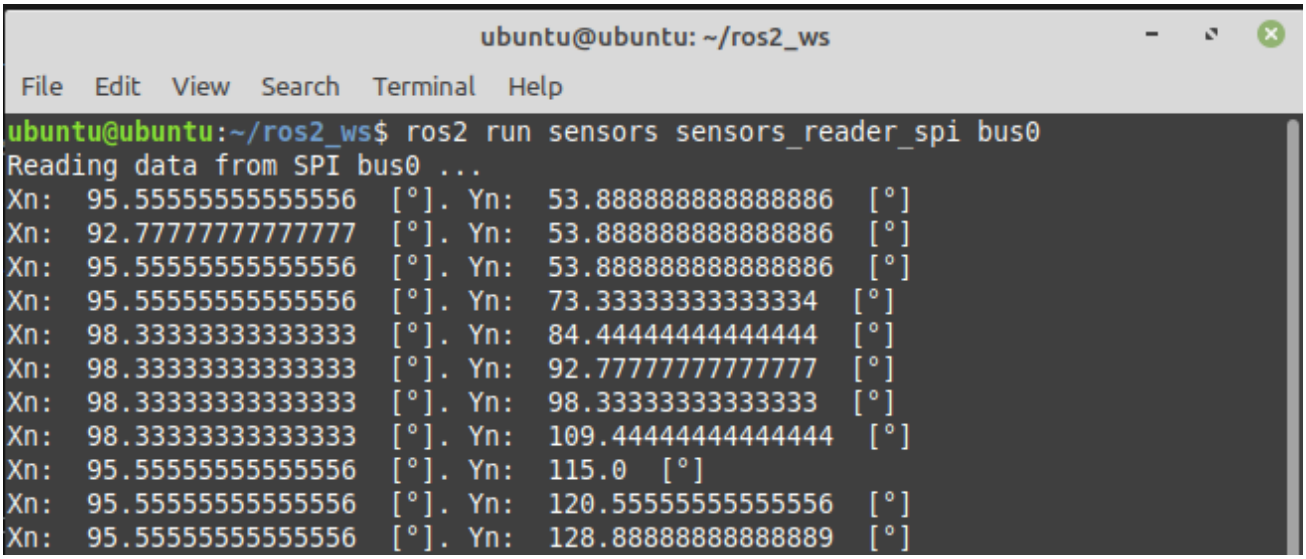

Figure 2. 21: Yn changing test

The results are correct and compatible with the ones obtained from the Xn moving test

2.2.7 Raspberry PI 3 B+

The Raspberry Pi 3 B+ is a single-board computer of small dimensions that can be equipped with different Linux based operating systems (mainly Raspbian and Ubuntu). The board doesn't have an integrated hard disk, so the installation of the operating system is done with the flashing from an SD card.
Raspberry is often used for academic usage but also in companies for rapid prototyping as control unit in projects of all size and application fields, mainly because is a low-cost board, is simple to configure and to use and has an high efficiency in terms of CPU consumption.
Considering the older models, Raspberry Pi 3 B+ has an extended GPIO (*General Purpose Input/Output*) with 40 pins. The board and its GPIO scheme can be seen in Figure 2. 22.

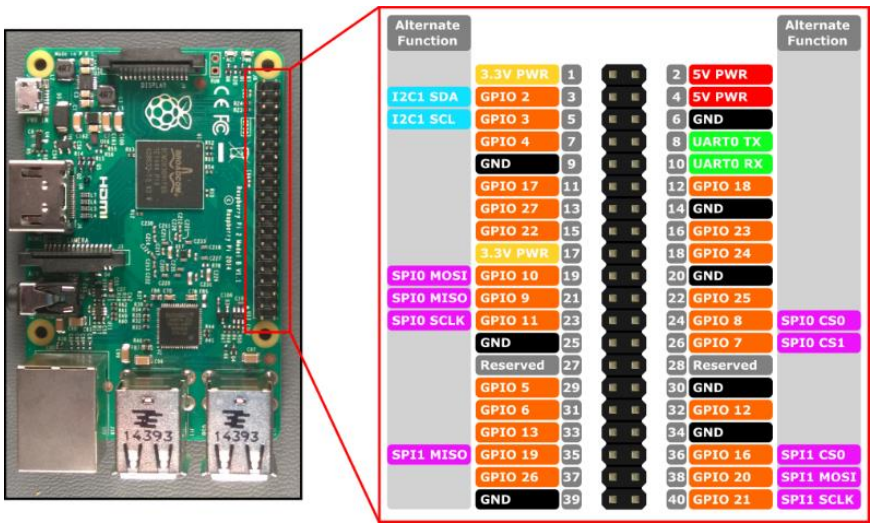

Figure 2. 22: Raspberry Pi 3 B+ board and GPIO scheme

For the aim of this thesis work, the connection of the following pins is necessary:

- Supply: Pins “1, 17” for the 3.3 V and pins “2, 4” for 5 V supply
- SPI communication: Pins “19, 21, 23, 24” in order to communicate through SPI protocol with the sun sensor mounted on the sensor module
- I2C communication: Pins “3, 5” in order to communicate through I2C protocol with the magnetometer and the temperature sensor mounted on the sensor module
- GND: Pins “14, 30, 34” are used for ground connection

These connections must be done like the representation in the schematic of Figure 2. 2 resulting in this real circuit presented in Figure 2. 23:

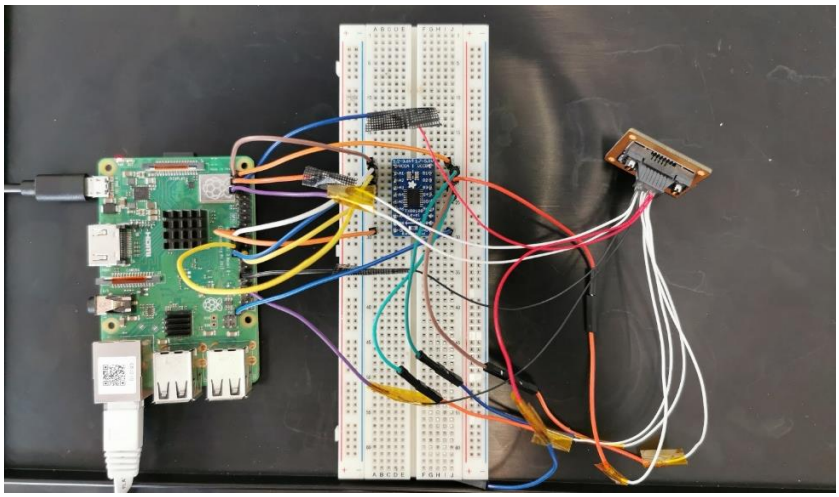

Figure 2. 23: Final circuit with: Raspberry PI, logic level converter and sensor module

2.3 Software configuration

In the following section is presented the software configuration used for developing the thesis work.
As presented in section 2.2.7, the used board for testing the ROS2 software is the Raspberry Pi 3 B+. The first step for starting to develop with an embedded system is to install an OS (operating system) suitable for the aim of the work. Generally, for what concerns embedded systems, there are two different possibilities for installing an OS:

- The first one is realizing an image, generally composed by bootloader, kernel and rootFS, with an automatized toolbox, like Buildroot or YOCTO, that generates *embedded linux* images and then flash it on the system following a certain procedure that may be different

from board to board .
Buildroot provides a graphical user interface which allows to select on a menu the bootloader, kernel, rootFS, predefined or custom packages and everything that we would need on our board . It may be a hard procedure to obtain a working image (specially for customized boards), but some boards may need this solution because of their strong customization.

- The second solution is to download an existing operating system (like Debian or Ubuntu) and then flash it on the board following the proper procedure. For example, with Raspberry is very easy since you can just upload the OS image on the SD and then insert it in the SD slot.

Since the purpose of this thesis is to develop a software framework based on ROS2, an OS image that has ROS2 installed is necessary.
To obtain this result, the first solution is not the preferred one since in order to have ROS2 on the image, according to ROS official installation page, the only available method is following the "build from source" procedure which means to download the ROS2 source code and then cross-compile it for the Raspberry Pi processor, which can be a difficult procedure to do (and not so intuitive).
Proceeding with the second solution because of its immediacy, once the operating system is downloaded and mounted on the SD card, is just a matter of following the procedure "Installing ROS2 via Debian Packages" described in the ROS official installation page. The only existing operating system that can support the last version of ROS2 (Foxy) is Ubuntu 20.04, so it's the one used for this thesis work.

2.3.1 ROS2 overview and advantages

The Robot Operating System (ROS) is not a real operating system as the name may suggest, but a set of software libraries and tools, also called *"middleware"*, for building robot applications. Since ROS was started in 2007, a lot has changed in the robotics and ROS community. and the goal of the ROS2 project is to adapt to these changes leveraging what is great about ROS1 and improving what isn't; the most interesting part of this updating procedure is that you can always connect the latest version of ROS2 in use with ROS1, with a mechanism called ***bridge***, in order to not lose any functionality neither of one nor the other.
ROS is heavily used in robotics, but it can be used in general for autonomous/semi-autonomous systems that may need to read sensors, have perception of their position and attitude in space and to control actuators. For these reasons it is a very good choice for developing a software framework also for aerospace applications, like nanosatellites.
In this thesis project the latest version of ROS2 is used and it is called *ROS2 Foxy Fitzroy*. There are many versions of ROS2 and most of them are constantly updated and supported until their EOL date (End of life); the actual situation is portrayed in Figure 2. 24:

Distro	Release date	Logo	EOL date
Foxy Fitzroy	June 5th, 2020		May 2023
Eloquent Elusor	Nov 22nd, 2019		Nov 2020
Dashing Diademata	May 31st, 2019		May 2021

Figure 2. 24: ROS2 latest distributions and EOL dates

Beyond the reasons explained above there are other benefits for using ROS:

- It is totally open-source and constantly updated by developers all around the world for many application fields.
- Creating truly robust, modular and efficient robot/mechatronics software is hard, so ROS provides plug-and-play solutions to common problems in developing software frameworks.
- Is based on the DDS standard for the managing of data distribution for real-time systems, that provides an easy publish-subscribe paradigm.
- Comes with many ready-to-use tools for debugging, data visualization and simulation.
- Possibility to develop software in python and C++ and to get connected with Matlab/Simulink.

Another great advantage of using ROS/ROS2 is the possibility to integrate a generic ROS system with MATLAB and Simulink by using the official ROS Toolbox. This feature is fundamental for the MBSD approach, addressed in the introduction, since the toolbox natively provides a function for autogenerating C++ code (with Simulink Coder), from Simulink models, for ROS/ROS2 nodes. The ROS Toolbox provides an interface able to connect MATLAB and Simulink with ROS and ROS2 enabling the creation of a distributed network of ROS/ROS2 nodes among the target embedded system, running the ROS software, and the local PC with MATLAB/Simulink.
The toolbox includes MATLAB functions and Simulink blocks to import, analyze ROS/ROS2 messages sent and received from specific topics.

At the heart of any ROS 2 system is the ROS graph. The ROS graph refers to the network of nodes in a ROS system and the connections between them by which they communicate. This graph is

made up by the elementary concepts of ROS that are:

- Nodes: are the smallest entities constituting every complex system. They can be seen as processes, intended for few and elementary operations, that can communicate with other nodes over topics. Each node can be a subscriber or a publisher of a certain topic.

- Topics: each topic has a name and a specific kind of message that it can handle. They are the "hubs" where messages are collected, when sent by publisher nodes, and sent to subscriber nodes (ref. Figure 2. 25)

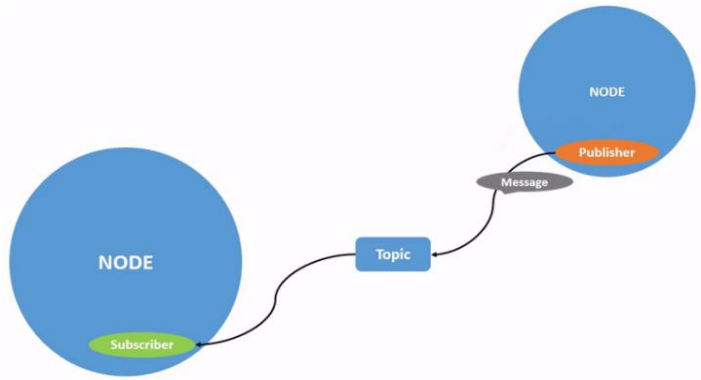

Figure 2. 25: Publisher "Node" sends a message over the topic "Topic"

- Services: another method of communication for nodes based on a call-and-response model. While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client. A representation of this system is presented in Figure 2. 26

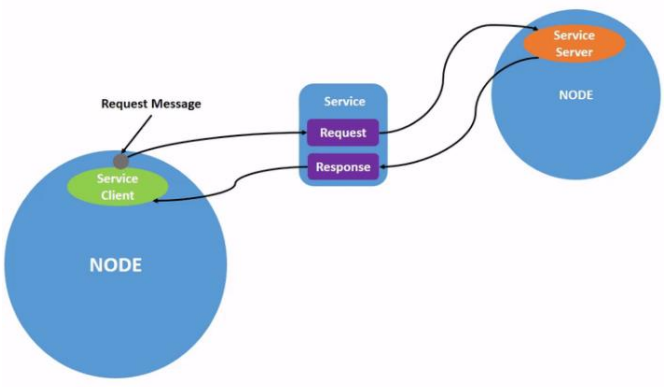

Figure 2. 26: Call-and-response method implemented by the service

By means of these simple components we can establish really complex systems like robots or even nanosatellites. At the end of our development , including sensors reading, control of actuators and storing of useful data, it is really helpful for debugging and analysis to represent the overall system in its nodes and topics using the ***rqt_graph.*** A simple but clear example of this functionality is represented in Figure 2. 27 extracted from the official ROS2 tutorial.

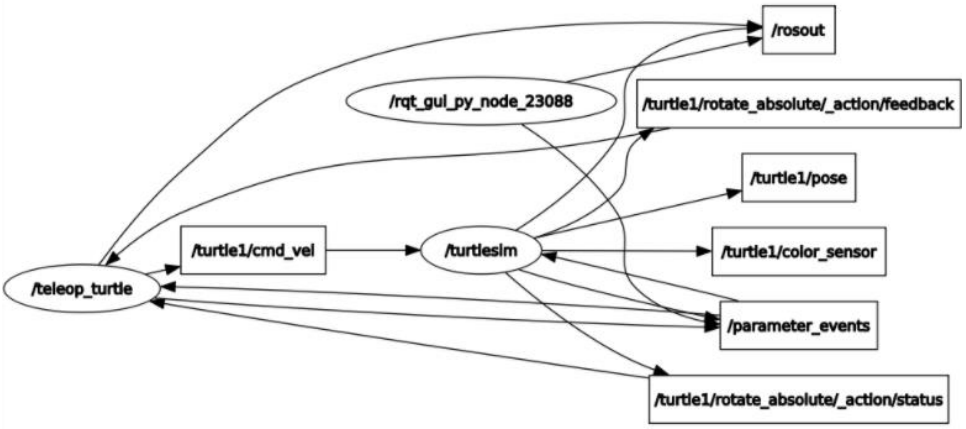

Figure 2. 27: rqt_graph of the official teleop turtle tutorial

3 ROS2 FLIGHT SOFTWARE FRAMEWORK

In this chapter is described the proposed solution for the fundamental nodes implemented for a draft of ROS2 based flight software framework. As stated in the introduction the applications selected are related to the sensor data reading and storing and to a Watchdog for monitoring the overall system status. These applications will be implemented as ROS2 nodes; all the details are reported in the following sections.

3.3 Watchdog node

The watchdog is an electronic or software timer that is used to detect and recover from system malfunctions, in order to make the whole system running properly. Particularly, its main duty is to check if the applications that it has to monitor are active and properly running or not and, in case they are not, to re-start them again.
In general a software watchdog is a process that perform these operations after being configured by reading the needed informations, contained in a specific configuration file (written in YAML, JSON or other data-serialization language), that watchdog reads when it is launched.
Is always a good safety precaution to have a software watchdog in an automatic system, but it is necessary in critical systems that must be active for a long period like nanosatellites since if a process crashes it's necessary to immediately re-start it, to not compromise all the system.
An example of watchdog application in a complex software framework is the one used on the MK-1 framework produced by Tyvak International. Its working flow is presented in Figure 3. 1.

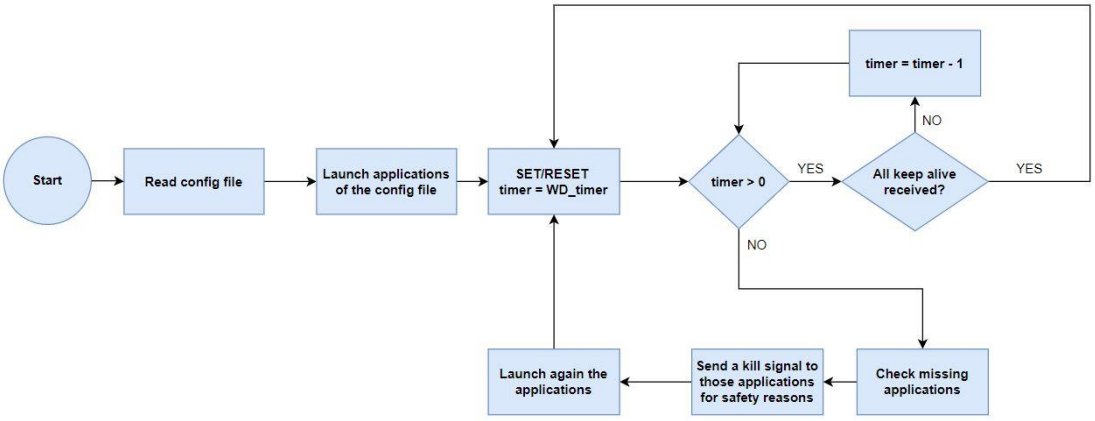

Figure 3. 1: Mark-1 watchdog flow chart

When the watchdog application is started, it reads a configuration file (written in YAML) and stores informations about the applications that it has to control, among other settings regarding the timer period and so on. These executables are then launched by the watchdog itself. Each application then is intended to send an heartbeat/*keep alive* message with a specified frequency in order to signal to the watchdog that is running correctly. To check this, an infinite loop with

the operations described below is performed:

- A watchdog timer with a specified frequency is set.
- If the timer is greater than zero, the watchdog checks if all the "*keep alive*" messages has been collected from the applications to be guarded. If this doesn't happen it decreases its timer, otherwise the timer is reset and the loop restarts.
- If the watchdog timer is equal to zero, it means that one or more applications did not send the "*keep alive*" message. This could happen for many reasons, for example the applications could be stuck in an infinite loop or it could be crashed.
  The watchdog checks the missing applications and it sends a *kill* signal to those processes for safety reasons. After that, it restarts the missing applications and resets the watchdog timer.

In the ROS2 developed framework, the working principle of the watchdog node is different since the desired application works mainly with pre-existent ROS2 API (*Application programming interface*). Since an API called *"get_nodes_names",* which returns a list with the names of the active nodes, is already existent in ROS2, the usage of the "*keep alive*" messages became useless for detecting which nodes are alive or not.
This gives an important advantage for the system communications because it reduces the amount of messages that a node has to send through topics. Moreover, in order to re-start the nodes that are not alive, the ROS2 *launch file* service is used.
ROS2 launch files are Python scripts that allow to start up and configure a number of executables containing ROS2 nodes simultaneously. These files include the package name and the executable name of the node to be launched, and other parameters like the arguments to pass at the launch command. They must be contained in a suitable "*launch*" folder and they can be executed through the "*ros2 launch*" command from a shell, but there is also a provided API called "*launch_a_launch_file*" that allows to launch other nodes programmatically, by passing as argument the path to the correspondent launch file of the desired node.
Attributes and methods of the Watchdog class are presented in Figure 3. 2:

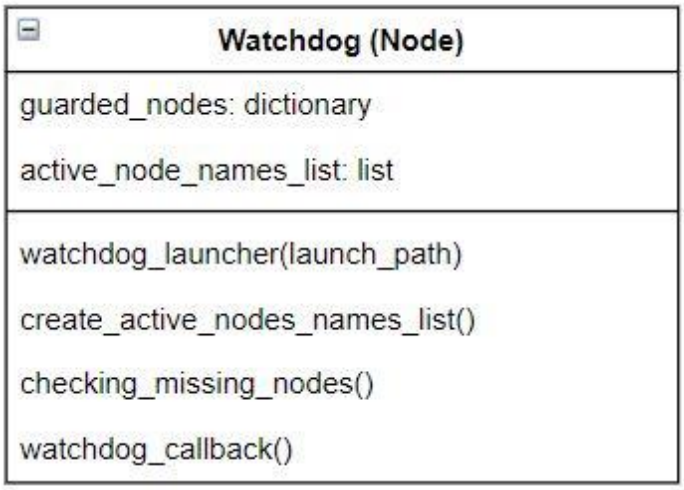

Figure 3. 2: Watchdog class

The flow chart of the developed ROS2 based watchdog is presented in Figure 3. 3:

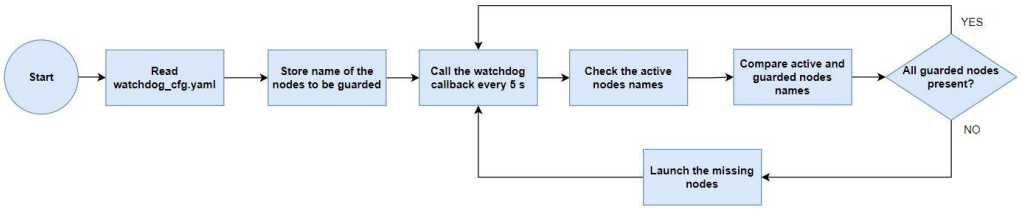

Figure 3. 3: ROS2 based watchdog flow chart

When the watchdog node is started, it reads the configuration file (written in YAML) in which are stored the names of the nodes to be guarded and the path to their launch file, and it stores the names in *"self.guarded_nodes"* field of the class. An example of the YAML file is presented in Figure 3. 4:

```yaml
guarded_nodes:
  node1:
    name: 'i2c_bus1'
    launch_path: '/home/ubuntu/ros2_ws/src/sensors/launch/i2c_bus1_launch.py'
  node2:
    name: 'spi_bus0'
    launch_path: '/home/ubuntu/ros2_ws/src/sensors/launch/spi_bus0_launch.py'
```

Figure 3. 4: Watchdog config YAML file

The YAML file is organized as a dictionary with a key called "*guarded_nodes*", which value is the list of the nodes to be guarded. Each node is a list itself that contains two keys: the name of the node and the path to its launch file.

The core function of the watchdog node is the "*watchdog callback*" which is called with a frequency of 5 seconds. When the callback is called, the Watchdog stores the list of the active nodes into the specific list, using the method "*create_active_nodes_name_list*" and the API "*get_nodes_name*" presented above. Then, a method called "*checking missing nodes*" is executed in order to compare the guarded nodes list and the active nodes one. If one ore more nodes are not present, the "*watchdog_launcher*" method is executed through a subprocess call (present in the *multiprocessing* Python library).
This method executes the launch file of the missing nodes using the API "*launch_a_launch_file*" presented above. Once these operations are done, the callback is called again after 5 s.
The Watchdog node can be executed through the "*ros2 run*" command via shell.
For the purpose of this thesis work, the nodes that are guarded by the watchdog are the sensors nodes presented in the following paragraphs.
Considering its implementation, the realized watchdog node does not acts like a publisher or a subscriber node but it is like a stand-alone node which autonomously controls the status of other important nodes, needed for the correct working of the whole system.

3.3.1 Watchdog Testing

In order to check the correct performances of the designed Watchdog node, some tests are performed. The first situation is the one in which the sensors nodes to be guarded are running, and the Watchdog needs only to print a message with a list of the active nodes. The results obtained from this scenario are presented in Figure 3. 5:

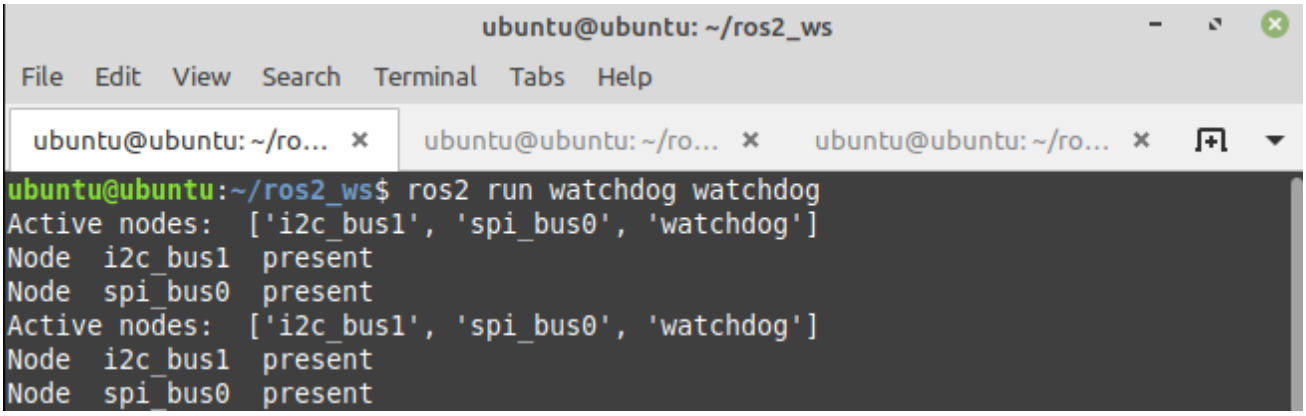

Figure 3. 5: Watchdog test. All nodes present

As it can be seen, Watchdog correctly print a list of the active nodes (including itself) and a message that shows that the sensors nodes are correctly running.

The second situation is the one in which one of the two guarded nodes (for example the one that read data from the SPI bus) is not running. The Watchdog is in charge of start this node up. The results are presented in Figure 3. 6:

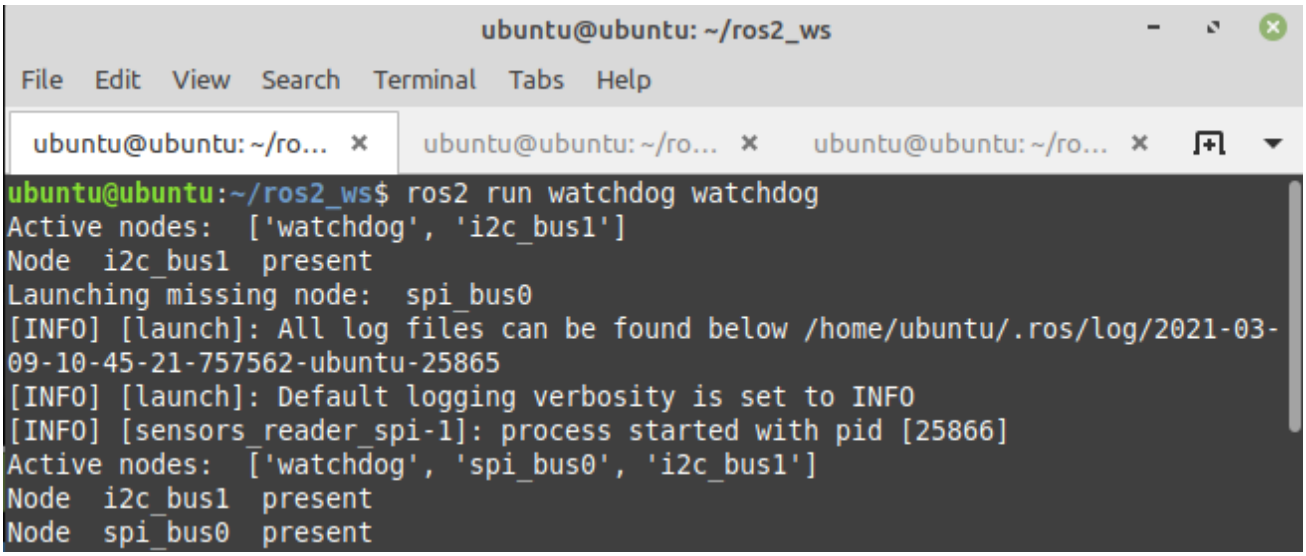

Figure 3. 6: Watchdog Test. SPI node missing

As it can be seen, when the Watchdog callback is called for the first time, the only node present in the active nodes list, except the Watchdog, is the one that read data from the I2C bus. For this reason, the Watchdog launches the SPI node and print an info message that contain the pid of the process started. After that, when the callback is called for the second time, all the nodes

are present in the list of active nodes and the execution process proceed normally.
The last scenario is the one in which both nodes are not present and Watchdog needs to start them up. This test is performed in order to check that the Watchdog can start more nodes simultaneously. The results are presented in Figure 3. 7:

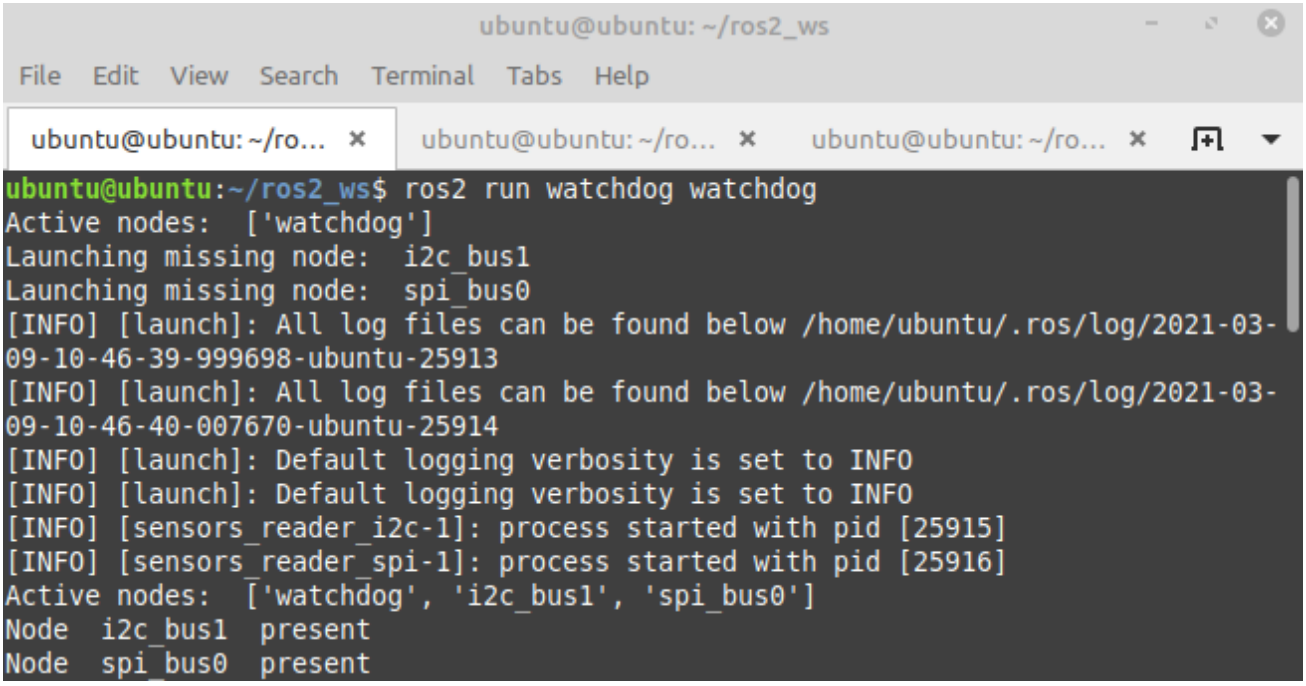

Figure 3. 7: Watchdog test. Both nodes missing

The obtained results are pretty similar to the ones of the previous test. Firstly only the Watchdog node is present and the sensors nodes are missing. So, the Watchdog start them up and print two messages with their pid. When the callback is called for the second time, all the nodes are correctly present and the execution process proceed normally.

3.4 Sensors Bus node

When we have different digital devices that need to communicate one with another, there is always a communication system that enables this data exchange.
In the case presented in this thesis there is a sensor module, instrumented with several sensors, that can communicate with an external device by means of dedicated buses, and in particular: the AD7415 temperature sensor and the HMC5883L magnetometer can be interfaced through an I2C bus, while the E910.86 sun sensor with an SPI bus.
The detailed description of these two communication systems is reported in the successive sections while here only the architectural choice of how the ROS2 framework will handle the sensors, and why, is discussed.

The first possible implementation that has been examined is also the most intuitive one: one ROS2 node for each sensor.
In this way is possible to obtain a very easy to visualize system where each node is referred to one single sensor and so it can be also easy to handle each sensor in different ways. But there are also two significant problems with this implementation that made the second solution to be

the best one.
Imagining a very usual situation like the one depicted in the following figure:

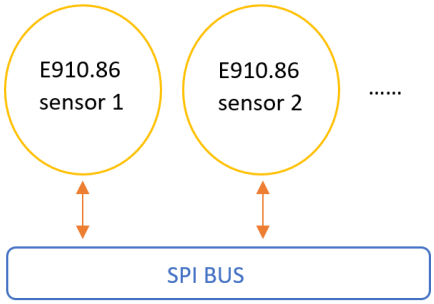

Figure 3. 8: SPI bus example with several identical sensors

where there are many identical sensors that have to perform exactly the same type of measurement and in the same manner, for example on a satellite we may have many sun sensors (such as in Figure 3. 8) or magnetometers collecting data for attitude determination; in cases like these the solution "one node one sensor" is not so optimal form the software engineer point of view because there will be many identical nodes performing exactly the same tasks and each one of them is implemented exactly in the same way.
This totally goes against the efficiency and reusability philosophy of ROS2 and object programming in general.
The second significant problem is related to the message traffic that our system would bear whenever each node, representing each sensor, have to send messages over topic, containing the collected data, at very high frequencies.

The second implementation analysed solves these two issues in this way: each node represents a particular bus used by many sensors.
Referring to the Figure 3. 8 in this implementation the node will represent the SPI bus and not each sensor attached to it, drastically reducing the redundancy of exactly the same piece of code. From the message traffic point of view the situation in improved because now the node representing the bus collects all the data from each sensor and then it works as an hub for sorting the messages and send them to the right topic, instead of having many nodes continuously sending messages at each collection of data.

For fully understand the differences between the two approaches we can consider a more realistic situation, as the one presented in Figure 3. 9:

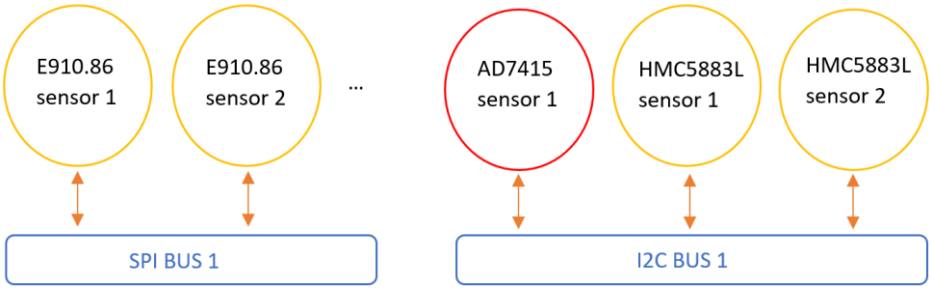

Figure 3. 9: Realistic situation with many sensors on two different buses

The first presented method for handling sensors with ROS2 node, would lead to have 5 nodes for collecting data coming from the sensors connected to different buses, while with the second solution only two nodes will be created.

3.4.1 I2C bus node

I2C (Inter Integrated Circuit) is a serial communication system used in embedded systems. It's a master/slave communication that normally have one master and one or more slaves. Each of them is recognizable by a unique hexadecimal address. The hardware protocol needs two serial lines for the communication: SDA (*Serial data*) for data and SCL (*Serial Clock*) for the clock (mandatory since I2C is a synchronous bus). Two other lines are used: one for the reference connection (called GND) and one for the voltage supply (typically 5 or 3.3 V). The hardware representation of the I2C protocol can be found in Figure 3. 10:

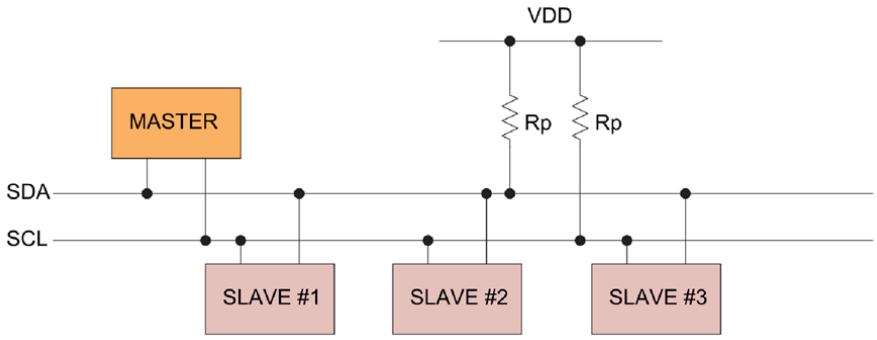

Figure 3. 10: I2C protocol representation

Considering the ROS2 based framework developed, one node for each I2C bus present on the used board is created. The node can be created with the command "*ros2 run sensors sensors_reader_i2c busN*" where N is the number of the bus that is wanted to be read. Raspberry Pi, used for this work, has only one I2C bus (*bus 1*) but other boards could have more than one bus so it's necessary to specify which bus is wanted to be read.
To handle the i2c communication, *smbus2* python library is used. It is the commonly used library for this kind of communication and it provides several useful functions to open/close the communication with a specified bus and read/write data to a specific slave address.
For what regard the purposes of this thesis work, two sensors communicate through I2C bus: an AD7415 temperature sensor and a HMC5883L magnetometer, both described in the previous paragraphers.
Since the I2C bus node must acts like a publisher and send a message that contains the sensors data read on a dedicated topic, a custom message that can contain these informations must be created. All the custom messages created for this thesis work are contained in a suitable folder. The structure of sensors message is presented in Figure 3. 11:

```
int32[2] temp_raw
float64 temp
int32[6] mag_raw
float64[3] mag
int32 sun_raw
float64[2] sun
```

Figure 3. 11: Sensors custom message structure

In the "*raw*" fields of the message are contained the raw values returned by the related sensor without any kind of conversions (binary value). The other fields of the message contain the data values of the related sensors that can be used for computation for other nodes of the system. Since all the possible kinds of sensors are present in the message and some of them may communicate through SPI protocol (they will be present in the following paragraph), their fields will always be empty when considering an I2C bus node. Otherwise, the I2C bus sensors fields will be empty when an SPI node is created.
Considering I2C bus node software, its flow chart is presented in Figure 3. 12:

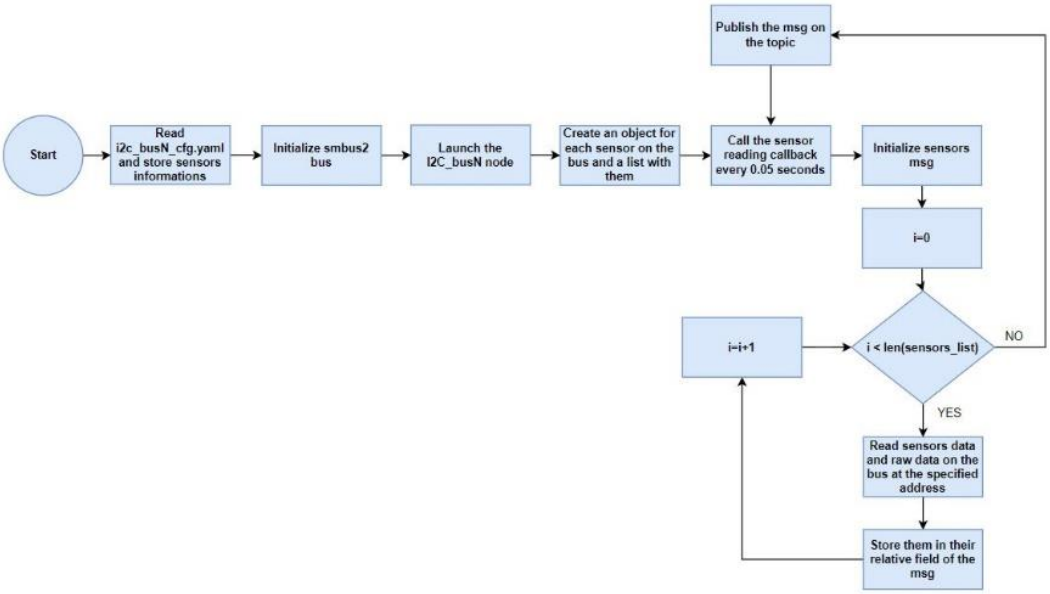

Figure 3. 12: I2C bus node flow chart

Its attributes and methods are then presented in Figure 3. 13:

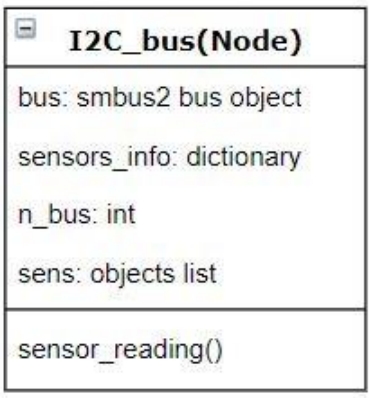

Figure 3. 13: I2C bus node class

After the node is launched, it reads the configuration file (written in YAML) presented in Figure 3. 14:

```
n_bus: 1
sensors:
  sensor1:
    type: 'temp'
    addr:  0x49
  sensor2:
    type: 'mag'
    addr:  0x1E
```

Figure 3. 14: I2C bus node YAML configuration file

Each bus is characterized by two keys: its number and a list of the sensors present on the bus. Each element of the list has two keys: the type of the sensor and its address on the I2C bus. The number of the bus and the list of sensors are stored in suitable python variables by scrolling the YAML file as a dictionary structure. The I2C bus is then initialized using the dedicated *smbus2* function and after that the node is created.
In the constructor of the I2C bus node, an object list of sensors is created by scrolling the list retrieved from the YAML file and creating an object for each of them.
The core function of the I2C bus node is the "*sensor_reading*" callback, called with a frequency of 0.05 seconds. Every time that this function is called, a new sensors message is initialized. A for loop is performed by scrolling on the list of sensors objects created in the constructor. The raw and data values are read and stored into the message related fields for each sensors.
The message is then published on the topic and the callback is called again after 0.05 seconds.

3.4.2 SPI bus node

The SPI protocol (Serial peripheral interface) is a serial communication protocol used for establishing a connection between microcontrollers or in general digital devices and, just like

the I2C system, it uses a master-slave paradigm. In this communication system we don't have an address for each slave, instead there is the chip/slave select signal that is used for identifying a slave among the others.
The SPI protocol connection between master and slaves is performed by four signal lines:

- SCLK: serial clock emitted by the master
- MISO: Master input slave output, that is the signal collecting data by the master
- MOSI: Master output slave input, like the MISO but in the inverse direction
- SS: Slave select, that is the signal emitted by the master for selecting the slave it wants to communicate with

The hardware representation of the SPI protocol is depicted in Figure 3. 15:

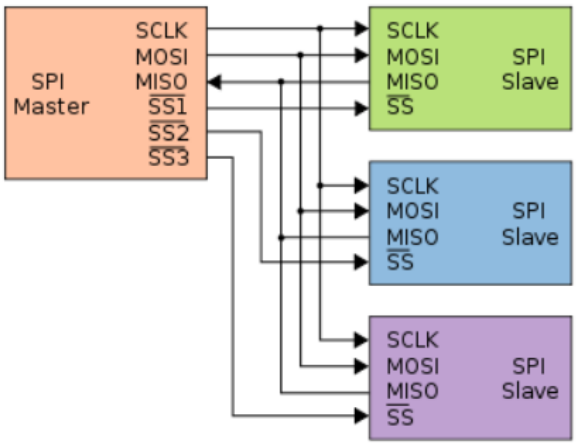

Figure 3. 15: SPI communication protocol example with a Master and three slaves

Just like the I2C bus node, the ROS2 framework can create a node representing a specific SPI bus. The node can be created with the command " *ros2 run sensors sensors_reader_spi busN* " where N is the number of the bus where there are sensors wanted to be read. For the Raspberry used in this project the SPI bus where the sun sensor is connected, is the number 0.
In order to access via software the SPI interface, the *spidev* python library is used.
For what concerns the message definition of the SPI bus node and the functional concept of the implementation, is possible to refer to the previous section (3.4.1 section) where all these details are presented and explained.

Considering the SPI bus node implementation, its class diagram and flow chart are presented in Figure 3. 16 and Figure 3. 17 below.

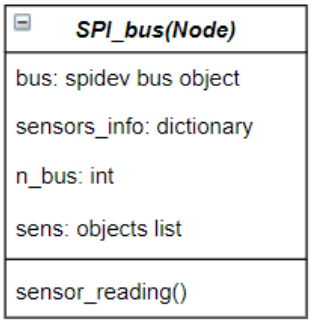

Figure 3. 16: SPI_bus node class diagram

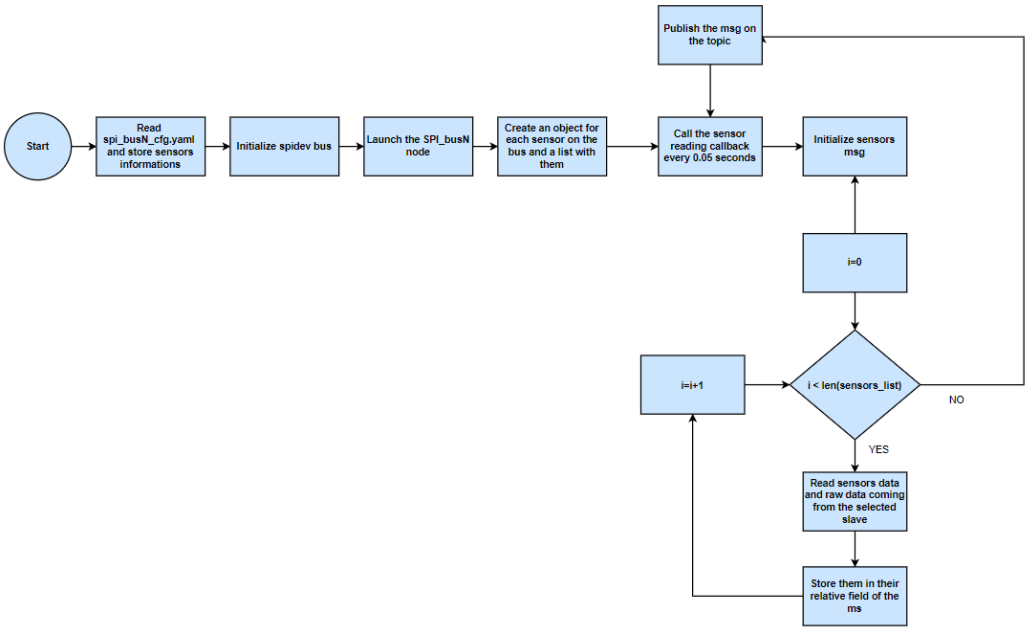

Figure 3. 17: SPI_bus node execution flowchart

As is possible to see the class diagram is the same as the I2C bus node and also the flowchart is actually very similar.
The main difference between an I2C bus node and an SPI bus node is in its config file, where instead of having an "*addr*" section now there is a "*cs*" section representing the chip select signal of the slave:

```
1  n_bus: 0
2  sensors:
3    sensor1:
4      type: 'sun'
5      cs:   0
```

Figure 3. 18: SPI_bus node configuration file

3.5 Sensors Telemetry node

The Telemetry is a technology that allows to measure and store informations of interest for the designer or operators who want to know relevant data of the system. Telemetry data can be sent in real-time, but they can also be collected in a suitable file (for example a binary file) and sent once the file has reached a defined size of after a certain amount of time. Telemetry is widely used in complex systems like nanosatellites for monitoring the status of its subsystems. In this way, they can send the most critical data (*downlink*) to ground operators who know how to interpret them.
For what concerns the ROS2-based software developed, the data that must be stored using telemetry are those that come from the sensors nodes described in the previous paragraphs.
A Telemetry node is created for each I2C or SPI bus to store all the sensors data that are present in that bus both in *raw* and *interpreted* form. When a predefined number of messages has been collected, a new telemetry file is created. All the sensors telemetry files are collected inside a folder called "*sensors_log*" inside the "*src*" folder of the telemetry package.
The files in which the data are stored can be created with different extensions. For what concerns this thesis work, two different approaches were implemented. The results are compared by means of the size of the produced files and then the smaller one is selected as the suitable one.

The first attempt was done by using database (*db3*) files that can be easily read by using a software that supports SQL files. The advantage of this kind of files is that they can be easily read by an operator since the data are organized in database tables. On the other hand, the produced files have a big size and, if the amount of data is large, the folder in which those files are contained can became very large.

The second attempt was done by writing the data on binary (*bin*) files. These files are not easy to read and the structure of the written data must be known a-priori, but they are compact and their size is almost the half of a db3 file so this choice was the used one. The name of the binary files is composed by the type of the bus (I2C or SPI), the number of the bus and a timestamp with date and creation time. The structure of an I2C or SPI bus telemetry node is the same; the only thing that changes are the sensors that are present on the bus and so the kind of data stored. The attributes and methods of an I2C or SPI bus telemetry node are presented in Figure 3. 19:

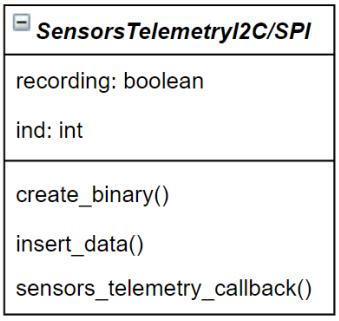

Figure 3. 19: I2C/SPI bus sensors telemetry class

The flow chart of an I2C/SPI bus telemetry node is shown in Figure 3. 20:

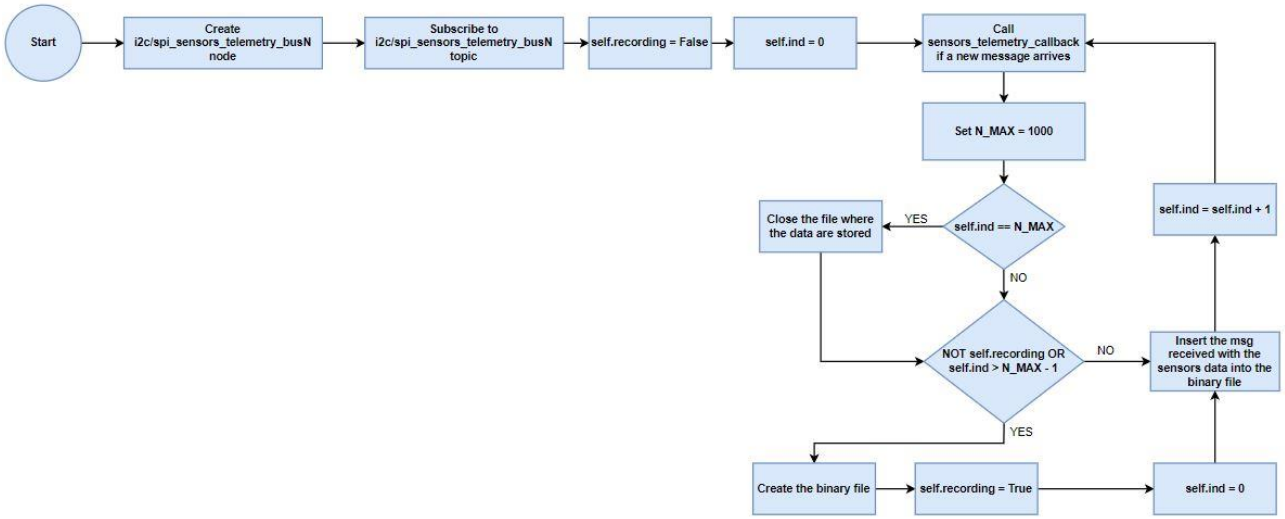

Figure 3. 20: I2C/SPI bus sensors telemetry class

A telemetry node can be created by using the shell command "*ros2 run telemetry sensors_telemetry_i2c/spi busN*" to start recording data of the sensors present on the I2C or SPI specified bus.
The created node acts like a subscriber on the topic where the specified bus publishes its data. Once the node is created and the subscription to the topic has been done, a boolean variable "*recording*" is initialized to check if the desired topic is already recorded. Particularly, if the variable is set to False the topic is not recorded, otherwise it is recorded. Another variable "*ind*" is initialized to zero and it is used to count the number of messages arrived.
The "*sensors_telemetry_callback*" is called every time a new message is published on the desired topic by the related sensors node. When the callback is called, a variable "*N_max*" is set to define the maximum number of messages to collect inside a binary file and, once this number of messages is reached, a new binary file is created.
The operations performed when the callback is called are:

- Checking if the actual value of "*ind*" is equal to "*N_max*". If yes, it means that the maximum number for a binary file is reached so the binary file is closed.
- Checking if "*ind*" is greater than "N_max" -1 or if the topic is already recorded by using the variable "*recording*". If yes is necessary to: create a new binary file, set the recording value to true and reset "*ind*" to zero
- The message received is then written inside the binary file using the Python library "*struct*".

After that, the "*ind*" variable is increased by 1 and the callback is called again when a new message arrives on the topic.

3.5.1 Telemetry node testing

Since the behaviour is the same for both I2C and SPI nodes, only the I2C telemetry node is presented in this paragraph. In order to check that a new file is created every time that the maximum number of messages is reached, the "*N_MAX*" variable is set to 5 in order to rapidly check the correct behaviour. The output obtained is presented in Figure 3. 21:

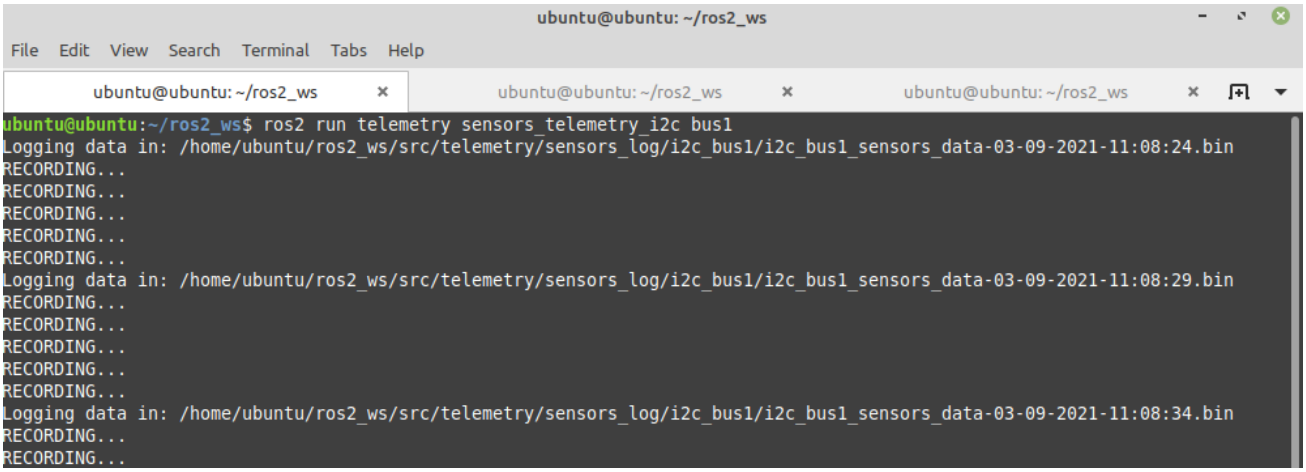

Figure 3. 21: Telemety node testing

The first line shows the creation of the first file in which the data of the I2C sensors are stored. After that, a "*Recording…*" message is printed every time a new message is stored in the file. Once the "*N_MAX*" number of messages is reached, a new file is correctly created and filled with the new messages.
In order to demonstrate that the data are stored correctly, a Python file is prepared to read the created binary files. This script uses the "*unpack*" function of the "*Struct*" Python library.
The data read from the script are presented in Figure 3. 22:

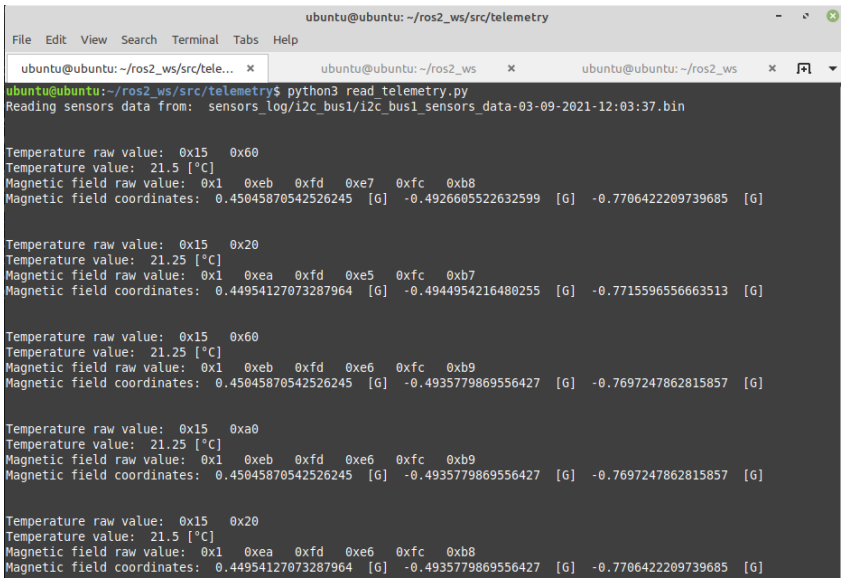

Figure 3. 22: Reading Telemetry data

4 ATTITUDE DETERMINATION

When a spacecraft or in general an autonomous system must perform some actions and interact with an environment, there is always the problem of determining its position in the space and its attitude. These two informations are fundamental and need to be mathematically defined with respect to a well-defined reference frame.
In this thesis only the attitude information is needed for performing the attitude determination, so the position in space of our system is neglected.
In the following sections the mathematical tools for determining the attitude of our spacecraft are presented.

4.1 Rotation matrices and quaternions

Let's suppose that we are in a situation like the one depicted in Figure 4. 1:

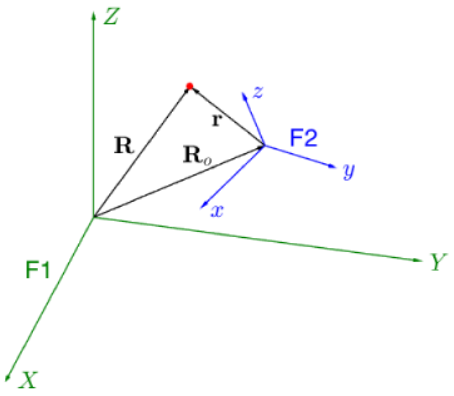

Figure 4. 1: F1, F2 reference frames and a generic particle

There is a generic particle (in red in the figure) and two reference frames (F1, F2) that are translated and not aligned, so a mathematical tool for representing the relative position and attitude between them is needed.
To this aim is possible to analyze the situation by representing the position of the particle with respect to the two reference frames:

$$\mathbf{R} = X\mathbf{I} + Y\mathbf{J} + Z\mathbf{K} \quad \text{position of the particle in F1}$$
$$\mathbf{R}_o = X_o\mathbf{I} + Y_o\mathbf{J} + Z_o\mathbf{K} \quad \text{position of the origin of F2}$$
$$\mathbf{r} = x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \quad \text{position of the particle in F2}$$

Figure 4. 2: Position of the particle with respect to F1, F2

The mathematical tool needed is such that it can represent the relationship between the coordinates (X,Y,Z) and (x,y,z). To this aim is possible to rewrite each coordinate of **R** in this way:

$$X = \mathbf{R}\cdot\mathbf{I} = (\mathbf{R}_o + \mathbf{r})\cdot\mathbf{I} = X_o + x\mathbf{I}\cdot\mathbf{i} + y\mathbf{I}\cdot\mathbf{j} + z\mathbf{I}\cdot\mathbf{k}$$
$$Y = \mathbf{R}\cdot\mathbf{J} = (\mathbf{R}_o + \mathbf{r})\cdot\mathbf{J} = Y_o + x\mathbf{J}\cdot\mathbf{i} + y\mathbf{J}\cdot\mathbf{j} + z\mathbf{J}\cdot\mathbf{k}$$
$$Z = \mathbf{R}\cdot\mathbf{K} = (\mathbf{R}_o + \mathbf{r})\cdot\mathbf{K} = Z_o + x\mathbf{K}\cdot\mathbf{i} + y\mathbf{K}\cdot\mathbf{j} + z\mathbf{K}\cdot\mathbf{k}$$

$$\Downarrow$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_o \\ Y_o \\ Z_o \end{bmatrix} + \mathbf{T}\begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad \mathbf{T} \doteq \begin{bmatrix} \mathbf{I}\cdot\mathbf{i} & \mathbf{I}\cdot\mathbf{j} & \mathbf{I}\cdot\mathbf{k} \\ \mathbf{J}\cdot\mathbf{i} & \mathbf{J}\cdot\mathbf{j} & \mathbf{J}\cdot\mathbf{k} \\ \mathbf{K}\cdot\mathbf{i} & \mathbf{K}\cdot\mathbf{j} & \mathbf{K}\cdot\mathbf{k} \end{bmatrix}$$

Figure 4. 3: R written in matrix form in function of (x,y,z)

As is possible to see from the relation above (Figure 4. 3), each element of the T matrix is a dot product between the F1 and F2 versors, that are called the *direction cosines*. These elements represent the orientation of each axis of one frame with respect to each axis of the other one, and due to this the T matrix is usually called *Direction Cosine Matrix* (DCM). An interesting feature deriving from this analysis is that is possible to split the problems of translation and rotation and to treat them independently, since the *T* matrix is referred only to the rotation while the R_0 vector is only referred to the translation between the reference frames.
The DCM *T* can be interpreted in two ways, and is fundamental to understand which interpretation is being used:

- Alias: is referred to the transformation of coordinates. For example *T* can be interpreted as a coordinate transformation $F_2 \rightarrow F_1$.
- Alibi: is referred to the rotations. For example *T* can be interpreted as the rotation matrix such that $F_1 \rightarrow F_2$.

The rotation matrices are a minimal and useful mathematical tool that can be easily employed for representing the attitude of a spacecraft, but their affected by a well known and dangerous limitation. Since matrices are used for representing the actual attitude of a generic system, it happens that in certain configurations the matrix loses a degree of freedom. In these situations, there is a *singularity* corresponding to the loss of an information, and that's exactly what happens when the so called *Gimbal-lock* occurs. This problem can be overcome by using non-minimal representations of the attitude.

A possible alternative to the DCM is the *quaternions*. They are mathematical objects used as a generalization of complex numbers to a 3D space, but they can also be used for representing rotations. They're based on the Euler's theorem and the elements of the quaternion are four variables called *Euler parameters*, that are used for describing a rotation around a specific axis. The advantages with respect to other representations are:

- Efficiency from a computational point of view

- Less sensitive to rounding errors
- Gimbal-lock avoided since it is a non-minimal representation

A quaternion can be written using these notations that are equivalent:

$$\begin{aligned}
\mathfrak{q} &= q_0 + \mathbf{q} \\
&= q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k} \\
&= \cos\tfrac{\beta}{2} + \mathbf{u}\sin\tfrac{\beta}{2} \\
&= e^{\mathbf{u}\frac{\beta}{2}} \\
&= \left(\cos\tfrac{\beta}{2}, u_1\sin\tfrac{\beta}{2}, u_2\sin\tfrac{\beta}{2}, u_3\sin\tfrac{\beta}{2}\right) \\
&= (q_0, q_1, q_2, q_3) \\
&= (q_0, \mathbf{q}) = \begin{bmatrix} q_0 \\ \mathbf{q} \end{bmatrix} = \begin{bmatrix} \cos\frac{\beta}{2} \\ \mathbf{u}\sin\frac{\beta}{2} \end{bmatrix}
\end{aligned}$$

Figure 4. 4: Quaternion equivalent notations

The q_0 is the real part of the quaternion while the $\mathbf{q}$ is its imaginary part, when the real part is null the quaternion is said to be pure. The $\mathbf{u}$ and β are respectively the axis of rotation and the angle around the body is rotating, that can be found by applying the *Euler's theorem* computing the eigenvalues and eigenvectors of the rotation matrix describing the rotation.
Let's now introduce some properties and algebra related to quaternions:

- The null quaternion is such that its real and imaginary part are null
  The identity quaternion is such that the real part is $q_0 = 1$ while the imaginary part is null.
- The complex conjugate of a quaternion is just like the quaternion but with the imaginary part sign inverted: $\mathbf{q}_{conj} = -\mathbf{q}_{init}$.
- The products involving quaternions are the following:

Quaternion product (Hamilton product)

$$\begin{aligned}
&\mathfrak{q} \otimes \mathfrak{p} = (q_0 + \mathbf{q}) \otimes (p_0 + \mathbf{p}) = \dots \\
&= (q_0 p_0 - \mathbf{q}\cdot\mathbf{p}) + (q_0\mathbf{p} + p_0\mathbf{q} + \mathbf{q}\times\mathbf{p})
\end{aligned}$$

dot product

$$\mathbf{q}\cdot\mathbf{p} = \textstyle\sum_{i=1}^{3} q_i p_i$$

cross product

$$\mathbf{q}\times\mathbf{p} = \begin{bmatrix} q_2p_3 - q_3p_2 \\ q_3p_1 - q_1p_3 \\ q_1p_2 - q_2p_1 \end{bmatrix}$$

Figure 4. 5: Algebra of quaternions

- Given a rotation defined by a quaternion, is possible to represent the inverse of the rotation by computing the conjugate of the quaternion.

With the properties listed above, quaternions are a suitable non-minimal representation of rotations that are widely adopted nowadays for defining the attitude of complex systems like robots, spacecrafts and so on.
Is also possible to pass from a representation to the other by using the proper formulas:

Quaternions $\rightarrow$ DCM:

$$\mathbf{T} = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 + q_0 q_2) \\ 2(q_1 q_2 + q_0 q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2 q_3 - q_0 q_1) \\ 2(q_1 q_3 - q_0 q_2) & 2(q_2 q_3 + q_0 q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}$$

DCM $\rightarrow$ quaternions ($q_0 \neq 0$):

$$q_0 = \frac{1}{2}\sqrt{T_{11} + T_{22} + T_{33} + 1}$$

$$\mathbf{q} = \frac{1}{4q_0} \begin{bmatrix} T_{32} - T_{23} \\ T_{13} - T_{31} \\ T_{21} - T_{12} \end{bmatrix}$$

Figure 4. 6: DCM ↔ Quaternions formulas

4.2 Reference Frames

A reference frame is specified by an ordered set of three mutually orthogonal, possibly time dependent, unit-length direction vectors. In order to describe the orbital motion of satellites around the Earth, there exist a set of standardized coordinate reference frames that can be used. The most relevant ones are:

- ECEF (*Earth Centred Earth Fixed*): also known as conventional terrestrial system, the point (0, 0, 0) denotes the centre of the Earth. X-Y plane is coincident with the equatorial plane and its versors point in the directions of longitude 0° and 90° while the Z axis is orthogonal to them and points in direction of the true North Pole. The ECEF frame is presented in Figure 4. 7:

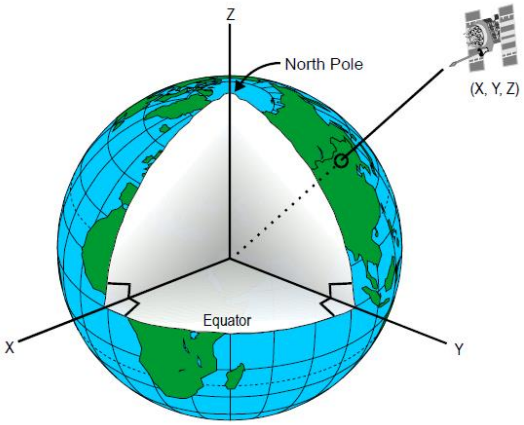

Figure 4. 7: Representation of ECEF frame

This frame rotates tied with the Earth so rotate with respect to the stars. It is a non inertial, accelerated frame. It is commonly used to describe motion of objects on Earth surface.

- ECI (*Earth Centred Inertial*) frame: has its origin at the centre of mass of Earth like the ECEF frame and its axis lays on the same plane of the ECEF frame but it is fixed with respect to the stars and inertial (non accelerated). An equinox occurs when the earth is at a position in its orbit such that a vector from the earth toward the sun points to where the ecliptic intersects the celestial equator. The equinox that occurs near the first day of spring is called the vernal equinox. It can be used as a principal direction for ECI frame. It is useful to describe the motion of celestial bodies and spacecraft. The location of an object can be defined by using right ascension and declination (spherical coordinates like longitude and latitude) or using Cartesian coordinates. One commonly used ECI frame is defined with the Earth's Mean Equator and Equinox at 12:00 Terrestrial time on 1 January 2000 and is called *J2000.* The x-axis is aligned with the mean equinox and z-axis is aligned with the Earth's rotation axis.

- LVLH (*Local vertical, local horizontal coordinates*) are a geographical coordinate system based on local vertical direction and Earth's axis of rotation. The axes are positioned as follows: one axis is on the northern axis, one along the local eastern axis and on represents the vertical position. If the third axis is positive when it points up the frame is called *ENU* (East North Up), otherwise is called *NED* (North East Down). These frames are used to represent state vectors (set of data that describe where an object is located in space). A representation of an *ENU* frame with respect to the ECEF is presented in Figure 4. 8:

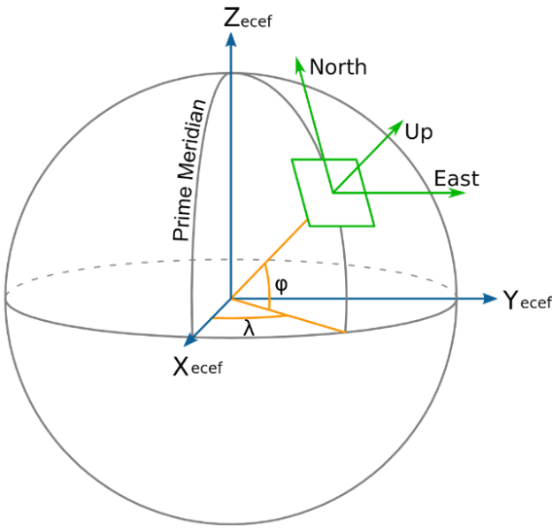

Figure 4. 8: ENU frame with respect to ECEF

- *Body-fixed* frames are tied to a named body and rotate with it. The axes can be placed as wanted. Considering the system of this thesis work, the body frame considered is the one used by the sun sensor E910.86 to provide the sun coordinates and it is presented in Figure 4. 9.

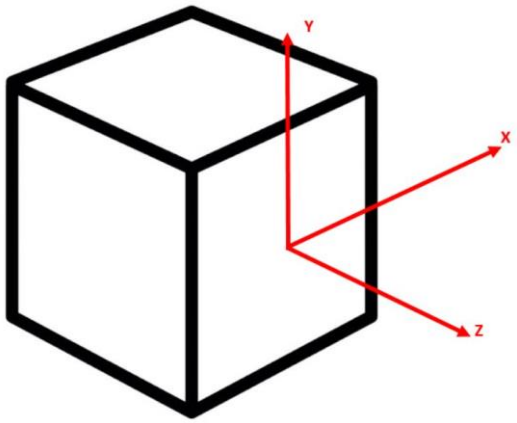

Figure 4. 9: Body frame used representation

The z-axis is pointing out of the sensor body, the y-axis points up with respect to it and x-axis is orthogonal to them.

4.3 General overview of AD systems

The attitude determination is one of the fundamental problem of an aerospace system and the first step that is needed to realize and ADCS application.
Determine the attitude of a spacecraft means identify its orientation in space with respect to a specific reference frame (normally ECEF or an ECI frame) by using the measurements obtained from suitable sensors mounted on the spacecraft in order to know its orientation with respect to relevant objects in the space like the Earth, the Sun or relevant star constellations. Particularly, the objective is determine the attitude matrix A that describes the rotation from the considered frame (normally ECEF or ECI) to the body frame of the spacecraft.
To obtain the attitude determination of an object in the space, many kinds of sensors can be employed. The most commonly used are star trackers, magnetometers and sun sensors. These sensors are widely used since they express the position of the body frame with respect to relevant objects in space like the Earth magnetic field , the Sun or star constellations positions.
Attitude determination problem can be divided in two main categories:

- **Static attitude determination**: all measurements are taken at the same time. The problem becomes up of optimally solving the geometry of the measurements.
- **Dynamic attitude determination**: measurements are taken over time. Is a much harder problem since the informations collected need to be blended together by using mathematical tools like Kalman filter.

For the lack of simplicity, since the objective of this thesis work is to demonstrate that a ROS2 framework can be applied to nanosatellite using an attitude determination application, static attitude determination is considered.
The first problem to obtain an attitude determination is to determine how many measurements (unit direction vectors) are needed to identify the orientation. Unit vectors are considered since the length of the measurements do not provide any useful information.
If a 2D attitude problem is considered as starting point, the answer is that only one unit direction vector (with the unit length constraints) provides all the required information to determine the orientation.
Considering the 3D problem, one measurement is no more sufficient so, a minimum of two observation vectors are required. With only one measurement, a rotation about that axis cannot be sensed. Measuring a second direction fix the complete three dimension orientation in space.
To demonstrate this consider the example in Figure 4. 10:

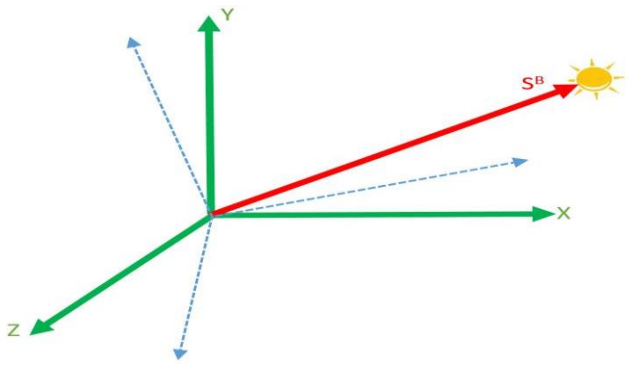

Figure 4. 10: Explanation of attitude determination

Suppose that the sun position is measured using a sun vector and it is expressed in the body frame by the unit direction vector S^b. Knowing only this information is not sufficient since an infinite number of orientations around this axis are possible solutions for the attitude problem. Considering the example of Figure 4. 10, the blue and the green frame can be both solution of attitude problem since they can express the same sun position vector with different orientations. Two unit vectors measurements determine the attitude matrix but, in fact, they overdetermine it. That's because the spacecraft attitude is represented by a 3x3 orthogonal matrix A such that $A^TA = I$ and $\det(A)=1$. This means that A is a rotation matrix and so an element of the three-parameter group SO(3).
Euler's theorem states that the general motion of a rigid body with one fixed point is a rotation about some axis. This shows that SO(3) is a three-parameters group (the three parameters can be taken as the rotation angle) and two parameters specifying a unit vector along the rotation axis.
To obtain a complete attitude determination, it is also necessary to know the components of the two measured vectors in some reference frame like ECEF or ECI.

In this thesis work, the measurements used are the ones provided by a sun vector and a magnetometer to respectively obtain the informations about Earth magnetic field and Sun position. The attitude problem considered is to determine the orientation of the spacecraft with respect to the ECEF frame, so the corresponded informations about the Earth magnetic field and Sun position expressed in this frame are taken into account and they will be explained in the following paragraphers.
The following notation will be used:

- S^b: sun unit vector expressed in the body frame
- m^b: Earth magnetic field unit vector expressed in the body frame
- S^{ECEF}: sun unit vector expressed in the ECEF frame
- m^{ECEF}: Earth magnetic field unit vector expressed in the ECEF frame

4.4 IGRF Earth magnetic field

The Earth magnetic field is a magnetic dipole with the magnetic field S pole coincident with the Earth geographic north pole and the magnetic field N coincident with the Earth geographic south pole. Its representation is presented in Figure 4. 11:

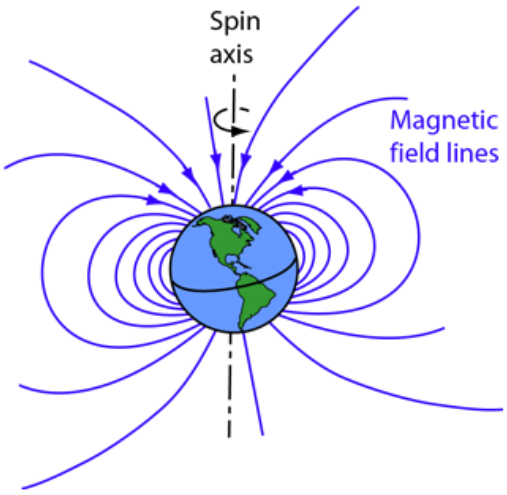

Figure 4. 11: Earth magnetic field representation

The magnitude of the Earth magnetic field at its surface ranges from 25 to 65 μT and it is mostly caused by electric currents in the liquid outer core.
The currents that create the magnetic field started up millions years ago. Magnetometers can detect minute deviations in the magnetic field caused by iron artifacts, some kind of stone structures and archaeological geophysics.
In order to represent the Earth magnetic field and its secular variations, different kind of mathematical models are proposed. The used one for this thesis work is the *International Geomagnetic Reference Field* (IGRF). This model was obtained by combining data and informations from many satellites and research institutes from all around the world. The last version released is the 13th generation and it is provided by the International Association of Geomagnetism and Aeronomy (IAGA). The magnetic field can be calculated as the negative gradient of a scalar potential V which can be represented by a truncated series expansion:

$$V(r,\vartheta,\Phi,t) = a\sum_{n=1}^{N}\sum_{m=0}^{n}(\frac{a}{r})^{n+1}[g_n^m(t)\cos(m\Phi) + h_n^m(t)\sin(m\Phi)]P_n^m\cos(\vartheta)$$

Where a is the mean radius of the Earth (approximately 6371,2 km), g_n^m and h_n^m are the Gauss coefficients (available in tabular form), r,ϑ,Φ are the spherical coordinates of the observation point and $P_n^m\cos(\vartheta)$ is the Legendre associated function of order n and m.
To easily compute in Python the value of the Earth magnetic field, a library named "*pyIGRF*" is provided by IAGA. Particularly, this function "*pyIGRF.igrf_value*" requires as input the latitude, the longitude and the altitude of the observation point and the current date. The latitude and the longitude are expressed in geocentric coordinates. Taking as reference Figure 4. 12:

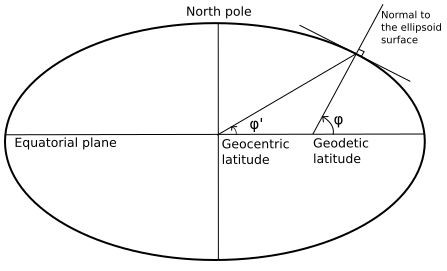

Figure 4. 12: Geocentric and geodetic coordinates

The considered latitude and longitude are expressed as geocentric coordinates i.e. their angles are measured with respect to the centre of the Earth.
The "*pyIGRF.igrf_value*" function provides as output:

- D: declination (positive east) i.e. the angle between the magnetic north and the true north
- I: inclination (positive down) i.e. the angle between the horizontal plane and the total field vector
- H: horizontal intensity
- X: north component expressed in NED coordinates
- Y: east component expressed in NED coordinates
- Z: vertical component (positive down) expressed in NED coordinates
- F: total intensity unit in nT

The informations about the X, Y and Z coordinates with respect to NED frame are the one used for the realization of attitude determination and their usage will be explained in the next paragraph.

The output results of the Earth magnetic field coordinates (i.e. M^{ECEF} vector) are presented in Figure 4. 13:

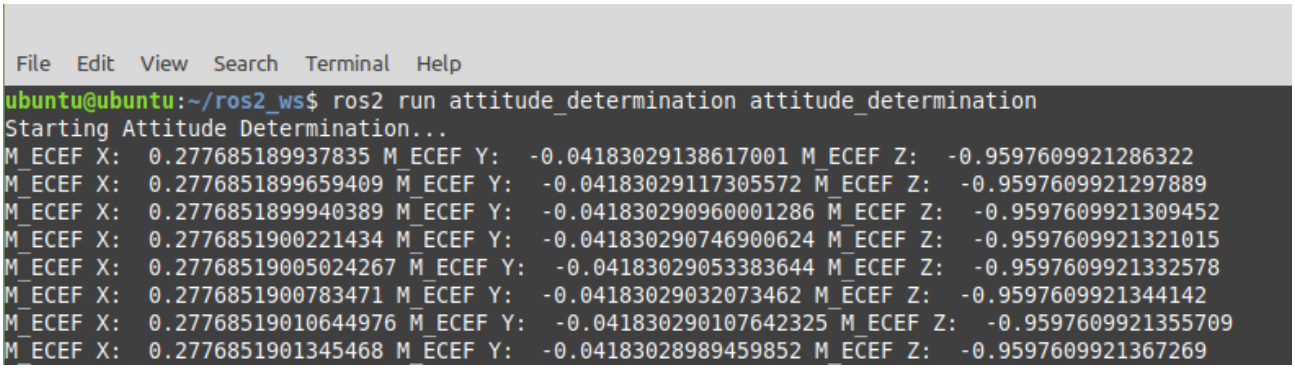

Figure 4. 13: M_ECEF coordinates

4.5 Sun position vector in ECEF frame

In order to retrieve the informations about the sun position with respect to the ECEF frame (i.e. S^{ECEF}) to achieve the attitude determination, the Skyfield Python library is used.
Skyfield is a library developed by Rhodes Mill and it is widely used to computes positions for the stars, planets and satellites in orbit around the Earth.

The first step to compute the sun position is to obtain the Sun and Earth ephemeris. These are tables that contain values, computed in a particular range of time, of different astronomical quantities like magnitudes, orbital parameters, coordinates or distances from planets.
That can be easily done in Skyfield by downloading the "*de421.bsp*" file released by the JPL's Guidance, Navigation and Control section. This file contains the main ephemeris of all the planets of the Solar system and it can be easily loaded in a python script by using the "*load_file*" function provided by Skyfield. The output of this function is a dictionary structure. For example, if the earth ephemeris are needed, and the output of the "*load_file*" function is stored in the dictionary named "*planets*", the result can be easily obtained as "*earth = planets ['earth']*".
The computation of the sun vector need also the informations about the actual date-time expressed in UTC. Conventionally, in the astronomic field, the date-time is expressed using the Julian date (which express the number of days spent from 1st January 4713 a.C.).
The Timescale object returned by "*load.timescale*" function in Skyfield, manages the conversions between different time scales. The supported ones are UTC, UT1 (Universal time), TA1 (International Atomic Time), TT (Terrestrial time) and TDB (Barycentric Dynamical Time). By using the "*now*" function Skyfield can retrieve the current Julian date expressed in UTC.
Once these informations are known, the Sun position expressed in ECEF frame can be easily retrieved with a few lines of Python code presented in Figure 4. 14:

```
# S_ECEF computation
apparent = earth.at(t).observe(sun).apparent()
sun_info = apparent.frame_xyz(framelib.itrs)
S_ECEF=numpy.array(sun_info.au)
```

Figure 4. 14: S_ECEF computation in Skyfield

The first line of code computes the Earth position with respect to the Sun position using the functions "*at*" to compute the Barycentric position at the specified Julian date and "*apparent*" to compute the apparent position. The result is in geocentric coordinates i.e. with respect to the centre of the Earth.
The second line set the reference frame as ITRS (International Terrestrial Reference Frame).
The obtained results are expressed in au (Astronomical unit) and are stored into an array by using the "*numpy*" library.

The output results of the Sun position coordinates (i.e. S^{ECEF} vector) are presented in Figure 4. 15:

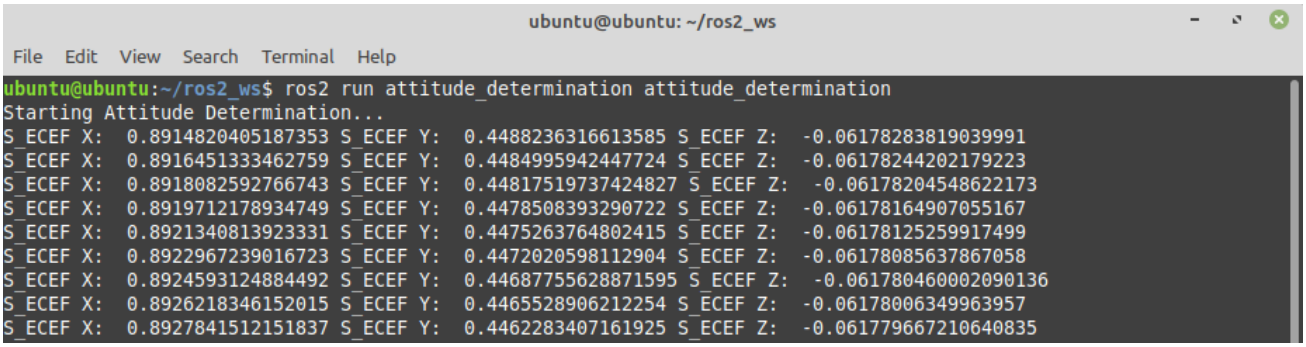

Figure 4. 15: S_ECEF coordinates

4.6 TRIAD algorithm

With the knowing measuremnt S^b, m^b, S^{ECEF} and m^{ECEF}, the attitude problem can be resolved by considering the TRIAD (*three-axis attitude determination*) algorithm. TRIAD algorithm is one of the simplest attitude algorithm and it can provide good results and solve the Wahba's problem which is intended to find a rotation matrix (particularly an orthogonal matrix) between two coordinate systems from a set of vector observations.
The algorithm considers that one vector measurement is more accurate than the other one. Since the sun vector is more precise than the Earth magnetic field vector, it is taken as the more reliable one. The result of TRIAD algorithm is the attitude matrix A that computes the rotation from the ECEF frame to the body frame of the nanosatellite.
The algorithm is intended to computes two matrices t^b and t^i that represent two support reference frames that combined allow to obtain the attitude matrix.
The operations to obtain the column vectors of these matrices is presented below:

- The first column vector of the two matrices is equal to the more accurate measurement, in this case the sun vector. In this case:

$$t_1^b = S^b \qquad t_1^i = S^{ECEF}$$

- The second column vector of the two matrices is chosen to be perpendicular to both measured vectors. In this case:

$$t_2^b = \frac{S^b \times m^b}{\|S^b \times m^b\|} \qquad t_2^i = \frac{S^{ECEF} \times m^{ECEF}}{\|S^{ECEF} \times m^{ECEF}\|}$$

- The third column vector of the matrices is computed as:

$$t_3^b = t_2^b \times t_1^b \qquad t_3^i = t_2^i \times t_1^i$$

Finally, the two 3x3 computed matrices are:

$$t^b = [t_1^b \; t_2^b \; t_3^b] \qquad t^i = [t_1^i \; t_2^i \; t_3^i]$$

By noting that $[t_1^b \; t_2^b \; t_3^b] = A_{3\times3}[t_1^i \; t_2^i \; t_3^i]$, the attitude matrix can be computed as:

$$A_{3\times3} = [t_1^b \; t_2^b \; t_3^b] \; [t_1^i \; t_2^i \; t_3^i]^T$$

The attitude matrix A is in the form of a DCM (*direct cosine matrix*). After that, it can be easily convered into a quaternion due to their advantages presented in the paragraph 4.1.
A graphical representation of t^b is presented in Figure 4. 16 to better clarify the physical meaning of the constructed frames.

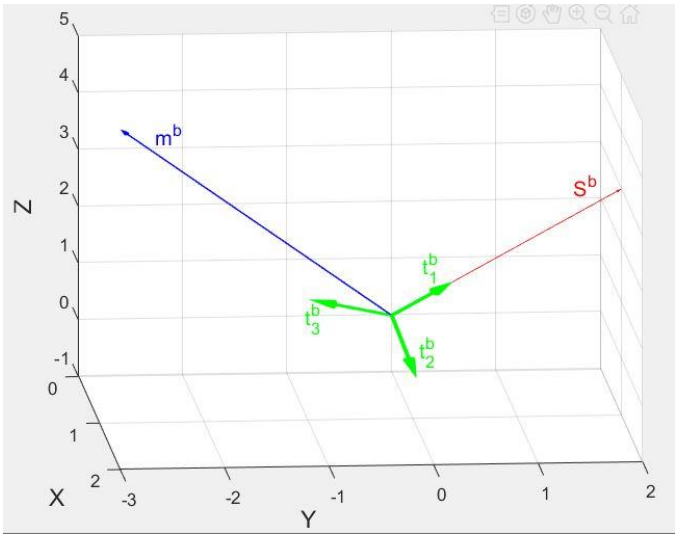

Figure 4. 16: TRIAD algorithm reference frame

The same representation can be obtained for the frame t^i by considering S^{ECEF} and m^{ECEF} instead of S^b and m^b.

4.7 Attitude determination node

In this paragraph, the realization of the attitude determination using the TRIAD algorithm explained in the previous sections is implemented into a ROS2 node.
The basic principle is the same as the other ROS2 nodes described in the chapter dedicated to the flight software framework. The realized attitude determination node acts like a subscriber to the topics where the sensors data of the I2C/SPI buses are published but it also acts like a publisher on a dedicated topic where it publish the attitude quaternion that describe the rotation from the ECEF frame to the body frame at each time that the callback is called. These data are needed to realize the Matlab real-time simulation that will be presented in the next chapter.
The class diagram that contains the attributes and methods of the node is presented in Figure 4. 17:

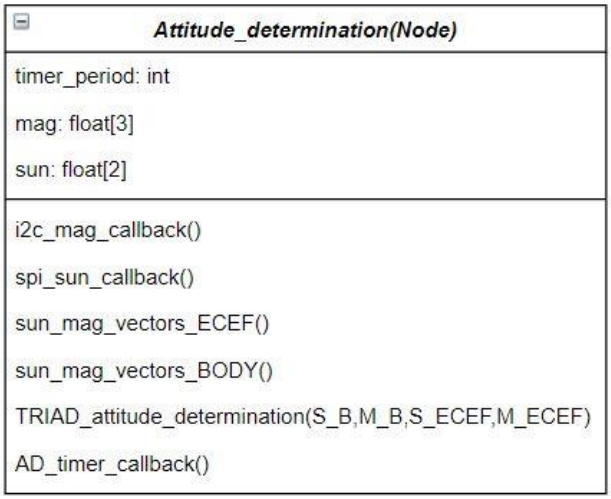

Figure 4. 17: Attitude determination class diagram

The attributes that are necessary for the implementation of the node are:

- integer "*timer_period*" to set the frequency of the attitude determination callback (which is set to 2 Hz i.e. 0.5 seconds)

- float array of three elements to store the Earth magnetic field vector coordinates provided by the magnetometer (expressed in its reference frame) each time that a new sensor data is published

- float array of dimension two to store the XZ and YZ angles provided by the Sun sensor (expressed in the body frame) each time that a new sensor data is published.

The methods that have been implemented are:

- *i2c_mag_callback()* and *spi_sun_callback()* to store the last data published from the magnetometer and the sun sensor in the related topics into the attributes *mag* and *sun*.

- *sun_mag_vectors_ECEF()* and *sun_mag_vectors_BODY()* to calculate the sun vector and the Earth magnetic field vector in both ECEF and body frame.

- *TRIAD_attitude_determination()* to compute the attitude determination through TRIAD algorithm using the informations of the sun and Earth magnetic field vectors both expressed in ECEF and body frame.

The flow chart of the ROS2 node realization is presented in Figure 4. 18:

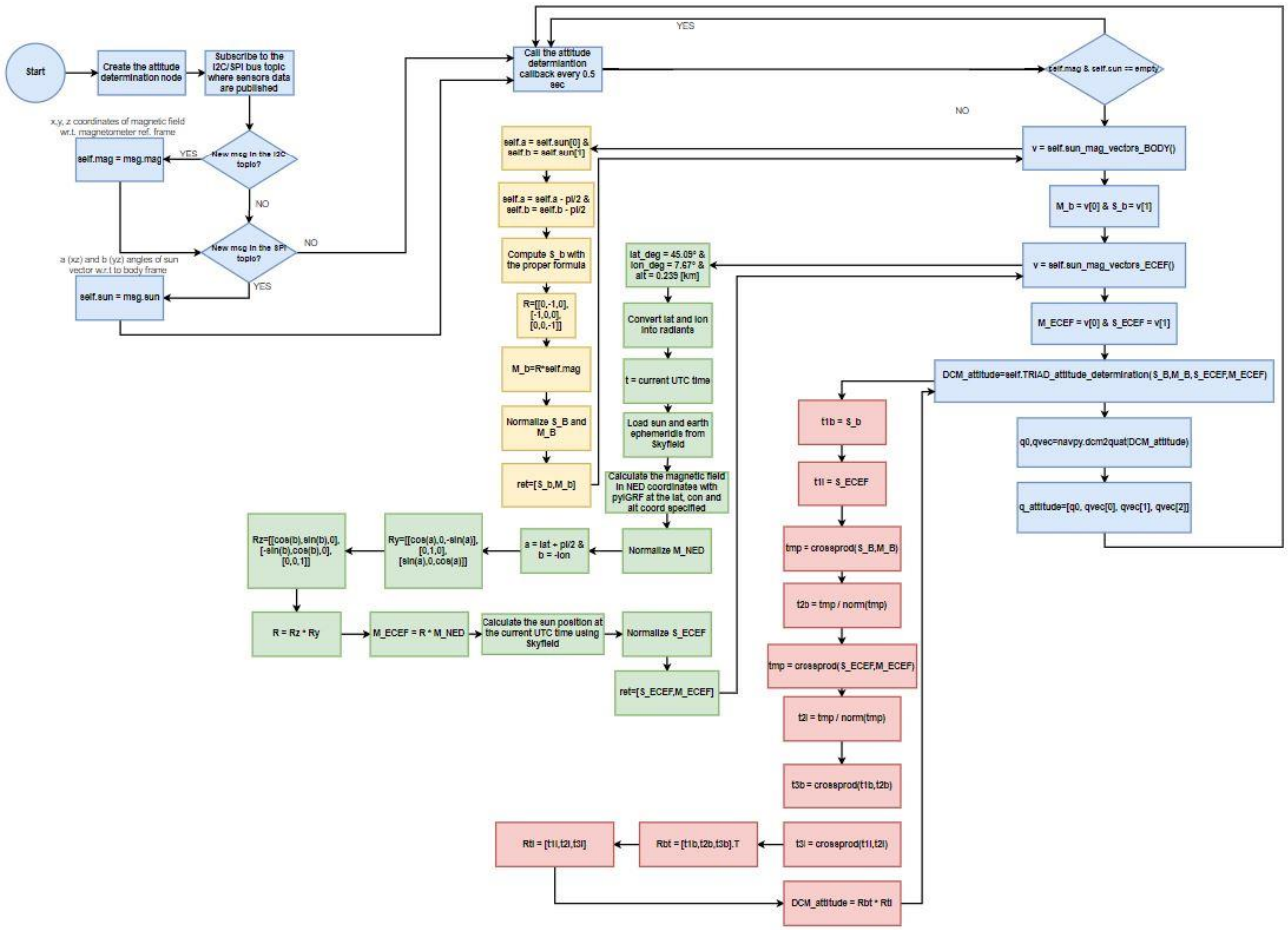

Figure 4. 18: Attitude determination flow chart

After that, the attitude determination callback is called with the specified frequency. The first step is to check if *self.mag* or *self.sun* are not empty. If at least one of them is empty, the attitude determination cannot be performed since one information from the sensors is not present yet. Otherwise, all the informations have been collected and the computations can start.
The first step is to compute the sun vector S^b and the Earth magnetic field vector m^b (both expressed in the body frame) by calling the method *sun_mag_vectors_BODY()*. The operations performed by this method are presented in Figure 4. 18 colored in yellow.

The value of the XZ-plane and YZ-plane angles provided by the sun vectors are stored into two variables (named *a* and *b*). The coordinates of the sun vector S^b are retrieved from these two angles with the method proposed in paragraph 2.2.5.
For what concerns the Earth magnetic field vector m^b, it must be noted that this measurement provided by the magnetometer and S^b of the Sun position provided by the Sun sensor are not expressed in the same reference frame. The representation of the two reference frames is presented in Figure 4. 19:

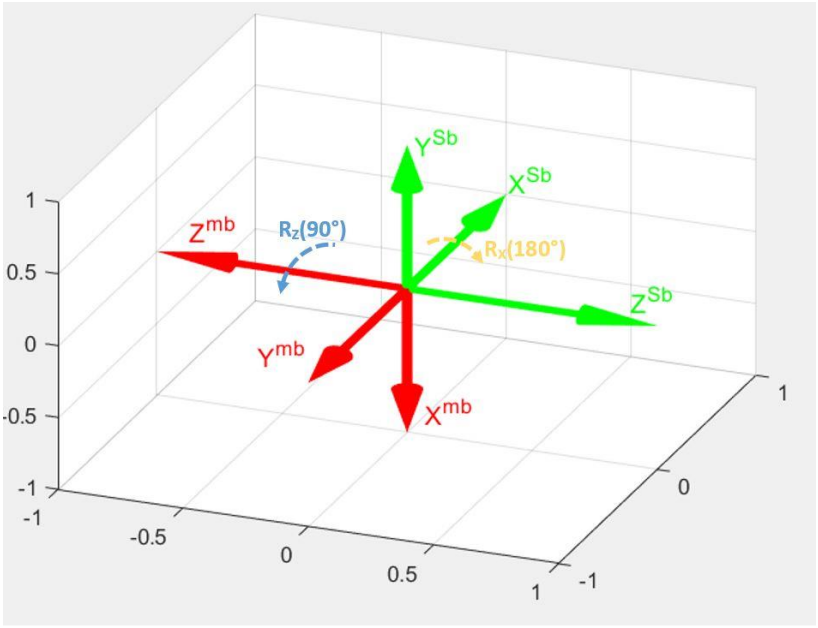

Figure 4. 19: Representation of magnetometer and sun sensor reference frame

Since the sun sensor frame (represented in green in Figure 4. 19) is chosen as body frame, the measurement of the magnetometer (which reference frame is represented in red in Figure 4. 19) needs to be expressed in this frame. In order to do this, two rotations are performed:

- A rotation around the Z axis of the magnetometer reference frame of 90° to align the X axis of this frame with the one of the body frame
- After that a rotation around the X axis of the obtained frame of 180° is performed to align the obtained frame with the body frame

The resulting rotation matrix that represents the rotation from magnetometer frame to body frame is:

$$R_{mag\_frame}^{body\_frame} = Rot_x(180°)Rot_z(90°) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Since a transformation of coordinates is performed (and not a rotation) the transposed resulting matrix is applied. Finally, the Earth magnetic field vector expressed in body frame coordinates can be easily computed as:

$$m^b = R_{mag\_frame}^{body\_frame^T} m^{mag\_frame}$$

Once S^b and m^b are computed, the resulting vectors are normalized.
After that, the computation of the corresponding Earth magnetic field and sun vector expressed in ECEF frame is performed using the realized method *sun_mag_vectors_ECEF().* The operations performed in this method correspond to the green blocks of the attitude determination flow chart. The first step is to define the variables needed for the computation. The computation of the Earth magnetic field require the knowledge of the altitude, the latitude and the longitude (for this work is used the coordinate of Tyvak International offices, otherwise the in-orbit coordinates are needed if the nanosatellite is flying) while the Sun vector requires the current time (expressed in UTC) and date.

After that, the NED coordinates of the Earth magnetic field (expressed in nT) are computed using the library pyIGRF as explained in paragraph 4.4. These coordinates need to be converted into ECEF coordinates. The NED and ECEF frames are presented in Figure 4. 20:

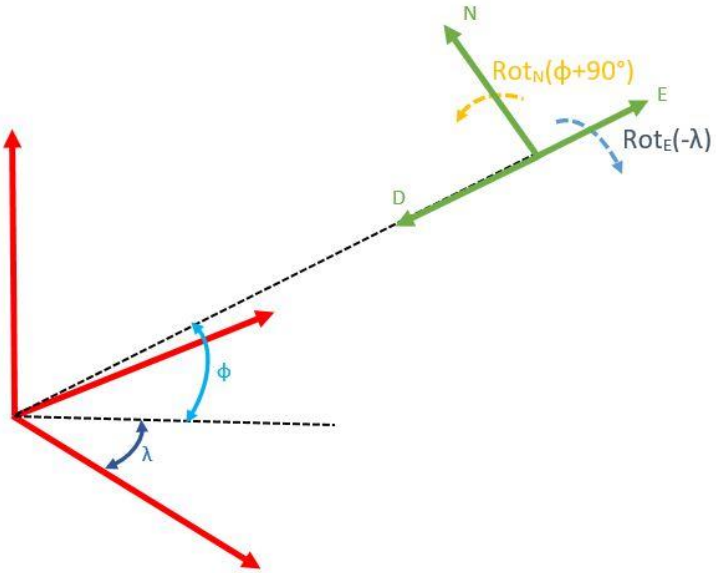

Figure 4. 20: NED to ECEF frame

The angle Φ represents the latitude while the angle λ represents the longitude. In order to convert the NED coordinates of the Earth magnetic field into ECEF coordinates the following rotations are performed:

- A rotation around the E axis of the NED frame of an angle Φ+90° in order to align the N axis with the ECEF frame

- A rotation around the N axis of the obtained frame of an angle – λ in order to align the frame obtained from the previous rotation with the ECEF frame

The resulting matrix that represents the rotation from NED frame to ECEF frame is:

$$R_{NED\_frame}^{ECEF\_frame} = Rot_N(-\lambda)Rot_E(\Phi + 90°) =$$
$$= \begin{bmatrix} \cos(-\lambda) & \sin(-\lambda) & 0 \\ -\sin(-\lambda) & \cos(-\lambda) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\Phi + 90°) & 0 & -\sin(\Phi + 90°) \\ 0 & 1 & 0 \\ \sin(\Phi + 90°) & 0 & \cos(\Phi + 90°) \end{bmatrix}$$

Since a transformation of coordinates is performed (and not a rotation) the transposed resulting matrix is applied. Finally, the Earth magnetic field vector expressed in ECEF frame coordinates can be easily computed as:

$$m^{ECEF} = R_{NED\_frame}^{ECEF\_frame^T} m^{NED\_frame}$$

The sun vector coordinates S^{ECEF} can be easily computed with Skyfield as explained in paragraph 4.5. Finally, m^{ECEF} and S^{ECEF} vectors are normalized.
After that, all the informations to obtain the attitude determination through TRIAD algorithm

are achieved so the method *TRIAD_attitude_determination()* can be called by passing as arguments S^b, m^b, S^{ECEF} and m^{ECEF}.
The operations performed by this method are just those explained in the paragraph 4.6 related to the TRIAD algorithm and are represented by the red blocks in the attitude determination flow chart. The output of this function is the attitude matrix in DCM form.
Once the DCM attitude matrix is obtained, it is converted into an attitude quaternion by using the *dcm2quat* function provided by the *navpy* Python library.
Finally, the results is published on the dedicated topic and it will be retrieved by Matlab as it will be explained in the next chapter.

4.8 Attitude determination node testing

The attitude determination application require the informations from both I2C and SPI sensors to achieve a result using TRIAD as it is presented in Figure 4. 21, that shows the architecture of the designed framework using rqt graph.

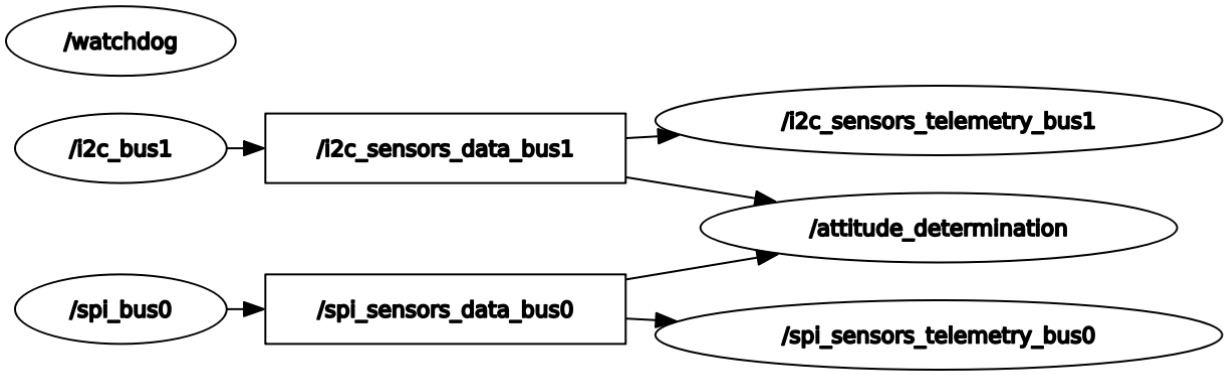

Figure 4. 21: Architecture of the framework

Once the two nodes that read the sensors data on the SPI and I2C buses are started, the attitude determination application can compute the quaternion (from ECEF frame to body frame).
The results obtained are presented in Figure 4. 22:

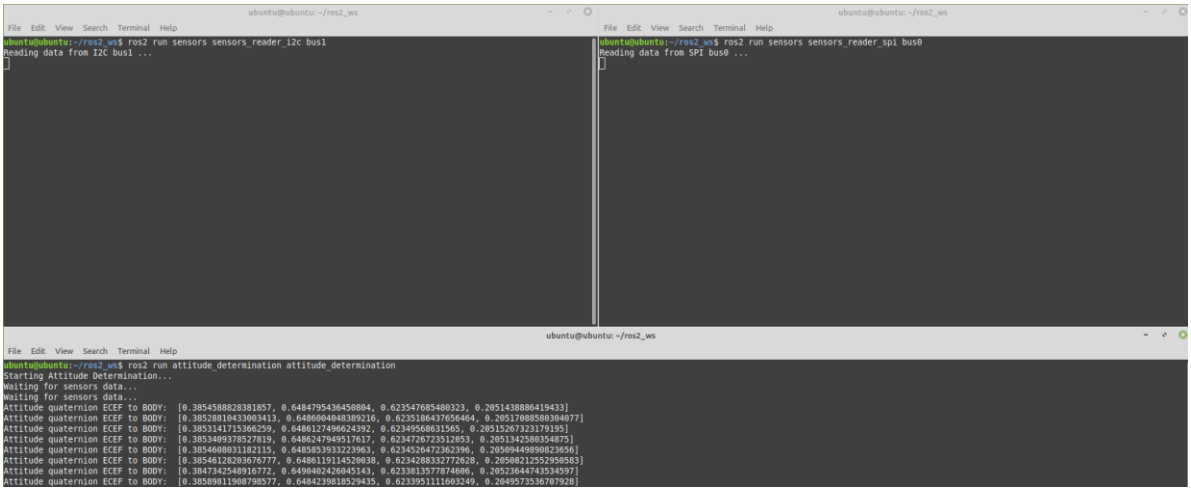

Figure 4. 22: Attitude determination testing

As it can be seen, in the first two shells of the picture are present the nodes that are intended to read sensors data while, in the third shell, is performed the attitude determination. Once the node is started, a message is printed to confirm that the node has been created. Until both data from magnetometer and sun sensor are not present, a "*Wait for sensors data…*" message is printed. Once at least one message is received from the sensors, the attitude determination is performed and the quaternion is correctly printed and published on a dedicated topic.

5 REAL-TIME MATLAB ANIMATION

In this chapter of the thesis is presented a real-time animation that represent the attitude determination of a 3U satellite. This result is achieved through ROS toolbox that allows the communication between ROS / ROS2 and Matlab/Simulink. The simulation is performed by running the designed attitude determination application (and the sensors nodes) on the Raspberry Pi and the Matlab scripts on a computer.

5.1 Settings of ROS toolbox

In order to allow the communication between the ROS2 nodes designed for the flight software framework and Matlab, the usage of ROS toolbox is taken into account. The ROS2 nodes are run on the Raspberry Pi (that can be accessed from a PC using ssh protocol) while Matlab with ROS toolbox is run on a PC. In order to let the two machines see each other, a xml file named "*DEFAULT_FASTRTPS_PROFILES*" has to be inserted both in the ROS2 workspace folder on the Raspberry Pi and in the Matlab folder in which are present the script for the animation. The xml file is structured as presented in Figure 5. 1:

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles>
    <transport_descriptors>
        <transport_descriptor>
            <transport_id>veelpeers</transport_id> <!-- string -->
            <type>UDPv4</type> <!-- string -->
            <maxInitialPeersRange>100</maxInitialPeersRange> <!-- uint32 -->
        </transport_descriptor>
    </transport_descriptors>
    <participant profile_name="participant_somename" is_default_profile="true">
        <rtps>
            <builtin>
                <initialPeersList>
                    <locator>
                        <udpv4>
                            <address>192.168.10.2</address>
                        </udpv4>
                    </locator>
                    <locator>
                        <udpv4>
                            <address>192.168.10.1</address>
                        </udpv4>
                    </locator>
                </initialPeersList>
            </builtin>
            <userTransports>
            <transport_id>veelpeers</transport_id>
            </userTransports>
            <useBuiltinTransports>false</useBuiltinTransports>
        </rtps>
    </participant>
</profiles>
```

Figure 5. 1: DEFAULT_FASTRTPS_PROFILES.xml file

The two devices communicate through Ethernet cable and they must belong to the same subnet. Their IP addresses (192.168.10.1 for the PC and 192.168.10.2 for the Raspberry Pi) must be

present in both Matlab and ROS2 xml files.
The first step that is necessary to realize the real-time animation is to generate a suitable custom message that is able to handle the data of the received quaternions. In order to do this, a file named "*AttitudeDetermination.msg*", that contain the structure of the message to be received (an array of float with dimension four), is generated. After that, the ROS Toolbox function "*ros2genmsg*" is able to generate the specified custom message by passing as argument the path to the "*AttitudeDetermination.msg*" file. The operations performed by this function are presented in Figure 5. 2:

```
>> ros2genmsg('C:\Users\MatteoPascucci\Desktop\ros2_matlab\custom_msgs')
Identifying message files in folder 'C:/Users/MatteoPascucci/Desktop/ros2_matlab/custom_msgs'..Done.
Validating message files in folder 'C:/Users/MatteoPascucci/Desktop/ros2_matlab/custom_msgs'..Done.
[1/1] Generating MATLAB interfaces for custom message packages... Done.
Running colcon build in folder 'C:/Users/MatteoPascucci/Desktop/ros2_matlab/custom_msgs/matlab_msg_gen/win64'.
Build in progress. This may take several minutes...
Build succeeded.build log
```

Figure 5. 2: ros2genmsg operations

To allow the generation of custom messages, Visual Studio 2017 with is related cross-compiler for C++ applications must be installed on the used machine.
Once this operations are performed, the Matlab environment is ready to receive data from the nodes of the ROS2 framework. After that, the lines presented in Figure 5. 3 are executed:

```
1    attitude_visualizer=ros2node("/attitude_visualizer");
2    att_sub=ros2subscriber(attitude_visualizer,"/attitude","custom_msg/AttitudeQuaternion",@att_callback);
```

Figure 5. 3: Generation of attitude visualizer node

With the command "*ros2node*" is possible to generate a new ROS2 node (named attitude visualizer) to handle the data of the computed quaternion. After that, with the command "*ros2subscriber*" the generated attitude visualizer node is subscribed to the topic "*attitude*" (in which the attitude quaternion data are published) by defining that the messages that will be received are the same type of "*AttitudeQuaternion*" messages. The Matlab file that handle the operations to be performed when a new message arrives on the topic is "*att_callback*".
Finally the produced files to realize the real-time animation are:

- *"att_callback"* to perform the operations to rotate the 3U satellite. This file is called every time a new quaternion is published on the "*attitude*" topic. The operations performed will be described in the following paragraph

- "*animation*" to define the 3U satellite object to be plotted

5.2 Real- time animation of a 3U satellite

Using the ROS Toolbox as presented before, the data of the quaternion (or the equivalent rotation matrix) computed by the designed ROS2 application, can be collected in a Matlab script to animate in real-time a 3U satellite. So, once a new rotation data is published on its specific topic, a specific Matlab callback is called and it stores the attitude information to use it to rotate

a 3U satellite simulated by creating a hypercube. This object is defined by its vertices collected into an array. The obtained object is presented in Figure 5. 4.

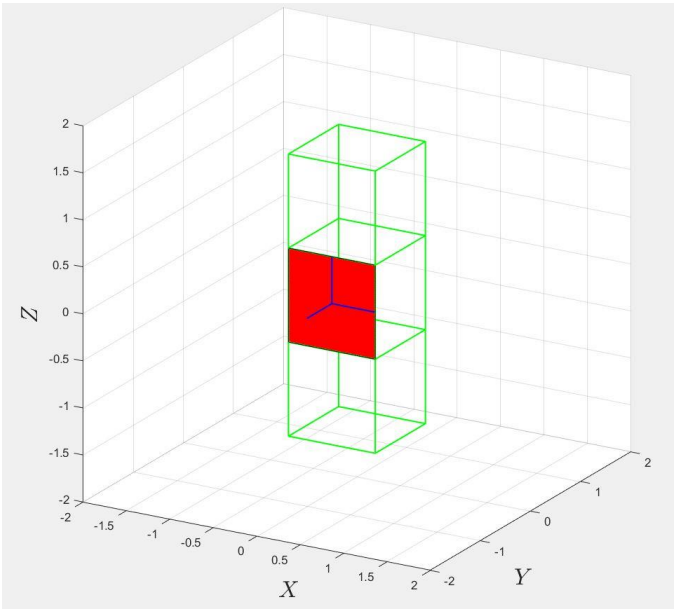

Figure 5. 4: 3U satellite simulation

In the presented Matlab plot are present the 3U satellite and its related body frame (correspondent with the magnetometer location on the satellite) with Y axis pointing up, Z axis pointing out and X axis to complete the right hand reference frame. The face of the satellite in which lays the body frame is colored in red.
Since the realized function to visualize the real-time animation uses rotation matrices, the first step to perferm is to convert the received quaternion, using the "*dcmtoqua*" function, to obtain the related rotation matrix. Once the simulation is started, the first quaternion received, its related rotation matrix and their conjugates are stored into suitable variables (respectively *q0 and R0*). This is done in order to visualize the rotation with respect to the first collected quaternion. For how the object is drawn, there is the problem that the body reference frame of the satellite is not aligned with the Matlab reference frame as it is shown in Figure 5. 5:

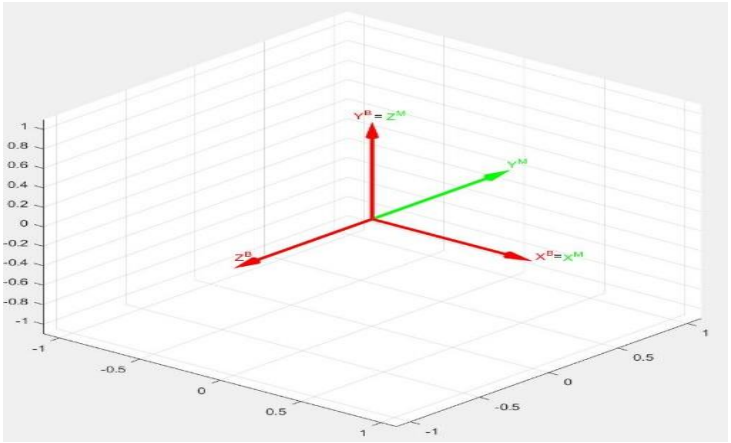

Figure 5. 5: Rotation of the Matlab frame w.r.t body frame

This cause that, for example, if a rotation around the Y axis of the body frame (indicated as Y^B in the figure) is intended, Matlab perform a rotation around its Z axis (indicated with Z^M in the figure). To fix this problem, each time that the callback is called, a rotation of -90° around the X axis is performed to align the satellite body frame with the Matlab frame.
Basically, each time that the callback is called, these operations are performed:

- The quaternion received from the ROS2 node is stored and converted into its related rotation matrix.

- If it is the first message received, the quaternion is stored in *q0* and the rotation matrix in *R0* and the conjugate of the quaternion is computed and stored into *q0_conj.* From this, it is retrieved the rotation matrix from the body frame to the ECEF frame using the *qua2dcm* function and it is stored into *R0_conj.*

- The rotation from the body frame to the Matlab frame is defined as:

$$R_{Matlab}^{body} = Rot_X(-90°) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}$$

- The rotation matrix to be applied to the satellite is finally computed as follows:

$$R = R_{conj} * R_{body}^{ECEF} * R_{Matlab}^{body}$$

- Finally, the "*animation_rot_R*" function is called to rotate the satellite with a rotation correspondent to the computed matrix R. This function just requires as argument the satellite object and the computed rotation matrix R.

5.3 Tests of the real-time animation

In order to test the realized real-time application, some animation can be performed to test the performances of the attitude determination. Particularly, the sensor module (that simulates the behaviour of a 3U satellite) is rotated around its three axes of its body reference frame. The obtained results are shown in the following figures.

Rotation about Z axis:

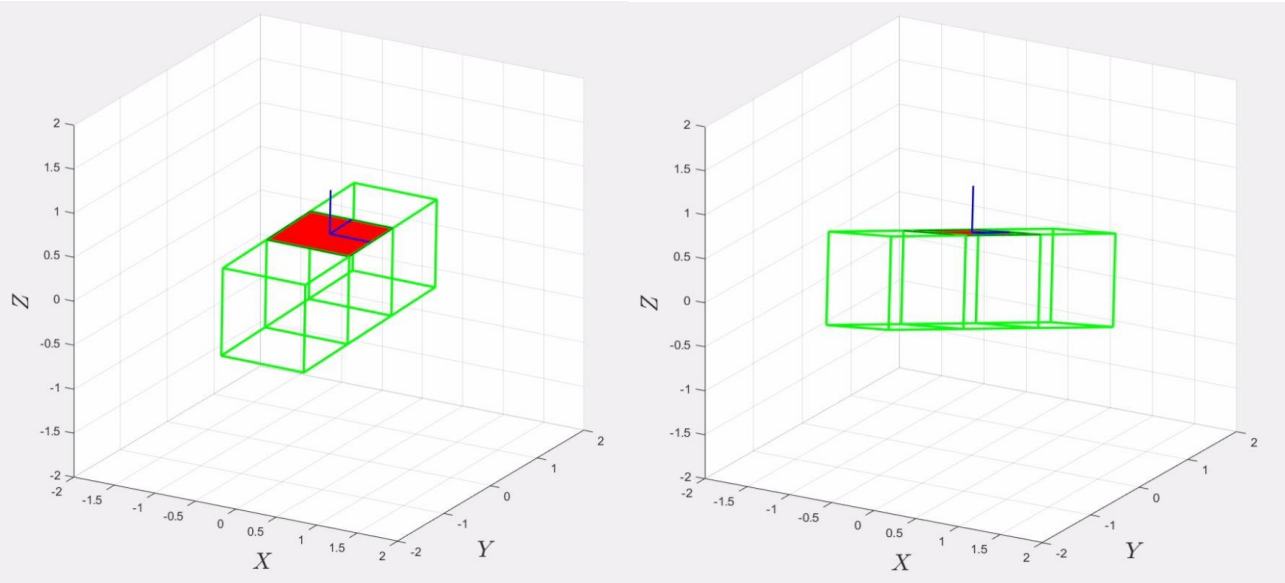

Figure 5. 6: Rotation about Z axis

Rotation about Y axis:

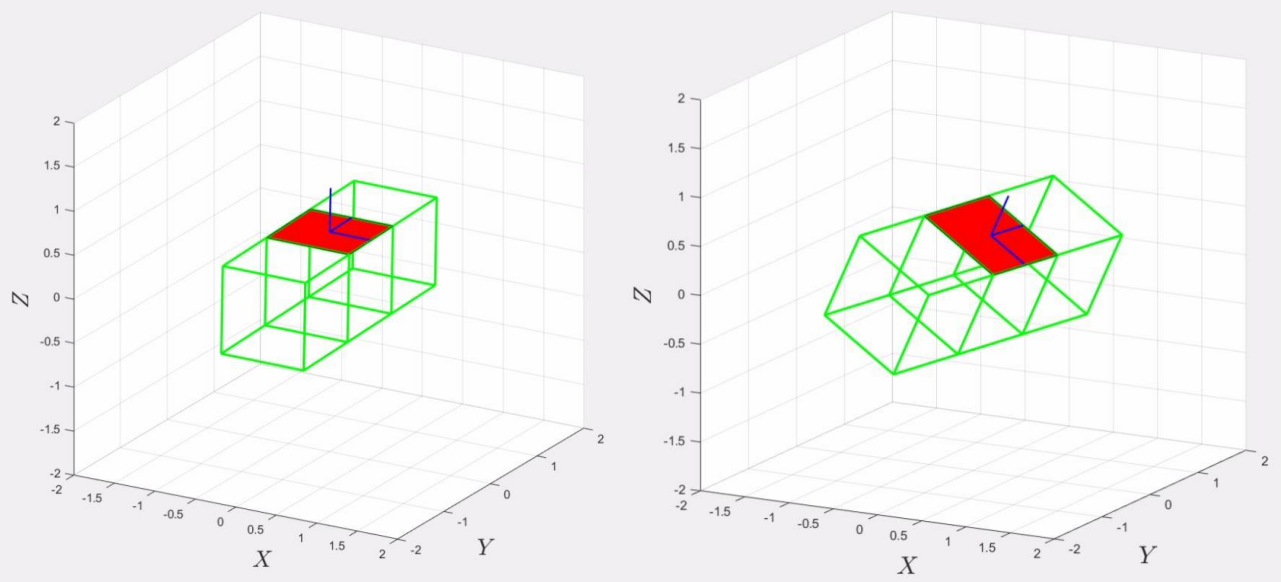

Figure 5. 7: Rotation about Y axis

Rotation about X axis:

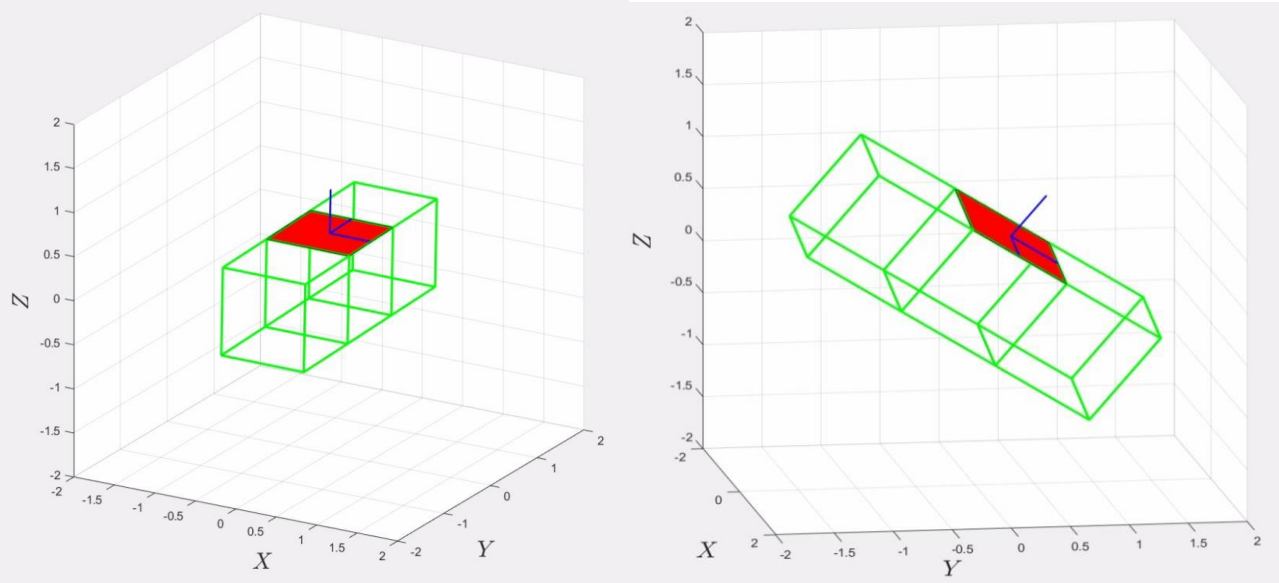

Figure 5. 8: Rotation about X axis

As it is reported in the above pictures the results are pretty good, even if some errors (due to some light reflections and noise of the sensors measurements) occurs. To take account of the entity of these errors, an analysis of the noise of the attitude determination is performed in the next paragraph.

5.4 Noise analysis of the attitude determination

The realized attitude determination application is subject to some imprecision in the determination of the orientation. This is due to different factors that influence the performance. In the first place, TRIAD is not the optimal algorithm to determine the attitude and some computation error may occur. Moreover, sensors are subject to some noise that affect the collected measurements. For this reason, an analysis to measure the order of magnitude of the noise that affect the attitude determination measurements is performed.
To do this, the attitude determination application is ran while the sensor module is hold fixed in a position. The following operations are so performed into the callback of the Matlab script:

- If it is the first iteration, the quaternion received is stored into the variable *q0.* Otherwise it is stored into the variable q_{body}.

- The conjugate of the quaternion received is computed and stored into a variable *q_body_conj*

- To compute the noise that occur in the attitude determination, it can be computed the rotation:

$$q_{error} = q_0 * q_{body\ conj}$$

The considered product is the quaternion Hamilton product. By doing this operation, it is computed the error that occurs between the first collected measurement *q0* and the current measurement quaternion *q_body* (i.e. the noise of the attitude determination)

- The current time and *q_error* are finally stored into suitable arrays to plot them in a graph

Since the sensor module is hold fixed, the expected result is that no rotation occurs and that *q_error* is equal to the identity quaternion (i.e. [1 0 0 0]). The collected results are plotted and are presented in Figure 5. 9.

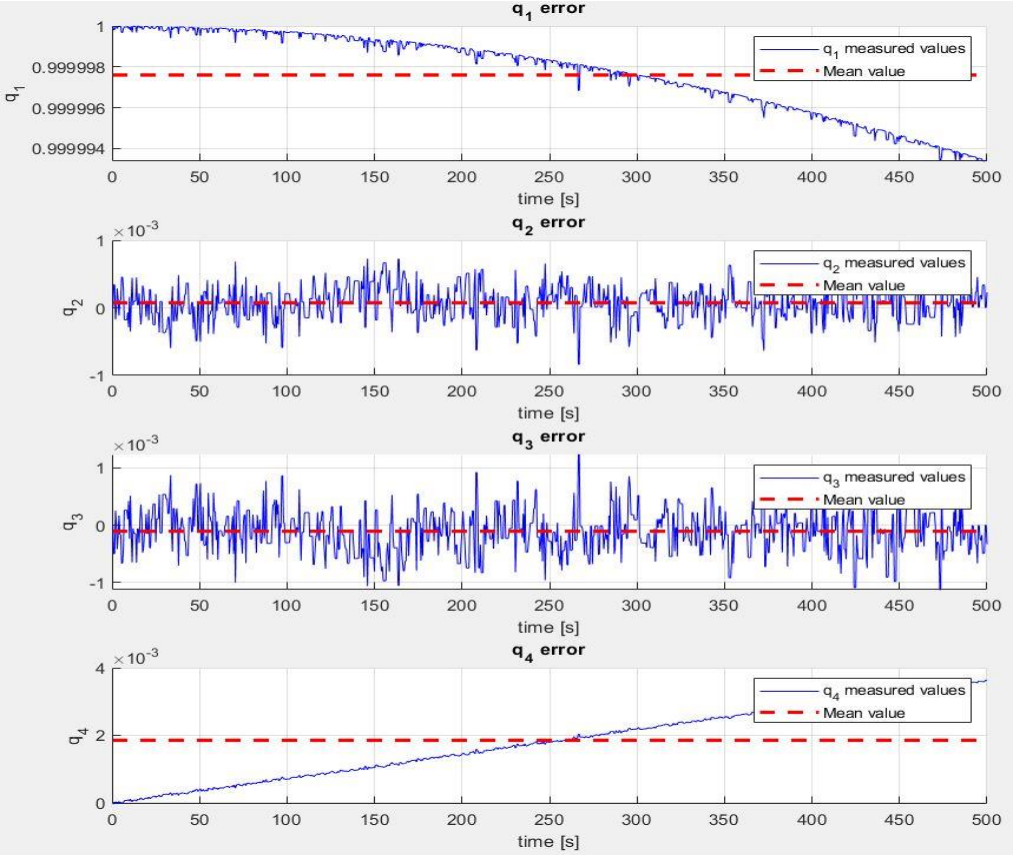

Figure 5. 9: Attitude determination noise analysis

The blue lines in the plot represented the collected values of q_{error} for each component of the quaternion. The dotted red lines represents the mean values of the measurements for each components of the quaternion computed as:

$$mean\ value(q_x) = \frac{\sum_{i=1}^{N} q_{x_i}}{N}$$

Where N is the number of the measurements and q_x is one of the four quaternion components. It is noted that the proposed attitude determination present good performances with very small error in both real and imaginary part of the quaternion. The decrease of the real part and the increase of the q_4 component over time is due to the movement of the Earth with respect to the Sun that causes small changes in the S^{ECEF} vector in the attitude determination algorithm. Beside that, the noise error is approximately in the order of 10^{-3}.

6 CONCLUSIONS

In this thesis work it was demonstrated that ROS2 can be a good choice for the design of software framework for complex systems like nanosatellites. Particularly, thanks to its modularity, it is easily possible to let different applications to communicate and exchange messages through topics.
Moreover, the integration of ROS2 with Matlab and Simulink using the ROS toolbox allows to apply the Model based software design philosophy by designing and simulating different control strategies and auto generate the C++ code of the ROS2 nodes using tools like embedded coder.

Since the aim of this thesis was to design a first implementation of the flight software framework, different applications can be implemented in the future in order to complete the realization of the nanosatellite architecture. Particularly:

- A node that is able to manage different kind of actuators like magnetorquers or reaction wheels to realize the required action computed by the controller

- Nodes that contain the control laws for different scenarios like detumbling manoeuvre or Earth pointing control in order to complete the design of a full Attitude determination and Control system. Particularly, these nodes can be auto generated directly from Matlab / Simulink.

- Nodes that handle all the communications, particularly the radio communications and the uplink or downlink of the data.

Moreover, this realization of the framework was performed on a standard board like a Raspberry Pi. Since Tyvak develop its own custom electronic boards, the framework will be ported into them. In order to do this, a suitable image that contain ROS2 and all the designed packages can be generated by using a suitable tool like Buildroot.
Once all the applications are designed and the system is ported into a custom board, the software framework can be considered completed and the integration testing phase will start.

7 APPENDIX A: BUILDROOT

Nowadays, many companies prefer to design their own customized electronic boards instead of using standard ones. Even if it can be an hard process in terms of R&D, it guarantees many advantages in terms of hardware since different combinations of devices can be mounted on it to achieve better performances for the desired task. On the other hand, it is necessary to realize a suitable image to properly communicate with the board. For this reason, different tools like Buildroot or Yocto have been realized to easily realize images for embedded boards.
Buildroot is a tool that is used in order to generate embedded Linux images for different types of boards using cross-compilations. It provides as outputs the root filesystem, the kernel, the bootloader and all the files that are needed for a specific board to build correctly an embedded Linux image. Moreover, Buildroot provides a lists of configurations files with a great number of boards and processors that are available on the market (for examples Raspberry Pi and SAM processor) that allow to build working images for that devices.
Buildroot allows the configuration of the images through and easy user interface called "*menuconfig*" that is presented in Figure 7. 1:

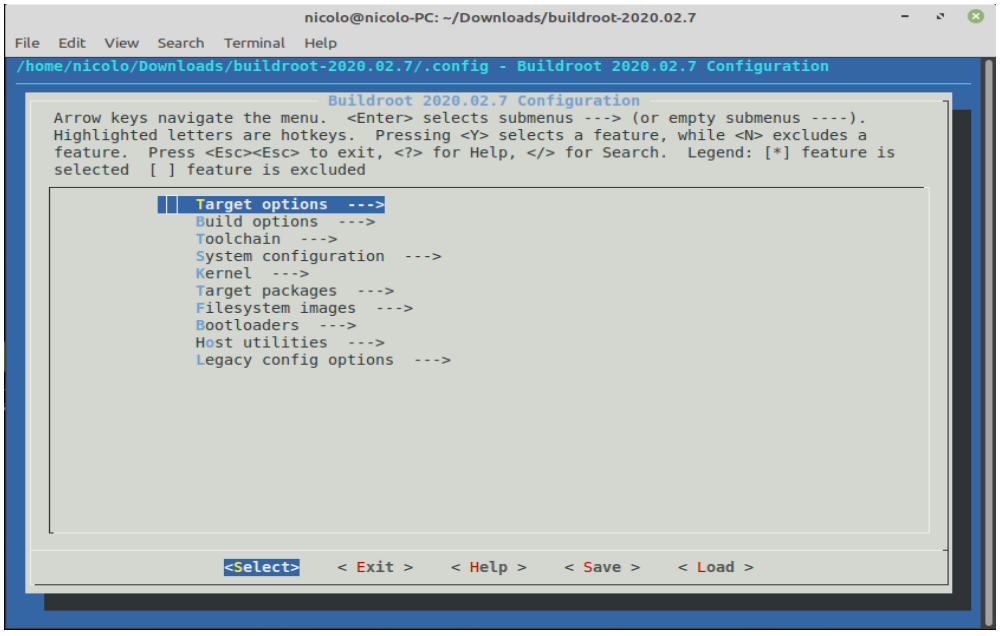

Figure 7. 1: Buildroot 2020 menuconfig

Analysing the available options of the menuconfig:

- Target options: it allows to set the architecture of the target CPU by choosing from a list of the most commonly used ones (like Intel or ARM architecture)
- Build options: it allows to configure the setting for the build like how many jobs to run simultaneously, enable the compiler cache, set the location of the download and host directory and optimization tools for the gcc compiler
- Toolchain: it allows to choose between a Buildroot or an external toolchain. Moreover, it is possible to configure the kernel headers, the version of the gcc cross-compiler, the options for uClibc (C libraries), activate the WCHAR support and enable the support to programming languages like C++ or Fortran

- System configuration: it allows to configure the whole system settings like the hostname, the system banner, activate the login with password, set the root password, set the path to the permission tables, activate timezones info and run custom scripts before or after the creation of the filesystem or inside the fakeroot environment
- Kernel: it allows to configure the kernel options like its version, patches and eventually a defconfig file, the output format of the kernel (the considered one is zImage), if a compression of the kernel is necessary, if it is necessary a *Device Tree Blob* (DTB) or if install the kernel in the "*/boot*" folder of the target
- Target Packages: all the packages that are present in Buildroot and that can be installed on the target like audio and video, compressors and decompressors for files, debug tools, graphical libraries, support for programming languages (Python, C++, PHP ecc…), tool for hardware support (i2c-detect, spidev ecc…) or text editors. In this section, it can be inserted custom packages.
- Filesystem Images: it allows to choose the output format of the generated filesystem (cpio, tar, jffs2 ecc…) and if it is necessary a compression. Moreover, it allows to integrate it as initramfs inside the kernel
- Bootloader: it allows to choose the desired bootloader (like U-boot) from a list and manage its configurations
- Host utilities: it allows to configure support tools for the host
- Legacy config options: packages that were present in older Buildroot versions

The original intention of this thesis project was to realize an image with Buildroot, that had ROS2 installed on it and to flash it on a custom board developed by Tyvak (called *EAB*) that mount an *ATSAM9G20* processor.
Tyvak provided a working image for the EAB realized with Buildroot 2012 to take it as starting point to understand which components are necessary to realize the new image using Buildroot 2020.
The first attempt was to realize an embedded linux image using standard files that are natively present in Buildroot. In the list of the supported boards of Buildroot 2020 is natively present the *AT91SAM9G20-EK* (that mount an *ATSAM9G20* processor) board. By using the command "*make atsam9g20dfc_defconfig*", the configuration described by this file is set in the options of the menuconfig and it can be built to produce a standard embedded Linux image that is compatible with this processor. To flash an image on the EAB, Tyvak uses a customized version of a tool named *Sam-ba,* which is commonly used to flash images on the SAM boards. The main problem is that, using the image produced by Buildroot, the flashing procedure is successful but the board does not boot up.
Analysing the image produced by Tyvak, it can be noted that all its component (kernel, filesystem and bootstrap) and some features in the settings are customized. In order to boot up, the EAB requires all those files and, if one or more of them are replaced with standard files produced by Buildroot, the booting procedure always fails.
For the reasons explained above and since the objective of this thesis was to demonstrate the feasibility of the design of a flight software in ROS2, the realization of the framework was moved, as explained in the related chapters, to a Raspberry Pi that mount Ubuntu 20.04 as operating system.

8 APPENDIX B: ROS2 CODE

In this chapter is presented the realized code for the ROS2 flight software. . The code was realized with Python 3.8.

8.1 Watchdog node code

```
#####################################################################
##
#
#                              WATCHDOG NODE
#
#####################################################################
##


import rclpy

import time

import os

import yaml


from rclpy.node import Node

from custom_msg.msg import Wdmsg

from ros2launch.api import *         # for launch_a_launch_file
function

from ros2node.api import *           # for get_node_names function

from multiprocessing import Process # for relaunching nodes with
Process()


# WATCHDOG FUNCTIONALITIES

#

# The provided Watchdog checks if the nodes provided by the yaml
configuration file and stored in a suitable dictionary, are active.

# This is done through the API provided by ROS2 "get_node_names".
If a node of the guarded list is not present, a suitable ROS2 API
```

```
# "launch_a_launch_file" is called by using the node unique ID, in
order to re-launch the node.


class Watchdog(Node):


def __init__(self, guarded_nodes):

super().__init__('watchdog')

watchdog_freq=5.0 # sec. Frequency of the watchdog callback

self.tmr_wd=self.create_timer(watchdog_freq,
self.watchdog_callback)

self.guarded_nodes=guarded_nodes # controlled by watchdog


def watchdog_launcher(self, launch_path): # Launch the missing node
launch file

launch_a_launch_file(launch_file_path=launch_path,launch_file_argum
ents="")


def create_active_nodes_names_list(self): # retrieving the list of
active nodes

self.active_node_names_list=[]

with NodeStrategy(self) as node:

node_list = get_node_names(node=node, include_hidden_nodes=False)

i=0

while(i<len(node_list)):

self.active_node_names_list.append(node_list[i].name)

i+=1


def checking_missing_nodes(self): # missing nodes checking

for node in self.guarded_nodes.values():

node_check=False

for j in range(0,len(self.active_node_names_list)):

if(node['name']==self.active_node_names_list[j]):
```

```
print("Node ",node['name']," present")
node_check=True
if(not node_check):
print('Launching missing node: ', node['name'])
p=Process(target=self.watchdog_launcher,args=(node['launch_path'],)
)
p.start()

def watchdog_callback(self): # Watchdog core
self.create_active_nodes_names_list()
print('Active nodes: ', self.active_node_names_list)
self.checking_missing_nodes()

def main(args=None):

rclpy.init(args=args)

# collecting Bus informations from Yaml file
stream=open('/home/ubuntu/ros2_ws/src/watchdog/watchdog/watchdog_cf
g.yaml', 'r')
cfg=yaml.load(stream, Loader=yaml.FullLoader)

guarded_nodes=cfg['guarded_nodes']

watchdog = Watchdog(guarded_nodes) # initialize watchdog

rclpy.spin(watchdog)

# Destroy the node explicitly
# (optional - otherwise it will be done automatically
# when the garbage collector destroys the node object)
```

```
watchdog.destroy_node()
rclpy.shutdown()

if __name__ == '__main__':
main()
```

8.1.1 Watchdog launch file code

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='talker',
            executable='talker'
        ),
        Node(
            package='listener',
            executable='listener'
        )
    ])
```

8.2 I2C reading node code

```
##############################################################################
#
#                    I2C BUS SENSORS READER NODE
#
##############################################################################
```

```
import rclpy

import os

import smbus2

import yaml

import sys


from . import Sensors

from rclpy.node import Node

from custom_msg.msg import SensorsMsg

from multiprocessing import Process     # for launching nodes with
Process()


global requested_bus # to change node name corresponding to the
specified bus (n=busN)


# I2C SENSORS READER FUNCTIONALITIES

#

# The provided node is intended for reading sensors attached to a
specific i2c bus. Using the command "ros2 run sensors
sensors_reader_i2c bus1/bus2/.../busN" is possible to launch

# a node for each specified i2c bus to handle, using the associated
YAML configuration file. Each I2C bus node creates a sensor object
for each sensor and reads the collected data.

# These data are published on a specific topic called
"i2c_sensors_data".


class I2C_bus(Node):


def __init__(self, bus, sensors_info, n_bus):

super().__init__('i2c_'+requested_bus)

self.bus=bus

self.sensors_info=sensors_info
```

```
self.n_bus=n_bus
self.sens=[]        # for storing sensors objects

print("Reading data from I2C",sys.argv[1],"...")

# creating objects for each sensor
for sensor in self.sensors_info.values():
if(sensor['type']=='temp'):
# sensor AD7415 object
self.sens.append(Sensors.AD7415(self.bus,sensor['addr'],None))
if(sensor['type']=='mag'):
# sensor HMC5883L object
self.sens.append(Sensors.HMC5883L(self.bus,sensor['addr'],None))
self.sens[-1].initialize()

self.publisher_ = self.create_publisher(SensorsMsg,
'i2c_sensors_data_'+requested_bus, 10)
timer_period = 0.001  # seconds
self.timer = self.create_timer(timer_period, self.sensor_reading)

def sensor_reading(self):
msg = SensorsMsg()
for i in range(len(self.sens)):
# reading sensors
if(self.sens[i].name=='AD7415'):                #Temperature sensor
msg.temp_raw=self.sens[i].read_sensor_raw()
msg.temp=self.sens[i].read_sensor()
if(self.sens[i].name=='HMC5883L'):             #Magnetometer sensor
msg.mag_raw=self.sens[i].read_sensor_raw()
msg.mag=self.sens[i].read_sensor()
# print(msg.mag)             #just for debug
```

```
self.publisher_.publish(msg)


def main(args=None):


rclpy.init(args=args)


common_path='/home/ubuntu/ros2_ws/src/sensors/sensors/'


global requested_bus
requested_bus=sys.argv[1]


# collecting Bus informations from Yaml file
stream=open(common_path+'i2c_'+requested_bus+'_cfg.yaml', 'r')
cfg=yaml.load(stream, Loader=yaml.FullLoader)
sensors_info=cfg['sensors']


# create and launch the node
bus_i2c=smbus2.SMBus(cfg['n_bus'])    # initializing the bus with
smbus2
i2c_bus = I2C_bus(bus_i2c,sensors_info,cfg['n_bus']) # creating bus
node
#p=Process(target=rclpy.spin, args=(i2c_bus,))
#p.start()
rclpy.spin(i2c_bus)


# Destroy the node explicitly
# (optional - otherwise it will be done automatically
# when the garbage collector destroys the node object)
i2c_bus.destroy_node()
rclpy.shutdown()
```

```
if __name__ == '__main__':
main()
```

8.3 SPI reading node code

```
#######################################################################
##
#
#                     SPI BUS SENSORS READER NODE
#
#######################################################################
###


import rclpy
import os
import spidev
import math
import yaml
import sys


from . import Sensors
from rclpy.node import Node
from custom_msg.msg import SensorsMsg
from multiprocessing import Process     # for launching nodes with
Process()


global requested_bus # to change node name corresponding to the
specified bus (n=busN)


# SPI SENSORS READER FUNCTIONALITIES
#
```

```
# The provided node is intended for reading sensors attached to a
specific SPI bus. Using the command "ros2 run sensors
sensors_reader_spi bus0/bus1/.../busN" is possible to launch

# a node for each specified SPI bus to handle, using the associated
YAML configuration file. Each SPI bus node creates a sensor object
for each sensor and reads the collected data.

# These data are published on a specific topic called
"spi_sensors_data".


class SPI_bus(Node):


def __init__(self, bus, sensors_info, n_bus):

super().__init__('spi_'+requested_bus)

self.bus=bus

self.sensors_info=sensors_info

self.n_bus=n_bus

self.sens=[]        # for storing sensors objects


print("Reading data from SPI",sys.argv[1],"...")


# creating objects for each sensor

for sensor in self.sensors_info.values():

if(sensor['type']=='sun'):

# sensor E91086 object

self.sens.append(Sensors.E91086(self.bus,None,sensor['cs']))


self.publisher_ = self.create_publisher(SensorsMsg,
'spi_sensors_data_'+requested_bus, 10)

timer_period = 0.001

self.timer = self.create_timer(timer_period, self.sensor_reading)


def sensor_reading(self):

msg = SensorsMsg()
```

```
for i in range(len(self.sens)):
# reading sensors
if(self.sens[i].name=='E91086'):                #Sun sensor
self.bus.open(self.n_bus,self.sens[i].cs)
self.sens[i].initialize()
msg.sun_raw=self.sens[i].read_sensor_raw()
msg.sun=self.sens[i].read_sensor()
# print(msg.sun)             #just for debug
# print("MAG_X: ",msg.sun[0],"[G]"," MAG_Y:
",msg.sun[1],"[G]","MAG_Z: ",msg.sun[2],"[G]")
self.publisher_.publish(msg)
self.bus.close()

def main(args=None):

rclpy.init(args=args)

common_path='/home/ubuntu/ros2_ws/src/sensors/sensors/'

global requested_bus
requested_bus=sys.argv[1]

# collecting Bus informations from Yaml file
stream=open(common_path+'spi_'+requested_bus+'_cfg.yaml', 'r')
cfg=yaml.load(stream, Loader=yaml.FullLoader)
sensors_info=cfg['sensors']

# create and launch the node
spi = spidev.SpiDev()                   # initializing the bus with
spidev
spi_bus = SPI_bus(spi,sensors_info,cfg['n_bus']) # creating bus
node
```

```
rclpy.spin(spi_bus)

# Destroy the node explicitly
# (optional - otherwise it will be done automatically
# when the garbage collector destroys the node object)
spi_bus.destroy_node()
rclpy.shutdown()



if __name__ == '__main__':
main()
```

8.4 Sensors code

```
import math

def word2bytearray(word):
array = [word >> 8, word & 0x00FF]
return array

class Sensors():

def __init__(self,bus,addr=None,cs=None):   # default none values
for addr and cs because we could have both I2C and SPI devices
self.bus=bus
self.addr=addr   # in case of I2C bus sensors
self.cs=cs       # in case of SPI bus sensors

def read_sensor_raw(self):
raise NotImplemented

def read_sensor(self):
```

```
raise NotImplemented

class AD7415(Sensors):
""" Analog Devices AD7415 Temperature Sensor.

Please refer to the datasheet (Rev. F) for further information.
"""
# Name
name='AD7415'

# Registers
REG_TEMP = 0x00
REG_CONFIG = 0x01

# Values
VAL_TRIGGER = 0x04
VAL_MSB_POSITION_SHIFTED = 0x02
VAL_SIGN_EXTEND_MASK = 0xFC

def read_sensor_raw(self):
# Reads inputs
raw = self.bus.read_byte_data(self.addr, self.REG_CONFIG)
self.bus.write_byte_data(self.addr,
self.REG_CONFIG, raw | self.VAL_TRIGGER)
temp_raw = self.bus.read_i2c_block_data(self.addr, self.REG_TEMP,
2)
return temp_raw

def read_sensor(self):
raw=self.read_sensor_raw()
# Purge low 6 bits
```

```
temp = (int.from_bytes(raw, byteorder='big', signed=False) >> 6)

temp_raw = word2bytearray(temp)

if temp_raw[0] & self.VAL_MSB_POSITION_SHIFTED:

temp_raw[0] = temp_raw[0] | self.VAL_SIGN_EXTEND_MASK

temp = (int.from_bytes(temp_raw, byteorder='big', signed=True))/4.
# 0.25 degC/LSB

return temp


class MMC5883MA(Sensors):     #NOTA CHE IL SENSORE USA LITTLE
ENDIAN

""" Memsic MMC5883MA 3-Axis Magnetometer.


Please refer to the datasheet (Rev. C) for further information.

"""

# Name

name='MMC5883MA'


# Registers

REG_DATA_OUT = 0x00

REG_CONTROL_0 = 0x08


# Values

VAL_TRIGGER_MAG = 0x01


def read_sensor_raw(self):

self.bus.write_byte_data(self.addr, self.REG_CONTROL_0,

self.VAL_TRIGGER_MAG)

field_raw = self.bus.read_i2c_block_data(self.addr,
self.REG_DATA_OUT, 6)

return field_raw


def read_sensor(self):
```

```
raw=self.read_sensor_raw()
# swapping positions because it works in Little Endian
rawX = [raw[1], raw[0]]
rawY = [raw[3], raw[2]]
rawZ = [raw[5], raw[4]]
field = [(int.from_bytes(rawX, byteorder='big', signed=False)-
32768)/4096.,
(int.from_bytes(rawY, byteorder='big', signed=False)-32768)/4096.,
(int.from_bytes(rawZ, byteorder='big', signed=False)-32768)/4096.]
return field

class HMC5883L(Sensors):
""" Honeywell HMC5883L 3-Axis Magnetometer.

Please refer to the datasheet (#ref?) for further information.
"""

name='HMC5883L'

# Settings
DEF_MA = 0b11
DEF_DO = 0b100
DEF_MS = 0b00
DEF_GN = 0b001
DEF_HS = 0b0
DEF_MD = 0b00

# Registers
REG_CONFIG_A = 0x00
REG_CONFIG_B = 0x01
REG_MODE = 0x02
```

```
REG_DATA_X_MSB = 0x03
REG_DATA_X_LSB = 0x04
REG_DATA_Y_MSB = 0x05
REG_DATA_Y_LSB = 0x06
REG_DATA_Z_MSB = 0x07
REG_DATA_Z_LSB = 0x08

# Values
VAL_GAIN = {0b000: 1370,
0b001: 1090,
0b010: 820,
0b011: 660,
0b100: 440,
0b101: 390,
0b110: 330,
0b111: 230}

def initialize(self):
# Apply default configs, then the user can change them when they
want
# Don't initialize at object creation so we can temporally separate
# object creation and object existence on the bus (sometimes
needed)
self.configure()

def configure(self, ma=DEF_MA, do=DEF_DO, ms=DEF_MS, gn=DEF_GN,
hs=DEF_HS, md=DEF_MD):
# CONFIG A
# MSb <   X    X    X    X    X    X    X    X   > LSb
#         0    MA1  MA0  DO2  DO1  DO0  MS1  MS0
self.configA = ((0b00000011 & (ms << 0)) |
```

```
(0b00011100 & (do << 2)) |
(0b01100000 & (ma << 5)))
# CONFIG B
# MSb <   X    X    X    X    X    X    X    X   > LSb
#         GN2  GN1  GN0  0    0    0    0    0
self.configB = 0b11100000 & (gn << 5)
# MODE
# MSb <   X    X    X    X    X    X    X    X   > LSb
#         HS   0    0    0    0    0    MD1  MD0
self.mode = ((0b10000000 & (hs << 7)) |
(0b00000011 & (md << 0)))
self.gain = self.VAL_GAIN[(0b111 & gn)]
self.bus.write_byte_data(self.addr, self.REG_CONFIG_A,
self.configA)
self.bus.write_byte_data(self.addr, self.REG_CONFIG_B,
self.configB)
self.bus.write_byte_data(self.addr, self.REG_MODE, self.mode)

def read_sensor_raw(self):
rawX = [self.bus.read_byte_data(self.addr, self.REG_DATA_X_MSB),
self.bus.read_byte_data(self.addr, self.REG_DATA_X_LSB)]
rawY = [self.bus.read_byte_data(self.addr, self.REG_DATA_Y_MSB),
self.bus.read_byte_data(self.addr, self.REG_DATA_Y_LSB)]
rawZ = [self.bus.read_byte_data(self.addr, self.REG_DATA_Z_MSB),
self.bus.read_byte_data(self.addr, self.REG_DATA_Z_LSB)]
field_raw=[rawX[0], rawX[1], rawY[0], rawY[1], rawZ[0], rawZ[1]]
return field_raw

def read_sensor(self):
raw=self.read_sensor_raw()
field = [float((int.from_bytes([raw[0],raw[1]], byteorder='big',
signed=True)))/self.gain,
```

```
float((int.from_bytes([raw[2],raw[3]], byteorder='big',
signed=True)))/self.gain,

float((int.from_bytes([raw[4],raw[5]], byteorder='big',
signed=True)))/self.gain]

return field


class E91086(Sensors):

""" E910.86 Sun Sensor.


Please refer to the datasheet for further information.

"""

# Name

name='E91086'


def initialize(self):

self.bus.mode = 0 # set SPI mode

self.bus.max_speed_hz = 500000 # set the frequency


self.bus.xfer2([0x90,0x18]) # command 1001000000011000 for
configuration (datasheet for more details)

self.bus.xfer2([0x00,0x00])


def read_sensor_raw(self):

ret=self.bus.xfer2([0x00,0x00]) # command for reading sensors

tmp = (ret[0]<<8 | ret[1])  & 0x3FFF  # concatenate the two
returned bytes since the output is 0100XXXXXXYYYYYY and set to 0
everything but the data XY

return tmp


def read_sensor(self):

tmp=self.read_sensor_raw()

Xdata = tmp >> 6
```

```
Ydata = tmp & 0x3F

Xn_deg=75*Xdata/27+15
Yn_deg=75*Ydata/27+15

# print("Degrees: ",Xn_deg,Yn_deg)

Xn_rad=Xn_deg*math.pi/180
Yn_rad=Yn_deg*math.pi/180

sun_angles=[Xn_rad, Yn_rad]
return sun_angles
```

8.5 I2C Sensors telemetry node code

```
#######################################################################
##
#
#                       I2C SENSORS TELEMETRY NODE
#
#######################################################################
##

import rclpy
import os
import struct
import sys

from rclpy.node import Node
from custom_msg.msg import SensorsMsg
```

```
from datetime import datetime


global requested_bus


# I2C SENSORS TELEMETRY FUNCTIONALITIES
#
# The provided node is intended for logging the data coming from i2c sensors in a suitable binary file. It splits the log files
# whenever a predefined threshold for the max number of messages stored is exceeded. So a new binary log file is created, if
# the threshold is exceeded or if the topic is not recorded yet, and stored in a predefined directory within its timestamp


class SensorsTelemetryI2C(Node):


def __init__(self):
super().__init__('i2c_sensors_telemetry_'+requested_bus)
self.subscription_i2c = self.create_subscription(
SensorsMsg,
'i2c_sensors_data_'+requested_bus,
self.sensors_telemetry_callback,
10)
self.subscription_i2c  # prevent unused variable warning
self.recording=False # to check if the log file is already created
self.ind=0 # to count the messages recorded


def create_binary(self):     # Create the log file in the sensors_log folder
path="/home/ubuntu/ros2_ws/src/telemetry/sensors_log/i2c_"+requested_bus
if not os.path.exists(path): # If the folder is not present, it'll be created
os.mkdir(path)
```

```
name_db=path+"/i2c_"+requested_bus+"_sensors_data-
"+str(datetime.now().strftime("%m-%d-%Y-%H:%M:%S"))+".bin" #
timestamp log file creation

print('Logging data in: '+name_db)

self.recording=True # log file created flag

self.file=open(name_db,'wb')

self.ind=0 # messages number reset


def insert_data(self, msg): # Insert the sensors data into the log
file created

if(self.ind< self.n_max):

tmp=struct.pack('ffffffffffff',

msg.temp_raw[0],msg.temp_raw[1],

msg.temp,

msg.mag_raw[0],msg.mag_raw[1],msg.mag_raw[2],msg.mag_raw[3],msg.mag
_raw[4],msg.mag_raw[5],

msg.mag[0],msg.mag[1],msg.mag[2]

)

self.file.write(tmp)

self.ind+=1


def sensors_telemetry_callback(self, msg):

self.n_max=1000

if(self.ind == self.n_max):

self.file.close()

if(not self.recording or self.ind > self.n_max-1):

self.create_binary()

self.insert_data(msg)

print("RECORDING...")



def main(args=None):
```

```
rclpy.init(args=args)

global requested_bus
requested_bus=sys.argv[1]

sensors_telemetry_i2c = SensorsTelemetryI2C()

rclpy.spin(sensors_telemetry_i2c)

# Destroy the node explicitly
# (optional - otherwise it will be done automatically
# when the garbage collector destroys the node object)
sensors_telemetry_i2c.destroy_node()
rclpy.shutdown()


if __name__ == '__main__':
main()
```

8.6 SPI sensors telemetry node code

```
##############################################################################
##
#
#                       SPI SENSORS TELEMETRY NODE
#
##############################################################################
##

import rclpy
import os
```

```
import struct
import sys

from rclpy.node import Node
from custom_msg.msg import SensorsMsg
from datetime import datetime

global requested_bus

# SPI SENSORS TELEMETRY FUNCTIONALITIES
#
# The provided node is intended for logging the data coming from spi sensors in a suitable binary file. It splits the log files
# whenever a predefined threshold for the max number of messages stored is exceeded. So a new binary log file is created, if
# the threshold is exceeded or if the topic is not recorded yet, and stored in a predefined directory within its timestamp

class SensorsTelemetrySPI(Node):

def __init__(self):
super().__init__('spi_sensors_telemetry_'+requested_bus)
self.subscription_spi = self.create_subscription(
SensorsMsg,
'spi_sensors_data_'+requested_bus,
self.sensors_telemetry_callback,
10)
self.subscription_spi  # prevent unused variable warning
self.recording=False # to check if the log file is already created
self.ind=0 # to count the messages recorded
```

```
def create_binary(self):     # Create the log file in the
sensors_log folder

path="/home/ubuntu/ros2_ws/src/telemetry/sensors_log/spi_"+requeste
d_bus

if not os.path.exists(path): # If the folder is not present, it'll
be created

os.mkdir(path)

name_db=path+"/spi_"+requested_bus+"_sensors_data-
"+str(datetime.now().strftime("%m-%d-%Y-%H:%M:%S"))+".bin" #
timestamp log file creation

print('Logging data in: '+name_db)

self.recording=True # log file created flag

self.file=open(name_db,'wb')

self.ind=0 # messages number reset


def insert_data(self, msg): # Insert the sensors data into the log
file created

if(self.ind< self.n_max):

tmp=struct.pack('fff',

msg.sun_raw,

msg.sun[0],msg.sun[1]

)

self.file.write(tmp)

self.ind+=1


def sensors_telemetry_callback(self, msg):

self.n_max=1000

if(self.ind == self.n_max):

self.file.close()

if(not self.recording or self.ind > self.n_max-1):

self.create_binary()

self.insert_data(msg)

print("RECORDING...")
```

```
def main(args=None):

rclpy.init(args=args)

global requested_bus
requested_bus=sys.argv[1]

sensors_telemetry_spi = SensorsTelemetrySPI()

rclpy.spin(sensors_telemetry_spi)

# Destroy the node explicitly
# (optional - otherwise it will be done automatically
# when the garbage collector destroys the node object)
sensors_telemetry_spi.destroy_node()
rclpy.shutdown()


if __name__ == '__main__':
main()
```

8.7 Attitude determination node code

```
import rclpy
import os
import numpy
import math
import pyIGRF
import datetime
import navpy
```

```
from rclpy.node import Node
from custom_msg.msg import SensorsMsg
from custom_msg.msg import AttitudeQuaternion
from PyAstronomy import pyasl
from skyfield import framelib
from skyfield.api import load_file
from skyfield.api import load

class AttitudeDetermination(Node):

def __init__(self):
super().__init__('attitude_determination')
self.sun = None
# self.sun_safe_b= 1.61927769490585 # 92.7° (or 1.522314958683943
for 87.3° )
# self.sun_safe_a= 1.61927769490585 # 92.7° (or 1.522314958683943
for 87.3° )
self.mag = None
self.subscription_i2c = self.create_subscription(
SensorsMsg,
'i2c_sensors_data_bus1',
self.i2c_mag_callback,
10)
self.subscription_spi = self.create_subscription(
SensorsMsg,
'spi_sensors_data_bus0',
self.spi_sun_callback,
10)
self.quat_publisher = self.create_publisher(AttitudeQuaternion,
'attitude', 10)
print("Starting Attitude Determination...")
```

```
self.subscription_i2c  # prevent unused variable warning

self.subscription_spi  # prevent unused variable warning

timer_period=0.01       # 10 Hz

self.AD_timer = self.create_timer(timer_period,
self.AD_timer_callback)


def i2c_mag_callback(self, msg):     # callback collecting mag
sensor data

self.mag=msg.mag


def spi_sun_callback(self, msg):     # callback collecting sun
sensor data

self.sun=msg.sun


def sun_mag_vectors_ECEF(self):      # method computing ECEF frame
vectors

# Variables needed for M_ECEF vector computation

lat_deg=45.09221603086248

lon_deg=7.670356843569824

lat_rad=lat_deg*math.pi/180

lon_rad=lon_deg*math.pi/180

alt=0.239               #km

date=pyasl.decimalYear(datetime.datetime.now())

# Variables needed for S_ECEF vector computation

ts = load.timescale()

t = ts.now()            # Julian date hour expressed in UT (-1h wrt
Italy)

planets =
load_file('/home/ubuntu/ros2_ws/src/attitude_determination/attitude
_determination/ephemeris/de421.bsp')

sun = planets['sun']

earth = planets['earth']

# M_NED, M_ECEF computation
```

```
mag_info=pyIGRF.igrf_value(lat_deg, lon_deg, alt, date)

M_NED=numpy.array([mag_info[3],mag_info[4],mag_info[5]])    #nT
(North,East,Down coordinates)

M_NED=M_NED/(numpy.linalg.norm(M_NED))                       #
normalization

a=lat_rad+math.pi/2

b=-lon_rad

Ry=numpy.array([[math.cos(a),0,-
math.sin(a)],[0,1,0],[math.sin(a),0,math.cos(a)]])

Rz=numpy.array([[math.cos(b),math.sin(b),0],[-
math.sin(b),math.cos(b),0],[0,0,1]])

R=numpy.dot(Rz,Ry)
# rotation matrix: NED FRAME -> ECEF FRAME

R=R.T                                                        #
transformation matrix from NED frame -> ECEF FRAME

M_ECEF=numpy.dot(R,M_NED)

# S_ECEF computation

apparent = earth.at(t).observe(sun).apparent()

sun_info = apparent.frame_xyz(framelib.itrs)

S_ECEF=numpy.array(sun_info.au)

S_ECEF=S_ECEF/(numpy.linalg.norm(S_ECEF))

ret=[M_ECEF,S_ECEF]

return ret


def sun_mag_vectors_BODY(self):      # method computing BODY frame
vectors

# Sb computation

b=self.sun[0]-math.pi/2 # angle XZ-plane

a=self.sun[1]-math.pi/2 # angle YZ-plane

# if (abs(b-math.pi/2)<0.047):

#     b=self.sun_safe_b

# self.sun_safe_b=b

# if (abs(a-math.pi/2)<0.047):
```

```
#     a=self.sun_safe_a

# self.sun_safe_a=a

# print("beta: ",b)

# print("alpha: ",a)

S_B=numpy.array([math.tan(b),math.tan(a),1])       # general
relation for 2-axis digital sun sensors

#print("S_B non normalizzato: ",S_B)

# Mb computation

R=numpy.array([[0,-1,0],[-1,0,0],[0,0,-1]])  # rotation matrix: MAG
sensor FRAME -> SUN sensor FRAME

R=R.T                                        # transformation
matrix: MAG sensor FRAME -> SUN sensor FRAME

M_B=R.dot(numpy.array(self.mag))

# normalize vectors

S_B=S_B/(numpy.linalg.norm(S_B))

M_B=M_B/(numpy.linalg.norm(M_B))

ret=[M_B,S_B]

return ret


def TRIAD_attitude_determination(self,S_B,M_B,S_ECEF,M_ECEF):

# creating the triads: USING S_B as "best" measure

# 1st components

t1b=S_B

t1i=S_ECEF

# 2nd components

tmp=numpy.cross(S_B, M_B)

t2b=tmp/(numpy.linalg.norm(tmp))

tmp=numpy.cross(S_ECEF, M_ECEF)

t2i=tmp/(numpy.linalg.norm(tmp))

# 3rd components

t3b=numpy.cross(t1b, t2b)

t3i=numpy.cross(t1i, t2i)
```

```
# attitude matrix computation

Rbt=(numpy.array([t1b,t2b,t3b])).T   # rotation matrix: BODY FRAME
-> TRIAD FRAME

Rti=numpy.array([t1i,t2i,t3i])    # rotation matrix: TRIAD FRAME ->
ECEF FRAME

DCM_attitude=numpy.dot(Rbt,Rti)

return DCM_attitude


def AD_timer_callback(self):         # Timed callback computing
attitude (refer to "timer_period") via TRIAD algorithm

if (self.sun is not None and self.mag is not None):

# store body frame vectors

v=self.sun_mag_vectors_BODY()

M_B=v[0]

S_B=v[1]

# store ECEF frame vectors

v=self.sun_mag_vectors_ECEF()

M_ECEF=v[0]

S_ECEF=v[1]

# print("S_B: ",S_B)

# print("\n")

# print("M_B: ",M_B)

# print("S_ECEF: ",S_ECEF)

# print("M_ECEF: ",M_ECEF)

# TRIAD ALGORITHM

DCM_attitude=self.TRIAD_attitude_determination(S_B,M_B,S_ECEF,M_ECE
F)

q0,qvec=navpy.dcm2quat(DCM_attitude)

q_attitude=[q0, qvec[0], qvec[1], qvec[2]]

# print("Attitude DCM Matrix: ")

# print(DCM_attitude)

# print("Attitude quaternion: ")
```

```
# print(q_attitude)
# print("\n")
msg = AttitudeQuaternion()
msg.quat=q_attitude
msg.r1=DCM_attitude[0]
msg.r2=DCM_attitude[1]
msg.r3=DCM_attitude[2]
self.quat_publisher.publish(msg)

def main(args=None):
rclpy.init(args=args)

attitude_determination = AttitudeDetermination()

rclpy.spin(attitude_determination)

# Destroy the node explicitly
# (optional - otherwise it will be done automatically
# when the garbage collector destroys the node object)
attitude_determination.destroy_node()
rclpy.shutdown()


if __name__ == '__main__':
main()
```

9 APPENDIX C: MATLAB CODE

In this chapter is presented the Matlab code used in order to realize the real-time animation of the attitude determination of a 3U satellite. As described in the previous chapters, the code takes usage of the ROS toolbox to allow the communication between Matlab and ROS2.

9.1 Attitude callback

```
function att_callback(msg)
global sat
global bodyf
global q0
global q0_conj
global R0_conj
global q_stored
global R_stored
global t
global time
global check

%Satellite animation
q_body=(msg.quat)';
R_body=[msg.r1; msg.r2; msg.r3];

if (check==0)
    q0_conj=[q_body(1); -q_body(2:end)];
    q0=q_body;

%     R0_conj=(quat2dcm(q_body'))';

    check=check+1;
end
% ROTATING OUR QUATERNION/MATRIX ATTITUDE WITH A ROTATION OF -
90° WRT X AXIS
q_btom=[0.7071, -0.7071, 0, 0]; % for aligning bf to matlab
frame
q_tmp=quatprod(q0_conj,q_btom');
q=quatprod(q_body,q_tmp);
% animation_rot_q(sat,bodyf,q)

%for evaluating q error
q_body_conj=[q_body(1); -q_body(2:end)];
q_err=quatprod(q0,q_body_conj);

R_btom=[1 0 0; 0 0 1; 0 -1 0]; % for aligning bf to matlab frame
R_tmp=R0_conj*R_btom;
R=R_body*R_tmp;
animation_rot_R(sat,bodyf,R)
```

```
    % quaternion plot
    t=t+0.5;
    time=[time, t];
    q_stored=[q_stored,q_err];
    % R_stored=[R_stored; R];

    end
```

9.2 Animation

```
clc; clear;


%% Initial Satellite (3U)
global sat
global bodyf
global q_stored
global R_stored
global time
global t
global check
global q_conj
global R0_conj
global q0

q0=[];
q_conj=[];
R0_conj=[];
q_stored=[];
R_stored=[];
sat=[];
time=[];
t=0;
check=0;

% 3U Satellite definition and body reference frame model
for i=0:2
    U=[-0.5 -0.5  0.5  0.5 -0.5 -0.5 -0.5  0.5  0.5 -0.5  0.5  0.5
0.5  0.5 -0.5 -0.5 -0.5 -0.5;
        0.5 -0.5 -0.5  0.5  0.5  0.5 -0.5 -0.5  0.5  0.5  0.5  0.5
-0.5 -0.5 -0.5 -0.5 -0.5  0.5;
       -1.5 -1.5 -1.5 -1.5 -1.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -1.5
-1.5 -0.5 -0.5 -1.5 -0.5 -0.5 ];
    U(3,:)=U(3,:)+i;
    sat=[sat, U];
end
% Body frame definition (same of Sun Sensor): BF->MATLAB=ROTATION -
90° WRT X AXIS OF BODY FRAME
```

```
% WRT X AXIS
bodyf_origin=[0; -0.5; 0];
bodyf_z=[0; -1; 0];
bodyf_x=[0.5; -0.5; 0];
bodyf_y=[0; -0.5; 0.5];
bodyf=[bodyf_origin, bodyf_z, bodyf_origin,bodyf_x, bodyf_origin,
bodyf_y, bodyf_origin];

% plotting initial conditions
% figure(1); grid on; hold on;
% daspect([1 1 1])
% view(30,20)
% fs1=20;
% l1=2;
% xlim([-l1 l1])
% ylim([-l1 l1])
% zlim([-l1 l1])
% xlabel('$X$','interpreter','latex','fontsize',fs1)
% ylabel('$Y$','interpreter','latex','fontsize',fs1)
% zlabel('$Z$','interpreter','latex','fontsize',fs1)
% fac=[17 21 26 25];
%
plot3(bodyf(1,:),bodyf(2,:),bodyf(3,:),'color','b','tag','initial',
'linewidth',1.2); hold on; grid on;
%
plot3(sat(1,:),sat(2,:),sat(3,:),'color','g','tag','initial','linew
idth',1.2);
%
patch('Vertices',sat','Faces',fac,'FaceVertexCData',hsv(1),'FaceCol
or','flat')

%% Rotating Satellite
% declaring a subscribe to attitude topic
attitude_visualizer=ros2node("/attitude_visualizer");
att_sub=ros2subscriber(attitude_visualizer,"/attitude","custom_msg/
AttitudeQuaternion",@att_callback);

% figure(2);
% subplot(411); hold on; grid on;
% plot(t,q_stored(1,:),'b');
% subplot(412); hold on; grid on;
% plot(t,q_stored(2,:),'k');
% subplot(413); hold on; grid on;
% plot(t,q_stored(3,:),'r');
% subplot(414); hold on; grid on;
% plot(t,q_stored(4,:),'m');

%% for evaluating q_error
figure(3);
subplot(221); hold on; grid on; title('q_1'); xlim([0,500]);
plot(time,q_stored(1,:));
```

```
subplot(222); hold on; grid on; title('q_2'); xlim([0,500]);
plot(time,q_stored(2,:));
subplot(223); hold on; grid on; title('q_3'); xlim([0,500]);
plot(time,q_stored(3,:));
subplot(224); hold on; grid on; title('q_4'); xlim([0,500]);
plot(time,q_stored(4,:));
```

10. Bibliography and Sitography

Attitude Determination using Two measurements. (s.d.). Tratto da ReaserchGate: https://www.researchgate.net/publication/4706531_Attitude_Determination_Using_Two_Vector_Measurements

Buildroot, Presentation Slides. (s.d.). Tratto da Buildroot: https://bootlin.com/doc/training/buildroot/buildroot-slides.pdf

CubeSat. (s.d.). Tratto da Wikipedia: https://it.wikipedia.org/wiki/CubeSat

Earth's Magnetic Field. (s.d.). Tratto da Earth's Magnetic Field: https://web.ua.es/docivis/magnet/earths_magnetic_field2.html

International Geomagnetic Reference Field. (s.d.). Tratto da National Center for Environment Informations: https://www.ngdc.noaa.gov/IAGA/vmod/igrf.html

International Geomagnetic Reference Field. (s.d.). Tratto da Wikipedia: https://it.wikipedia.org/wiki/International_Geomagnetic_Reference_Field

Novara, C. (2020). *Course Slides, "Nonlinear Control and Aerospace Applications".* Politecnico di Torino.

Novara, C., Canuto, E., Montenegro, C. P., Massotti, L., & Carlucci, D. (2018). *Spacecraft Dynamics and Control: The Embedded Model.*

ROS2 Foxy. (s.d.). Tratto da ROS Index: https://docs.ros.org/en/foxy/index.html

Satellite Miniaturizzato. (s.d.). Tratto da Wikipedia: https://it.wikipedia.org/wiki/Satellite_miniaturizzato

Schaub, H. (s.d.). *Spacecraft Dynamics and Control Specialization*. Tratto da Coursera: https://www.coursera.org/specializations/spacecraft-dynamics-control

Skyfield. (s.d.). Tratto da Rhodes Mill: https://rhodesmill.org/skyfield/

TRIAD Algorithm. (s.d.). Tratto da Satellite Wiki: https://www.aero.iitb.ac.in/satelliteWiki/index.php/Triad_Algorithm

Tyvak International Website. (s.d.). Tratto da Tyvak International: https://www.tyvak.eu/

Wertz, J. R. (1978). *Spacecraft Attitude Determination and Control.*