

# Politecnico di Torino



*Master's degree course in "Mechatronic Engineering"*

*Master's Degree Thesis:*

***"Development of a ROS2 flight software framework &  
Attitude Control application for nanosatellites"***

**Candidate:**

Pascucci Matteo

**Supervisor:**

Prof. Corpino Sabrina

**Tutor:**

Eng. Zanotti Andrea

Academic year 2020/2021





# TABLE OF CONTENTS

<b>1. INTRODUCTION .....</b>	<b>7</b>
1.1. Nanosatellites overview .....	7
1.2. Thesis objective and context .....	8
1.3. Structure of the thesis .....	10
<b>2. SYSTEM ARCHITECTURE .....</b>	<b>12</b>
2.1. Project overview .....	12
2.2. Hardware configuration .....	12
2.2.1. Temperature sensor AD7415 .....	13
2.2.1.1. AD7415 test .....	15
2.2.2. Magnetometer HMC5883L .....	16
2.2.2.1. HMC5883L test .....	17
2.1.1. Sun sensor E910.86 .....	18
2.1.1.1. E910.86 test .....	21
2.1.2. Raspberry PI 3 B+ .....	23
2.2. Software configuration .....	24
2.2.1. ROS2 overview and advantages .....	25
<b>3. ROS2 FLIGHT SOFTWARE FRAMEWORK .....</b>	<b>29</b>
3.1. Watchdog node .....	29
3.1.1. Watchdog node test .....	32
3.2. Sensors Bus node .....	33
3.2.1. I2C bus node .....	35
3.2.2. SPI bus node .....	37
3.3. Sensors Telemetry node .....	39
3.3.1. Telemetry node test .....	41
<b>4. ATTITUDE CONTROL .....</b>	<b>43</b>
4.1. Rotation matrices and quaternions .....	43
4.2. Reference Frames .....	46
4.3. Satellite dynamical model .....	48
4.4. General overview of AC systems .....	50
4.4.1. Magnetorquers .....	51
4.4. Attitude control scenarios .....	52
4.4.1. B-dot control for detumbling phase .....	53

4.4.2. Earth-pointing control.....	54
<b>5. MATLAB/SIMULINK MODELS .....</b>	<b>57</b>
5.1. Orbit and Earth magnetic field propagator.....	58
5.2. Satellite model.....	61
5.3. Magnetorquers model.....	64
5.4. B-dot Bang controller model .....	64
5.5. Earth-pointing controller model.....	66
<b>6. SIMULATIONS AND CODE AUTO-GENERATION .....</b>	<b>67</b>
6.1. B-dot detumbling simulations .....	67
6.2. Earth-pointing simulations .....	72
6.3. ROS Toolbox and code auto-generation.....	77
<b>7. CONCLUSIONS .....</b>	<b>80</b>
<b>8. APPENDIX A: BUILDROOT .....</b>	<b>81</b>
<b>9. APPENDIX B: ROS2 CODE .....</b>	<b>84</b>
<b>10. APPENDIX C: MATLAB CODE.....</b>	<b>98</b>
REFERENCES .....	102

# LIST OF FIGURES:

Figure 1: Tyvak's Commtrail nanosatellite (3U) .....	7
Figure 2: V-shape process flow of software design .....	8
Figure 3: E91086 MISO output voltages. Vdd=4.5V to 5.5V .....	13
Figure 4: Components connections circuit diagram.....	13
Figure 5: AD7415 Register structure .....	14
Figure 6: AD7415 Configuration register bits definition.....	14
Figure 7: AD7415 Temperature value register readings output.....	14
Figure 8: AD7415 circuit diagram .....	15
Figure 9: AD7415 sensor test.....	15
Figure 10: AD7415 sensor heated test .....	16
Figure 11: HMC5883L channel X data output registers A and B .....	17
Figure 12: HMC5883L circuit diagram .....	17
Figure 13: HMC5883L sensor test.....	18
Figure 14: physical model of Xn angle .....	18
Figure 15: digital output – angles relation .....	19
Figure 16: sun vector model .....	20
Figure 17: E910.86 circuit diagram .....	21
Figure 18: E910.86 testing setup .....	21
Figure 19: E910.86 Xn, Yn angles at 90° .....	22
Figure 20: E910.86 Xn changing test.....	22
Figure 21: E910.86 Xn changing test.....	23
Figure 22: Raspberry Pi 3 B+ board and GPIO scheme. ....	23
Figure 23: final circuit with: Raspberry PI, logic level converter and sensor module .....	24
Figure 24: ROS2 latest distributions and EOL dates.....	26
Figure 25: Publisher “Node” sends a message over the topic “Topic” .....	27
Figure 26: Call-and-response method implemented by the service .....	27
Figure 27: rqt_graph of the official teleop turtle tutorial .....	28
Figure 28: Mark-1 watchdog flow chart.....	29
Figure 29: Watchdog(Node) class .....	30
Figure 30: ROS2 based watchdog flow chart.....	31
Figure 31: watchdog config YAML.....	31
Figure 32: Watchdog node test: all the guarderd nodes are running.....	32
Figure 33: Watchdog node test: SPI sensors reader node is missing .....	32
Figure 34: Watchdog node test: all the guarderd nodes are missing.....	33
Figure 35: SPI bus example with several identical sensors.....	34
Figure 36: Realistic situation with many sensors on two different buses .....	35
Figure 37: I2C protocol representation.....	35
Figure 38: Sensors custom message structure .....	36
Figure 39: I2C bus node flow chart .....	36
Figure 40: I2C bus node class .....	37
Figure 41: I2C bus node YAML configuration file.....	37
Figure 42: SPI communication protocol example with a Master and three slaves .....	38

Figure 43: SPI_bus node class diagram.....	38
Figure 44: SPI_bus node execution flowchart.....	39
Figure 45: SPI_bus node configuration file.....	39
Figure 46: I2C/SPI bus sensors telemetry class.....	40
Figure 47: I2C/SPI bus sensors telemetry class.....	40
Figure 48: Telemetry node test: creation of a new file.....	41
Figure 49: Telemetry node test: reading stored data .....	42
Figure 50: F1, F2 reference frames and a generic particle.....	43
Figure 51: position of the particle with respect to F1, F2 .....	43
Figure 52: R written in matrix form in function of (x,y,z) .....	44
Figure 53: quaternion equivalent notations.....	45
Figure 54: Algebra of quaternions.....	45
Figure 55: DCM $\leftrightarrow$ Quaternions formulas.....	46
Figure 56: representation of ECEF frame .....	46
Figure 57: ENU frame with respect to ECEF.....	47
Figure 58: body frame used representation .....	48
Figure 59: spacecraft dynamical model block diagram.....	48
Figure 60: Attitude Control block scheme.....	50
Figure 61: Earth magnetic field dipole representation .....	51
Figure 62: ECI, Body and LVLH reference frames .....	54
Figure 63: Tyvak-0092/Commtrail nanosatellite TLE .....	57
Figure 64: Orbit propagation log_orbit.txt file snippet.....	58
Figure 65: Orbit propagation LVLH_orbit.txt file snippet .....	58
Figure 66: Magnetic flux density components: Bx, By and Bz (top to bottom).....	59
Figure 67: Orbit propagator Simulink model.....	60
Figure 68: Detumbling scenario (Satellite model and environmental interactions).....	61
Figure 69: Detumbling control: "Satellite dynamical model" insight .....	62
Figure 70: Earth-pointing control: "Satellite dynamical model" insight .....	63
Figure 71: Magnetometer model.....	63
Figure 72: Magnetorquers subsystem (top) and implementation (bottom) .....	64
Figure 73: B-dot bang bang subsystem (top) and implementation (bottom) .....	64
Figure 74: B-dot bang bang dead band implementation .....	65
Figure 75: original B-dot_x (top) and "filtered" B-dot_x (bottom).....	65
Figure 76: Earth-pointing controller subsystem (top) and implementation (bottom) .....	66
Figure 77: Detumbling test 1: angular velocities.....	68
Figure 78: Detumbling test 1: control currents.....	68
Figure 79: Detumbling test 2: angular velocities.....	69
Figure 80: Detumbling test 2: control currents.....	69
Figure 81: B-dot controller simulations performance.....	70
Figure 82: Simulink detumbling model.....	71
Figure 83: controller performances comparison between Kp=300 and Kp=100.....	72
Figure 84: Earth-pointing controller, impact scenario simulation.....	73
Figure 85: Earth-pointing controller, error angle evaluation for the 5 simulations.....	74
Figure 86: Earth-pointing controller, angular velocity error for the 5 simulations.....	75
Figure 87: Simulink Earth-pointing model .....	76
Figure 88: ROS Toolbox activation in Simulink.....	77

Figure 89: XML file for enstablishing the ROS/MATLAB connection .....	78
Figure 90: ROS Toolbox auto-generation of code final report.....	79
Figure 91: ROS2 generated nodes in execution.....	79
Figure 92: Buildroot 2020 “menuconfig” menu .....	81

## LIST OF TABLES:

Table 1: HMC5883L register list .....	16
Table 2: E910.86 write and read commands used .....	19
Table 3: B-dot Dead-band tests settings.....	67
Table 4: B-dot controller, 5 simulations random initial conditions (attitude and ang. vel.) .....	70
Table 5: B-dot controller, 5 simulations various initial conditions (attitude and ang. vel.) .....	74

# 1. INTRODUCTION

## 1.1. Nanosatellites overview

In the recent years the space sector 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 different 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 in 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 can be very 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 fundamental 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.

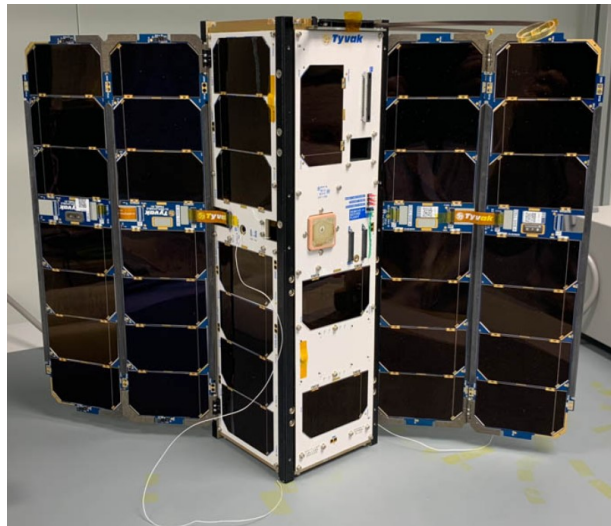


Figure 1: Tyvak's Commtrail nanosatellite (3U)

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's "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 2:

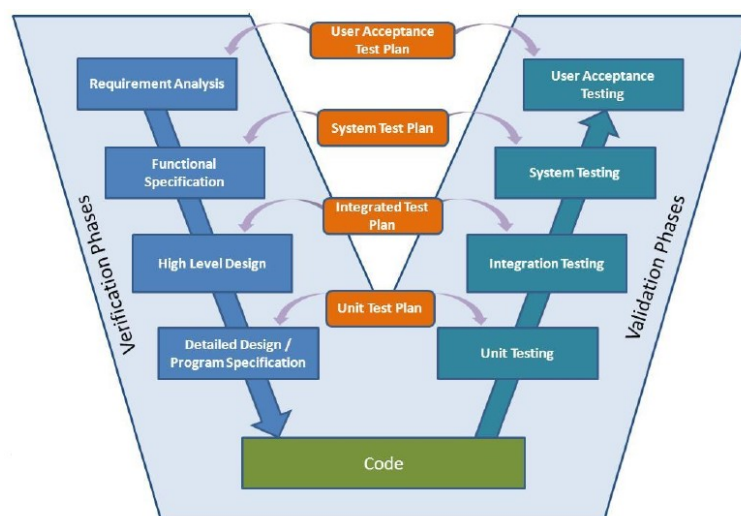


Figure 2: V-shape process flow of software design

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 a personal flight software framework for nanosatellites using ROS2 and to study the problem of Attitude Control and the auto-generation of code.

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 it is a start-up born by the American counterpart called “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*) has been 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 are reported
- Chapter 4: the mathematical tools and the actuators used for the attitude control are presented and explained. Then the scenarios of “detumbling” phase and “Earth-pointing”

task performed with a suitable control system are addressed.

- Chapter 5: the MATLAB/Simulink simulations for testing the desired control algorithms are described and the obtained results presented. Finally, the auto generation of the code for the control system is performed.
- Chapter 6: some personal conclusions about the project and suggestions for future improvements and developments.

## 2. SYSTEM ARCHITECTURE

### 2.1. Project overview

The main objective of the thesis is to realize a first version of a new flight software framework, based on ROS2, and in order to study in detail 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 understanding 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 3, 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
<b>SPI DC Characteristics, output terminal MISO</b>							
1	Output voltage low	$I = 0.5\text{mA}$	$V_{\text{MISOL}}$			0.4	V
2	Output voltage high	$I = -0.2\text{mA}$	$V_{\text{MISOH}}$	$V_{\text{DD}} - 0.4$			V

Figure 3: E91086 MISO output voltages.  $V_{\text{dd}}=4.5\text{V}$  to  $5.5\text{V}$

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

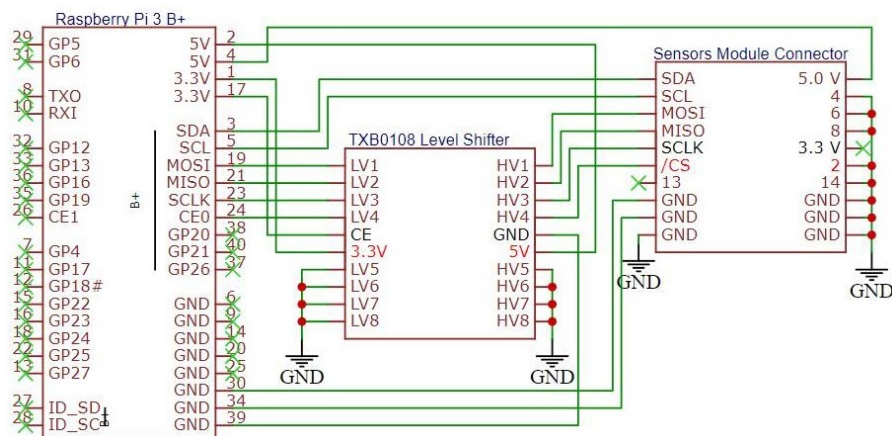


Figure 4: Components connections circuit diagram

This is the reference used for performing all the connections between the components and the real implementation is reported in Figure 23.

### 2.2.1. Temperature sensor AD7415

The AD7415 sensor is a standalone digital temperature sensor, widely used in several fields of applications, from automotive to aerospace, 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:

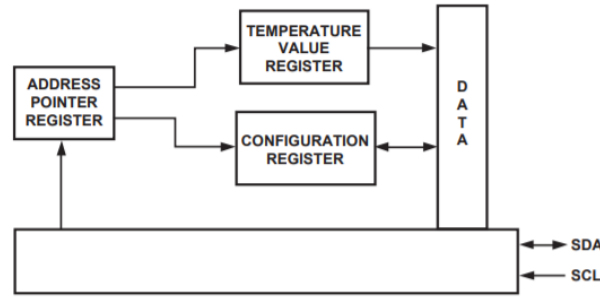


Figure 5: AD7415 Register structure

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

<sup>1</sup> Default settings at power-up.

D7	D6	D5	D4	D3	D2	D1	D0
PD	FLTR	TEST MODE		ONE SHOT		TEST MODE	
0 <sup>1</sup>	1 <sup>1</sup>	0s <sup>1</sup>		0s <sup>1</sup>		0s <sup>1</sup>	

Figure 6: 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 “read” operations 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 7: AD7415 Temperature value register readings output

As written in Figure 7 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, the following function can be used:

$$Temperature[^{\circ}C] = \frac{Raw\_digital\_temperature_{[decimal]}}{4}$$

the value of the temperature in °C is obtained.  
The circuit diagram of the sensor is reported in Figure 8.

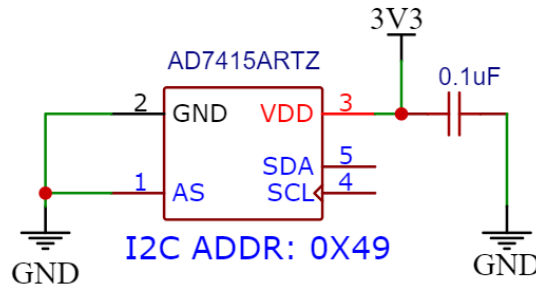


Figure 8: AD7415 circuit diagram

### 2.2.1.1. AD7415 test

In this section some tests are performed in order to check the correct behaviour of the sensor and the software drivers used for interfacing with it.

The first test consists in an easy “read” operation and to display the sensed temperature in degrees [°C]. The sensor is left still on the table in the company office, so we expect to read a value around 20~23 °C.

```
ubuntu@ubuntu: ~/ros2_ws
File Edit View Search Terminal Help
ubuntu@ubuntu:~/ros2_ws$ ros2 run sensors sensors_reader_i2c bus1
Reading data from I2C bus1 ...
Temperature: 23.5 [°C]
Temperature: 23.0 [°C]
Temperature: 23.25 [°C]
Temperature: 23.25 [°C]
Temperature: 23.5 [°C]
Temperature: 23.0 [°C]
Temperature: 23.25 [°C]
Temperature: 23.0 [°C]
```

Figure 9: AD7415 sensor test

As we can see from the picture above the temperature has been properly read (with a frequency of 0.5 Hz), and its values are the expected ones. Obviously the sensor is affected by noise and so the outputs are oscillating around 23.25 °C. Let's now see what happens if the sensor module is heated, for example by holding it in an hand.

```

ubuntu@ubuntu: ~/ros2_ws
File Edit View Search Terminal Help
ubuntu@ubuntu:~/ros2_ws$ ros2 run sensors sensors_reader_i2c bus1
Reading data from I2C bus1 ...
Temperature: 28.75 [°C]
Temperature: 28.25 [°C]
Temperature: 28.5 [°C]
Temperature: 28.25 [°C]
Temperature: 29.25 [°C]
Temperature: 29.25 [°C]
Temperature: 29.5 [°C]
Temperature: 29.5 [°C]
Temperature: 29.75 [°C]
Temperature: 29.75 [°C]
Temperature: 29.75 [°C]
Temperature: 30.0 [°C]
Temperature: 30.0 [°C]

```

*Figure 10: AD7415 sensor heated test*

As expected the temperature is increased, till reaching 30 °C, due to the heating of the sensor module at contact with an higher temperature “object”.

## 2.2.2. 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  $\pm 8$  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 the table below:

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

*Table 1: 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 this mode the device is expected to continuously perform measurements and to place the result in the data register at each iteration.

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 11: HMC5883L channel X data output registers A and B

Taking for example the A and B output registers of the X channel (in the figure above) is possible to see that each register contains 8-bit (the number in the 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 from “0xF800” address to “0x07FF” address.

The circuit diagram of the sensor is reported below.

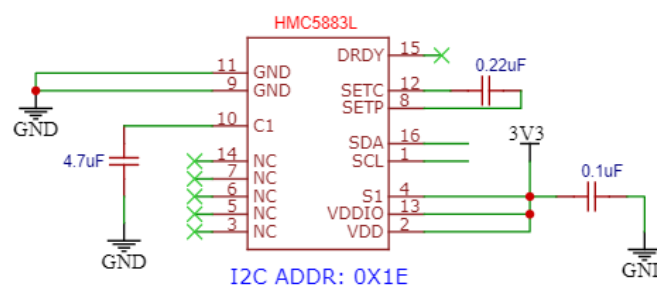


Figure 12: HMC5883L circuit diagram

### 2.2.2.1. HMC5883L test

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

Obviously is difficult to say if this values are the correct ones since the magnetometer is

measuring the magnetic field present in the desk of the company office, so there may be various noises affecting the measurements. The shell in which the results are printed is reported in Figure 13.

[illegible]

Figure 13: HMC5883L sensor test

### 2.1.1. Sun sensor E910.86

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

- The angle of light incidence in both XZ ( $X_n$ ) and YZ ( $Y_n$ ) 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  $X_n$  angle, with respect to the magnetometer reference frame, can be seen in Figure 14:

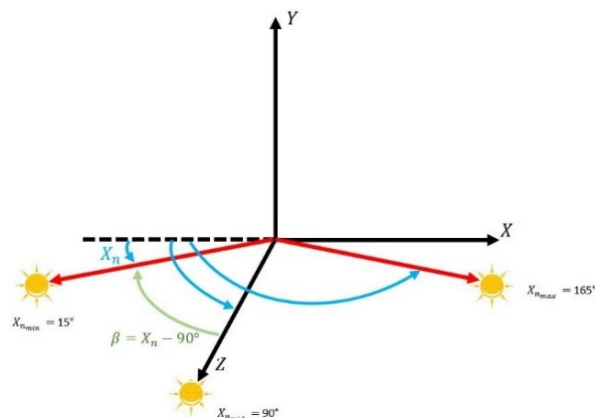


Figure 14: physical model of  $X_n$  angle

Obviously the same model can be used for the  $Y_n$  angle by considering 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. The datasheet of the sensor states that “Read” commands start with a “00” and “write” commands start with “10” while the SPI response word always starts with “01”. According to the sensor datasheet, the commands used in order to initialize the sensor and to perform a “read” operation are:

Command	Operation	SPI response	Data
10x100XXYYPSZDDD	Write E910.86 and analog output status	011100XXYYPSZDDD	E910.86 and analog output status
X0x000xxxxxxxxxx	Read $X_n$ and $Y_n$ sensor angle data	0100X <sub>5</sub> X <sub>4</sub> X <sub>3</sub> X <sub>2</sub> X <sub>1</sub> X <sub>0</sub> Y <sub>5</sub> Y <sub>4</sub> Y <sub>3</sub> Y <sub>2</sub> Y <sub>1</sub> Y <sub>0</sub>	X and Y sensor data Y <sub>n</sub> = angle yz-plane X <sub>n</sub> =angle xz-plane

Table 2: 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  $X_n$  and  $Y_n$  data obtaining the following digital value: X<sub>5</sub>X<sub>4</sub>X<sub>3</sub>X<sub>2</sub>X<sub>1</sub>X<sub>0</sub>Y<sub>5</sub>Y<sub>4</sub>Y<sub>3</sub>Y<sub>2</sub>Y<sub>1</sub>Y<sub>0</sub>.

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 15.

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

Is important to note that the angles value can span from a minimum of 15° to a maximum of 165° with a resolution of 2.7°.

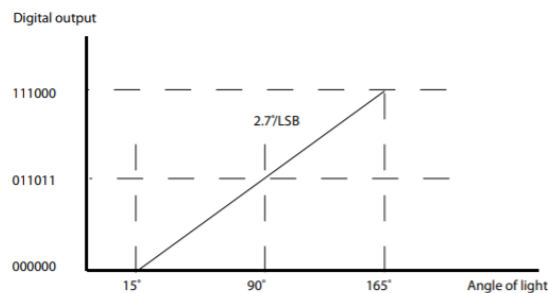


Figure 15: digital output – angles relation

Once the conversions are computed, the resulting values are the  $X_n$  and  $Y_n$  angles (in radians). Now that these angles are known is possible to compute the sun vector referring to the model depicted in the figure below.

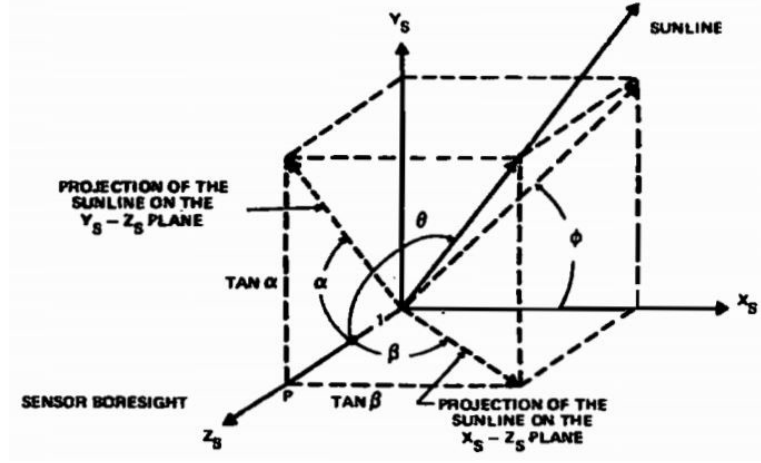


Figure 16: sun vector model

Referring to the Figure 14, the angle  $\beta$  can be computed as:  $\beta = X_n - 90^\circ$ . In this way, when  $X_n$  is ranging from  $15^\circ$  (the minimum value that can be obtained from the sensor) to  $90^\circ$ , the value of  $\beta$  is negative; instead, when  $X_n$  is ranging from  $90^\circ$  to  $165^\circ$  (the maximum value that can be obtained from the sensor),  $\beta$  is positive. In this way we are setting as our “0°” angle the output value of  $90^\circ$ , that is sensed when the light is positioned right in front of the sensor as is possible to see in the following tests. Obviously the same model can be used for the angle  $Y_n$ , using angle  $\alpha$  instead of  $\beta$ .

In this way, using the values of  $\alpha$  and  $\beta$ , and referring to the picture in Figure 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 resulting vector is not normalized because the third component is always set to “1”. Since only the direction of the vector is in general required for several purposes, for example the “Attitude determination” based on the information of the position of the sun with respect to the spacecraft, the vector obtained from the previous formula can must be normalized. Finally the circuit diagram of the sensor, used for performing all the connection with the rest of the hardware, is reported in Figure 17.

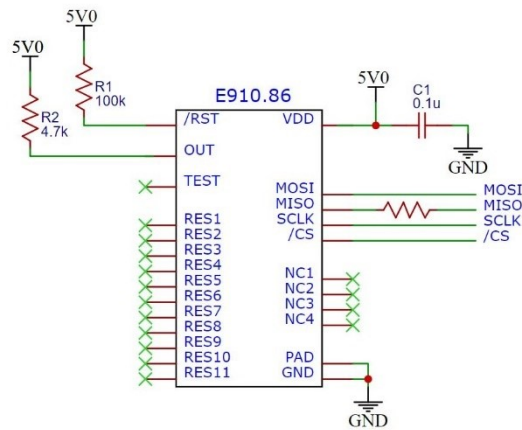


Figure 17: E910.86 circuit diagram

### 2.1.1.1. E910.86 test

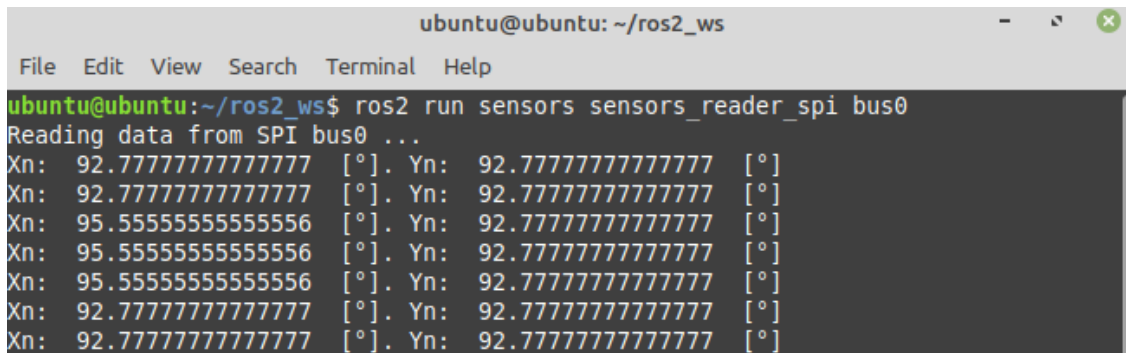
As stated in the previous section, the output values provided by the Sun sensor are the angles described by the incident light in the XZ and YZ planes (respectively named  $X_n$  and  $Y_n$ ) of the sensor reference frame. Knowing this information a first test has been performed in order to check if the sensor correctly measures these angles. A simple situation is selected in order to easily verify if the output values are correct or not, indeed the sensor is needed to sense an angle of  $90^\circ$  on both XZ and YZ plane when a light is positioned in front of it, as it is shown below:



Figure 18: E910.86 testing setup

Knowing that the measured angle when the light is right in front of the sensor should be  $90^\circ$ , the

outputs obtained from this experiment, presented in Figure 19, are compared with the expected ones.



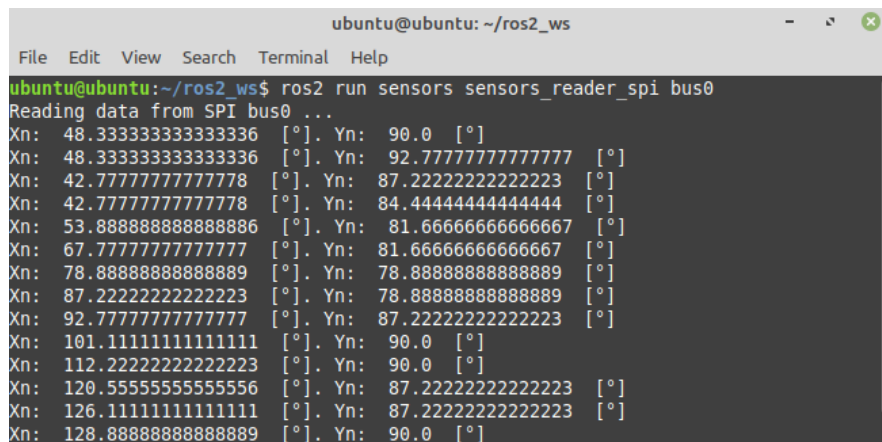
```

ubuntu@ubuntu: ~/ros2_ws
File Edit View Search Terminal Help
ubuntu@ubuntu:~/ros2_ws$ ros2 run sensors sensors_reader_spi bus0
Reading data from SPI bus0 ...
Xn: 92.77777777777777 [°]. Yn: 92.77777777777777 [°]
Xn: 92.77777777777777 [°]. Yn: 92.77777777777777 [°]
Xn: 95.55555555555556 [°]. Yn: 92.77777777777777 [°]
Xn: 95.55555555555556 [°]. Yn: 92.77777777777777 [°]
Xn: 95.55555555555556 [°]. Yn: 92.77777777777777 [°]
Xn: 92.77777777777777 [°]. Yn: 92.77777777777777 [°]
Xn: 92.77777777777777 [°]. Yn: 92.77777777777777 [°]

```

Figure 19: E910.86 Xn, Yn angles at 90°

As it can be seen, the output angles are correct and this proves the proper behaviour of the sensor, but there is an error of 2.7° affecting the measurements and this is due to the resolution of the sensor. Now a second test is performed by moving the light along the X axis of the Sun sensor reference frame, in particular from the left side of the sensor to the right side, and checking if the Xn angle changes accordingly. The results are presented in Figure 20:



```

ubuntu@ubuntu: ~/ros2_ws
File Edit View Search Terminal Help
ubuntu@ubuntu:~/ros2_ws$ ros2 run sensors sensors_reader_spi bus0
Reading data from SPI bus0 ...
Xn: 48.33333333333336 [°]. Yn: 90.0 [°]
Xn: 48.33333333333336 [°]. Yn: 92.77777777777777 [°]
Xn: 42.77777777777778 [°]. Yn: 87.22222222222223 [°]
Xn: 42.77777777777778 [°]. Yn: 84.44444444444444 [°]
Xn: 53.88888888888886 [°]. Yn: 81.66666666666667 [°]
Xn: 67.77777777777777 [°]. Yn: 81.66666666666667 [°]
Xn: 78.88888888888889 [°]. Yn: 78.88888888888889 [°]
Xn: 87.22222222222223 [°]. Yn: 78.88888888888889 [°]
Xn: 92.77777777777777 [°]. Yn: 87.22222222222223 [°]
Xn: 101.11111111111111 [°]. Yn: 90.0 [°]
Xn: 112.22222222222223 [°]. Yn: 90.0 [°]
Xn: 120.55555555555556 [°]. Yn: 87.22222222222223 [°]
Xn: 126.11111111111111 [°]. Yn: 87.22222222222223 [°]
Xn: 128.88888888888889 [°]. Yn: 90.0 [°]

```

Figure 20: E910.86 Xn changing test

The obtained results are correct since the Xn values are changing going from lower values to higher ones (due to the movement of the light). The Yn angle is correctly maintained to a value of approximately 90° but the precision is about 10° since the light is moved by hand and some changes also on the YZ planes are encountered due to the movement that is not perfectly fixed to the Y axis.

Finally the very same test is performed by moving the light along the Y axis to check if the Yn values are correctly changing during the movements of the light. The results are presented in Figure 21:

```

ubuntu@ubuntu: ~/ros2_ws
File Edit View Search Terminal Help
ubuntu@ubuntu:~/ros2_ws$ ros2 run sensors sensors_reader_spi bus0
Reading data from SPI bus0 ...
Xn: 95.55555555555556 [°]. Yn: 53.888888888888886 [°]
Xn: 92.77777777777777 [°]. Yn: 53.888888888888886 [°]
Xn: 95.55555555555556 [°]. Yn: 53.888888888888886 [°]
Xn: 95.55555555555556 [°]. Yn: 73.33333333333334 [°]
Xn: 98.33333333333333 [°]. Yn: 84.44444444444444 [°]
Xn: 98.33333333333333 [°]. Yn: 92.77777777777777 [°]
Xn: 98.33333333333333 [°]. Yn: 98.33333333333333 [°]
Xn: 98.33333333333333 [°]. Yn: 109.44444444444444 [°]
Xn: 95.55555555555556 [°]. Yn: 115.0 [°]
Xn: 95.55555555555556 [°]. Yn: 120.55555555555556 [°]
Xn: 95.55555555555556 [°]. Yn: 128.88888888888889 [°]

```

Figure 21: E910.86 Xn changing test

Also for this test the results are the expected ones and so the sensor is working properly even if the light source is fixed or moving along the axis of the sensor reference frame.

## 2.1.2. Raspberry Pi 3 B+

The Raspberry Pi 3 B+ is a widely used 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 22 .

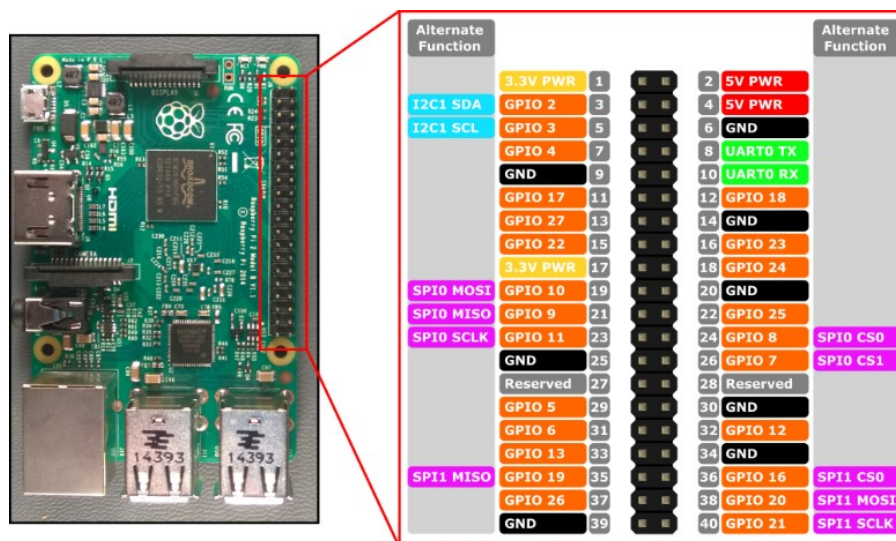
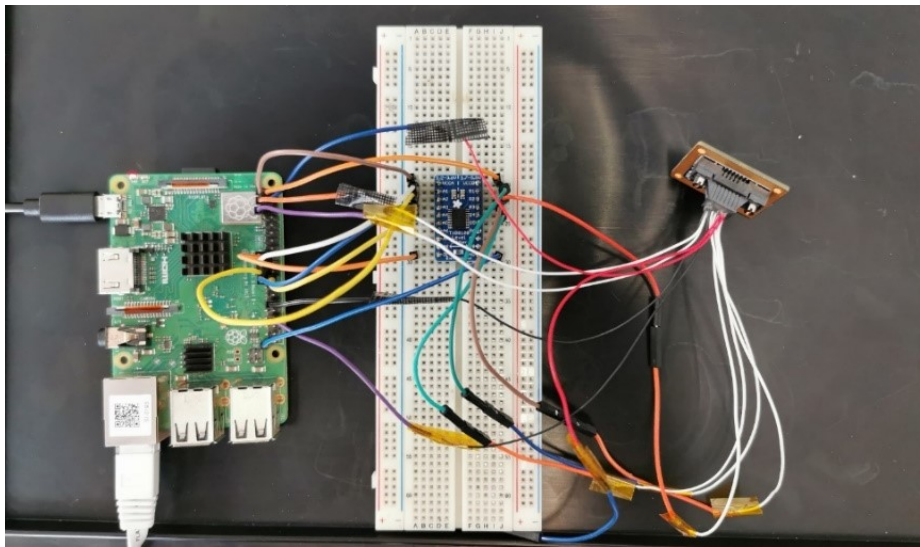


Figure 22: Raspberry Pi 3 B+ board and GPIO scheme.

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 as proposed in the schematic of Figure 4, resulting in this real circuit:



*Figure 23: final circuit with: Raspberry PI, logic level converter and sensor module*

## 2.2. Software configuration

In the following section is presented the software configuration used for developing the thesis work.

As presented in section 2.1.2, 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. This approach is explained in the Appendix A [8].

- 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.




### 2.2.1. 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, generally 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 drones or in this case 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 24 :

Distro	Release date	Logo	EOL date
<a href="#">Foxy Fitzroy</a>	June 5th, 2020		May 2023
<a href="#">Eloquent Elusor</a>	Nov 22nd, 2019		Nov 2020
<a href="#">Dashing Diademata</a>	May 31st, 2019		May 2021

*Figure 24: ROS2 latest distributions and EOL dates*

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

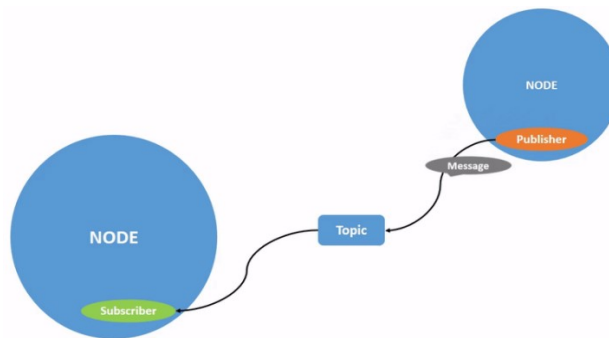
- It is totally open-source and constantly updated by developers all around the world for many application fields.
- Creating truly robust, modular and efficient robot/mechatronics software is in general an hard job, so ROS provides plug-and-play solutions to common problems in developing software frameworks.
- 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 for testing and code auto-generation.

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 and analyze ROS/ROS2

messages sent and received from specific topics.

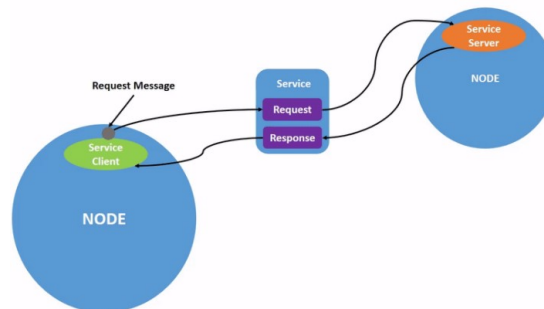
At the heart of any ROS 2 system is the ROS graph. The ROS graph refers to the network of nodes in a ROS system and the connections between them by which they are able to communicate. This graph is made up by the elementary concepts of ROS that are:

- **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. Obviously the core concept at the basis of the nodes is the modularity of the system, indeed using nodes is very simple to add functionalities just by integrating it in the already present ROS network.
- **Topics:** each topic has a name and a specific kind of message that it can handle. They are the principal and simplest “hubs” where messages are collected, when sent by publisher nodes, and sent to subscriber nodes.



*Figure 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 here:



*Figure 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 27 extracted from the official ROS2 tutorial.

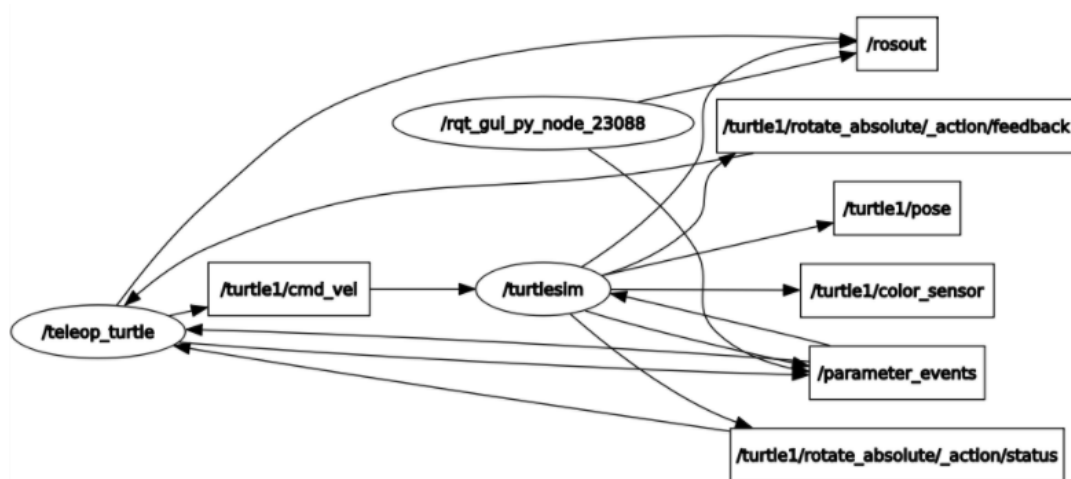


Figure 27: *rqt\_graph* of the official teleop turtle tutorial

## 3. ROS2 FLIGHT SOFTWARE FRAMEWORK

In this chapter is described the proposed solution for the fundamental nodes implemented for a draft of the 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.

### 3.1. 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 28.

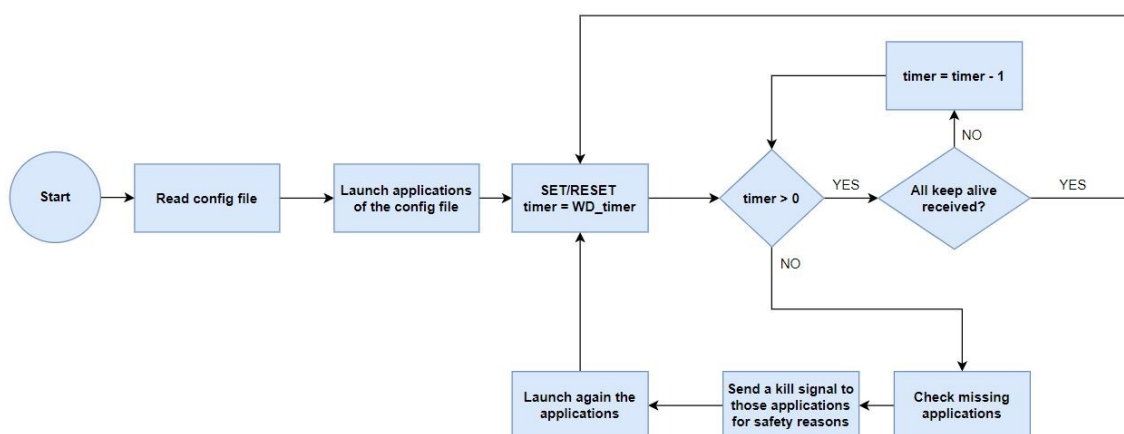


Figure 28: Mark-1 watchdog flow chart

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 29:

Watchdog (Node)
guarded_nodes: dictionary
active_node_names_list: list
watchdog_launcher(launch_path)
create_active_nodes_names_list()
checking_missing_nodes()
watchdog_callback()

Figure 29: Watchdog(Node) class

The flow chart of the developed ROS2 based watchdog is presented in the figure below:

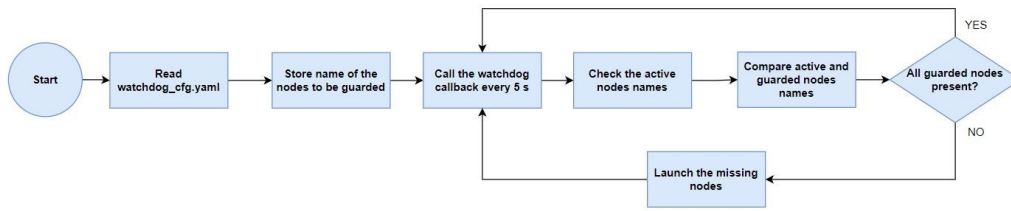


Figure 30: ROS2 based watchdog flow chart

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 here:

```

1 guarded_nodes:
2   node1:
3     name: 'i2c_bus1'
4     launch_path: '/home/ubuntu/ros2_ws/src/sensors/launch/i2c_bus1_launch.py'
5   node2:
6     name: 'spi_bus0'
7     launch_path: '/home/ubuntu/ros2_ws/src/sensors/launch/spi_bus0_launch.py'

```

Figure 31: watchdog config YAML

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 or 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.

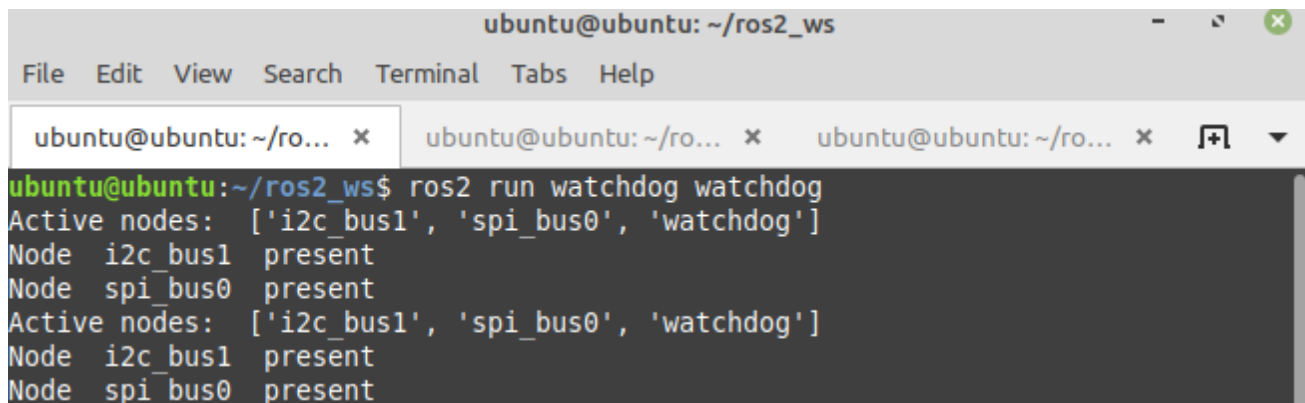
The Watchdog node can be executed through the “*ros2 run*” command via shell.

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.

### 3.1.1. Watchdog node test

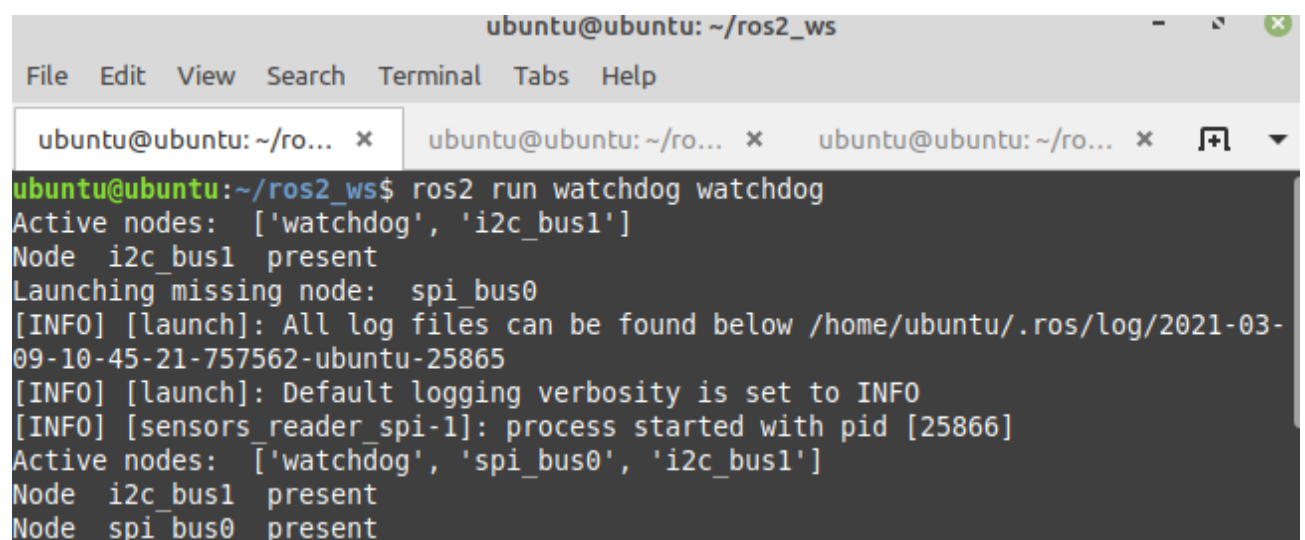
In order to check the correct performances of the designed Watchdog node, some tests are performed. The first situation is the one in which all the node that must be guarded from the Watchdog are already running, and the Watchdog just needs to acknowledge this and to print a message with the list of the active nodes. The results obtained from this scenario are presented in Figure 32:

A terminal window titled 'ubuntu@ubuntu: ~/ros2\_ws' with a menu bar (File, Edit, View, Search, Terminal, Tabs, Help). The terminal shows the command 'ros2 run watchdog watchdog' being executed. The output lists active nodes as ['i2c\_bus1', 'spi\_bus0', 'watchdog'] and confirms the presence of 'i2c\_bus1' and 'spi\_bus0' nodes.

```
ubuntu@ubuntu: ~/ros2_ws
File Edit View Search Terminal Tabs Help
ubuntu@ubuntu: ~/ro... x  ubuntu@ubuntu: ~/ro... x  ubuntu@ubuntu: ~/ro... x
ubuntu@ubuntu:~/ros2_ws$ ros2 run watchdog watchdog
Active nodes: ['i2c_bus1', 'spi_bus0', 'watchdog']
Node i2c_bus1 present
Node spi_bus0 present
Active nodes: ['i2c_bus1', 'spi_bus0', 'watchdog']
Node i2c_bus1 present
Node spi_bus0 present
```

*Figure 32: Watchdog node test: all the guarded nodes are running*

The Watchdog correctly print a list of the active nodes (including itself) and a message that shows that the sensors nodes are correctly running, so additional operations are not required. 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 recognize the missing node and to start this node up. The results are presented below:

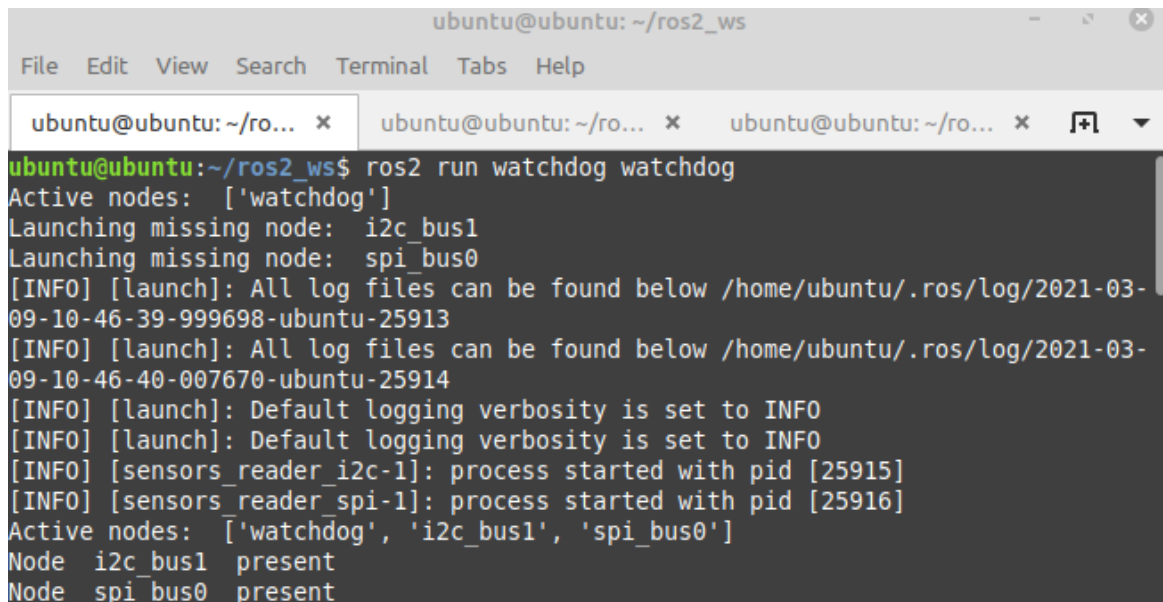
A terminal window titled 'ubuntu@ubuntu: ~/ros2\_ws' with a menu bar (File, Edit, View, Search, Terminal, Tabs, Help). The terminal shows the command 'ros2 run watchdog watchdog' being executed. The output lists active nodes as ['watchdog', 'i2c\_bus1'], identifies 'spi\_bus0' as a missing node, and shows the process being launched with pid [25866]. The final output lists active nodes as ['watchdog', 'spi\_bus0', 'i2c\_bus1'] and confirms the presence of 'i2c\_bus1' and 'spi\_bus0' nodes.

```
ubuntu@ubuntu: ~/ros2_ws
File Edit View Search Terminal Tabs Help
ubuntu@ubuntu: ~/ro... x  ubuntu@ubuntu: ~/ro... x  ubuntu@ubuntu: ~/ro... x
ubuntu@ubuntu:~/ros2_ws$ ros2 run watchdog watchdog
Active nodes: ['watchdog', 'i2c_bus1']
Node i2c_bus1 present
Launching missing node: spi_bus0
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2021-03-09-10-45-21-757562-ubuntu-25865
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [sensors_reader_spi-1]: process started with pid [25866]
Active nodes: ['watchdog', 'spi_bus0', 'i2c_bus1']
Node i2c_bus1 present
Node spi_bus0 present
```

*Figure 33: Watchdog node test: SPI sensors reader node is 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 contains 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 proceeds normally.

The last scenario is the one in which both nodes are not running and Watchdog needs to start them up. This test is performed in order to check that the Watchdog can start more nodes simultaneously when requested. The results are presented in Figure 34:



```
ubuntu@ubuntu: ~/ros2_ws
File Edit View Search Terminal Tabs Help
ubuntu@ubuntu: ~/ro... x  ubuntu@ubuntu: ~/ro... x  ubuntu@ubuntu: ~/ro... x  [+]
```

```
ubuntu@ubuntu:~/ros2_ws$ ros2 run watchdog watchdog
Active nodes: ['watchdog']
Launching missing node: i2c_bus1
Launching missing node: spi_bus0
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2021-03-09-10-46-39-999698-ubuntu-25913
[INFO] [launch]: All log files can be found below /home/ubuntu/.ros/log/2021-03-09-10-46-40-007670-ubuntu-25914
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [sensors_reader_i2c-1]: process started with pid [25915]
[INFO] [sensors_reader_spi-1]: process started with pid [25916]
Active nodes: ['watchdog', 'i2c_bus1', 'spi_bus0']
Node i2c_bus1 present
Node spi_bus0 present
```

*Figure 34: Watchdog node test: all the guarderd nodes are 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 PIDs. When the callback is called for the second time, all the nodes are correctly present and the execution process proceed normally.

## 3.2. 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.

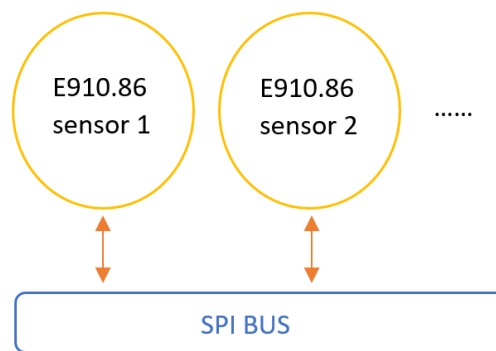
In the case presented in this thesis there is a sensor module, instrumented with several sensors, that can communicate with an external device by means of dedicated buses, and in particular: the AD7415 temperature sensor and the HMC5883L magnetometer can be interfaced through an I2C bus, while the E910.86 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 35: 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 35) or magnetometers collecting data for attitude determination. In cases like these the solution “one node one sensor” is not so optimal from 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 at very high frequencies, containing the collected data.

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 35, 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 is 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 36:

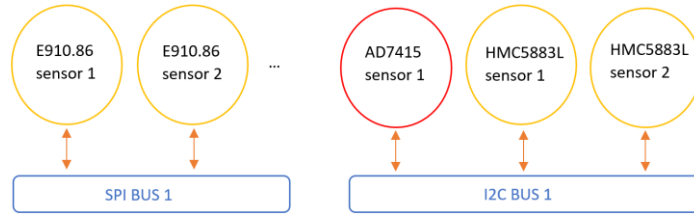


Figure 36: Realistic situation with many sensors on two different buses

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.

### 3.2.1. 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 :

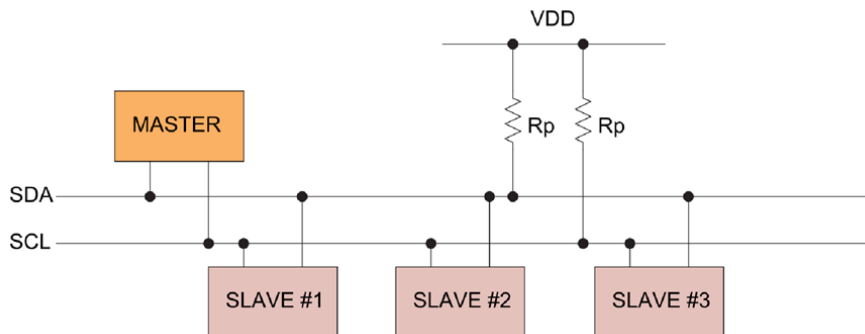


Figure 37: I2C protocol representation

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.

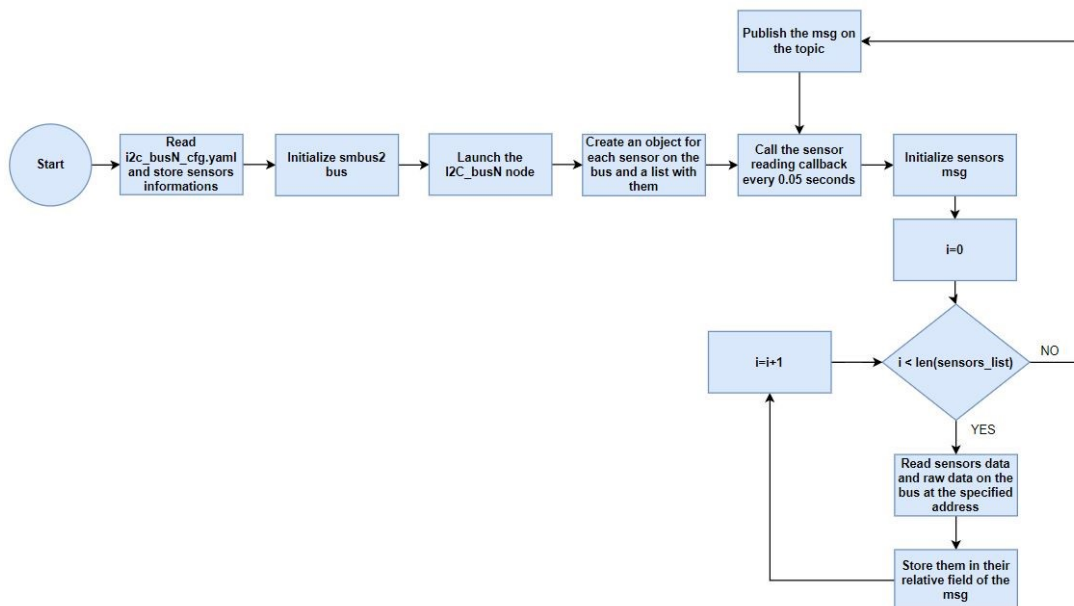
Since the I2C bus node must acts like a publisher and send a message that contains the sensors data read on a dedicated topic, a custom message that can contain these informations must be created. All the custom messages created for this thesis work are contained in a suitable folder. The structure of sensors message is presented in Figure 38.

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

*Figure 38: 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 39:



*Figure 39: I2C bus node flow chart*

Its attributes and methods are then presented in Figure 40:

I2C_bus(Node)
bus: smbus2 bus object
sensors_info: dictionary
n_bus: int
sens: objects list
sensor_reading()

Figure 40: I2C bus node class

After the node is launched, it reads the configuration file (written in YAML) presented below:

```
1  n_bus: 1
2  sensors:
3    sensor1:
4      type: 'temp'
5      addr: 0x49
6    sensor2:
7      type: 'mag'
8      addr: 0x1E
```

Figure 41: 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.

### 3.2.2. 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 the following figure:

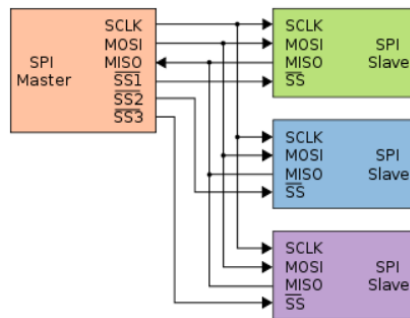


Figure 42: SPI communication protocol example with a Master and three slaves

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 (3.2.1 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 43 Figure 44. 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:

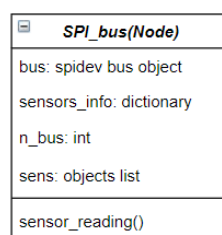


Figure 43: SPI\_bus node class diagram

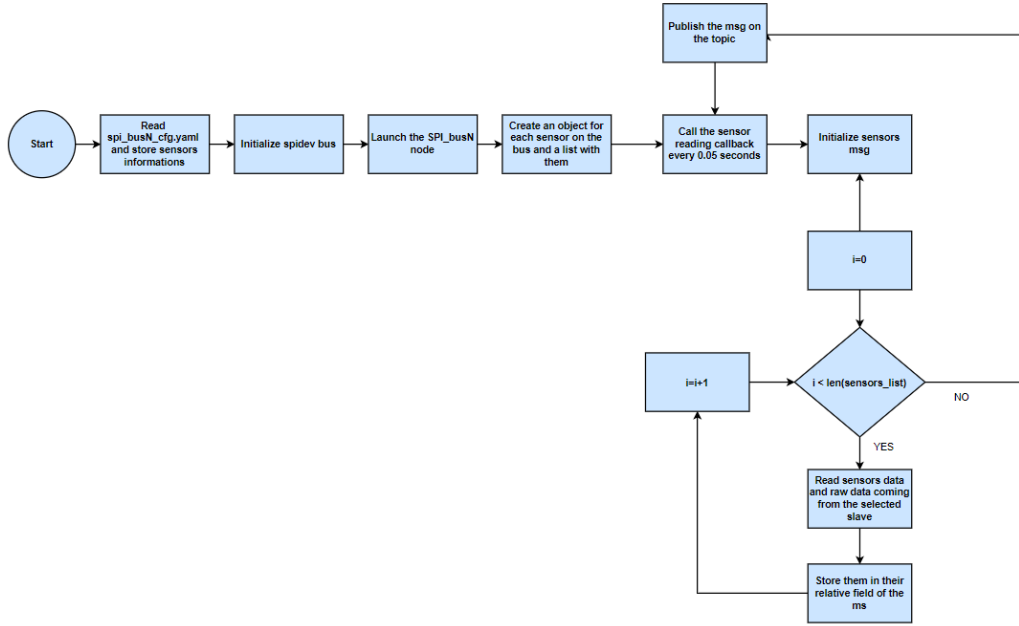


Figure 44: SPI\_bus node execution flowchart

```

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

```

Figure 45: SPI\_bus node configuration file

### 3.3. 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 or 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 become 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 46:

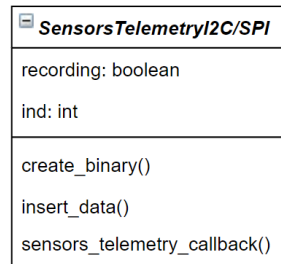


Figure 46: I2C/SPI bus sensors telemetry class

The flow chart of an I2C/SPI bus telemetry node is shown in Figure 47:

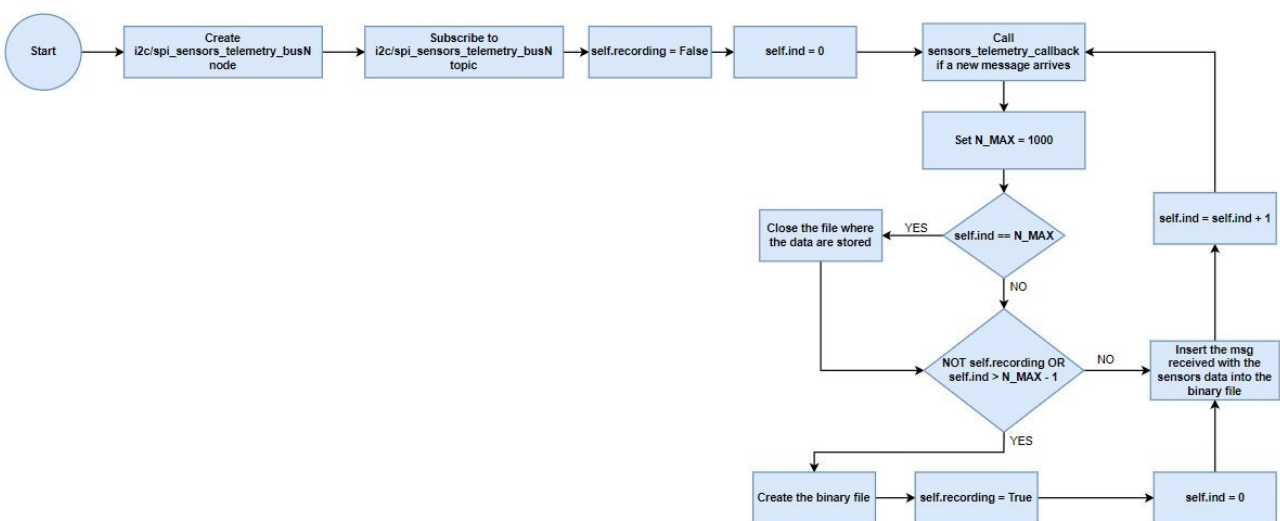


Figure 47: I2C/SPI bus sensors telemetry class

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.

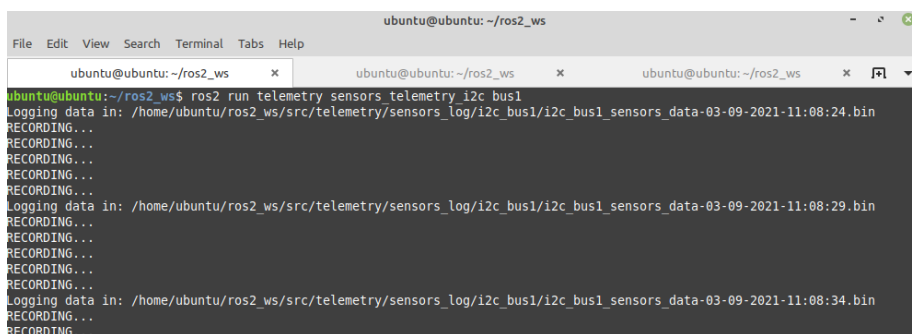
The operations performed when the callback is called are:

- 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.

### 3.3.1. Telemetry node test

Since the behaviour is the same for both I2C and SPI nodes, only the I2C telemetry node is considered for testing. 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 below:



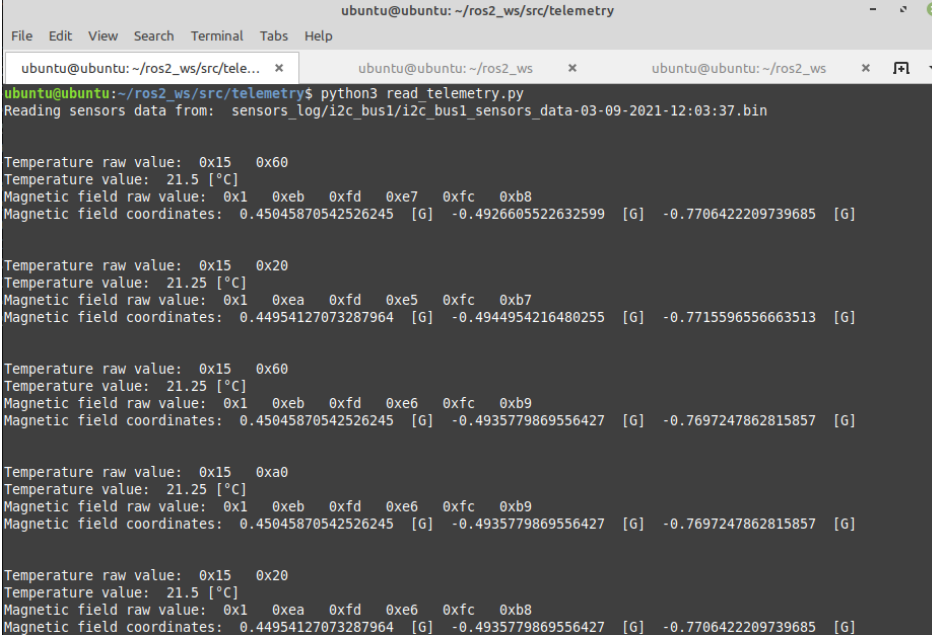
```
ubuntu@ubuntu: ~/ros2_ws
File Edit View Search Terminal Tabs Help
ubuntu@ubuntu: ~/ros2_ws x ubuntu@ubuntu: ~/ros2_ws x ubuntu@ubuntu: ~/ros2_ws x [?] v
ubuntu@ubuntu:~/ros2_ws$ ros2 run telemetry sensors_telemetry_i2c bus1
Logging data in: /home/ubuntu/ros2_ws/src/telemetry/sensors_log/i2c_bus1/i2c_bus1_sensors_data-03-09-2021-11:08:24.bin
RECORDING...
RECORDING...
RECORDING...
RECORDING...
RECORDING...
Logging data in: /home/ubuntu/ros2_ws/src/telemetry/sensors_log/i2c_bus1/i2c_bus1_sensors_data-03-09-2021-11:08:29.bin
RECORDING...
RECORDING...
RECORDING...
RECORDING...
RECORDING...
Logging data in: /home/ubuntu/ros2_ws/src/telemetry/sensors_log/i2c_bus1/i2c_bus1_sensors_data-03-09-2021-11:08:34.bin
RECORDING...
```

Figure 48: Telemetry node test: creation of a new file

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, 5 as we can see from the picture, 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 :



```

ubuntu@ubuntu: ~/ros2_ws/src/telemetry
File Edit View Search Terminal Tabs Help

ubuntu@ubuntu: ~/ros2_ws/src/tele... x  ubuntu@ubuntu: ~/ros2_ws x  ubuntu@ubuntu: ~/ros2_ws x  [?] v

ubuntu@ubuntu:~/ros2_ws/src/telemetry$ python3 read_telemetry.py
Reading sensors data from: sensors_log/i2c_bus1/i2c_bus1_sensors_data-03-09-2021-12:03:37.bin

Temperature raw value: 0x15 0x60
Temperature value: 21.5 [°C]
Magnetic field raw value: 0x1 0xeb 0xfd 0xe7 0xfc 0xb8
Magnetic field coordinates: 0.45045870542526245 [G] -0.4926605522632599 [G] -0.7706422209739685 [G]

Temperature raw value: 0x15 0x20
Temperature value: 21.25 [°C]
Magnetic field raw value: 0x1 0xea 0xfd 0xe5 0xfc 0xb7
Magnetic field coordinates: 0.44954127073287964 [G] -0.4944954216480255 [G] -0.7715596556663513 [G]

Temperature raw value: 0x15 0x60
Temperature value: 21.25 [°C]
Magnetic field raw value: 0x1 0xeb 0xfd 0xe6 0xfc 0xb9
Magnetic field coordinates: 0.45045870542526245 [G] -0.4935779869556427 [G] -0.7697247862815857 [G]

Temperature raw value: 0x15 0xa0
Temperature value: 21.25 [°C]
Magnetic field raw value: 0x1 0xeb 0xfd 0xe6 0xfc 0xb9
Magnetic field coordinates: 0.45045870542526245 [G] -0.4935779869556427 [G] -0.7697247862815857 [G]

Temperature raw value: 0x15 0x20
Temperature value: 21.5 [°C]
Magnetic field raw value: 0x1 0xea 0xfd 0xe6 0xfc 0xb8
Magnetic field coordinates: 0.44954127073287964 [G] -0.4935779869556427 [G] -0.7706422209739685 [G]

```

Figure 49: Telemetry node test: reading stored data

## 4. ATTITUDE CONTROL

When a spacecraft, or in general an autonomous system, is asked to perform some actions and interact with an environment, there is always the problem of determining its position (in the orbit) in the space and its attitude (orientation with respect to a ref. frame). These two informations are fundamental and needed 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 the orbit 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 50:

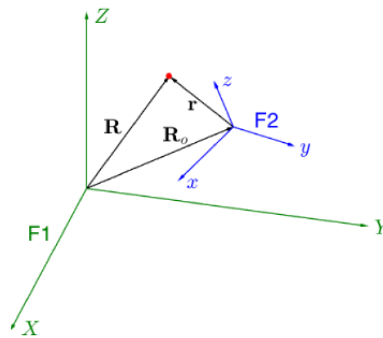


Figure 50: F1, F2 reference frames and a generic particle

There is a generic particle (red dot in the figure) and two reference frames (F1 green, F2 blue) 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:

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

Figure 51: 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  $\mathbf{R}$  in this way:

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

↓

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

Figure 52:  $R$  written in matrix form in function of (x,y,z)

As is possible to see from the relation above (Figure 52), each element of the  $T$  matrix is a dot product between the  $F_1$  and  $F_2$  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  $\mathbf{R}_0$  vector is only referred to the translation between the reference frames.

The DCM  $T$  can be interpreted in two ways, and is fundamental to always 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 are 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:

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

Figure 53: 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  $\beta$  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 parts 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)

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

dot product

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

cross product

$$\mathbf{q} \times \mathbf{p} = \begin{bmatrix} q_2 p_3 - q_3 p_2 \\ q_3 p_1 - q_1 p_3 \\ q_1 p_2 - q_2 p_1 \end{bmatrix}$$

Figure 54: 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 55: DCM  $\leftrightarrow$  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 exists 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° (passing through Greenwich meridian) 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 the figure below:

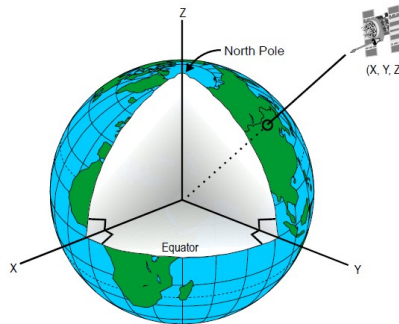


Figure 56: representation of ECEF frame

This frame rotates with respect to the stars because it is tied with the Earth and so it is a non inertial frame (with respect to the stars). ECEF reference frame is in general used for its simplicity in describing the motion of objects that are moving on the Earth's surface.

- **ECI (*Earth Centred Inertial*)** frame: has its origin at the centre of mass of the Earth, like the ECEF frame, and its axes lay on the same planes of the ECEF frame but it is fixed with respect to the stars and so it is considered inertial (with respect to the stars). 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, the y axis completes the right-handed triad.
- **LVLH (*Local vertical, local horizontal coordinates*)**: is a geographical coordinate system based on the tangent plane defined by the local vertical direction and Earth's axis of rotation. The axes are positioned as follows: one axis is pointed towards the northern pole, one along the local eastern axis and one 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 57:

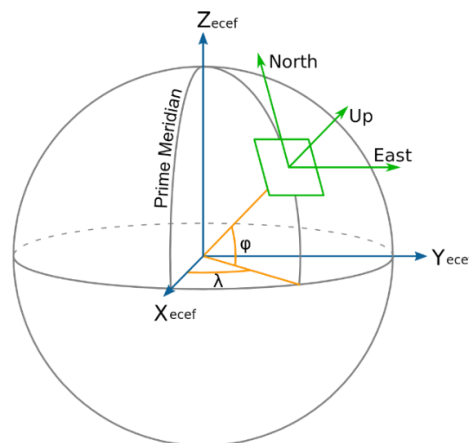
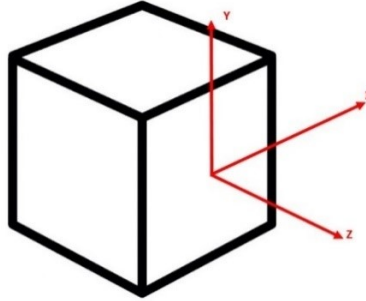


Figure 57: *ENU frame with respect to ECEF*

- **Body-fixed** frames: these frames are tied to a predefined body and move/rotate with it. The axes can be placed as wanted accordingly to the simplifications that may occur in orienting the frame in a certain manner or another, and it is centered in the center of mass

of the body. Considering the system of this thesis work, the body frame considered is the one coincident with the sun sensor E910.86 frame, used to provide the sun coordinates and it is presented in Figure 58.

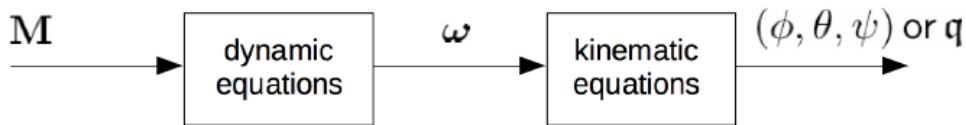


*Figure 58: body frame used representation*

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. Satellite dynamical model

In this paragraph the derivation of a suitable dynamical model for representing a nanosatellite is presented. Generally a spacecraft can be approximately described as a rigid body moving with respect to an inertial frame (in general an ECI frame), and its motion can be decomposed into two independent blocks:



*Figure 59: spacecraft dynamical model block diagram*

Referring to Figure 59 the first block is related to the dynamics of the rigid body, so it relates the angular velocities to the forces/torques applied to it, while the second block represents the changing of the attitude of the body when certain angular velocities are present.

The kinematic equations of an approximated model of a spacecraft can be easily computed with respect to different attitude representations (DMC, euler angles, quaternions...) and for this thesis the quaternion representation is chosen since it ensures the avoidance of singularities. So the goal here is to describe the time evolution of the attitude quaternion  $\mathbf{q}$  in function of the angular velocities around each axis of the body frame. Both the quaternion and the angular velocities are depending on time, so si possible to represent the quaternion at a generic time

instant  $q(t + \Delta t)$ , with respect to the quaternion  $q(t)$ , by using the quaternion properties described in 4.1:

$$q(t + \Delta t) = q(t) \otimes \Delta q(t)$$

Where  $\Delta q(t)$  represents the variation of the quaternion along the time interval  $\Delta t$ . Under the assumption of very small  $\Delta t$  the rotation angle performed is  $\omega \Delta t$  and considering  $\mathbf{u}$  as the rotation axis it follows that  $\omega = \omega \mathbf{u}$ . And so for small  $\Delta t$ , the incremental quaternion  $\Delta q(t)$  can be written as:

$$\Delta q(t) = \begin{bmatrix} 1 \\ \frac{\omega \Delta t}{2} \end{bmatrix}$$

Now is possible to derive the quaternion derivative:

$$\dot{q} = \lim_{\Delta t \rightarrow 0} \frac{q(t+\Delta t) - q(t)}{\Delta t} = \lim_{\Delta t \rightarrow 0} \frac{q \otimes \Delta q - q}{\Delta t} = \lim_{\Delta t \rightarrow 0} \frac{q \otimes ((1, \frac{\omega \Delta t}{2}) - (1, 0))}{\Delta t} = \frac{1}{2} q \otimes \omega^q, \quad \omega^q = \begin{bmatrix} 0 \\ \omega \end{bmatrix}$$

Finally is possible to rewrite everything in the following form:

$$\dot{q} = \frac{1}{2} \mathbf{Q} \omega, \quad \mathbf{Q} = \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \end{bmatrix}$$

This form is widely used when representing the kinematics of nanosatellites, and this is general in the sense that it can represents the satellite attitude variation related to the nanosatellite angular velocity (this interpretation will be used for the detumbling control scenario) but also the variation of the quaternion error related to the angular velocity error (this interpretation will be used in the Earth-pointing control scenario).

The second block of the block diagram proposed in Figure 59 has been defined, let's now define the dynamic equations. The most important aspect of the dynamics of the nanosatellite is that the input  $\mathbf{M}$ , that can represent the actuators torque or even disturbances torques, can be easily related to angular velocities triggered of the nanosatellites. This allows to close the chain that connects the actuators action to the outcome in terms of quaternions, and so the attitude of the spacecraft. The dynamics derivation is based on the second law of dynamics for a rotating body which states that:

$$\dot{\mathbf{H}} = \mathbf{M}$$

Where  $\mathbf{H}$  is the angular momentum (moment of momentum) and  $\mathbf{M}$  is the generic torque applied to the body. Recalling that:

$$\dot{\mathbf{H}} = \dot{\mathbf{H}}_B + \boldsymbol{\omega} \times \mathbf{H}, \quad \mathbf{H} = \mathbf{J}\boldsymbol{\omega}, \quad \dot{\mathbf{H}}_B = \mathbf{J}\dot{\boldsymbol{\omega}}$$

We obtain the *Euler moment equation*:

$$\mathbf{J}\dot{\boldsymbol{\omega}} + \boldsymbol{\omega} \times \mathbf{J}\boldsymbol{\omega} = \mathbf{M}$$

This equation is nonlinear and in general no analytical solution is available. With this relationship the nanosatellite dynamical model can be easily implemented in Simulink, along with the kinematics block defined before.

#### 4.4. General overview of AC systems

One of the most important subsystems for a nanosatellite, but in general for any spacecraft, is the **GNC** subsystem. GNC stands for “ Guidance Navigation & Control “ and is intended for sensing the actual state of the spacecraft and, eventually, to perform control actions in order to manipulate it for accomplishing a given task. In general the GNC subsystem can be represented as the combination of two subsystems that are the ADCS (attitude determination & control) and the ODCS (orbit determination & control). As the names can suggest, the ADCS is in charge to perform the determination of the spacecraft attitude and to change it whenever is needed. In this thesis only the Attitude control part is deepened.

In order to accomplish the attitude control, the following classical control scheme is taken as reference:

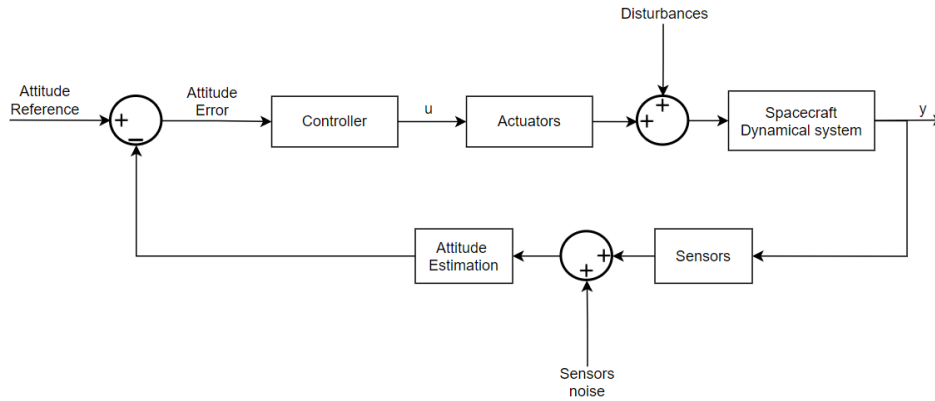


Figure 60: Attitude Control block scheme

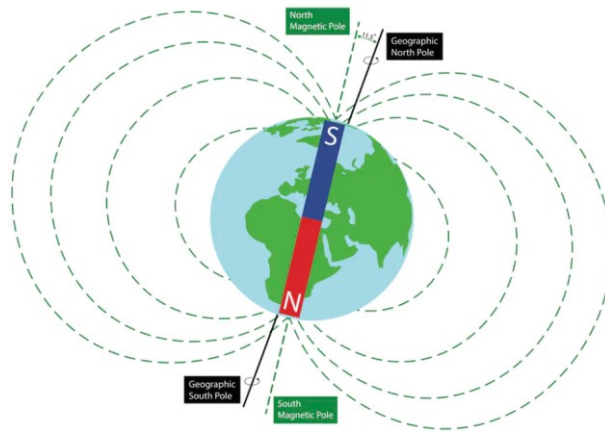
But for the thesis purposes the attitude estimation part is not represented and the spacecraft state are supposed to be all measured at each time instant. The ACS (attitude control system) objectives can be the spacecraft attitude stabilization about a reference attitude or the reference tracking in attitude manoeuvres; we can classify the attitude control system as:

- Passive: based on the body dynamics and/or environmental forces.
- Semi-active (semi-passive): based on reaction wheels and/or interactions with the Earth Magnetic Field.
- Active: based on thrusters.

The selection of the right set of actuators for a nanosatellites is crucial, since it defines which manoeuvres, and their accuracy, can be performed once in orbit. For this thesis the only kind of actuators mounted on the simulated nanosatellites are the magnetorquers.

#### 4.4.1. Magnetorquers

Magnetic control actuators are actuators capable of exerting a torque on the nanosatellite by interacting with the Earth Magnetic Field. As is well known the magnetic field of the earth can be represented as a magnetic dipole whose dipoles are located at the magnetic poles of the earth.



*Figure 61: Earth magnetic field dipole representation*

The interaction between the nanosatellite and the earth magnetic field follows a really simple physical phenomenon that also makes the compass needle to point the magnetic North. Indeed when there is a magnetic dipole immersed in a magnetic field, a torque is produced by the interaction between the magnetic moment of the immersed dipole  $\mathbf{m}$  and the magnetic flux density  $\mathbf{B}$ ; the mathematical relation is the following:

$$\boldsymbol{\tau} = \mathbf{m} \times \mathbf{B}$$

By exploiting this relation, magnetic actuators find their place in actuation systems because they are lightweight, very cheap and require low power for actuating a control action. Compared to other actuators, like thrusters, that can be used by consuming a power source that is fixed and that cannot be produced once in orbit (like propellant), magnetorquers are more reliable from this point of view since they need only electrical energy that can be stored

and reproduced by solar panels. A further advantage that increases their reliability over other actuators, like reaction wheels, is the absence of moving parts.

The main drawbacks of these actuators are:

- strong dependence on the Earth magnetic field (or in general a persistent magnetic field) so they are not suitable for deep space missions but ideal for LEO missions.
- the actuation system composed only by magnetorquers is underactuated since the vectorial product between  $\mathbf{m}$  and  $\mathbf{B}$  produces torques that can act only on a plane perpendicular to  $\mathbf{B}$ .

The magnetorquer construction design is really simple and it consists on a coil with a defined area and number of turns depending on the required performances. There are three types of magnetorquers, different from each other but based on the same concept:

- Air-core magnetorquer: this is the very basic concept of magnetorquer, a conductive wire wrapped around a non-conductive support anchored to the satellite. This kind of magnetorquer can provide a consistent magnetic dipole with an acceptable mass and encumbrance.
- Embedded coil: constructed creating a spiral trace inside the PCBs of solar panels which generates the effect of the coil. In this way is possible to minimize the impact on the satellite as it is entirely contained within the solar panels. By the way this implementation it is not able to produce an high value of the magnetic dipole, and so produced torques will be smaller.
- Torquerod: this is the most efficient solution in terms of produced dipole moments. Is made by conductive wire wrapped around a ferromagnetic core which is magnetized when excited by the coil. The drawback is the presence of a residual magnetic dipole that remains even when the coil is turned off because of the hysteresis in the magnetization curve of the core. It is therefore necessary to demagnetize the core with a proper demagnetizing procedure.

Independently to their construction, magnetorquers can produce a magnetic dipole:

$$\mathbf{m} = N \cdot I \cdot \mathbf{A}$$

Where  $N$  is the number of windings of the coil,  $I$  is the current flowing on the coil and  $\mathbf{A}$  is the area vector of the coil.

#### 4.4. Attitude control scenarios

As stated in previous chapters the simulated control system only comprises a set of magnetorquers as actuators, so the right selection of the control scenarios to simulate must be performed, taking into account the under actuation of the control system.

Considering the type of system available and which control actions can be implemented, the choice was made on two important applications for magnetic actuators: satellite detumbling and earth-pointing.

In this chapter only the theoretical treatment of the control problems is detailed, while on chapter 5 we will see the MATLAB/Simulink implementation and finally in chapter 6 the simulation results and the code generation of the control system.

#### 4.4.1. B-dot control for detumbling phase

Generally when a nanosatellite is deployed from the launcher, it is pushed out by a deployer and this procedure causes unwanted rotations of the nanosatellite that would result in an unstable system; in these situations the nanosatellite is said to be “tumbling”.

So the first task that the attitude control system must perform is to detumble the spacecraft, in other words it must mitigate these rotations until reaching a condition where the satellite has a little (ideally zero) angular momentum. Finally when the nanosatellite is detumbled, the ACS can start its nominal work.

The most reliable, and used, way to detumble a satellite in those orbits where the magnetic flux density of earth magnetic field is not neglectable, is by using magnetic actuators.

The main idea behind the concept of detumbling a satellite by means of magnetorquers is that the magnetometers mounted on the satellite, in tumbling phase, will measure at each time instant a different magnetic flux density  $\mathbf{B}$  and depending on the angular velocities is possible to obtain a derivative of the magnetic flux density  $\dot{\mathbf{B}}$ . The concept of the B-dot control algorithm is to actuate a torque of opposite sign with respect to the magnetic flux density variation, in order to dampen the rotations. There are several possible implementations of a B-dot control involving proportional terms or current control, but since in Tyvak International the detumble of nanosatellites is performed by using a B-dot bang controller, only this variant will be detailed.

The B-dot bang controller is characterized by the fact that the magnetic dipole produced will not be proportional to variation of the magnetic flux density  $\dot{\mathbf{B}}$ , indeed the control system will always produce the maximum absolute value of the magnetic dipole. One advantage of this controller is the faster spin decay compared to other approaches since the highest control action is always used for counteracting the angular rotations.

As previously stated the torque produced by the magnetorquers is given by the vectorial product between the magnetic dipole and the magnetic flux density, this relation obviously holds also for a B-dot bang bang controller but let's check how the B-dot bang bang magnetic dipole can be defined mathematically:

$$m_{i \text{ ctrl}} = -m_{i \text{ max}} \cdot \text{sign}(\dot{B}_i) = \begin{cases} m_{i \text{ max}} & \text{for } \dot{B}_i < 0 \\ -m_{i \text{ max}} & \text{for } \dot{B}_i > 0 \end{cases}, \text{ for } i = 1,2,3$$

As we can see each component of the magnetic dipole will be set at the maximum value with a sign discordant with the variation of each component of the magnetic flux density. Recalling that the magnetic dipole is related to the number of windings  $N$ , the current flowing on the coil  $I$  and the area vector of the coil  $A$ ; is possible to write the previous relation in function of the current, that will be our control signal on the actuators:

$$m_{i\_ctrl} = (N \cdot I_{i\_ctrl} \cdot A_i) = -(N \cdot I_{i\_max} \cdot A_i) \cdot \text{sign}(\dot{B}_i)$$

$$\downarrow$$

$$I_{i\_ctrl} = -I_{i\_max} \cdot \text{sign}(\dot{B}_i) = \begin{cases} I_{i\_max} & \text{for } \dot{B}_i < 0 \\ -I_{i\_max} & \text{for } \dot{B}_i > 0 \end{cases}$$

Since the Bang-Bang controller already involves only the usage of maximum current there is no need of saturators in the Simulink model.

#### 4.4.2. Earth-pointing control

When a satellite is designed, the type of mission that it will have to carry out is always kept in mind: for example the payload that is mounted on it that can be a camera or a particular sensor that must be pointed towards the earth for performing the right task. To this aim the control system design is very important for orienting the attitude of the satellite in the proper manner. After the deploying in orbit, as we have already discussed in previous paragraph, the satellite needs to be detumbled in order to achieve a “stable” state from which the ADCS can start to perform the required tasks. For some applications the “Earth-pointing” control (or Nadir pointing) is the nominal operational situation for the spacecraft, for example if it is needed to take photos of the Earth all along the orbit. For visualizing the kind of the desired attitude of the spacecraft for the Earth-pointing control we can refer to the following figure:

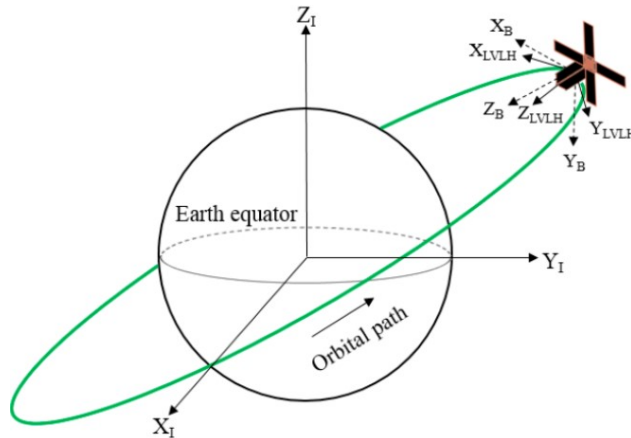


Figure 62: ECI, Body and LVLH reference frames

As is possible to notice, in the simplest Earth-pointing control scenario only three different reference frames are involved: the standard ECI frame which components are denoted with the **I** subscript (used as “fundamental” reference frame), the body frame denoted with the **B** subscript (representing the attitude of the spacecraft) and finally the LVLH frame denoted with the **LVLH** subscript. The LVLH frame (already discussed in 4.2) can be detailed in this way for the Earth-pointing control scenario:

- Origin: coincident with the origin of the body frame, in the centre of the spacecraft.
- Z axis: along the direction of the vector connecting the origin of the ECI frame with the origin of the Body frame, with verse pointing towards the Earth.
- Y axis: perpendicular to the orbital path, with opposite verse with respect to the orbit normal.
- X axis: perpendicular to the Z/Y axes in order to form a right-handed triad.

The Earth-pointing control action is required to stabilize the nanosatellite attitude, represented with the body frame orientation with respect to the ECI frame, around the LVLH frame for having the **Z<sub>B</sub>** axis pointing towards the Earth. This control problem can be easily performed with a fully actuated control system, for example by using reaction wheels, without encountering strange problems since in that case the actuators can exert the needed torque, computed by the controller, to the satellite.

In our case this can't be done, because the system is underactuated and, moreover, the control torques that can be applied are strictly related to the environment, in particular to the Earth magnetic field acting on the nanosatellite. Due to these two problems the challenge of controlling a spacecraft with only magnetic actuators for accomplishing the Earth-pointing task has been addressed by the scientific community and many papers have been published proposing effective and mathematically supported control algorithms. For this thesis the theoretical work presented in the [6] is used as reference for setting the control problem.

The fundamental idea of this control law, and its capability to achieve the task, is based on the particularity of the Earth magnetic field, along a LEO orbit, to be periodic and so even if the control system is underactuated, it can be able to decrease the error between the desired and the actual attitude of the spacecraft to zero (ideally). The first assumption that must be taken into account is that the LEO orbit of the satellite can be approximated to a circular one, and this is our case since the eccentricity of our orbit is very small. This assumption is very important for the problem setting since it allows to easily compute the angular velocity error between the body frame angular velocity and the LVLH one. Let's now present the control law and its terms:

$$m_{desired} = (K_p q_v + K_d \omega_{bo}^b)$$

Where:

- $q_v$ : vectorial part of the quaternion representing the attitude error between the body and the LVLH frame.

- $\omega_{bo}^b$ : angular velocity error between the body and the LVLH frame.
- $K_p$ : 3x3 matrix containing the coefficients proportional to the  $q_v$  error.
- $K_d$ : 3x3 matrix containing the coefficients proportional to the  $\omega_{bo}^b$  error.

The control law is very simple since it is a PD (proportional-derivative) control action, where the proportional action  $K_p$  is related to the quaternion error  $q_v$ , while the derivative term  $K_d$  to the angular velocity error  $\omega_{bo}^b$ , that defines the changing rate of  $q_v$ . This particularity of having matrices instead of coefficients allows to improve the performances of the control action since there are more degree of freedom for designing the control algorithm and is also possible to mitigate the coupling dynamics involving different axis, but for simplicity in this thesis the  $K_p$  and  $K_d$  matrices are considered diagonal and each element of them is equal to the other. The relationship above for computing the  $m_{desired}$  is not directly used in control algorithms since we know that the control system is intrinsically underactuated and in particular that the control torque available in the end will be perpendicular to the magnetic field. Is possible to exploit this information in order to avoid useless power consumption for producing the dipole: since we know that only perpendicular torques can be produced, with respect to the magnetic field, we can compute the actual  $m_{ctrl}$  by projecting  $m_{desired}$  on the plane perpendicular to the magnetic field  $B$ . This is done because in general only the part of  $m_{desired}$  that is perpendicular to the magnetic field, is responsible for the production of a torque on the spacecraft and so the magnetic dipole  $m_{ctrl}$  and the correspondant produced torque can be computed as follows:

$$m_{ctrl} = (K_p q_v + K_d \omega_{bo}^b) \times B$$

$$\tau = m_{ctrl} \times B$$

And this will be the final relationships that will be implemented in Simulink for representing the controller.

## 5. MATLAB/SIMULINK MODELS

In order to simulate and test the control algorithms, a simulation environment is needed for representing the satellite in its orbit and its interactions with the Earth magnetic field. To this aim an orbit propagator is designed for collecting the useful data needed for reconstruct the magnetic flux density along the orbit and the position of the satellite.

Since the *Skyfield* python library is widely used for this purposes, even in Tyvak International, a python script implementing its functions is used for the propagation of the orbit. First of all we have to select a desired orbit: for this thesis a LEO is preferred and in particular the one of the Tyvak International nanosatellite “Tyvak-0092” (Commtrail), that has been in orbit for over a year now. The informations and characteristics about a certain orbit can be collected and represented by means of the so called **TLE**: the two-line element set is a data format encoding a list of orbital elements of an Earth-orbiting object for a given point in time; TLEs are used for describing trajectories only of Earth-orbiting objects like satellites and debris. The TLE representing Commtrail orbit is the following:

```
TYVAK-0092
1 44852U 19089A 21052.22088221 .00000573 00000-0 51478-4 0 9999
2 44852 36.9677 65.5974 0007145 313.1735 46.8406 15.00042925 65725
```

Figure 63: Tyvak-0092/Commtrail nanosatellite TLE

A TLE contains all the informations needed for identifying an object orbiting around the Earth in only two coded lines, the informations are organised in this way:

- Line 1: contains informations about the Satellite catalog number (used for univocally identify an artificial object), informations about the launch that brought the satellite in its orbit and on the mean motion about that orbit.
- Line 2: contains the classical orbital elements for identifying the orbit like the eccentricity, inclination, argument of Perigee etc...

By using Skyfield functions for computing the position along the desired orbit, is possible to obtain and store the desired geocentric coordinates (latitude, longitude and elevation) based on the *WGS84* (World Geodetic System 84) used for representing an ellipsoid approximating the Earth geometry. In particular the WGS84 is defined as follows:

- Center : in the Earth mass center.
- Z axis: passing in the North pole.
- X axis: chosen in order to have the Greenwich meridian laying on the XZ plane.
- Y axis: for completing the right-handed triad.

As is possible to imagine, this triad can be interpreted as an ECEF reference frame.

Finally by running the script, a "*log\_orbit.txt*" file is produced containing all the informations required, as is possible to see in the following picture:

```
year month day hours minutes seconds time[s] lat[°] lon[°] elev[m]
2021 2 9 0 0 0 19.471655784372214 150.7692389703974 564446
2021 2 9 0 0 1 19.438185989226824 150.8212413633378 564439
2021 2 9 0 0 2 19.40469871285869 150.87322082905044 564432
2021 2 9 0 0 3 19.371193994589518 150.92517740765658 564425
2021 2 9 0 0 4 19.337671873704895 150.97711113931493 564418
2021 2 9 0 0 5 19.304132389458115 151.0290220642223 564411
2021 2 9 0 0 6 19.270575581068748 151.0809102226062 564405
2021 2 9 0 0 7 19.237001487723354 151.13277565472396 564398
```

Figure 64: Orbit propagation *log\_orbit.txt* file snippet

Is important to notice that the file contains all the data stored as columns, where we can find the UTC date along with the latitude, longitude and elevation. This kind of structure is crucial for using the *log\_orbit.txt* file as a data file to import in MATLAB, in this way we can easily obtain all the required data in MATLAB and to store them in suitable variables. Along with the previously detailed file, another one is needed for computing the quaternion representing the attitude of the LVLH frame with respect to the ECI. The same script also produces another log file that is the following:

```
time[s] x1 x2 x3 y1 y2 y3 z1 z2 z3
0 0.6425168625952835 0.733593183876978 -0.22138852815532983 -0.4111208185225473 0.08620999509130706 -0.9074950511559493 -0.6466464361380889 0.674098468758051 0.3569868919833884
1 0.6418111101360169 0.7343285480146995 -0.22099737891673085 -0.41111990617723626 0.08621103777102104 -0.9074953656756228 -0.6473474968288846 0.6732971920301253 0.3572283717088072
2 0.6411045917679788 0.7350630378725106 -0.2206059660156829 -0.41111899498523347 0.08621208132155642 -0.9074956795878426 -0.6480477874841546 0.6724951119819992 0.3574694245627315
3 0.6403973083292488 0.7357966525723675 -0.22021428991907935 -0.4111180849484275 0.08621312574003892 -0.907495992892025 -0.6487473072667183 0.6716922295661536 0.3577100502566736
4 0.6396892606597325 0.7365293912363668 -0.21982235109447967 -0.4111171760686037 0.08621417102367551 -0.9074963055876263 -0.6494460553394712 0.6708885457370014 0.35795024850234664
5 0.6389804495995601 0.7372612529883351 -0.219430150009467 -0.41111626834759923 0.08621521716961822 -0.9074966176740844 -0.6501440308668709 0.6700840614491678 0.3581900190122073
6 0.6382708759899116 0.737992236952992 -0.21903768713202626 -0.4111153617872549 0.08621626417500936 -0.9074969291508359 -0.6508412330141481 0.6692787776583952 0.3584293614991771
7 0.6375605406730597 0.7387223422559238 -0.21864496293053243 -0.41111445638939 0.0862173120369985 -0.9074972400173278 -0.6515376609472863 0.6684726953215777 0.35866827567662607
8 0.6368494444916185 0.7394515680243479 -0.21825197787333792 -0.4111135521558046 0.08621836075274494 -0.9074975502730132 -0.6522333138337509 0.6676658153959109 0.3589067612586271
```

Figure 65: Orbit propagation *LVLH\_orbit.txt* file snippet

This file includes the components, expressed with respect to ECI frame, of the LVLH frame at each time step.

In the following paragraphs we will see how these data are used and how to model in Simulink all the subsystems needed for the simulation and the design of the control system.

## 5.1. Orbit and Earth magnetic field propagator

Once all the data have been imported in MATLAB and saved in suitable *timeseries* variables, we can use the following (Figure 67) Simulink model for computing the Earth magnetic field flux density "*B<sub>eci</sub>*", expressed in ECI frame, and the quaternion "*q<sub>eci2lvlh</sub>*" describing the orientation of the LVLH frame (local orbital frame) with respect to the ECI frame. As is possible to see in Figure 67, the Simulink model is made up by two branches: the upper branch devoted to the computation of the Earth magnetic flux density and the lower one designed for computing

the orientation of the local orbital frame with respect to ECI frame. Let's deepen each branch and see how the computations are performed:

- **Magnetic flux density computation:** the geocentric coordinates describing the orbit are taken as input by the *International Geomagnetic Reference Field model IGRF-13* (Simulink Aerospace Blockset), that computes in output the magnetic flux density vector expressed in NED coordinates. Since we want it expressed in ECI frame, two coordinate transformations are performed: the NED to ECEF coordinate transformation is done by using a suitable transformation matrix  $R_{ned}^{ecef}$  and then the ECEF to ECI coordinate transformation by means of the "*ecef2eci()*" function contained in the Aerospace Blockset.
- **LVLH quaternion with respect to ECI:** The computation of the LVLH quaternion is really simple once the components of the LVLH axis, with respect to the ECI frame, are stored. This is because in general the  $R_{lvlh}^{eci}$  rotation matrix, that represents the attitude of the LVLH frame with respect to the ECI frame, has for columns the components of each axis computed with respect to the ECI frame. So as is possible to see from the Simulink model we just need to concatenate these vectors for obtaining  $R_{lvlh}^{eci}$ . The obtained matrix can be easily converted in quaternions with the "*dcm2qua()*" function, obtaining the desired quaternion that will be used as reference in Earth-pointing control. Notice that this quaternion will be used only for the computation of the initial condition of the quaternion error in Earth-pointing scenario.

After the simulation the propagated magnetic flux density  $\mathbf{B}$  expressed in ECI coordinates, that the nanosatellite will experience in its orbit has the following behaviour:

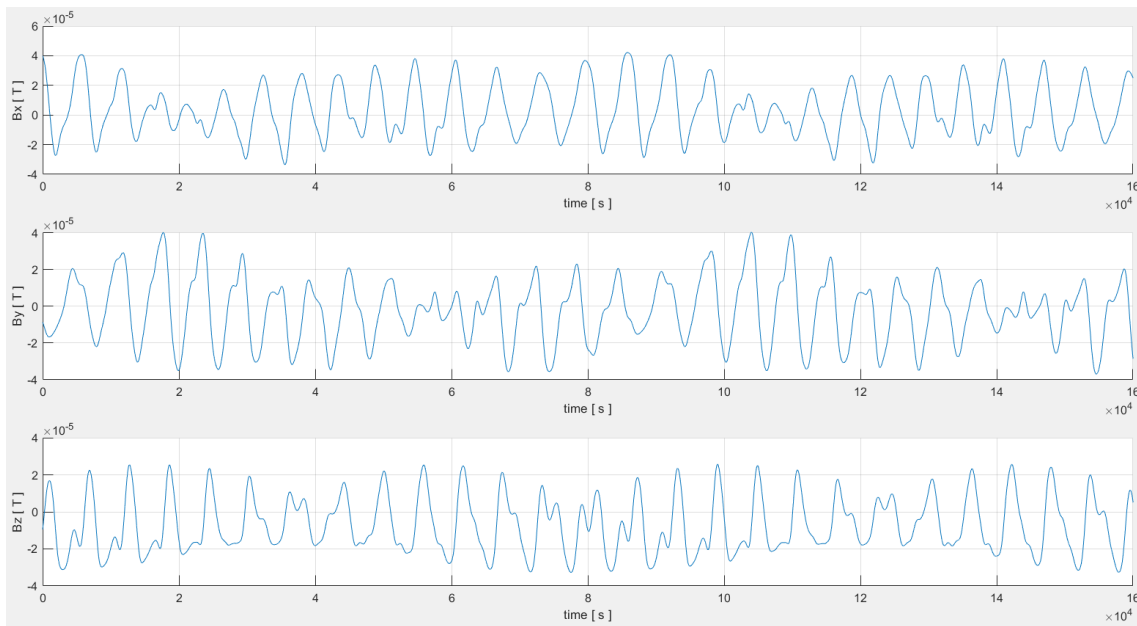


Figure 66: Magnetic flux density components:  $B_x$ ,  $B_y$  and  $B_z$  (top to bottom)

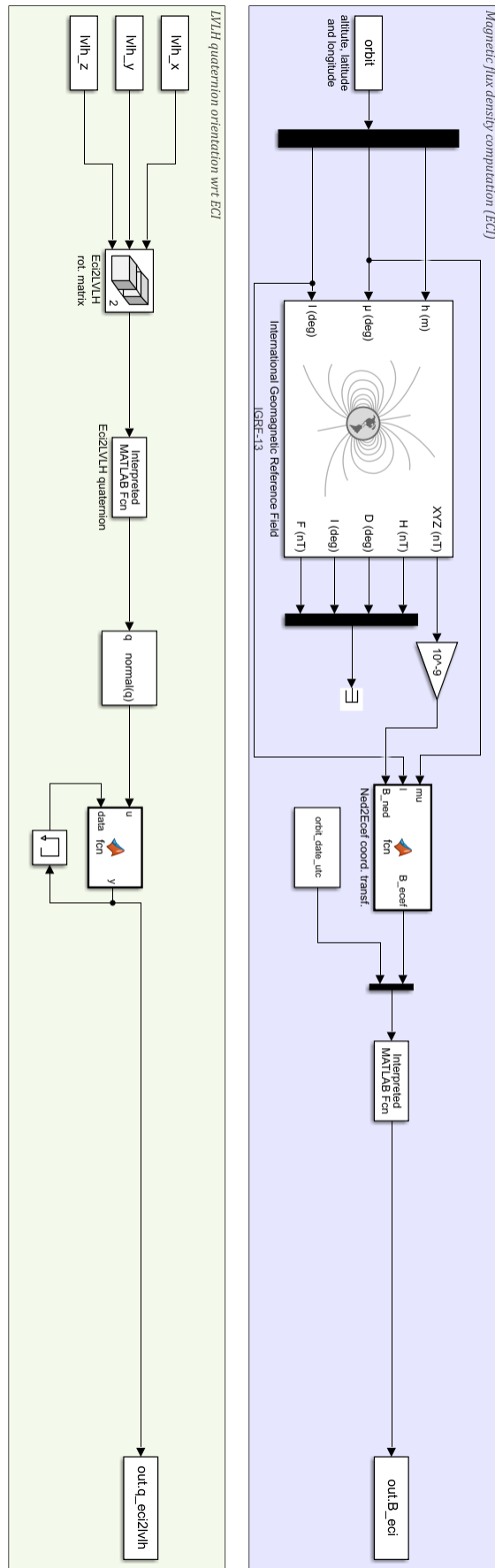


Figure 67: Orbit propagator Simulink model

## 5.2. Satellite model

In this section the “Satellite dynamical model” implementation in Simulink is presented and discussed. As anticipated in 4.3 a spacecraft can be represented by means of two input-output blocks, one describing the dynamics of the rigid body and the other one the kinematics.

For the detumbling scenario the setting is very simple, indeed the dynamics and kinematics described before have been just implemented as they are. In this scenario also environmental disturbances have been included. This is the overall block diagram representing our satellite and environmental interactions:

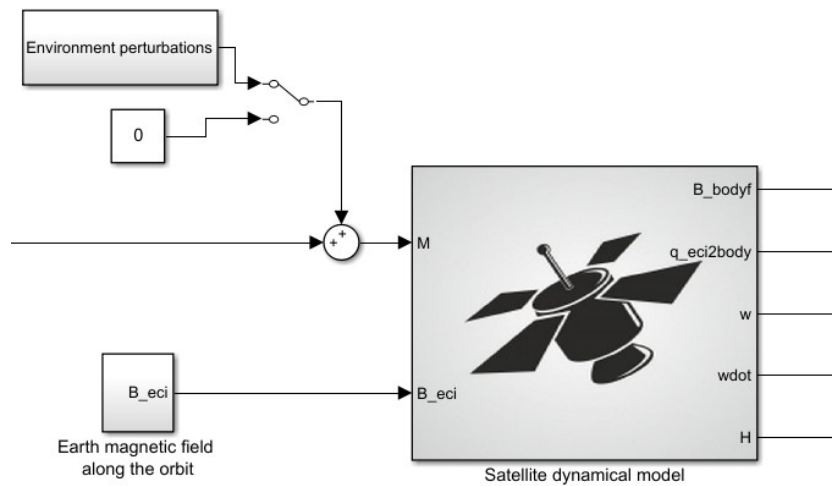


Figure 68: Detumbling scenario (Satellite model and environmental interactions)

Let's describe each block in details:

- **Environmental perturbations:** when a spacecraft is deployed in orbit and in general is moving in the space, there are many sources of torques and forces that can affect its dynamical behaviour, these forces/torques can be seen as additive disturbances acting on the actuator control action. For the thesis purposes these disturbances have not been deeply addressed, for example by modelling gravity gradient or including a residual dipole moment. They are just uniformly distributed random signals with a maximum magnitude of:  $|5 \cdot 10^{-7}| Nm$ .
- **Earth magnetic field  $B_{eci}$ :** this is simply the Earth magnetic field computed in 5.1 and fed to the satellite model. Notice that at each time step the  $B_{eci}$  vector is expressed with respect to the ECI frame, while the control algorithm needs all the signals expressed in body frame. To this aim a conversion inside the satellite dynamical model is performed.
- **Satellite dynamical model:** as said before this block implements the dynamics and kinematics of the satellite, along with the  $B_{eci} \rightarrow B_{bodyf}$ . As can be seen in the fig.XX.

below, we have the “Euler equation” and the “quat kinematics” blocks that implements the relationships defined in 4.3. Then there is a coordinate transformation from ECI to Body coordinates of the  $B_{eci}$ , by using the DCM retrieved from the quaternion for each time instant. This  $B_{bodyf}$  signal will be used by the controller and for computing the torque acting on the satellite due to magnetic interaction with the Earth magnetic field.

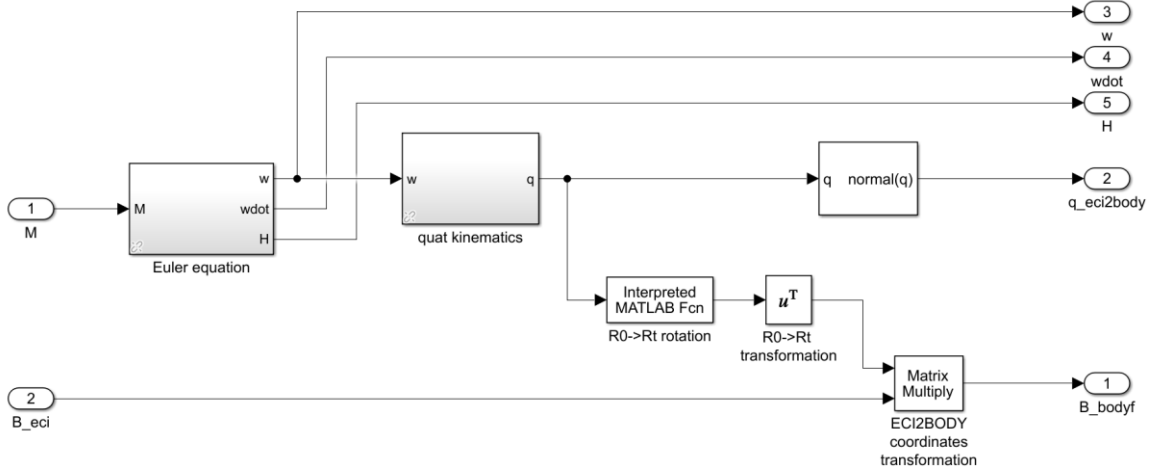


Figure 69: Detumbling control: “Satellite dynamical model” insight

Is possible to see that in this scenario, all the state variables of the satellite and the other signals used for the control purpose are defined with respect to the ECI frame.

For the Earth-pointing scenario the fundamental concept is quite the same, but in this case the “Satellite dynamical model” will implement the dynamics and kinematics concerning the quaternion error and the angular velocity error; the disturbances from the environment in this case are not considered. The decision of representing the situation, and in particular the satellite dynamical model, not strictly referring to our satellite but to its relative attitude and angular velocity with respect to the desired ones, makes the problem easier to be studied and solved. Indeed all the papers and researches studying the magnetic actuation of nanosatellites for Earth-pointing are based on this setting. As said before here we don’t have environmental disturbances so the only block present in Simulink related to the satellite is the one implementing dynamics and kinematics of the error.

In Figure 70 the insight of the “Satellite dynamical model” is depicted and as expected the general block diagram is very similar to the one in the detumbling scenario, but here we are focusing on the error affecting the overall system involving the body frame and LVLH frame. To this aim the outputs of the system are “ $w_{tilde}$ ” and “ $q_{tilde}$ ”. Notice that the “*quat\_kinematics*” block, that is exactly the same of the detumbling scenario, is used only for retrieving the Earth magnetic field acting on the satellite, in body coordinates. The block “*quat\_error\_kinematics*” implementing the kinematics is the classical one, but the quaternion variations are triggered by the angular velocity error, so its output will be the quaternion representing the rotation required for passing from body frame to LVLH frame, that is the error “ $q_{tilde}$ ”.

The diagram illustrates the ECI2BODY model architecture. It starts with two inputs:  $M$  (labeled 1) and  $B_{eci}$  (labeled 2). The input  $M$  is processed by the **error\_Euler\_equation** block, which also receives  $q\_tilde$  as feedback. The output of this block is  $w\_bo\_b$ . The input  $B_{eci}$  is processed by the **quat\_error kinematics** block, which also receives  $w\_bo\_b$  as feedback. The output of this block is  $q\_bo$ . The  $q\_bo$  signal is then processed by the **quat kinematics** block, which also receives  $w\_bo\_b$  as feedback. The output of this block is  $q$ . The  $q$  signal is then processed by the **Interpreted MATLAB Fcn** block, which performs an **R0->Rt rotation**. The output of this block is  $u^T$ . The  $u^T$  signal is then processed by the **Matrix Multiply** block, which performs an **ECI2BODY coordinates transformation**. The output of this block is  $B_{bodyT}$  (labeled 1). The  $B_{bodyT}$  signal is then processed by the **Interpreted MATLAB Fcn** block, which performs an **R0->Rt rotation**. The output of this block is  $u^T$ . The  $u^T$  signal is then processed by the **Matrix Multiply** block, which performs an **ECI2BODY coordinates transformation**. The output of this block is  $B_{bodyT}$  (labeled 1). The  $B_{bodyT}$  signal is then processed by the **Interpreted MATLAB Fcn** block, which performs an **R0->Rt rotation**. The output of this block is  $u^T$ . The  $u^T$  signal is then processed by the **Matrix Multiply** block, which performs an **ECI2BODY coordinates transformation**. The output of this block is  $B_{bodyT}$  (labeled 1).

The diagram illustrates a control system for a magnetic field measurement. It features three reference signals (sine waves) that are fed into a multi-input summing junction. The output of this junction is fed into a switch. A block labeled '0' is also fed into the switch. The switch output is fed into a second summing junction, which also receives input from a 'Magnetometer sampling' block. The output of the second summing junction is the measured signal  $B_{bodyf\_measured}$ .

63

### 5.3. Magnetorquers model

The magnetorquers Simulink model is just an implementation of the concepts and relations described in the section 4.4.1, where magnetorquers were introduced and detailed.

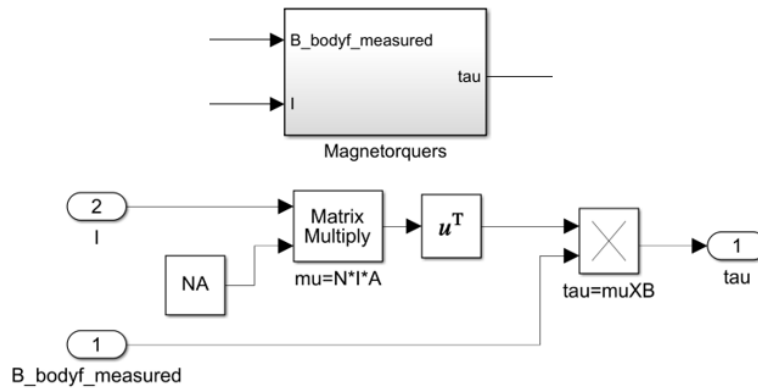


Figure 72: Magnetorquers subsystem (top) and implementation (bottom)

As is possible to see from the picture above the modelization of magnetorquers can be really simple if it is intended as a two inputs-one output system, where the current  $I$  is the control action exerted by the controller and  $B\_bodyf\_measured$  is the magnetic flux density sensed by the simulated magnetometer. The output is just the torque  $\tau$  produced by the interaction between the magnetic dipole and the magnetic flux density. A peculiarity of the B-dot bang bang controller is that we don't need a saturation of the control signal, in this case the current, because in the concept of the controller we want to exert always the max current whenever is needed.

### 5.4. B-dot Bang controller model

The B-dot bang controller model is very simple since, just as for the magnetorquers, it is just an implementation of the relations detailed in the section 4.4.1. The system is depicted in Figure 73 below:

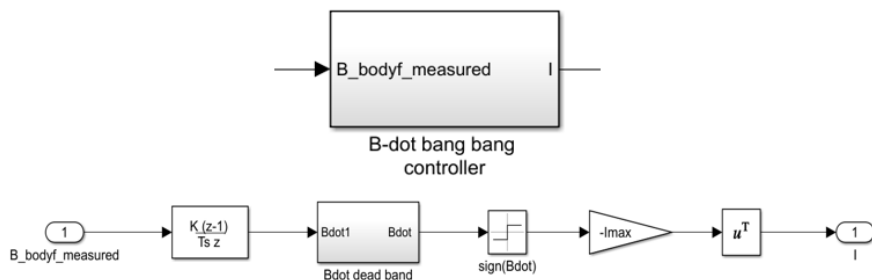


Figure 73: B-dot bang bang subsystem (top) and implementation (bottom)

The controller is a simple SISO (single input-single output) system where the data collected by the magnetometer are the only data concerning the “state” of the satellite that are needed. As is possible to see in the previous figure the  $\dot{\mathbf{B}}$  is computed by a discrete-derivative block that implements the formula:  $\dot{\mathbf{B}} = K \cdot \left( \frac{\mathbf{B}_t - \mathbf{B}_{t-1}}{T_s} \right)$ , with  $K=1$  (unitary gain). Then a dead band is needed:

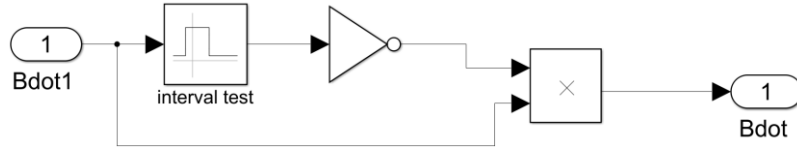


Figure 74: B-dot bang bang dead band implementation

The concept is really simple: since the sign of  $\dot{\mathbf{B}}$  can directly determine the triggering of a null current or a max current (in both verses), the controller is really sensitive to every variations of sign of  $\dot{\mathbf{B}}$ . This can be a real problem for the hardware and the power consumption since even a very small value different from zero will trigger the controller to send the max current, resulting in a continuous switching on the current verse. For fixing this problem the dead band technique is adopted, in particular an interval around zero is selected and whenever the signal is within that interval, the output signal will be set to 0.

The “interval test” block checks if the components of the  $Bdot1$  signal are within the interval, which amplitude has been selected after a trial and error approach, and in this case the output will be set to a Boolean TRUE (represented as 1) otherwise to FALSE. At this point a NOT operator is used because we want to set to “0” each component that is within the interval, with a successive element-wise product. Let’s see how the dead band works and the results in output:

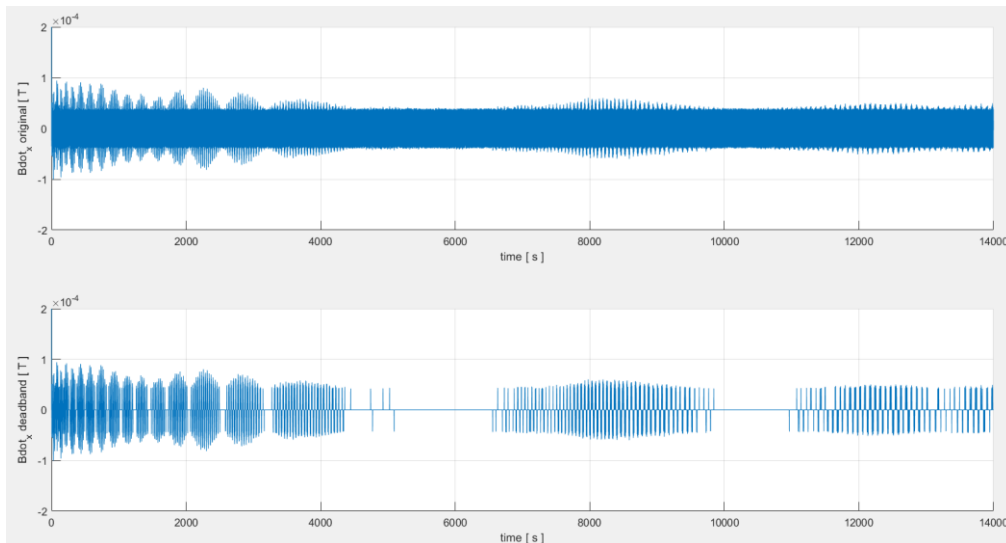


Figure 75: original B-dot\_x (top) and “filtered” B-dot\_x (bottom)

As we can see from the picture, the dead band is really useful to cut down these natural oscillations that would trigger the currents continuously, especially at steady state where the variation in the magnetic flux density cannot be exactly 0 due to the behaviour of the magnetic field in orbit. The dead- band amplitude used for this test is  $4.6 \cdot 10^{-5}$ .

## 5.5. Earth-pointing controller model

Regarding to the Earth-pointing scenario let's see how the control law, defined in 4.4.2, can be implemented in Simulink:

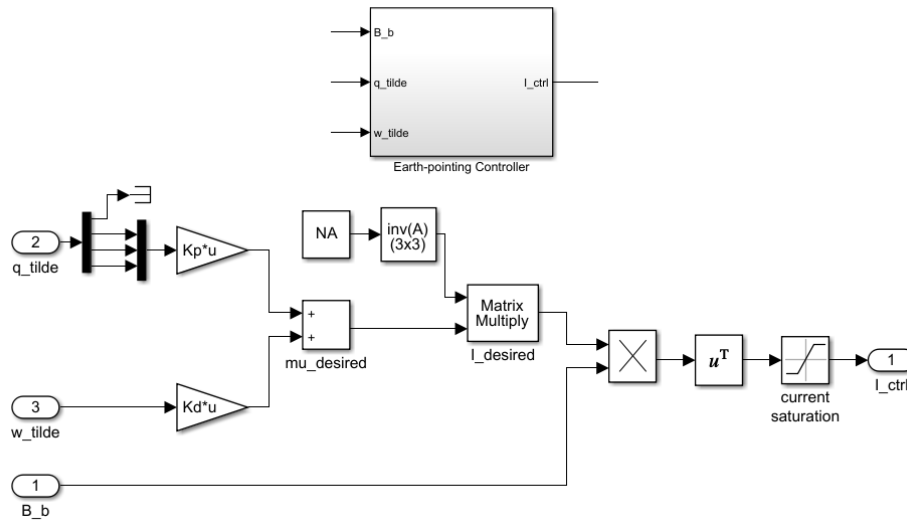


Figure 76: Earth-pointing controller subsystem (top) and implementation (bottom)

From the Figure 76 above is clear that the left part of the controller implements the control law for computing the desired dipole, starting from the errors related to the attitude and the angular velocity. Then the correspondent current is computed and saturated according to the magnetorquers capabilities.

## 6. SIMULATIONS AND CODE AUTO-GENERATION

In this chapter the simulations steps and their results will be explained, in particular how the simulation has been set. The simulation environment is made up by using MATLAB/Simulink where all the data management and the initial conditions setting have been implemented in a MATLAB script, while all the models presented in chapter 5 are used in Simulink for simulating the satellite and the control actions applied to it. The Simulink solver is a fixed step Ode5 (Dormand-Prince) with a 0.01 fundamental sample time. Finally after the simulations the two control algorithms will be translated into code by auto-generation using the ROS Toolbox.

### 6.1. B-dot detumbling simulations

The first scenario that has been simulated is the detumbling of the satellite; referring to the Simulink model in Figure 82 .

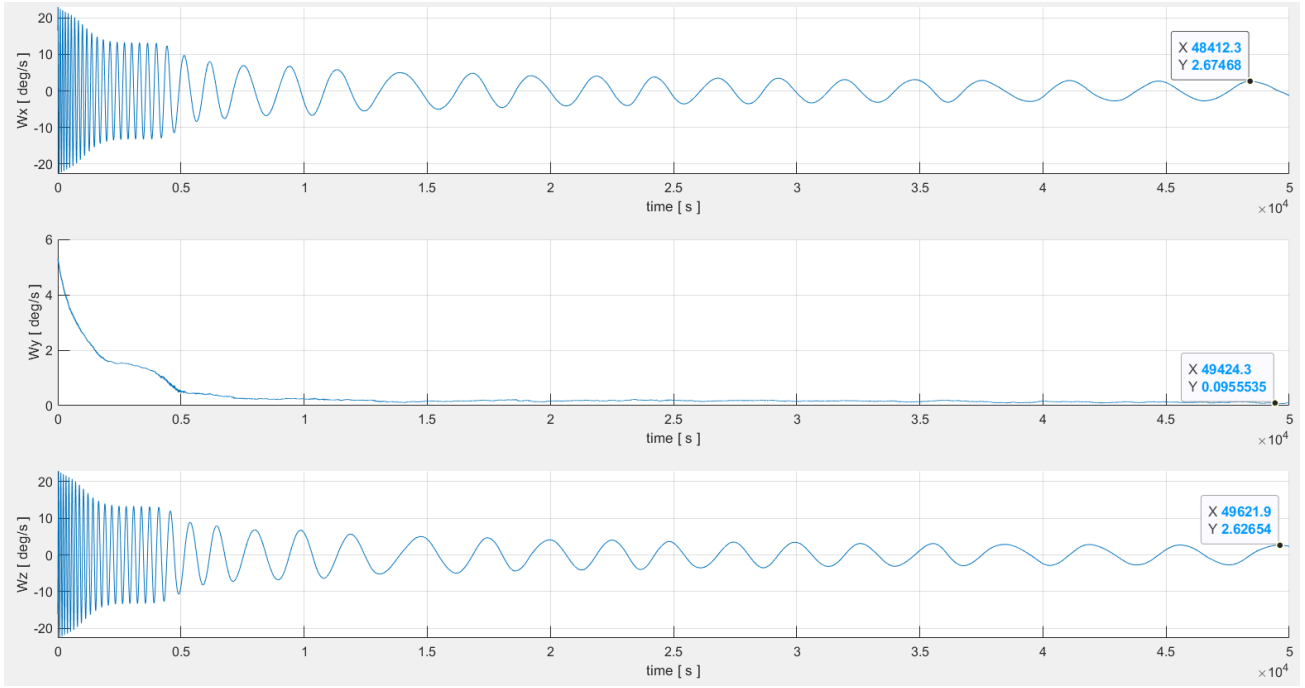
At this point once the simulation environment is set and the satellite initial conditions regarding initial attitude and angular velocities, have been defined in MATLAB, we can run the simulation for testing the B-dot bang controller. These are the conditions used for testing the dead-band action during the simulations:

<i>Simulation</i>	$\omega_0[deg/s]$	$q_0$ body	$J$ inertia matrix	<i>dead_band amplitude</i>
Test 1	[16.53; 5.29; -16.09]	[1; 0; 0; 0]	<i>diag</i> [0.0111; 0.0022; 0.0111]	$4.6 \cdot 10^{-5}$
Test 2	[16.53; 5.29; -16.09]	[1; 0; 0; 0]	<i>diag</i> [0.0111; 0.0022; 0.0111]	$4.2 \cdot 10^{-5}$

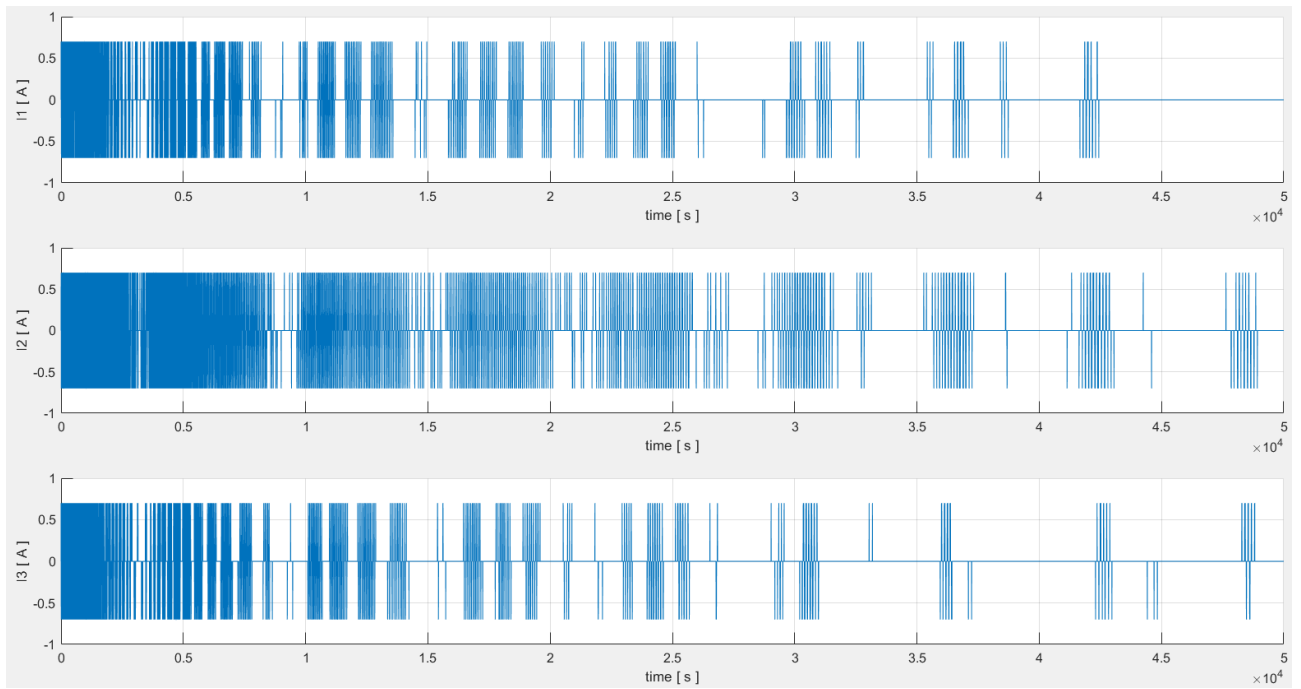
*Table 3: B-dot Dead-band tests settings*

As is possible to see from the Table 3 these tests are performed for evaluating the B-dot bang control performances when a different dead-bands are used, and how this difference reflects on the control current signal sent to the magnetorquers. The initial angular velocities have been randomly selected in MATLAB with the function “rand” in the interval  $[-20, 20]$  deg/s.

The obtained results for the test 1 are reported in Figure 77 Figure 78. below: as we can see the B-dot bang controller is able to dampen the angular velocities of the satellite, starting with relatively high values, since  $\omega_0 = [16.53; 5.29; -16.09]$  [deg/s] (randomly selected), till reaching low values that are  $\approx 2.6$  deg/s around the X and Z axes, while for the Y axis (corresponding to the lowest inertia axis) the angular velocity is almost 0. The control currents has a reasonable behaviour that becomes less pronounced when approaching to the “steady-state” condition. Obviously since we have a varying magnetic flux density and high frequency disturbances acting on the measurements and the dynamics of the system, we cannot expect a null control action when the B-dot controller is activated. Indeed in real satellites a check is performed for enabling and disabling the controller: for example when the angular velocities are below a certain threshold the detumble can be considered achieved, otherwise the controller must be activated. The obtained results are in line with those that can be verified in real systems.



*Figure 77: Detumbling test 1: angular velocities*



*Figure 78: Detumbling test 1: control currents*

From the obtained results of the test 2 (Figure 79 and Figure 80) we can see that decreasing the dead-band amplitude increases the performances of the B-dot bang controller in terms of time required for dampening the angular velocities and also for what concerns the amplitude of the velocities, that can be further dampened since the control action is more frequent.

Indeed the dampening action is faster and the “*steady-state*” angular velocities performances around X and Z axe are improved, with respect to the test 1, in particular they are  $\approx 1.5 \text{ deg/s}$ .

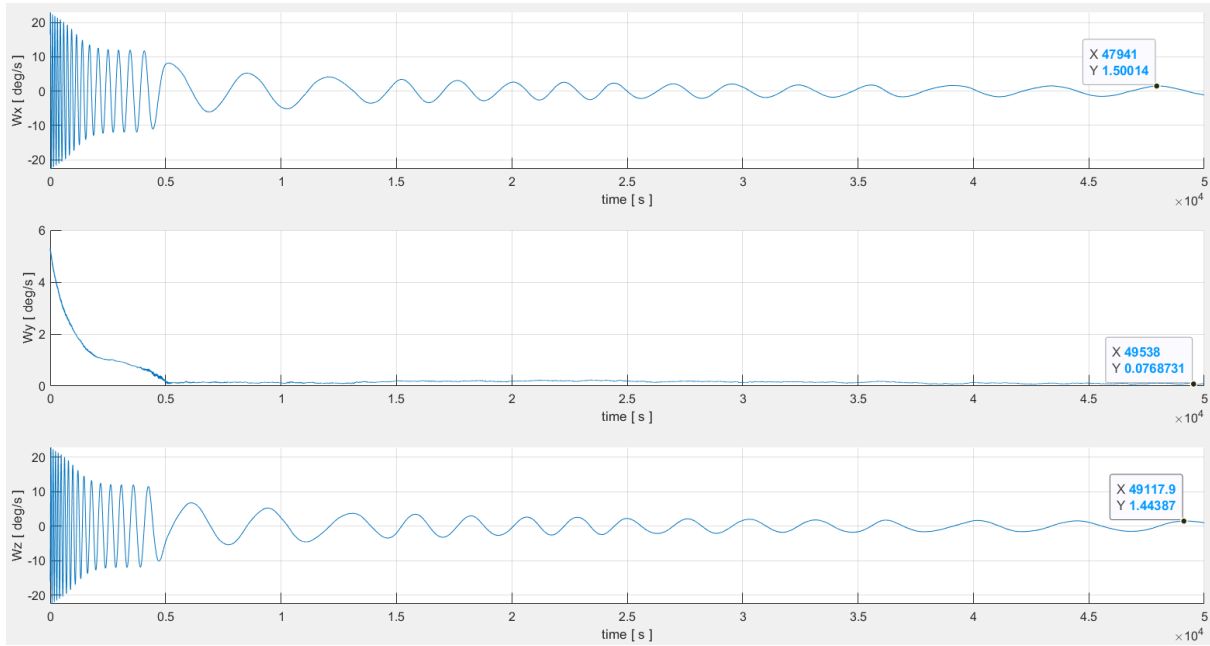


Figure 79: Detumbling test 2: angular velocities

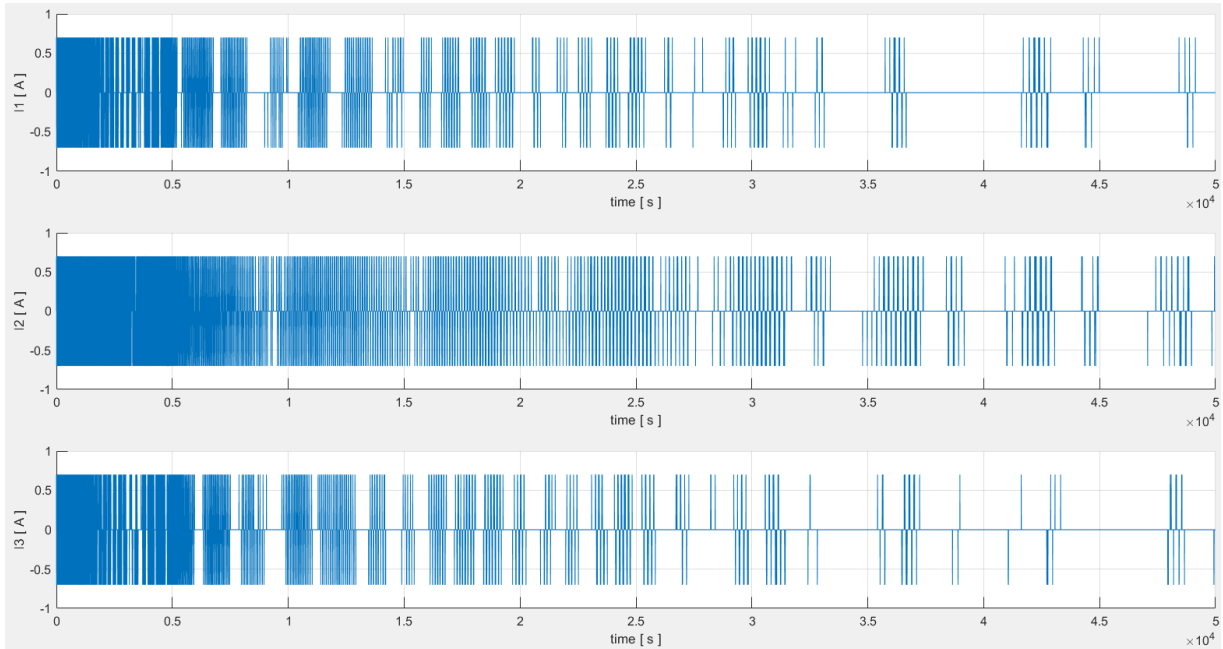


Figure 80: Detumbling test 2: control currents

On the other hand the control currents exerted are switching more frequently with respect to the test 1, this can be a real problem for the hardware. So a trade-off between performances and control action must be done in order to obtain the best compromise. Notice that small changes

in the dead-band amplitude can significantly affect the overall performances, since the variations of the magnetic flux density are of very small order of magnitude.

At this point let's perform some simulations for evaluating the robustness of the control system. To this aim different initial conditions, randomly generated, have been selected and used for testing the B-dot controller. The following simulations have been performed by including environmental disturbances and magnetometer noise (both described in 5.2) and with a dead-band amplitude of  $4.2 \cdot 10^{-5}$  and inertia matrix  $J = \text{diag}[0.0111; 0.0022; 0.0111]$

Simulation	$\omega_0[\text{deg/s}]$	$q_0 \text{ body}$
1 (blue)	[16.68; -8.56; 10.28]	[-0.1329; 0.9619; 0.1976; 0.1341]
2 (red)	[-17.84; 1.23; 11.16]	[0.5679; -0.1657; 0.1467; -0.7928]
3 (yellow)	[-19.52; -6.51; -13.51]	[0.9655; -0.2490; 0.0700; -0.0308]
4 (purple)	[4.07; -9.48; 6.16]	[0.6261; -0.6616; 0.0746; -0.4057]
5 (green)	[-10.84; 16.53; -13.90]	[0.2600; 0.6020; -0.0683; -0.7519]

Table 4: B-dot controller, 5 simulations random initial conditions (attitude and ang. vel.)

As we can see from the B-dot controller is able to detumble the satellite with good performances; the angular velocities at steady state are always below  $|1.56|$  deg/s even with disturbances affecting the dynamics of the satellite and the magnetometer measurements.

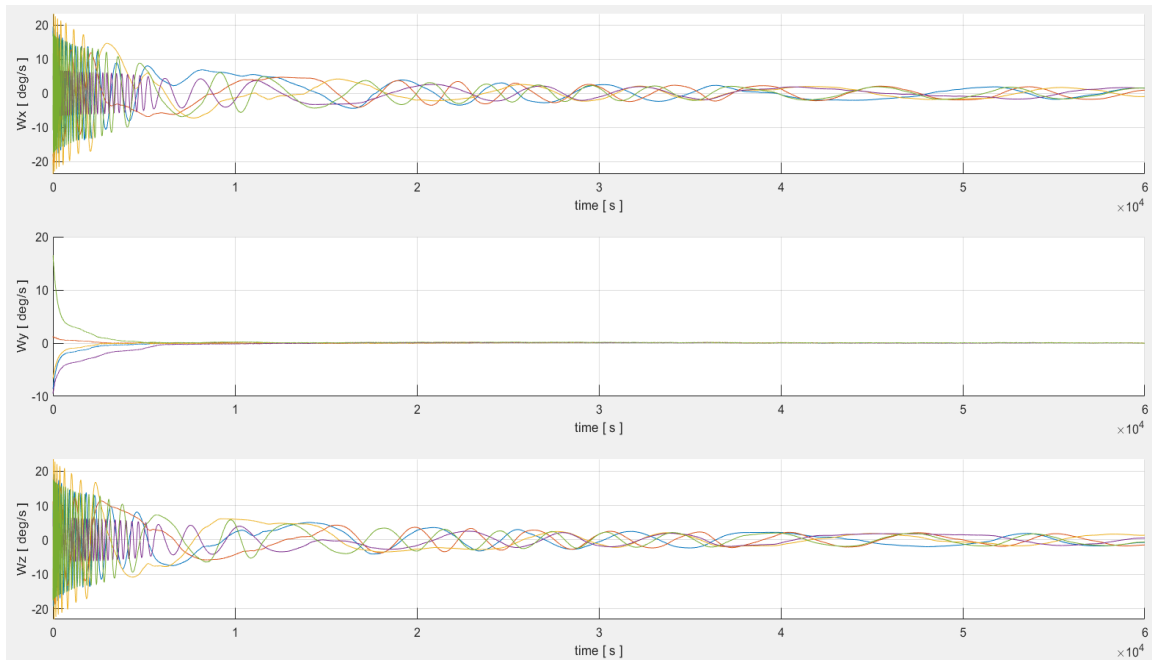


Figure 81: B-dot controller simulations performance

The performances are good and they can be even improved by tuning the dead-band amplitude in order to achieve lower residual angular velocities, even if the present results are acceptable for a real application.

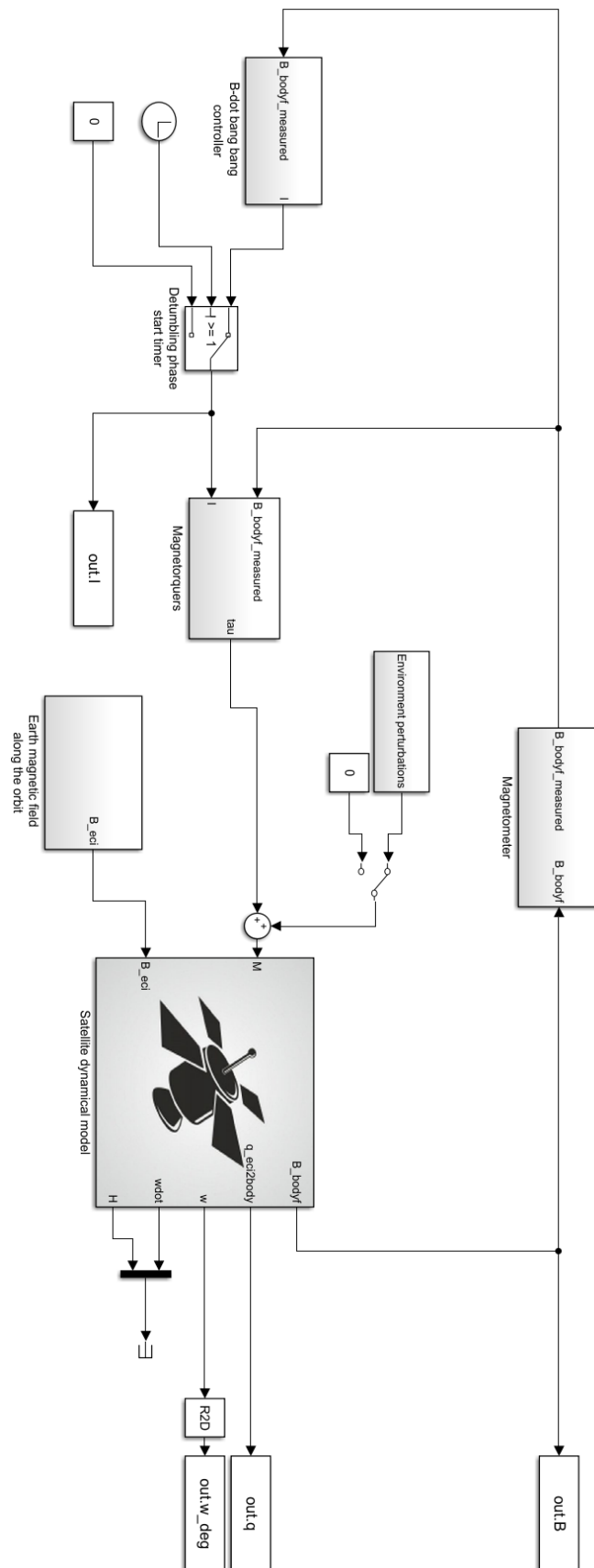


Figure 82: Simulink detumbling model

## 6.2. Earth-pointing simulations

Finally the controller discussed in 5.5 has been tested and validated in Simulink by considering different initial conditions of the satellite. This time the initial quaternion and angular velocities are defined relatively to the LVLH ones, so now the errors  $q_v$  and  $\omega_{bo}^b$  between the body frame and the LVLH frame are the initial conditions of the kinematics and dynamics of our system; moreover the environmental disturbances have been not considered.

Recalling that  $m_{ctrl} = (K_p q_v + K_d \omega_{bo}^b) \times B$ , the first consideration that can be made is relative to the  $(K_p, K_d)$  and how these parameters affects the performances of our control system. In general the parameters of a PID or PID-like controller can be computed by solving an optimization problem (LQR) in order to guarantee stability or performance requirements. In this thesis, since the situation is really similar to the one already studied and analysed in [6], a tuning procedure by simulating the system with different values for the parameters has been adopted, starting with values similar to the ones reported in the paper has optimal values.

So let's consider a situation in which our satellite is not aligned with the desired LVLH frame, in particular the attitude error is such that the body frame is rotated of  $180^\circ$  around the z axis of the LVLH frame, and that the angular velocity error is about 2 deg/s around x,y and z axis of the LVLH frame. In this situation let's test how the parameters of the controller can affect the performances of the control system. In particular, as is possible to see from the figure below, the proportional term has been lowered in order to avoid unwanted oscillations.

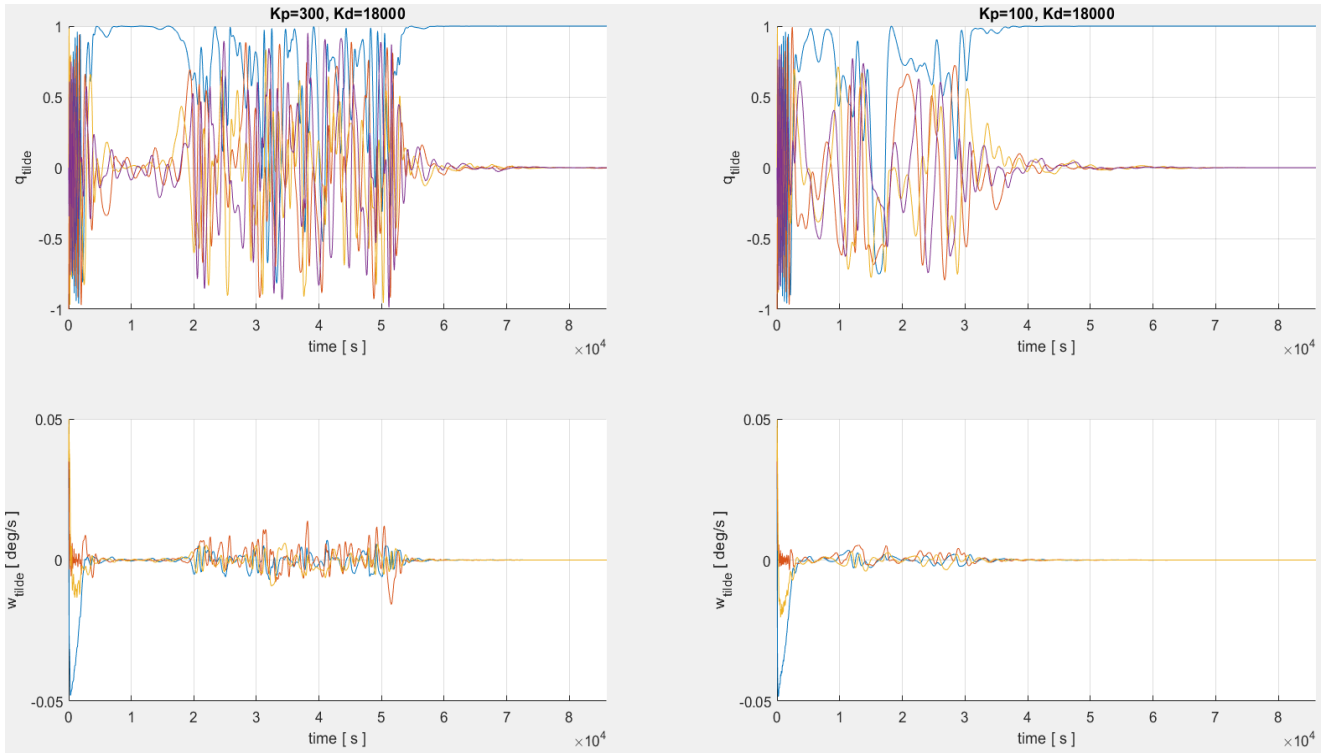
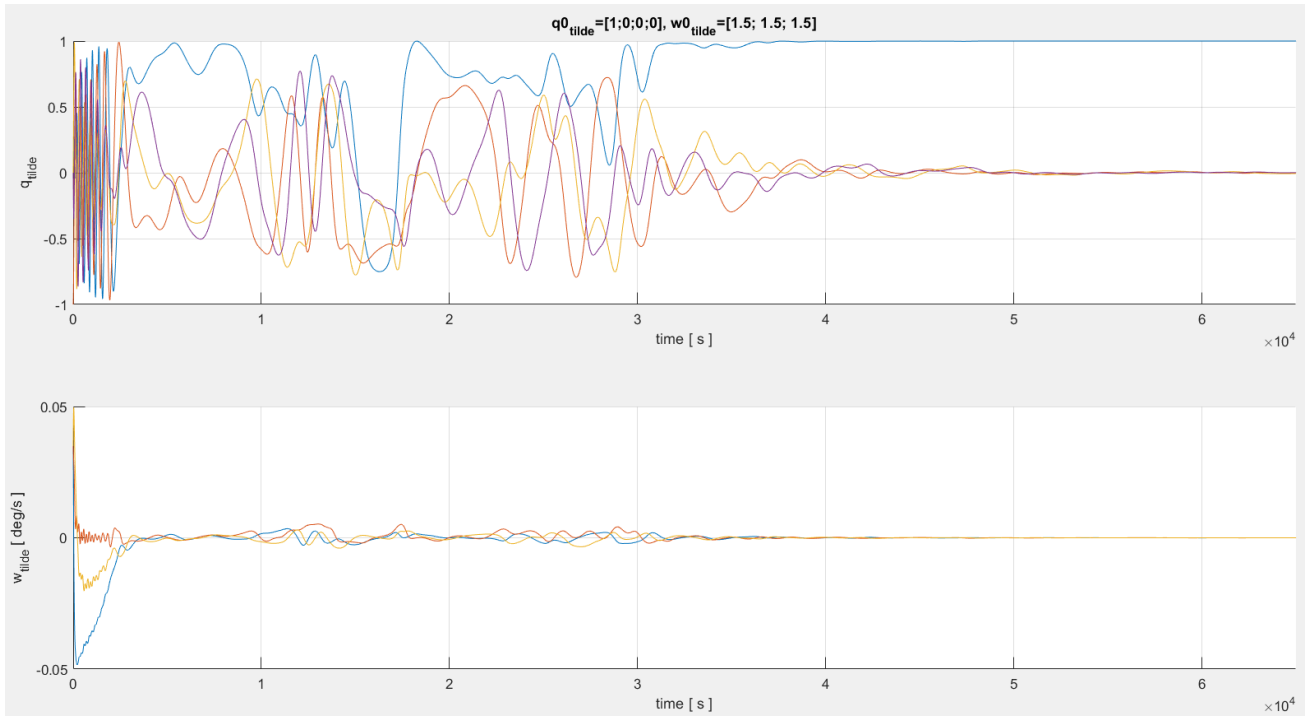


Figure 83: controller performances comparison between  $K_p=300$  and  $K_p=100$

As we can see having the  $K_p$  term equal to 300 leads to a faster reaching of the steady state at the beginning (around 10000s), with respect to the other case, but due to the future interactions with the Earth magnetic field having the proportional term at 300 leads to high oscillations and to a settling time that is longer than having the  $K_p=100$ . This is something that can be expected since the proportional term in general leads to a quicker system but depending on the disturbances and interactions with the environment, the performances can be easily degraded. For what concerns the derivative term instead,  $K_d=18000$  was the best choice.

Now let's analyse two cases in which this controller can be used. Obviously a full magnetically actuated ADCS mounted on a satellite is not the best choice due to the well known problem of underactuation that in some situations can be critical. For this reason a full magnetic control system can be equipped in couple with a principal ADCS, and used for these situations where a contingent control system is needed: for example for the detumbling of the satellite, as seen before, or in particular situations where the actuation through other actuators like reaction wheels is not convenient or impossible (failure for example). So as first case a situation in which the satellite has been aligned with the LVLH frame is considered, and then the impact with a generic body is taken in consideration. The impact will trigger the rotations of the satellite, so we'll take as initial conditions for the system:  $q_v = [0; 0; 0]$ ,  $\omega_{bo}^b = [1.5; 1.5; 1.5] \text{ deg/s}$ .



*Figure 84: Earth-pointing controller, impact scenario simulation*

As is possible to see from the figure above, the controller is capable to counteract the angular velocity error and to reach the desired configuration in about 40000s, that means in about 7.5 orbits considering the TLE of the Commtrail nanosatellite.

At this point a general evaluation of the performances of the controller, considering different

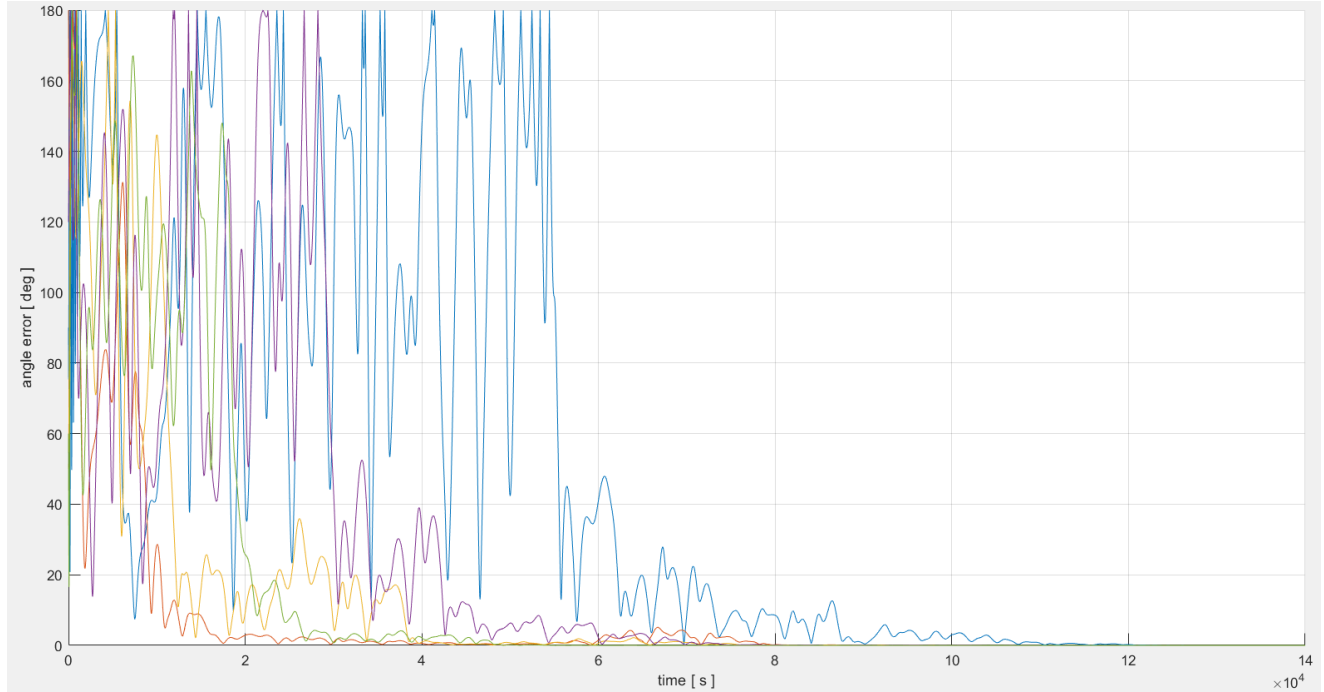
initial conditions can be performed, just as has been done for the detumbling controller. To this aim is possible to refer to the Table 5 below collecting the settings of the simulations that will be carried out.

<i>Simulation</i>	$\omega_{\tilde{0}} [deg/s]$ (wrt LVLH)	$q_{\tilde{0}} body$ (wrt LVLH)
1 (blue)	[1.5; 1.5; 1.5]	[0.7071; 0; -0.7071; 0] (90° around Y)
2 (red)	[0.5; 0.5; 0.5]	[0; 0; -1; 0] (180° around Y)
3 (yellow)	[-0.2; -0.2; 0.7]	[0.906; -0.422; 0; 0] (50° around X)
4 (purple)	[-0.1; 0.1; -0.3]	[0.5; 0; 0; -0.866] (120° around Z)
5 (green)	[1; 0.5; 0]	[0.866; -0.5; 0; 0] (60° around X)

*Table 5: B-dot controller, 5 simulations various initial conditions (attitude and ang. vel.)*

And the simulation results are analysed considering the angle error, expressed by the quaternion error, and the angular velocity error. Indeed recalling that the quaternion express a rotation in function of the angle of rotation  $\beta$  around an axis of rotation  $\mathbf{u}$ , it is possible to retrieve the angle error from the quaternion error in the following way:

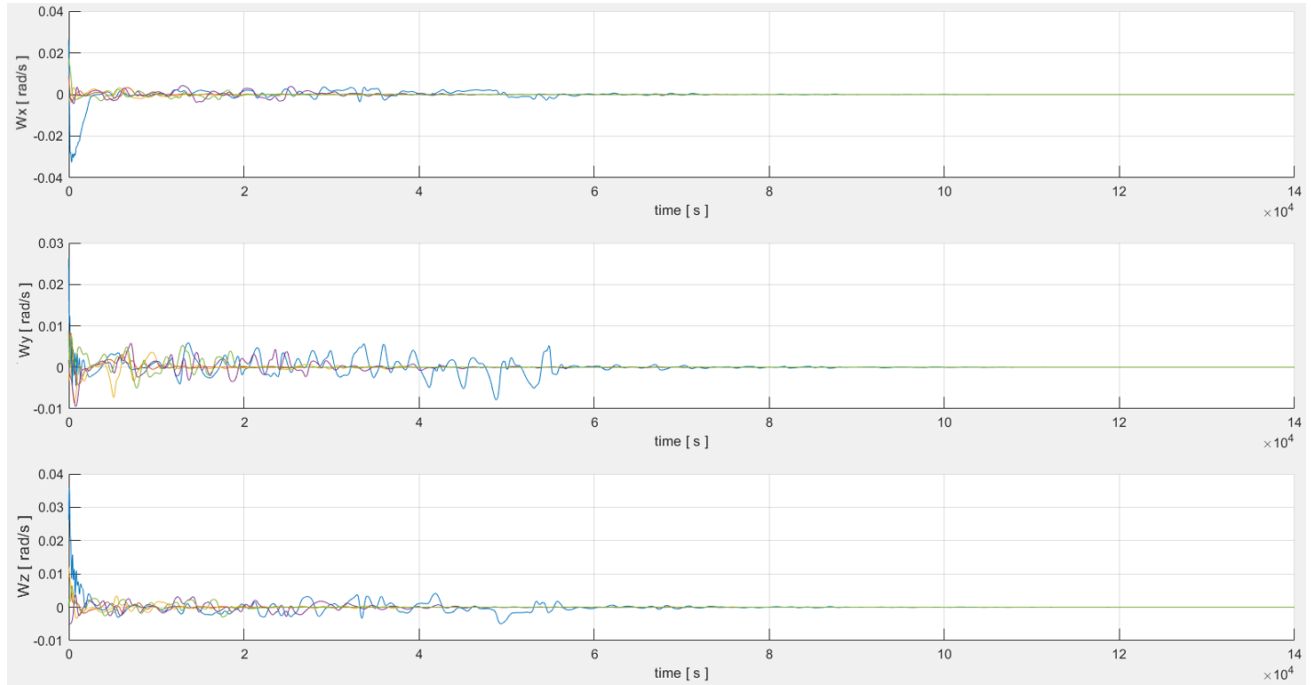
$$\mathbf{q}_{\tilde{}} = \begin{bmatrix} q_{0_{\tilde{}}} \\ \mathbf{q}_{v_{\tilde{}}} \end{bmatrix} = \begin{bmatrix} \cos\left(\frac{\beta}{2}\right) \\ \mathbf{u} \cdot \sin\left(\frac{\beta}{2}\right) \end{bmatrix} \rightarrow \beta = 2 \cdot \arccos(|q_{0_{\tilde{}}}|)$$



*Figure 85: Earth-pointing controller, error angle evaluation for the 5 simulations*

By checking the Figure 85 is possible to see that the designed controller, with the selected parameters values, is able to accomplish the task. Indeed the angle error reported in the figure

reaches the  $0^\circ$  at steady state, showing the alignment between the body frame and the LVLH frame. Also the angular velocities are controlled and, from the Figure 86 below, is possible to see that the angular velocity error is reduced to 0 rad/s.



*Figure 86: Earth-pointing controller, angular velocity error for the 5 simulations*

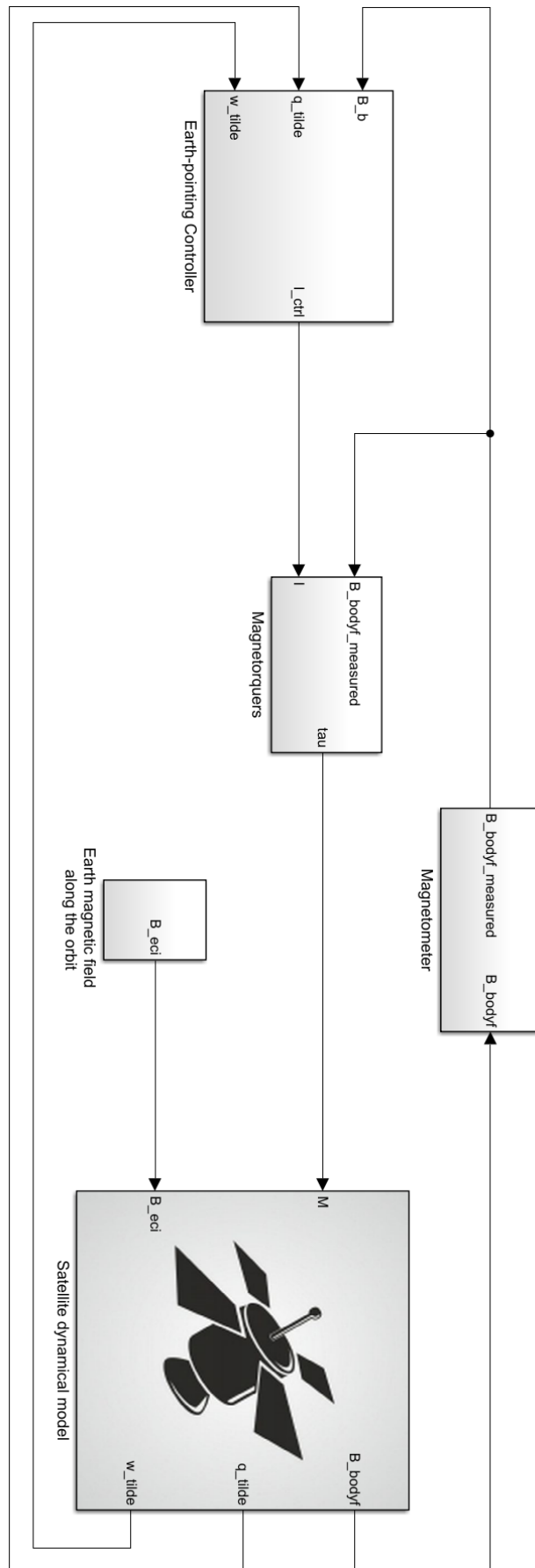


Figure 87: Simulink Earth-pointing model

### 6.3. ROS Toolbox and code auto-generation

After the control algorithms have been tested in different scenarios and their performances have been analysed, the last step is to transfer the Simulink model of the controller into a suitable embedded system. This approach is known as “Model Based Design”, as introduced in the introduction of this thesis, and is widely used in industry since all the effort that is needed for implementing the desired control system is related to the modelling and simulation in MATLAB/Simulink, while the coding part is done automatically. Also for this part ROS has its advantages that would make it a good candidate framework for building a flight software; thanks to the official support received by “MathWorks”, is possible to interface MATLAB/Simulink with ROS/ROS2.

In order to integrate our ROS network with MATLAB/Simulink, the official “ROS Toolbox” must be installed on the host machine (our PC with MATLAB/Simulink): the installation is very simple as this toolbox can be downloaded and installed as any other MATLAB add-on and after this the ROS Toolbox offers an interface able to create a node network running in part on the target system (Raspberry Pi 3 B+ in this case) and in part on the host system. It offers also premade functions and blocks for MATLAB and Simulink that allow to design and analyse the node network. Finally the toolbox allows to connect the ROS network with external simulators like Gazebo in order to visualize the correct behaviour of the overall system or a particular part of it, and then, when all the simulations are concluded is possible to generate the code automatically and to easily deploy the node in the network.

For deploying an automatically generated ROS node in the network we can follow two paths: generate the code and run it directly on the network or generate the code and then build the obtained package directly on the target with a manual procedure. Both ways have its own peculiarities, for example the automatic build/run procedure can be useful for testing small functionalities that must be implemented within a node and we don’t want to lose time by doing the manual procedure.

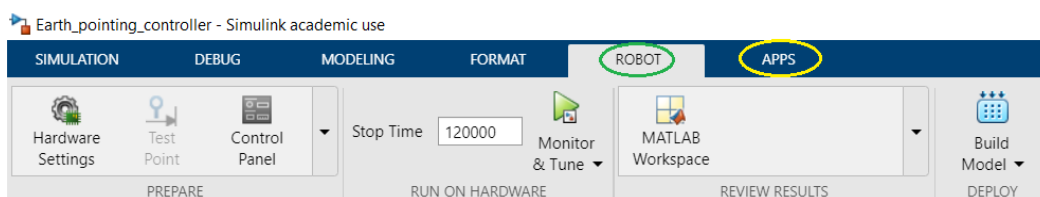


Figure 88: ROS Toolbox activation in Simulink

In this case there is an additional step needed for integrating the ROS network and the MATLAB system: an XML file must be included in both ROS and MATLAB workspace in order to make the two systems (previously connected to the same internet network) seeing each other. An example of this file can be seen in Figure 89 below, the structure is almost standard for every situation and the only thing that you have to configure is related to the IP addresses of our target and host systems.

For the thesis purposes this path is not the preferred one since only the procedure for bringing a Simulink model in the ROS network is studied, the test of the node is not performed since the provided hardware and facilities don’t allow this.

The other procedure is very simple and intuitive: first of all the system that is wanted to be converted in code must be isolated and then on Simulink, in the “Apps” section, the ROS Toolbox must be launched. As is possible to see in the Figure 88 above, the “ROBOT” menu appears.

```
<?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>
            <udp4>
              TARGET IP ADDRESS ON THE NETWORK
              <address>192.168.10.1</address>
            </udp4>
          </locator>
          <locator>
            <udp4>
              HOST IP ADDRESS ON THE NETWORK
              <address>192.168.10.2</address>
            </udp4>
          </locator>
        </initialPeersList>
      </builtin>
      <userTransports>
        <transport_id>veelpeers</transport_id>
      </userTransports>
      <useBuiltinTransports>false</useBuiltinTransports>
    </rtps>
  </participant>
</profiles>
```

Figure 89: XML file for establishing the ROS/MATLAB connection

The last step that must be done is to enter in the “Hardware Settings” section and to set everything is needed for performing the code auto-generation for our target system. In particular the following settings have been set:

- Solver: set a fixed step solver and its step size, 0.01 (100 Hz) for the controller. This is mandatory for auto-generating code.
- Hardware implementation: set ROS2 as hardware board and the ARM Cortes-A microprocessor for the Raspberry. Set the “Build” in build options instead of build and run.
- Simulation target: selected the C++ language because compatible with ROS2.
- Code generation: set the “Generate code only” and “Create code generation report” for analysing the obtained result.

So once everything is suitably set, we just have to press the “Build model” button in the “ROBOT” section and wait for the procedure to complete. Finally after the completion two windows will pop up reporting the eventual errors and warnings encountered. If the procedure has gone smoothly we would obtain something like in Figure 90, stating that the code has been successfully generated from the Simulink model.

The “Code Generation Report” is a very helpful tool where are reported the generated interfaces and variables and how the code has been organised: the main file, the files generated directly

from the Simulink model and all the other files containing the constant values defined in the workspace (“Data file”) along with the interfaces files.

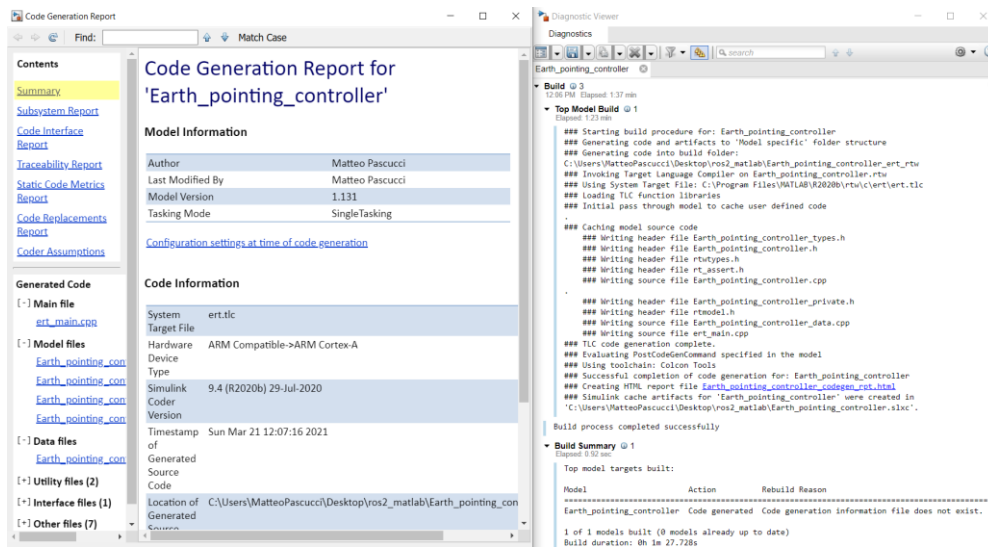


Figure 90: ROS Toolbox auto-generation of code final report

At the end of the procedure a “src” folder containing all the generated code is obtained. Inside this folder there is the package that will be manually deployed on the target, it contains everything needed by the ROS2 framework for enabling the generated node. The package must be placed in the ROS2 workspace in the target system and then all the workspace must be built (“colcon build” command from the workspace folder). Finally the ROS2 system is integrated with the auto-generated node, for checking if everything went good let’s launch the node: since the node is installed in the workspace we need to launch the executable generated in the “install/package\_name/lib/package\_name/” folder for launching it. At this point, as is possible to see from figure below, is easy to check if the node is in execution.

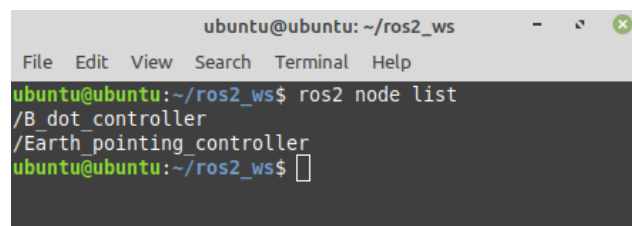


Figure 91: ROS2 generated nodes in execution

Both the controllers have been generated and successfully executed as ROS2 node after auto-generation.

## 7. CONCLUSIONS

This thesis project is part of an R&D project devoted to build a new avionic system and the flight software too. It is a smart choice to analyse and check if the ROS2 framework is suitable even for this kind of applications among the others in which it is widely used. The first part of the work is focused on setting up the system, both hardware and software, that can be used for developing the ROS2 software and how to integrate it with the provided sensor module. After this the flight software architecture has been addressed, in particular with the supervisor we spent time thinking about the possible applications that could be implemented and tested among the ones already present in the flight software normally used in the company. Due to my interests and possibilities of practically testing them, this choice involved the sensors. So first of all a watchdog node is analysed and implemented in order to have the other nodes correctly working, then the drivers for interfacing with the sensors have been written in python using pre-existing libraries for I2C and SPI protocols.

After this the project moved in MATLAB/Simulink for addressing the interesting problem of attitude control with only magnetic actuators. This control system can be cheaper and lighter with respect to other types but is affected by the problem of under actuation, and this has been a big problem to overcome in the Earth-pointing control scenario. Fortunately there are many papers and scientific researches on-line for addressing this problem under some assumptions. Finally after the validation of the control algorithms, the auto-generation of the code from Simulink models is performed and the generated node have been exported in the target system and launched along with the other nodes.

Considering improvements and future developments related to this work, I would suggest to bring all the present work into the custom board provided by the company since the Raspberry is a good system for starting to develop but will not be used directly inside the nanosatellite for performing its tasks in orbit. After this the work and the ROS2 network could be expanded by adding other applications, whenever there is availability of hardware in the company. There are many possibilities because a flight software, and in general the avionic system, is composed by many parts: related to the communication system, the power production with solar panels and etc... In my opinion it could be pretty interesting to implement a node devoted to manage the actuators and to exert the control actions, because by adding other actuators like reaction wheels the control system wouldn't be underactuated anymore.

Another possible work could be to translated all the code from python into C++ for improving the performances of the overall system.

## 8. APPENDIX A: BUILDROOT

In section 2.2 it has been introduced the possibility of installing a custom version of Linux-embedded for an embedded system.

Nowadays many companies, depending on their field of application, 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 capabilities that are introduced right for that particular application, since different combinations of devices can be mounted on it to achieve better performances for the task. On the other hand from the software point of view, in order to interface with the board it is necessary to realize a suitable custom image. For this reason, different tools like Buildroot or Yocto have been realized to easily realize images for embedded boards. In this Appendix the Buildroot tool is explained.

Buildroot is a tool that is used in order to generate embedded Linux images for different types of boards and finally cross-compile the image for the specified board. It provides as outputs the root filesystem, the kernel, the bootloader and all the files that are needed for a specific board to correctly flash 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.

In order to produce an image containing all the necessary, Buildroot must be configure and luckily it offers an intuitive user interface that can be summoned by typing the command “menuconfig” in a Linux shell. The “menuconfig” window appears and is presented in Figure 92 below.

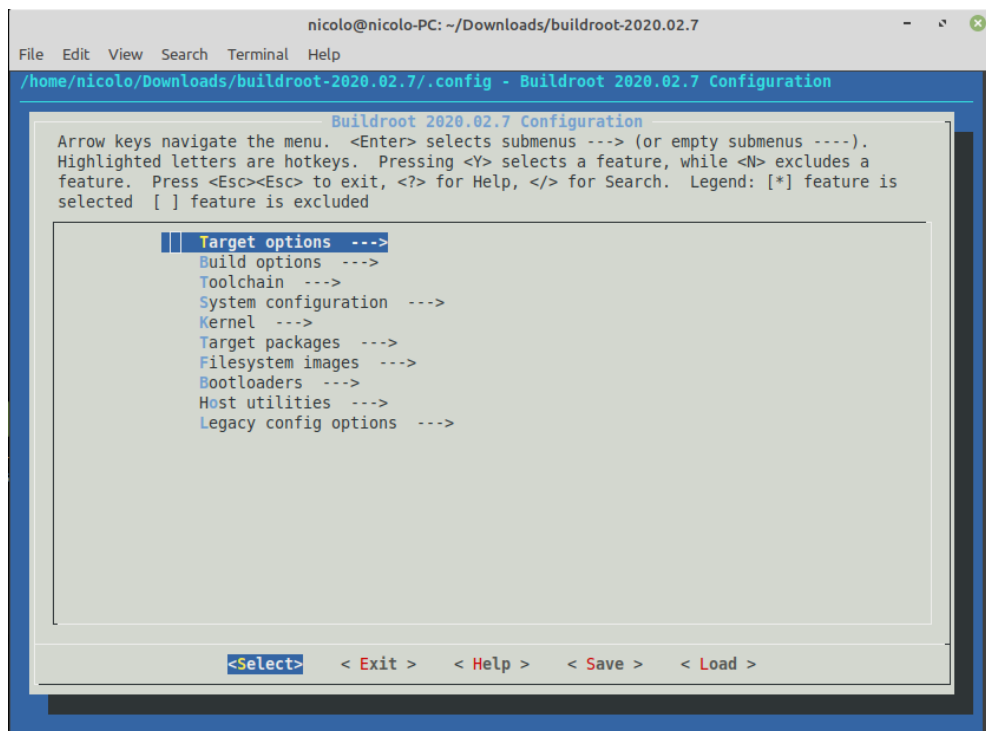


Figure 92: Buildroot 2020 “menuconfig” menu

Here the various sections of the menu are analysed:

- 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 an internal Buildroot or an external toolchain that will be used for the cross-compilation. 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 time-zones 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 to 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 is also possible to insert custom packages just like ROS2 and other particular dedicated applications.
- Filesystem Images: it allows to choose the output format of the generated filesystem (cpio, tar, jffs2 ecc...) and, if it is necessary, a compression method. 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 to manage its configuration.
- Host utilities: it allows to configure support tools for the host that can be useful in the building and cross-compiling processes.
- Legacy config options: packages that were present in older Buildroot versions that has been repropose.

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 *AT91SAM9G20* microprocessor.

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 already present in Buildroot. In the list of the supported boards of Buildroot 2020 is natively present the *AT91SAM9G20-EK* (that mounts an *AT91SAM9G20* processor) board. By using the command “*make at91sam9g20dfc\_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 flashing tool named “*Sam-ba*” (SAM Boot Assistant), which is commonly used to flash images on the SAM microprocessors. The main problem is that, using a standard image produced by Buildroot, the flashing procedure is successful but the board does not boot up because some parts are missing.

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, like done by using the standard configuration for 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.

## 9. APPENDIX B: ROS2 CODE

WATCHDOG NODE PYTHON 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_arguments="
")

    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_cfg.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()

```

## SPI BUS SENSORS READER NODE PYTHON 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()

```

## I2C BUS SENSORS READER NODE PYTHON 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()

```

## SPI BUS SENSORS TELEMETRY NODE PYTHON 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_"+requested_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()

```

## I2C BUS SENSORS TELEMETRY NODE PYTHON 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()

```

## ATTITUDE DETERMINATION NODE PYTHON 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_deter
mination/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_ECEF)
    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()

```

## 10. APPENDIX C: MATLAB CODE

ATTITUDE CONTROL SETTINGS FILE:

```
clc; clear
format long;

%% Orbit propagation data and LVLH frames import
%Storing altitude, latitude and longitude of the satellite in its
orbit
orbit_data=importdata('log_orbit.txt');
s=orbit_data.data(:,7); %seconds (for timeseries
structure creation)
mu=orbit_data.data(:,8); %latitude
l=orbit_data.data(:,9); %longitude
h=orbit_data.data(:,10); %altitude
orbit=timeseries([h, mu, l],s);

%Storing UTC time (year month day hours minutes seconds)
y=orbit_data.data(:,1);
mon=orbit_data.data(:,2);
d=orbit_data.data(:,3);
h=orbit_data.data(:,4);
min=orbit_data.data(:,5);
sec=orbit_data.data(:,6);
orbit_date_utc=timeseries([y, mon, d, h, min, sec],s);

% LVLH triads data
lvlh_data=importdata('LVLH_orbit.txt');
lvlh_x=timeseries([lvlh_data.data(:,2), lvlh_data.data(:,3),
lvlh_data.data(:,4)],s);
lvlh_y=timeseries([lvlh_data.data(:,5), lvlh_data.data(:,6),
lvlh_data.data(:,7)],s);
lvlh_z=timeseries([lvlh_data.data(:,8), lvlh_data.data(:,9),
lvlh_data.data(:,10)],s);

%Simulation
sim('orbit_data_computation');
B_eci=ans.B_eci;
q_ref=ans.q_eci2lvlh;

%B_eci plot
figure(1);
subplot(311); hold on; grid on; xlabel('time [ s ]'); ylabel('Bx [
T ]'); xlim([0 16*10^4]);
plot(B_eci.Time,B_eci.Data(:,1));
subplot(312); hold on; grid on; xlabel('time [ s ]'); ylabel('By [
T ]'); xlim([0 16*10^4]);
plot(B_eci.Time,B_eci.Data(:,2));
```

```

subplot(313); hold on; grid on; xlabel('time [ s ]'); ylabel('Bz [
T ]'); xlim([0 16*10^4]);
plot(B_eci.Time,B_eci.Data(:,3));

%% Settings definition
% Magnetorquers
N=1000; %Number of windings
D=0.03; %Coils diameter [m]
A_coil=D^2*pi/4; %Coils area [m^2]
Imax=0.7; %Max current [A] try 0.3 and 0.7
mu_max=N*Imax*A_coil; %max dipole [A*m^2]

Ax=[A_coil;0;0]; %Magnetorquer 1 area vector (aligned with
x-axis of body frame)
Ay=[0;A_coil;0]; %Magnetorquer 2 area vector (aligned with
y-axis of body frame)
Az=[0;0;A_coil]; %Magnetorquer 3 area vector (aligned with
z-axis of body frame)
A=[Ax,Ay,Az];
NA=N*A;
I=[Imax; Imax; Imax]; %Coils current matrix

% Satellite 3U with each U: mass 1.33Kg and length side 1dm
% Inertia components (Ix=0.0111 , Iy=0.0022 , Iz=0.0111)
q0=[1;0;0;0]; %ECI TO BODY FRAME quat
J=diag([0.0111; 0.0022; 0.0111]); %Inertia matrix
[Kg*m^2]
IJ=inv(J);

% Earth pointing control, reference angular velocity and Kp, Kd
matrices
mean_motion=15.00050640; % revolutions per days
n=(2*pi*mean_motion)/86400; % orbital rate, approximating to
circular orbit. 86400=seconds in a day
w_oio=[0; -n; 0]; % angular velocity of orbital frame
wrt inertial, written in orbital [rad/s]
q_ref0=q_ref.Data(1,:); % qeci2orbit initial
Kp=diag([1*10^2, 1*10^2, 1*10^2])
Kd=diag([1.8*10^4, 1.8*10^4, 1.8*10^4])
% initial conditions
% I want the body frame rotated of theta° wrt x/y/z axis of orbit
frame
th=deg2rad(60);
q_o2b=[cos(th/2); sin(th/2); 0; 0]; %wrt x
% q_o2b=[cos(th/2); 0; sin(th/2); 0]; %wrt y
% q_o2b=[cos(th/2); 0; 0; sin(th/2)]; %wrt z
q0=quatprod(q_ref0, q_o2b); %qeci2body initial
q0_tilde=[q_o2b(1); -q_o2b(2:end)]; %initial q_tilde (body to
orbit)

% w0=deg2rad([0; 0; 0]);

```

```

w_tilde0=deg2rad([1;0.5;0]);           %input in degrees
% w_tilde0=[0.01; 0.01; 0.01];         %input in radians

%% Animations/plots after simulation
% random initial conditions simulations for detumbling
w0_deg_sim=[];
q0_sim=[];
figure(2);
subplot(311); hold on; grid on; xlabel('time [ s ]'); ylabel('Wx [
deg/s ]');
subplot(312); hold on; grid on; xlabel('time [ s ]'); ylabel('Wy [
deg/s ]');
subplot(313); hold on; grid on; xlabel('time [ s ]'); ylabel('Wz [
deg/s ]');
for i=1:5
    q0=randrot(1).compact';
    q0_sim=[q0_sim, q0];
    w0_deg=-20+40.*rand(3,1);           %angular velocity in
degrees [deg] (randommly chosen in [-20,20]deg)
    w0_deg_sim=[w0_deg_sim, w0_deg];
    w0=deg2rad(w0_deg);                 %angular velocity in
radians [rad]
    sim('Detumbling_Bdot_model');
    w_deg=ans.w_deg;
    subplot(311); plot(w_deg.Time,squeeze(w_deg.Data(1,:)));
    subplot(312); plot(w_deg.Time,squeeze(w_deg.Data(2,:)));
    subplot(313); plot(w_deg.Time,squeeze(w_deg.Data(3,:)));
end

```

## EARTH-POINTING SATELLITE DYNAMICAL MODEL EULER EQUATION ERROR:

```
function w_bo_b_dot = fcn(M, q_tilde, w_bo_b)

mean_motion=15.00050640;      % revolutions per days
n=(2*pi*mean_motion)/86400;   % orbital rate, approximating to
circular orbit. 86400=seconds in a day
w_o_i_o=[0; -n; 0];          % angular velocity of orbital frame
wrt inertial, written in orbital [rad/s]
J=diag([0.0111; 0.0022; 0.0111]); %Inertia matrix
[Kg*m^2]
IJ=inv(J);

C=qua2dcm(q_tilde);

tmp=C*w_o_i_o;
s1=[0 -tmp(3) tmp(2); tmp(3) 0 -tmp(1); -tmp(2) tmp(1) 0];
tmp=w_bo_b;
s2=[0 -tmp(3) tmp(2); tmp(3) 0 -tmp(1); -tmp(2) tmp(1) 0];
tmp=J*C*w_o_i_o;
s3=[0 -tmp(3) tmp(2); tmp(3) 0 -tmp(1); -tmp(2) tmp(1) 0];

sum=(-J*s1-s2*J+s3-s1*J)*w_bo_b-s1*J*C*w_o_i_o+M;
w_bo_b_dot=IJ*sum;
```

## EARTH-POINTING SATELLITE DYNAMICAL MODEL QUATERNION KINEMATICS:

```
function w_bi_b = fcn(q_bo, w_bo_b)
mean_motion=15.00050640;      % revolutions per days
n=(2*pi*mean_motion)/86400;   % orbital rate, approximating to
circular orbit. 86400=seconds in a day
w_o_i_o=[0; -n; 0];          % angular velocity of orbital frame
wrt inertial, written in orbital

C=qua2dcm(q_bo);
w_o_i_b=C*w_o_i_o;
w_bi_b=w_bo_b+w_o_i_b;
```

## REFERENCES

1. ai-solutions, Attitude Reference Frames. s.d.  
[https://aisolutions.com/help/Files/attitude\\_reference\\_frames.htm](https://aisolutions.com/help/Files/attitude_reference_frames.htm).
2. Analog Devices, AD7415 sensor. s.d.  
[https://www.analog.com/media/en/technical-documentation/data-sheets/AD7414\\_7415.pdf](https://www.analog.com/media/en/technical-documentation/data-sheets/AD7414_7415.pdf).
3. Carlo Novara, Enrico Canuto, Luca Massotti, Donato Carlucci, Carlos Perez Montenegro. *Spacecraft Dynamics and Control*. s.d.
4. Documentation, ROS2 Foxy. s.d.  
<https://docs.ros.org/en/foxy/index.html>.
5. Elmos, E910.86 Sun sensor. s.d.  
[https://www.mouser.com/datasheet/2/594/910\\_86-224506.pdf](https://www.mouser.com/datasheet/2/594/910_86-224506.pdf).
6. Fabio Celani, Gain Selection for Attitude Stabilization of Earth-Pointing Spacecraft Using Magnetorquers. s.d. <https://link.springer.com/article/10.1007/s42496-020-00062-2#Abs1>.
7. James, R Wertz. *Spacecraft attitude Determination and Control*. s.d.
8. MathWorks, Aerospace Blockset Toolbox. s.d.  
<https://it.mathworks.com/matlabcentral/fileexchange/70030-aerospace-blockset-cubesat-simulation-library>.
9. MathWorks, Generate Code to Manually Deploy a ROS 2 Node from Simulink. s.d.  
<https://it.mathworks.com/help/ros/ug/generate-code-to-manually-deploy-ros-2-node.html>.
10. MathWorks, ROS Toolbox. s.d.  
<https://it.mathworks.com/products/ros.html>.
11. Parallax, Compass module HMC5883L magnetometer. s.d.  
[https://components101.com/asset/sites/default/files/component\\_datasheet/HMC5883L-Module-Datasheet.pdf](https://components101.com/asset/sites/default/files/component_datasheet/HMC5883L-Module-Datasheet.pdf).
12. Richard Becker, Dealing with the Quaternion Antipodal. s.d.  
<https://apps.dtic.mil/sti/pdfs/AD1043624.pdf>.
13. Satellite Wiki, Bdot law. s.d.  
[https://www.aero.iitb.ac.in/satelliteWiki/index.php/B\\_Dot\\_Law](https://www.aero.iitb.ac.in/satelliteWiki/index.php/B_Dot_Law).
14. Skyfield, Earth Satellites. s.d.  
<https://rhodesmill.org/skyfield/earth-satellites.html>.
15. Wikipedia, Raspberry Pi 3 B+. s.d.  
[https://it.wikipedia.org/wiki/Raspberry\\_Pi](https://it.wikipedia.org/wiki/Raspberry_Pi).