Politécnico do Porto

Instituto Superior de Engenharia do Porto

Design and Implementation of a Health and Usage Monitoring System (HUMS) for UAVs

Ricardo Sousa Craveiro

LEEC

Degree in Electrical and Computer Engineering

Departamento de Engenharia Eletrotécnica

Instituto Superior de Engenharia do Porto

September, 2023

This report partially satisfies the requirements defined for the Project/Internship course, in the 3rd year, of the Degree in Electrical and Computer Engineering.

Candidate: Ricardo Sousa Craveiro, No. 1191000, `1191000@isep.ipp.pt`

Scientific Guidance: David Pereira, `drp@isep.ipp.pt`

Company: CISTER/ISEP - Research Centre in Real-Time and Embedded Computing Systems

Advisor: Syed Aftab Rashid, `syara@isep.ipp.pt`

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200–072 Porto

September, 2023

Acknowledgements

First, I want to show my appreciation to the Research Centre in Real-Time & Embedded Computing Systems (CISTER) for allowing me to do this project. I also want to acknowledge Syed Aftab Rashid and David Pereira. Without their mentoring and help this project would not come to life.

In addition I need to show my appreciation to *Instituto Superior de Engenharia do Porto* (ISEP) alongside my teachers during this last years which helped me learn so much and helped me develop as a soon to be professional.

To finalise, I would like to show my gratitude towards my parents and my brother for all the support in the good and the bad times during my academic life and outside of it.

Abstract

The main goal of this project is the Design and Implementation of a Health and Usage Monitoring System (HUMS) for Unmanned Aerial Vehicles (UAVs).

Drones are becoming mainstream both in civilian life and in commercial applications. To ensure the safety and reliability of Drones used in mission/safety-critical applications, it is important to have a safety monitoring mechanism that ensures that each system/subsystem within the Drone is working as intended. This is done using a Health and Usage Monitoring System (HUMS). The Health and Usage Monitoring System (HUMS) collects, processes, and records data from a number of different sensors and systems on the Drone. This data is stored in a Data Base and is also communicated to the Ground Station to identify sensors/system's operational conditions.

Therefore, in this project is created a way to view the drone data in real-time and to store it in order to analyse it afterwards.

Keywords: Drones, HUMS, UAVs.

Resumo

O objetivo principal deste projeto é criar e implementar um sistema de monitorização da saúde e estado de uso (HUMS) para veículos aéreos não tripulados.

Drones estão a ganhar importância tanto na vida civil como em aplicações comerciais. Para assegurar a segurança e a confiabilidade dos Drones usados em missões críticas ou outras aplicações de alto risco é necessário ter um mecanismo de monitorização que confirma que cada sistema/sub-sistema do drone está a funcionar como pretendido. Isto é feito através do sistema de monitorização da saúde e estado de uso (HUMS). O HUMS obtem, processa e guarda informações de diferentes sensores e sitemas do drone. Esta informação é armazenada numa base de dados e enviada para a base terrestre onde é possivel identificar a condição dos sensores/sistemas do drone.

Neste projeto, é então criado uma forma de ver em tempo real o estado do drone e de guardar essas informações de forma a poderem ser posteriormente guardadas e analisadas.

Palavras-Chave: Drones, HUMS, UAVs.

Contents

List of Figures

Listings

List of Acronyms

API	Application Programming Interface
CSV	Comma-separated values
ESC	Electronic Speed Controller
GPS	Global Positioning System
GS	Ground Station
HUMS	Health and Usage Monitoring System
IMU	Inertial Measurement Unit
ISEP	*Instituto Superior de Engenharia do Porto*
RPM	Rotations per Minute
UAV	Unmanned Aerial Vehicle
UAVs	Unmanned Aerial Vehicles
UDP	User Datagram Protocol

Chapter 1

Introduction

1.1 Motivation

Unmanned Aerial Vehicles (UAVs) such as Drones are becoming mainstream in our day and age, both in civilian as well as in commercial applications. To ensure the safety and reliability of Drones used in mission/safety-critical applications, it is important to have a safety monitoring mechanism that ensures that each system/-subsystem within the Drone is working as intended. This is usually done using a Health and Usage Monitoring System (HUMS). Moreover besides continuously verifying the drone telemetry and sensors, saving that same information is of the most importance. Having all data keept for later analysis enables the enterprises and drone creators to find patterns which makes it easier to fix and find errors. Overall making drone production and their flights safer and more efficient.

1.2 Project Description

The goal of this project is to develop and integrate a HUMS for UAVs. The HUMS collects, processes, and records data from a number of different sensors and systems on the Drone. This data is stored in a Data Base (black box) and is also communicated to the Ground Station (GS) to identify sensors/system's operational conditions. This system should have information displayed in a graphical interface, similarly to a car dashboard. Examples of information to be displayed are the drone speed, how long the flight is taking, battery status and motor status. It is also expected to have all the information stored for later analysis. In order to accomplish this, various frameworks will be used. A more in depth description about frameworks in use will be in the Chapter 2 of this report. The basis of this project starts with the drone simulator PX4 Autopilot. Which by the use of the Python programming language connects to the MAVSDK libraries and controls the drone. With MAVSDK we can also get information about the status of the drone and it's sensors. Afterwards our python code connects with the database InfluxDB where all data is saved and sent into Grafana where the dashboard is created and displayed. A C++ code was also developed, in the that code is gathered the same information as in Python, the difference is in the fact that in C++ instead of the data been sent to a database and afterwards to Grafana, the information is saved in different CSV files, creating a less effective but easier to implement HUMS version.

1.3 Objectives

The main objectives of this project are:

- Collect telemetry information from the drone;
- Store all the Drone information;
- Create a real-time dashboard with the drone telemetry information;

1.4 Contributions

In this project two separate versions of HUMS are developed. One version being C++ and the other one being developed in Python.

Both versions are meant to store and analyse drone data. The difference between both versions is the fact that the C++ version is intended to be used post-flight and the Python during or after flight. This happens because the C++ versions saves all the data in different Comma-separated values (CSV) files and the Python versions creates a dashboard where all data is displayed in real-time.

1.5 Report Organization

This report is organized as follows. It starts with an introduction to the project, as well as the motivation and this project goals. Afterwards a brief history about UAVs is given. Additionally there is a brief introduction about a well known HUMS in the market, and in the same chapter there is also a presentation of the frameworks used to make this project possible. Further along there is an explanation about how this project was done and the methods used in it's development cycle. First the report starts by explaining the C++ version of the HUMS which is an early version of the final product and afterwards its discussed the actual HUMS with a real time dashboard which was done using Python. To finish the report, conclusions about the work done are mentioned, in addition suggestions to make this project better are made.

Chapter 2

State of the Art

2.1 Unmanned Aerial Vehicles

Unmanned Vehicles are vehicles that do not need a driver or a pilot present in the vehicle itself to be controlled[1], for a long time they were just a dream to most people. Thinking about cars, planes or boats who could safely drive and go to places without anybody present was only a mirage of the future, just like flying cars today are far from being something that we will see in our daily lives.

As most of major inventions achieved in the last decades, Unmanned Vehicles were first created by the military. They appeared in the late 1840s and were Unmanned Aerial Vehicles in the form of balloons [1]. They were used by the Austrian Navy as incendiary balloons, having hot air and an incendiary device attached. Although not very effective they were the first major start to this technological revolution.

Unmanned Aerial Vehicles as we know it, like automated drones or remotely piloted aerial vehicles were first developed in Britain and in the USA during the First World War. During this war were developed radio-controlled aircraft, called Aerial Target, as well as aerial torpedoes, the first to fly being called Kettering Bug. Neither were used during the war, but without a doubt paved the way into UAVs as it is known today. Afterwards, unmanned aircraft were used for various reasons. They were used as targets for military training, acted as decoys during other wars and soon after drones were able to launch missiles into fixed targets.[2]

Nowadays, Unmanned Vehicles are getting more and more attention. They are becoming more common than ever and are recognised as the future. Everyone wants an autonomous vehicle. Everyone wants to commute from point A to point B be it by car, plane or boat without wanting to drive the machine itself. This will be the future. Today we already see car brands like Tesla developing fully automated cars, who drive on the roads like humans but without the human error factor. They can communicate with each other and find when is the right moment to accelerate, break or turn. Nevertheless, Unmanned Vehicles are all the more important than ever been and not only automated cars are important.

Fully automated aircraft are of crucial importance to our future. Even though, almost everybody focus on automated cars, the future can not be only travelled by land. Land nowadays is filled with traffic, houses and much more. As we did before with land and water, we now need to fully conquer the air, this being by the use of Unmanned Aerial Vehicles. Drones are becoming increasingly more important and used, being used by military purpose or even by everyday purposes, like taking videos or photos from places we can not reach by ourselves. UAVs can be also used for visual place recognition and to help in crisis situations, like for example, if the need arises drones are able to assist people in places of hard access.

Portugal already noticed UAVs importance. We can understand this if we take a look at our national project called FLY.PT. FLY.PT aims to develop a personal air transport system, which consists of an autonomous aerial vehicle as well as an autonomous land vehicle. This project is being developed nation wide and one of it's developers being *Instituto Superior de Engenharia do Porto* (ISEP), more specifically being developed by the Research Centre in Real-Time and Embedded Computing Systems[3].

2.2 QGroundControl

QGroundControl is an example of a established HUMS, developed by Dronecode, which is a non-profit foundation. It mainly functions as a Ground Control station, giving you the ability to control the drone by the MAVLink Protocol, which is a drone controller programming protocol, while also being able to check the drone telemetry. Furthermore, it is also capable of showing the location of the drone in the world map. It's a powerful tool suited for beginners and to professional drone users. It supports drone controllers like PX4 Autopilot and ArduPilot.

2.3 Frameworks in Use

To create the HUMS, various frameworks were used. Figure 2.1 represents how the frameworks work in order.

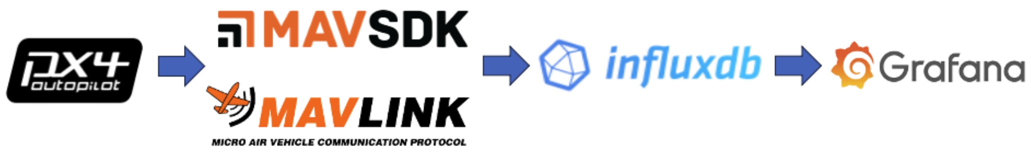


Figure 2.1: Framework Working Order

If need be this framework can be adapted to cases such as if the drone is in danger or in a crisis situation. In this case MAVSDK could give indications to PX4 Autopilot.

Every framework used in this project has tutorials in how to download and configure to each desired outcome.

2.3.1 PX4 Autopilot

PX4 is an open source software, and like MAVSDK and QGroundControl is part of Dronecode [4]. It's main use is to control drones and other unmanned vehicles. It provides support to drone hardware and software. It can be used in consumer drones or even in industrial applications. Being an open source software it's always improving and anyone can help developing it. It has an active community on places like GitHub, having dedicated community channels. Furthermore, is beginner friendly, having tutorials and walkthroughs on the basics of PX4.[5]

Giving the fact that during the making of this project real-life drones were not available, in order to develop the Health and Usage Monitoring System, simulators were used in connection with PX4. PX4 itself supports a different number of simulators, in the making of this project the main simulators used were jMAVSim and Gazebo. The main difference between both simulators are in the native supported vehicles, with Gazebo supporting quad drones, like jMAVSim, moreover Gazebo also supports planes, rovers and submarines[6] [7]. Nevertheless, more types of vehicles can be added through SITL (Software in the Loop) simulation, they can be added to both of the simulators mentioned before.

2.3.2 MAVSDK

Also an open source software owned by Dronecode, MAVSDK consists in a collection of libraries mainly written in C++, with wrappers for languages like Python, Swift, Java and many more[8]. This libraries are a bridge between the MAVLink protocol and the user. MAVSDK provides a simple Application Programming Interface (API) between various number of vehicles at the same time. It's able to provide control over missions, movement and other operations while also providing information about telemetry.

2.3.3 Pymavlink

Pymavlink is, like MAVSDK, a group of libraries. These libraries, like MAVSDK, provide a easy communication between the user and the MAVLink Protocol. In this project Pymavlink, more precisely the mavutil module is used to enable messages which MAVSDK does not support. This protocol works by requesting and signing to those messages. Examples of messages needed in this project are messages like Electronic Speed Controller (ESC) Rotations per Minute (RPM) values, as well as the their voltage and current consumption. These libraries are used in systems like QGroundControl, DroneKit and much more.[9]

2.3.4 InfluxDB

InfluxDB was the chosen database to save all flight data. Owned and developed by InfluxData. Widely used from startups to enterprises, it is mainly used for networking, Internet-of-Things and real-time analytical monitoring, it can be hosted locally or in the cloud. It has built-in support for a ranging number of source codes, from languages like Go, Arduino, Node.js, and Python, InfluxDB as a wide range of options.

The version used is InfluxDB 2.0, which has it's own query language called Flux, designed for querying and analyzing data. Furthermore, InfluxDB 2.0 also has the feature to create graphical interfaces with the stored data. The reason why their graphical interface was not used was because of it's refresh time. Only being able to refresh every 10 seconds automatically, it had a lengthy refresh-rate in comparison with what this project intended, which is a real-time dashboard to analyse all data as fast and efficiently as possible.[10]

2.3.5 Grafana

To finalize, the last framework used was Grafana. Similarly to InfluxDB, Grafana can also be used both locally and in the cloud. It provides an powerful and easy way to display data. With full support connecting to several databases, one of them being InfluxDB, and with multiple plugins it was the best option to create a dashboard. Grafana connects to InfluxDB by the Flux querying language, previously mentioned. Alongside InfluxDB, Grafana displays the information in a dashboard costume made by the user. It has an array of different options, from gauges to graphics or even tables. Every single type of display format has even more options to tailor into a perfect dashboard, being without a doubt one of the strongest graphical interfaces builder tools. Moreover, it has the ability to change the refresh-rate to as low as a 100ms rate.

Chapter 3

C++ Implementation

The Health and Usage Monitoring System implementation was done using PX4 Autopilot as a drone controller. When simulators such as Gazebo or jMAVSim simulate UAVs with PX4 Autopilot, they are able to send MAVLink messages. This messages are the core of how the HUMS works. These messages work as an actual drone message would, making it almost as real as they could be and making this project ready to be adapted into a real life drone flight. In order to develop this code, the example "fly_mission.cpp" was used as a base for gathering information. The code developed in this chapter consists in functions whose purpose is data collection.

This C++ version of the HUMS developed is not a real-time one. In this version the data is collected and saved into various CSV files, making it ideal to be used when a real time display is not needed.

3.1 C++ preparation for Information Gathering

Like previously mentioned, to make this project possible, first is created various CSV files, to which afterwards is written all the data.

```

    std::ofstream myfile1;
    myfile1.open("flightvalues.csv");
    myfile1 << "Flight Time; Battery %; Battery (V); Temp;
        Air Pressure; NED Velocity; Speed; Velocity; FRD
        Accel; Accel module; Altitude; Latitude; Longitude\n
        ";
    myfile1.close();

    std::ofstream myfile2;
    myfile2.open("esc_status.csv");
    myfile2 << "esc_rpm1; esc_rpm2; esc_rpm3; esc_rpm4;
        esc_voltage1; esc_voltage2; esc_voltage3;
        esc_voltage4; esc_current1; esc_current2;
        esc_current3; esc_current4;\n";
    myfile2.close();

    std::ofstream myfile3;
    myfile3.open("esc_info.csv");
    myfile3 << "escfail1; escfail2; escfail3; escfail4;
        esc1_fails; esc2_fails; esc3_fails; esc4_fails;
        esc_temperature1; esc_temperature2; esc_temperature3
        ; esc_temperature4; \n";
    myfile3.close();

    std::ofstream myfile4;
    myfile4.open("vfr_hud.csv");
    myfile4 << "Airspeed; Groundspeed; Heading; Throttle;
        Altitude; Climbrate\n";
    myfile4.close();

    //It saves the time the drone started flying
    const auto start_time = std::chrono::steady_clock::now
        ();
```

Listing 3.1: CSV file creation

These four different files are the files to which all information is saved. The first file saves all general information provided by the C++ MAVSDK version and the last three is where MAVSDK Passthrough information is saved, each CSV files contains information from different drone received messages. After the creation of the CSV files, a variable is created where is saved the moment when the drone started flying. To finish the preparations for collecting data, is created a time thread.

The time thread is a function that, in this case, waits a second and afterwards it repeats itself and with each repetition all of the functions inside it occur. Here is the time thread created:

```

    std::thread timer_thread([&](){
        while (true) {
            telemetry_callback(telemetry, start_time);

            send_esc_status(mavlink_passthrough);
            send_esc_info(mavlink_passthrough);
            send_vfr_hud(mavlink_passthrough);

            send_raw_rpm(mavlink_passthrough);      //the
                callback does not happen
            send_efi_status(mavlink_passthrough);   //the
                callback does not happen

            // Wait for 1 second
            std::this_thread::sleep_for(std::chrono::
                seconds(1));
        }
    });
```

Listing 3.2: Time Thread

3.2 Acquiring Telemetry information from MAVSDK C++

In listing 3.2, information about the drone is extracted in two different ways. One way being the directly from the MAVSDK libraries. This direct extraction is done in the function "telemetry_callback(telemetry, start_time)". Here is the function mentioned:

```

void telemetry_callback(mavsdk::Telemetry& telemetry, std::chrono::steady_clock::time_point start_time)
{

    // Read the battery information
    const auto battery = telemetry.battery();
    //std::cout << "Battery? " << battery.voltage_v << '\n';

    // Read the North-East-Down Velocity
    const auto velocity_ned = telemetry.velocity_ned();
    const auto speed1 = cbrt(velocity_ned.north_m_s*velocity_ned.north_m_s+velocity_ned.east_m_s*velocity_ned.east_m_s+velocity_ned.down_m_s*velocity_ned.down_m_s);

    const auto speed = telemetry.raw_gps();

    // Read imu -> acceleration, angular velocity, magnetic field, temperature and timestamp_us
    const auto imu = telemetry.imu();
    const auto accel = cbrt(imu.acceleration_frd.forward_m_s2*imu.acceleration_frd.forward_m_s2+imu.acceleration_frd.right_m_s2*imu.acceleration_frd.right_m_s2+imu.acceleration_frd.down_m_s2*imu.acceleration_frd.down_m_s2);

    //Remote control status
    const auto rcstatus = telemetry.rc_status();

    // Pressure
    const auto pressure = telemetry.scaled_pressure();

    //GPS
    const auto gps = telemetry.get_gps_global_origin();

    // Read the current time
    const auto current_time = std::chrono::steady_clock::now();

    // Calculate the flight time
    const auto flight_time = std::chrono::duration_cast<std::chrono::seconds>(current_time - start_time);
```

Listing 3.3: "telemetry_callback" Function Part 1

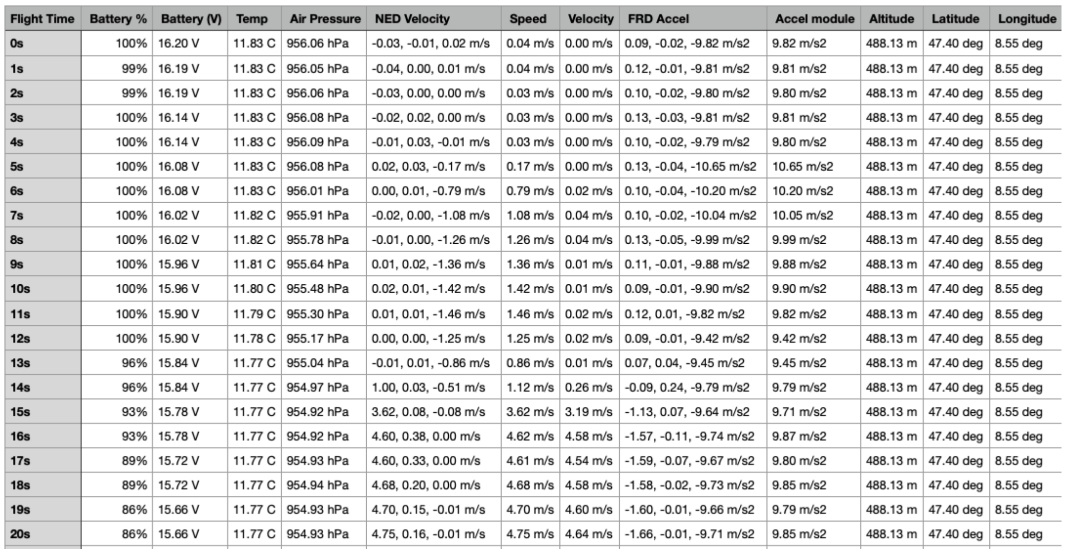
```


    std::ofstream myfile1;
    myfile1.open("flightvalues.csv", std::ios::cur);
    myfile1 << flight_time.count() << "s;" << std::fixed <<
         std::setprecision(0) << battery.remaining_percent
       *100 << " %; "
            << std::fixed << std::setprecision(2) <<
                battery.voltage_v << " V;"
            << std::fixed << std::setprecision(2) << imu.
                temperature_degc << " C;"
            << std::fixed << std::setprecision(2) <<
                pressure.absolute_pressure_hpa << " hPa;"
            << std::fixed << std::setprecision(2) <<
                velocity_ned.north_m_s
            << ", " << std::fixed << std::setprecision(2)
                << velocity_ned.east_m_s
            << ", " << std::fixed << std::setprecision(2)
                << velocity_ned.down_m_s << " m/s;"
            << speed1 << " m/s;"
            << speed2.velocity_m_s << " m/s;"
            << std::fixed << std::setprecision(2) << imu.
                acceleration_frd.forward_m_s2
            << ", " << std::fixed << std::setprecision(2)
                << imu.acceleration_frd.right_m_s2
            << ", " << std::fixed << std::setprecision(2)
                << imu.acceleration_frd.down_m_s2 << " m/s2;
                "
            << accel << " m/s2;"
            << std::fixed << std::setprecision(2) << gps.
                second.altitude_m << " m;"
            << std::fixed << std::setprecision(2) << gps.
                second.latitude_deg << " deg;"
            << std::fixed << std::setprecision(2) << gps.
                second.longitude_deg << " deg\n";
    myfile1.close();

}
```

Listing 3.4: "telemetry_callback" Function Part 2

In the function above a lot of different information is given. By sending the "Telemetry" variable, which contains the system class "Telemetry", we can read all the telemetry data structures contained in the Telemetry class. Not all of the information contained is in the function above for simplicity reasons as well as because not every information is needed. Examples of information provided by the code above are the air pressure where the drone is, the battery percentage and the status of the drone remote control.

After collecting some data, everything is sent into a CSV file, where is stored and kept to be analyzed if the need raises.

Here is an example of part of the CSV file generated:



Flight Time	Battery %	Battery (V)	Temp	Air Pressure	NED Velocity	Speed	Velocity	FRD Accel	Accel module	Altitude	Latitude	Longitude
0s	100%	16.20 V	11.83 C	956.06 hPa	-0.03, -0.01, 0.02 m/s	0.04 m/s	0.00 m/s	0.09, -0.02, -9.82 m/s2	9.82 m/s2	488.13 m	47.40 deg	8.55 deg
1s	99%	16.19 V	11.83 C	956.05 hPa	-0.04, 0.00, 0.01 m/s	0.04 m/s	0.00 m/s	0.12, -0.01, -9.81 m/s2	9.81 m/s2	488.13 m	47.40 deg	8.55 deg
2s	99%	16.19 V	11.83 C	956.06 hPa	-0.03, 0.00, 0.00 m/s	0.03 m/s	0.00 m/s	0.10, -0.02, -9.80 m/s2	9.80 m/s2	488.13 m	47.40 deg	8.55 deg
3s	100%	16.14 V	11.83 C	956.08 hPa	-0.02, 0.02, 0.00 m/s	0.03 m/s	0.00 m/s	0.13, -0.03, -9.81 m/s2	9.81 m/s2	488.13 m	47.40 deg	8.55 deg
4s	100%	16.14 V	11.83 C	956.09 hPa	-0.01, 0.03, -0.01 m/s	0.03 m/s	0.00 m/s	0.10, -0.02, -9.79 m/s2	9.80 m/s2	488.13 m	47.40 deg	8.55 deg
5s	100%	16.08 V	11.83 C	956.08 hPa	0.02, 0.03, -0.17 m/s	0.17 m/s	0.00 m/s	0.13, -0.04, -10.65 m/s2	10.65 m/s2	488.13 m	47.40 deg	8.55 deg
6s	100%	16.08 V	11.83 C	956.01 hPa	0.00, 0.01, -0.79 m/s	0.79 m/s	0.02 m/s	0.10, -0.04, -10.20 m/s2	10.20 m/s2	488.13 m	47.40 deg	8.55 deg
7s	100%	16.02 V	11.82 C	955.91 hPa	-0.02, 0.00, -1.08 m/s	1.08 m/s	0.04 m/s	0.10, -0.02, -10.04 m/s2	10.05 m/s2	488.13 m	47.40 deg	8.55 deg
8s	100%	16.02 V	11.82 C	955.78 hPa	-0.01, 0.00, -1.26 m/s	1.26 m/s	0.04 m/s	0.13, -0.05, -9.99 m/s2	9.99 m/s2	488.13 m	47.40 deg	8.55 deg
9s	100%	15.96 V	11.81 C	955.64 hPa	0.01, 0.02, -1.36 m/s	1.36 m/s	0.01 m/s	0.11, -0.01, -9.88 m/s2	9.88 m/s2	488.13 m	47.40 deg	8.55 deg
10s	100%	15.96 V	11.80 C	955.48 hPa	0.02, 0.01, -1.42 m/s	1.42 m/s	0.01 m/s	0.09, -0.01, -9.90 m/s2	9.90 m/s2	488.13 m	47.40 deg	8.55 deg
11s	100%	15.90 V	11.79 C	955.30 hPa	0.01, 0.01, -1.46 m/s	1.46 m/s	0.02 m/s	0.12, 0.01, -9.82 m/s2	9.82 m/s2	488.13 m	47.40 deg	8.55 deg
12s	100%	15.90 V	11.78 C	955.17 hPa	0.00, 0.00, -1.25 m/s	1.25 m/s	0.02 m/s	0.09, -0.01, -9.42 m/s2	9.42 m/s2	488.13 m	47.40 deg	8.55 deg
13s	96%	15.84 V	11.77 C	955.04 hPa	-0.01, 0.01, -0.86 m/s	0.86 m/s	0.01 m/s	0.07, 0.04, -9.45 m/s2	9.45 m/s2	488.13 m	47.40 deg	8.55 deg
14s	96%	15.84 V	11.77 C	954.97 hPa	1.00, 0.03, -0.51 m/s	1.12 m/s	0.26 m/s	-0.09, 0.24, -9.79 m/s2	9.79 m/s2	488.13 m	47.40 deg	8.55 deg
15s	93%	15.78 V	11.77 C	954.92 hPa	3.62, 0.08, -0.08 m/s	3.62 m/s	3.19 m/s	-1.13, 0.07, -9.64 m/s2	9.71 m/s2	488.13 m	47.40 deg	8.55 deg
16s	93%	15.78 V	11.77 C	954.92 hPa	4.60, 0.38, 0.00 m/s	4.62 m/s	4.58 m/s	-1.57, -0.11, -9.74 m/s2	9.87 m/s2	488.13 m	47.40 deg	8.55 deg
17s	89%	15.72 V	11.77 C	954.93 hPa	4.60, 0.33, 0.00 m/s	4.61 m/s	4.54 m/s	-1.59, -0.07, -9.67 m/s2	9.80 m/s2	488.13 m	47.40 deg	8.55 deg
18s	89%	15.72 V	11.77 C	954.94 hPa	4.68, 0.20, 0.00 m/s	4.68 m/s	4.58 m/s	-1.58, -0.02, -9.73 m/s2	9.85 m/s2	488.13 m	47.40 deg	8.55 deg
19s	86%	15.66 V	11.77 C	954.93 hPa	4.70, 0.15, -0.01 m/s	4.70 m/s	4.60 m/s	-1.60, -0.01, -9.66 m/s2	9.79 m/s2	488.13 m	47.40 deg	8.55 deg
20s	86%	15.66 V	11.77 C	954.93 hPa	4.75, 0.16, -0.01 m/s	4.75 m/s	4.64 m/s	-1.66, -0.01, -9.71 m/s2	9.85 m/s2	488.13 m	47.40 deg	8.55 deg

Figure 3.1: CSV File Generated by "telemetry_callback" Function

3.3 Acquiring Telemetry information from MAVLink Protocol

All the information intended to be displayed in the HUMS is not available within the MAVSDK libraries. For that reason a direct link with the MAVLink Protocol was needed. In order to make that through MAVSDK, the MAVSDK C++ is the only MAVSDK library which provides full access to MAVLink. The Phython version does not enable the user direct communication with the MAVLink Protocol due to security reasons. Having this in mind more functions were made to gather even more real-time drone data. This functions being:

- send_esc_status(MavlinkPassthrough& mavlink_passthrough)
- send_esc_info(MavlinkPassthrough& mavlink_passthrough)
- send_vfr_hud(MavlinkPassthrough& mavlink_passthrough)

The first two functions created focus on the drone ESC which are the drone motors. In the simulations created the drone had 4 ESC motors. In "send_esc_status" is received data from each individual motor RPM, voltage and current used. In "send_esc_info", like in the previous function, there is information about each motor, this information being if the ESC is failing, how many times it failed up to that moment as well as their respective temperature. The last function is "send_vfr_hud" in which we receive information about the Unmanned Aerial Vehicle (UAV) ground speed and where the drone is heading in degrees, from 0 to 360 degrees, with zero being north.

Here is the "send_esc_status" function:

```
void send_esc_status (MavlinkPassthrough& mavlink_passthrough)
{
 mavlink_passthrough.subscribe_message_async(
        MAVLINK_MSG_ID_ESC_STATUS,
        [](const mavlink_message_t& message) {

            mavlink_esc_status_t status;
            mavlink_msg_esc_status_decode(&message, &status);
            {
                auto esc_rpm1 = status.rpm[0];
                    //std::cout << "ESC 1 rpms ?" << esc_rpm1 << '\n';
                auto esc_rpm2 = status.rpm[1];
                    //std::cout << "ESC 2 rpms ?" << esc_rpm2 << '\n';
                auto esc_rpm3 = status.rpm[2];
                    //std::cout << "ESC 3 rpms ?" << esc_rpm3 << '\n';
                auto esc_rpm4 = status.rpm[3];
                    //std::cout << "ESC 4 rpms ?" << esc_rpm4 << '\n';

                auto esc_voltage1 = status.voltage[0];
                    //std::cout << "ESC 1 voltage ?" << esc_voltage1 << '\n';
                auto esc_voltage2 = status.voltage[1];
                    //std::cout << "ESC 2 voltage ?" << esc_voltage2 << '\n';
                auto esc_voltage3 = status.voltage[2];
                    //std::cout << "ESC 3 voltage ?" << esc_voltage3 << '\n';
                auto esc_voltage4 = status.voltage[3];
                    //std::cout << "ESC 4 voltage ?" << esc_voltage4 << '\n';
```

Listing 3.5: Function "send_esc_status" Part 1

```

                auto esc_current1 = status.current[0];
                     //std::cout << "ESC 1 current ?" << esc_current1 << '\n';
                auto esc_current2 = status.current[1];
                     //std::cout << "ESC 2 current ?" << esc_current2 << '\n';
                auto esc_current3 = status.current[2];
                     //std::cout << "ESC 3 current ?" << esc_current3 << '\n';
                auto esc_current4 = status.current[3];
                     //std::cout << "ESC 4 current ?" << esc_current4 << '\n';


                std::ofstream myfile2;
                myfile2.open("esc_status.csv", std::ios::cur);
                myfile2 << std::fixed << std::setprecision(0) << esc_rpm1 << " rpms;"
                    << std::fixed << std::setprecision(0) << esc_rpm2 << " rpms;"
                    << std::fixed << std::setprecision(0) << esc_rpm3 << " rpms;"
                    << std::fixed << std::setprecision(0) << esc_rpm4 << " rpms;"
                    << std::fixed << std::setprecision(2) << esc_voltage1 << " V;"
                    << std::fixed << std::setprecision(2) << esc_voltage2 << " V;"
                    << std::fixed << std::setprecision(2) << esc_voltage3 << " V;"
                    << std::fixed << std::setprecision(2) << esc_voltage4 << " V;"
                    << std::fixed << std::setprecision(2) << esc_current1 << " A;"
                    << std::fixed << std::setprecision(2) << esc_current2 << " A;"
                    << std::fixed << std::setprecision(2) << esc_current3 << " A;"
                    << std::fixed << std::setprecision(2) << esc_current4 << " A\n";
                myfile2.close();
            }
        });
}
```

Listing 3.6: Function "send_esc_status" Part 2

These functions work in a similar way to the Phython functions, which will be seen later in this report. However in this case a MavlinkPassthrough variable is needed. This variable is the only way to directly communicate with the MAVLink Protocol. In the example provided before, the code subscribes to the message, in this case to the message "ESC_STATUS" and waits for a response from the PX4 Autopilot simulator. When receiving the answer the message is saved and is decoded into a variable called status, which type "mavlink_esc_status_t" is specific to decoding "ESC_STATUS" messages. Every message type has a different decoder. Afterwards the information of the message, like the respective ESC RPM, voltage and current is read. To receive and decode different messages the same principle of requesting messages subscriptions and messages decoding is applied, after the decoding is done everything is saved in the CSV file. Not every MAVLink message is sent from the simulators used, because of that more functions were created but could not be used. This functions were the function "send_raw_rpm(MavlinkPassthrough& mavlink_passthrough)" and the function "send_efi_status(MavlinkPassthrough& mavlink_passthrough)". The first function retrieves information from a RPM sensor which says the motor frequency as well as the sensor index. The second function can retrieve much more data, in this case it gathers information about a drone electronic fuel injection system. The function can provide data on the fuel consumed, fuel flow, engine load, spark dwell time, as well as information such as injection timing and ignition timing, more information is provided by this function besides what was already mentioned.

Nevertheless, if the need rises the subscription to new messages is of ease implementation. All of the MAVLink Messages can be found here: `https://mavlink.io/en/messages/common.html`.

Chapter 4

Python to Grafana Implementation

This chapter of the dissertation will be focused on the more complex and complete version of the HUMS. Here is where an explanation on what was done, from the information gathering moments up until the part in which the data is displayed, in this case it will be explained what was done in Python, the Python to database connection, the database to Grafana connection, as well as the Grafana user-interface development. This version of HUMS is optimal for real-time display of drone information. Making it the best version for long flights or for more important drone missions.

In order to get the telemetry values a mission flight example code from MAVSDK was used, the example code is called "mission.py". This code connects to the PX4 Autopilot and imports a mission to the drone. After that moment the drone starts flying in a fixed path. Having this in consideration this chapter will give an explanation in how this HUMS was developed.

The next image shows one of the simulators used in combination with PX4, in this case shows the jMAVSim simulator:

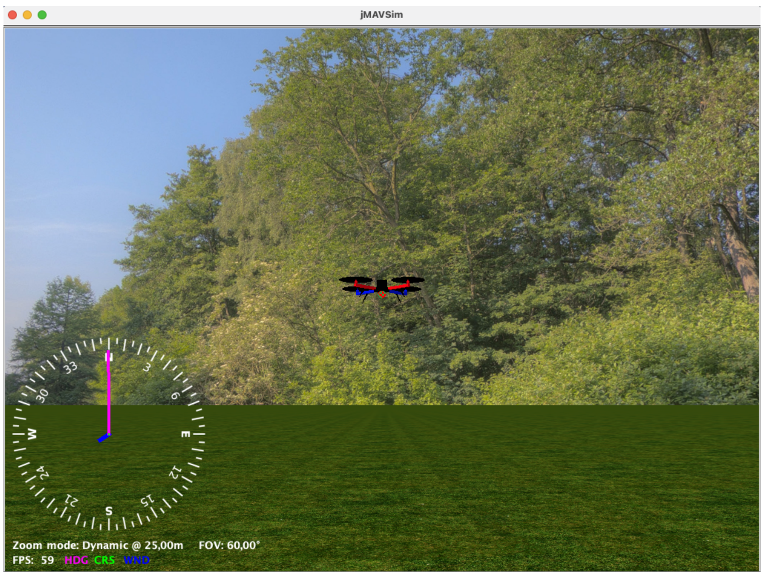


Figure 4.1: jMAVSim Simulator

4.1 Acquiring Telemetry information from MAVSDK Python

As mentioned before, MAVSDK is a library that simplifies MAVLink protocol messages. With this being said, MAVSDK Python was the mainly focused language of this project. Python was chosen because of the abundance of information online as well as the immense community support and ease of use. The Python version enables you to control your drone as well giving you it's telemetry data without any problems. The way it works it simple. When launching an MAVSDK code you have a "system()" class, in this class there are a various number of plugins which you can use to control the UAV as well as getting telemetry values, in this case from the "Telemetry" plugin. With this being said, various asynchronous functions were developed in order to get the drone telemetry and send the data into the database. This functions work separately and like just mentioned in an asynchronous form. This means that the functions can occur at the same time, in different times as well as at different rates, depending on the PX4 Autopilot messages sent back into the code developed.

The following list contains the functions directed to MAVSDK information gathering in this project:

- flight_time(start_time, drone)
- print_battery(start_time, drone)
- imu(drone)
- air_pressure(drone)

- ned_velocity(drone)
- rawgps(drone)
- fixedwingmetrics(drone)

Like the name suggests the "flight_time" function returns back to the server how long the drone is flying. This is done by saving in a variable the time when the drone started flying and subtracting that time with the current time. The "print_battery" function reads from the telemetry class the battery values, receiving the battery voltage and percentage remaining. The next function is called "imu", which stands for Inertial Measurement Unit (IMU) receives data like the outside temperature in Celsius and the front, right and down acceleration in meters per second squared and calculates the module of the three axis of acceleration. Subsequently the "air_pressure" functions collects the air pressure values in hectopascals. In "ned_velocity" the values of north, east and down velocity is received in meters per second, and their module is also calculated. In the "rawgps" it's collected the values of the drone Global Positioning System (GPS) velocity, as well as the drone altitude, latitude and longitude values. To finalize the "fixedwingmetrics" receives the data of the drone air speed, throttle percentage and the drone climb rate.

The following example shows one of the functions written, in this case shows the "fixedwingsmetrics" function:

```

async def fixedwingmetrics(drone):
    async for fixedwing_metrics in drone.telemetry.
        fixedwing_metrics():
        #print(f"Air Speed (m/s): {fixedwing_metrics.
            airspeed_m_s}")
        p = influxdb_client.Point("HUMS").field("Air Speed
            (m/s)", fixedwing_metrics.airspeed_m_s)
        write_api.write(bucket=bucket, org=org, record=p)

        #print(f"Throttle Percentage (%): {
            fixedwing_metrics.throttle_percentage}")
        throttle= fixedwing_metrics.throttle_percentage *
            100
        p = influxdb_client.Point("HUMS").field("Throttle
            (%)", throttle)
        write_api.write(bucket=bucket, org=org, record=p)

        #print(f"Climb Rate (m/s): {fixedwing_metrics.
            climb_rate_m_s}")
        p = influxdb_client.Point("HUMS").field("Climb Rate
             (m/s)", fixedwing_metrics.climb_rate_m_s)
        write_api.write(bucket=bucket, org=org, record=p)
```

Listing 4.1: "fixedwingmetrics" Function

All functions are similar. The functions are called in the asyncio python loop and when called they take as parameters the "drone" variable. In which the access to the telemetry plugin is granted and its different classes. In the example above the class to which was accessed was the "fixedwing_metrics()" class, giving, like mentioned prior, access to it's different values. After having the values they are sent into the database, how this is done will be discussed more in depth further along in this chapter.

Every function mentioned above sends the telemetry values to the database directly after getting them. More functions were made, but they are not being used because they were not needed, nevertheless if need be they are ready to be used.

4.2 Pymavlink - mavutil

Unfortunately not every information available from PX4 Autopilot can be collected by only using MAVSDK Python. In order to receive the rest of the information accessible we have to use the Pymavlink Libraries. The module used from these libraries was "mavutil". In a first instance, a connection to the drone as to be made and afterwards data can be extracted.

```

    # Start a mavutil connection
    connection = mavutil.mavlink_connection('udpin:
        localhost:14550', dialect="common")

    # Wait for the connection confirmation
    connection.wait_heartbeat()
    print("Heartbeat from system (system %u component %u)"
        % (connection.target_system, connection.
        target_component))
```

Listing 4.2: mavutil connection

The connection is made by declaring a User Datagram Protocol (UDP) port which will work as a communication channel between the the drone and the HUMS, two different UDP ports are used in this project, one for the MAVSDK libraries and other for the Pymavlink libraries. It's preferred that the dialect of the model is also indicated in order to reduce the possibility of mistakes like using the wrong dialect. Afterwards the system waits for a heartbeat in order to make sure the connection is established.

The next step is now request and receive information. In this system all mavutil information is gathered in on asynchronous function like the MAVSDK functions. This is done by asking for a certain type of message. In this case the messages requested were the ESC_STATUS, ESC_INFO and VFR_HUD messages. More messages can be requested, but to no answer, this happens because Gazebo and jMAVSim while working with PX4 Autopilot do not send every type of message possible, just a list of them.

```

async def mavlinkmessages(connection, drone):
    async for _ in drone.telemetry.position():
        msg = connection.recv_match(type=["ESC_STATUS", "
            ESC_INFO", "VFR_HUD"], blocking=False)

        msg_type = msg.get_type()
```

Listing 4.3: Resquesting Messages

The example above is how messages types were requested. The message type is saved in the msg_type variable and goes through a "if" cycle to check which of the three messages were received and the message is decoded. Once a certain type of message is received all of it's data is saved and sent into the data base. An example of how it works could be how the program reads the VFR_HUD message. In the first instance it compares if the message receive was the right one, and if so it collects the data like the UAV ground speed as well as were the drone is headed. Here is the VFR_HUD section the function presented before:

```

        elif msg_type == "VFR_HUD":
            groundspeed = msg.groundspeed
            #print(groundspeed)
            p = influxdb_client.Point("HUMS").field("
                groundspeed", groundspeed)
            write_api.write(bucket=bucket, org=org, record=
                p)

            heading = msg.heading
            #print(heading)
            p = influxdb_client.Point("HUMS").field("
                heading", heading)
            write_api.write(bucket=bucket, org=org, record=
                p)
```

Listing 4.4: VFR_HUD message decode

Like mentioned previously the same is done for the other two types of messages requested for this project. Each type of message contains different data. The ESC_SATUS message contains information about each of the in this case four ESC RPM values, as well as their respective voltage and current consumption. The ESC_INFO messages contains information like if the ESC is currently failing, how many times did each ESC fail and their respective temperature.

Using this method was how the rest of the important and available information was gathered. The code created can be adapted to other types of MAVLink Messages with ease once the desired messages were chosen. All of the MAVLink Messages can be found here: `https://mavlink.io/en/messages/common.html`.

4.3 Sending Everything Into InfluxDB

The next important step after having all the information needed is to save it all somewhere. Since the HUMS is supposed to work in a real-time basis a time-series data base was chosen.

After having all of the InfluxDB dependencies installed the data base client has to be created. Here is an example of how it can be done:

```

import influxdb_client
from influxdb_client import InfluxDBClient
from influxdb_client.client.write_api import SYNCHRONOUS
from pymavlink import mavutil

token="X"
org = "Y"
url = "http://localhost:8086"
write_client = InfluxDBClient(url=url, token=token, org=org
    )
bucket="Z"
write_api = write_client.write_api(write_options=
    SYNCHRONOUS)
```

Listing 4.5: InfluxDB Client Creation

To create and connect the client to InfluxDB the right Pyhton imports have to be done. It's also needed to have the database information. The information changes according to each user, each user can also have different types of Buckets in which there can have different pointers to here data is meant to be kept. In the example above, the token, the org and the bucket variables are modified due to security reasons.

After the client is created and the connection to the database is secured, the next step is actually send data into InfluxDB. Data is sent throughout all of the code. One example to dissect how data is sent can be found in the "print_battery" function:

```

async def print_battery(drone):
    async for battery in drone.telemetry.battery():
        batP= battery.remaining_percent*100
        #print(f"Battery (%): {batP}")
        p = influxdb_client.Point("HUMS").field("Battery
            (%)", batP)
        write_api.write(bucket=bucket, org=org, record=p)

        BatV= battery.voltage_v
        #print(f"Battery (V): {BatV}")
        p = influxdb_client.Point("HUMS").field("Battery (V
            )", BatV)
        write_api.write(bucket=bucket, org=org, record=p)
```

Listing 4.6: "print_battery" Function

To better explain how this is done let's consider the BatV variable. In this variable the battery voltage is saved. After the data is sent into the p variable in which specifies how the data should be sent into the database, first the pointer of the Bucket is chosen, afterwards is chosen the database variable name as well as it´s indicated which variable is going to be sent to InfluxDB. After that the "write_api" is called and the data is sent into the database. This as to be done to every piece of information that is intended to be sent into the database. Once this is done all information will start to appear in the InfluxDB user interface while the HUMS is running. When all data is sent into InfluxDB the data is finally ready to be sent into Grafana in order to develop a real-time dashboard.

4.4 Grafana Implementation

Grafana implementation is a straight forward one. Since Grafana has a plugin which supports InfluxDB 2.0 the data transfer is simple, otherwise would be more complicated and time consuming.

Assuming that Grafana is already configured to communicate with the database and assuming that the dashboard area of Grafana is already created, now is the moment to make Grafana receive the data. To make that happen is needed to know how to address every variable from InfluxDB. Here is an example of InfluxDB variable addressing:

```

from(bucket: "Z")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "HUMS")
  |> filter(fn: (r) => r["_field"] == "Air Pressure")
  |> aggregateWindow(every: v.windowPeriod, fn: mean,
     createEmpty: false)
  |> yield(name: "mean")
```

Listing 4.7: Addressing to Air Pressure in Flux language

This addressing is done in Flux, which is the InfluxDB querying language. Since all data is in the same bucket and in the same measurement pointer (HUMS) to refer to other variable the only thing that needs to be changed is the field variable. After knowing how to refer to every variable the next step is adding the variable into the dashboard and choosing how to present the variable. There are a lot of possibility's to present data. Data can be presented in gauges, tables, candlesticks, bar charts and much more visualization options. At the end the dashboard created for this project has the three following panels:

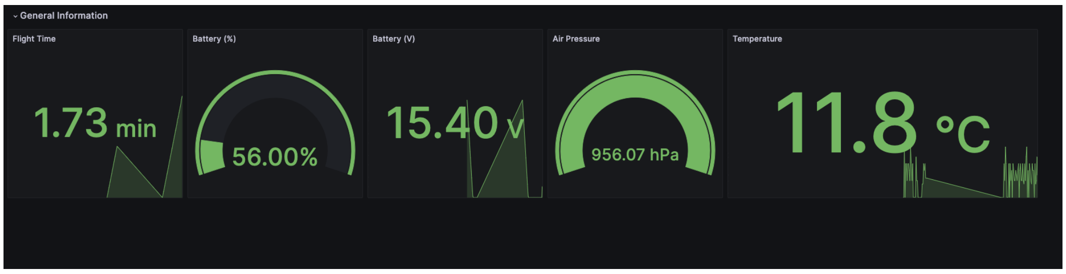


Figure 4.2: General Information Row

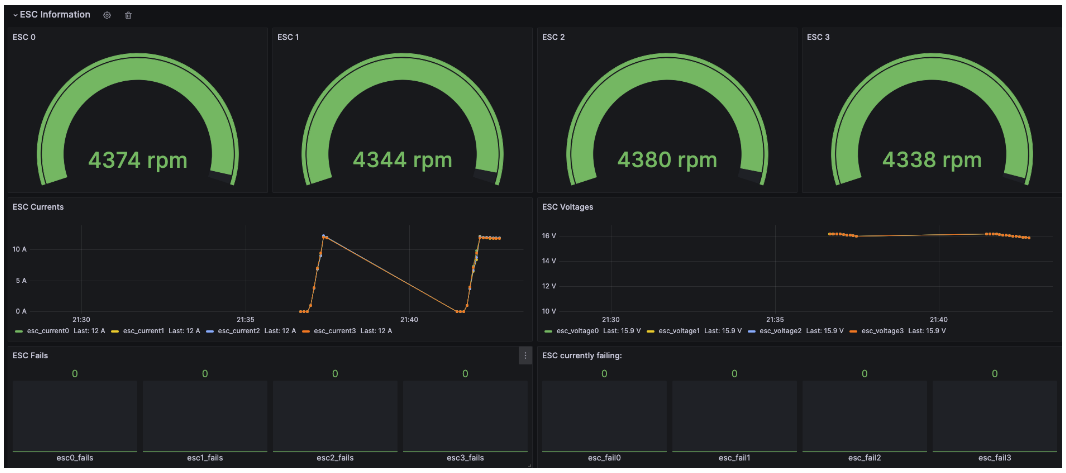


Figure 4.3: ESC Information Row

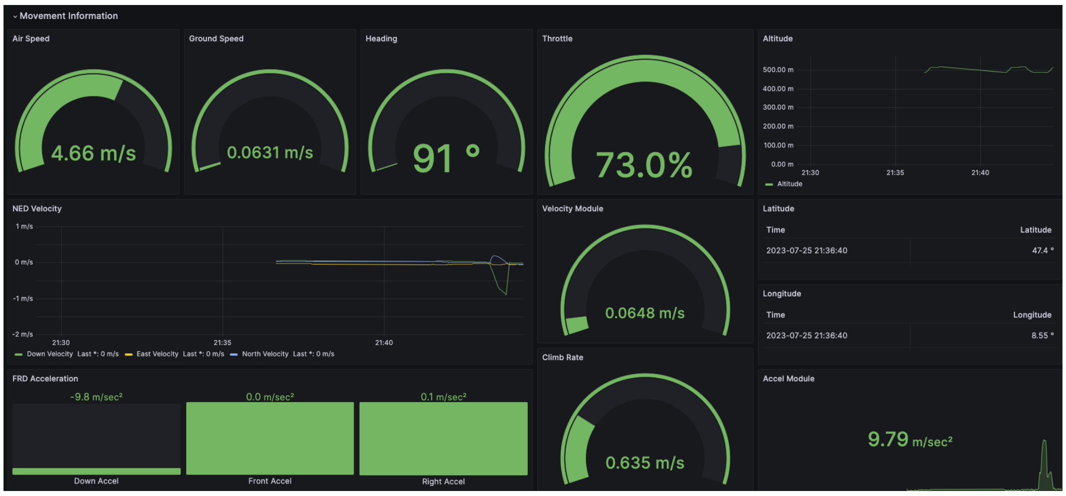


Figure 4.4: Movement Information Row

Chapter 5

Conclusions

To conclude, this project provided insights into the use of different programming languages as well as insights in the use of different frameworks.

The project lived up to the desired expectations, creating an easy to adapt and easy to understand Health and Usage Monitoring System (HUMS), ready to be used in real life. One being done in C++ and the other being developed in Python with the help of programs such as InfluxDB and Grafana.

The C++ version being more useful when the data is not needed in real-time, and the Python version being more intended to real-time use and drone monitoring. Both versions can be made better by optimizing the code as well as by adding new features such as, for example, making the drone land in a safe position if the battery level gets too low.

Nevertheless this project was a success and can be used as a base layer to the development of a greater and more powerful Health and Usage Monitoring System (HUMS).

All code as well as all information about this project will be available here: `https://github.com/ricardocraveir5/HUMS`

References

[1] S. Baig, “History series: The beginning of unmanned vehicles.” Available at https://www.vertiq.co/blog/history-series-the-beginning-of-unmanned-vehicles, Aug. 2022. (Último acesso em 04/09/2023). [Cited on page 5]

[2] IWM, “A brief history of drones.” Available at https://www.iwm.org.uk/history/a-brief-history-of-drones. (Último acesso em 04/09/2023). [Cited on page 5]

[3] FLY.PT. Available at http://flypt.pt/. (Último acesso em 04/09/2023). [Cited on page 6]

[4] P. Autopilot. Available at https://px4.io/. (Último acesso em 04/09/2023). [Cited on page 7]

[5] P. Autopilot, “Software overview.” Available at https://px4.io/software/software-overview/. (Último acesso em 04/09/2023). [Cited on page 7]

[6] P. Autopilot, “Gazebo simulation.” Available at https://docs.px4.io/v1.12/en/simulation/gazebo.html, June 2021. (Último acesso em 04/09/2023). [Cited on page 7]

[7] P. Autopilot, “jmavsim with sitl.” Available at https://docs.px4.io/v1.12/en/simulation/jmavsim.html, June 2021. (Último acesso em 04/09/2023). [Cited on page 7]

[8] Dronecode, “Mavsdk(main).” Available at https://mavsdk.mavlink.io/main/en/index.html, Mar. 2023. (Último acesso em 04/09/2023). [Cited on page 7]

[9] MAVLink, “Using pymavlink libraries (mavgen).” Available at https://mavlink.io/en/mavgen_python/. (Último acesso em 04/09/2023). [Cited on page 8]

[10] InfluxData, “Get started with flux and influxdb.” Available at https://docs.influxdata.com/influxdb/v2.7/query-data/get-started/. (Último acesso em 04/09/2023). [Cited on page 8]