DRONE DETECTION AND INHIBITION

A Master's Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Erick Medina Moreno

In partial fulfilment

of the requirements for the degree of

MASTER IN TELECOMMUNICATIONS ENGINEERING

Advisor: Josep Paradells

Barcelona, February 2020

Title of the thesis: **Drone Detection and Inhibition**

Author: **Erick Medina Moreno**

Advisor: **Josep Paradells**

Abstract

In the recent years, there has been an increase in the demand and use of unmanned aerial vehicles (UAV) know as drones. Even though there are restrictions for its use by licensed drivers and in determined areas, it has been notorious that these devices are used carelessly in open areas putting in risk the integrity of people. In order to tackle these issues, we propose the creation of a prototype of a compound device formed by a Software Defined Radio (SDR) named HackRF One and a Raspberry Pi, that can help detect drones and interfere its communications with its remote controller.

We have built a software tool in Python and using GNURadio as the telecommunications framework, that can be executed in a Raspberry Pi using Ubuntu Server. It consists of five different scripts with its respective user interface that obtain the power values of frequencies between 1MHz and 6 GHz and help detect unusual signals in the spectrum, than can visually be analyzed and related to drones, besides a script that generates noise signals to interfere the communications.

It was demonstrated that despite the computation limits of a Raspberry Pi, the scripts can indeed be executed and obtain the power values of the spectrum between 1MHz and 6 GHz. and generate interfering signals that can affect the communications of a drone.

Dedicated

To the patience and support of my wife and kids.

To my parents and close family.

Acknowledgments

To Ecuador's “Secretaría de Educación Superior, Ciencia y Tecnología” (Senescyt) as the scolarship's sponsor.

To my father for his guidance during the course of this work.

To all teachers and friends who have contributed to my knowledge during my studies at UPC.

Revision history and approval record

Revision	Date	Purpose
0	09/12/2019	Document creation
1	07/05/2020	Document revision
2	15/05/2020	Document revision

Written by:		Reviewed and approved by:	
Date	09/12/2019	Date	18/05/2020
Name	Erick Medina	Name	Josep Paradells
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

List of Tables

List of Code Snippets

1. Introduction

The proliferation in the demand and use of unmanned aerial vehicles (UAV) otherwise know as drones, due to its affordable cost and its multimedia capabilities have posed a threat to security, since they can be used by unlicensed drivers and in areas that are not authorized for its flight, such as natural protected areas or touristic points of interest.

To tackle this problem, we must follow two lines of work in order to take actions to countermeasure the drone presence. The first one is the drone detection, that can be handled with different alternatives such as sound recognition, image recognition, radar detection and radio detection. The other one is the drone control or take-down which can include radio interference, radio control, laser guns attacks, or physical catch.

In the scope of this project we will be using both radio detection and radio interference as the means to detect and take-down the drone communications.

To accomplish this task we will use a Software Defined Radio (SDR) peripheral called HackRF One, and the GNU Radio software development kit. The programming language Python will be used along with GNU Radio in order to create custom components according to the needs of this project.

1.1. Unmanned Aerial Vehicles (UAV) or Drones

Unmanned aerial vehicles as it names states are vehicles that can fly without the need of a human pilot on board. Commercially they are also know as drones. They can be used for different purposes going from security and military, up to multimedia acquisition, and their applications keep expanding.

Certainly in the last decade, it has been their capacity to obtain aerial photographs and videos, the main cause of their soaring popularity. And since the materials and electronic components to build commercial drones have plummeted, their acquisition price has made it become an affordable gadget to a broader sector of the population, not only to aeronautics enthusiasts.

Drones can be found freely available to buy on the internet and physically in retailers and toy stores. Even though in some countries like Spain, you need a license to operate drones, it is not a requirement to buy them, making it possible that unqualified people have access to controlling these types of aircraft.

Aeronautical regulations in Spain state the requirement of a license to pilot remotely a drone professionally, they also indicate that in areas with people and buildings it is not allowed to fly a drone unless authorized explicitly.

Even regulation exist, there have been a few security-incidents related to drone activity. The most recent, as of the date of writing of this document, was reported in Madrid in February 2020, with the shutdown of the aerial space surrounding the Barajas airport for more than two hours affecting 26 flights [1]. All of this due to the unauthorized presence of drones near the airport, that were detected visually by pilots.

Another recent cases occurred in Barcelona between October and December 2019, with a failed display of banners carried by drones inside the Camp Nou Stadium in a Barcelona-Real Madrid match [2], and the detection of around 83 unauthorized drones in the streets of Barcelona during independence related protests [3].

Indeed drones, pose a security threat for people and governments if they are not effectively controlled. It is crucial to understand how drones work in order to choose the right strategies to detect and disable them safely.

1.2. Drone anatomy

Drones are composed of both hardware and software parts within its body. The hardware components are in charge of providing the aircraft the ability to takeoff, land, fly,maneuver, communicate and sense. Software is in charge of controlling the aircraft.

Propellers are in charge of generating torque and thrust for all movement related actions. Depending on the drone characteristic the manufacturer can choose the quantity, number of blades, length and material of propellers, but one of the most widely known design is the four propeller drone in the form of a quadcopter.

Figure 1.1: Image of the quadcopter drone DJI Mavic Pro 2

Motors are the components in charge of providing the torque and thrust to the propellers. Most common models are the brush-less motors, and the most important characteristics should be chosen according to the drone's total weight, and they are efficiency, revolutions per minute, and thrust.

The electronic speed controller (ESC) is in charge of providing the voltage to the motors, and change the drone's speed and direction according to the voltage provided, among other features. This component is fundamental for radio controlled drones.

Battery is in charge of providing the energy to the drone and mainly to the motors. Although its characteristics depend on the drone's features, one of the most commonly used is the Lithium Polymer (LiPo) rechargeable battery.

Sensors included within a drone can vary according to its features, but most frequently we can find distance sensors and orientation sensors. They are in charge of feeding the flight controller with information such as accelerations, movement changes, distance to closest objects. If the drone has radio capabilities, here we will find the radio sensors that allow communication with a remote controller or with other telecommunications providers.

The flight controller is in charge of controlling all the drone's functions. Here all the sensor's information is processed and taken into account to make flight decisions. If the drone is configured to work in standalone mode, all the flight instructions are set here. If it is configured in a remote controlled mode, all the information received from the radio controller is processed and passed into the motors in order to execute the user's maneuvers.

The software of a drone is embedded in its flight controller. All decisions made by it are programmed in software, and take into account the data from the different sensors. The language used depends on the flight controller, but the most used are Python, C, C++ or C#.

The remote controller is an external component of the drone. It allows to maneuver the drone. It provides buttons and rods that change the direction, angle and altitude. It has an antenna that emits the control data to the drone. Depending on the complexity of the drone it can also show telemetry data from the drone and multimedia [4].

The previously described components are the most common found in commercial drones. But as it has been stated before, depending on the purpose of the drone, more complex components can be found in it.

There are drones equipped with water hoses that can help extinguish fires [5] and companies such as Aerones and Walkera are working closely with firefighters department to make them a reality.

Farming drones are equipped with heat camera sensor, to control the cattle, but these have been used successfully in rescue scenarios. In October 2019 in Minnesota, United States of America, a farming drone helped find a missing child in a forest with the use of the heat camera [6].

1.3. Drone footprints and detection

Since the components of drones have been already explained in the previous chapter, we can start discussing about the traces or footprints that they leave while in use. We could classify them in visual, audible, and radio footprints.

Visually a drone can be identified when in the air by its shape, which is in most cases a quadcopter. It can become more complicated if a drone has airplane-like structure or more complex structures as in military uses. With a photo or video camera it is possible to develop drone image recognition algorithms. Also with the use of radars the presence of a drone can be sensed [7].

When the drone is flying, the motors and propellers moving the air produce a characteristic noise that leaves an audible footprint. The intensity of the noise can vary from model to model. With the use of microphones it can be possible to develop drone recognition algorithms [8].

If the drone uses radio communications with a remote controller while it is flying, then it leaves a radio frequency footprint with all the data transmission between them. With the use of radio receivers it is possible to sense the presence of drone communications.

It can be discussed with a broader perspective the pros and cons of each one of the four drone detection possibilities mentioned in the above paragraphs. Nevertheless, we will focus on the radio-frequency detection approach since we count with equipment capable of detecting and generating RF signals.

1.4. Drone communications

For the scope of this work, we will focus on drones that have the capability to communicate with a remote controller, and will use the radio-frequency footprint detection approach. And for this purpose it is fundamental to understand how communications work in a drone.

There are at least four motivations to perform communications operations by the drone, and not all necessarily are addressed to a remote controller: command and control, telemetry, multimedia and orientation/navigation.

Command and control, and telemetry refer to all communications between the remote controller and the drone to send flight orders and receive back data about the flight conditions, sensors, and altitude among others. The multimedia refers to the images, video or audio obtained by the drone and sent back to the user in real time. And navigation refers to the communications with satellites to obtain the drone's position for user controlled or autonomous flights.

Each one of the communications can be maintained at different frequency bands and the transmission system varies according to the manufacturer and drone model.

Table 1.Drone communications technology [9]

COMMUNICATION TYPE	FREQUENCY BAND	TRANSMISSION SYSTEM
Command and control, Telemetry	27 MHz 35 MHz 40 MHz 72 MHz 433 MHz 868 MHz 2.4 GHz 5.8 GHz	DSM2 (Spektrum) [10] DSMX (Spektrum) [11] ACCESS (FrSky) [12] FASST (Futaba) [13] S-FHHS (Futaba) [13] T-FHHS (Futaba) [13] OcuSync (DJI) [14] Lightbridge (DJI) [14]
Multimedia	328-334 MHz 433 MHz 2.4 GHz 5.8 GHz	-
Navigation	L1: 1.563 – 1.587 GHz L2: 1.215 – 1.2396 GHz L5: 1.164 – 1.189 GHz G1: 1.593 – 1.592 GHz G2: 1.237 – 1.254 GHz G3: 1.189 – 1.214 GHz E1: 1.559 – 1.592 GHz E5a/b: 1.164 – 1.215 GHz E6: 1.260 – 1.300 GHz	-

As we can see in the above table, drones communications are established in different parts of the spectrum from 27 MHz up to 5.8 GHz with several distinct transmission systems.

1.5. Solution Adopted

The objective of this project in general terms is to build a software tool that can help detect drones and jam its communications and navigation capabilities using a HackRF One, and GNURadio with Python. The built software must be able to be executed in a Raspberry Pi.

The approach to perform the detection is by measuring the power levels across the spectrum from 1MHz to 6 GHz, and establishing a base level as our reference. We choose ~~this~~ limits both because of the HackRF range, and because as we saw in Table 1, UAVs generate and receive signals from frequencies as low as 27 MHz~~,~~ up to 5.8 GHz.

Once the previous step has been completed, when we perform a real time scanning we could compare the values with the base previously obtained, so we can detect unusual

activity and determine the frequencies were this is happening. We can give the user a graphical tool to analyze the existing signals with its respective power level in real time, and determine visually the presence of drone activity.

For jamming the ~~drone~~ communication, we give the user the option to generate a noise signal that can make interference or interrupt definitively the communications between the drone and its remote controller.

This solution is intended to be the base ~~for~~ a more robust drone detection and inhibition system, in which DSP techniques can be applied, and automatic detection techniques based in ML or AI can be implemented.

2. State of the art

2.1. Drone Detection and jamming

Drone detection and jamming has become in the recent years a field of interest due to several security problems caused by civil UAVs as we explained in the introduction. This has led to a scenario in which police forces must acquire drone take-down equipment to face this menace, and where cities with its airports must be equipped with drone detection systems to prevent UAV activity in restricted air space.

Spain's Police has posted in social networks that they have counter-drones guns, specifically the UAV-D04JAI by China's Hikvision, which jams the communication between a drone and its remote controller for a distance of up to 1 km, with a power of up to 40 dBm , and with GNSS satellites with a power of up to 33 dBm. According to its data sheet, the equipment can jam communications in 1.5 GHz, 2.4 GHz and 5.8 GHz [15].

Figure 2.1: Spain's Police officer with a drone jammer [16]

In Madrid, the Police has integrated to its emergency services a system called Caelus, which is in charge of detecting drones in the city's aerospace in heights between 250 to 400 meters. The solution counts with antennas distributed across the city and an artificial intelligence system in charge of identifying the drone's flight information, and even estimate the location of the remote controller [17]. The solution is provided by Barcelona's ASDT System Europe.

Airports in Paris and London have already implemented solutions to detect and track drones using holographic radars, which gets 3D position of the UAV and its movements. The radar has a range of up to 5 km. This solution is provided by UK's Aveillant and is commercialized under the name of Gamekeeper 16U [18].

A more complete solution is offered by USA's Raytheon, which offers an electro-optical/infrared sensor, to detect, identify and track rogue drones. And once identified it activates a high-energy laser weapon system capable of taking the drone down in a matter of seconds [19]. This solution has gained interest of the US government and is now being tested by its military forces, and is expected to be used against civil and military UAVs.

There are plenty of academic publications both addressing the detection and jamming of UAVs. Focusing in the RF detection, we use a passive approach in which we eavesdrop the communications in specific frequency bands and according to the signal's characteristics determine the presence of a drone [20]. One way to detect the presence of drones is by analyzing its RF characteristics such as bandwidth, modulation, frequency hopping mechanism and decoding its information, but in practice with a vast quantity of protocols and different bandwidths, doesn't seem ~~feasible~~ without knowledge of all these RF characteristics for all drones to be detected [21].

Even though having information on all available drones seems like an unfeasible task, there is an impulse to create a large dataset with RF data from widely used drone models as the Bebop, AR and Phantom. This initiative is called the Drone-RF dataset [22], and it fosters the use of this dataset to detect and identify drones with the use of deep neural networks or machine learning algorithms [23].

For jamming the communication of the UAV with the remote controller, the approach is to generate a noise signal that can be transmitted depending on the communication mechanism used by the drone. If the drone is using the WiFi operation mode, then we can transmit the noise in a specific WiFi channel [24]; or if it using a different mode with frequency hopping, transmit the noise along the whole 2.4 GHz band [25].

A different approach for this issue is the hijacking of the drone by exploiting protocol security issues. A security researcher exploited failures in the DSMX [11] protocol and with a little device called Icarus, has proven to hijack drones by injecting a malicious control packet in the drone communications, replacing the the control information source and leaving the original remote controller unable to control the aircraft [26].

3. Methodology / project development

This project's outcome is a software that helps in the detection and jamming of drone's communications. This will result in the creation of five different Python scripts. The scripts will give the user the possibility to scan the spectrum from 1 MHz to 6 GHz and establish reference power levels. Also the scripts will detect unusual signal activity comparing real time power values against the reference values, giving also graphic tools to see the signals in a determined bandwidth. One script will allow the user to generate a jamming signal that can interfere or take down drone communications. The last and final script is the controller script which launches the other scripts and gives the user the chance to browse graphically and statistically through the obtained data.

For the development of the project we will use the following components:

Hardware used for implementation

- HackRF One
- Raspberry Pi 3B+
- Laptop Dell Inspiron

Hardware used for testing

- Toy Car remote controller
- WhyEvo garage remote controller
- DJI Mavic Pro drone and its remote controller

Software used for implementation

- GNURadio framework and GNURadio Companion IDE.
- Python
- QT

Software used for testing

- Htop

3.1. Hardware used for implementation

This section includes the hardware necessary to design, implement and run our project, such as the computers that host the program and the peripheral SDR equipment to receive and transmit RF signals.

3.1.1. HackRF One

HackRF One is a software defined radio (SDR) open source hardware used as an USB peripheral, created by the company Great Scott Gadgets. It is capable of receiving and transmitting at frequencies from 1MHz to 6 GHz, in a half-duplex configuration. It can be used as a OSMOCOM [27] source in GNURadio, and is also compatible with SDR#.

It offers three different sample rates in the order of the millions: 2 Msps, 10 Msps and 20 Msps. The sample's resolution is 8-bit (8-bit I and 8-bit Q) [28].

The transmission and reception gains, the baseband filter, and the antena port power can be configurable by software. It provides three different analog gain controls on RX and two on TX. The three RX gain controls are at the RF ("amp", 0 or 14 dB), IF ("lna", 0 to 40 dB in 8 dB steps), and baseband ("vga", 0 to 62 dB in 2 dB steps) stages. The two TX gain controls are at the RF (0 or 14 dB) and IF (0 to 47 dB in 1 dB steps) stages

The transmission power varies according to the frequency, but offers the best performance around 2.4 GHz. All of its values are reflected in the following table.

Table 2.HackRF One transmission power [29]

FREQUENCY RANGE	MAXIMUM TX POWER RANGE
10 MHz – 2150 MHz	5 dBm – 15 dBm
2150 MHz – 2750 MHz	13 dBm – 15 dBm
2750 MHz – 4000 MHz	0 dBm – 5 dBm
4000 MHz – 6000 MHz	-10 dBm – 0 dBm

Although the manufacturer provides us with these maximum transmission power values, they can't be set to a specific value in the device and we expect ~~the the~~ output power to be in the ranges specified.

The manufacturer indicates the the maximum reception power allowed is -5 dBm, and powers above this value can result in the equipment's damage. Its sensitivity its not specified, as it depends on several factors [29] [30].

It can be used as a standalone device with its own functions scanning up to 8 GHz per second, and offers also libraries and internal APIs for a more customized control of the device [31]. However these implementations will only work for this device only and will not be compatible with other SDR devices~~.~~

The device doesn't provide certifications that indicate that the transmissions comply with government regulations, so the user is the sole responsible of using it adequately.

HackRF One is a quadrature sampling system, which means that all samples are IQ samples. As a consequence we can find that in the graphics we will find a DC offset spike right in the center of the received spectrum [30]. Since it is common to quadrature sampling systems, we will ignore it for this project's prototype. As we are working with complex sampling, the sample rate (Msps) indicates the bandwidth (MHz).

The HackRF can be acquired for a price range of 183€-293€.

We have chosen the HackRF One as our SDR equipment, due to its wide range of operation from 1 MHz to 6 GHz, which includes our frequencies of interest reflected in Table 2; its half-duplex capability, which allows us to both receive and transmit, even though not simultaneously; and also because of its affordable price.

Figure 3.1: HackRF One SDR peripheral case.

3.1.2. Raspberry Pi 3B+

The Raspberry Pi is a low cost, small sized computer that plugs into a monitor or TV, and uses a standard keyboard and mouse to operate. The 3B+ model has a 1.4 GHz 64bit quadcore processor, with 1 GB SD RAM memory. For networking purpose it has a Ethernet port , and also a dual band wireless card. It also offers support for Bluetooth 4.2 and BLE [32].

For this project we will use it with a Linux distribution, specifically the Ubuntu Server for Raspberry 3, together with the Mate Lite desktop. It will be configured with the official 7" Touch Screen, and for the initial setup it will count with a wireless keyboard and mouse.

The Raspberry will be configured with support for Python, GNURadio and OSMOCOM so that our scripts can be executed. Its setup steps can be found in the Annex I.

The Raspberry Pi 3B+ can be acquired for a price of 38€, and with additional components as charger, case and SD card for 74€. The touch screen can be found for a price of 70€.

We have chosen to execute this project in a Raspberry, to prove that a spectrum analyzer and jammer custom solution can be implemented in a low-processing-power computer, opening the door to implement this kind of projects in portable hand-sized devices.

Figure 3.2: Raspberry Pi model 3B + with 7" touch screen.

3.1.3. Laptop Dell Inspiron

This laptop will be used for the development of the whole project and will be used for tests too, before migrating the scripts to the Raspberry.

The laptop has an Intel Core i3 1.90GHz 64bit quadcore processor with 8 GB RAM memory, running Linux Ubuntu 18.04.4 LTS.

3.2. Hardware used for testing

This section includes the hardware used to validate that our project can detect RF signals across the 1 MHz to 6 GHz spectrum, and that it can jam communications within the same frequency bands.

3.2.1. Toy Car Remote Controller

It is a remote controller of a toy car that operates at 27 MHz, and we'll use it as a generator to verify that our project works for low frequencies, since some drones use this frequency for operation.

Figure 3.3: Toy car remote controller (27 MHz)

3.2.2. Why EVO garage remote controller

It is a remote controller that can work in frequencies from 300 MHz to 868 MHz. We'll use it in this project to generate signals at 433 MHz, and verify that our project works correctly with this frequencies.

Figure 3.4: Why EVO garage remote controller (433 MHz)

3.2.3. DJI Mavic Pro (M1P)

Mavic Pro is a high end drone of the DJI brand released in late 2016, which offers 4K video capabilities and autonomy of up to 27 minutes. It comes with the GL200A remote controller. As for the communication, it offers transmission over WiFi with a maximum distance of 80 mt. and maximum height of 50 mt., and with remote controller (RC) it offers maximum distance of 4 km. and maximum height of 120 mt. for Europe. For this region the drone is authorized to use the 2.4 to 2.4835 GHz band with a power up to 20 dBm, and the 5.725 to 5.850 GHz band with a power of up to 13 dBm. The remote controller is only authorized for communications in the 2.4 to 2.483 GHz band with a power up to 20 dBm. It doesn't have a built-in GPS so it doesn't offer autonomous flight mode [33].

Figure 3.5: DJI Mavic Pro (M1P) drone and its GL200A remote controller with a mobile smartphone used for first-person view.

Table 3.Operation modes of the DJI Mavic Pro

OPERATION MODE	BAND	TX POWER	COMMUNICATION
WiFi	2.4-2.4385 GHz 5.725-5.850 GHz	<= 20 dBm <= 13 dBm	Drone to smartphone
RC	2.4-2.438 GHz	<= 20 dBm	Drone to remote controller

When it works in WiFi mode, the drone acts as an Access Point (AP) and creates its own WiFi network. A mobile device with WiFi capabilities must connect to the drone's network and use the DJI Go 4 mobile application in order to send and receive data to and from the drone. There are led indicators in the drone that indicate with a green light that the connection is established and with a yellow that it has no connection established.

In the mobile app we have options to see the activity in the WiFi channels and can see the channel in which the UAV has established the communication. We can see that it offers communication in the 2.4 GHz and 5 GHz band, in fixed band or dual mode.

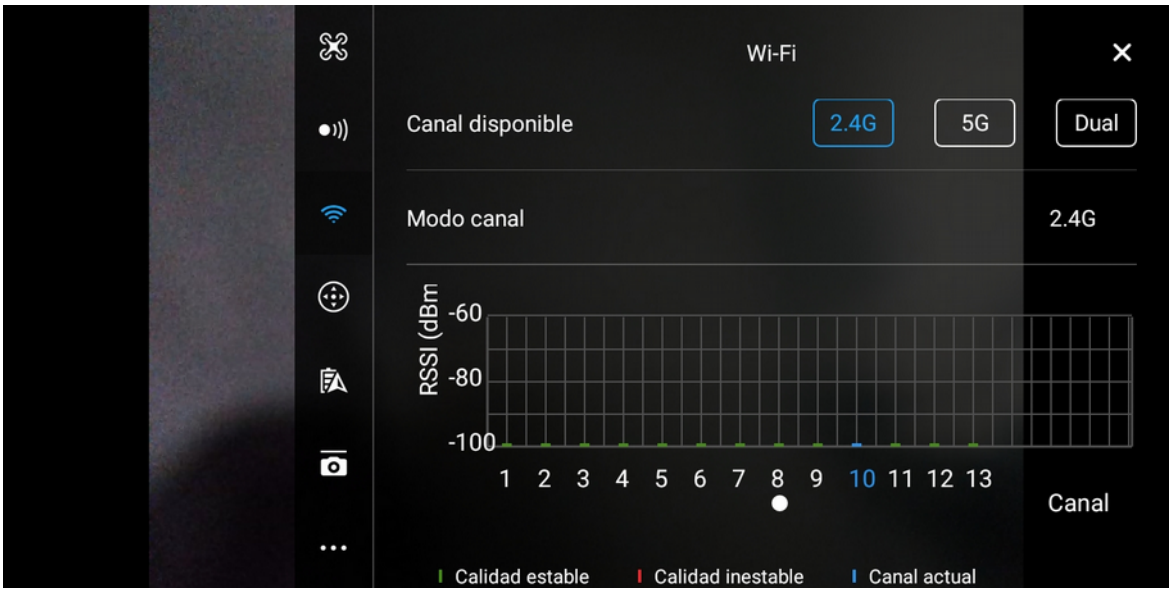


Figure 3.6: Screenshot of the DJI Go 4 app in the WiFi communications option. Communication is established in the 2.4 GHz band in channel 10.

When it works in the RC mode the drone establishes connection with its remote controller using the 2.4 GHz band and the OcuSync protocol [14]. The controller must be connected via USB to a mobile smartphone with the DJI Go 4 app, in order to establish the

communication with the aircraft. Performing an FCC ID search for the remote controller [34], we can find the RF test report in which we can find information on the operation modes and the channels with its respective bandwidth and frequencies. From this report we can obtain the following table about its RC operation mode.

Table 4.RC operation modes of the DJI Mavic Pro

BANDWIDTH	NUMBER OF CHANNELS	FREQUENCY RANGE	SPAN BETWEEN CHANNELS
1.4 MHz	38	2403.5 – 2477.5 MHz	2 MHz
10 MHz	73	2405.5 – 2477.5 MHz	1 MHz
20 MHz	63	2410.5- 2472.5 MHz	1 MHz

In the mobile app we have options to see the 2.4 GHz band RF activity in real time, and we find there are two operation modes: automatic and fixed. The automatic mode selects the channel which is cleaner to perform the transmission, and selects by default the 20 MHz ~~band~~ and the right end of the 2.4 GHz band. The fixed mode lets the user set the bandwidth and center frequency among the 174 channel options available. For the connection we established with our drone it allows only to configure transmission in 10 MHz and 20 MHz modes, so we can choose among 136 different channels for our communication.

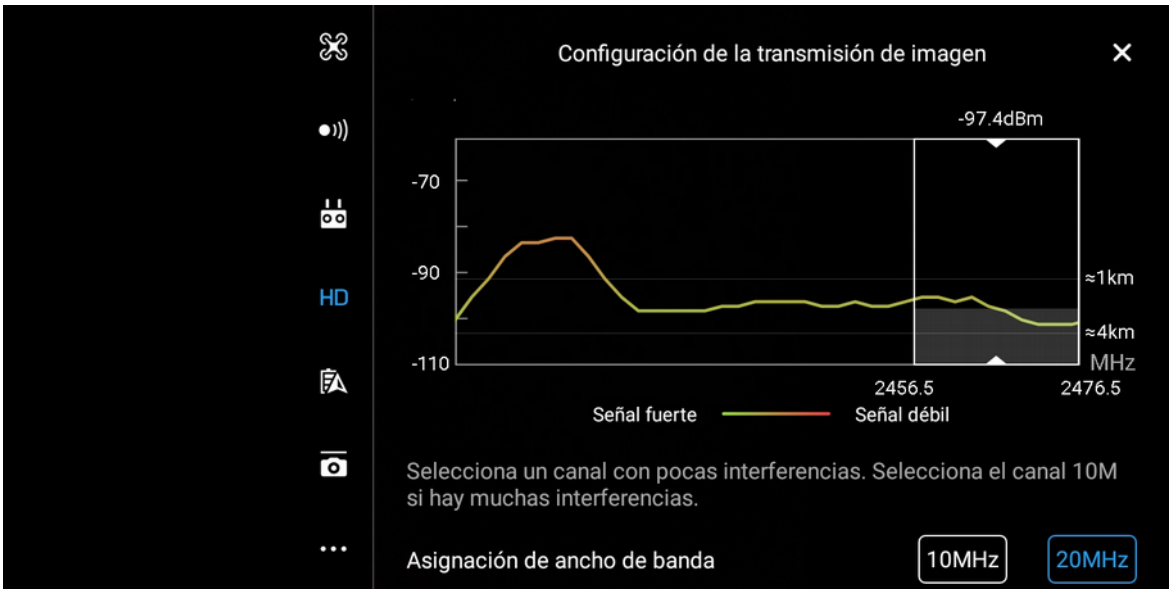


Figure 3.7: Screenshot of the DJI Go 4 app in the RC communications option. Communication is established in in 2466.6 MHz with 20 MHz bandwidth.

The OcuSync ~~protocol~~ is a propietary ~~protocol~~ owned by DJI, and its detailed information is not public so we don't know how does this drone performs the frequency hopping in case of bad channel conditions. As we will see in the jamming tests performed, the UAV's communication change its transmission's center frequency rapidly when the channel's conditions worsen, and it's a technique used by drones to sort interferences, jamming, or eavesdropping. Some UAVs use frequency hopping as their default transmission mode to handle all communications just like Futaba's drones using FHHS ~~base~~ protocols [13]; and some do it when they find channels with interferences as the Mavic Pro does.

3.3. Software used for implementation

This section includes the software necessary to develop the source code of our project. We'll use the GNURadio SDK together with Python as the programming language and QT as the framework for the creation of the user interface and user controls that will allow the user to choose among the different parameters available.

3.3.1. GNURadio

GNU Radio is a open-source software development toolkit (SDK) that provides signal processing blocks to implement software radios. It can be used with readily-available low-cost external RF hardware to create software-defined radios, or without hardware in a simulation-like environment [35].

It comes with GNURadio Companion, an integrated development environment (IDE) which provides a ~~a~~ graphical interface to create diagrams connecting the GNURadio blocks. It comes with readily available blocks as sources, sinks, filters, modulators, among others. It also gives support for creating custom blocks.

Files are created under the *grc* extension, so that they can be edited, but it also generates a Python script which is the final executable file. These scripts have the GNURadio components but also user interface components under the QT framework.

One limitation we found in the GNURadio Companion, is that the interaction is fully dependant on the user, and that automatic recurrent tasks can't be created from the IDE. So, actions like changing the center frequency or the sample rate have to be performed by the user manually. But this major limitation can be overcome by manually modifying the Python script, and adding these features as we will explain in later sections.

GNURadio has been used as a tool for research and academic purposes, since it can simulate telecommunication systems with its available blocks, but also has evolved into a tool for commercial use, since more compatible SDR equipment is used for production-ready telecommunications projects. Among the most known solutions supporting GNURadio we can find the Ettus Research USRPs, LimeSDR, BladeRF, rtl-sdr TV tuners and our Great Scott Gadget's HackRF One [36]. Among these devices we can find affordable hobbyist equipment for prices around the 20€ range, up to high-end high-bandwidth equipment for prices above 4.000€.

In most cases, solutions built with this SDK can be used with all sort of GNURadio compatible devices, respecting their hardware limitations, giving us a flexibility to replace the SDR equipment without the need to adapt our project's source code.

In order to install GNURadio in Linux Ubuntu, we must follow the instructions found in Annex II.

3.4. Software used for testing

The software created in this project can be tested by executing the scripts and verifying their respective outcomes. However we want to also keep track of the CPU performance of the computer executing the scripts, and for this purpose we'll use *htop*.

3.4.1. htop

Htop is a tool that allows us to see the running processes in an Unix system [37]. All information is displayed as text and we can see the current CPU usage of all the cores that the computer has, the RAM memory usage and all the processes running and their respective CPU and memory usage.

3.5. The outcome

Even though the GNURadio framework can be installed in MacOS and Windows, it was chosen to work in a Linux computer, due to an easier installation process and more stability. In a MacOS environment, an update of the OS or a specific program such as XCode can make the framework unusable.

The process consists of making the project in the laptop computer and then port it to the Raspberry Pi, being fundamental that both devices have the same versions of Python and GNURadio installed for a correct compatibility.

For the project we have programmed five executable Python scripts initially created in GNURadio Companion but later modified as wished according to our real needs. Additionally we have created three custom GNURadio components under the *gr-tfm* package which will be in charge of processing data and create a file database.

The main outcome of this project is a computer software that helps in the detection of drone RF activity and the inhibition of its signals. Python is the chosen programming language to develop this task, because of the compatibility with both GNURadio and QT frameworks. ~~This project is intended to be the starting point of a portable drone detector, and we will focus our work on setting the basis as a functional software that can acquire meaningful data, and that can adapt to the SDR hardware characteristics and to the user's needs, giving several controls to customize the execution of the program. At the same time, leave the door open for new add-ons related to DSP or tools like ML.~~

The main program is one script containing four buttons among other options, that open other four scripts. The base script is in charge of obtaining the base values of the spectrum power without drone activity. ~~s~~pectrum and band scan scripts are in charge of scanning in real time the spectrum power and compare with the base values. And the jamming script is in charge of generating jamming signals.

The scripts use GNURadio blocks and QT user interface components. Among the GNURadio components we have three custom ones. One is in charge of converting the data from the HackRF source into ~~power~~ with a Hamming window. Another is the *power_analyzer*, which is in charge of creating the base powers file database with a specific format. The last one is the *power_comparator*, which is in charge of comparing the real time power values with the base and create another file database with the comparison values.

We will analyze in detail all the scripts and custom blocks in the next sections, giving us an insight on how things work behind the user interface.

To setup the custom block we must follow the steps indicated in the Annex III, and for the execution of the scripts we must follow the Annex IV. The source code for the *xml* and Python files ~~can be~~ found in the Annex. GNURadio Companion files with *grc* extensions have not been annexed because they contain more than 2000 lines of code, which makes them impractical to add to this document.

We have added a User's Manual in the Annex V in which we explain all actions that a user can take with our software.

3.5.1. Custom Block: logpowerfft_hamming

This block is in charge of converting the stream of IQ data coming from the HackRF source into a power vector with a specific length, after applying the FFT with a Hamming window. GNURadio has a default block called Log Power FFT (Figure 3.8) which does the same operations but with a Blackman-Harris window and it can't be changed (Figure 3.9). We prefer a Hamming window because it reduces the side-lobs compared to the main lobe, making it an ideal option for frequency selective analysis [38].

Basically we have copied the original block and applied a Hamming window. The input of this block is a stream of IQ data in form of complex numbers and the output is a vector of numeric values which turns to be the ~~power~~. The significant parameters to be configured are the sample rate, the vector length or FFT size and the frame rate.

First it decimates the complex values stream coming at the specified sample rate, into vectors with a rate specified in the frame rate parameter and a length specified in the vector length parameter. These values are transformed with the FFT and a Hamming window, then we obtain the squared magnitude of these complex values, and its respective logarithmic value with decimal base obtaining the ~~power~~ (Code 3.1).

Since we had learnt previously that the sample rate is the bandwidth, what we do here is to transform the complex values stream into a 1024 (vector length/FFT size parameter) points vector that occupy 20 MHz (sample rate parameter). So if we get to know which is the center frequency set in the source, we have 1024 power values for frequencies 10 MHz above and below it. That means a power value each ~~19,531~~ kHz for this scenario.

That concept is going to drive the logic behind the file database creation of the next two custom blocks.

The file involved in this block are *tfm_logpowerfft_win.xml* (Annex VI) and *logpowerfft_win.py* (Annex VII).

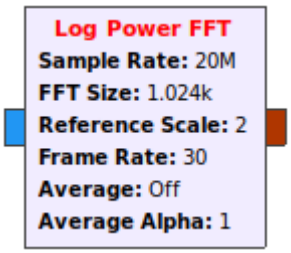


Figure 3.8: Default Log Power FFT block

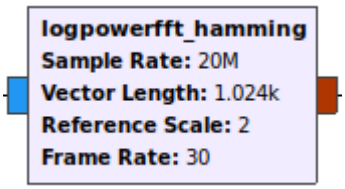


Figure 3.9: Custom Log Power FFT block with Hamming window

```
self._sd = blocks.stream_to_vector_decimator(item_size=gr.sizeof_gr_complex,
                                             sample_rate=sample_rate,
                                             vec_rate=frame_rate,
                                             vec_len=fft_size)

        fft_window = fft_lib.window_hamming(fft_size)
        fft = fft_lib.fft_vcc(fft_size, True, fft_window, True)
        window_power = sum([x*x for x in fft_window])

        c2magsq = blocks.complex_to_mag_squared(fft_size)
        self._avg = filter.single_pole_iir_filter_ff(1.0, fft_size)
        self._log = blocks.nlog10_ff(10, fft_size,
                                     -20*math.log10(fft_size)             # Adjust for number of bins
                                     -10*math.log10(float(window_power) / fft_size) # Adjust for
windowing loss
                                     -20*math.log10(float(ref_scale) / 2))      # Adjust for reference
scale
        self.connect(self, self._sd, fft, c2magsq, self._avg, self._log, self)
```

Code 3.1: logpowerfft_hamming operations code

3.5.2. Custom Block: power_analyzer

This is the block that performs the data operations to obtain the base power values and create a file database with them.

The input is a numeric vector of size specified in the vector length parameter and the output is a document created in the specified directory and its name depends on the other parameters: sample rate, center frequency and vector length (Figure 3.10).

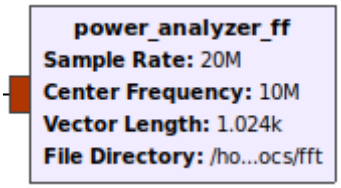


Figure 3.10: Custom power analyzer block

The name format of the output files is the following:

power_*(center_frequency)*MHz_*(sample_rate)*Msps_*(vector_length)*FFT.txt

So for the default values in the upper figure we have the creation of a document with the following name:

power_10MHz_20Msps_1024FFT.txt

A change in any of the three parameters will incur in the creation of a different file.

The document has a value at the top of it indicating the ~~number input vectors~~ that have been analyzed. This value is know as the file index and is completely important since this document works with averages, as we will see later.

Below this value we will find as many lines as specified in the vector length parameter. Each line will have the following format:

(power)@(frequency)

Frequencies will have a six digit decimal precision and are expressed in MHz; power will have two digit decimal precision and are expressed in dBm. The power value is an average of all the powers received at a specific frequency (Code 3.2). So a sample value will look like:

-80.61@2820.039101

```
73
-80.01@2820.000000
-80.95@2820.019550
-80.61@2820.039101
-79.81@2820.058651
-79.62@2820.078201
```

Code 3.2: Structure of a power file database

Every time we have a new input vector, we open the current file and create a new one with the same file name format followed by a *_tmp* suffix. This newly created temporary file will have all the new calculated values. Once all calculations are finished, the current file will be deleted, since it will be outdated, and the temporary file will be renamed without the *_tmp* suffix replacing the deleted file. We follow this process to avoid a more difficult process of editing line by line an already created file.

To calculate the new values, we simply multiply the index value per the power, add the new power value and calculate the new average for every frequency (Code 3.3).

As it was stated before, a change in any of the three parameters generates a completely different document. This can be explained by the fact that we cannot make operations between a file that has 20 MHz distributed in 1024 points, with another one that has the same 20 MHz distributed in 2048 points. Frequencies wouldn't match to calculate the new averages. This principle is exploded in the base scan script since we need to create the power averages for all the spectrum possible, and in that case the varying parameter will be the center frequency.

It is important to make an emphasis on the novel structure of the file database proposed in this project. Normally the output data stream of the HackRF with a sample rate of 10 Msps or 20 Msps, when is written to a single file can reach sizes of the order of the MB in a few seconds, and in the order of the GB after a few minutes. This data file contains a list of complex numbers that are difficult to understand their meaning and ~~to relate to its corresponding frequency.~~ To handle files of these sizes in the analysis phase, we would need a processing unit with high computation power, and with sufficient space in the disk to store the files. But since our project needs to be executed in a Raspberry with low-computation power and limited storage capacity, we created this file format to store the data.

This novel format restricts the file size since it doesn't grow indiscriminately in time, and at the same time the data it carries can be easily readable for a human, which helps us verify that the data that we are storing in the file reflects the ~~obtained~~ power values obtained by the HackRF. With this format the information in the file won't be increased when receiving new data, since for each new vector received, only the averaged power and the index will change, independently of the time the script runs.

Files for a vector length of 1024 will occupy around 20 kB, and for a length of 2048 its size will be around 40 kB, which make a it very efficient information storage method, because the file size will be almost the same for a scan of a few seconds, or ~~scan~~ of minutes or hours.

The files involved in this block are *tfm_power_analyzer_ff.xml (Annex VIII)* and *power_analyzer_ff.py* (Annex IX).

```
while not iterator.finished:
            current_freq = (iterator.index * self.freq_delta) + start_freq
            cached_power = 1000
            if file_exists:
               try:
                  cached_power = float(file.readline().split("@")[0]) #read power
               except Exception:
                  cached_power = 1000
            power = iterator[0]
            if cached_power != 1000:
               power = ((cached_power * file_index) + power) / (file_index+1)
            temp_file.write("%.2f@%.6f" % (power, current_freq/1e6))
            if (iterator.index != self.vlen-1):
               temp_file.write("\n")
            iterator.iternext()
         file.close()
         temp_file.close()
         os.remove(filename)
         os.rename(filename_temp, filename)
```

Code 3.3: Power operations and file replacement in the power_analyzer block

3.5.3. Custom Block: power_comparator

This block is in charge of receiving the real time data, compare against the base values and create a comparison file database for all frequencies.

The input is a numeric vector of size specified in the vector length parameter and the output is a document created in the specified directory and its name depends on the other parameters: sample rate, center frequency and vector length.

For this block to work, it is mandatory that the base power values file database ~~are~~ created and in the same directory as the specified in the parameters.

This block has two working modes that can be selected in the mode option: Percentage (Figure 3.11) or Fixed Value (Figure 3.12). Both modes require a numeric value for them to work. We'll explain both modes later in this chapter.

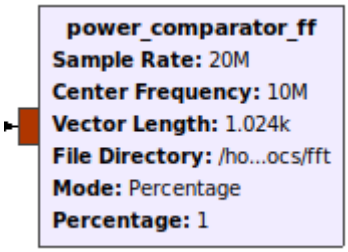


Figure 3.11: Custom power comparator block in mode fixed value

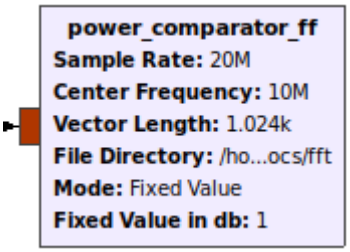


Figure 3.12: Custom power comparator block in mode percentage

The name format of the output file is similar to the created by the power analyzer and is the following:

compare_*(center_frequency)*MHz_*(sample_rate)*Msps_*(vector_length)*FFT.txt

So for the default values in the upper figure we have the creation of a document with the following name:

compare_10MHz_20Msps_1024FFT.txt

A change in any of the three parameters will incur in the creation of a different file.

Identically like in the power analyzer, the document has a value at the top of it indicating the number of input vectors that have been analyzed. This value is know as the file index and is completely important since this document works with averages too.

Below this value we will find as many lines as specified in the vector length parameter. Each line will have the following format:

(value1);(value2);(value3);(value4);(value5)@(frequency)

Value 1 correspond to number of values above threshold.

Value 2 correspond to an average of values above threshold with respect to the file index.

Value 3 correspond to the minimum difference between all values above threshold and the base value expressed in dB.

Value 4 correspond to the average difference between all values above threshold and the base value expressed in dB.

Value 5 correspond to the maximum difference between all values above threshold and the base value expressed in dB.

The last value indicates the frequency at which all the previous values are calculated in MHz.

Values 1 to 5 will have a two digit decimal precision and frequencies will have a six digit decimal precision (Code 3.4). So a sample value will look like:

21;0.66;0.31;5.16;10.20@60.273705

It means that 21 of all the values have exceeded the threshold, and it accounts for a 66% of all the received values. Of all the values compared, the minimum difference with respect to the ~~threshold~~ was 0.31 dB and the maximum 10.20 dB, and the average of all values above the threshold was 5.16 dB. All of this correspond to frequency 60.273705 MHz.

```
32
19;0.59;0.52;2.33;3.71@60.000000
17;0.53;0.18;2.22;4.36@60.019550
20;0.62;0.23;3.48;8.92@60.039101
22;0.69;0.22;3.17;9.14@60.058651
```

Code 3.4: Structure of a compare file database

When we have a new input vector, we open the base value file, the compare values file if exists, and create a new compare file with the same name format followed by a *_tmp* suffix. This newly created temporary file will have all the new calculated values. Once all calculations are finished, the current file will be deleted, since it will be outdated, and the temporary file will be renamed without the *_tmp* suffix replacing the deleted file. We follow this process just like in the power analyzer.

To calculate the values we have two different modes as explained earlier. Basically all the process will be similar except for the calculation of the threshold value. We read first the base power value and the previous comparison values if they exist, then ~~obtain~~ the threshold value and compare it to the real time input value. If the value exceeds the threshold, then we compare the values to the minimum (value 3) and maximum (value 5) and ~~replace~~ if necessary. Likewise, we calculate the average of the exceeded values (value 4) with the help of the file index, and increase the number of values above threshold (value 1) and its average (value 2). All of this is done for all frequency values in the input vector.

To calculate the threshold value in percentage mode, we simply calculate it multiplying the base value per one plus the percentage value specified in parameters. So if the base value is 50 dBm and the percentage value is 10, we multiply 50 * (1 + 0.10) and obtain a threshold value of 55 dBm. The threshold value in fixed value mode is easier to obtain

since we just add the value specified in the parameters with the base value, so if the base value is 50 dBm and the fixed value is 5, the threshold is 55 dBm (Code 3.5)

```
if self.mode == 1: #percentage
    threshold = cached_power*(1+self.diff_percentage/100)
else: #fixed db
    threshold = cached_power+self.diff_db
if power > threshold:
    exceeded_diff = power - cached_power
    exceeded_diff_min = numpy.minimum(exceeded_diff_min,exceeded_diff)
    exceeded_diff_average = ((exceeded_diff_average * exceeded_number) +
            exceeded_diff) / (exceeded_number+1)
    exceeded_number = exceeded_number+1
    exceeded_diff_max = numpy.maximum(exceeded_diff_max,exceeded_diff)
exceeded_average = exceeded_number/(file_result_index+1)
temp_file.write("%.0f;%.2f;%.2f;%.2f;%.2f@%.6f" % (exceeded_number,
            exceeded_average, exceeded_diff_min,
            exceeded_diff_average, exceeded_diff_max, current_freq/1e6))
```

Code 3.5: Threshold and values calculations in the power_comparator block

Therefore each frequency has its own threshold, which gives us flexibility, so we don't have an unique threshold value. The differences from the threshold are used so we can spot easily unusual peaks and also with the help of averages we can see how often values are above threshold for each frequency.

The novel file structure follows the advantages described in the power analyzer custom block. The file size for a 1024 vector length will be around 40 kB, and for a 2048 vector length around 80 kB. This proves to be a very efficient data storage method because the file size won't vary significantly if the scan lasts a few seconds, or several minutes.

The file involved in this block are *tfm_power_comparator.xml* (Annex X) and *power_comparator_ff.py* (Annex Xi).

3.5.4. Script: Scan Base

This script is in charge of obtaining the average power values for all the spectrum range from 1 MHz up to 6 GHz.

The script is created initially in GNURadio Companion with three components which are the ~~source~~, the custom ~~log power~~ block with Hamming window, and our custom ~~power analyzer~~ block (Figure 3.13). We also have six variables and four user interface components. The variables are defined to hold the values of the most important ~~values~~ such as the frequency, sample rate and directory. The user interface components will show the values of the previously mentioned variables, ~~and we have and additional component t~~hat will control the time between frequency jumps (Figure 3.14). We'll explain how this work, later in this chapter.

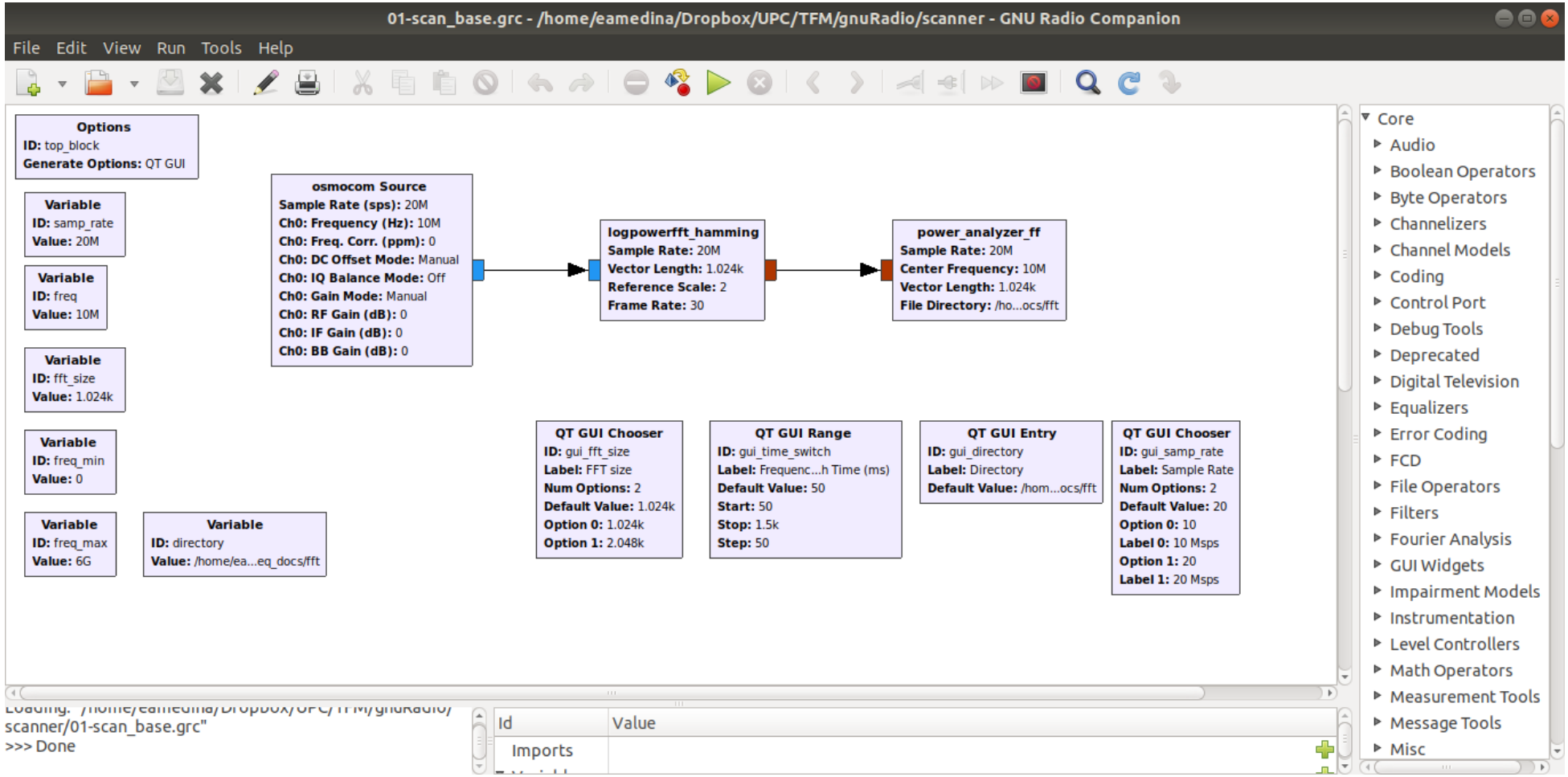


Figure 3.13: GNURadio Companion block structure for base scan

As it was explained in the GNU Radio Companion chapter, we have found several limitations that impede us ~~to~~ accomplish ~~our objectives~~ using only the IDE. ~~We need to change the central frequency at which this script's components operate each certain amount of time, so we can cover all the supported spectrum by the HackRF~~. For this purpose we must edit the script manually and we need a solid understanding of GNURadio blocks, QT and Python.

~~Straight out of GNURadio Companion,~~ the script indicates the central frequency to the HackRF source and it feeds with the data stream the log power block, which decimates the data and converts it into a vector which size is determined by the vector length parameter. This output is fed into the custom analyzer block which generates the power values database file for that specific frequency, sample rate and vector length. It works correctly, but for one and only center frequency.

In order to adjust the script to our requirements of analyzing the spectrum from 1 MHz to 6 GHz, we need to create a process that changes the value of the center frequency every certain amount of time. For that purpose we choose a Timer from the QT framework, which changes the value of the frequency every certain time defined in the *time_switch* variable. This process starts with the execution of the script, and is changing the frequency value while the script is under execution.

The function in charge of changing the frequency is *recurring_timer*, which sets it in a value between the determined limits and dependent on the sample rate parameter (Code 3.6). When we call the method *set_freq* we change the value of the frequency variable and ~~inform both the source and the analyzer that the frequency has changed~~, so both can be synchronized and the analyzer can know to which frequencies correspond the values that it receives.

```
def startTimer(self):
    self.timer = QtCore.QTimer()
    self.timer.setInterval(self.time_switch)
    self.timer.timeout.connect(self.recurring_timer)
    self.timer.start()

def recurring_timer(self):
    if (self.get_freq()+self.samp_rate >= self.get_freq_max()):
        self.set_freq(self.get_freq_min()+self.samp_rate/2)
    else:
        self.set_freq(self.get_freq()+self.samp_rate)

def set_freq(self, freq):
    self.freq = freq
    self.osmosdr_source_0.set_center_freq(self.freq, 0)
    self.tfm_power_analyzer_ff_0_0.set_center_freq(self.freq)
```

Code 3.6.Timer in charge of switching the frequency

The only value that can be modified by the user is the frequency ~~time switch~~, which is specified in milliseconds, and indicates the time at which the timer make the calls to change the frequency. ~~We recommend a minimum value of 50 ms., since we have~~

acknowledged that this time allows to have at least one input vector at the analyzer block per frequency jump.

Using a sample rate of 20 Msps in the 1 MHz – 6 GHz range, we must perform 300 frequency jumps accounting for a total time of around 15 seconds for the whole spectrum range. This time can be greater, which means that we could have more than one reading at every center frequency, but the time to make a total spectrum loop will take longer.

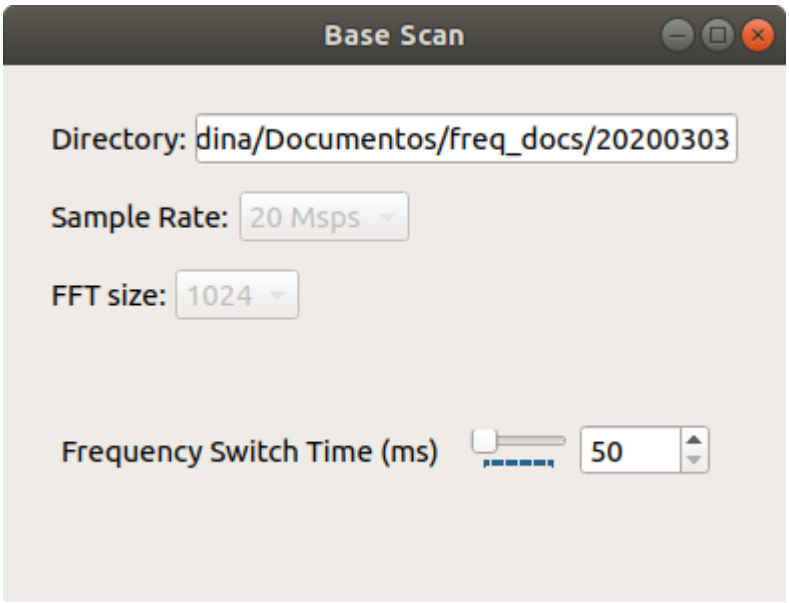


Figure 3.14: User interface components in base scan script

As we learned in the section of the *power_analyzer* custom block**,** its output is a file with power averages whose name depends on the frequency, sample rate and vector length. Since in this script the frequency value is changing, the power analyzer block will generate as many files as frequency jumps. So, if it is the case of the previously explained example with 300 frequency hops, we will have 300 files with the averaged powers for each determined frequency. All these files will account for a total size of around 6 MB. This size won't vary significantly if the script runs for a few seconds or for hours, proving to be a very efficient data storage method.

All the files generated by this script will be fundamental to run the other spectrum analysis scripts and for the main script which displays comparison graphics and a comparison value table. Therefore it is mandatory to be the first executed script when using this project's software.

The files involved with this script are *01-scan_base.grc* and *01-scan_base.py* (Annex XII).

3.5.5. Script: Spectrum Scan

This script is in charge of comparing the real time power values with respect of the base power values for all the spectrum range from 1 MHz up to 6 GHz.

The script is created initially in GNURadio Companion with three components which are the source, the custom log power block with Hamming window, and our custom power *comparator* block (Figure 3.15). We also have seven variables and eight user interface components, to give the user a greater control over its execution. As in the previous script, the variables are defined to hold the values of the most important values such as the central frequency, sample rate, directory, and frequency switch time. Additionally we define variables to control the upper and lower frequencies limits, and also variables to control the operation mode of the *comparator* custom block.

The user interface components will show the values of the static variables, and will provide components that will allow the user to control the time between frequency jumps, the upper and lower frequencies boundaries, and the *comparator* mode and value (Figure 3.16). We'll explain how this work, later in this chapter.

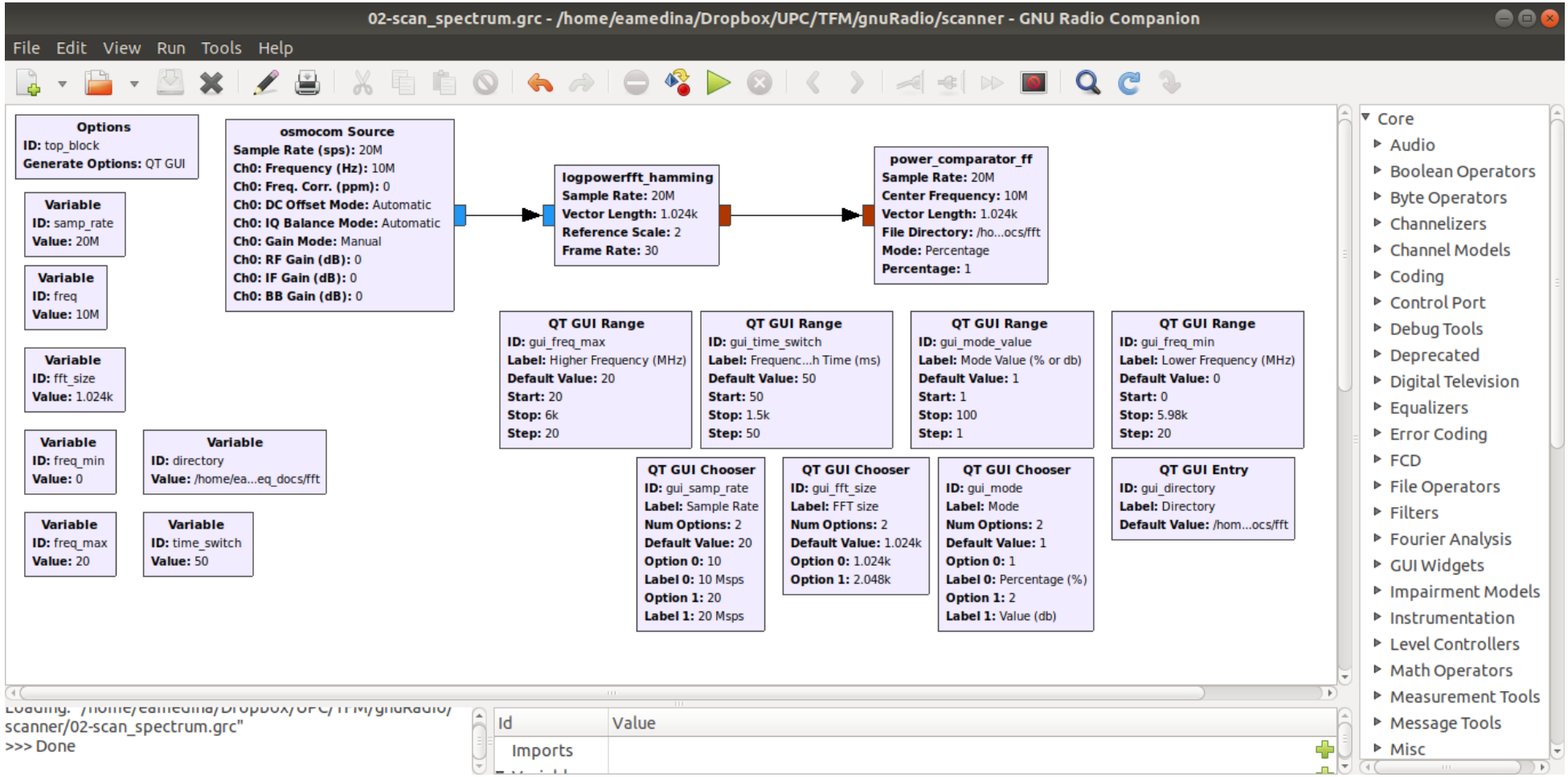


Figure 3.15: GNURadio Companion block structure for spectrum scan

~~Just like~~ in the base scan script, we face the impossibility to create a process that changes the central frequency, so we have to do it manually modifying the script generated by GNURadio Companion.

The original script indicates the central frequency to the source and it feeds with the data to the *logpowerfft_hamming* block, which decimates the data and converts it into a vector which size is determined by the vector length parameter. This output is fed into the custom *comparator* block which generates the comparison values database file for that specific frequency, sample rate and vector length. It works correctly, but for one and only center frequency.

In this script we also analyze the spectrum from 1 MHz to 6 GHz, or user defined boundaries, so we need to create a process that changes the value of the center frequency every certain amount of time. We reuse the same solution that was used in the base scan script, choosing a QT Timer, which changes the value of the frequency every certain time defined in the *time_switch* variable, and this process begins with the execution of the script.

The functions that perform the task of changing the frequency are the same as in the base scan script, and the code is the same as in Code 3.6.

Besides using a different custom block as the file generator, in this script we give the user the chance to change more variables to her convenience. The user can change the value of the frequency switch time, the lower and upper frequency boundaries, and the operation mode and value of the *comparator*. This will give the user the flexibility to adapt the frequency band she wants to scan, so it doesn't scan the whole 1 MHz to 6 GHz spectrum, but focuses on bands which are of interest for the user. At the same time, the operation mode and mode value let the user modify the threshold values during the execution of the script and adapt the results obtained to her changing needs.

The frequency switch time indicates the value at which the timer executes the frequency changing function. For this script we have set a default value of 250 milliseconds, because the *comparator* has more operations to perform with the data and works with multiple files, and with this time we assure that we produce the corresponding file at every frequency jump. For a sample rate of 20 Msps, 300 frequency jumps will be performed accounting for a total time of 75 seconds to analyze the total spectrum from 1 MHz to 6 GHz. This time will vary if the user defines different boundary frequencies.

The user has the option to modify the lower and upper boundary frequencies, limiting the range at which the frequency jumps will be performed. This allows the user to focus on more specific frequencies rather than in the whole spectrum. These variables can change dynamically while the script is in execution providing flexibility to the program.

As we learned in the section of the *power_comparator* custom block**,** its output is a file with differences and averages compared to the base values, and whose name depends on the frequency, sample rate and vector length. Since in this script the frequency value is changing, the power analyzer block will generate as many files as frequency jumps. So, if it is the case of the previously explained example, we will have 300 files with the differences and averages for each determined frequency. All these files will account for a total size of around 12 MB. This size won't vary significantly if the script runs for a few seconds or for hours, proving to be a very efficient data storage method.

All the files generated by this script will be important to the main script which displays a graphic with both the base values and the maximum difference value obtained from these files.

The files involved with this script are *02-scan_spectrum.grc* and *02-scan_spectrum.py* (Annex XIII).

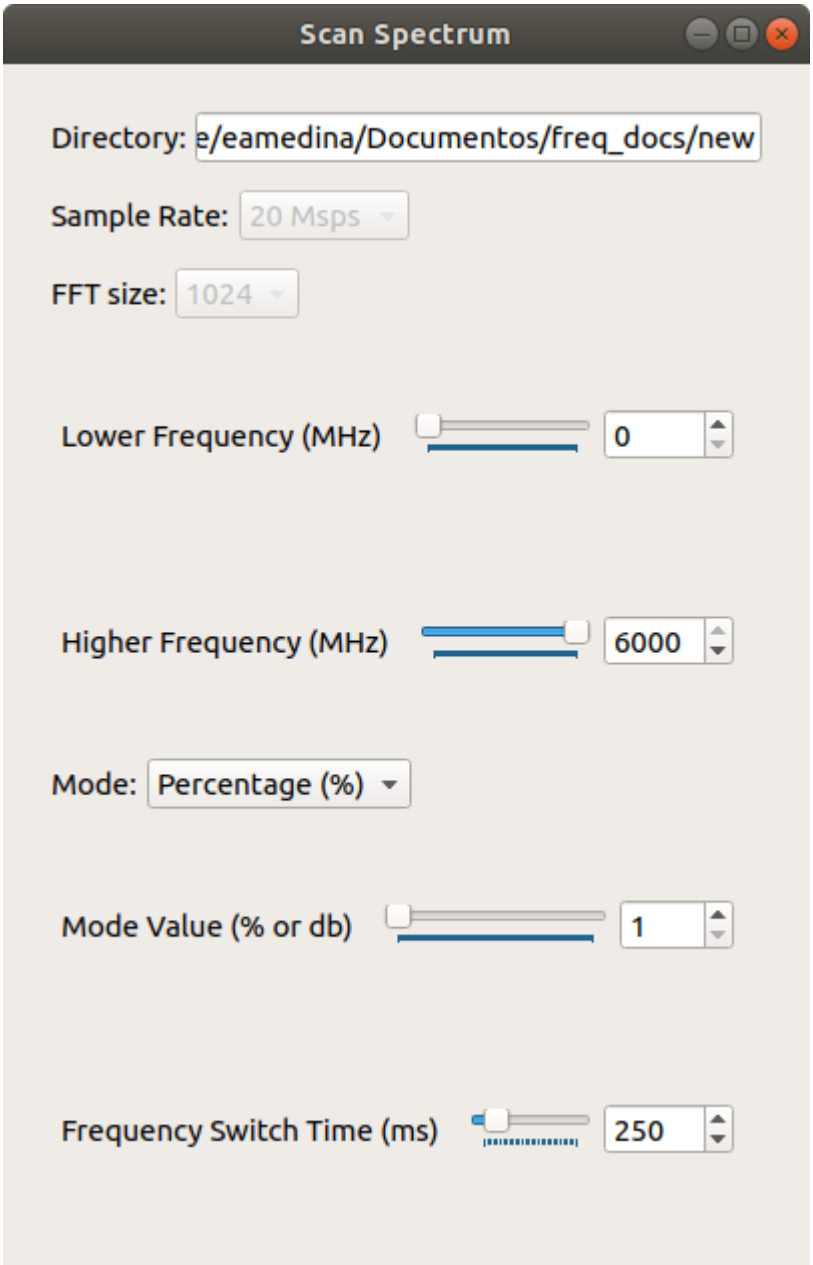


Figure 3.16: User interface components in spectrum scan script

3.5.6. Script: Band Scan

This script is in charge of comparing the real time power values with respect of the base power values for a limited bandwidth of up to 20 MHz and with graphical tools included.

The script is created entirely in GNURadio Companion with four components which are the source, the custom log power block with Hamming window, our custom power *comparator* block, and a native *QT GUI* sink (Figure 3.17). We also have six variables and six user interface components (Figure 3.18). As in the previous scripts, the variables are defined to hold the values of the most important values such as the center and boundaries frequencies, sample rate, directory and operation mode and values of the *comparator* custom block.

Contrary to the base scan and spectrum scan scripts, we won't have automatic frequency changes, we will give the user the total control of the frequencies analyzed when she want to focus its analysis in a given bandwidth. For that reason this script doesn't need to be modified.

The source feeds with the data to the *QT GUI* sink and the log power block, which decimates the data and converts it into a vector which size is determined by the vector length parameter. This output is fed into the custom *comparator* block which generates the comparison values database file for that specific frequency, sample rate and vector length. The *QT GUI* sink is fed directly from the source, and has options to display a real time FFT plot or a waterfall plot.

The operation mode selection and values for the *comparator* block follow the same guidelines as in the spectrum scan script.

Since this script is for a limited bandwidth determined by the HackRF characteristics, we give the user an option to choose preset configurations in which we find the most common bands that can have drone RF activity. This preset bands option sets the variables to predefined values of center frequencies depending on the sample rate. We support only two sample rates, so for each option selected, we could have two different frequencies that work with its corresponding rate. One of the bands of interest is the 2.4 GHz band, and this band will be divided into chunks of a width determined by the sample rate, so for 20 Msps we can choose among five bands options, and for 10 Msps we can choose among ten bands options.

The main purpose of this script is to provide the user with a tool to inspect visually the band of interest. When using this script the file database will also be generated for the specific frequency and sample rate, so all values obtained during the use of this script will be reflected in the files.

The files involved with this script are *03-scan_band.grc* and *03-scan_band.py* (Annex XIV).

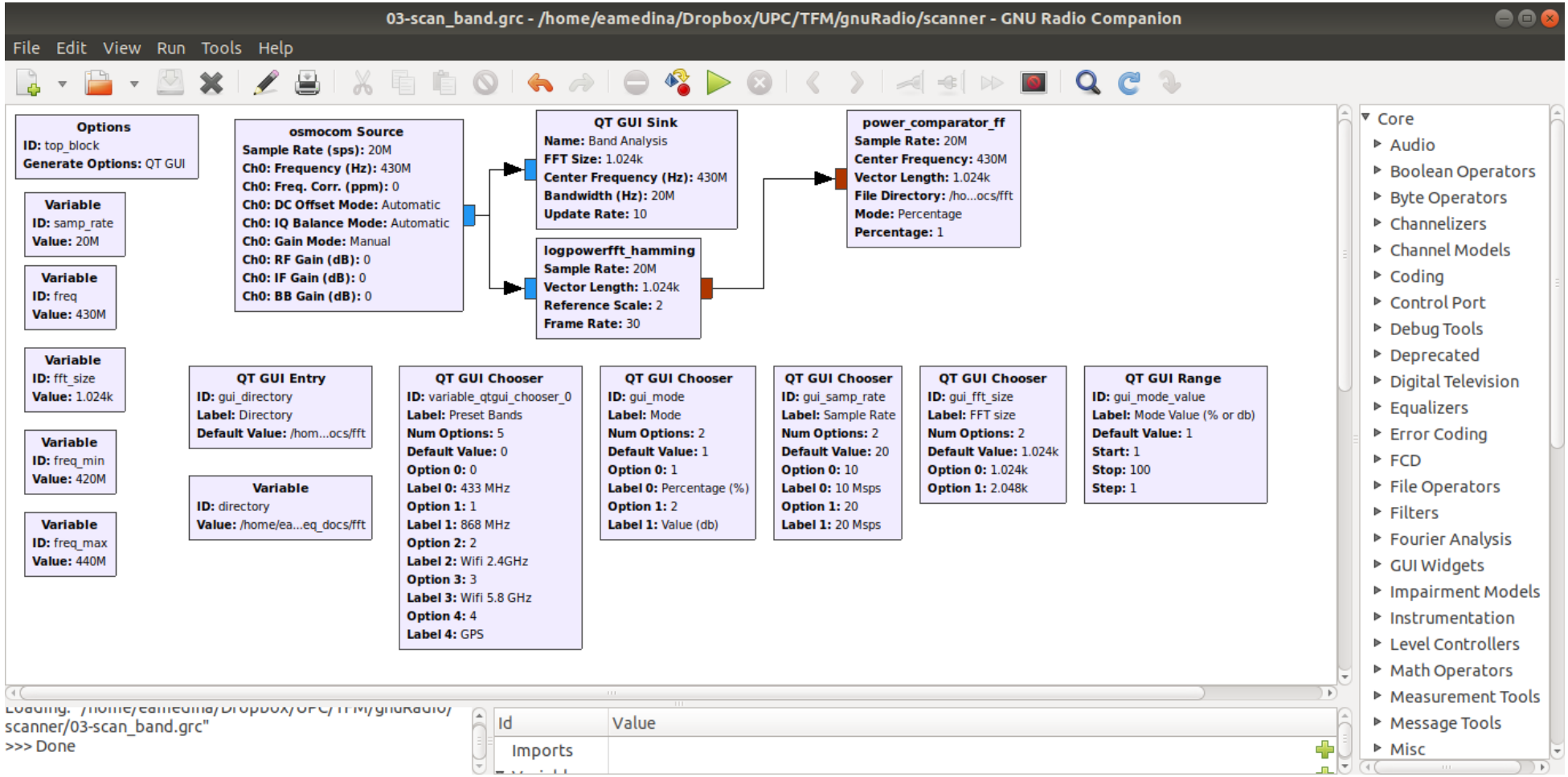


Figure 3.17: GNURadio Companion block structure for band scan

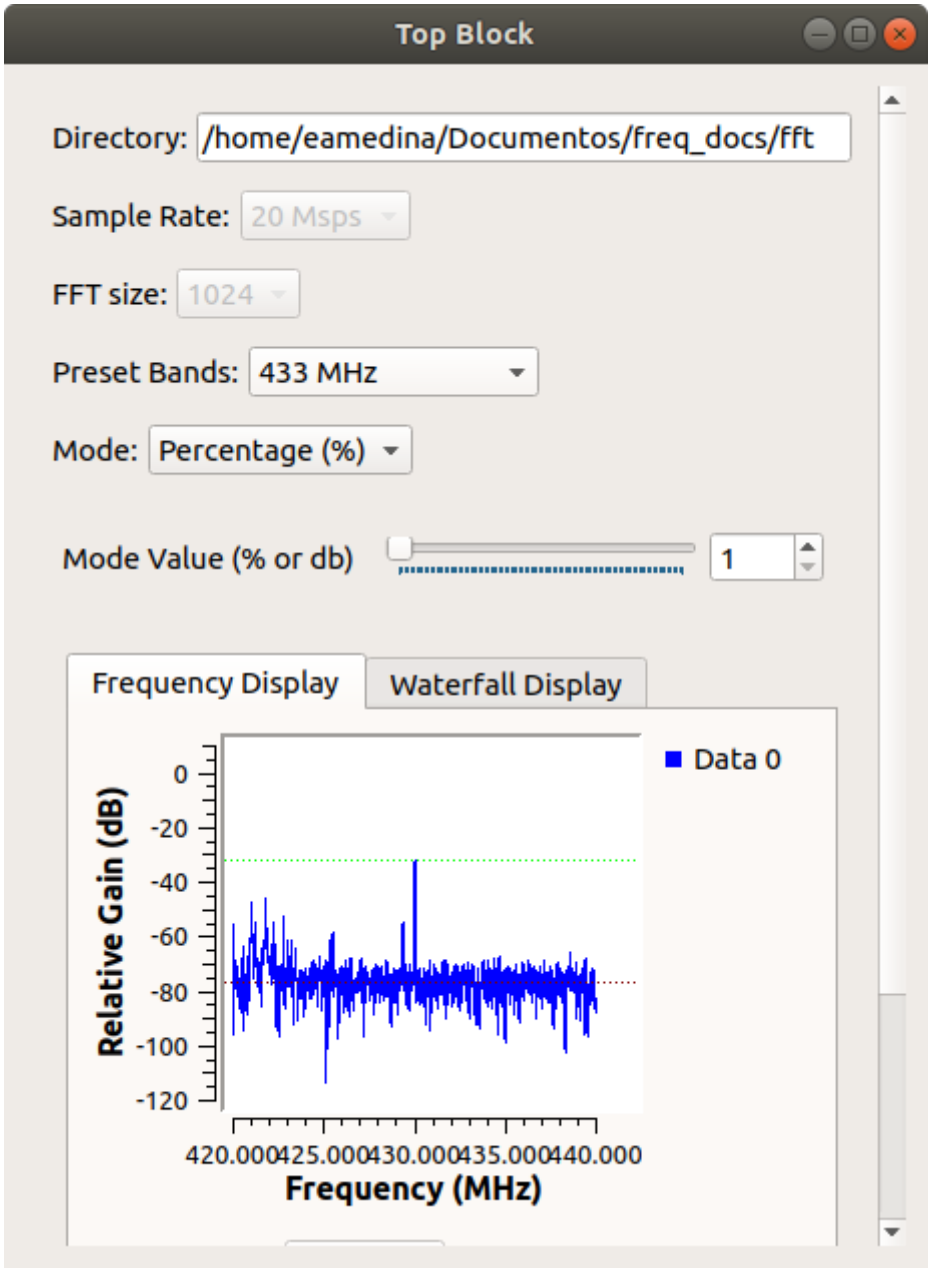


Figure 3.18: User interface components in band scan script

3.5.7. Script: Jammer

This script is in charge of generating a jamming signal of up to 20 MHz bandwidth in a frequency selected by the user.

The script is created initially in GNURadio Companion with four components which are the noise source, a multiply block, a throttle and the sink which in this case will be the HackRF (Figure 3.19). We also have four variables and nine user interface components (Figure 3.20). The variables are defined to hold the values of the center and boundaries frequencies, sample rate and frequency switch time. The user interface components will give the user the opportunity to control the frequencies at which the noise will be emitted, and to control the HackRF gain parameters. We give the user the option to generate the noise in two modes, where the central frequency can be static or dynamically changed. We'll explain how this modes work later in this chapter.

Even though the script generated by GNU Radio Companion can work out of the box, we want to provide the user with the option to generate the jamming noise in a frequency band greater than the maximum bandwidth allowed by the HackRF, so we need to modify this script to achieve this functionality.

The original script consists of a noise source, that will use the CPU power to generate all the data, connected to a multiply block, that gives the source and additional gain, which is attached to a throttle block, in charge of optimizing the use of the CPU power, and finally the HackRF sink which emits all data. This solution works for a determined frequency and sample rate.

The limitation of 20 Msps as the maximum sample rate supported by the HackRF, gives us a 20 MHz maximum bandwidth that we can interfere in. But in frequency bands such as 2.4 GHz, drones can use frequency hopping protocols along the whole band, so we must have the ability to interfere communications across all the band when this kind of behavior is present.

For that purpose, just like in the base and spectrum scan scripts, we need to create a process in which the frequency changes in time. We reuse the same solution that was used in the base scan script, choosing a QT Timer, in charge of changing the frequency value every certain time defined in the *time_switch* variable. But contrary to previous scripts in which this process started at the execution of the script, in this jammer script the timer process will be initiated and terminated by the user exclusively.

We have two operations modes: a fixed bandwidth mode which generates the noise signal with a bandwidth of 10 MHz or 20 MHz with an invariable center frequency; and a continuous mode in which the interfering signal will change its central frequency across the boundaries indicated by the lower and upper frequencies set by the respective user interface components.

Similar to what we saw in the band scan script, we provide the user with a set of pre-configured options based on the most common frequency bands in which we can find drone activities. Additionally, we provide a set of options corresponding to GNSS frequencies, so we could interfere in the navigation of drones operating in autonomous mode.

The files involved with this script are *04-jammer.grc* and *04-jammer.py* (Annex XIV).

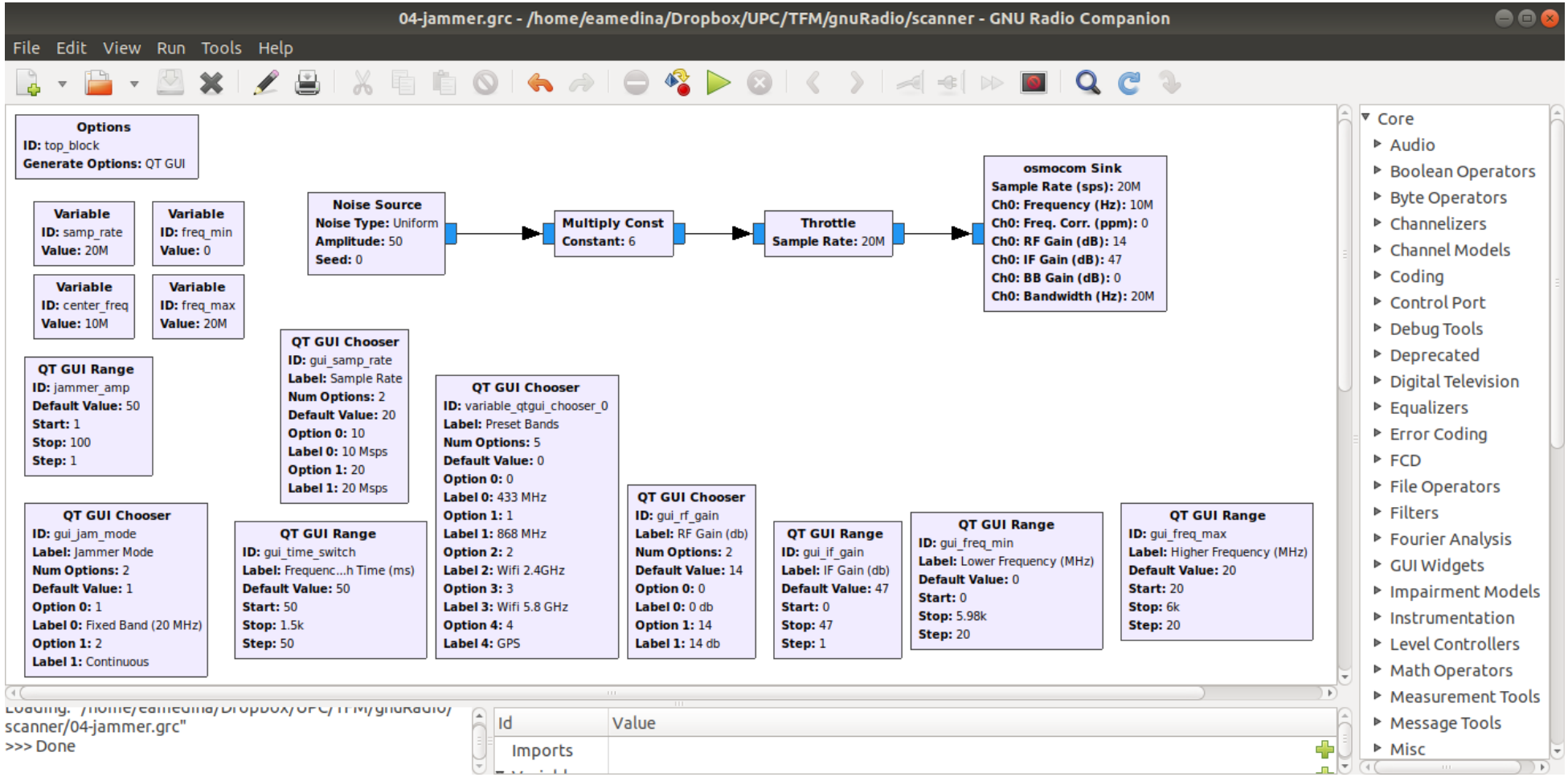


Figure 3.19: GNURadio Companion block structure for jammer script

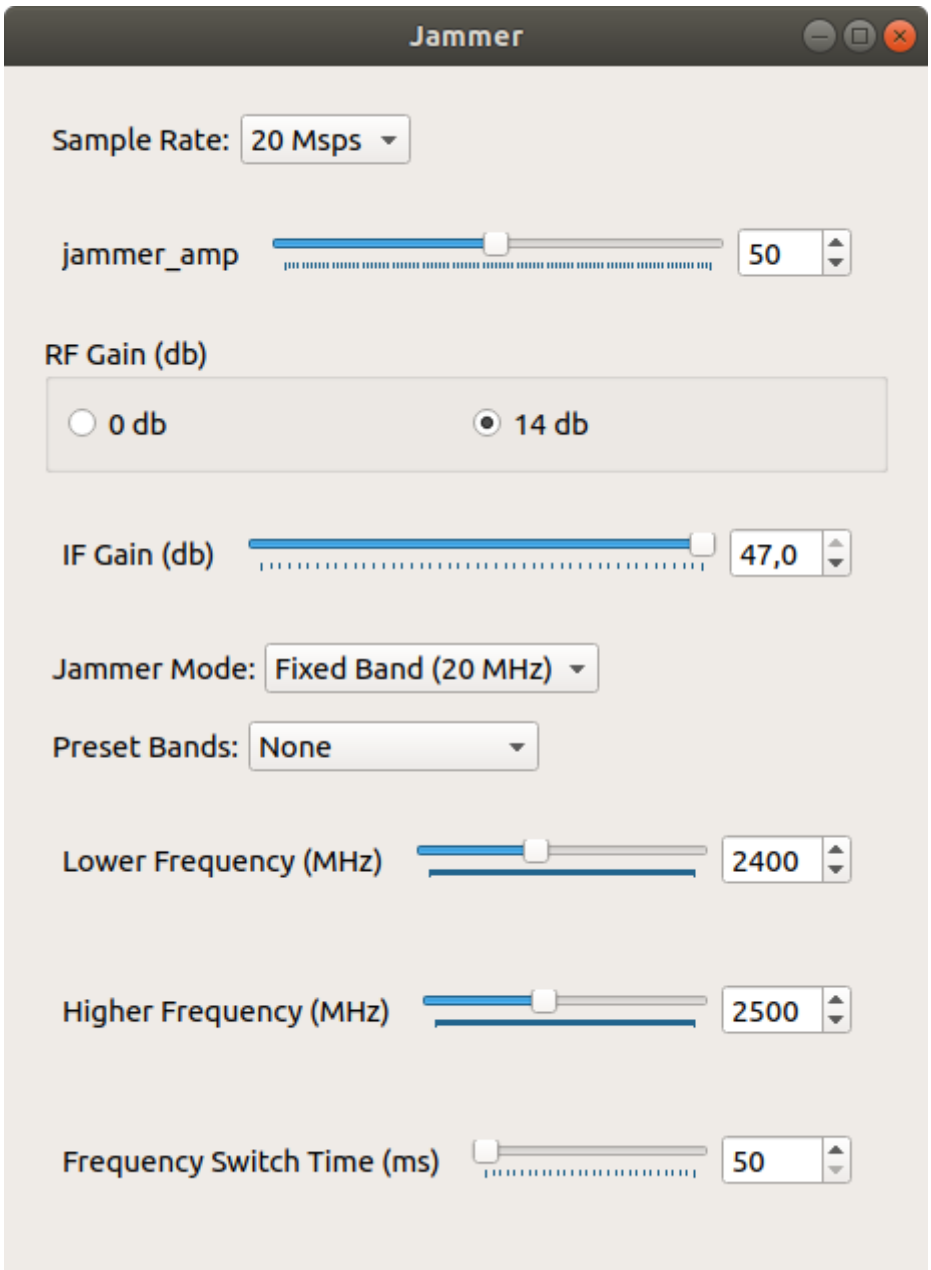


Figure 3.20: User interface components in jammer script

3.5.8. Script: Main Program

The main script of the project is the controller window. It has four buttons that execute the data acquisition scripts, the analysis scripts, and the inhibition script. It has three fields to configure the parameters that will be passed to the child scripts, and get the respective file to generate the graphics and table. These parameters are the directory, sample rate and FFT size.

In this project we are working with a file database approach, so that all information gathered and produced by our scripts will be stored in files that are located in the directory specified in the parameters. At the same time, these files depend on the values of sample rate and FFT size as we have seen previously.

In this main window, we also have a graphic section, a table and an additional option with the “Choose band” label. When there is no data in the directory, it shows an empty graphic and an empty table (Figure 3.21). If there is data, we see a graphic displaying a line with the amount of points specified in the FFT size parameter, in an interval of frequencies specified in the sample rate parameter (Figure 3.22). The blue line shows the data obtained from the Base Scan script, which are averages of the power received in that frequency band. The red line shows the data obtained in the Spectrum Scan and/or Band Scan scripts, specifically the maximum power differences from the base obtained in that frequency band.

The table below shows all the data processed from the Spectrum Scan/Band Scan scripts. The table shows all the frequencies measured, from 1 MHz to 6 GHz, with the processed data. Since we are dealing with differences from the base power values, here we display those values, by default the table will be sorted in descending order with the maximum difference value. All columns of this table can be sorted in ascending or descending order with a click in the respective header.

The “Choose Band” parameter is set by default in the “CONTINUOUS” option. This means that in the background a timer will change the data displayed in the graphic every 3 seconds. If the sample rate is set to 20 Msps, it will display graphics with power values over a frequency band of 20 MHz, changing the center frequency every 3 seconds.

It also has an option for “ALL” frequencies, which will display all the data from 1MHz to 6GHz. As it is does not provide a sharp look at the data, extra controls are given so that the user can change the frequency and the bandwidth of the graphic. It is important to understand the file generation process of the base and spectrum scan scripts to select with accuracy the parameters that will generate a graphic in this option. Other options available in the “Choose Band” are the most used bands for drone communications which are 433 MHz, 868 MHz, and the 2.4 GHz band. This provides an easy configuration to check the behavior of the spectrum graphically in bands of interest.

The purpose of the development of these two tools is to help the user to identify uncommon signals both visually and statistically, so the user can focus on analyzing a specific band and confirm the presence of a drone.

To execute the script and run all the scripts we have seen that are part of this software, we need to follow the indications of the Annex IV.

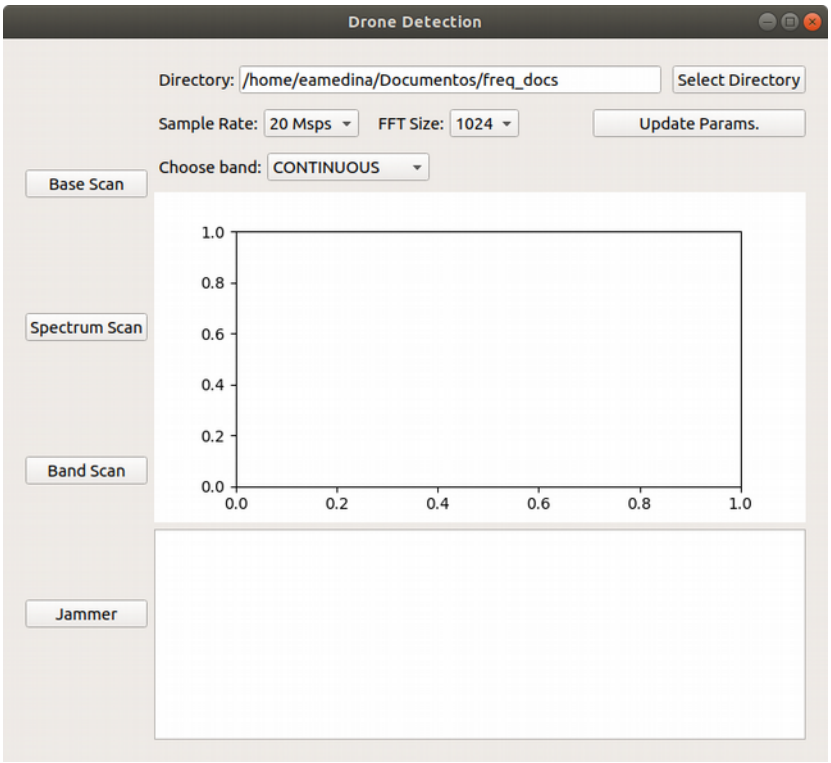


Figure 3.21: Main script with no data

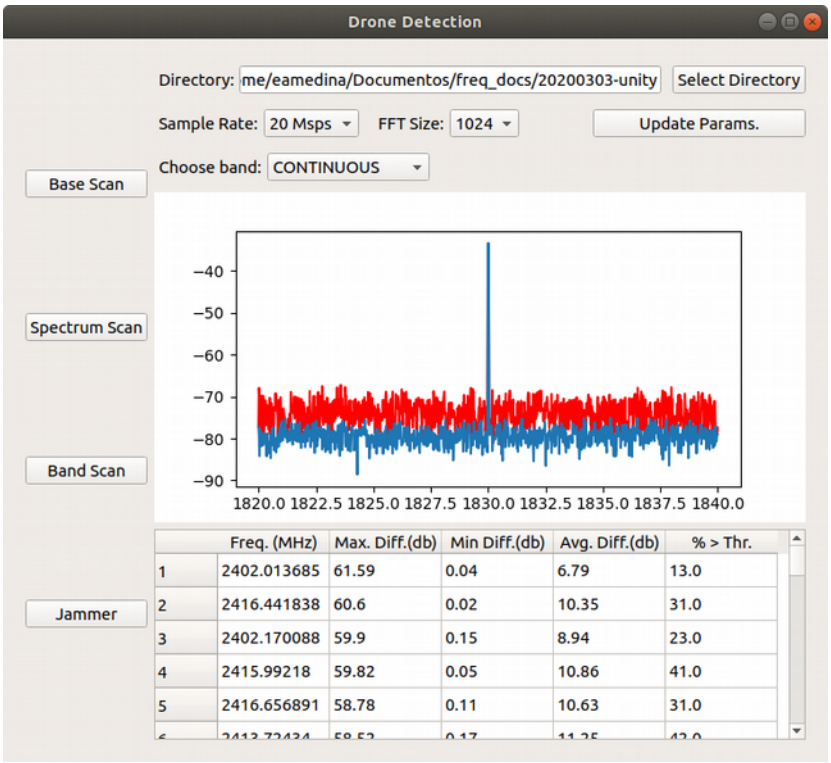


Figure 3.22: Main script with data in continuous mode

4. Tests and Results

To proceed with the testing of this project we will adopt two phases: first we will make tests on a laptop and then we will make the same tests in a Raspberry Pi. We will be analyzing both the accomplishment of the goals of this project and also verify the performance of the software.

The tests will consist of proving the correct functioning of all the scripts, and for the detection and interference of signals we will use three different RF systems operating at different bands.

The first system consists of a toy car with remote controller working at 27 MHz. The second is a universal garage remote controller working at 433 Mhz. And the last one is a drone working at 2.4 GHz. The intention to have diverse systems that aren't specifically drones is to prove that our solution can work in different segments of the spectrum.

4.1. Base Scan Script Tests

Our main focus in this script tests is to verify that this script creates the corresponding file database, and we will also monitor the system performance in the executing machine. We will try four different parameter combinations with 1024 and 2048 as the FFT size and 10 and 20 Msps as the sample rate, and each will have its own folder. We will execute /the script for five minutes for each combination in the laptop and fifteen minutes in the Raspberry with different frequency switch times, and we will explain its reason in the Raspberry tests section.

The setup for this test consist of the laptop or Raspberry running the script and the HackRF One connected to it via USB cable. All tests will have the same setup, since the only difference among them will be the parameters configured in the script. The tests will be performed in indoor environment.

Figure 4.1: Laptop base scan script test setup

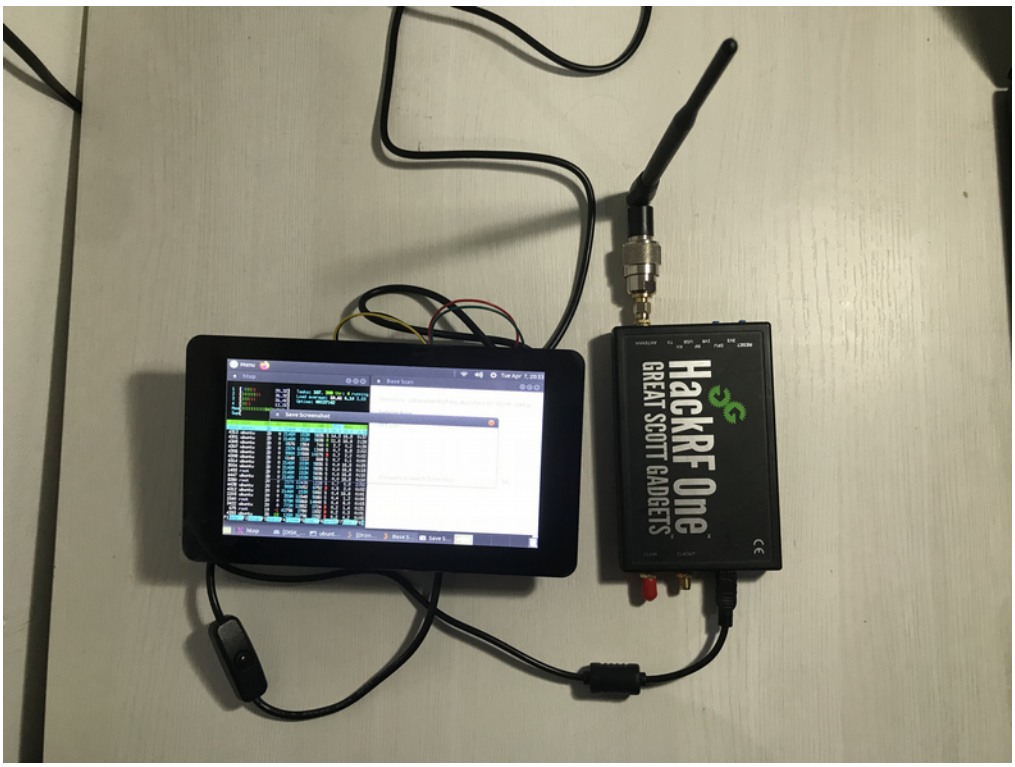


Figure 4.2: Raspberry base scan script setup

4.1.1. Laptop Test: 10Msps 1024FFT 50ms

For this configuration, the script creates 600 files that occupy 11.6 MB, accounting for a size of less than 20 kB each. When the script is running we see it uses around 82% of CPU cores and 2.00 of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 30% for any of the cores.

When we explore the files, we can see a correct format with power and frequencies, and files have been created mostly with 15 input vectors, that indicates that around 15 full spectrum scans have been performed.

We can see in the file that the frequency separation is 9.775 kHz which correspond to the formula 10 MHz (sample rate) over 1023 (FFT size minus one), since we are using zero indexed values.

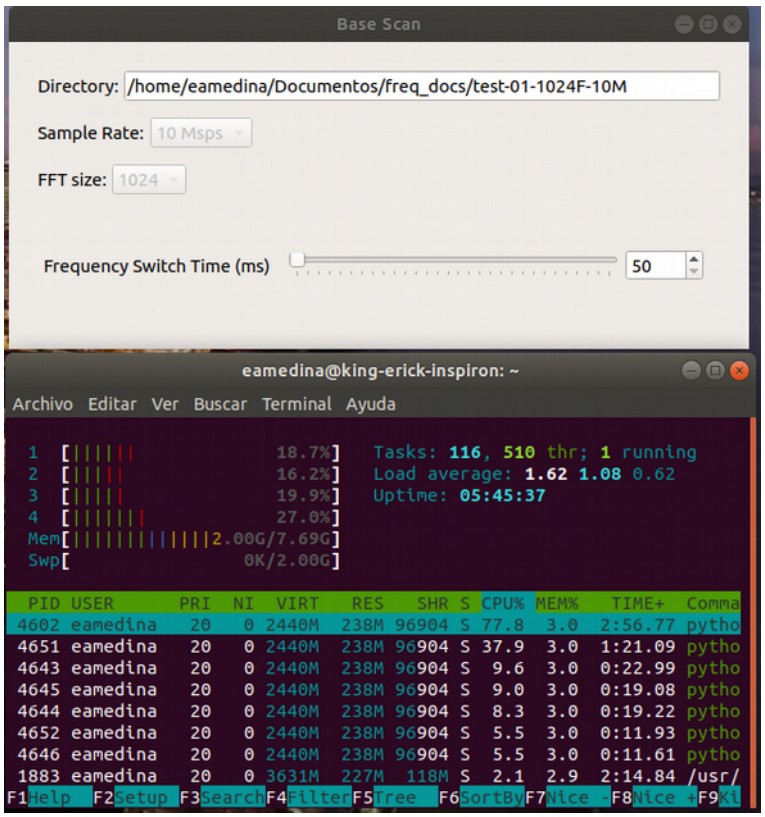


Figure 4.3: Laptop base script parameters configuration for 10 Msps-1024 FFT size-50 ms frequency jump time. CPU consumption below.

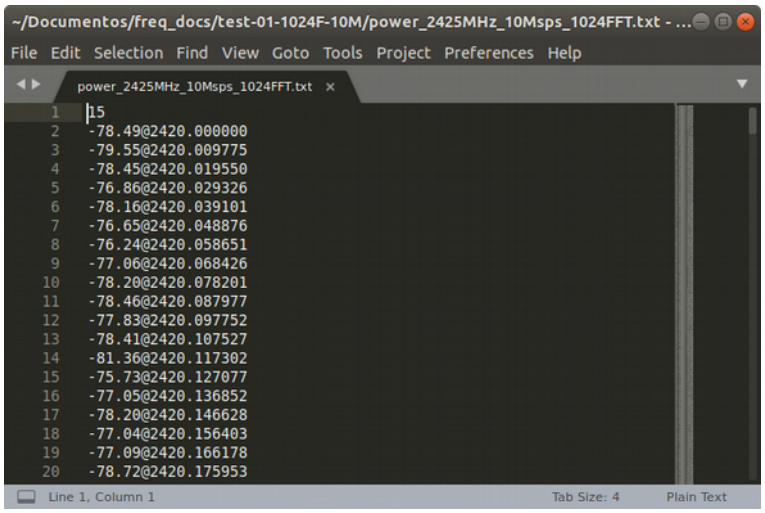


Figure 4.4: File structure created by base script with corresponding index and frequency values for 10 Msps-1024 FFT size-50 ms frequency jump time.

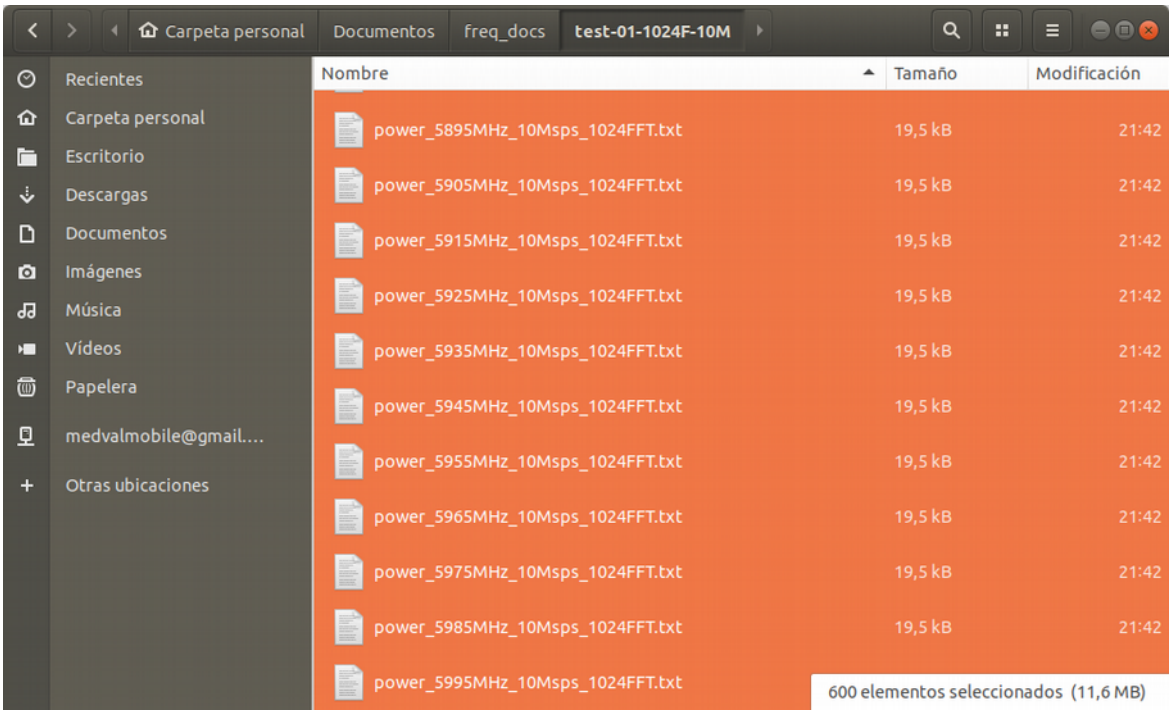


Figure 4.5: Files generated by base script with its names and sizes for 10 Msps-1024 FFT size-50 ms frequency jump time.

4.1.2. Laptop Test: 10Msps 2048FFT 50ms

For this configuration, the script creates 600 files that occupy 23.1 MB, accounting for a size of less than 40 kB each. When the script is running we see it uses around 180% of CPU cores and 2.06 of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 55% for any of the cores.

When we explore the files, we can see a correct format with power and frequencies, and files have been created mostly with 15 input vectors, that indicates that around 15 full spectrum scans have been performed.

We can see in the file that the frequency separation is 4.885 kHz which correspond to the formula 10 MHz (sample rate) over 2047 (FFT size minus one), since we are using zero indexed values.

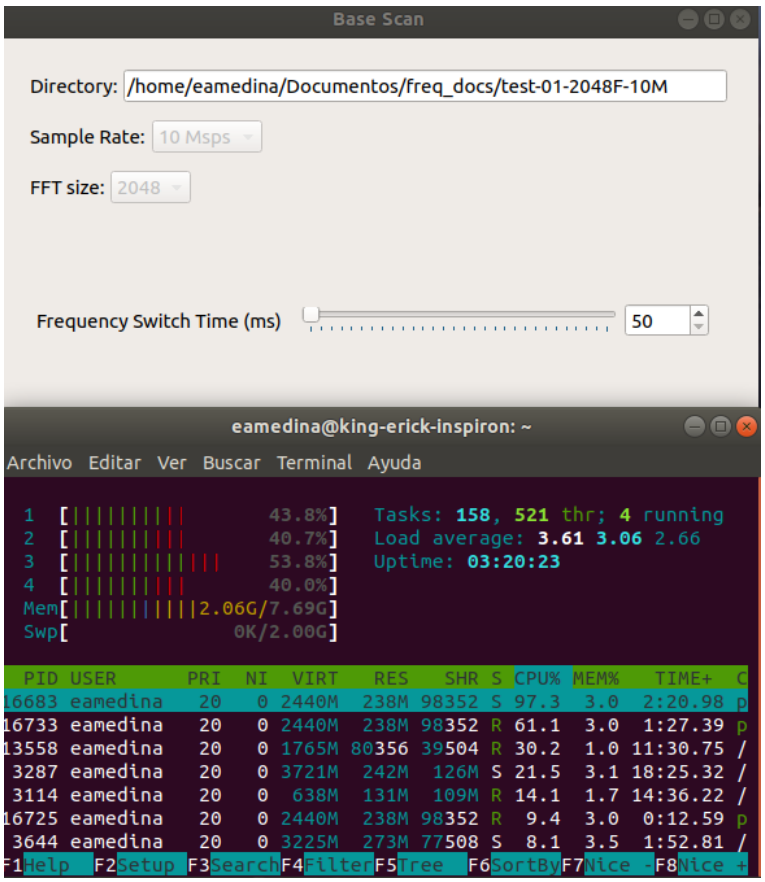


Figure 4.6: Laptop base script parameters configuration for 10 Msps-2048 FFT size-50 ms frequency jump time. CPU consumption below.

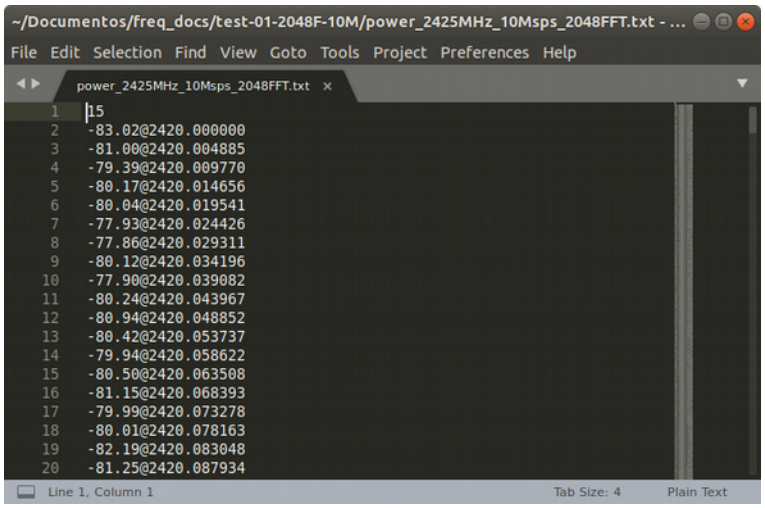


Figure 4.7: File structure created by base script with corresponding index and frequency values for 10 Msps-2048 FFT size-50 ms frequency jump time.

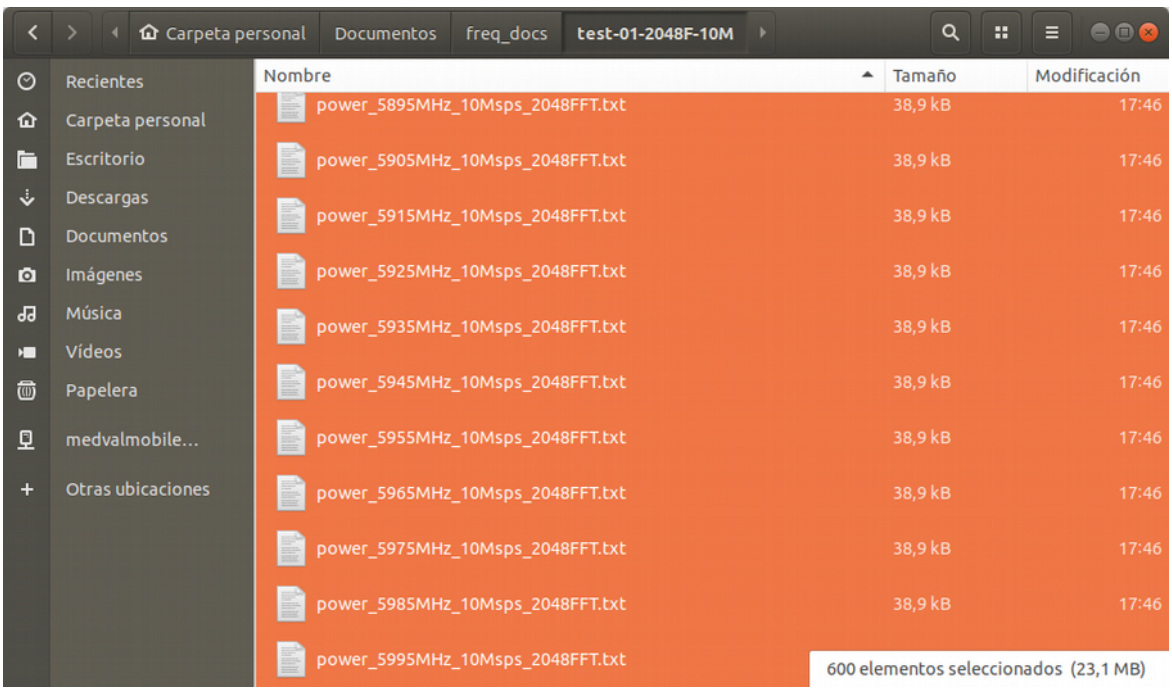


Figure 4.8: Files generated by base script with its names and sizes for 10 Msps-2048 FFT size-50 ms frequency jump time.

4.1.3. Laptop Test: 20Msps 1024FFT 50ms

For this configuration, the script creates 300 files that occupy 5.8 MB, accounting for a size of less than 20 kB each. When the script is running we see it uses around 161% of CPU cores and 2.09 of 7.6 GB of RAM memory, and the CPU usage peak does not exceed 55% for any of the cores.

When we explore the files, we can see a correct format with power and frequencies, and files have been created mostly with 35 input vectors.

We can see in the file that the frequency separation is 19.550 kHz which correspond to the formula 20 MHz (sample rate) over 1023 (FFT size minus one).

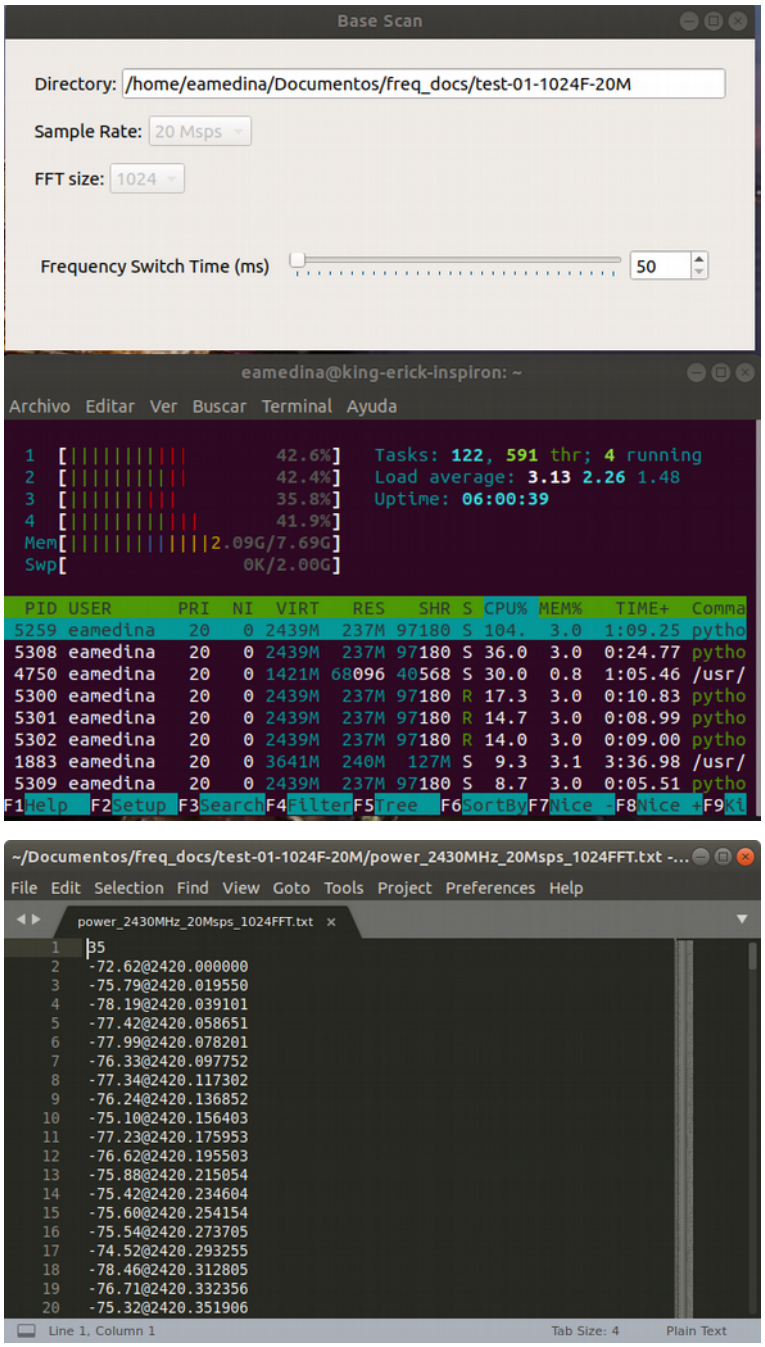


Figure 4.10: File structure created by base script with corresponding index and frequency values for 20 Msps-1024 FFT size-50 ms frequency jump time.

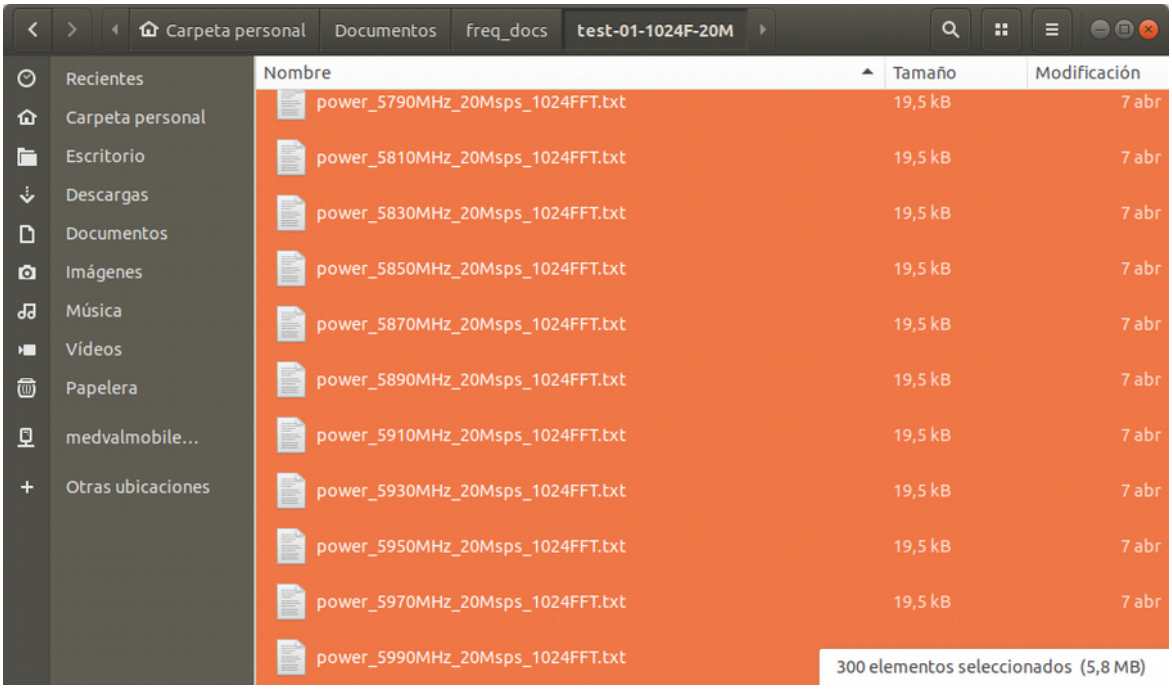


Figure 4.11: Files generated by base script with its names and sizes for 20 Msps-1024 FFT size-50 ms frequency jump time.

4.1.4. Laptop Test: 20Msps 2048FFT 50ms

For this configuration, the script creates 300 files that occupy 11.6 MB, accounting for a size of less than 40 kB each. When the script is running we see it uses around 212% of CPU cores and 2.05 of 7.6 GHz of RAM memory, and the CPU usage peak does not exceed 65% for any of the cores.

When we explore the files, we can see a correct format with power and frequencies, and files have been created mostly with 26 input vectors.

We can see in the file that the frequency separation is 9.770 kHz which correspond to the formula 20 MHz (sample rate) over 2047 (FFT size minus one).

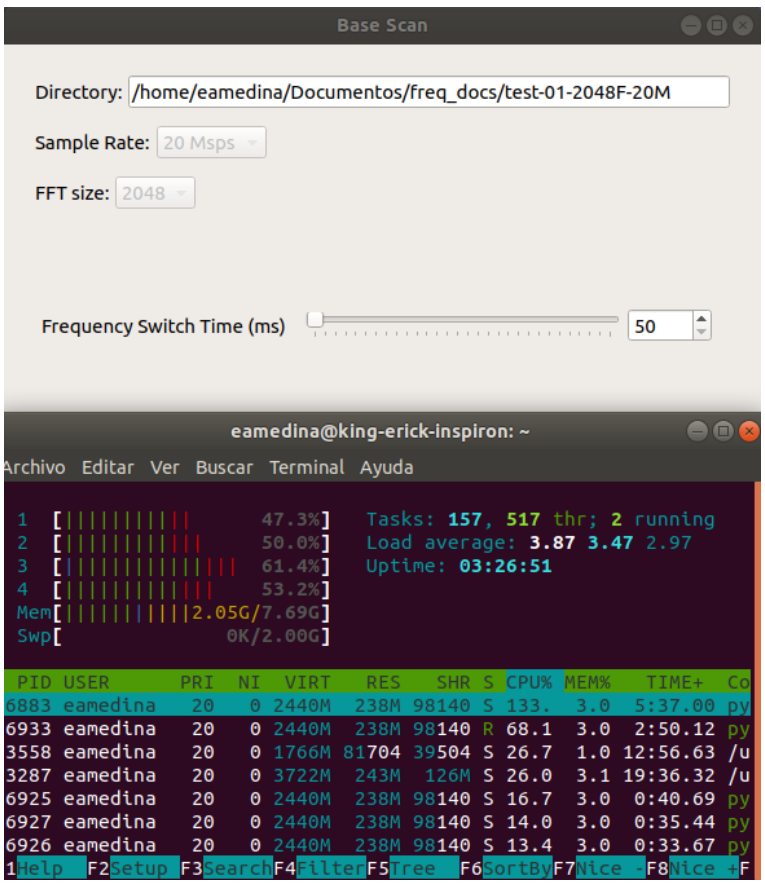


Figure 4.12: Laptop base script parameters configuration for 20 Msps-1024 FFT size-50 ms frequency jump time. CPU consumption below.

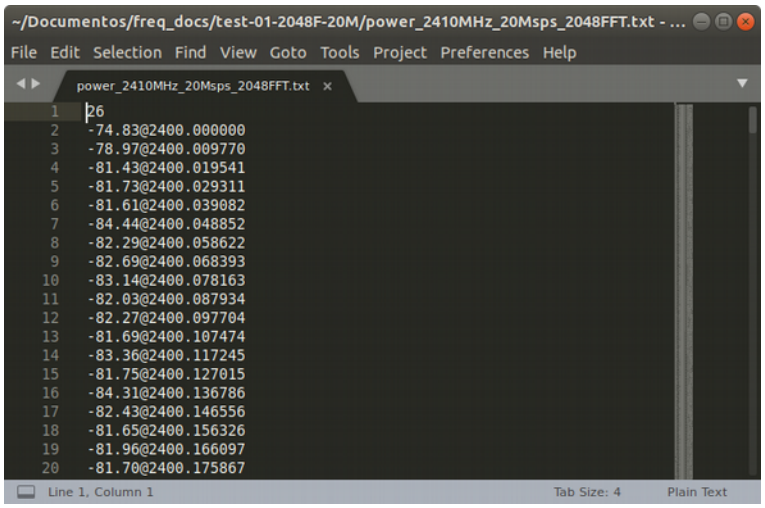


Figure 4.13: File structure created by base script with corresponding index and frequency values for 20 Msps-1024 FFT size-50 ms frequency jump time.

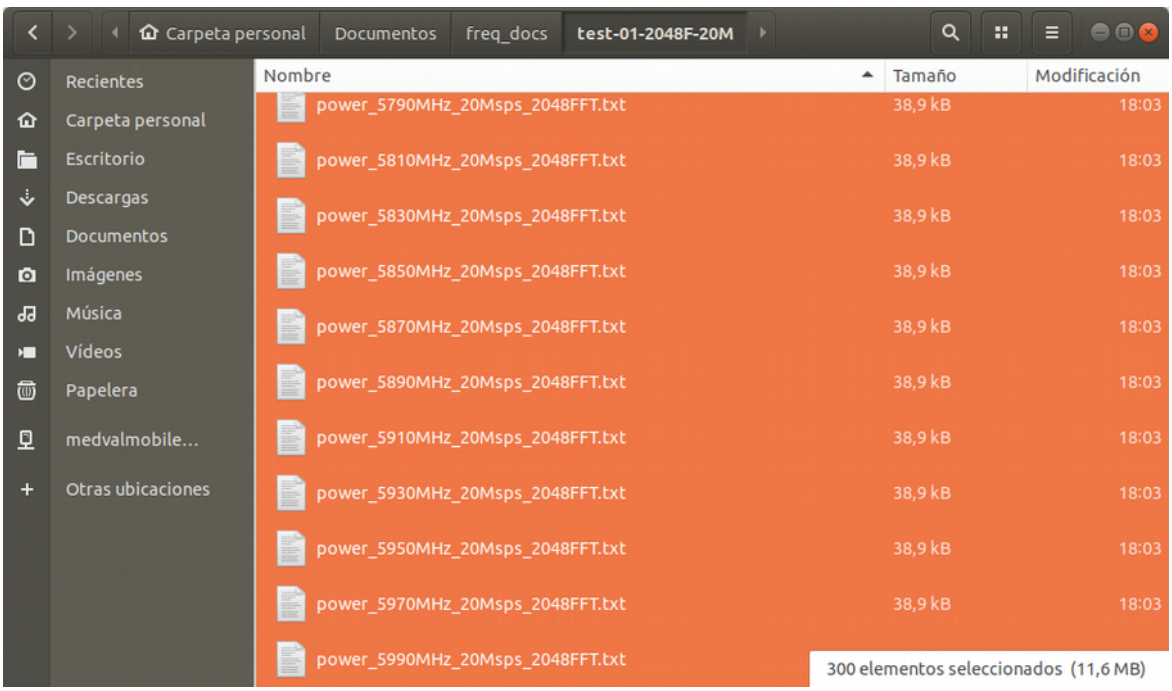


Figure 4.14: Files generated by base script with its names and sizes for 20 Msps-2048 FFT size-50 ms frequency jump time.

4.1.5. Laptop Test Analysis for Base Script

We have proved the base scanning script which is in charge of obtaining the averaged base power values for all the spectrum from 1 MHz to 6 GHz in a time lapse of five minutes. We have verified the creation of the expected number of files, that files have the correct naming format, that the data in it has the corresponding power and frequency values, and the CPU resources it uses while running. In the next table we make a summary of all the identified differences.

Table 5.Base script test results in laptop

	PARAMETER CONFIGURATION			
VALUE	**10 Msps 1024 FFT size 50 ms**	**10 Msps 2048 FFT size 50 ms**	**20 Msps 1024 FFT size 50 ms**	**20 Msps 2048 FFT size 50 ms**
FILES GENERATED	600	600	300	300
TOTAL FILES SIZE (MB)	11.6 MB	23.1 MB	5.8 MB	11.6 MB
MAX. FILE SIZE (kB)	19.5 kB	38.9 kB	19.5 kB	38.9 kB
FILE NAMING FORMAT	power_5995MHz_10Msps_1024FFT	power_5995MHz_10Msps_2048FFT	power_5990MHz_20Msps_1024FFT	power_5990MHz_20Msps_2048FFT
FILE INDEX VALUE	15	15	35	26
FREQUENCY SEPARATION (kHz)	9.775 kHz	4.885 kHz	19.550 kHz	9.770 kHz
% CPU USED	82%	180%	161%	212%
MEMORY USED (GB)	2.0	2.06	2.09	2.05
SPECTRUM SWEEP TIME (seconds)	30 s	30 s	15 s	15 s

The script works accurately as in the number of files created for each configuration. If we divide the 6 GHz in 10 MHz bands we have 600 parts, and dividing it in 20 MHz bands we have 300 parts. This indicates the number of files that must be created, and this number depends only on the sample rate parameter. All configurations accomplish their expected result.

The files generated accomplish the naming format for each configuration, identifying correctly the central frequency, sample rate and FFT size.

We can see differences in the file sizes and the total size for all the files of configuration. This is explained by the fact that we have a different number of files created when the sample rate changes, but also the quantity of information that we store in the file varies greatly with the FFT size. For FFT size 1024, a file contains 1024 frequencies with its corresponding power value, plus the file index at the top. But for FFT size 2048, we will have 2048 power values with its frequency, plus the file index. That's double the information, and we can see which that files with FFT size 2048 is always double the size of files with FFT size 1024. This indicates that the parameter that influences the file size is the FFT size, and that all configurations with same FFT size will have the same file size.

It is important to note that the file size doesn't increase in time, it is because all the power values and frequency values follow the same format with a fixed length, and it is only the index length which varies in time, but its variations don't affect the file size significantly.

As for the total size of the files generated by the script, we can see that the most efficient configuration is with 20 Msps and 1024 FFT size with a value under 6 MB, where all the spectrum information can be contained in 300 files with 1024 lines of power and frequency information plus the index. With this configuration we obtain a ratio of 1 MB of information per each GHz analyzed. The biggest total file size is under 24 MB, for the 10 Msps and 2048 FFT size configuration, which consists of 600 files containing 2048 lines of power and frequency information plus the index. This accounts for a ratio of 4 MB per GHz. We can see that coincidentally the two remaining configurations 10 Msps-1024FFT size and 20Msps-2048FFT size have the same total file size is under 12 MB even though in the first we have 600 files and in the second we have 300 files, but this can be explained since files in the second configuration carry the double of information than in the first one. The ratio for both these configurations is 2 MB of information per each GHz of spectrum.

The file index indicates the number of times a new input vector has been used to calculate the power averages. And this value can be used to determine the number of times a value was received for a given frequency band, helping us determine the times the script made a total spectrum scan. Even though the index can be different for some files, most files have an index value that repeats more frequently than others. For a sample rate of 10 Msps, with a frequency jump time of 50 ms, we expect a total spectrum scan time of 30 seconds. And in five minutes we expect to have at least ten total spectrum scans. But in reality we can see that for both configurations, the most common index is 15, which indicates at least 15 spectrum scans.

For a sample rate of 20 Msps, with a frequency jump time of 50 ms, we expect a total spectrum scan time of 15 seconds. For the five minutes of scan we expect to have 20 total spectrum scans but we can see that with 1024 FFT size we have an index of 35, and for 2048 FFT size we have an index of 26. These differences can be explained by the fact that in those 50 ms in which the Hack RF is scanning a given center frequency, it can feed the power analyzer block with more than one data vector, resulting in a greater value than the expected. As this situation depends on the HackRF and the computation power of the laptop, we can have different index values in different frequency files within the same configuration.

The frequency separation is the value between two adjacent frequencies after the FFT transformation, and it is different for all configurations. We see that we obtain the greatest separation at 20Msps-1024 FFT size, with 19.550 kHz between frequency values, and the minimum at 4.885 kHz for 10Msps-2048FFT size.

We wanted to check the performance of the laptop while running the script. Even though we only obtained snapshots in a given time of the CPU performance, this helped us identify the configurations that require more computation power. We can assume that configurations with 2048 FFT size, will require more power, since they make the double of operations than in 1024 FTT size. This assumption is proved in the results obtained, since 10Msps-2048FFT uses 180% while 10Msps-1024FFT only 80%. The same is observed in the other configurations 20Msps-2048FFT uses 212% against 10Msps-1024FFT with 161%. It is necessary to note that the use percentage includes all processes run in the laptop at that time, and that all tests were performed under the same circumstances.

We can see that the usage of RAM memory doesn't vary greatly among the different configurations, and this can be explained by the fact that the only difference in memory usage between them is that the 2048 FFT loads double the information in memory than 1024 FFT, but it is negligible if we compare them in the GB order of magnitude.

Of all the four different configurations used to test the base scan script we can see that we can get better results using 20 Msps and 1024 FFT size. It gives us the smallest total file size for all the information with 5.8 MB, it also gave us the most total spectrum scans in five minutes with 35. It is also has a better CPU performance compared to configurations with 2048 FFT sizes.

4.1.6. Raspberry Test: 10Msps 1024FFT 500ms

Initially the tests for this script in Raspberry were run using 50 ms. as the frequency switch time, but we saw that due to the lower CPU power, less than 50% of the files were created at the end of the five minutes. Even though the total scan time was increased from five minute to ten minutes, it was not guaranteed that all frequencies would be scanned. So we opted to increase the frequency switch time from 50 ms. to 100 ms., then to 200 ms. and finally 500 ms. as the time in which we saw that all frequencies were analyzed during one full spectrum scan. Logically, the total time for the test had to be increased, since each total spectrum scan takes now 300 seconds or five minutes. So we defined the new total time for the test at 15 minutes.

This script's test results are similar to the obtained in the laptop: 600 files are created with a total size of 11.6 MB and each file's maximum size is 20 kB. The script is run altogether with the CPU monitoring tool *htop*, and we see a rapidly variant value of the CPU core usage percentage, however we obtain a snapshot value of 174% and 677 MB of RAM memory occupied.

The script generates correctly the files with its corresponding power and frequency values, and the frequency separation corresponds to its configuration, but we see that the file index values can be different for various files, so we won't take it into account for further analysis.

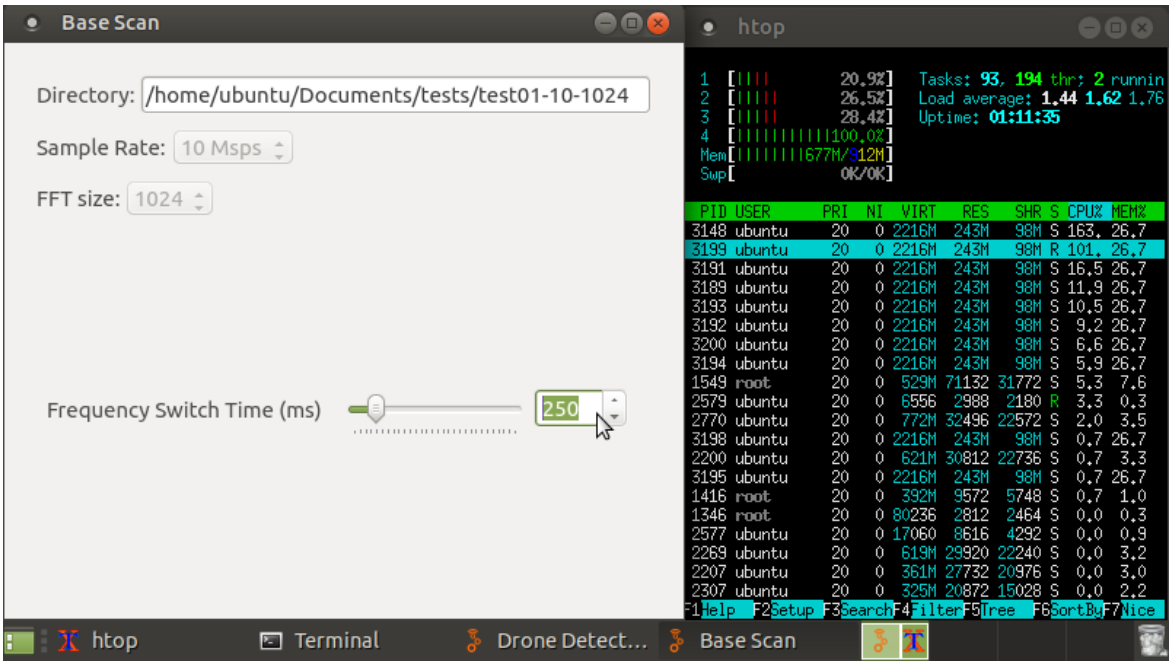


Figure 4.15: Raspberry base script parameters configuration for 10 Msps-1024 FFT size frequency jump time. CPU consumption to the right.

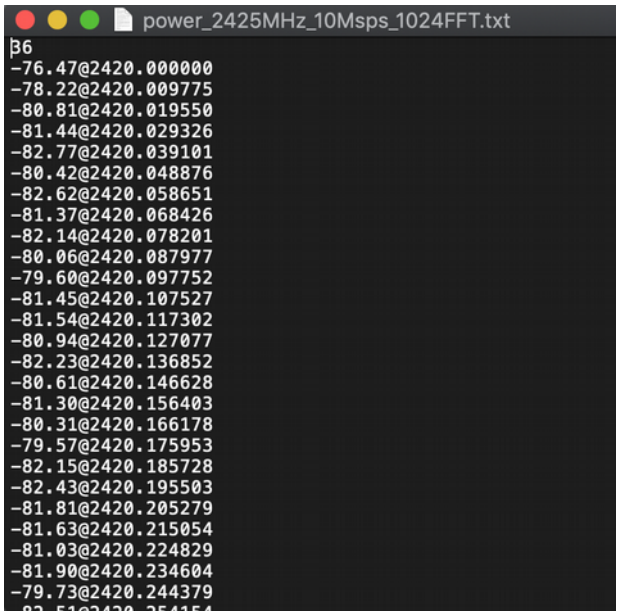


Figure 4.16: File structure created by base script with corresponding index and frequency values for 10 Msps-1024 FFT size.

4.1.7. Raspberry Test: 10Msps 2048FFT 500ms

This script's test results are similar to the obtained in the laptop: 600 files are created with a total size of 23.1 MB and each file's maximum size is under 40 kB. The script is run altogether with the CPU monitoring tool *htop*, and we see a rapidly variant value of the CPU core usage percentage, but we obtain a snapshot value of 163% CPU usage and 674 MB of RAM memory occupied.

The script generates correctly the files with its corresponding power and frequency values, and the frequency separation corresponds to its configuration, but we see that the file index values can be different for various files, so we won't take it into account.

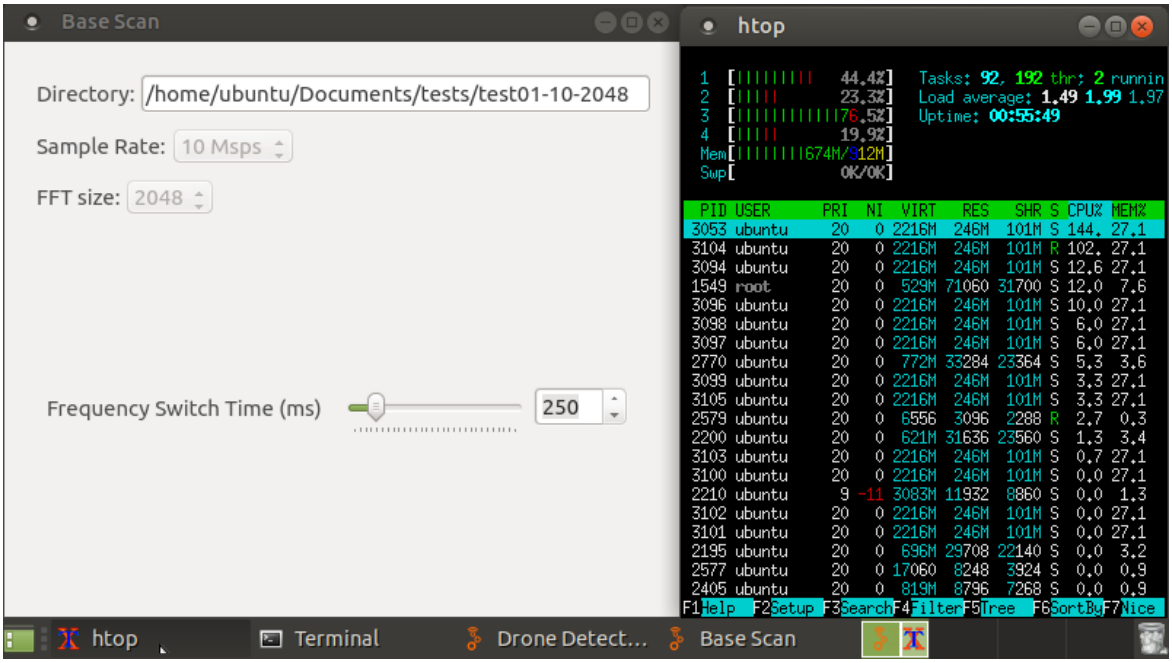


Figure 4.17: Raspberry base script parameters configuration for 10 Msps-2048 FFT size frequency jump time. CPU consumption to the right.

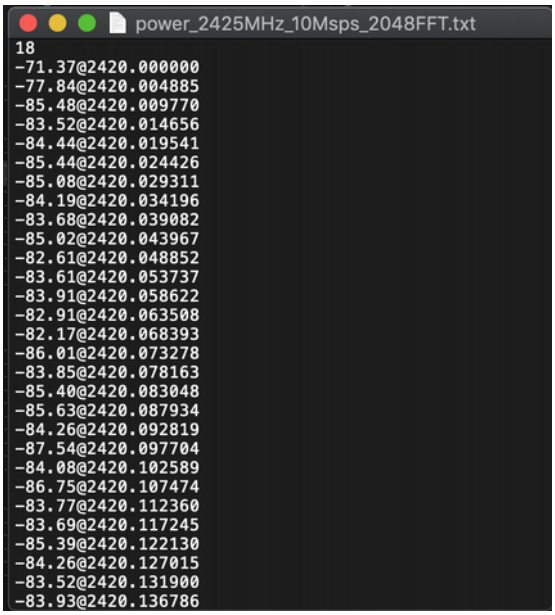


Figure 4.18: File structure created by base script with corresponding index and frequency values for 10 Msps-2048 FFT size.

4.1.8. Raspberry Test: 20Msps 1024FFT 500ms

This script's test results are similar to the obtained in the laptop: 300 files are created with a total size of 11.6 MB and each file's maximum size is under 40 kB. The script is run altogether with the CPU monitoring tool *htop*, and we see a rapidly variant value of the CPU core usage percentage, but we obtain a snapshot value of 241% CPU usage and 666 MB of RAM memory occupied.

The script generates correctly the files with its corresponding power and frequency values, the frequency separation corresponds to its configuration, but we see that the file index values can be different for various files, so we won't take it into account.

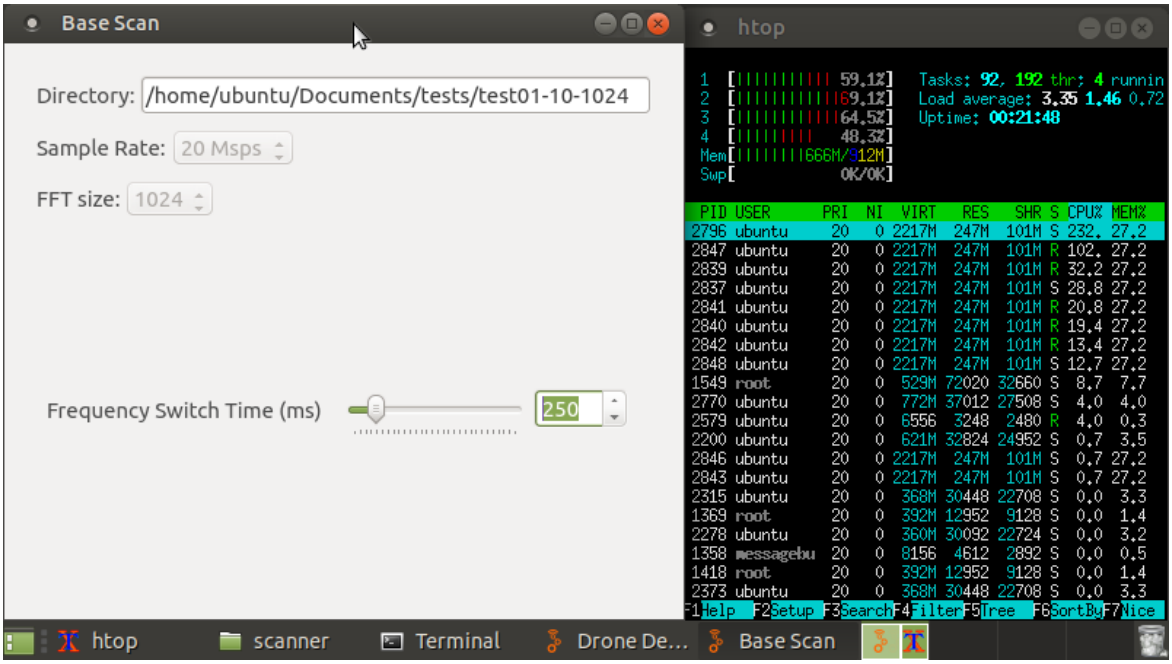


Figure 4.19: Raspberry base script parameters configuration for 20 Msps-1024 FFT size frequency jump time. CPU consumption to the right.

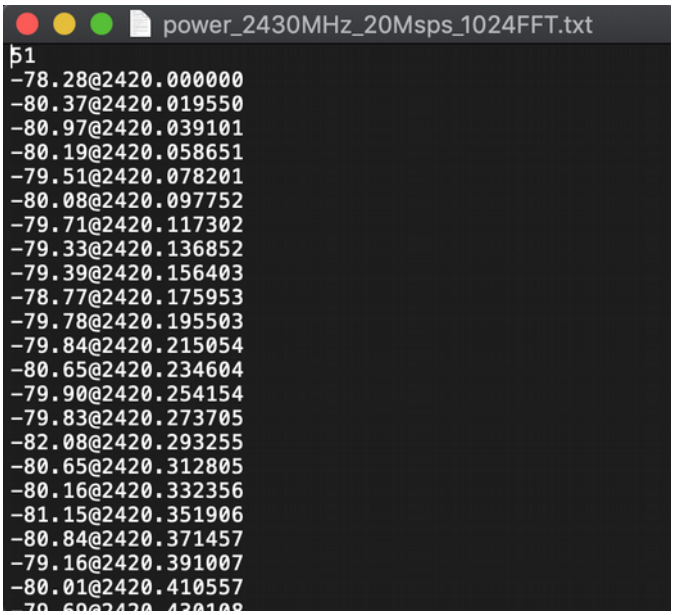


Figure 4.20: File structure created by base script with corresponding index and frequency values for 20 Msps-1024 FFT size.

4.1.9. Raspberry Test: 20Msps 2048FFT 500ms

This script's test results are similar to the obtained in the laptop: 300 files are created with a total size of 5.8 MB and each file's maximum size is under 20 kB. The script is run altogether with the CPU monitoring tool *htop*, and we see a rapidly variant value of the CPU core usage percentage, but we obtain a snapshot value of 222% CPU usage and 673 MB of RAM memory occupied.

The script generates correctly the files with its corresponding the power and frequency values, the frequency separation corresponds to its configuration, but we see that the file index values can be different for various files, so we won't take it into account.

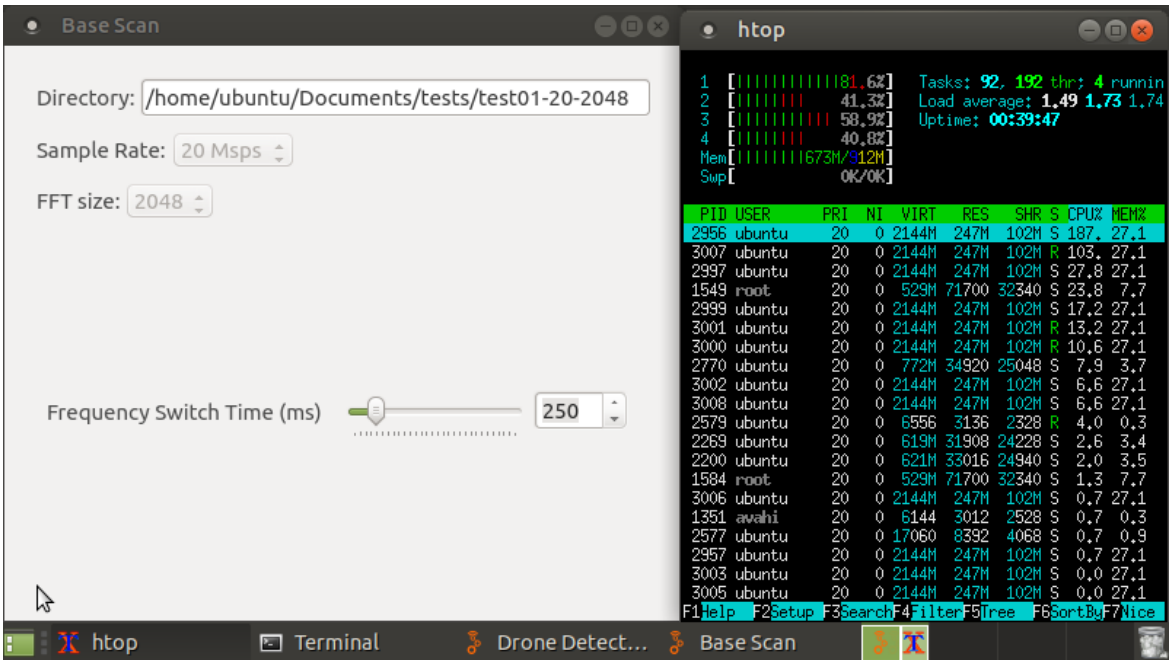


Figure 4.21: Raspberry base script parameters configuration for 20 Msps-2048 FFT size frequency jump time. CPU consumption to the right.

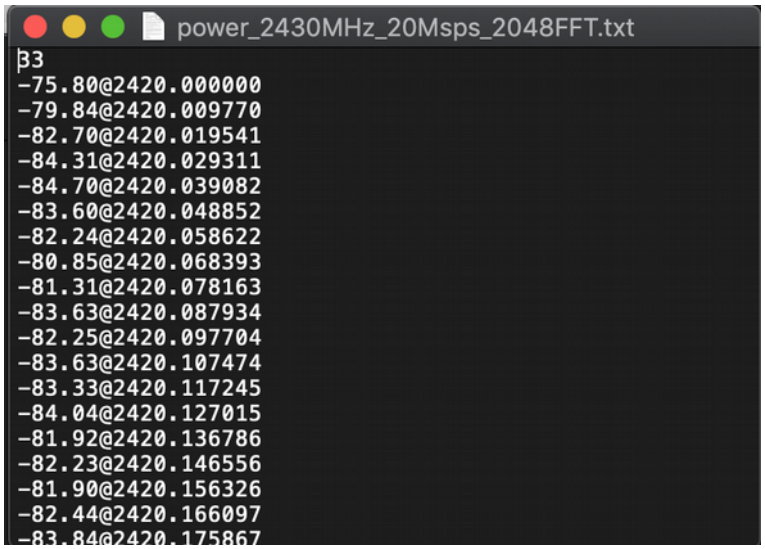


Figure 4.22: File structure created by base script with corresponding index and frequency values for 20 Msps-2048 FFT size.

4.1.10. Raspberry Test Analysis for Base Script

While performing the test we see that the Raspberry suffers from glitches reflected in the responsiveness of the user interface, some times freezing it for a few seconds. We have experienced with the Raspberry that other activities like opening the internet browser or opening a text editor usually takes some significant time and often leaves the user interface unresponsive. So, it is expected that the execution of our script causes some noticeable delays or temporal blocks.

We have detected also that the execution of the CPU monitoring tool *htop* affects directly the performance of the Raspberry and should not be recommended to be executed along the scripts for the following tests.

The most important result we obtain is that the script can indeed be executed in the Raspberry and that we can get the base powers database that we need to perform the tests of the spectrum and band scan script.

All the four parameters configurations have worked correctly and the obtained result for files generated and occupied space by the database are equal to the laptop results as expected.

The main difference is the frequency switch time which in this case is ten times that of the laptop tests, going from 50 ms to 500 ms. This increase in time has assured us that all frequencies can be analyzed and therefore their corresponding files are created successfully.

Finally we can reach to the same conclusion than in the laptop test, by selecting the 20 Msps and 1024 FFT size as the most efficient configuration as per the total number of files generated and its total size occupied.

4.1.11. Comparison of results from laptop and Raspberry

To analyze the resulting data obtained from the tests in laptop and Raspberry, we generate a graphic to compare the obtained values at a specific band, and we choose the 433 MHz band. These graphics are obtained using a custom Python function, created only for the purpose of generating the graphics. We can't obtain this type of graphics directly from our created tool, because the data of the Raspberry and the laptop tests are found in different directories.

We can see that results are similar in general, but we see smoother values in Raspberry and peakier values in laptop results. This difference is due to the fact that more time for the total test was given in Raspberry, and also a greater frequency switching time was configured, giving the chance that for every frequency we had more values to be averaged. Instead, in the laptop test a lower time for the total test was assigned, and the quicker frequency switch time results in less values to be received, hence the averaged value will be affected by any peak. For 10 Msps tests in laptop we have a 30 seconds time for a total 6 GHz scan, while in Raspberry we have five minutes. And for 20 Msps tests in laptop it takes 15 seconds for a total 6 GHz scan, while in Raspberry it takes two and a half minutes.

In conclusion, we see that this script has generated base power values that are similar even though the tests were done in different times, and that these values can be used to compare the real time values obtained with the other scanning scripts.

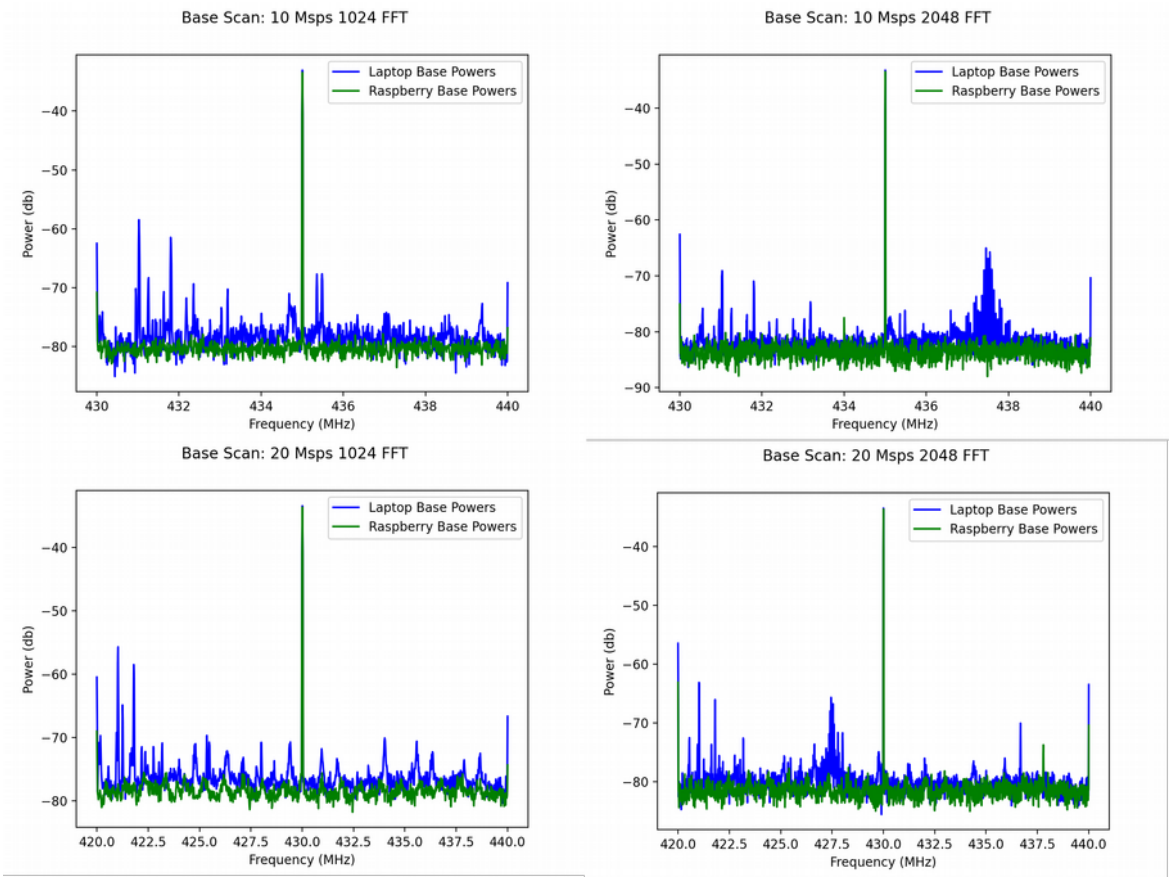


Figure 4.23: Comparison of results obtained in laptop and Raspberry for base power values generated by base scan script.

4.2. Spectrum Scan Script Tests

Our main focus in these tests is to verify that this script creates the corresponding file database, and we will also monitor the system performance in the executing machine. We will try four different parameter combinations with 1024 and 2048 as the FFT size and 10 and 20 Msps as the sample rate, and each combination will have its own directory.

We will execute the script for five minutes for each combination in the laptop and fifteen minutes in the Raspberry with different frequency switch times. For laptops, the frequency switch time is defined in 250 ms, and we set the operation mode to fixed with a threshold of 10 dBm from the base.

Additionally we will use the Why Evo and the toy remote controllers to generate signals in 27 MHz and 433 MHz respectively, to verify if generated signals can be detected by the script. These generators will be a distance under two meters away from the HackRF.

It is important to state, that the folders chosen to run this tests, have to be the same where the base script generated its respective files, since they will be used to be compared with the real time data.

The setup for this test consist of the laptop or Raspberry running the script and the HackRF One connected to it via USB cable. Also the remote controllers will be generating signals in 27 MHz and 433 MHz. All tests will have the same setup, since the only difference among them will be the parameters configured in the script. The location of this tests is in an indoor environment.

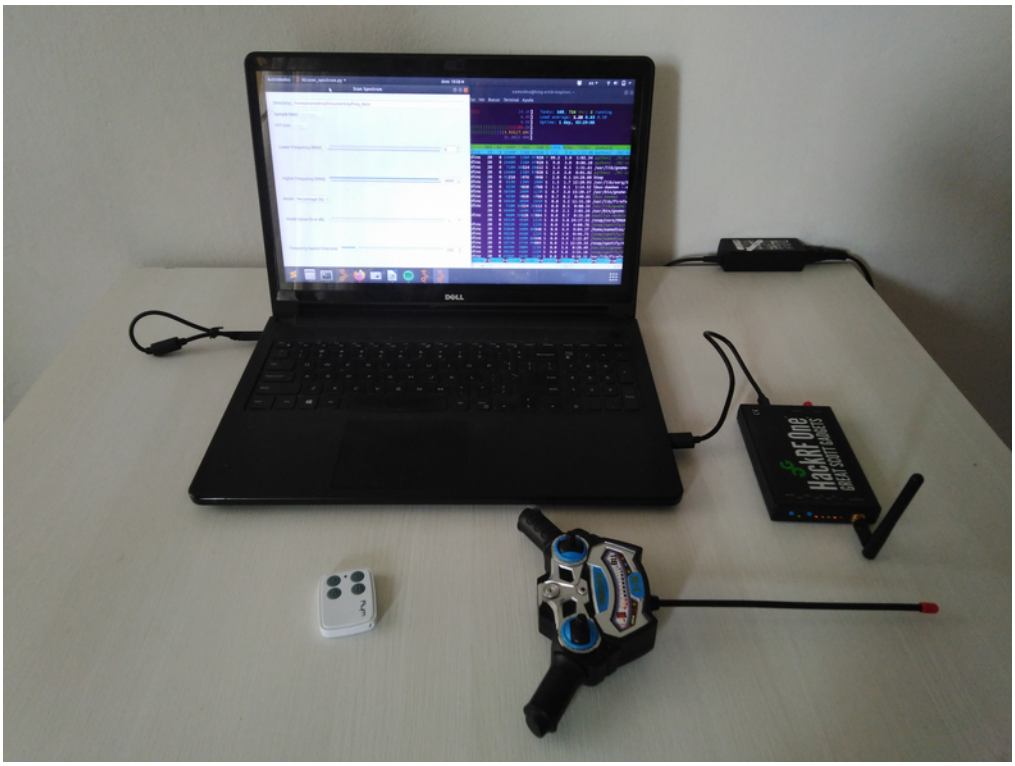


Figure 4.24: Laptop spectrum scan script test setup

Figure 4.25: Raspberry spectrum scan script test setup

4.2.1. Laptop Test: 10Msps 1024FFT 250ms

For this configuration, the script creates 600 files that occupy 22.4 MB, accounting for a size of less than 40 kB each. When the script is running we see it uses around 110% of all CPU cores and 1.69 of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 40% for any of the cores individually.

When we explore the files, we can see a correct format with differences, averages, minimums and maximums with their respective frequencies, and files have been created mostly with 16 input vectors. We can see in the file that the frequency separation is 9.775 kHz.

To check if the script has detected unusual activity in 27 MHz and in 433 MHz we open their respective files:

compare_25MHz_10Msps_1024FFT

compare_435MHz_10Msps_1024FFT

For the first file we see that a peak was detected around 27.15 MHz, with a value 44 dB above the threshold. For the second file we see a peak detected around 433.92 MHz, with a value 55 dB above the threshold. We can confirm that the script has effectively detected the signals that we have generated for this test.

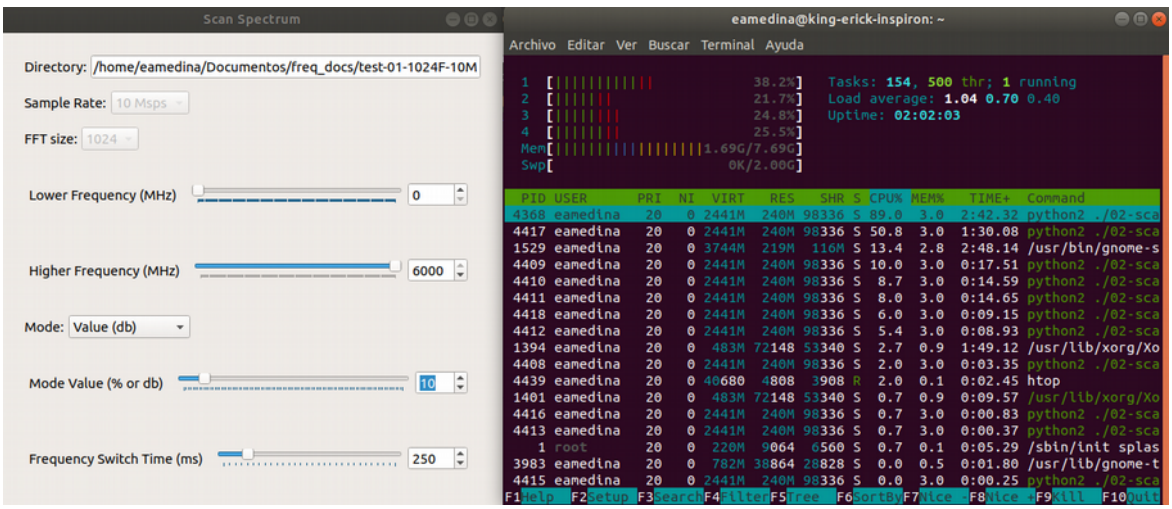


Figure 4.26: Laptop spectrum script parameters configuration for 10 Msps-1024 FFT size. CPU consumption in the right.

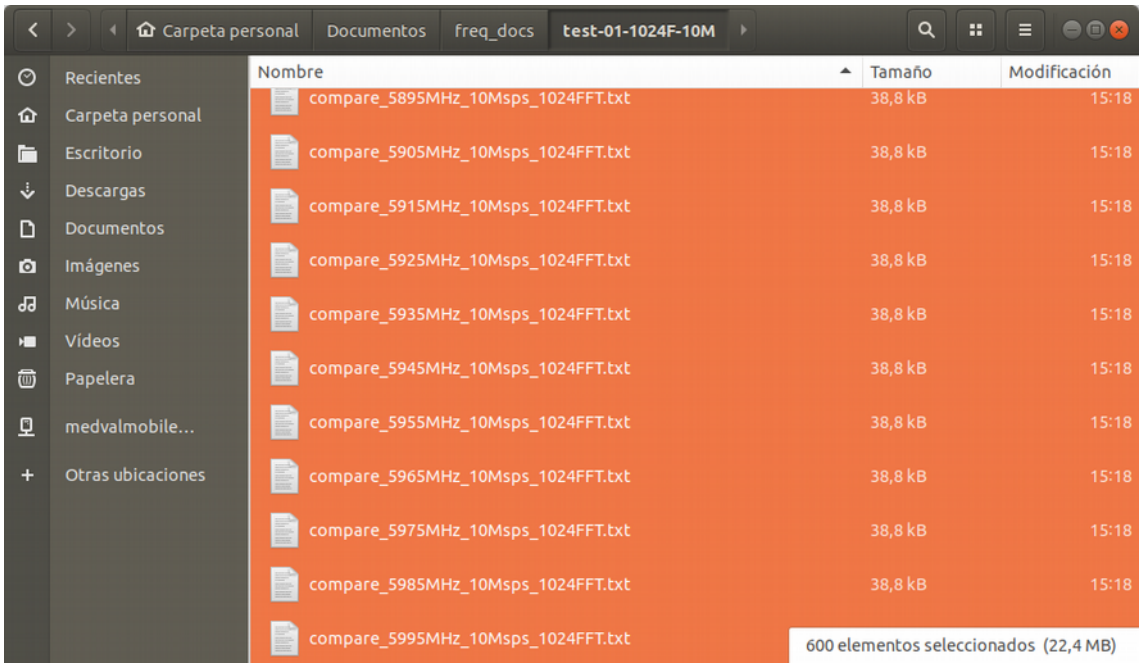


Figure 4.27: Files generated by spectrum script with its names and sizes for 10 Msps-1024 FFT size.

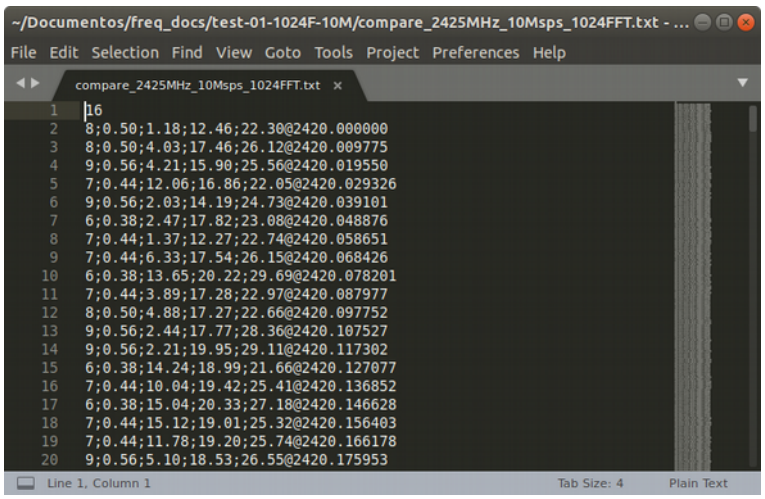


Figure 4.28: File structure created by spectrum script with corresponding index and frequency with differences values for 10 Msps-1024 FFT size.

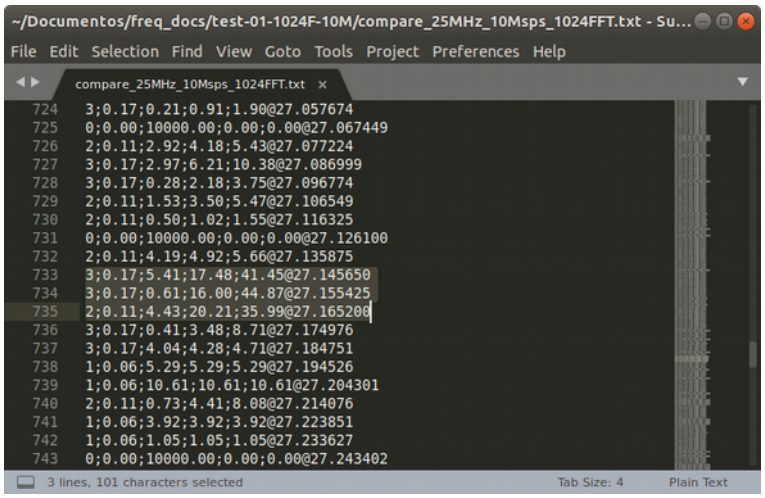


Figure 4.29: File data with values of a peak detected in 27 MHz with spectrum script for 10 Msps-1024 FFT size.

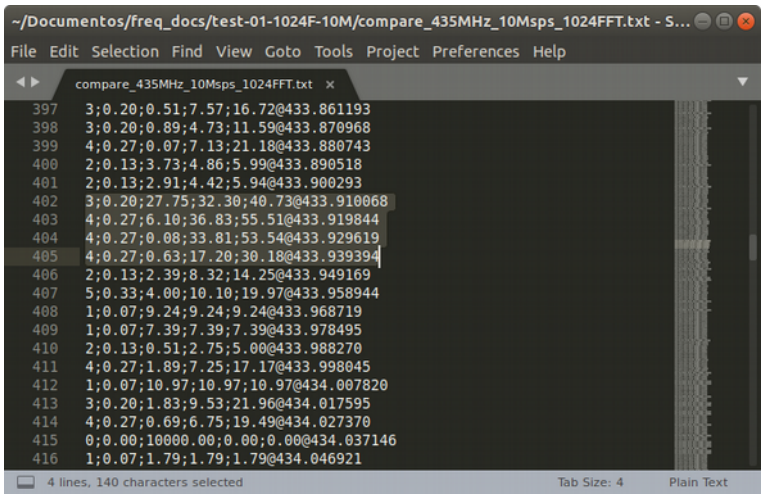


Figure 4.30: File data with values of a peak detected in 433 MHz with spectrum script for 10 Msps-1024 FFT size.

4.2.2. Laptop Test: 10Msps 2048FFT 250ms

For this configuration, the script creates 600 files that occupy 46 MB, accounting for a size of less than 78 kB each. When the script is running we see it uses around 132% of all CPU cores and 1.80 of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 40% for any of the cores individually.

When we explore the files, we can see a correct format with differences, averages, minimums and maximums with their respective frequencies, and files have been created

mostly with 16 input vectors. We can see in the file that the frequency separation is 4.885 kHz.

To check if the script has detected unusual activity in 27 MHz and in 433 MHz we open their respective files:

compare_25MHz_10Msps_2048FFT

compare_435MHz_10Msps_2048FFT

For the first file we see that a peak was detected around 27.22 MHz, with a value 32 dB above the threshold. For the second file we see a peak detected around 433.92 MHz, with a value 60 dB above the threshold. We can confirm that the script has effectively detected the signals that we have generated for this test.

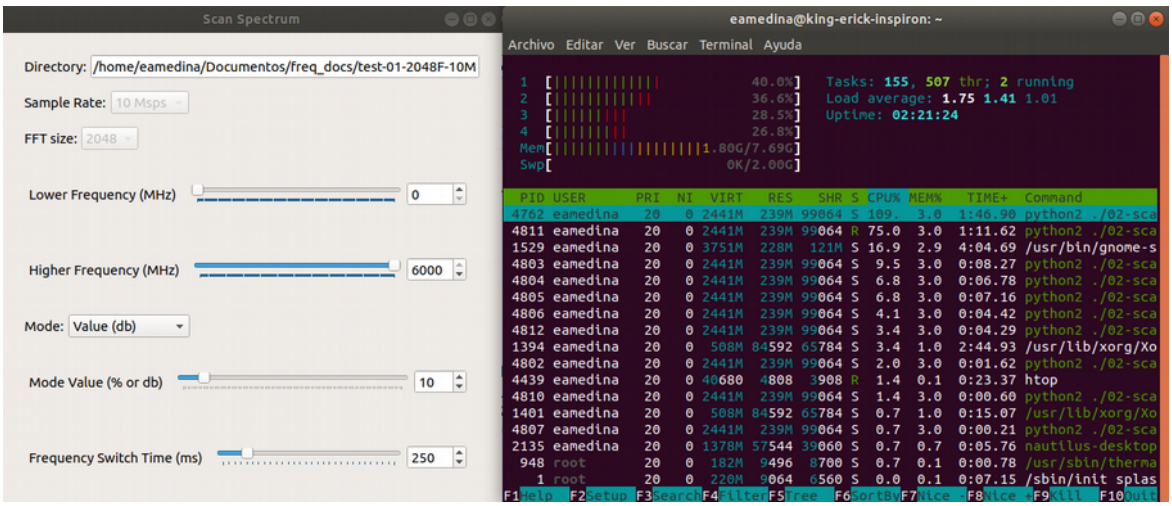


Figure 4.31: Laptop spectrum script parameters configuration for 10 Msps-2048 FFT size. CPU consumption in the right.

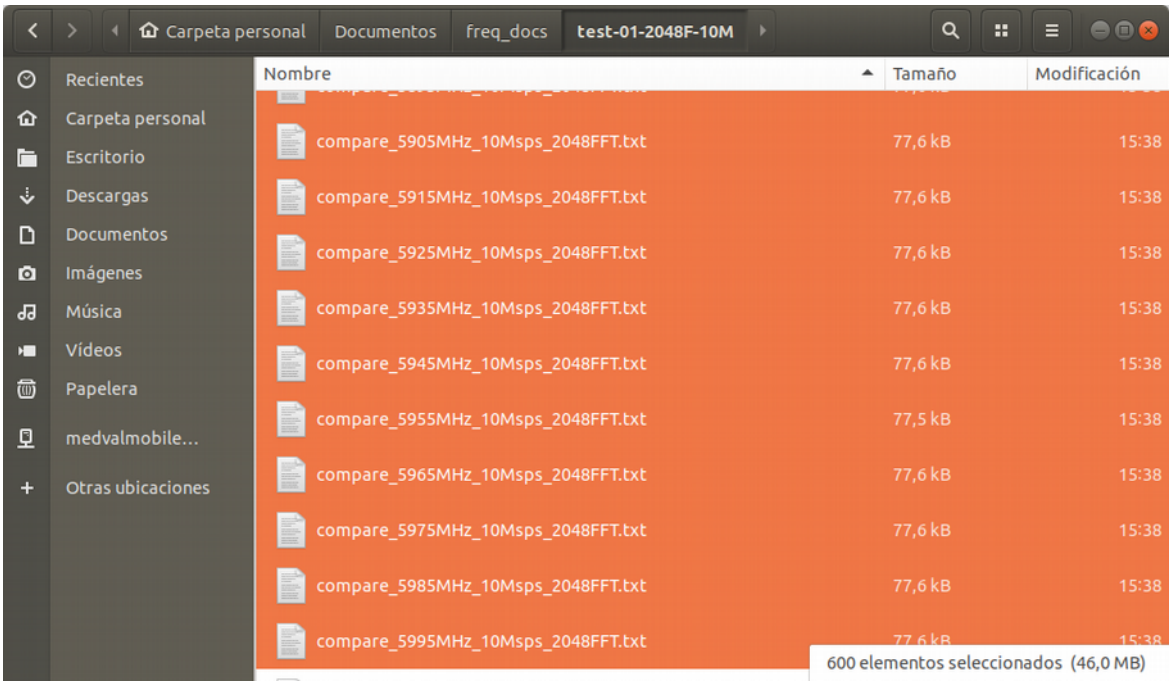


Figure 4.32: Files generated by spectrum script with its names and sizes for 10 Msps-2048 FFT size.

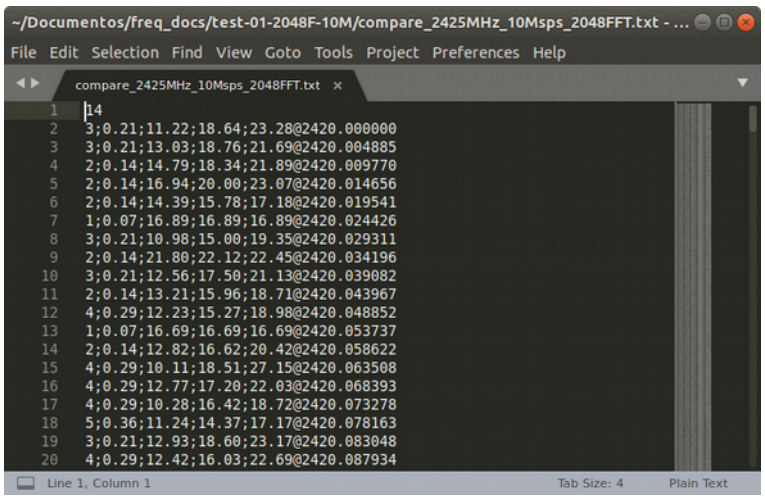


Figure 4.33: File structure created by spectrum script with corresponding index and frequency with differences values for 10 Msps-2048 FFT size.

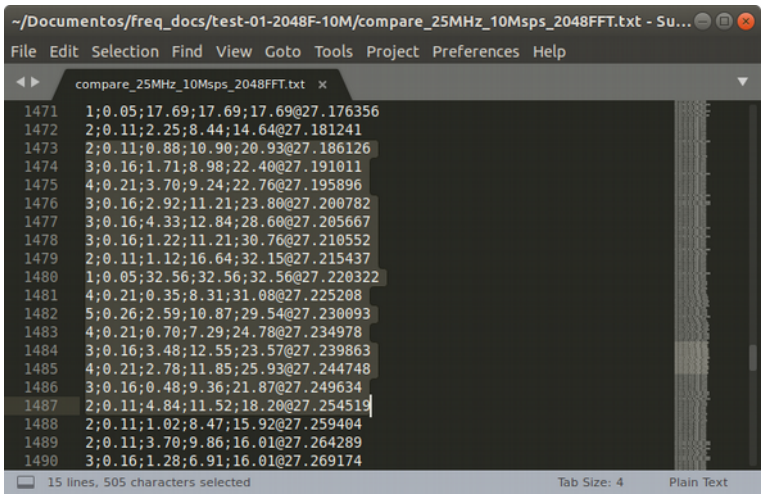


Figure 4.34: File data with values of a peak detected in 27 MHz with spectrum script for 10 Msps-2048 FFT size.

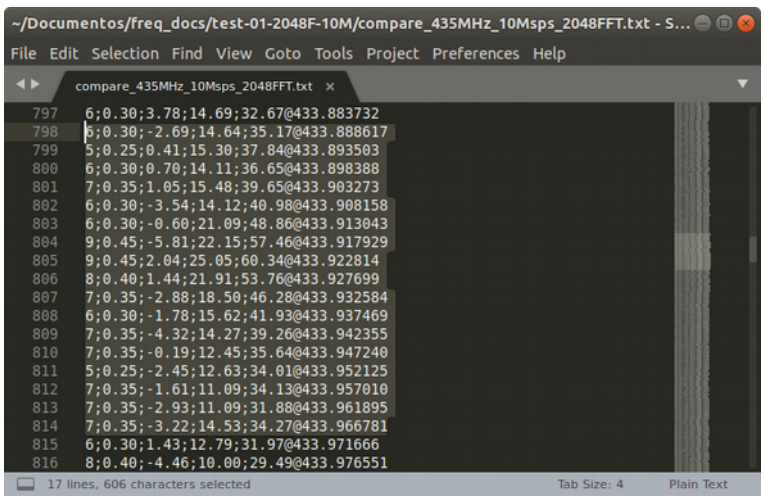


Figure 4.35: File data with values of a peak detected in 433 MHz with spectrum script for 10 Msps-2048 FFT size.

4.2.3. Laptop Test: 20Msps 1024FFT 250ms

For this configuration, the script creates 300 files that occupy 11.4 MB, accounting for a size of less than 40 kB each. When the script is running we see it uses around 123% of all CPU cores and 1.81 out of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 35% for any of the cores individually.

When we explore the files, we can see a correct format with differences, averages, minimums and maximums with their respective frequencies, and files have been created mostly with 31 input vectors. We can see in the file that the frequency separation is 19.550 kHz.

To check if the script has detected unusual activity in 27 MHz and in 433 MHz we open their respective files:

compare_30MHz_20Msps_1024FFT

compare_430MHz_20Msps_1024FFT

For the first file we see that a peak was detected around 27.17 MHz, with a value 31 dB above the threshold. For the second file we see a peak detected around 433.93 MHz, with a value 50 dB above the threshold. We can confirm that the script has effectively detected the signals that we have generated for this test.

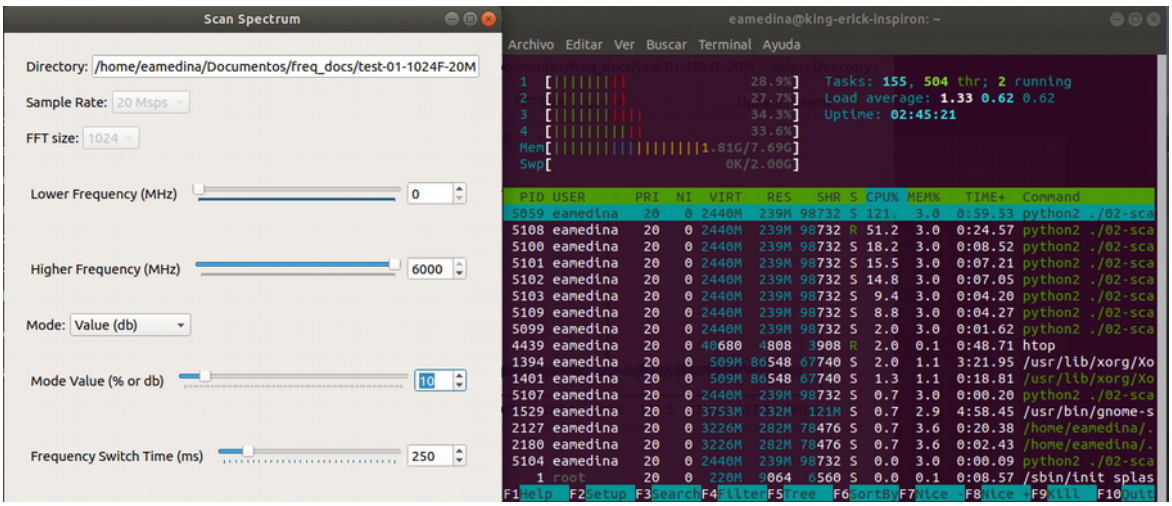


Figure 4.36: Laptop spectrum script parameters configuration for 20 Msps-1024 FFT size. CPU consumption in the right.

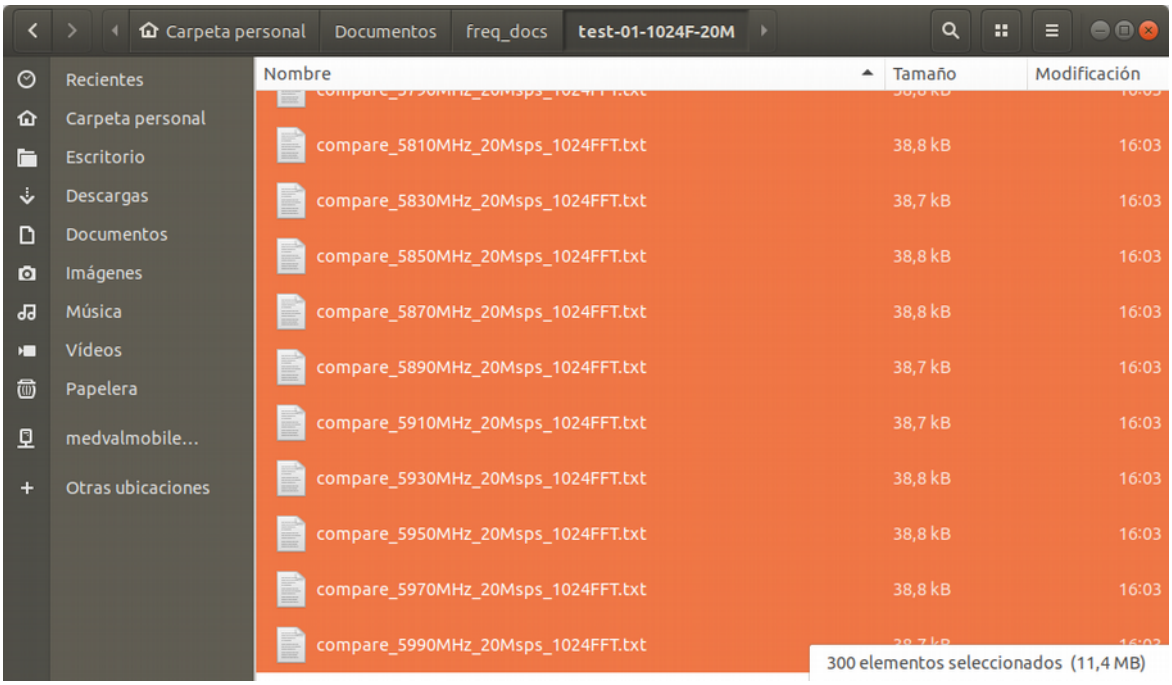


Figure 4.37: Files generated by spectrum script with its names and sizes for 20 Msps-1024 FFT size.

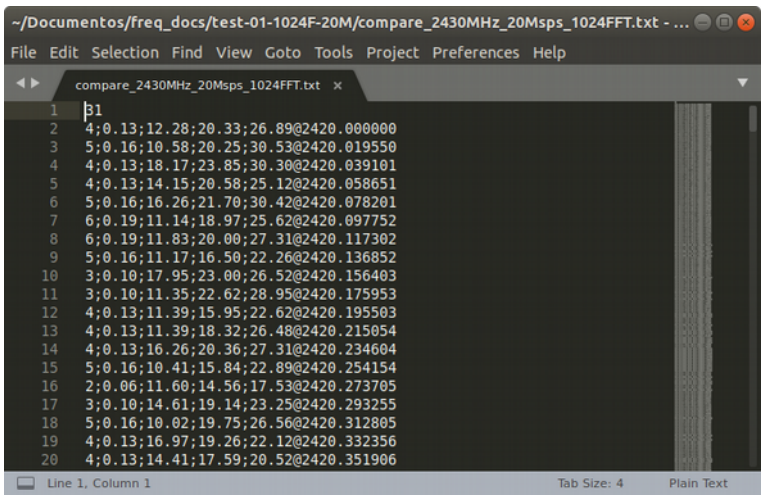


Figure 4.38: File structure created by spectrum script with corresponding index and frequency with differences values for 20 Msps-1024 FFT size.

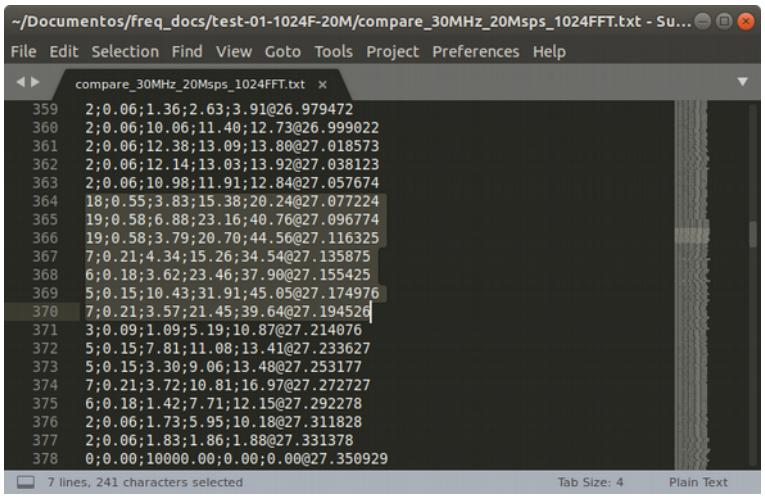


Figure 4.39: File data with values of a peak detected in 27 MHz with spectrum script for 20 Msps-2048 FFT size.

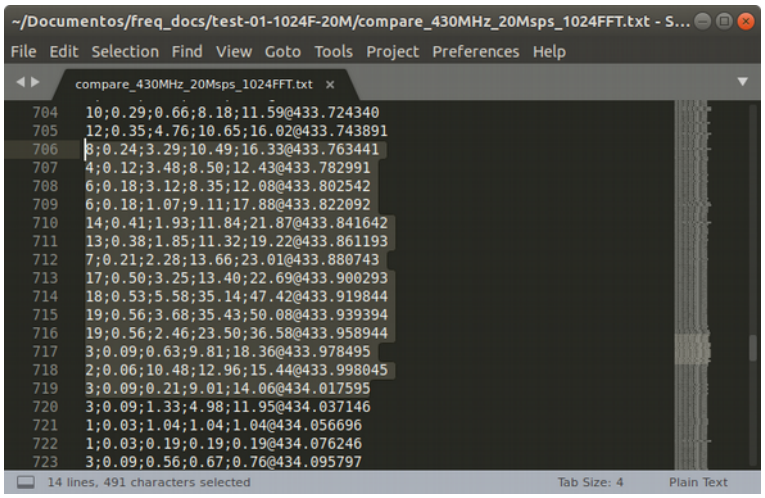


Figure 4.40: File data with values of a peak detected in 433 MHz with spectrum script for 20 Msps-2048 FFT size.

4.2.4. Laptop Test: 20Msps 2048FFT 250ms

For this configuration, the script creates 300 files that occupy 22.9 MB, accounting for a size of less than 78 kB each. When the script is running we see it uses around 215% of all CPU cores and 2.05 out of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 65% for any of the cores individually.

When we explore the files, we can see a correct format with differences, averages, minimums and maximums with their respective frequencies, and files have been created

mostly with 30 input vectors. We can see in the file that the frequency separation is 9.770 kHz.

To check if the script has detected unusual activity in 27 MHz and in 433 MHz we open their respective files:

compare_30MHz_20Msps_2048FFT

compare_430MHz_20Msps_2048FFT

For the first file we see that a peak was detected around 27.09 MHz, with a value 26 dB above the threshold. For the second file we see a peak detected around 433.92 MHz, with a value 53 dB above the threshold. We can confirm that the script has effectively detected the signals that we have generated for this test.

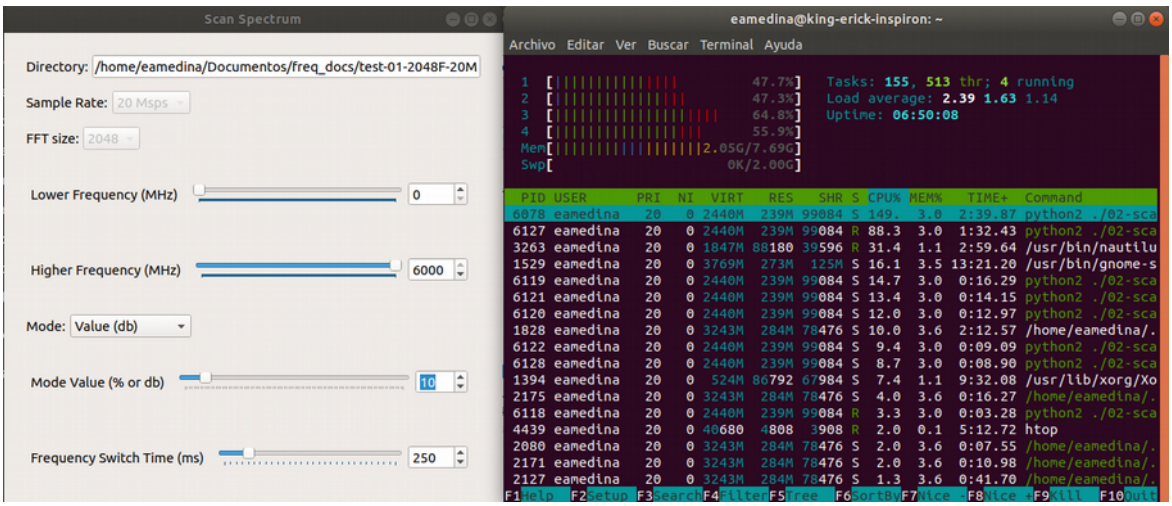


Figure 4.41: Laptop spectrum script parameters configuration for 20 Msps-2048 FFT size. CPU consumption in the right.

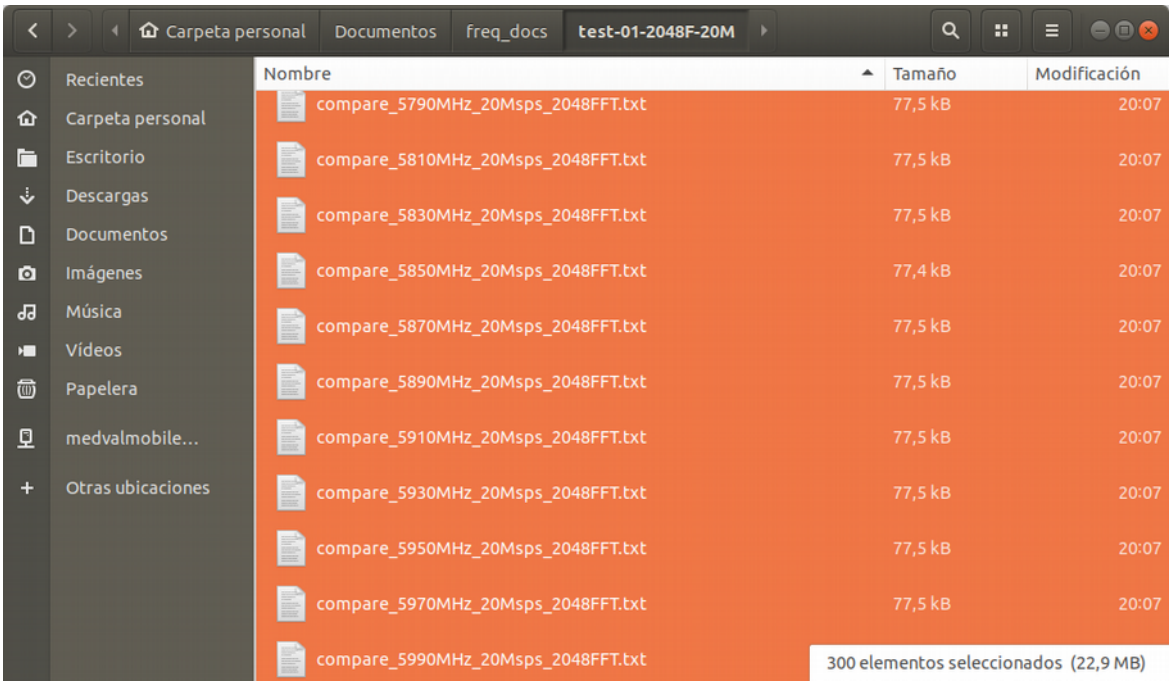


Figure 4.42: Files generated by spectrum script with its names and sizes for 20 Msps-2048 FFT size.

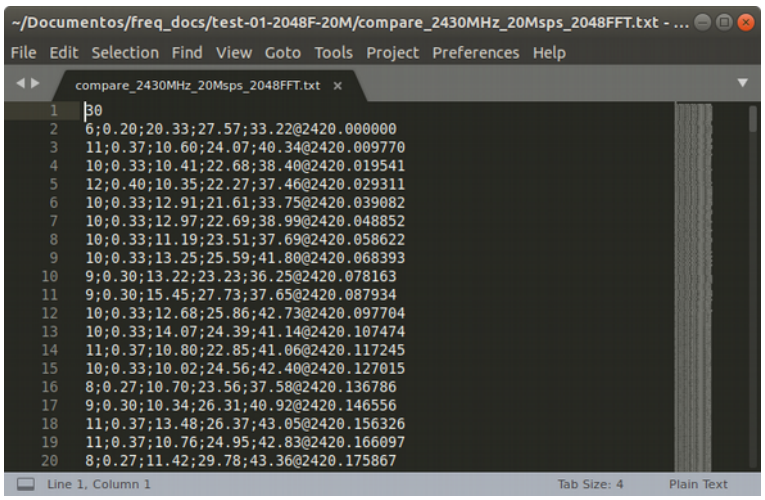


Figure 4.43: File structure created by spectrum script with corresponding index and frequency with differences values for 20 Msps-2048 FFT size.

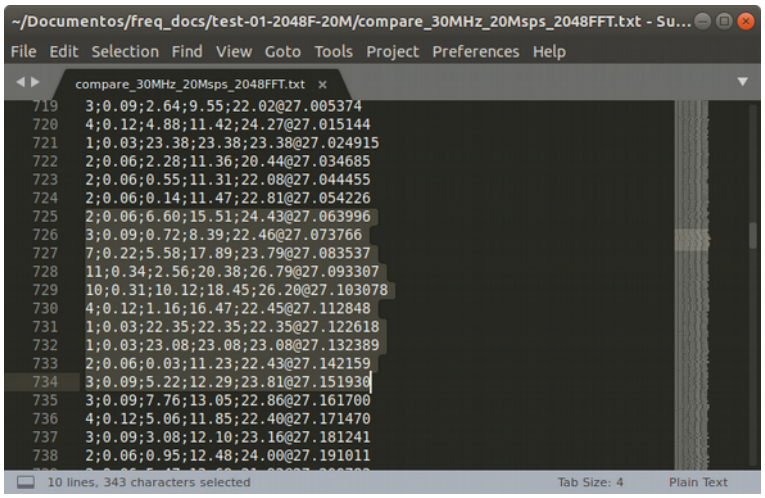


Figure 4.44: File data with values of a peak detected in 27 MHz with spectrum script for 20 Msps-2048 FFT size.

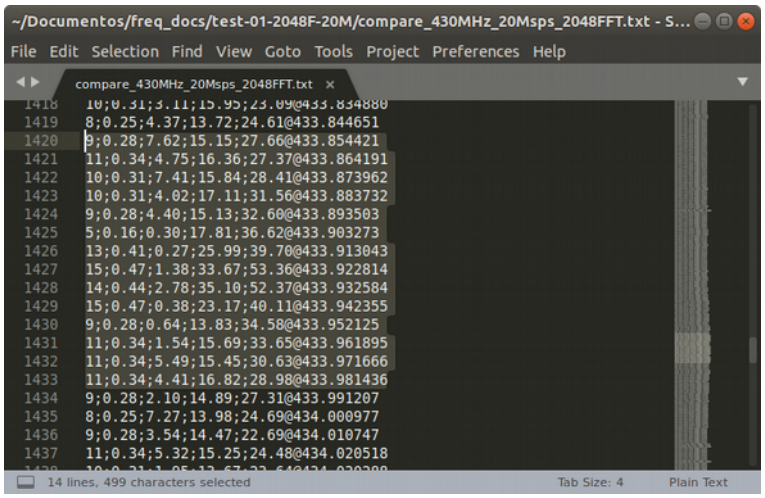


Figure 4.45: File data with values of a peak detected in 27 MHz with spectrum script for 20 Msps-2048 FFT size.

4.2.5. Laptop Test Analysis for Spectrum Script

We have tested the spectrum scan script which is in charge of comparing the real time values obtained from the HackRF against the averaged base power values obtained with the base scan script.

Even though, the UI controls of the script allow to modify the frequencies at which the scan occurs, we have made the tests with all the spectrum from 1 MHz to 6 GHz in a time lapse of five minutes for each configuration. We have set the comparison to fixed value mode with the threshold value to 10 dB above the base power value.

We have verified the creation of the expected number of files; that files have the correct naming format, that the data in it has the corresponding power differences, averages and frequency values, and the CPU resources it uses while running. In the next table we make a summary of all the identified differences.

Table 6.Spectrum script test results in laptop

	PARAMETER CONFIGURATION			
VALUE	**10 Msps 1024 FFT size 50 ms**	**10 Msps 2048 FFT size 50 ms**	**20 Msps 1024 FFT size 50 ms**	**20 Msps 2048 FFT size 50 ms**
FILES GENERATED	600	600	300	300
TOTAL FILES SIZE (MB)	22.4 MB	46.0 MB	11.4 MB	22.9 MB
MAX. FILE SIZE (kB)	38.8 kB	77.6 kB	38.8 kB	77.5 kB
FILE NAMING FORMAT	compare_5995 MHz_10Msps_1 024FFT	compare_5995 MHz_10Msps_2 048FFT	compare_5990 MHz_20Msps_1 024FFT	compare_5990 MHz_20Msps_2 048FFT
FILE INDEX VALUE	16	14	31	30
SPECTRUM SWEEP TIME (seconds)	150 s	150 s	75 s	75 s
FILE FREQUENCY SEPARATION (kHz)	9.775 kHz	4.885 kHz	19.550 kHz	9.770 kHz
% CPU USED	110%	132%	123%	215%
MEMORY USED (GB)	1.69	1.80	1.81	2.05
27 MHZ PEAK FREQUENCY	27.15 MHz	27.22 MHz	27.17 MHz	27.09 MHz
27 MHZ PEAK MAX DIFF	44 dBm	32 dBm	31 dBm	26 dBm

	PARAMETER CONFIGURATION			
433 MHZ PEAK FREQUENCY	433.92 MHz	433.92 MHz	433.93 MHz	433.92 MHz
433 MHZ PEAK MAX DIFF	55 dBm	60 dBm	50 dBm	53 dBm

The script works accurately for the number of files created for each configuration, creating 600 files for 10 Msps configurations, and 300 files for 20 Msps configurations. All configurations accomplish their expected result.

The files generated accomplish the naming format for each configuration, identifying correctly the central frequency, sample rate and FFT size.

We can observe a similar behavior to the base scan script results in the total file size, and individual relations between the configurations, as both 2048 FFT configuration double the size of file to those of 1024 FFT configuration. Evidently 2048 FFT files store twice the information than 1024 FFT files. We can note that the file size and total files size doesn't increase in time, thanks to its information format.

As we saw in the *power_comparator* chapter, files contain information on differences between real time data and the base power values. The operation mode was set to Fixed Value establishing the threshold to 10 dB above the base value.

We will use Figure 4.42 to make a quick check that the data obtained in these files have sense. We see an index of 30, and in the line two we interpret that at frequency 2420 MHz, 6 values have gone above the threshold, accounting for a 20% of all the values analyzed. The minimum value that exceeded the base was 20.33 dB above it, the maximum was 33.22 dB and all values exceeded average 27.57 dB. We can see that the average exceeded number of values is correct, and that the exceeded values are above 10 dB, and the minimum, maximum and average values make sense.

As for the total size of the files generated by the script, we can see that the most efficient configuration is 20 Msps and 1024 FFT size with a size of 11.4 MB, since all the spectrum information can be contained in 300 files with 1024 lines of power differences and frequency information plus the index. With this configuration we obtain a ratio of 1.9 MB of information per each GHz analyzed.

The biggest total file size is under 46 MB, for the 10 Msps and 2048 FFT size configuration, which consists of 600 files containing 2048 lines of power differences and frequency information plus the index. This accounts for a ratio of 7.6 MB per GHz. We can see that coincidentally the two remaining configurations 10 Msps-1024FFT size and 20Msps-2048FFT size have almost the same total file size around 22.9 MB even though in the first we have 600 files and in the second we have 300 files, but this can be explained since files in the second configuration carry the double of information than in the first one. The ratio for both these configurations is around 3.8 MB of information per each GHz of spectrum.

The file index across most files is almost uniform for each configuration. For this script, this doesn't indicate us an approximate number for total spectrum scans, but we can see that in 20 Msps configurations we get twice the input data than in 10 Msps configurations. In 10 Msps configurations, with a frequency jump time of 250 ms, we expect a total scan to be performed in 150 seconds. That means that we expect two complete spectrum scans in five minutes. For 20 Msps configurations, with a frequency jump time of 250 ms, we expect a total scan to be performed in 75 seconds, that means we expect four total spectrum scans in five minutes.

The frequency separation doesn't change with respect to the base scan script values, and it is different for all configurations. We see that we obtain the greatest separation at 20Msps-1024 FFT size, with 19.550 kHz between frequency values, and the minimum at 4.885 kHz for 10Msps-2048FFT size.

We wanted to check the performance of the laptop while running the script. Even though we only obtained snapshots in a given time of the CPU performance, this helped us identify the configurations that consume more computation power. We can assume that configurations with 2048 FFT size, will require more power, since they make the double of operations than in 1024 FTT size, and it is demonstrated in the results obtained, since 10Msps-2048FFT uses 132% while 10Msps-1024FFT uses 110%. The same is observed in the other configurations 20Msps-2048FFT uses 215% against 10Msps-1024FFT with 123%. It is necessary to note that the use percentage includes all processes run in the laptop at that time, and that all tests were performed under the same circumstances.

Contrary to what we observed in the base script results, RAM memory usage varies according to the test configuration. This can be explained by the fact that in this script we work with up to three files simultaneously, make more operations and store data in memory while saving the information in files. As we saw in the *comparator* block, we use the base power files to make comparisons, so files and its content are loaded into memory, also the values of the existing compare file is loaded into memory, and the real time results for differences, averages, minimums and maximums are also stored in memory until the final values are stored in the database files. We can see that tests with 2048 FFT size configurations, use more RAM memory than those with 1024 FFT size.

To verify that the script indeed detects unusual signals generated during the tests, we have verified the database files for each configuration at frequencies 27 MHz and 433 MHz. All files reflect that peaks have been detected in the corresponding frequency,with power values twice, three or four times greater than the base power value.

Of all the four different configurations used to test the spectrum scan script we can see that we can get better results using 20 Msps and 1024 FFT size. It gives us a the smallest total file size for all the information with 11.4 MB, it also gives us the most input data vectors in five minutes with 31. It is also has a better CPU and RAM memory performance compared to configurations with 2048 FFT sizes.

4.2.6. Raspberry Test: 10Msps 1024FFT 1000ms

For the tests performed in Raspberry we found that 250 ms as the frequency switch time, wouldn't let the script generate the files for all the frequencies. So after a few tries, the time was defined as 1 second before moving to the next frequency, and all files would be created without skipping any. Logically, the total time for the test had to be increased, so we define the new total time as 15 minutes.

In order to dedicate the CPU resources exclusively to the execution of the script, the CPU monitoring tool *htop* wasn't used.

This script's test results are similar to the obtained in the laptop: 600 files are created with a total size of 22.9 MB and each file's maximum size is under 40 kB.

The script generates correctly the files with its corresponding power differences values and frequency values. The frequency separation corresponds to its configuration, but we see that the file index values can be different for various files, so we won't take it into account.

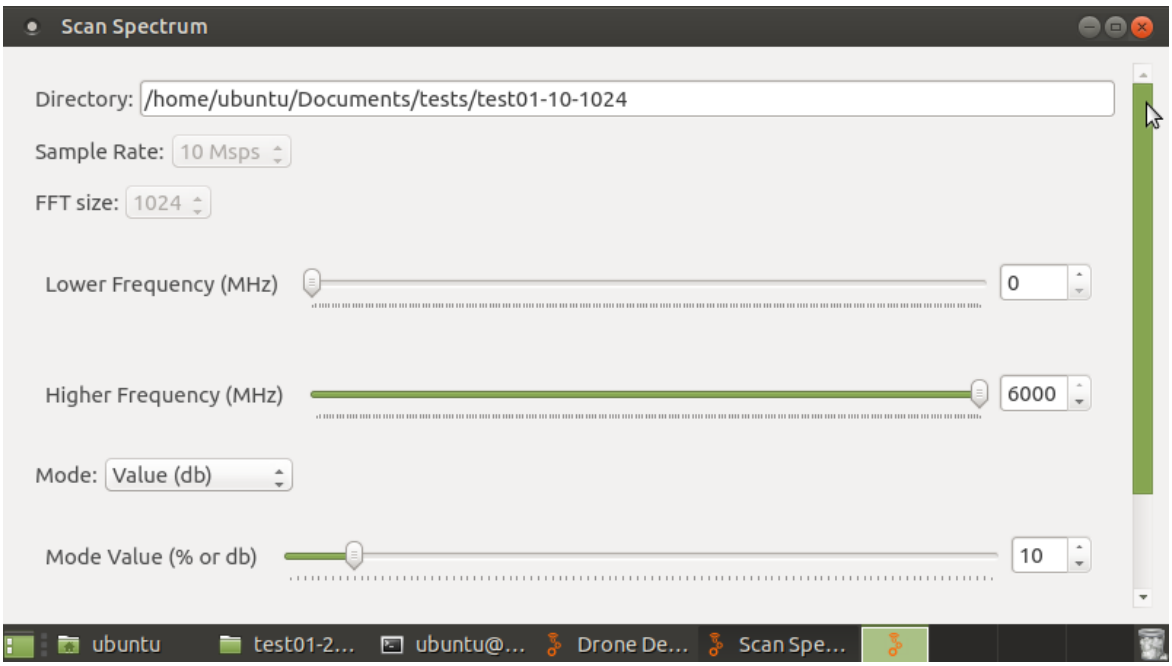


Figure 4.46: Raspberry spectrum script parameters configuration for 10 Msps-1024 FFT size.

4.2.7. Raspberry Test: 10Msps 2048FFT 1000ms

This script's test results are similar to the obtained in the laptop: 600 files are created with a total size of 44 MB and each file's maximum size is under 80 kB.

The script generates correctly the files with its corresponding power differences values and frequency values, the frequency separation corresponds to its configuration, but we see that the file index values can be different for various files, so we won't take it into account.

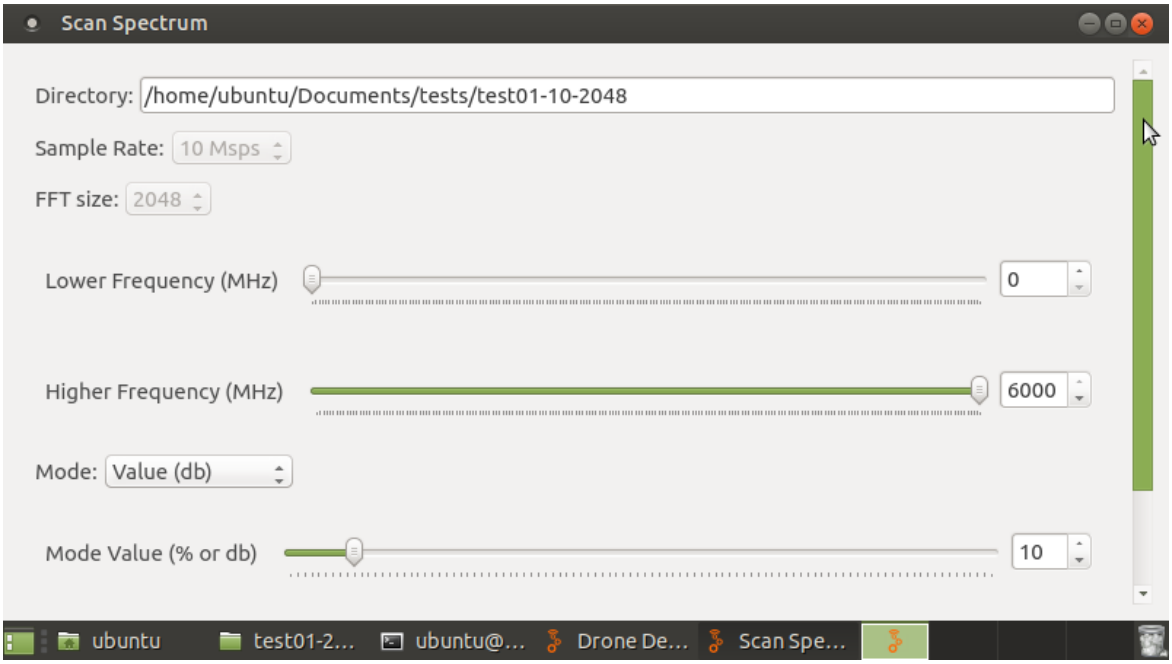


Figure 4.47: Raspberry spectrum script parameters configuration for 10 Msps-2048 FFT size.

4.2.8. Raspberry Test: 20Msps 1024FFT 1000ms

This script's test results are similar to the obtained in the laptop: 300 files are created with a total size of 11.4 MB and each file's maximum size is under 40 kB.

The script generates correctly the files with its corresponding power differences values and frequency values, the frequency separation corresponds to its configuration, but we see that the file index values can be different for various files, so we won't take it into account.

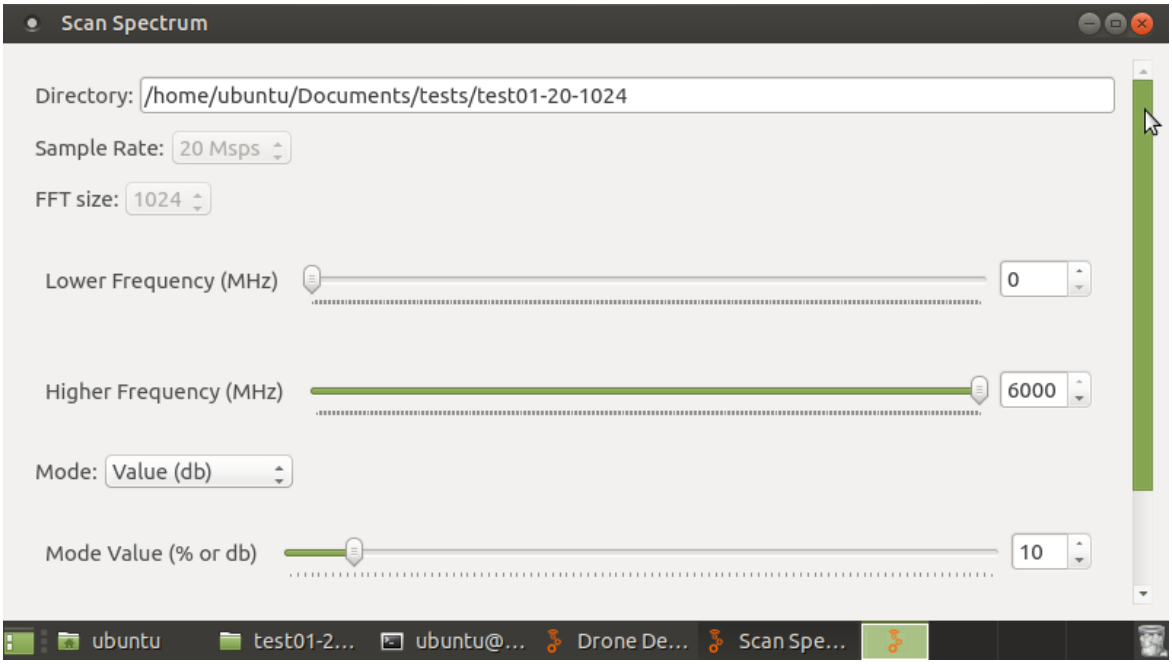


Figure 4.48: Raspberry spectrum script parameters configuration for 20 Msps-1024 FFT size.

4.2.9. Raspberry Test: 20Msps 2048FFT 1000ms

This script's test results are similar to the obtained in the laptop: 300 files are created with a total size of 22.9 MB and each file's maximum size is under 80 kB.

The script generates correctly the files with its corresponding power differences values and frequency values, the frequency separation corresponds to its configuration, but we see that the file index values can be different for various files, so we won't take it into account.

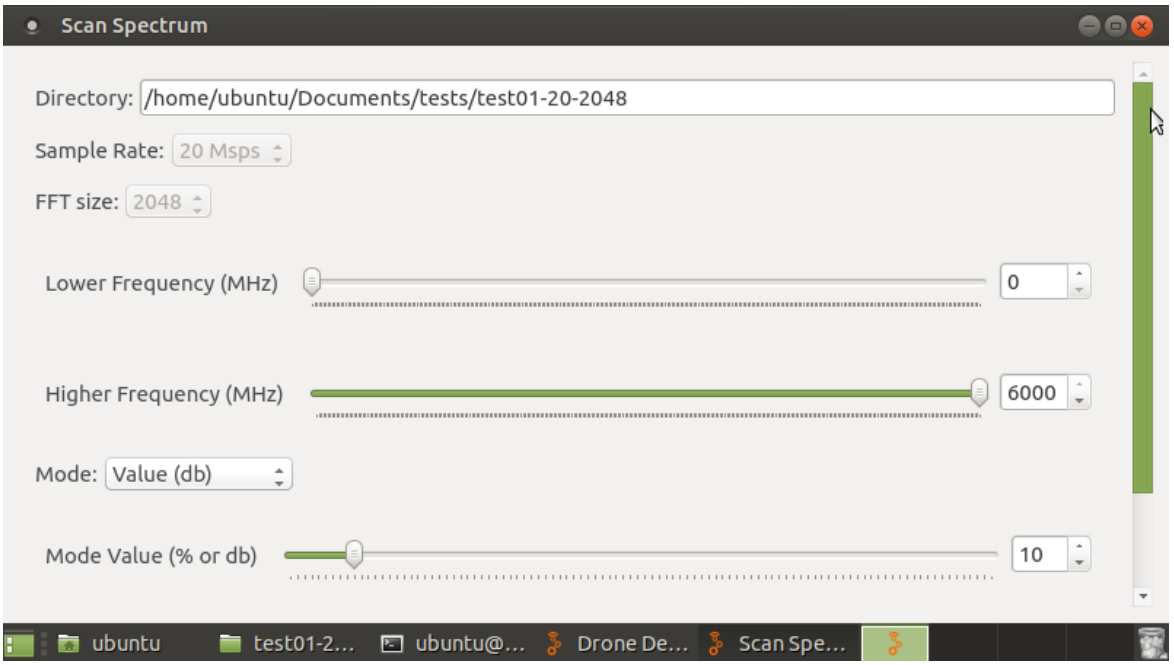


Figure 4.49: Raspberry spectrum script parameters configuration for 20 Msps-2048 FFT size.

4.2.10. Raspberry Test Analysis for Spectrum Script

For these tests we have decided to stop using the *htop* tool, so all the computation power is dedicated to the execution of the script test, trying to avoid glitches and blockings in the responsiveness of the Raspberry.

The most important result we obtain is that the script can indeed be executed in the Raspberry for all configurations, and that we can get the file database with the differences in the power values for all the spectrum from 1 MHz to 6 GHz.

All the four parameters configurations have worked correctly and the obtained result for files generated and occupied space by the database are similar to the laptop results as expected.

The main difference is the frequency switch time which in this case is four times that of the laptop tests, going from 250 ms to 1000 ms. This increase in time has assured us that all frequencies can be analyzed and therefore their corresponding files are created successfully.

Finally we can reach to the same conclusion than in the laptop test, by selecting the 20 Msps and 1024 FFT size as the most efficient configuration as per the total number of files generated and its total size occupied.

4.2.11. Comparison of results from laptop and Raspberry

When we analyze the resulting data obtained from the tests in laptop and Raspberry, we generate a graphic to compare the obtained values at a specific band, and we choose the 433 MHz band. As in the base scan script test results comparison, we create a custom function that generates the graphics that help us compare results obtained in different tests.

We can see very similar results in general among the two resulting databases, but we see that in the laptop the generated signals at 27 MHz and at 433 MHz were detected in all test. But only one in four tests detected the signals when performing the tests in Raspberry.

We must remember that the times for switching the frequency and the total test time is different for the laptop and Raspberry scenario. For laptop we have a total test time of five minutes, and for tests with 10 Msps parameter, the total scan time for the 6 GHz spectrum is two and a half minutes, and for 20 Msps the total scan time is one minute and a quarter. For Raspberry we have a total test time of 15 minutes, and for 10 Msps configurations we have a total scan time of 10 minutes and for 20 Msps the total scan time is five minutes. These differences explain that in the laptop we have less time between analysis of a single frequency, than in the Raspberry, so it is more probable to detect a signal if I analyze a given frequency each minute and a quarter, than each five minutes.

It is possible that for the tests done in Raspberry, the signal was generated while the scanner was analyzing other frequencies, so it couldn’t be detected, and when the

scanner was at the signal's frequency, it wasn't being generated. This determines that it could be a more difficult task to detect unusual activity with this script in Raspberry.

In conclusion, we see that this script has generated significant difference values comparing the real time power obtained from the HackRF with the base power values obtained from the base scan script.

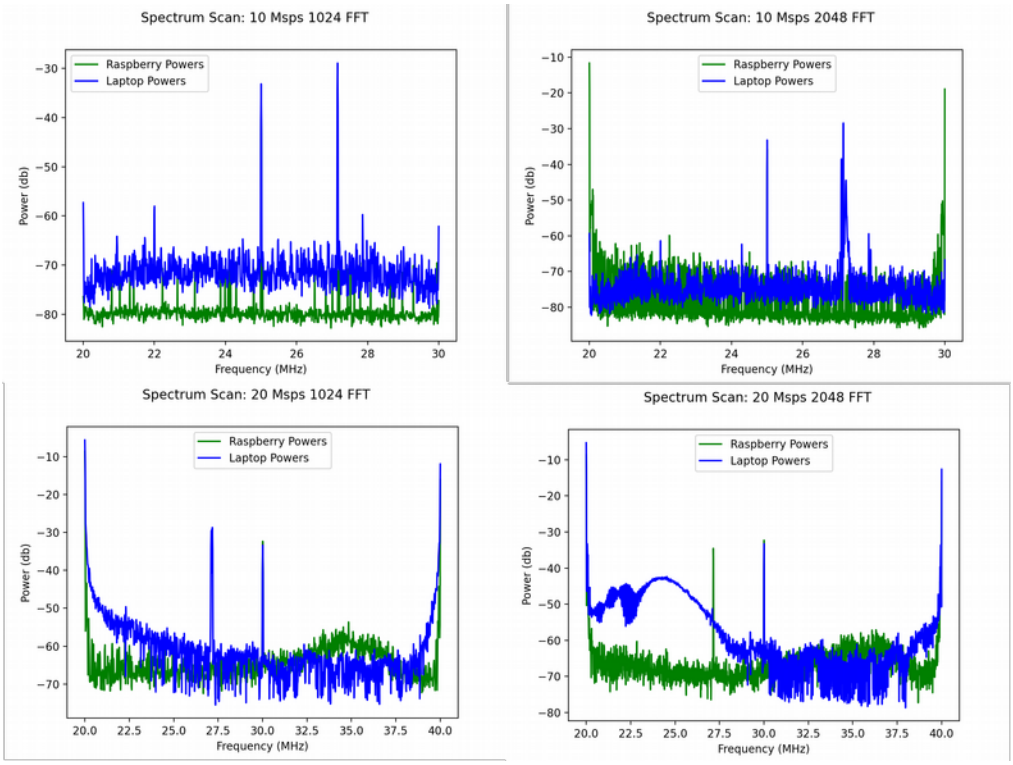


Figure 4.50: Comparison of results obtained in laptop and Raspberry for maximum values generated by spectrum scan script in 27 MHz

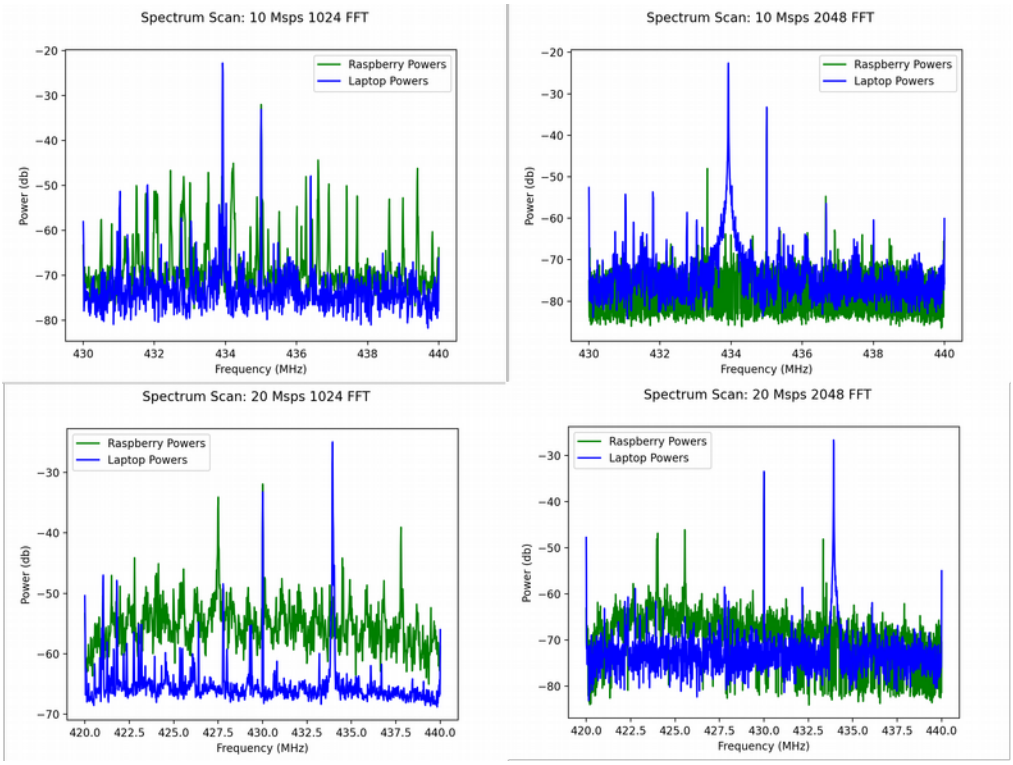


Figure 4.51: Comparison of results obtained in laptop and Raspberry for maximum values generated by spectrum scan script in 433 MHz

4.3. Spectrum Scan Script Tests

Our main focus in this script tests is to verify that this script creates the corresponding file database and that we can visually identify the unusual activity while monitoring the system performance in the executing machine. We will use the parameter configurations that has proven to be more efficient so far in the previous tests, and that is 20 Msps and 1024 FFT size.

First we will perform a spectrum scan without our generated signal activity, to see the values that are obtained under normal circumstances. Then we will generate three different signals: from the toy remote control at 27 MHz, from the garage remote controller at 433 MHz and from the drone at 2.4 GHz. This data will be compared with the one under normal circumstances.

We will use the *comparator* block in fixed value mode, so only power differences above 10 dB will be written to the files.

It is important to state that we will reuse the base power values obtained in the first test.

4.3.1. Laptop Test: Toy Remote Controller (27 MHz)

The setup for this test is the following: the laptop, the HackRF and toy car with its remote controller.

Figure 4.52: Band scan script test setup for toy remote controller (27 MHz)

Once we have our script executed, we can see the window with the user interface controls. Since 433 MHz band is the default configuration for this script, we must move the center frequency control down to 30 MHz, so we can get the band where we expect to see the generated signal.

While activating the remote control and keeping it activated, we can see in the graphic the presence of a peak in 27 MHz. Comparisons at the data level are being performed and the results stored in the file databases. Graphically we don't perform comparisons in this script, since it is done exclusively in the main script.

Exploring the file created during this test, we can see that the peak of the generated signal is located at 27.17 MHz, with a power value 46 dB above the base. The other data on this frequency shows that 72% of all the received power values in this frequency were above the threshold.

The script has helped detect both visually and in the database the signal generated by the toy car remote controller.

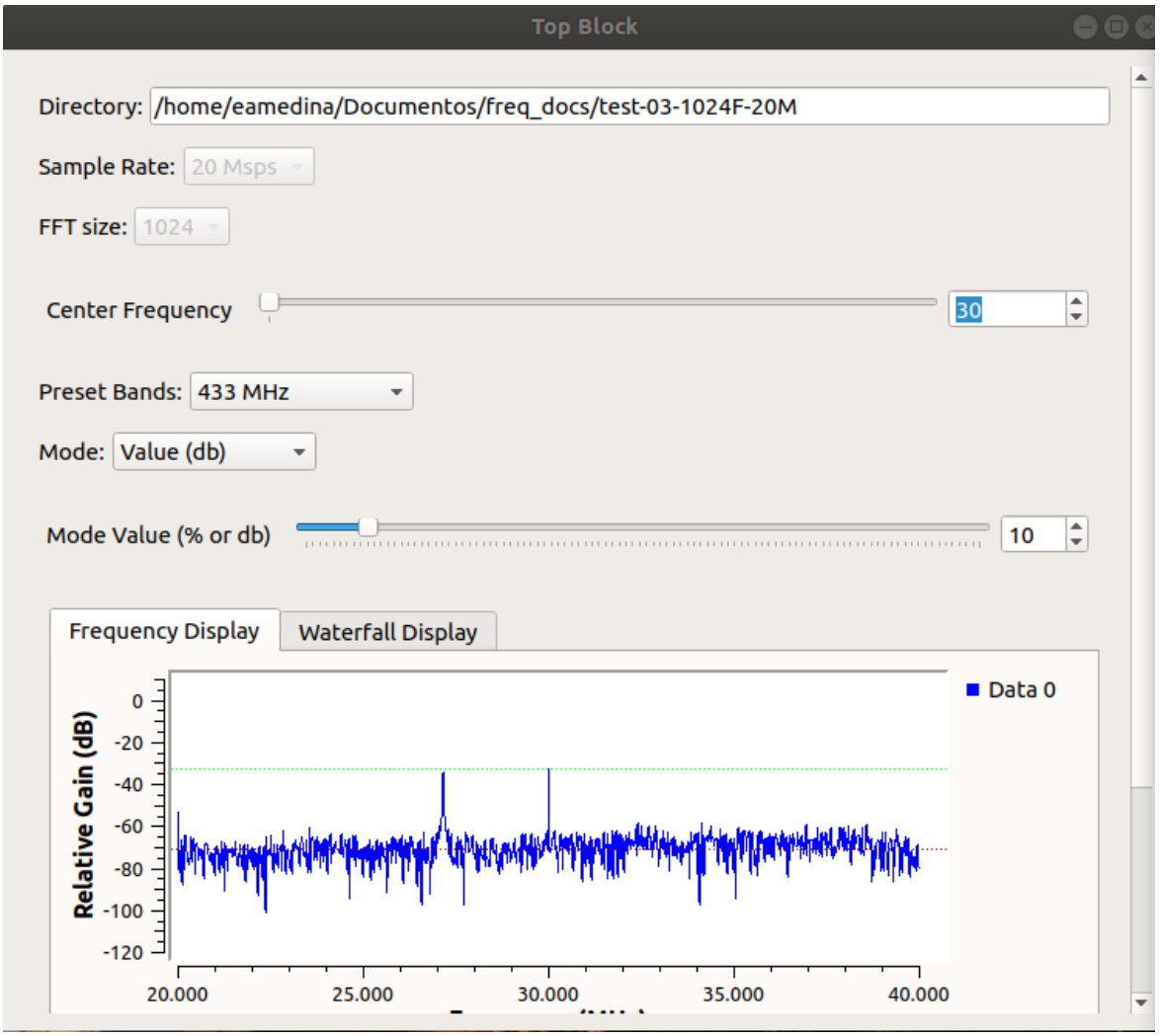


Figure 4.53: Band scan script graphic detection of an active signal in 27 MHz.

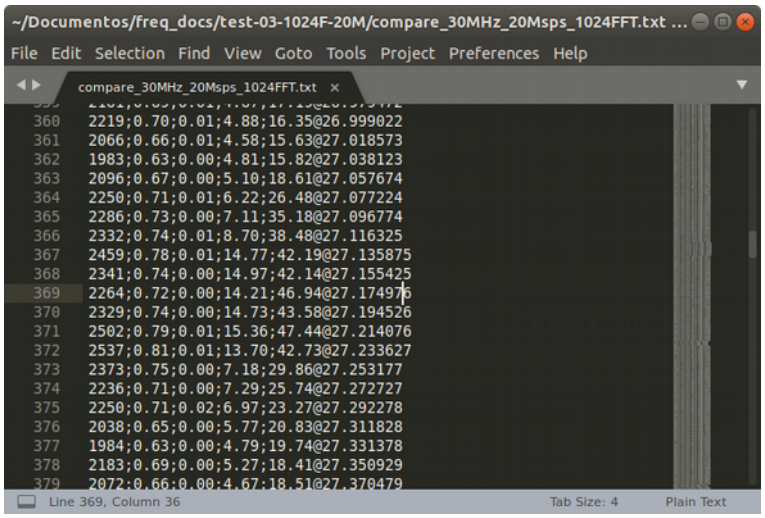


Figure 4.54: Band scan script file with detection of unusual activity at 27 MHz

We must compare the values obtained during this test, against the value obtained with no activity at 27 MHz to see that the behavior is identified in the data.

At 27.17 MHz, were the peak was detected while using the remote controller, we see that there is no activity above the threshold. And that neighboring frequencies have values with at most 11 dB above the base value.

In the contrary, the output file obtained while doing the test, shows that the peak's neighboring frequencies also have maximum values that are at least 3 to 4 times the values than those while no unusual activity happens in that band.

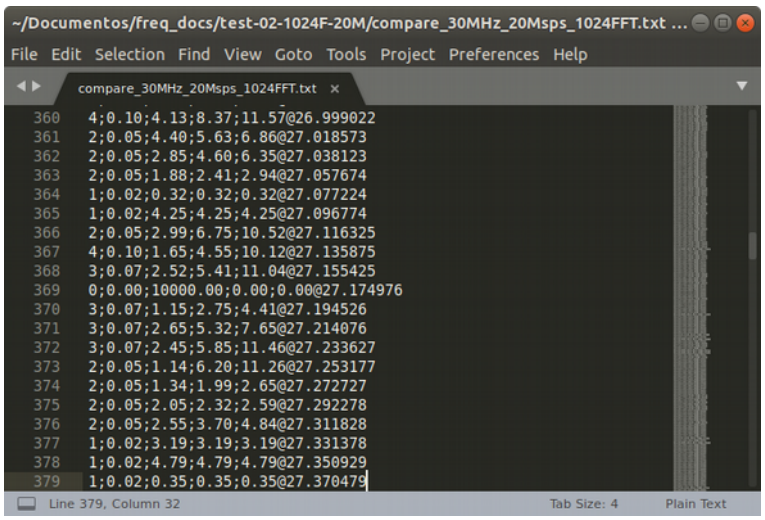


Figure 4.55: Spectrum scan script file with values donde before the test at 27 MHz

The graphic is generated using a created custom Python function that allows us to compare results from different tests.

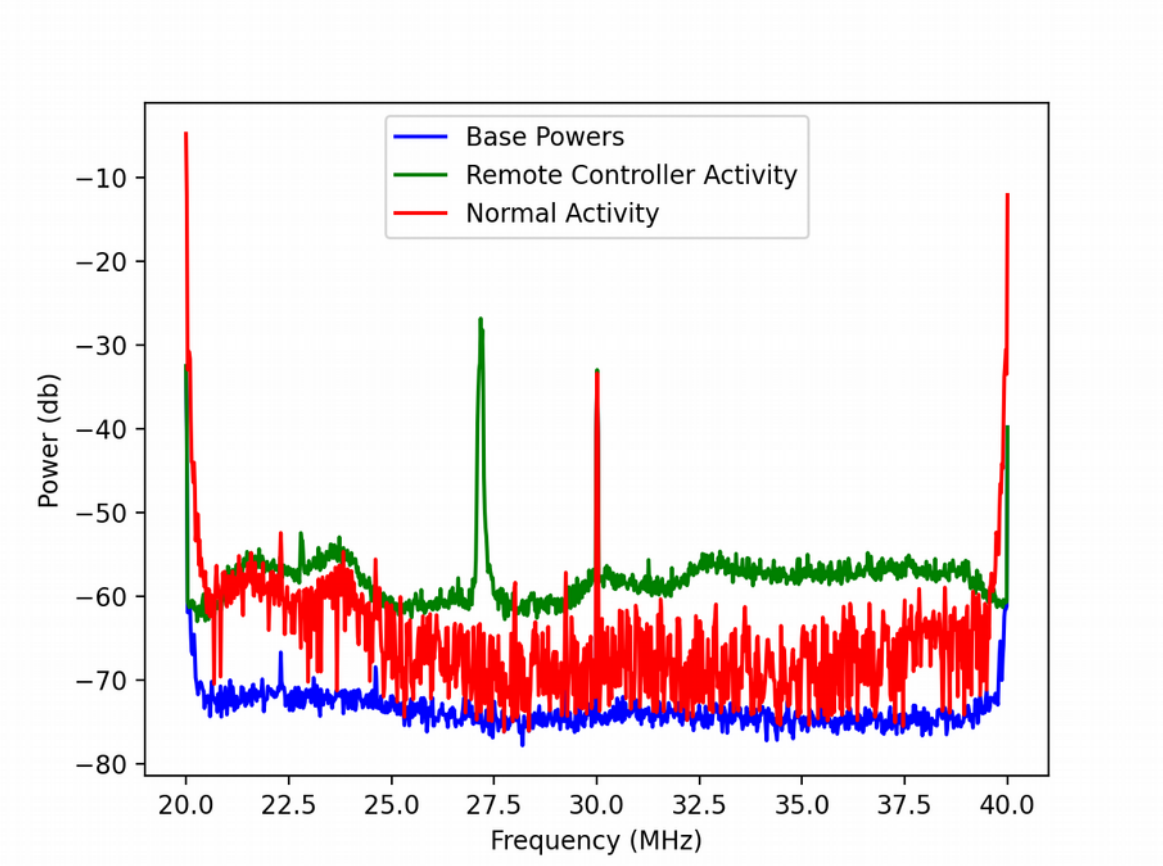


Figure 4.56: Laptop scan band script test results comparison for 27 MHz

4.3.2. Laptop Test: Garage Remote Controller (433 MHz)

The setup for this test is the following: the laptop, the HackRF and the *WhyEvo* garare remote controller.

Figure 4.57: Band scan script test setup for garage remote controller (433 MHz)

When we open this script and its user interface is displayed, we can start using it since 433MHz is the default selected band. So we start generating the signals from the remote controller in a repetitive way. In the graphic a signal arises in the 433 MHz frequency.

Exploring the file created during this test, we can see that the peak of the generated signal is located at 433.93 MHz, with a power value 66 dB above the base. The other data on this frequency shows that 61% of all the received power values in this frequency were above the threshold.

The script has helped detect both visually and in the database the signal generated by the garage remote controller.

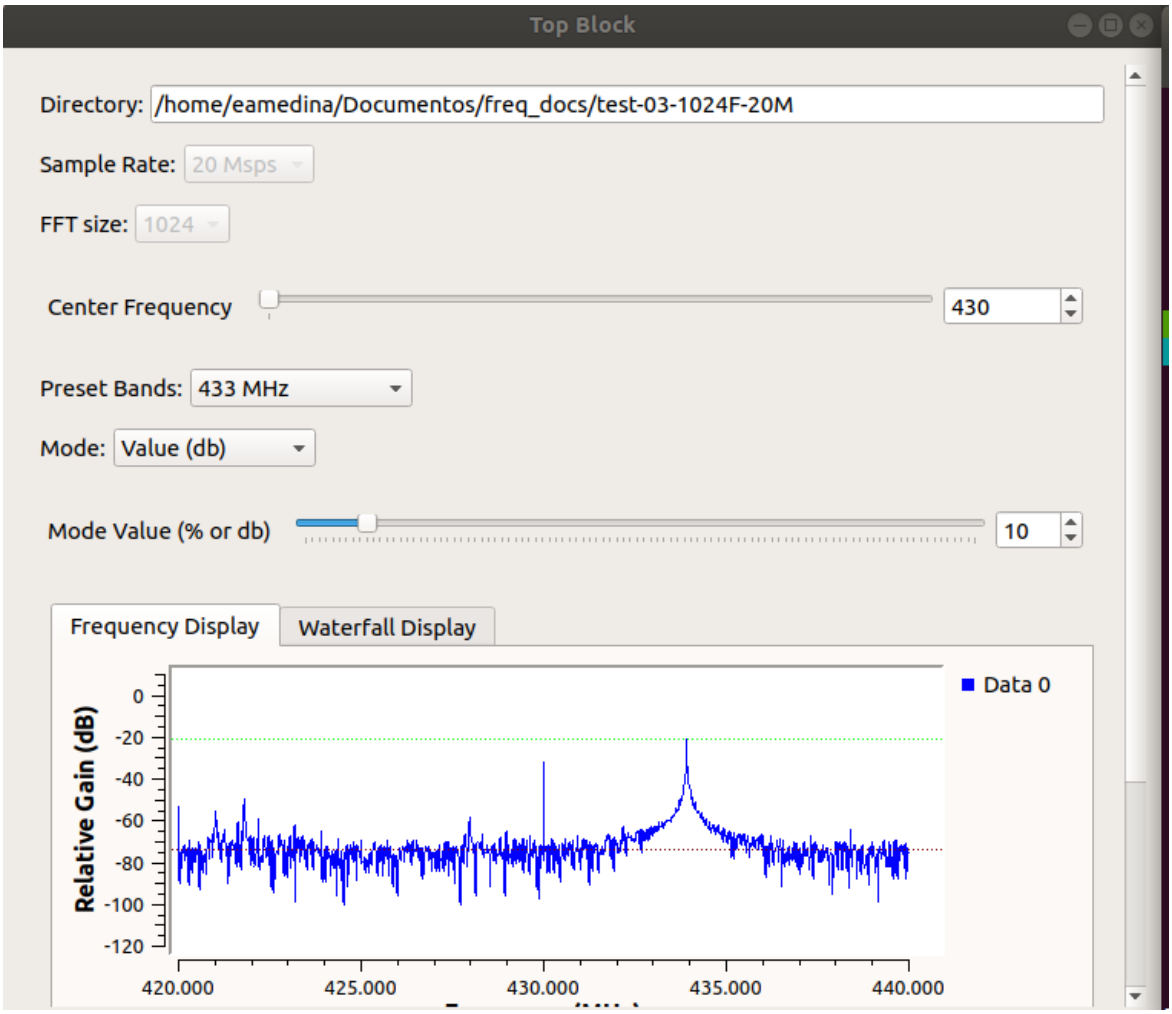


Figure 4.58: Band scan script graphic detection of an active signal in 433 MHz.

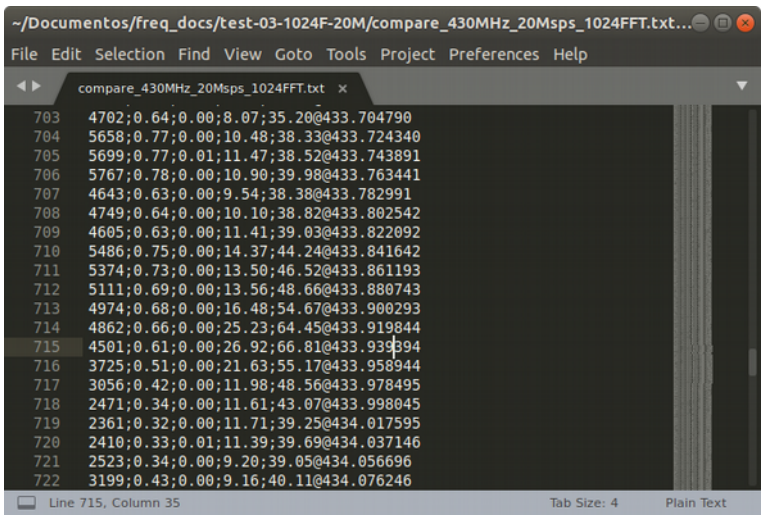


Figure 4.59: Band scan script file with detection of unusual activity at 433 MHz

We must now compare the values obtained during this test, against the value obtained with no activity at 433 MHz to see that the behavior is identified in the data.

At 433.93 MHz, were the peak was detected while using the remote controller, we see that there is activity above the threshold but not greater than 9 dB above the base. That means that the received power when there is activity is more than 50 dB above the power when there is no activity. We can see that the same behavior happens in the neighboring frequencies.

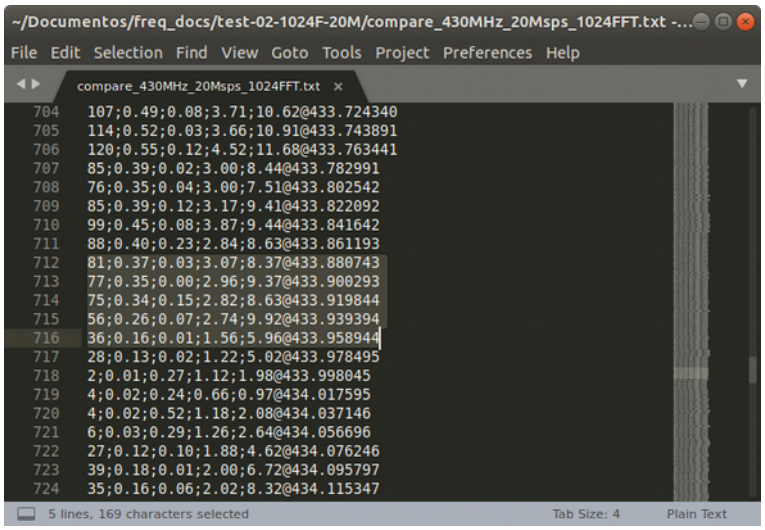


Figure 4.60: Spectrum scan script file with values donde before the test at 433 MHz

The graphic is generated using a created custom Python function that allows us to compare results from different tests.

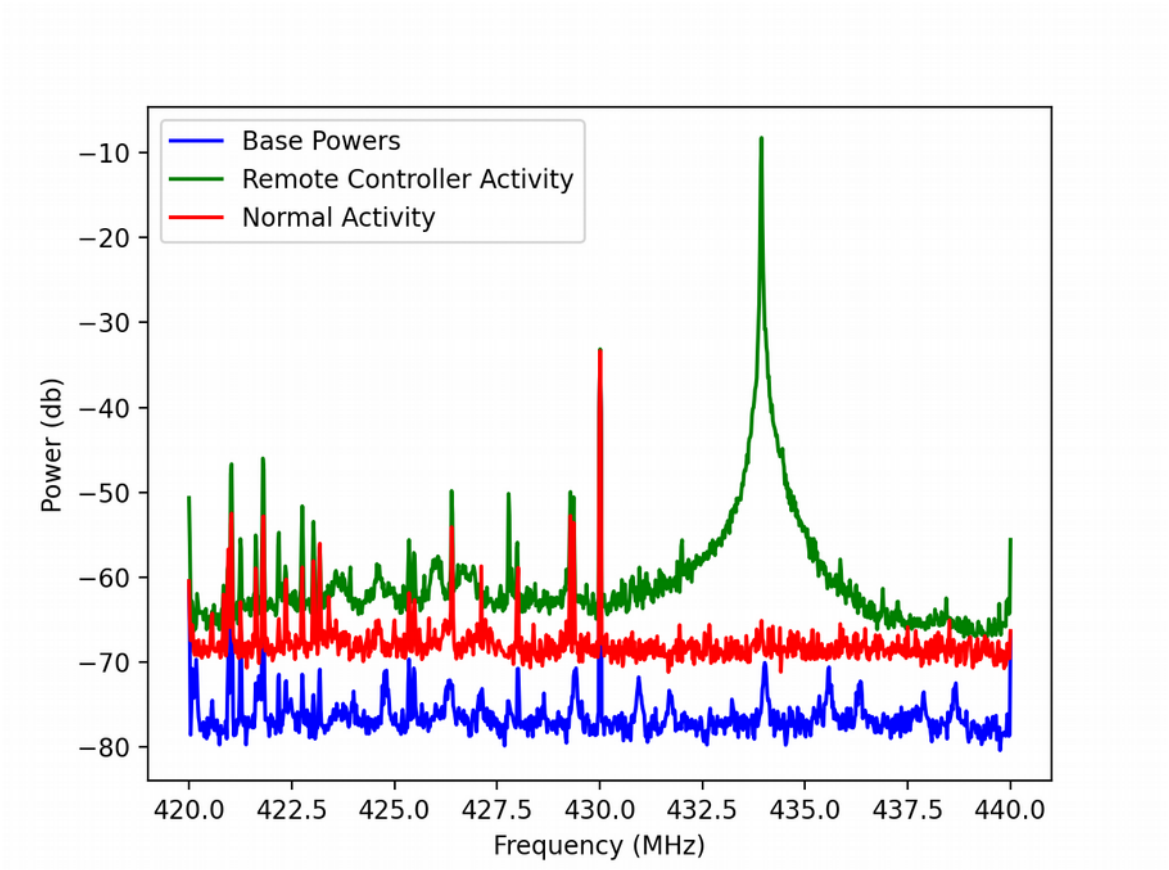


Figure 4.61: Laptop scan band script test results comparison for 433 MHz

4.3.3. Laptop Test: Drone (2.4 GHz)

The setup for this test is the following: the laptop, the HackRF and the DJI Mavic Pro drone with its remote controller. The location for this test is an indoor location in an urban area, so we expect a constant WiFi and Bluetooth activity in the 2.4 GHz band. The ideal scenario would be one, where there is no activity in the 2.4 GHz band and only the drone activity is expected, but this test in an urban environment proves that our solution can work in real scenarios.

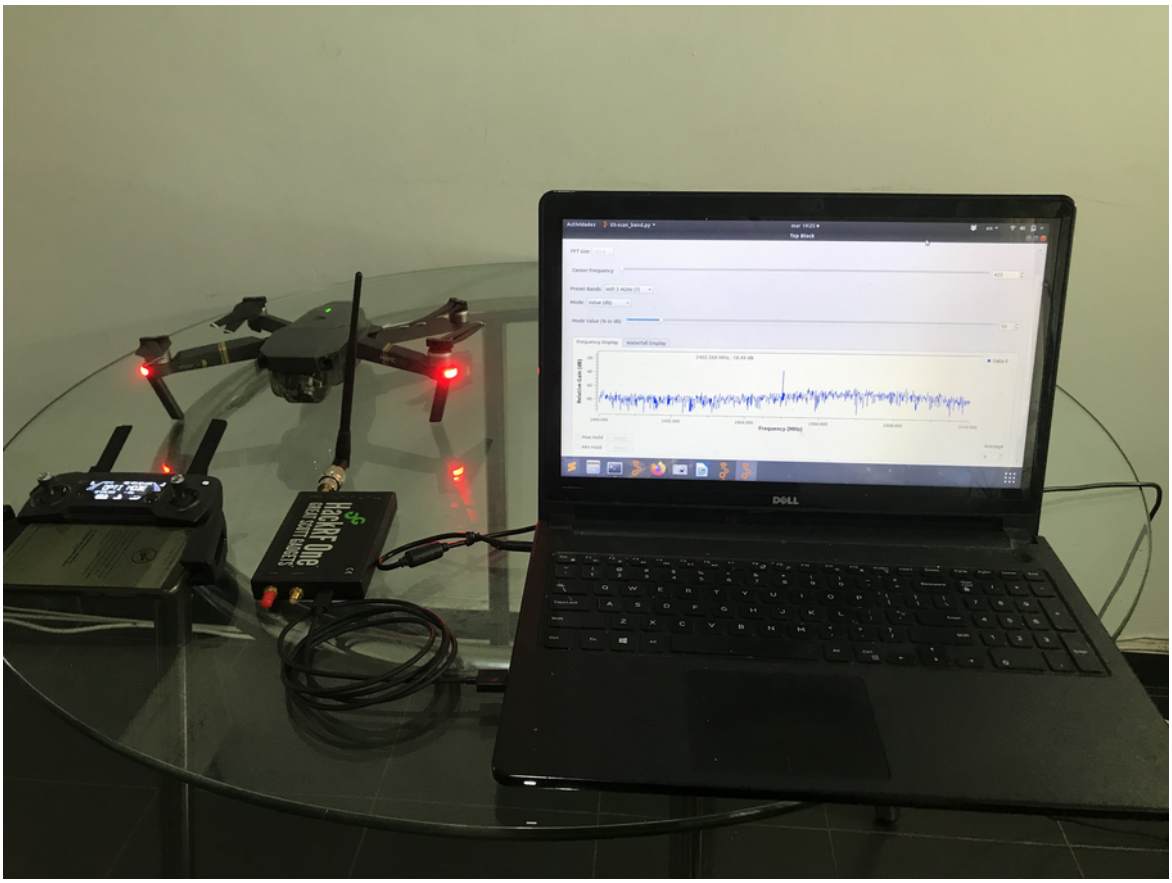


Figure 4.62: Laptop band scan script test setup for drone (2.4 GHz)

This test is completely different than the others. Since remote controllers have only one operation mode and we expect an unique peak in channels that are barely used, it can be relatively easier to detect an unusual signal both visually and by exploring the file database. But since we are doing this test with the DJI Mavic Pro drone, that has different operation modes, and it uses the 2.4 GHz band to communicate with its remote controller, we may face different difficulties.

The first one is that we can't be sure which transmission configuration is set in the drone, so we don't know if we are searching for a 1.4, 10 or 20 MHz signal, and also the frequency at which it appears can vary all across the 2400-2480 MHz spectrum. The second one is that the 2.4 GHz is widely used for WiFi and Bluetooth among others. So

we can expect all type of signals here. The last difficulty is that our HackRF scans only up to 20 MHz, and we need to verify at least 80 MHz.

In our script we provide preset configurations that help us change the visualized frequencies of the 2.4 GHz band, by changing the selection in the preset band parameter and exploring the band in chunks of 20 MHz.

Once we get the drone flying and communicating permanently with its remote controller in RC mode, we can explore the whole band, and we detect a raised signal that is in the 2460-2480 MHz band. This signal occupies around 18 MHz and its center is around 2470 MHz. Contrary to the behavior seen in the other chunks of the 2.4 GHz band, this signal remains raised all the time while the communication lasts between the drone and remote controller. Once the drone lands and the communication is dropped, this signal disappears.

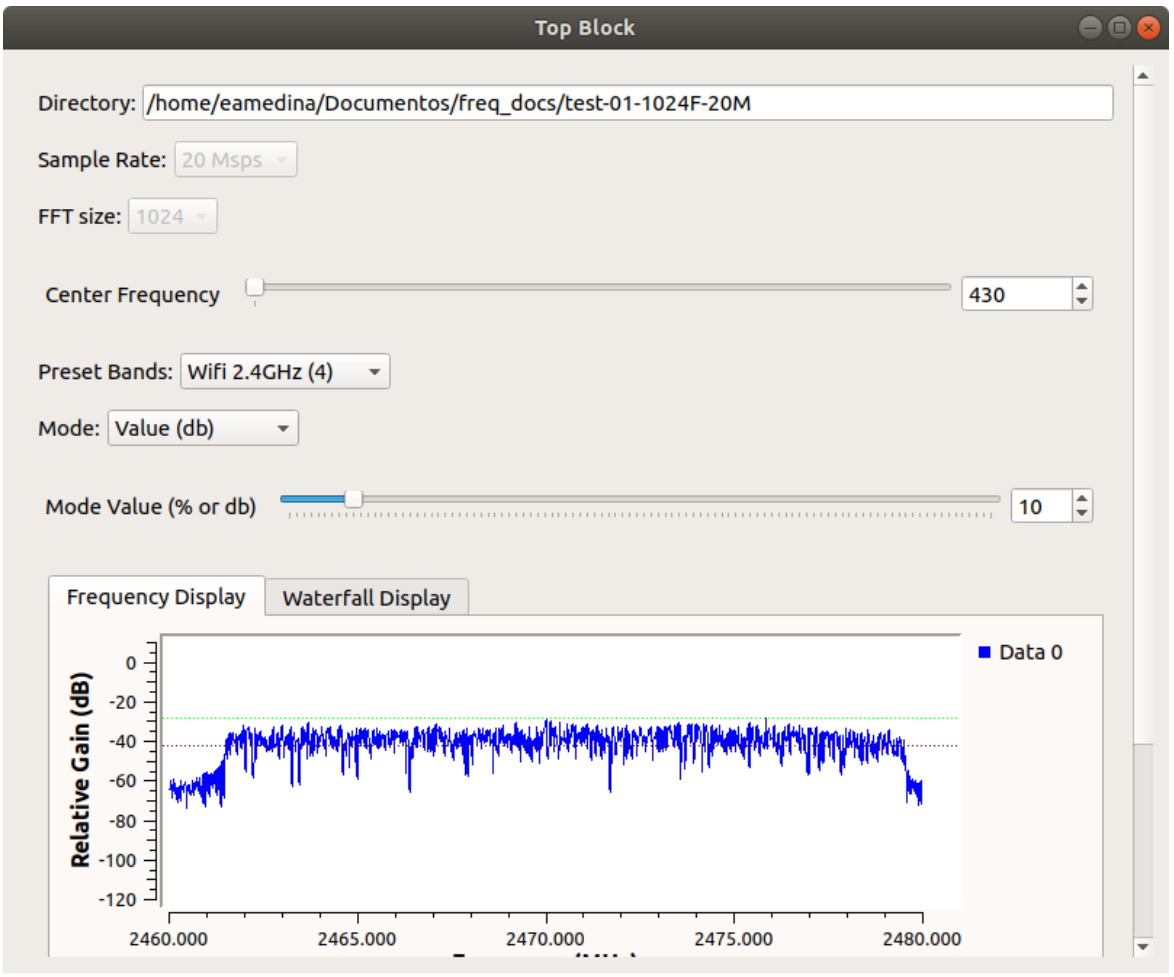


Figure 4.63: Band scan script graphic detection of an active signal in 2.4 GHz.

Using the FCC ID found in the drone remote controller, we can explore the RF tests done to the drone's communications. There we find a graphic of the operation mode of 20 MHz High Channel, which has its center at 2472.5 MHz and its bandwidth is 17.92 MHz. This

data confirms that the signal we have graphically detected corresponds to the drone communication in RC mode.

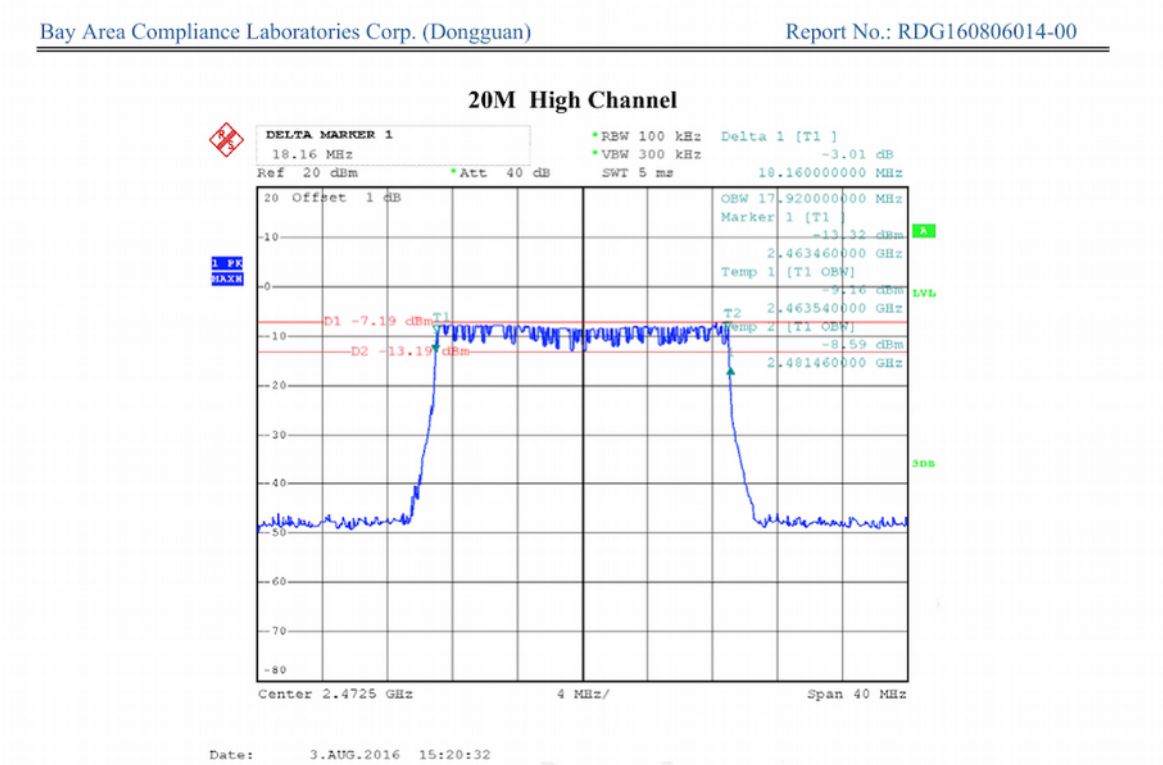


Figure 4.64: DJI Mavic Pro drone signal analysis in 20 MHz high channel operation mode.

Detecting the drone activity by reading the file database seems like an unsuccessful method. We will find WiFi, Bluetooth and other type of activity, and maximum values that share similar power values across neighboring frequencies. With the available options given by this project, the only approach that we can use to identify a drone signal is the visual one.

Comparing the values obtained from measuring the 2.4 GHz with and without drone activity, we see a rather similar behavior in the first 45 MHz of the band. This can be deducted by the fact that WiFi channels 1 and 6 are among the most used in WiFi communications and its activity can be found up to 2448 MHz. The remaining 35 MHz of the band show a a greater difference between values with and without drone activity, and the most curious characteristic is found at the end of the band, with an almost constant maximum value in the drone activity test result. This is due to the drone communications signal, but analyzing only this maximum data we can't get to a conclusion.

It is necessary that a different approach is taken when detecting drone activity in the 2.4 GHz band. Bandwidth and center frequencies of the signals appeared in this band must be analyzed and compared to the signals that actually drones use.

The graphic is generated using a created custom Python function that allows us to compare results from different tests.

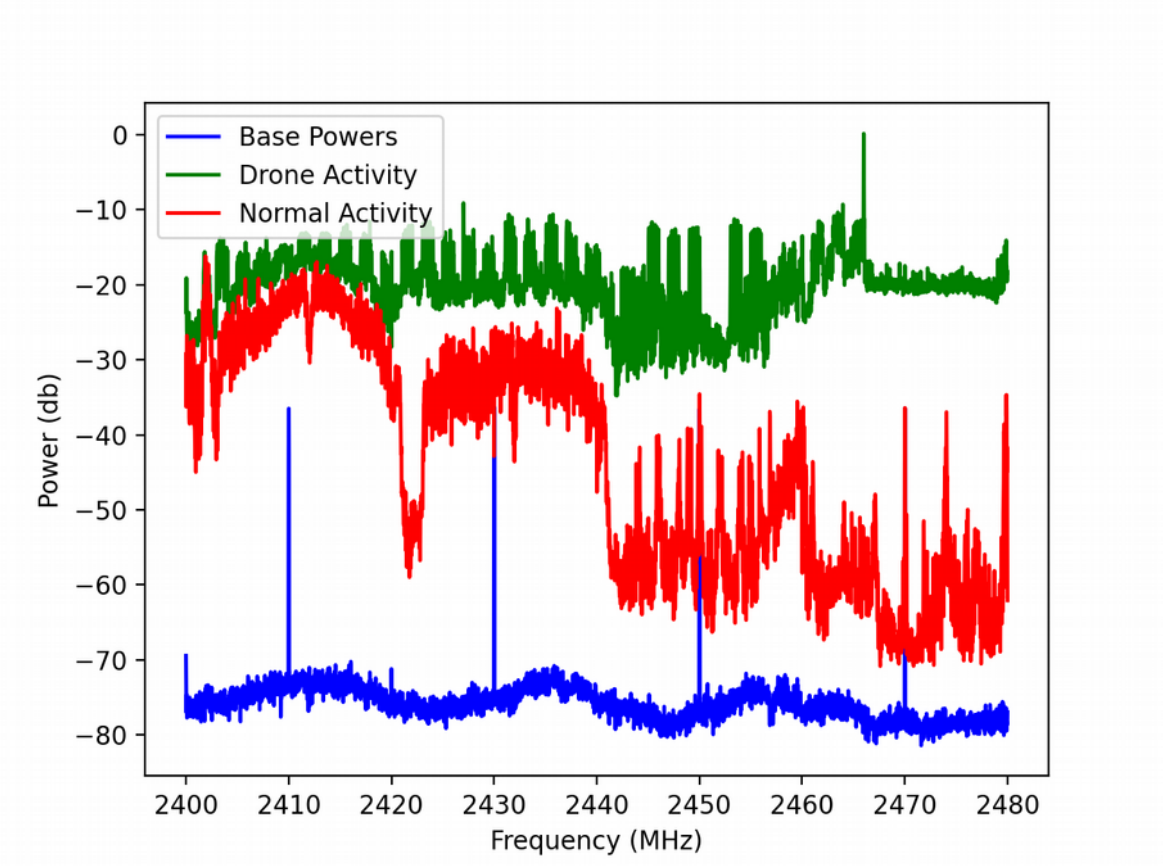


Figure 4.65: Laptop scan band script test results comparison for 2.4 GHz band

4.3.4. Laptop Test Analysis for Band Scan Script

We have tested the band scan script which has the same functions as the spectrum scan script, but this time limited in a specific band whose width is given by the sample rate. This script provides a real time graphic with the power values and extra controls so the user can visually explore the spectrum searching for unusual activity.

In the three test performed we have successfully identified unusual activity in 27 MHz, 433 MHz and 2.4 GHz, and we have come to the conclusion that the identified signals correspond to the ones we have generated with the remote controllers of the toy car, garage and drone, respectively.

For the signals we have generated in 27 MHz and 433 MHz, we have seen that using both the approaches of analyzing the data saved in the file database and exploring visually their band, we have been able to detect the activity generated by the remote controllers. Obtaining graphics from the data with and without our generated activity we

can see noticeable differences and even the signal itself. And with the graphic tool provided by the script, we have also been able to detect it.

Instead, when working in 2.4 GHz we have had issues to detect the unusual activity with the file database. But with the graphical tool, that allows to explore the whole band, we have been able to detect the signal generated by the drone remote controller. Our approach to work with maximum values obtained in real time for drones has proven as inadequate under the current testing scenario. For scenarios in which the WiFi and Bluetooth activity is negligible or nonexistent, such as in rural environments, the data analysis can lead to better results than the obtained here.

4.3.5. Raspberry: Toy Remote Controller (27 MHz)

The parameter configuration will be the same as in the laptop's test with 20 Msps and 1024 FFT, and before performing the test we have made a spectrum sCSN to have the power values without our generated activity and compare to the results obtained when generating the signals.

The setup for this test is the following: the Raspberry, the HackRF and toy car with its remote controller.

Figure 4.66: Raspberry band scan script test setup for remote controller (27 MHz)

We will follow the same process as in the laptop test, configuring the center frequency to 30 MHz with the script's controls.

While activating the remote control and keeping it activated, we can see in the graphic the presence of a peak in 27 MHz.

Figure 4.67: Band scan script graphic detection of an active signal in 27 MHz.

We must compare the values obtained during this test, against the value obtained with no activity at 27 MHz to see that the behavior is identified in the data.

At 27 MHz, were the peak was detected while using the remote controller, we see that there is no activity above the threshold under normal circumstances. In the contrary, the output file obtained while doing the test, shows the peak and the neighboring frequencies with values well above the normal ones.

The graphic is generated using a created custom Python function that allows us to compare results from different tests.

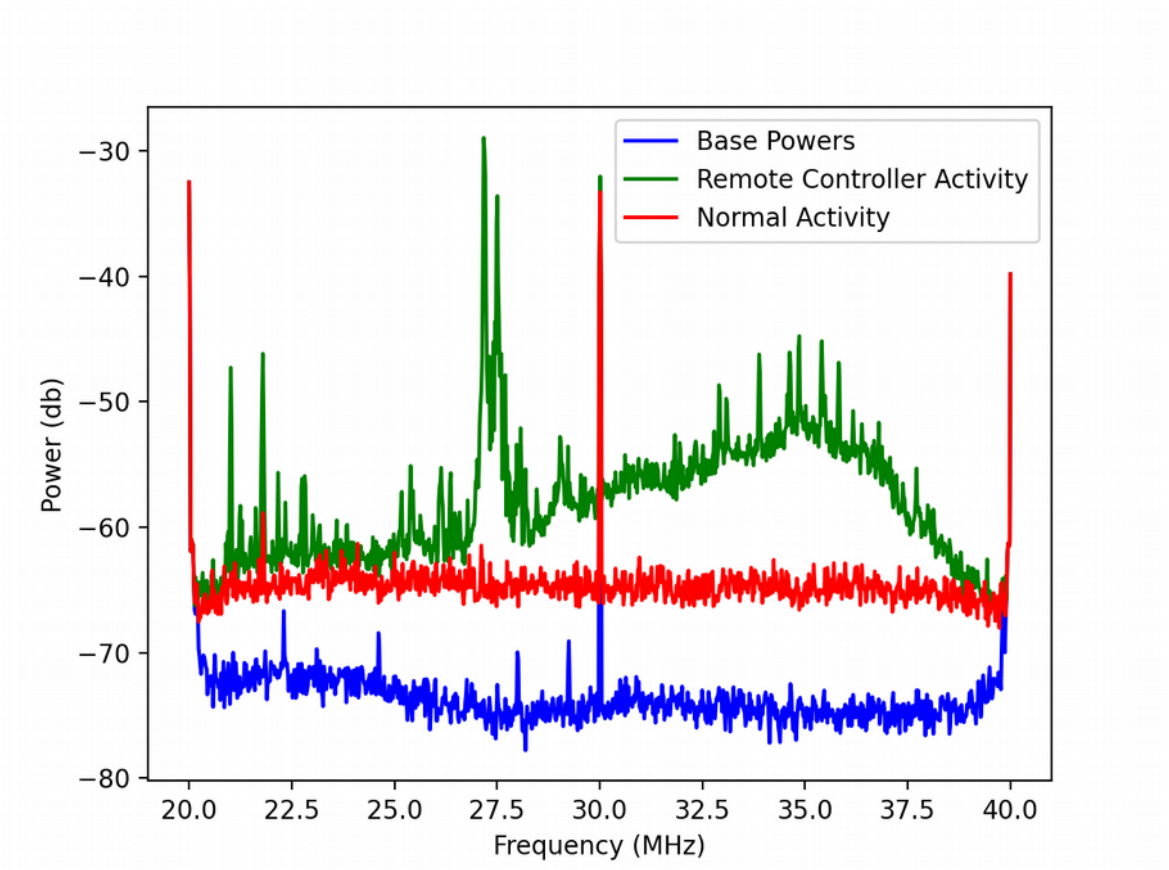


Figure 4.68: Scan band script test results comparison for 27 MHz

4.3.6. Raspberry: Garage Remote Controller (433 MHz)

The setup for this test is the following: the Raspberry, the HackRF and the *Why Evo* garage remote controller.

Figure 4.69: Raspberry band scan script test setup for remote controller (433 MHz)

When we open this script and its user interface is displayed, we can start using it since 433MHz is the default band. So we start generating the signals from the remote controller in a repetitive way. In the graphic a signal arises in the 433 MHz frequency.

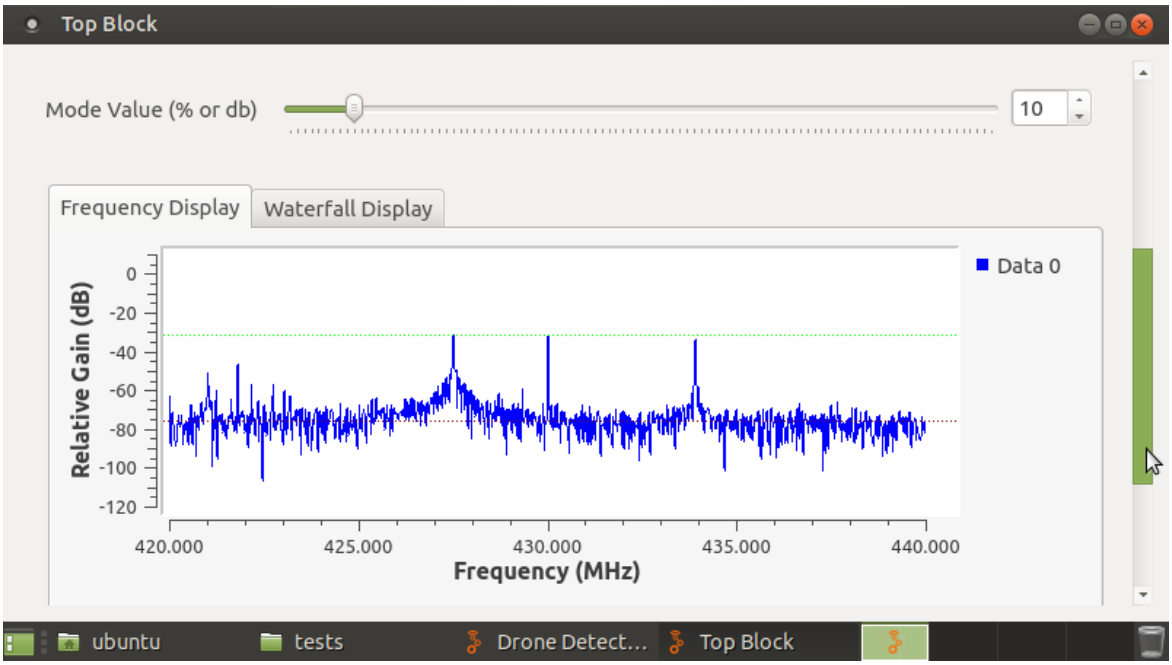


Figure 4.70: Band scan script graphic detection of an active signal in 433 MHz.

We must compare the values obtained during this test, against the value obtained with no activity at 433 MHz to see that the behavior is identified in the data.

At 433 MHz, were the peak was detected while using the remote controller, we see that there is no activity above the threshold under normal circumstances. In the contrary, the output file obtained while doing the test, shows the peak and the neighboring frequencies with values well above the normal ones.

The script has helped detect both visually and in the database the signal generated by the garage remote controller.

The graphic is generated using a created custom Python function that allows us to compare results from different tests.

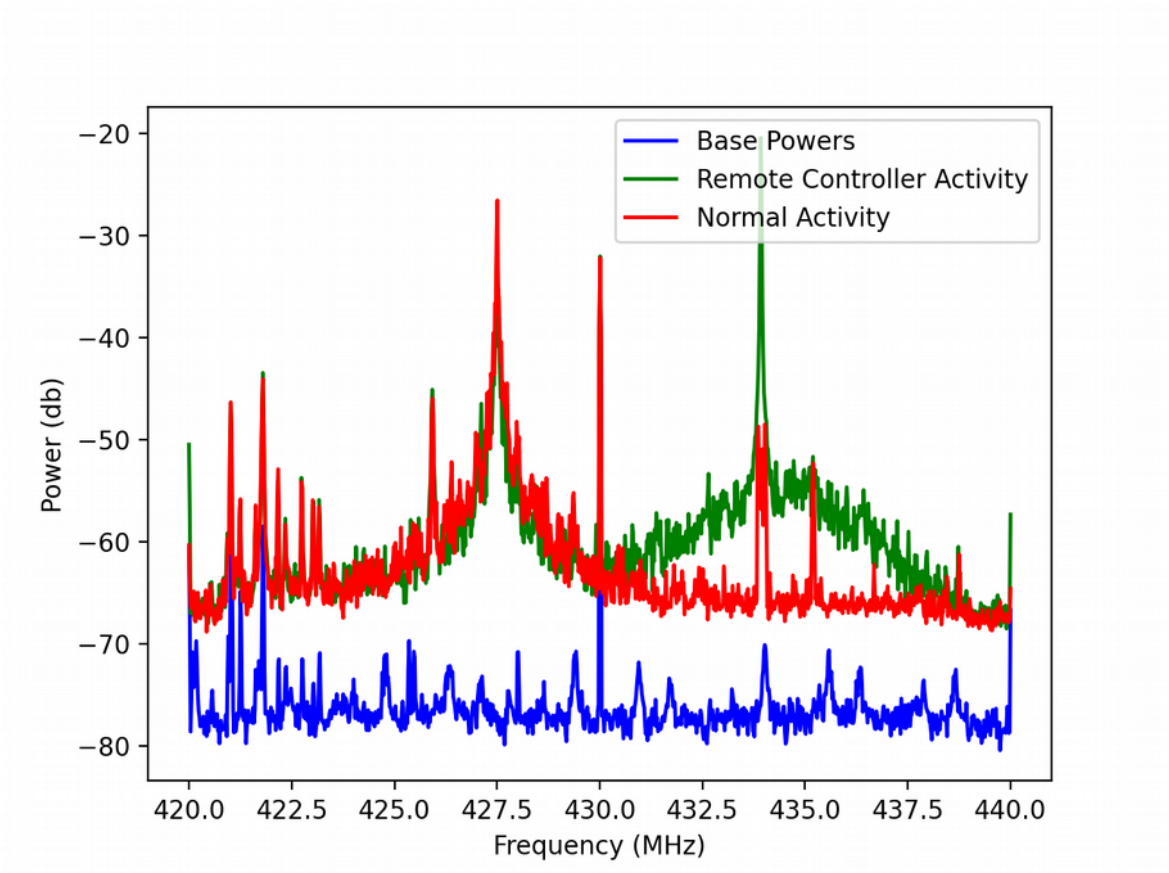


Figure 4.71: Raspberry scan band script test results comparison for 433 MHz band

4.3.7. Raspberry: Drone (2.4 GHz)

The setup for this test is the following: the Raspberry, the HackRF and the DJI Mavic Pro drone with its remote controller. The location and circumstances are the same as the test with laptop, so we expect WiFi and Bluetooth activity in the 2.4 GHz.

Figure 4.72: Raspberry band scan script test setup for drone (2.4 GHz)

Once we get the drone flying and communicating permanently with its remote controller, we can explore the whole band, and we detect a raised signal that is in the 2460-2480 MHz band,just like in the laptop test. This signal occupies around 18 MHz and its center is around 2470 MHz. The behavior is the same as the laptop test remaining raised all the time while the communication lasts between drone and remote controller and going down once the drone lands and the communication is dropped.

As we saw in the laptop test, this signal corresponds to the drone since it has the same characteristics as described in the FCC tests seen in the Figure 4.64.

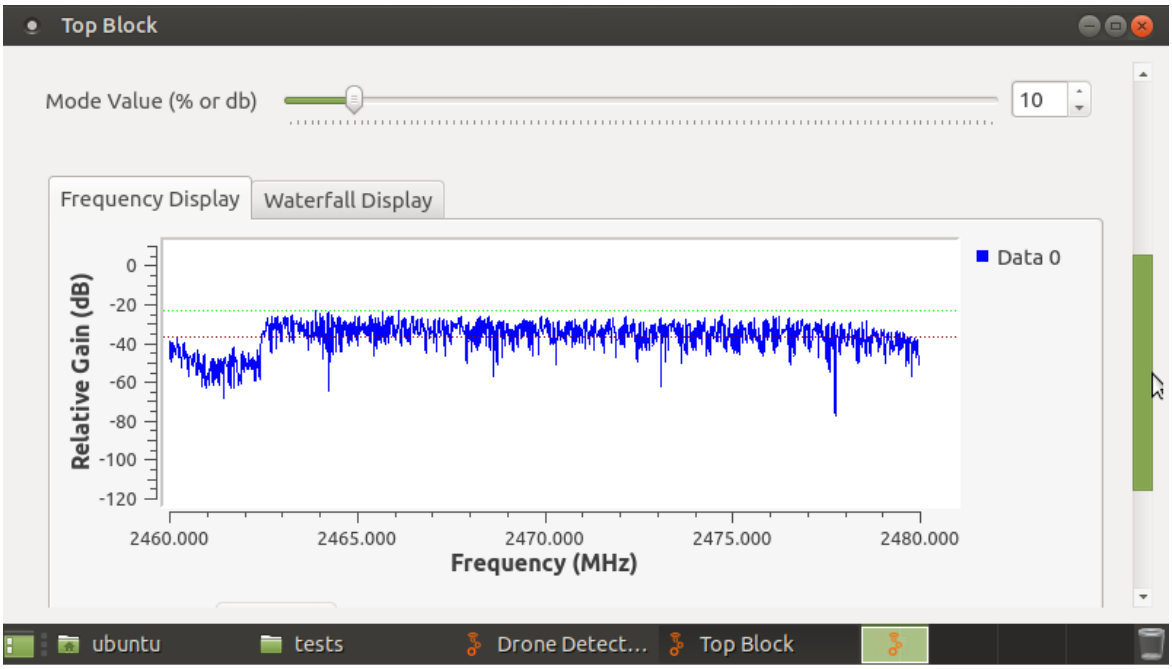


Figure 4.73: Band scan script graphic detection of an active signal in 2.4 GHz.

Since the drone's generated signal differs from the peaky behaviors of the control remote's, we can't identify it easily in the graphic, but nevertheless we see a constant and smooth signal in the last 20 MHz of the 2.4 GHz that definitely correspond to the drone. Comparing with the values obtained under normal circumstances, we see a noticeable difference between the maximum values. However, this analysis that can lead to the conclusion of determining a drone signal can only be done in a graphical manner.

The graphic is generated using a created custom Python function that allows us to compare results from different tests.

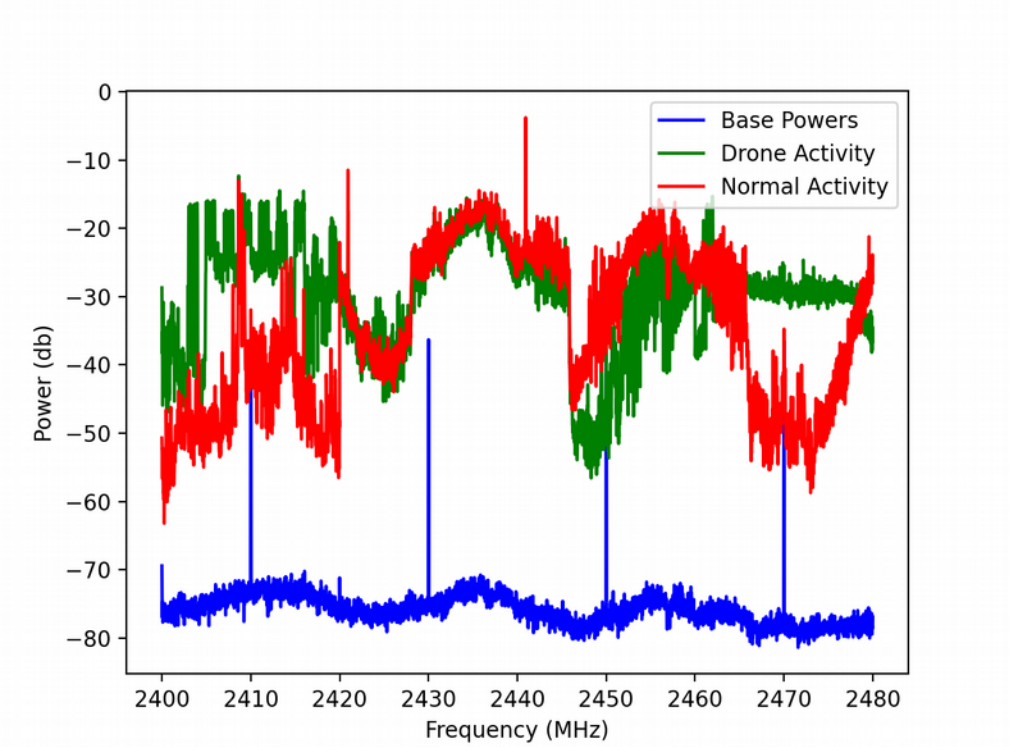


Figure 4.74: Raspberry scan band script test results comparison for 2.4 GHz band

4.3.8. Raspberry Test Analysis for Band Scan Script

In the three test performed we have successfully identified unusual activity in 27 MHz, 433 MHz and 2.4 GHz, and we have come to the conclusion that the identified signals correspond to the ones we have generated with the remote controllers of the toy car, garage and drone, respectively.

The script has proven to be successful in the detection of unusual activity, and has been successfully executed in the Raspberry with no problems. One of the main fears of this script was that it could demand a CPU usage that would make the Raspberry unresponsive, but it was rewarding to see that this script can run without setbacks and can indeed be used to identify graphically signals in real time.

4.4. Jammer Script Tests

In order to test this script we will use the following setup: the Raspberry with a HackRF One executing the jammer script and the laptop with a HackRF One executing the band scan script. We will use the 20 Msps and 1024 FFT configuration parameters. Previously to the test we will use the scan base script test data and will acquire new data that will help us get different reference levels to analyze if the jammer works correctly.

We will get data for 27 MHz, 433 MHz, and 2.4 GHz bands under normal circumstances, then with activity generated by the remote controllers and finally with the jammer script being executed.

Figure 4.75: Jammer script setup test with Raspberry as the jammer source and the laptop as spectrum analyzer.

4.4.1. Jammer test at 27 MHz

First we will execute the jammer with 30 MHz as the center frequency generating a noisy signal of 20 MHz bandwidth. The objective would be to block the communication between the toy remote control and the toy car. This is unsuccessful and the car responds to the commands generated by the controller.

Graphically in the laptop acting as spectrum analyzer we can verify that the HackRF is not capable of generating a significant noise signal that could interfere in the communications.

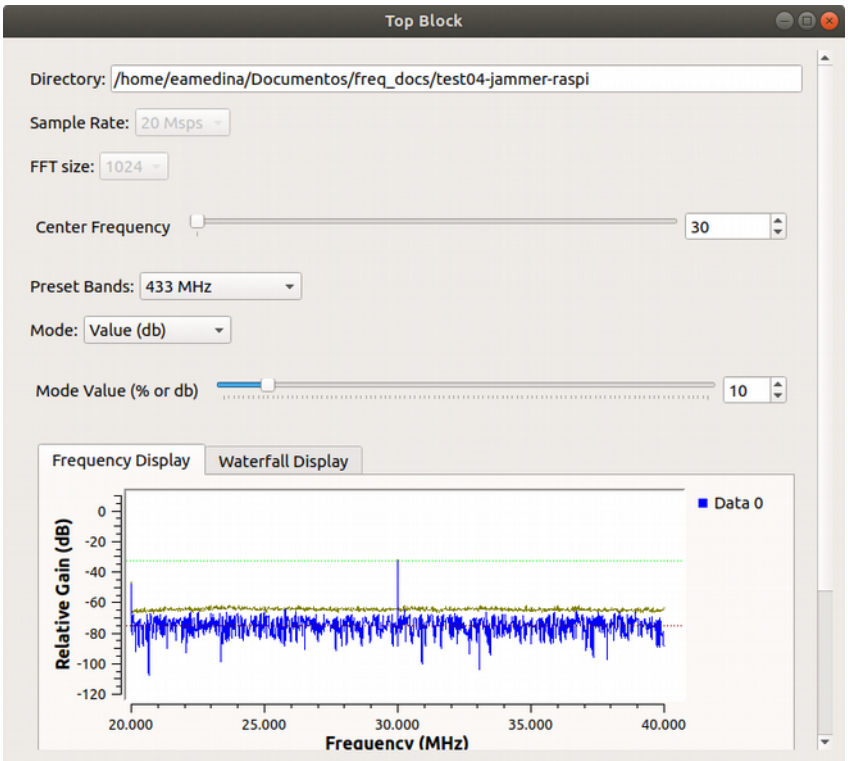


Figure 4.76: Jammer signal at 30 MHz

Even though when analyzing the graphic in real time we can't see any significant difference in the power level with the jammer activated. With the comparison graphic, we can see difference in the power levels with and without the jammer signal, but it seems almost negligible. This explains that the communication between the toy car and remote control didn't get affected.

Even though the HackRF states that it can transmit in frequencies between 1 MHz and 6 GHz, the power we generate from the system is negligible under these testing circumstances.

We weren't able to generate a significant noise signal in 27 MHz band nor interfere the toy car communications.

The graphic is generated using a created custom Python function that allows us to compare results from different tests.

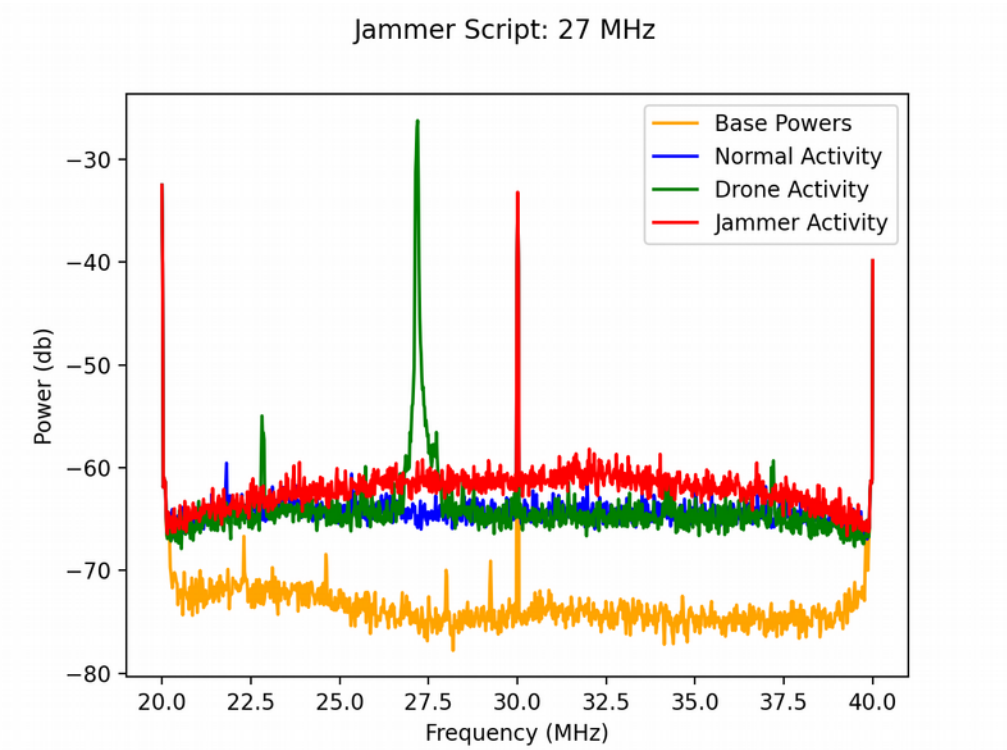


Figure 4.77: Comparison graphic of power values with jammer activity for 27 MHz

4.4.2. Jammer test at 433 MHz

Now we will execute the jammer with 430 MHz as the center frequency generating a noisy signal of 20 MHz bandwidth. Since with the garage remote controller we don't have a receptor, we won't be able to verify that the communication was indeed blocked. So we will focus our analysis only graphically.

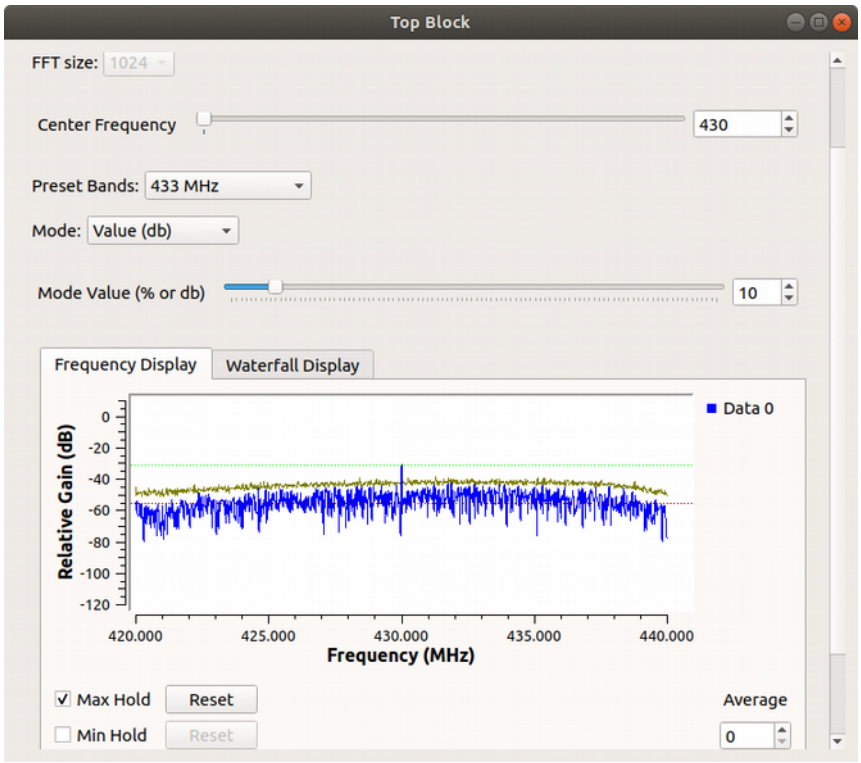


Figure 4.78: Jammer signal at 430 MHz

This time we can see graphically in the laptop, that the HackRF generates a significant noise signal that could interfere in the communications. We see that it has a power around 20 dB greater than the power under normal circumstances. Even though its power isn't greater than the peak of the signal generated by the remote controller.

We have been able to generate successfully an interfering signal in the 433 MHz band with the Raspberry.

The graphic is generated using a created custom Python function that allows us to compare results from different tests.

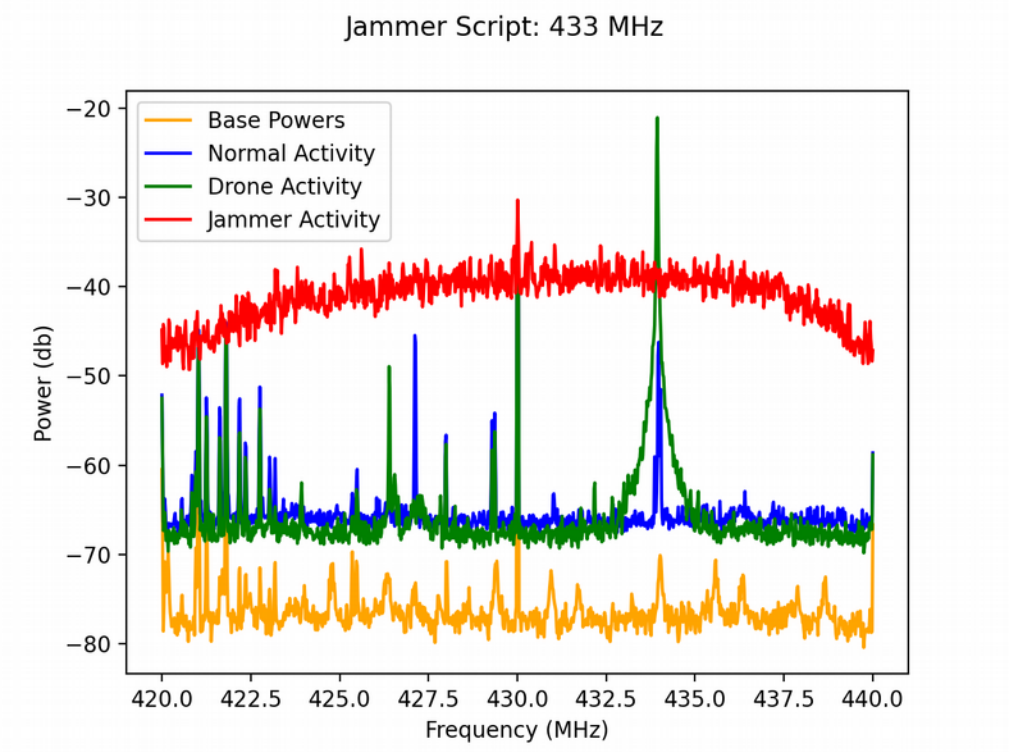


Figure 4.79: Comparison graphic of power values with jammer activity for 433 MHz

4.4.3. Jammer test at 2.4 GHz

Now we will execute the jammer in the 2.4 GHz band with the drone in RC mode.

We have detected the presence of a drone's signal in 2470 MHz, just like in the band script tests, so we will proceed generating the interfering signal with 2470 MHz as the center frequency.

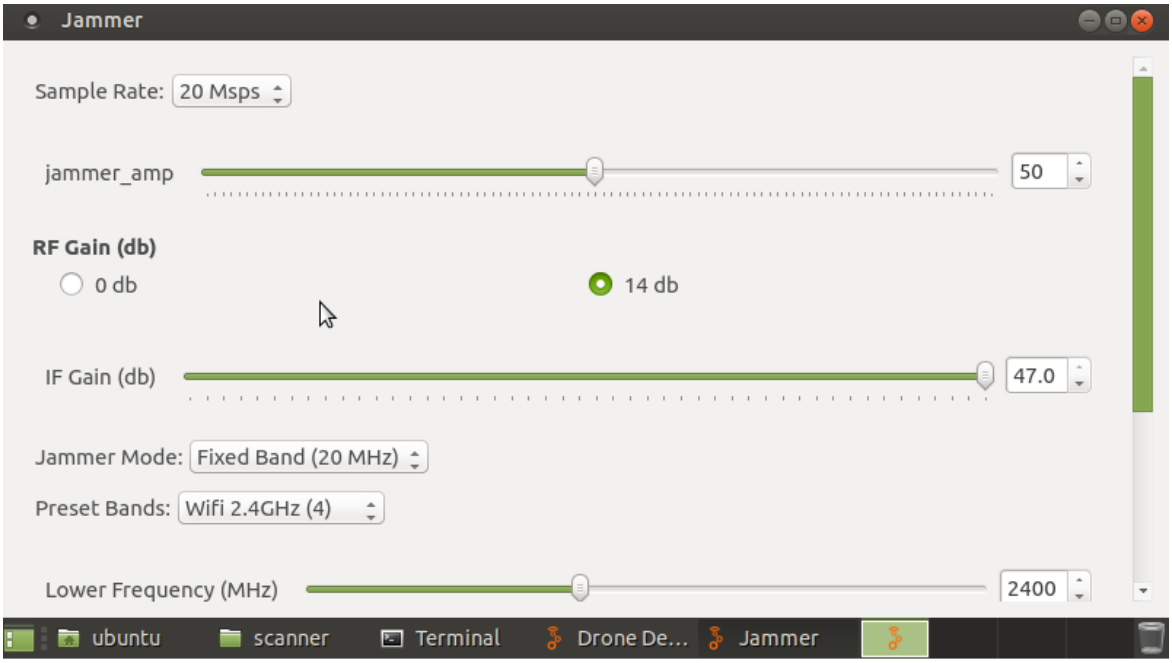


Figure 4.80: Jammer script execution parameters for 2470 MHz run in Raspberry

We can see that the HackRF generates a notorious noise signal in the 2470 MHz band. But we perceive that this signal is intermittent and doesn't remain all the time as a raised signal. Even though, we can see in the remote controller app that this signal indeed causes a noticeable interference to the drone, and it warns the user about it.

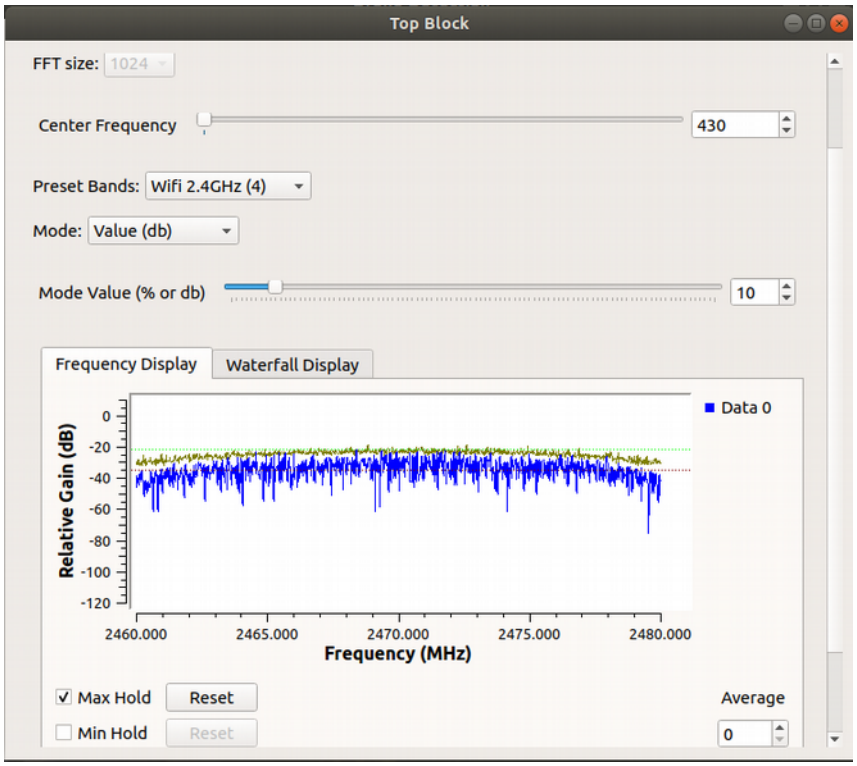


Figure 4.81: Jammer signal at 2470 MHz

Figure 4.82: Warning of interference in the drone remote controller app, caused by the jammer at 2470 MHz.

The drone communications, as we have said before, has different operation modes and when it detects interference in one frequency, moves its communications into another. We were able to detect that behavior when we were interfering at 2470 MHz, where the signal moved to other frequencies and even changed the bandwidth. We could catch the

drone's signal at 20 MHz in 2830 MHz and in 2410 MHz, at 10 MHz in 2476 MHz. According to the RF test of the drone's remote controller done by the FCC, the communications in 20 MHz can use 2410.5 MHz and 2428.5 MHz as center frequencies, and in 10 MHz they can use 2476.5 MHz. This means that the signals that we have caught graphically, correspond to those of the drone.

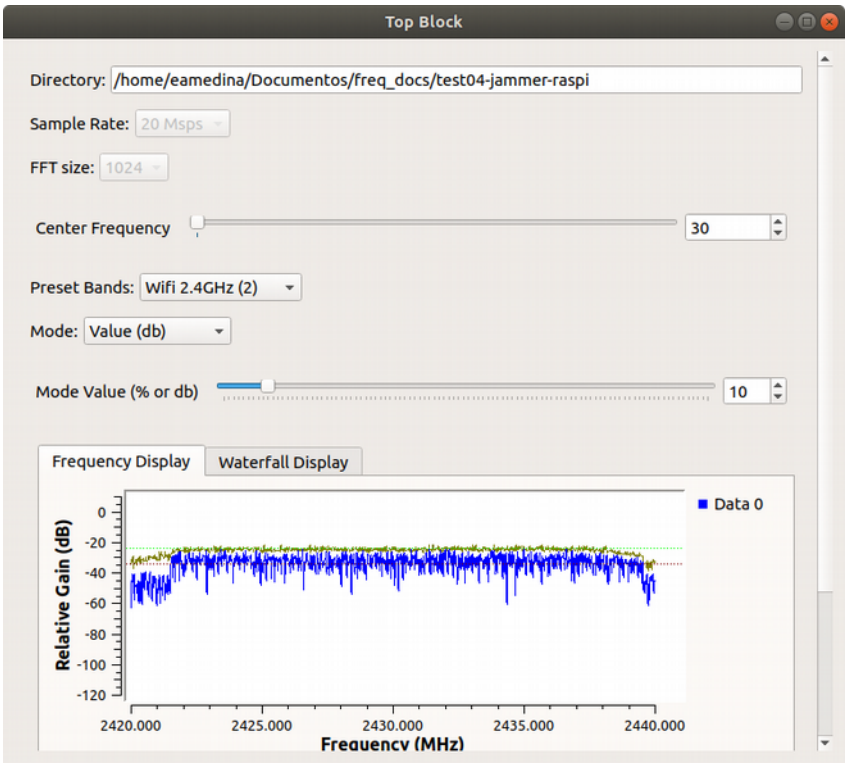


Figure 4.83: Drone operating at 2428.5 MHz – 20 MHz BW

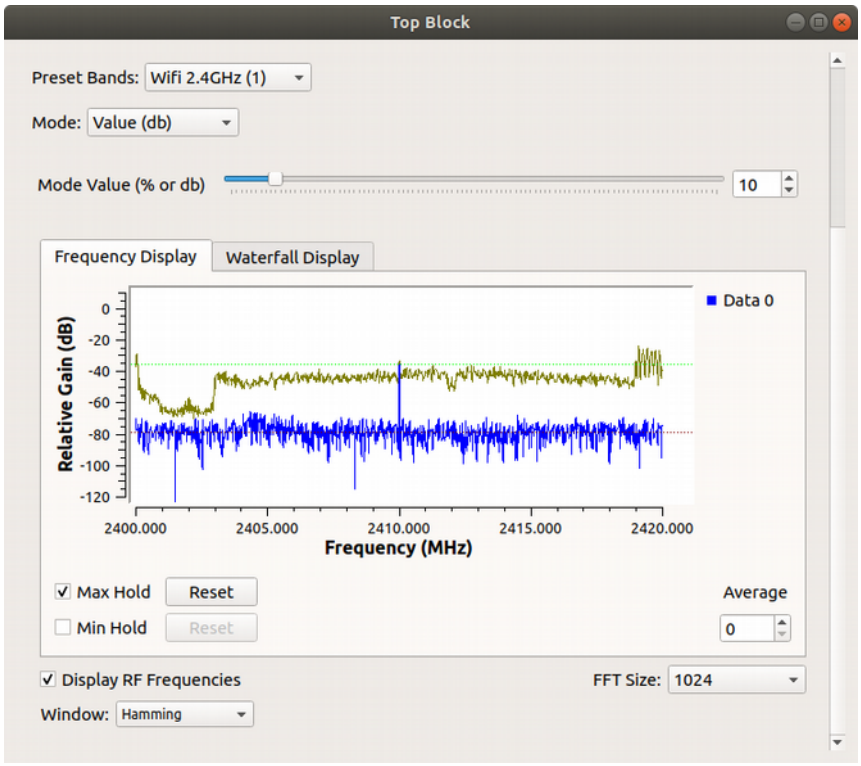


Figure 4.84: Drone operating at 2410.5MHz – 20 MHz BW

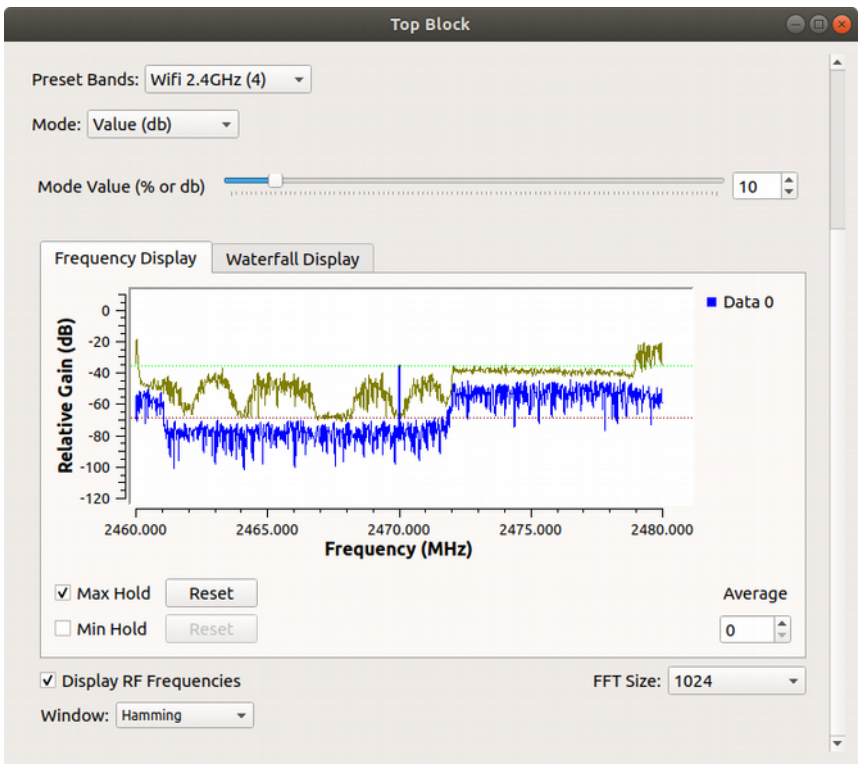


Figure 4.85: Drone operating at 2476.5MHz – 10 MHz BW

Now that we have proved that the drone's communications changes it frequency when it detects interference, which is a behavior we expected, we must perform the test in continuous mode.

This mode changes the center frequency at a time determined by the user, and between a minimum and maximum frequency. We set the time to 50 ms and frequencies between 2400 MHz and 2480 MHz. Even though we generate a noise signal across the whole band, the communication of the drone with the remote controller doesn't get affected and it is not blocked. The drone has the ability to overcome the interference we are generating and continues to be communicated with its remote controller by hopping rapidly its center frequency.

We perform the same test with the drone in WiFi mode, and the result is immediate. The communication is lost, and from the mobile app the connection status appears as disconnected and the drone's lights blink in yellow indicating a lost connection with its remote controller.

We have been able to generate successfully an interfering signal across the whole 2.4 GHz band with the Raspberry, blocking the communication between the remote controller and the drone when it is working in WiFi mode, but when the drone is in RC mode, the communication between the drone and its remote controller couldn't be taken down.

The graphic is generated using a created custom Python function that allows us to compare results from different tests.

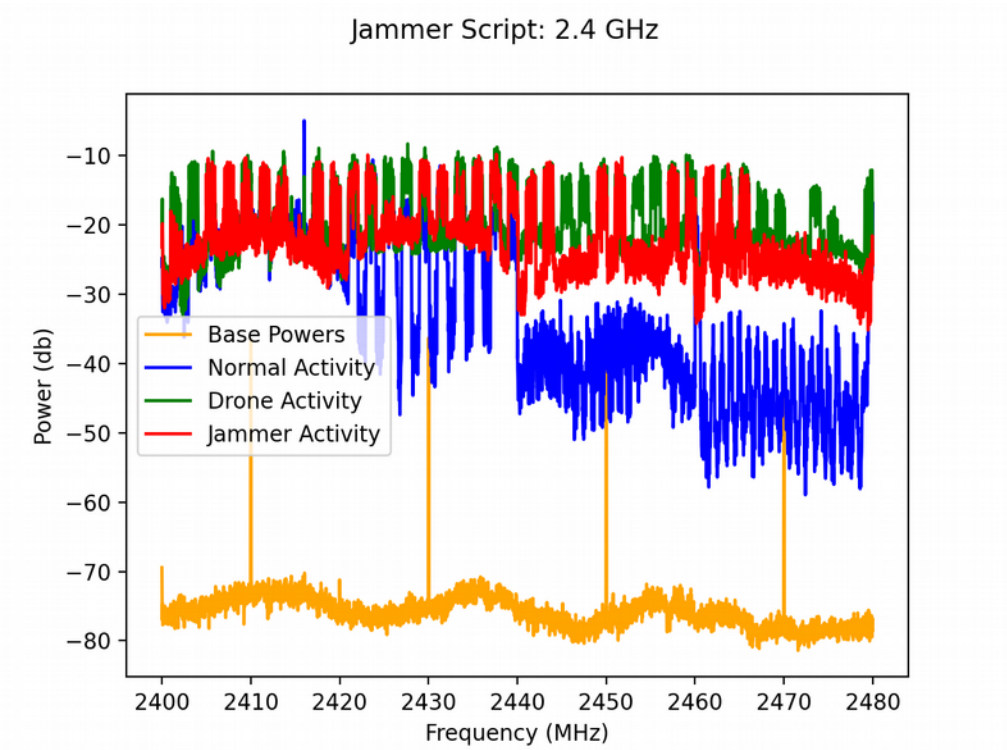


Figure 4.86: Comparison graphic of power values with jammer activity for 27 MHz

5. Retrospective, conclusions and future development

We started this project without knowledge of Python and GNURadio, neither had used an SDR equipment like the HackRF One, so the development of this project was an exercise of acquiring new knowledge and combine the experience in software development to build a software system capable of receiving and transmitting across the spectrum from 1 MHz to 6 GHz, that could be used as a tool to help detect signals generated by drones and to be able to interfere this transmission and take the communications down.

We have to add that the main purpose was that this tool could be used in a Raspberry Pi with a touch screen, so that it could become a portable prototype that could be used in the field without the need to be plugged to the electrical network.

There were many initial obstacles including the setup of the development environment, where we moved from installing GNU Radio in a powerful MacOS laptop, to create a Linux installation in a previously Windows OS laptop. This was done with the objective that our installation would be stable in time and not break with OS updates, as it happened to us while using MacOS .

The selection of the ideal Linux distribution for both the laptop and the Raspberry, came with its setbacks until finally testing and determining that the best working solution was with Ubuntu and Ubuntu Server respectively.

Later it was noticeable that GNURadio and GNURadio Companion had a lot of limitations that would avoid us to build our project directly from the IDE. We saw that this software is used mostly to obtain data and later analyze it, limited to a determined frequency and bandwidth unless changed manually by the user. And we weren't able to find projects with a similar approach to ours, of analyzing in real time a broad part of the spectrum, so we had to build a unique tool with few information to be guided from.

Another big barrier we found was that the HackRF was giving us information at a rate of 10 or 20 million samples per second, and each sample with a size of 16 bytes resulted in big files in the order of hundreds of MB in the first seconds and in the order of the GB after some minutes. This seemed as a non-viable approach specially if were to work on low CPU power, low storage computer like the Raspberry Pi.

The good part of all the difficulties found in the road, was that with the use of creativity we were able to overcome them, and that is how it turned out to be.

The approach taken to develop this project was rather simple: we wanted to have a database of all the power values from 1 MHz to 6 GHz, and then compare them with real time values to check for the presence of unusual activity based on the first measures. In order to create the database we had to make it in our own by creating a custom block that receives the power values and stores them in files. So the creation of the custom blocks *power_analyzer* and *power_comparator* contained not only the recently acquired knowledged of Python and GNURadio, but also the innovative solution to store the data of all the measurable spectrum, independently of the time it would take to make the spectrum analysis. This led us to a result of storing the data with a size ratio as low as 1 MB / GHz for the base scan script, and 1.9 MB / GHZ for the spectrum scan script.

Then to overcome the limitation of working with a unique frequency and not being able to change it without the user interaction, made us modify the scripts generated by the

GNURadio Companion software. The use of automatic tasks in Python in charge of modifying the frequency parameter for all the components involved was the chosen approach. It proved to be the best solution to this problem. It had to be synced so that the execution of the scripts reflect all the parameters changes that the user could request by modifying the controls provided in the interface, modifying values as time, frequency or threshold values. With this approach we came to complete a full spectrum sweep in a time as low as 15 seconds in the laptop and five minutes in the Raspberry for the base scan script. And a full spectrum sweep performed in as low as 75 seconds in the laptop and five minutes in Raspberry for the spectrum scan script.

As a conclusion we have developed successfully a software tool that can be executed in any environment with support for Python and GNURadio,including a Raspberry Pi. The software has proven to be successful in scanning automatically the power levels of frequencies between 1MHz and 6 GHz and creating a database, under different parameters configurations for the sampling rate and the FFT size. It has showed that is able to compare reference values with real time values of powers, and make operations to identify signals with unusual activity that differs from the reference values. This solution has proven that it works in different parts of the spectrum and is able to detect unusual activities not only in the most used bands drones for communications, but in any part of the available spectrum. This indicates that our solution could work detecting drones that use custom RF technologies in frequencies different than the most used bands for UAV communications.

This tool has also proved to be a dynamic software that can adapt to the user needs when searching for a signal. This flexibility is provided by means of controls that can change the parameters of the script while it is running, and among them we find the frequency switch time, the frequency boundaries when doing a continuous scan, the different operation modes to set the threshold, or the operation modes to overcome the bandwidth limitations of the HackRF when transmitting the jamming signal. We have also provided the user with graphical tools to analyze the spectrum in real time, or to analyze the already gathered data. And a mean to explore the comparison data in a table with different sorting options.

We have created successfully a schema to save the data of the performed spectrum scans, that doesn't increase the size of files with time, assuring the operation of the system during long periods of time in the executing machine. This file schema has the feature that can be easily interpreted by a human, and was intended to do so, because we needed a mean of proving that the data that was being stored corresponded to what we were measuring. This file format can be improved in case we need a more compact size ratio per GHz analyzed.

We have created and used successfully three custom GNURadio blocks that perform operations with the data obtained from the HackRF. These blocks also generate a custom file database with power values and power differences, so we can identify unusual activity in the spectrum.

We have successfully blocked communications between a drone and its remote controller by generating a noise signal with the HackRF, when the drone is working in WiFi mode. We failed to block completely the communications in RC mode, due to the fast frequency hopping algorithm of the communications protocol. Even if we generate the noise across the whole 2.4 GHz band hopping our noise signal, the hopping protocol of the UAV is a lot faster that the hopping time that we can achieve. So it would be necessary to block the

whole 2.4 GHz band in order to block completely the communications. This could be achieved by deploying more Raspberry devices with their respective HackRF.

We have provided the capability to block communications not only in the 2.4 GHz band, but also across the available spectrum. We tried jamming the GNSS frequencies with success, disabling the reception of GPS signal in the mobile smartphone used with the drone remote controller. These tests were not included in chapter four, because our testing drone doesn't have autonomous flight capabilities.

We have proved that it is possible to use a Raspberry Pi, to execute all the scanning and jamming scripts, and act as the controller unit of a drone detector and jammer. Even though the execution of these scripts can be performed without major issues, there are some moments in which the Raspberry Pi remains unresponsive for some seconds, but we have seen that it also happens executing other tasks. So it is advisable that this project be ported to the newly released Raspberry Pi 4 which offers a more powerful processor and more RAM memory.

Since this solution is using GNURadio as the telecommunications framework, provides us with the flexibility to change the SDR pheripheral. The HackRF provides us with a wide range of reception and transmission, but with a low IQ resolution, and a rather low transmission power. But this device can be replaced for a better one with improved characteristics and the created software wouldn't need to be changed in order to execute all the scripts.

We have also provided annexes with summarized steps for an easy installation and setup of all this projects components. Even though they seem like simple straightforward steps, there were many tests and hours invested into finding the right OS for the Raspberry Pi, so that the development tools could be installed and the scripts executed without setbacks.

We also provide the source code for all the generated elements in this project including the *xml* and Python files of the custom GNURadio blocks and the Python files of the scripts. With simple copy/pase actions and following the respective setup annex, any Raspberry could be used to execute our software. Even though there are a few steps to perform this setup, there are a lot of hours invested in generating the respective GNURadio blocks and Python scripts.

There are many improvements that can be made to this project, starting by implementing an automatic real time drone detector with the help of machine learning algorithms or artificial intelligence. Obtaining samples of the DJI Mavic Pro in operation can help build a model of the RF communications. And later in real time with the help of Python libraries, we could try to detect these models in real time. This can be done taking into account the computation power of the Raspberry used.

It can be possible that digital signal processing techniques can be applied to the data obtained from the HackRF so values are more realistic. With the help of DSP techniques we could explore performing detection of drone signals by using certain types of filters. Also one thing that can be done is to create a block that suppresses the DC offset from the HackRF data, so we could get a cleaner image of the spectrum.

Another room for improvements is the antenna configuration. We have used an omnidirectional antenna for both reception and transmission. But in the case of transmitting the jamming signal, we could use a directional antenna to have better results.

Also the use of amplifiers can improve the received and transmitted power of our current system.

In short, this project has proven to be a working starting point for a more robust drone detection system. The basis for future work base on this project have been set, and it would be desirable that more projects are released to improve the work already done here.

Bibliography

[1]“La presencia de drones provoca el cierre del aeropuerto de Madrid-Barajas durante hora y media,” , 03-Feb-2020. [Online]. Available: https://www.eldiario.es/tecnologia/Cierra-aeropuerto-Barajas-detectarse-inmediaciones_0_991801124.html}. [Accessed: 04-Apr-2020].

[2]“¿Cómo Frenaron Los Inhibidores A Los Drones De Tsunami En El Camp Nou?,” 19-Dec-2019. [Online]. Available: https://web.archive.org/web/20200510125333/https://www.lavanguardia.com/tecnologia/20191219/472359865897/inhibodores-drone-dron-partido-camp-nou-tsunami-democratic.html. [Accessed: 04-Apr-2020].

[3]“Los Mossos Detectan 85 Drones No Autorizados Sobrevolando Barcelona Durante Las Protestas,” La Vanguardia , 20-Oct-2019. [Online]. Available: https://www.lavanguardia.com/politica/20191020/471092218804/mossos-detectan-85-drones-protestas-barcelona-no-autorizados.html. [Accessed: 04-Apr-2020].

[4]M. Šustek and Z. Úředníček, “The Basics of Quadcopter Anatomy,” MATEC Web of Conferences, vol. 210, p. 01001, Jan. 2018, doi: 10.1051/matecconf/201821001001.

[5]Tech Insider, “Drone Could Help Firefighters By Putting Out Fires,” YouTube , 05-Apr-2018. [Online]. Available: https://web.archive.org/web/20200510125808/https://www.youtube.com/watch?v=Bm2BVTTir4c. [Accessed: 04-Apr-2020].

[6]ABC News, “Heat-seeking Drone Finds Missing 6-year-old Minnesota Boy In Cornfield,” ABC News , 16-Oct-2019. [Online]. Available: https://abcnews.go.com/US/drone-finds-missing-year-minnesota-boy-cornfield/story?id=66327536. [Accessed: 05-Apr-2020].

[7]I. Guvenc, O. Ozdemir, Y. Yapici, H. Mehrpouyan, and D. Matolak, “Detection, localization, and tracking of unauthorized UAS and Jammers,” 2017, pp. 1–10, doi: 10.1109/DASC.2017.8102043.

[8]S. Jeon, J.-W. Shin, Y.-J. Lee, W.-H. Kim, Y. Kwon, and H.-Y. Yang, “Empirical study of drone sound detection in real-life environment with deep neural networks,” 2017, pp. 1858–1862, doi: 10.23919/EUSIPCO.2017.8081531.

[9]J. Wei Lin, “Civil UAV monitoring techniques,” State Radio Monitoring Center of China, May 2020.

[10]“DSM2: Spektrum - The Leader In Spread Spectrum Technology,” Spektrum . [Online]. Available: https://web.archive.org/web/20200517150022/https://www.spektrumrc.com/Technology/DSM2.aspx. [Accessed: 05-May-2020].

[11]“DSMX: Spektrum - The Leader In Spread Spectrum Technology,” Spektrum . [Online]. Available: https://web.archive.org/web/20200517150329/https://www.spektrumrc.com/Technology/DSMX.aspx. [Accessed: 05-May-2020].

[12]“FrSky Advanced Communication Control Elevated Spread Spectrum - ACCESS Protocol Release! - FrSky - Lets You Set The Limits,” FrSky - Lets You Set The Limits , 22-Mar-2019. [Online]. Available: https://www.frsky-rc.com/frsky-advanced-communication-control-elevated-spread-spectrum-access-protocol-release/. [Accessed: 05-May-2020].

[13]“Protocol Information - FutabaUSA,” FutabaUSA . [Online]. Available: https://web.archive.org/web/20200517150941/https://futabausa.com/protocols/. [Accessed: 05-May-2020].

[14]“AirLink - DJI Mobile SDK Documentation,” DJI . [Online]. Available: https://web.archive.org/web/20200517151257/https://developer.dji.com/mobile-sdk/documentation/introduction/component-guide-airlink.html. [Accessed: 05-5-5].

[15]“Hikvision UAV-D04JAI - UAV Defender,” Hikvision . [Online]. Available: https://www.hikvision.com/mtsc/uploads/product/accessory/UAV-D04JAI__ES.pdf. [Accessed: 08-Apr-2020].

[16]Policía Nacional, “Rifle Anti-drones,” @policia On Twitter , 25-Sep-2019. [Online]. Available: https://twitter.com/policia/status/1176230756126838785. [Accessed: 08-Apr-2020].

[17]“Una Veintena De Drones Sobrevuela Madrid Cada Día, La Mayoría Se Sanciona,” La Vanguardia , 08-Feb-2020. [Online]. Available: https://www.lavanguardia.com/vida/20200208/473357962746/una-veintena-de-drones-sobrevuela-madrid-cada-dia-la-mayoria-se-sanciona.html. [Accessed: 04-May-2020].

[18]“Heathrow Airport Installs Anti-Drone System to Detect Threats,” Bloomberg , 14-Jan-2020. [Online]. Available: https://www.bloomberg.com/news/articles/2020-01-14/heathrow-airport-gets-thales-anti-drone-system-to-detect-threats. [Accessed: 04-May-2020].

[19]Raytheon Corporate, “Raytheon: Raytheon Delivers First Laser Counter-UAS System To U.S. Air Force - Oct 22, 2019,” Raytheon News Release Archive , 22-Oct-2019. [Online]. Available: http://raytheon.mediaroom.com/2019-10-22-Raytheon-delivers-first-laser-counter-UAS-System-to-U-S-Air-Force. [Accessed: 04-May-2020].

[20]P. Nguyen, M. Ravindranatha, A. Nguyen, R. Han, and T. Vu, “Investigating Cost-effective RF-based Detection of Drones,” 2016, pp. 17–22, doi: 10.1145/2935620.2935632.

[21]M. Haluza and J. Čechák, “Analysis and decoding of radio signals for remote control of drones,” in 2016 New Trends in Signal Processing (NTSP), 2016, pp. 1–5.

[22]M. F. Al-Sa’d, A. Al-Ali, A. Mohamed, T. Khattab, and A. Erbad, “RF-based drone detection and identification using deep learning approaches: An initiative towards a large open source drone database,” Future Generation Computer Systems, vol. 100, pp. 86–97, 2019, doi: 10.1016/j.future.2019.05.007.

[23]M. Ezuma, F. Erden, C. Anjinappa, O. Ozdemir, and I. Guvenc, Micro-UAV Detection and Classification from RF Fingerprints Using Machine Learning Techniques. .

[24]J. Duarte García, Software Defined Radio for Wi-Fi Jamming. .

[25]K. Pärlin, M. Alam, and Y. Le Moullec, “Jamming of UAV Remote Control Systems Using Software Defined Radio,” 2018, doi: 10.1109/ICMCIS.2018.8398711.

[26]R. Price, “There’s Now A Way To Hijack Nearly Any Drone Mid-flight Using A Tiny Gadget,” Insider , 27-Oct-2016. [Online]. Available: https://www.insider.com/icarus-gadget-hijack-almost-any-drone-mid-flight-2016-10. [Accessed: 05-May-2020].

[27]“Open Source Mobile Communications,” Osmocom. [Online]. Available: https://web.archive.org/web/20200517161352/https://osmocom.org/. [Accessed: 05-May-2020].

[28]M. Ossmann, “HackRF One,” Great Scott Gadgets . [Online]. Available: https://greatscottgadgets.com/hackrf/one/. [Accessed: 04-Apr-2020].

[29]M. Ossmann, “HackRF One - Wiki,” GitHub. [Online]. Available: https://github.com/mossmann/hackrf/wiki/HackRF-One. [Accessed: 04-Apr-2020].

[30]M. Ossmann, “HackRF One - FAQ,” GitHub. [Online]. Available: https://github.com/mossmann/hackrf/wiki/FAQ. [Accessed: 04-Apr-2020]

[31]M. Ossmann, “Mossmann/hackrf,” GitHub. [Online]. Available: https://github.com/mossmann/hackrf/wiki/libHackRF-API. [Accessed: 04-Apr-2020].

[32]“Raspberry Pi 3 Model B+,” Raspberry Pi.[Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/. [Accessed: 04-Apr-2020]

[33]“DJI Mavic Pro – Epecificaciones, FAQ, Tutoriales y Guías – DJI,” DJI Official . [Online]. Available: https://www.dji.com/es/mavic/info.

[34]“FCC ID SS3-GL200A1606 C2 By SZ DJI TECHNOLOGY CO., LTD,” FCC , 2016. [Online]. Available: https://fccid.io/SS3-GL200A1606. [Accessed: 04-Apr-2020].

[35]“What Is GNU Radio?,” GNU Radio. [Online]. Available: https://wiki.gnuradio.org/index.php/What_is_GNU_Radio%3F. [Accessed: 04-Apr-2020]

[36]“Hardware - GNU Radio,” GNURadio . [Online]. Available: https://wiki.gnuradio.org/index.php/Hardware. [Accessed: 04-Apr-2020].

[37]H. Muhammad, “Htop - An Interactive Process Viewer For Unix.” [Online]. Available: https://hisham.hm/htop/. [Accessed: 05-May-2020].

[38]“Hamming Window - An Overview | ScienceDirect Topics,” ScienceDirect . [Online]. Available: https://www.sciencedirect.com/topics/engineering/hamming-window. [Accessed: 05-May-2020].

Annex

Annex I – Raspberry setup

Get ubuntu server for Raspberry Pi 3 from official page, burn into sd card and run in Raspberry. Update OS.

$ sudo apt update && sudo apt upgrade

Install taskel for desktop selections.

$ sudo apt install tasksel

Open tasksel

$ sudo tasksel

Select ubuntu mate minimal and install

Annex II - GNURadio setup (Linux-Ubuntu)

Open terminal and install gnuradio

$ sudo apt-get install gnuradio

Install osmosdr

$ sudo apt-get install gr-osmosdr

Install make

$ sudo apinstall cmake

Run GNURadio Companion

$ gnuradio-companion

Annex III - Custom GNURadio block setup

Go to Documents or any directory where you want to locate the custom blocks and scripts. For this annex we create a gnuradio folder inside Documents.

Create a package.

$ gr_modtool newmod tfm

Go to the recently created gr-tfm folfer

Create the custom blocks

$ gr_modtool add -t sync -l python power_analyzer_ff
$ gr_modtool add -t sync -l python power_comparator_ff
$ gr_modtool add -t hier -l python logpowerfft_win

Go to grc folder and replace the custom blocks xml file.

tfm_power_analyzer_ff.xml
tfm_power_comparator_ff.xml
tfm_logpowerfft_win.xml

Go to python filder and replace the custom blocks python scripts.

power_analyzer_ff.py
power_comparator_ff.py
logpowerfft_win.py

Go back to gr-tfm and create a folder with name build and then go to it

Configure cmake

$ cmake ../
$ make
$ sudo make install

Now the blocks are available and recognizable by GNURadio

Annex IV - Scripts setup

Create the scripts from source code. The names must be respected. Make the scripts executable

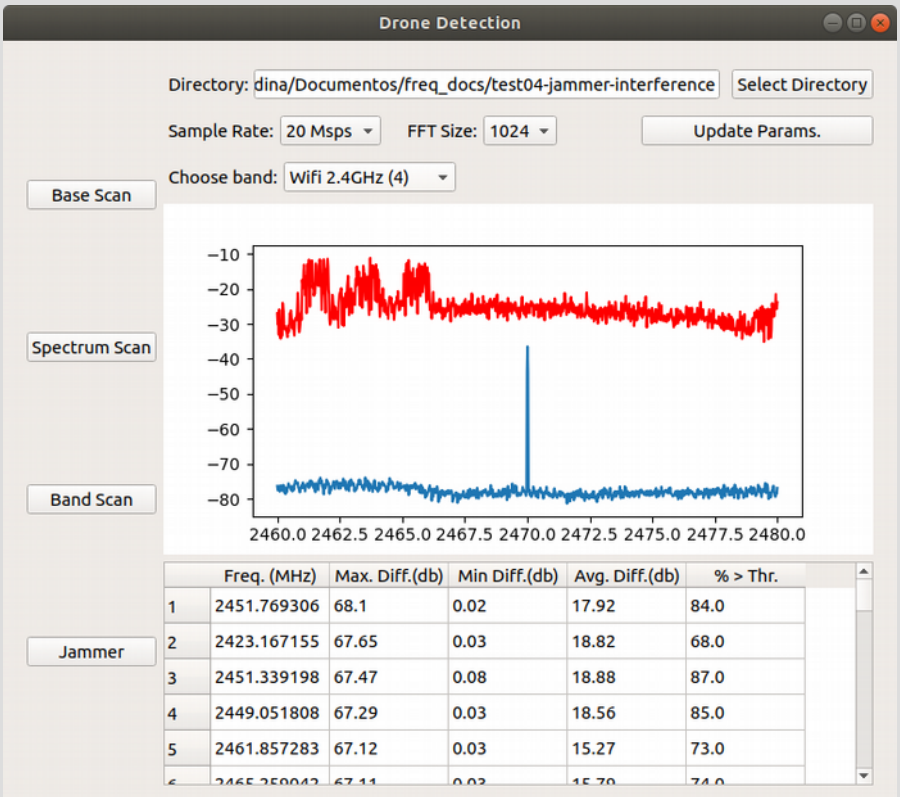
```
$ chmod +x 00-main.py
$ chmod +x 01-scan-base.py
$ chmod +x 02-scan-spectrum.py
$ chmod +x 03-scan-band.py
$ chmod +x 04-jammer.py
```

Now execute script 00

```
./00-main.py
```

Annex V – User Manual

Main Script



Directory: the folder where the files generated by the scripts are stored. It is recommended that for every new data acquisition, a new folder is created. Pushing the button “Select directory” allows the user to select a folder.

Sample Rate: two options of sample rate in the range of Million sample rate per second (Msps). 10 Msps or 20 Msps. This value indicates the bandwidth: 10 MHz or 20 MHz.

FFT Size: the size of bins in which the signal will be divided after FFT. 1024 or 2048 are the options available.

Choose band: indicates the mode at which the graphic works.

In “CONTINUOUS” mode, it will display one graphic at a time for each of the frequency files stored in the directory, and after 3 seconds will move on to display the next frequency graphic.

In “ALL” mode, it will display all the data found in the directory in one single graphic, usually all the files corresponding to frequencies from 1 MHz to 6 GHz. It also provides additional controls to set the lower and upper frequencies that must be displayed in the graphic.

There are also additional modes, that will display the graphic with preset frequencies, where drone activity is expected such as 433 MHz, 868 MHz and the whole 2.4 GHz.

Graphic: The graphic displays the power values obtained from the base scan script in blue color and the power values from the spectrum and band scan script in red values. All values are in dBm. The graphic will let us verify visually where in the spectrum we can find unusual RF activity.

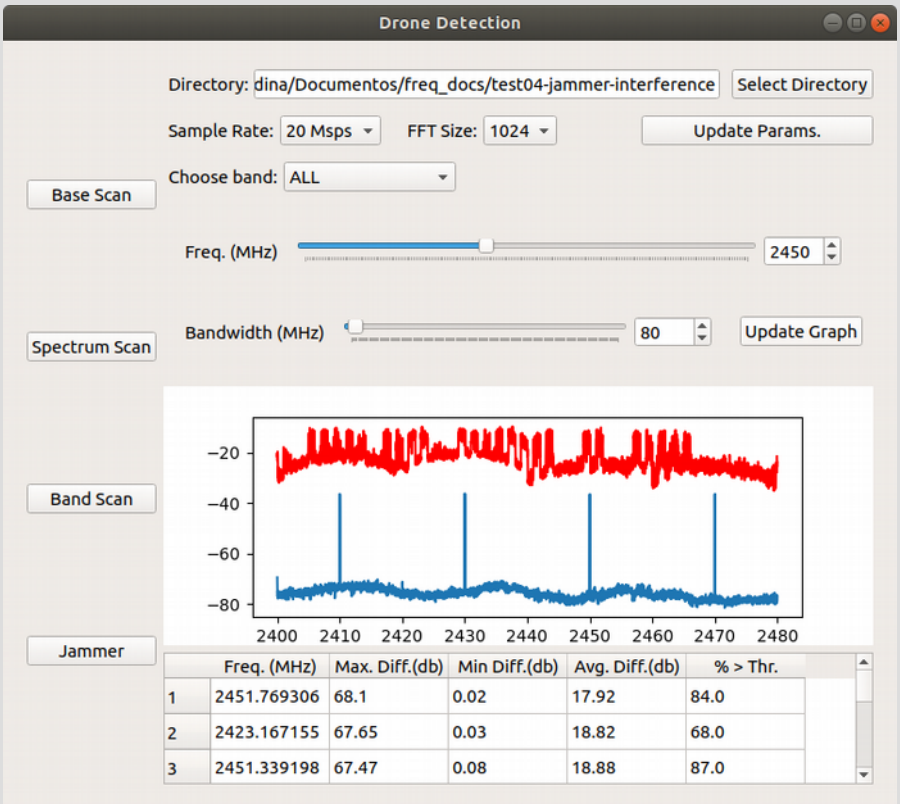
Table: The table displays all the data obtained from the spectrum scan script with the use of the data obtained from the base scan script. Here we find the frequency and the values that have exceede the threshold, with their respective maximum difference from the base, the minimum difference and the average difference. Also here is displayed the percentage of values that have been above the threshold, out of the total received values.

Base Scan: launches the execution of the base scan script to obtain the base averaged power values for all frequencies between 1 MHz and 6 GHz. The sample rate, FFT size and directory parameters will be passed to the script for the execution. This must be the first script to be executed, since we will always need the base power values for the others scanning scripts to work.

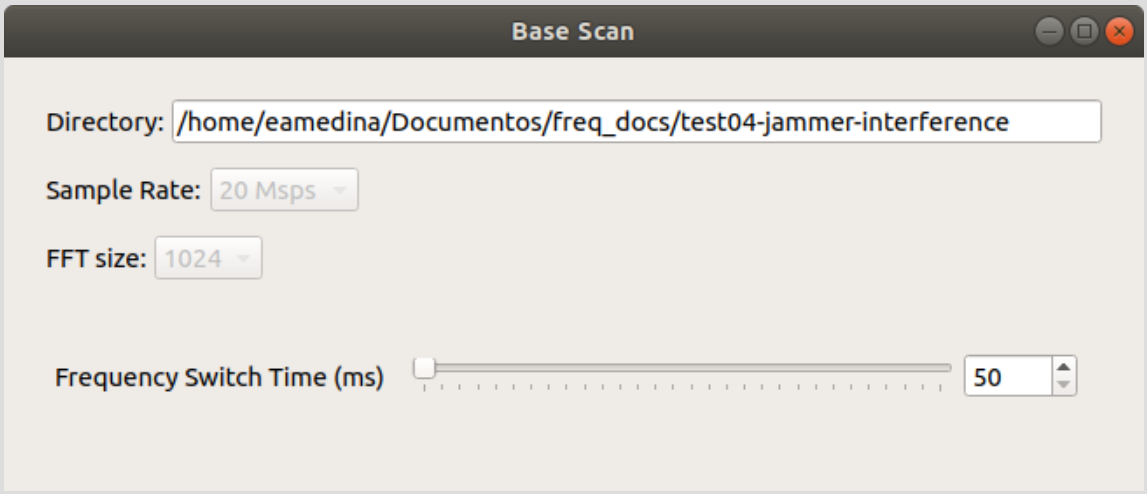
Spectrum Scan: launches the execution of the spectrum scan script to analyze the real time power values and compare them with the base power values. Initially the spectrum scan is for frequencies between 1 MHz and 6 GHz, but the user can change the lower and upper frequency, so the scan can be focused on bands of interest. The sample rate, FFT size and directory parameters will be passed to the script for the execution.

Band Scan: launches the execution of the band scan script to analyze the real time power values and compare them with the base power values. Contrary to the spectrum scan script, we are focused in a given bandwidth and not the whole spectrum available. This script also provides a graphical interface to see in real time the power values. The sample rate, FFT size and directory parameters will be passed to the script for the execution.

Jammer: launches the execution of the jammer script, which is in charge of generating a noise signal within a bandwidth determined by the user. This script is used to interfer or block the communications between a drone and its remote controller.



Base Scan



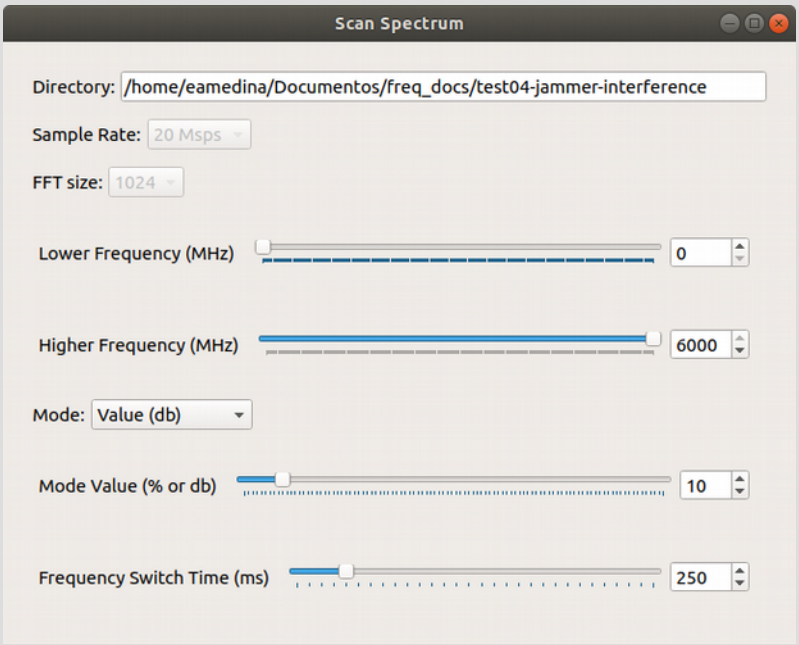
Directory: the folder where the files generated by this script are stored. Files will be generated with the *power* suffix and will be stored in *txt* format. This parameter is passed by the main script.

Sample Rate: This value indicates the sample rate and the bandwidth in MHz. This parameter is passed by the main script.

FFT Size: the size of bins in which the signal will be divided after FFT. This parameter is passed by the main script.

Frequency Switch Time (ms): The time the source generates data for a given frequency before moving to the next one. A lower time will produce that a complete spectrum sweep is done faster. A greater time will produce that each frequency has more time to get its data, but will take longer to sweep the whole spectrum.

Spectrum Scan



Directory: the folder where the files generated by this script are stored. Files will be generated with the *compare* suffix and will be stored in *txt* format. This parameter is passed by the main script.

Sample Rate: this value indicates the sample rate and the bandwidth in MHz. This parameter is passed by the main script.

FFT Size: the size of bins in which the signal will be divided after FFT. This parameter is passed by the main script.

Lower Frequency (MHz): This value indicates the minimum frequency at which the scan will be performed. By default this value is set to 0. This value can be modified by the user to focus on bands of interest and not in the whole spectrum available.

Higher Frequency (MHz): This value indicates the maximum frequency at which the scan will be performed. By default this value is set to 0. This value can be modified by the user to focus on bands of interest and not in the whole spectrum available.

Mode: Indicates the operation mode used to set the threshold value, at which all power values above it will be considered as unusual activity and saved to the file database as an exceeded power value.

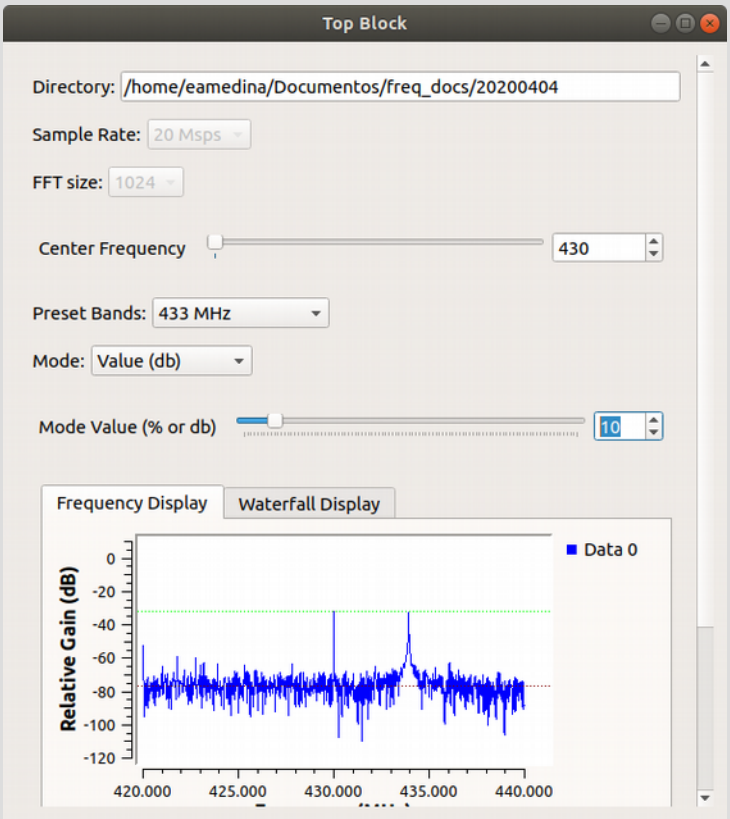
The two available modes are Percentage and Fixed. Percentage takes the base value and calculates its treshold based on the percentage value set. If it is set to 10%, all values above 10% of the base value will be considered as unusual.

Fixed mode establishes the threshold as a fixed dB value above the base for each frequency.

Mode value: Indicates the value that will help calculate the threshold. If the mode is percentage it will indicate a percentage value above the base. If the mode is fixed, it will indicate a dB value above the base.

Frequency Switch Time (ms): The time the source generates data for a given frequency before moving to the next one. A lower time will produce that a complete spectrum sweep is done faster. A greater time will produce that each frequency has more time to get its data, but will take longer to sweep the whole spectrum.

Band Scan



Directory: the folder where the files generated by this script are stored. Files will be generated with the *compare* suffix and will be stored in *txt* format. This parameter is passed by the main script.

Sample Rate: this value indicates the sample rate and the bandwidth in MHz. This parameter is passed by the main script.

FFT Size: the size of bins in which the signal will be divided after FFT. This parameter is passed by the main script.

Center Frequency (MHz): the center frequency at which this script operates. This parameter can be changed by the user and in consequence the graphic and the file generated will also reflect the changes.

Preset bands: configuration that will set this script's operating frequency to preset frequencies, where drone activity is expected such as 433 MHz, 868 MHz and the whole 2.4 GHz band.

Mode: Indicates the operation mode used to set the threshold value, at which all power values above it will be considered as unusual activity and saved to the file database as an exceeded power value.

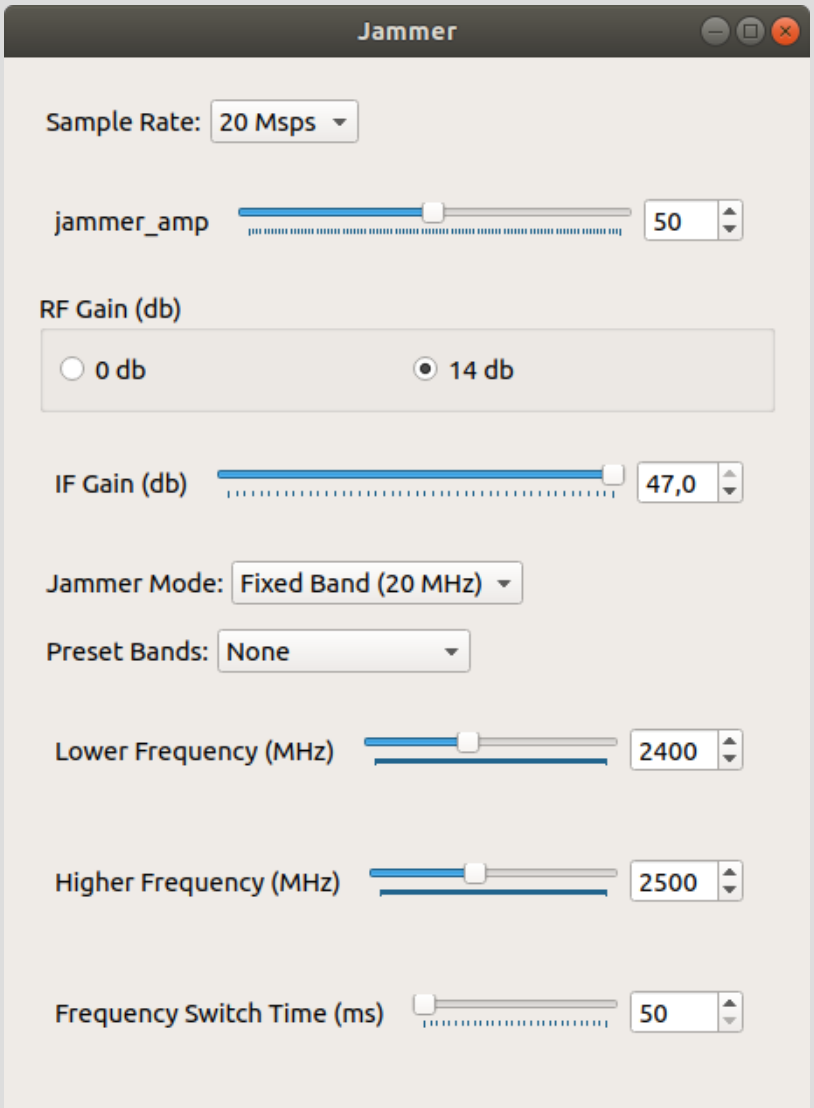
The two available modes are Percentage and Fixed. Percentage takes the base value and calculates its treshold based on the percentage value set. If it is set to 10%, all values above 10% of the base value will be considered as unusual.

Fixed mode establishes the threshold as a fixed dB value above the base for each frequency.

Mode value: Indicates the value that will help calculate the threshold. If the mode is percentage it will indicate a percentage value above the base. If the mode is fixed, it will indicate a dB value above the base.

Graphic: Has a real time representation of the power values received by the SDR. We can see a FFT plot or a waterfall plot. This graphic helps in the detection of signal activity in a given frequency band.

Jammer



Sample Rate: this value indicates the sample rate and the bandwidth in MHz. This parameter is passed can be selected by the user between 10 Msps and 20 Msps.

Jammer Amplitude: amplitude of the generated noise signal.

RF Gain, IF Gain: software defined gains in dB available by the SDR device.

Jammer Mode: the operating mode of the script. It can be fixed or continuous. In fixed mode we generate a noise signal of the chosen bandwidth that is transmitted in a

frequency determined by the value set in Lower Frequency parameter. In continuous mode, the noise signal can be transmitted at different frequencies, hopping between the values set in Lower and Upper frequencies. The hopping time is set by the Frequency Switch Time. The continuous mode helps us transmit the noise signal in a band greater than 20 MHz.

Preset bands: configuration that will set this script's operating frequency to preset frequencies, where drone activity is expected such as 433 MHz, 868 MHz, the whole 2.4 GHz band and GNSS bands.

Lower Frequency (MHz): This value in fixed mode indicates the frequency at which the noise will be transmitted. In continuous mode indicates the minimum frequency at which the noise will be transmitted. By default this value is set to 2400 MHz.

Higher Frequency (MHz): This value indicates the maximum frequency at which the noise will be transmitted. This parameter is only taken into account when the script is operating in continuous mode. By default this value is set to 2500 MHz.

Frequency Switch Time (ms): The time the source generates the noise for a given frequency before moving to the next one. This parameter is only taken into account when the script is operating in continuous mode.

Annex VI – GNURadio custom block: tfm_logpowerfft_win.xml

```
<?xml version="1.0"?>
<block>
  <name>logpowerfft_hamming</name>
  <key>tfm_logpowerfft_win</key>
  <category>[tfm]</category>
  <import>import tfm</import>
        <make>tfm.logpowerfft_win(self.samp_rate, self.FFT_size, $ref_scale,
$frame_rate)</make>

  <param>
    <name>Sample Rate</name>
    <key>sample_rate</key>
    <value>samp_rate</value>
    <type>float</type>
  </param>

  <param>
    <name>Vector Length</name>
    <key>vector_length</key>
    <value>FFT_size</value>
    <type>float</type>
```

```
 </param>

 <param>
   <name>Reference Scale</name>
   <key>ref_scale</key>
   <value>2</value>
   <type>float</type>
 </param>

 <param>
   <name>Frame Rate</name>
   <key>frame_rate</key>
   <value>30</value>
   <type>float</type>
 </param>

 <sink>
   <name>in</name>
   <type>complex</type>
 </sink>

 <source>
   <name>out</name>
   <type>float</type>
   <vlen>$vector_length</vlen>
 </source>

</block>
```

Annex VII – GNURadio custom block: logpowerfft_win.py

```
# -*- coding: utf-8 -*-
#
# Copyright 2019 ERICK MEDINA MORENO
#
# This is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# This software is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this software; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#
from __future__ import division

from gnuradio import gr
from gnuradio import blocks
from gnuradio import FFT as FFT_lib
import sys, math

try:
    from gnuradio import filter
except ImportError:
    sys.stderr.write('logpwrfft_win required gr-filter.\n')
    sys.exit(1)

class logpowerfft_win(gr.hier_block2):
    """
    docstring for block logpowerfft_win
    """
    def __init__(self, sample_rate, FFT_size, ref_scale, frame_rate):
        gr.hier_block2.__init__(self,
            "logpowerfft_win",
            gr.io_signature(1, 1, gr.sizeof_gr_complex),
            gr.io_signature(1, 1, gr.sizeof_float*FFT_size))


        self._sd = blocks.stream_to_vector_decimator(item_size=gr.sizeof_gr_complex,
                                                     sample_rate=sample_rate,
                                                     vec_rate=frame_rate,
                                                     vec_len=FFT_size)

        FFT_window = FFT_lib.window_hamming(FFT_size)
```

```
        FFT = FFT_lib.FFT_vcc(FFT_size, True, FFT_window, True)
        window_power = sum([x*x for x in FFT_window])

        c2magsq = blocks.complex_to_mag_squared(FFT_size)
        self._avg = filter.single_pole_iir_filter_ff(1.0, FFT_size)
        self._log = blocks.nlog10_ff(10, FFT_size,
                                     -20*math.log10(FFT_size)            # Adjust for number of bins
                                     -10*math.log10(float(window_power) / FFT_size) # Adjust for
windowing loss
                                     -20*math.log10(float(ref_scale) / 2))      # Adjust for reference
scale
        self.connect(self, self._sd, FFT, c2magsq, self._avg, self._log, self)

    def set_decimation(self, decim):
        """
        Set the decimation on stream decimator.
        Args:
            decim: the new decimation
        """
        self._sd.set_decimation(decim)

    def set_vec_rate(self, vec_rate):
        """
        Set the vector rate on stream decimator.
        Args:
            vec_rate: the new vector rate
        """
        self._sd.set_vec_rate(vec_rate)

    def set_sample_rate(self, sample_rate):
        """
        Set the new sampling rate
        Args:
            sample_rate: the new rate
        """
        self._sd.set_sample_rate(sample_rate)

    def set_average(self, average):
        """
        Set the averaging filter on/off.
        Args:
            average: true to set averaging on
        """
        self._average = average
        if self._average:
            self._avg.set_taps(self._avg_alpha)
        else:
            self._avg.set_taps(1.0)

    def set_avg_alpha(self, avg_alpha):
        """
        Set the average alpha and set the taps if average was on.
        Args:
            avg_alpha: the new iir filter tap
```

```
        """
        self._avg_alpha = avg_alpha
        self.set_average(self._average)

    def sample_rate(self):
        """
        Return the current sample rate.
        """
        return self._sd.sample_rate()

    def decimation(self):
        """
        Return the current decimation.
        """
        return self._sd.decimation()

    def frame_rate(self):
        """
        Return the current frame rate.
        """
        return self._sd.frame_rate()

    def average(self):
        """
        Return whether or not averaging is being performed.
        """
        return self._average

    def avg_alpha(self):
        """
        Return averaging filter constant.
        """
        return self._avg_alpha
```

Annex VIII – GNURadio custom block: tfm_power_analyzer_ff.xml

```
<?xml version="1.0"?>
<block>
 <name>power_analyzer_ff</name>
 <key>tfm_power_analyzer_ff</key>
 <category>[tfm]</category>
 <import>import tfm</import>
        <make>tfm.power_analyzer_ff(self.samp_rate, self.freq, self.FFT_size,
self.directory)</make>

 <param>
  <name>Sample Rate</name>
  <key>sample_rate</key>
  <value>samp_rate</value>
  <type>float</type>
 </param>

 <param>
  <name>Center Frequency</name>
  <key>center_frequency</key>
  <value>freq</value>
  <type>float</type>
 </param>

 <param>
  <name>Vector Length</name>
  <key>vector_length</key>
  <value>FFT_size</value>
  <type>float</type>
 </param>

 <param>
  <name>File Directory</name>
  <key>directory</key>
  <value>directory</value>
  <type>string</type>
 </param>

 <sink>
  <name>in</name>
  <type>float</type>
  <vlen>$vector_length</vlen>
 </sink>

</block>
```

Annex IX – GNURadio custom block: power_analyzer_ff.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#

##################################################
# GNU Radio Python Flow Graph
# Title: Drone Detection
# Author: Erick Medina Moreno
# Description: Script that scans from 1MHz-6GHz and creates files with averages for all the frequencies
# Generated: Wed Feb 12 12:57:33 2020
##################################################

import numpy
import os
from gnuradio import gr

class power_analyzer_ff(gr.sync_block):
    """
    docstring for block power_analyzer_ff
    """
    def __init__(self, sample_rate, center_frequency, vector_length, directory):
        self.vlen = vector_length
        self.samp_rate = sample_rate
        self.center_freq = center_frequency
        self.freq_delta = sample_rate/(vector_length-1)
        self.directory = directory
        print("delta: %.0f " % self.freq_delta)
        gr.sync_block.__init__(self,
            name="power_analyzer_ff",
            in_sig=[(numpy.float32,self.vlen)],
            out_sig=None)

    def set_center_freq(self, center_frequency):
        self.center_freq = center_frequency

    def set_directory(self, directory):
        self.directory = directory

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate

    def set_FFT_size(self, FFT_size):
        self.vlen = FFT_size

    def work(self, input_items, output_items):
        file_base = "power_%.0fMHz_%.0fMsps_%dFFT" % (self.center_freq // 1e6, self.samp_rate // 1e6, self.vlen)
        filename = "{dir}/{file}.txt".format(dir=self.directory, file=file_base)
```

```
    filename_temp = "{dir}/{file}_tmp.txt".format(dir=self.directory,file=file_base)
    in0 = input_items[0]
    start_freq = self.center_freq - self.samp_rate / 2
    for i, value in enumerate(in0):
        iterator = numpy.nditer(value, flags=['f_index'])
        file_exists = False
        try:
            file = open(filename, 'r')
            file_exists = True
        except IOError:
            file = open(filename, 'w+')
        file_index = 0
        if file_exists:
            try:
                file_index = int(file.readline()) #read number of values per row
            except Exception:
                file_index = 0
        temp_file = open(filename_temp, 'w+')
        temp_file.write("%d\n" % (file_index+1))
        while not iterator.finished:
            current_freq = (iterator.index * self.freq_delta) + start_freq
            cached_power = 1000
            if file_exists:
                try:
                    cached_power = float(file.readline().split("@")[0]) #read power
                except Exception:
                    cached_power = 1000
            power = iterator[0]
            if cached_power != 1000:
                power = ((cached_power * file_index) + power) / (file_index+1)
            temp_file.write("%.2f@%.6f" % (power, current_freq/1e6))
            if (iterator.index != self.vlen-1):
                temp_file.write("\n")
            iterator.iternext()
        file.close()
        temp_file.close()
        os.remove(filename)
        os.rename(filename_temp, filename)
    return len(input_items[0])
```

Annex X – GNURadio custom block: tfm_power_comparator_ff.xml

```
<?xml version="1.0"?>
<block>
  <name>power_comparator_ff</name>
  <key>tfm_power_comparator_ff</key>
  <category>[tfm]</category>
  <import>import tfm</import>
  <make>tfm.power_comparator_ff(self.samp_rate, self.freq, self.FFT_size, self.directory,
$mode, $diff_fixed_dBm, $diff_percentage)</make>

  <param>
    <name>Sample Rate</name>
    <key>sample_rate</key>
    <value>samp_rate</value>
    <type>float</type>
  </param>

  <param>
    <name>Center Frequency</name>
    <key>center_frequency</key>
    <value>freq</value>
    <type>float</type>
  </param>

  <param>
    <name>Vector Length</name>
    <key>vector_length</key>
    <value>FFT_size</value>
    <type>float</type>
  </param>

  <param>
    <name>File Directory</name>
    <key>directory</key>
    <value>directory</value>
    <type>string</type>
  </param>

<param>
    <name>Mode</name>
    <key>mode</key>
    <value>1</value>
    <type>float</type>
    <option>
      <name>Percentage</name>
      <key>1</key>
    </option>
    <option>
```

```
    <name>Fixed Value</name>
    <key>2</key>
   </option>
</param>

<param>
   <name>Percentage</name>
   <key>diff_percentage</key>
   <value>0</value>
   <type>float</type>
   <hide>#if $mode() == 1 then 'none' else 'all'#</hide>
 </param>

 <param>
   <name>Fixed Value in dBm</name>
   <key>diff_fixed_dBm</key>
   <value>0</value>
   <type>float</type>
   <hide>#if $mode() == 2 then 'none' else 'all'#</hide>
 </param>

 <sink>
   <name>in</name>
   <type>float</type>
   <vlen>$vector_length</vlen>
 </sink>

</block>
```

Annex XI – GNURadio custom block: power_comparator_ff.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
# Copyright 2019 ERICK ADOLFO MEDINA MORENO.
#
# This is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# This software is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this software; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#

import numpy
import os
from datetime import datetime
from gnuradio import gr

class power_comparator_ff(gr.sync_block):

    def __init__(self, sample_rate, center_frequency, vector_length, directory, mode,
diff_fixed_dBm, diff_percentage):
        self.vlen = vector_length
        self.samp_rate = sample_rate
        self.center_freq = center_frequency
        self.freq_delta = sample_rate/(vector_length-1)
        self.directory = directory
        self.mode = mode
        self.diff_dBm = diff_fixed_dBm
        self.diff_percentage = diff_percentage
        gr.sync_block.__init__(self,
            name="power_comparator_ff",
            in_sig=[(numpy.float32,self.vlen)],
            out_sig=None)

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate

    def set_center_freq(self, center_frequency):
        self.center_freq = center_frequency
```

```
    def set_mode(self, mode):
        self.mode = mode

    def set_diff_percentage(self, diff_percentage):
        print("Set Diff %")
        print(diff_percentage)
        self.diff_percentage = diff_percentage

    def set_diff_dBm(self, diff_fixed_dBm):
        print("Set Diff Db")
        print(diff_fixed_dBm)
        self.diff_fixed_dBm = diff_fixed_dBm
        self.diff_dBm = diff_fixed_dBm

    def work(self, input_items, output_items):
        in0 = input_items[0]
        file_base_power = "power_%.0fMHz_%.0fMsps_%dFFT" % (self.center_freq //
1e6, self.samp_rate // 1e6, self.vlen)
        file_base_compare = "compare_%.0fMHz_%.0fMsps_%dFFT" %
(self.center_freq // 1e6, self.samp_rate // 1e6, self.vlen)
        filename_power = "{dir}/{file}.txt".format(dir=self.directory,
file=file_base_power)
        filename_result =
"{dir}/{file}.txt".format(dir=self.directory,file=file_base_compare)
        filename_result_temp =
"{dir}/{file}_tmp.txt".format(dir=self.directory,file=file_base_compare)
        filename_log = "{dir}/log.txt".format(dir=self.directory)
        in0 = input_items[0]
        start_freq = self.center_freq - self.samp_rate / 2
        log_file = open(filename_log, 'a+')
        log_file.write(datetime.now().strftime('%Y%m%d %H:%M:%S:%f')+" ")
        log_file.write("files: " + filename_power+ ";" + filename_result +"\n")
        for i, value in enumerate(in0):
            file_power_exists = False
            try:
                file_power = open(filename_power, 'r')
                file_power_exists = True
            except IOError:
                log_file.write(datetime.now().strftime('%Y%m%d %H:%M:%S:%f')+" ")
                log_file.write("No database file for {file}\n".format(file=file_base_power))
                return 0
            iterator = numpy.nditer(value, flags=['f_index'])
            file_power_index = 0 #we must read this value because is the first line of the
file. Not needed for processing
            if file_power_exists:
                try:
                    file_power_index = float(file_power.readline()) #read number of values
per row of powers
                except Exception:
                        log_file.write("file power exception\n")
            file_result_exists = False
            try:
                file_result = open(filename_result, 'r')
                file_result_exists = True
```

```
        except IOError:
            file_result = open(filename_result, 'w+')
        file_result_index = 0
        if file_result_exists:
            try:
                file_result_index = float(file_result.readline()) #read number of values
per row of results
            except Exception:
                file_result_index = 0
        temp_file = open(filename_result_temp, 'w+')
        temp_file.write("%.0f\n" % (file_result_index+1))
        while not iterator.finished:
            current_freq = (iterator.index * self.freq_delta) + start_freq
            cached_power = 1000
            if file_power_exists:
                try:
                    line = file_power.readline()
                    cached_power = float(line.split("@")[0]) #read database power
                except Exception:
                    log_file.write(datetime.now().strftime('%Y%m%d %H:%M:%S:%f')+" ")
                    log_file.write("cached_power exception\n")
            power = iterator[0]
            data = "default"
            exceeded_number = 0
            exceeded_average = 0
            exceeded_diff_min = 10000
            exceeded_diff_average = 0
            exceeded_diff_max = 0
            if file_result_exists:
                try:
                    line = file_result.readline()
                    data = line.split("@")[0]
                    values = data.split(";")
                    exceeded_number = float(values[0])
                    exceeded_average = float(values[1])
                    exceeded_diff_min = float(values[2])
                    exceeded_diff_average = float(values[3])
                    exceeded_diff_max = float(values[4])
                except Exception:
                    nodata = True
            exceeded_diff = 0
            if self.mode == 1: #percentage
                threshold = cached_power*(1+self.diff_percentage/100)
            else: #fixed dBm
                threshold = cached_power+self.diff_dBm
            if power > threshold:
                exceeded_diff = power - cached_power
                exceeded_diff_min =
numpy.minimum(exceeded_diff_min,exceeded_diff)
                exceeded_diff_average = ((exceeded_diff_average * exceeded_number)
+ exceeded_diff) / (exceeded_number+1)
                exceeded_number = exceeded_number+1
                exceeded_diff_max =
numpy.maximum(exceeded_diff_max,exceeded_diff)
```

```
            exceeded_average = exceeded_number/(file_result_index+1)
            temp_file.write("%.0f;%.2f;%.2f;%.2f;%.2f@%.6f" % (exceeded_number,
exceeded_average, exceeded_diff_min,
                exceeded_diff_average, exceeded_diff_max, current_freq/1e6))
            if (iterator.index != self.vlen-1):
                temp_file.write("\n")
            iterator.iternext()
        file_power.close()
        file_result.close()
        temp_file.close()
        os.remove(filename_result)
        os.rename(filename_result_temp, filename_result)
    log_file.close()
    return len(input_items[0])
```

Annex XII – Base Scanner script: 01-scan-base.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: Base Scan
# Author: Erick Medina Moreno
# Description: Obtains base (averaged) power values from 1MHz to 6GHz
# Generated: Sun Jan 26 17:38:14 2020
##################################################

from distutils.version import StrictVersion

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"

from PyQt5 import Qt
from PyQt5 import Qt, QtCore
from PyQt5.QtCore import QObject, pyqtSlot
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from gnuradio.qtgui import Range, RangeWidget
from optparse import OptionParser
import osmosdr
import sys
import tfm
import time
from gnuradio import qtgui


class top_block(gr.top_block, Qt.QWidget):

    def __init__(self):
        gr.top_block.__init__(self, "Base Scan")
        Qt.QWidget.__init__(self)
        self.setWindowTitle("Base Scan")
        qtgui.util.check_set_qss()
        try:
            self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
        except:
            pass
```

```
        self.top_scroll_layout = Qt.QVBoxLayout()
        self.setLayout(self.top_scroll_layout)
        self.top_scroll = Qt.QScrollArea()
        self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
        self.top_scroll_layout.addWidget(self.top_scroll)
        self.top_scroll.setWidgetResizable(True)
        self.top_widget = Qt.QWidget()
        self.top_scroll.setWidget(self.top_widget)
        self.top_layout = Qt.QVBoxLayout(self.top_widget)
        self.top_grid_layout = Qt.QGridLayout()
        self.top_layout.addLayout(self.top_grid_layout)

        self.settings = Qt.QSettings("GNU Radio", "top_block")

        if StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
            self.restoreGeometry(self.settings.value("geometry").toByteArray())
        else:
            self.restoreGeometry(self.settings.value("geometry",
type=QtCore.QByteArray))

        ##################################################
        # Variables
        ##################################################

        arguments = sys.argv[1:]
        hasArguments = len(arguments) == 3

        self.gui_samp_rate = gui_samp_rate = 20 if not hasArguments else
int(sys.argv[2])
        self.samp_rate = samp_rate = gui_samp_rate * 1e6
        self.gui_directory = gui_directory = "/home/eamedina/Documentos/freq_docs/
new" if not hasArguments else sys.argv[1]
        self.freq_min = freq_min = 0
        self.gui_update_btn = gui_update_btn = 0
        self.gui_time_switch = gui_time_switch = 50
        self.gui_FFT_size = gui_FFT_size = 1024 if not hasArguments else
int(sys.argv[3])
        self.time_switch = gui_time_switch
        self.freq_max = freq_max = 6000e6
        self.freq = freq = freq_min+(samp_rate/2)
        self.FFT_size = FFT_size = gui_FFT_size
        self.directory = directory = gui_directory

        ##################################################
        # Blocks
        ##################################################
        self.tfm_power_analyzer_ff_0_0 = tfm.power_analyzer_ff(self.samp_rate,
self.freq, self.FFT_size, self.directory)
        self.tfm_logpowerfft_win_0 = tfm.logpowerfft_win(self.samp_rate,
self.FFT_size, 2, 30)
        self.osmosdr_source_0 = osmosdr.source( args="numchan=" + str(1) + " " +
'' )
        self.osmosdr_source_0.set_sample_rate(samp_rate)
        self.osmosdr_source_0.set_center_freq(freq, 0)
```

```
        self.osmosdr_source_0.set_freq_corr(0, 0)
        self.osmosdr_source_0.set_dc_offset_mode(1, 0)
        self.osmosdr_source_0.set_iq_balance_mode(0, 0)
        self.osmosdr_source_0.set_gain_mode(False, 0)
        self.osmosdr_source_0.set_gain(0, 0)
        self.osmosdr_source_0.set_if_gain(0, 0)
        self.osmosdr_source_0.set_bb_gain(0, 0)
        self.osmosdr_source_0.set_antenna('', 0)
        self.osmosdr_source_0.set_bandwidth(0, 0)

        _gui_update_btn_push_button = Qt.QPushButton('Update Params')
        self._gui_update_btn_choices = {'Pressed': 1, 'Released': 0}
        _gui_update_btn_push_button.pressed.connect(lambda:
self.set_gui_update_btn(self._gui_update_btn_choices['Pressed']))
        _gui_update_btn_push_button.released.connect(lambda:
self.set_gui_update_btn(self._gui_update_btn_choices['Released']))

        self._gui_time_switch_range = Range(50, 1500, 50, 50, 200)
        self._gui_time_switch_win = RangeWidget(self._gui_time_switch_range,
self.set_gui_time_switch, 'Frequency Switch Time (ms)', "counter_slider", float)

        self._gui_samp_rate_options = (10, 20, )
        self._gui_samp_rate_labels = ('10 Msps', '20 Msps', )
        self._gui_samp_rate_tool_bar = Qt.QToolBar(self)
        self._gui_samp_rate_tool_bar.addWidget(Qt.QLabel('Sample Rate'+": "))
        self._gui_samp_rate_combo_box = Qt.QComboBox()
        self._gui_samp_rate_combo_box.setEnabled(False)
        self._gui_samp_rate_tool_bar.addWidget(self._gui_samp_rate_combo_box)
        for label in self._gui_samp_rate_labels:
self._gui_samp_rate_combo_box.addItem(label)
        self._gui_samp_rate_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._gui_samp_rate_combo_box,
"setCurrentIndex", Qt.Q_ARG("int", self._gui_samp_rate_options.index(i)))
        self._gui_samp_rate_callback(self.gui_samp_rate)
        self._gui_samp_rate_combo_box.currentIndexChanged.connect(
          lambda i: self.set_gui_samp_rate(self._gui_samp_rate_options[i]))


        self._gui_FFT_size_options = (1024, 2048, )
        self._gui_FFT_size_labels = (str(self._gui_FFT_size_options[0]),
str(self._gui_FFT_size_options[1]), )
        self._gui_FFT_size_tool_bar = Qt.QToolBar(self)
        self._gui_FFT_size_tool_bar.addWidget(Qt.QLabel('FFT size'+": "))
        self._gui_FFT_size_combo_box = Qt.QComboBox()
        self._gui_FFT_size_tool_bar.addWidget(self._gui_FFT_size_combo_box)
        self._gui_FFT_size_combo_box.setEnabled(False)
        for label in self._gui_FFT_size_labels:
self._gui_FFT_size_combo_box.addItem(label)
        self._gui_FFT_size_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._gui_FFT_size_combo_box, "setCurrentIndex",
Qt.Q_ARG("int", self._gui_FFT_size_options.index(i)))
        self._gui_FFT_size_callback(self.gui_FFT_size)
        self._gui_FFT_size_combo_box.currentIndexChanged.connect(
          lambda i: self.set_gui_FFT_size(self._gui_FFT_size_options[i]))
```

```
        self._gui_directory_tool_bar = Qt.QToolBar(self)
        self._gui_directory_tool_bar.addWidget(Qt.QLabel('Directory'+": "))
        self._gui_directory_line_edit = Qt.QLineEdit(str(self.gui_directory))
        self._gui_directory_line_edit.setReadOnly(True)
        self._gui_directory_tool_bar.addWidget(self._gui_directory_line_edit)
        self._gui_directory_line_edit.returnPressed.connect(
            lambda:
self.set_gui_directory(str(str(self._gui_directory_line_edit.text().toAscii()))))

        self.top_layout.addWidget(self._gui_directory_tool_bar)
        self.top_layout.addWidget(self._gui_samp_rate_tool_bar)
        self.top_layout.addWidget(self._gui_FFT_size_tool_bar)
        self.top_layout.addWidget(self._gui_time_switch_win)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.osmosdr_source_0, 0), (self.tfm_logpowerfft_win_0, 0))
        self.connect((self.tfm_logpowerfft_win_0, 0), (self.tfm_power_analyzer_ff_0_0,
0))

        self.startTimer()

    def startTimer(self):
        self.timer = QtCore.QTimer()
        self.timer.setInterval(self.time_switch)
        self.timer.timeout.connect(self.recurring_timer)
        self.timer.start()

    def recurring_timer(self):
        if (self.get_freq()+self.samp_rate >= self.get_freq_max()):
            self.set_freq(self.get_freq_min()+self.samp_rate/2)
        else:
            self.set_freq(self.get_freq()+self.samp_rate)

    def closeEvent(self, event):
        self.settings = Qt.QSettings("GNU Radio", "top_block")
        self.settings.setValue("geometry", self.saveGeometry())
        event.accept()

    def get_gui_samp_rate(self):
        return self.gui_samp_rate

    def set_gui_samp_rate(self, gui_samp_rate):
        self.gui_samp_rate = gui_samp_rate
        self.set_samp_rate(self.gui_samp_rate * 1e6)
        self._gui_samp_rate_callback(self.gui_samp_rate)

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
```

```
        self.set_freq(self.freq_min+(self.samp_rate/2))
        self.osmosdr_source_0.set_sample_rate(self.samp_rate)
        self.tfm_power_analyzer_ff_0_0.set_samp_rate(self.samp_rate)
        self.tfm_logpowerfft_win_0.set_sample_rate(self.samp_rate)

    def get_gui_directory(self):
        return self.gui_directory

    def set_gui_directory(self, gui_directory):
        self.gui_directory = gui_directory
        self.set_directory(self.gui_directory)
        Qt.QMetaObject.invokeMethod(self._gui_directory_line_edit, "setText",
Qt.Q_ARG("QString", str(self.gui_directory)))
        self.tfm_power_analyzer_ff_0_0.set_directory(self.gui_directory)

    def get_freq_min(self):
        return self.freq_min

    def set_freq_min(self, freq_min):
        self.freq_min = freq_min
        self.set_freq(self.freq_min+(self.samp_rate/2))

    def get_gui_time_switch(self):
        return self.gui_time_switch

    def set_gui_time_switch(self, gui_time_switch):
        self.gui_time_switch = gui_time_switch
        self.set_time_switch(gui_time_switch)

    def get_time_switch(self):
        return self.time_switch

    def set_time_switch(self, time_switch):
        self.time_switch = time_switch
        self.timer.stop()
        self.startTimer()

    def get_freq_max(self):
        return self.freq_max

    def set_freq_max(self, freq_max):
        self.freq_max = freq_max

    def get_freq(self):
        return self.freq

    def set_freq(self, freq):
        self.freq = freq
        self.osmosdr_source_0.set_center_freq(self.freq, 0)
        self.tfm_power_analyzer_ff_0_0.set_center_freq(self.freq)

    def get_FFT_size(self):
        return self.FFT_size
```

```
    def set_FFT_size(self, FFT_size):
        self.FFT_size = FFT_size

    def get_directory(self):
        return self.directory

    def set_directory(self, directory):
        self.directory = directory

    def get_gui_FFT_size(self):
        return self.gui_FFT_size

    def set_gui_FFT_size(self, gui_FFT_size):
        self.gui_FFT_size = gui_FFT_size
        self.FFT_size = gui_FFT_size
        self._gui_FFT_size_callback(self.gui_FFT_size)

def main(top_block_cls=top_block, options=None):

    if StrictVersion("4.5.0") <= StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
        style = gr.prefs().get_string('qtgui', 'style', 'raster')
        Qt.QApplication.setGraphicsSystem(style)
    qapp = Qt.QApplication(sys.argv)

    tb = top_block_cls()
    tb.start()
    tb.show()

    def quitting():
        tb.stop()
        tb.wait()
    qapp.aboutToQuit.connect(quitting)
    qapp.exec_()


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

Annex XIII – Spectrum Scan Script: 02-scan-spectrum.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: Scan Spectrum
# Author: Erick Medina Moreno
# Description: Compares real time power values with base values
# Generated: Wed Jan 22 23:38:50 2020
##################################################

from distutils.version import StrictVersion

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"

from PyQt5 import Qt
from PyQt5 import Qt, QtCore
from PyQt5.QtCore import QObject, pyqtSlot
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from gnuradio.qtgui import Range, RangeWidget
from optparse import OptionParser
import osmosdr
import sys
import tfm
import time
from gnuradio import qtgui


class top_block(gr.top_block, Qt.QWidget):

    def __init__(self):
        gr.top_block.__init__(self, "Scan Spectrum")
        Qt.QWidget.__init__(self)
        self.setWindowTitle("Scan Spectrum")
        qtgui.util.check_set_qss()
        try:
            self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
        except:
            pass
```

```
        self.top_scroll_layout = Qt.QVBoxLayout()
        self.setLayout(self.top_scroll_layout)
        self.top_scroll = Qt.QScrollArea()
        self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
        self.top_scroll_layout.addWidget(self.top_scroll)
        self.top_scroll.setWidgetResizable(True)
        self.top_widget = Qt.QWidget()
        self.top_scroll.setWidget(self.top_widget)
        self.top_layout = Qt.QVBoxLayout(self.top_widget)
        self.top_grid_layout = Qt.QGridLayout()
        self.top_layout.addLayout(self.top_grid_layout)

        self.settings = Qt.QSettings("GNU Radio", "top_block")

        if StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
            self.restoreGeometry(self.settings.value("geometry").toByteArray())
        else:
            self.restoreGeometry(self.settings.value("geometry",
type=QtCore.QByteArray))

        ##################################################
        # Variables
        ##################################################

        arguments = sys.argv[1:]
        hasArguments = len(arguments) == 3

        self.gui_samp_rate = gui_samp_rate = 20 if not hasArguments else
int(sys.argv[2])
        self.samp_rate = samp_rate = gui_samp_rate*1e6
        self.gui_FFT_size = gui_FFT_size = 1024 if not hasArguments else
int(sys.argv[3])
        self.FFT_size = FFT_size = gui_FFT_size
        self.gui_time_switch = gui_time_switch = 100
        self.time_switch = time_switch = gui_time_switch
        self.gui_freq_min = gui_freq_min = 0
        self.freq_min = freq_min = gui_freq_min
        self.gui_freq_max = gui_freq_max = 6000e6
        self.freq_max = freq_max = gui_freq_max
        self.gui_directory = gui_directory = "/home/eamedina/Documentos/freq_docs/
new" if not hasArguments else sys.argv[1]
        self.gui_mode_value = gui_mode_value = 1
        self.gui_mode = gui_mode = 1
        self.freq = freq = freq_min+(samp_rate/2)
        self.directory = directory = gui_directory

        ##################################################
        # Blocks
        ##################################################
        self._gui_mode_value_range = Range(1, 100, 1, 1, 100)
        self._gui_mode_value_win = RangeWidget(self._gui_mode_value_range,
self.set_gui_mode_value, 'Mode Value (% or dBm)', "counter_slider", float)
```

```
        self._gui_freq_min_range = Range(0, 6000-self.samp_rate/1e6, self.samp_rate/
1e6, 0, 200)
        self._gui_freq_min_win = RangeWidget(self._gui_freq_min_range,
self.set_gui_freq_min, 'Lower Frequency (MHz)', "counter_slider", float)

        self.tfm_power_comparator_ff_0 = tfm.power_comparator_ff(self.samp_rate,
self.freq, self.FFT_size, self.directory, gui_mode, gui_mode_value,
gui_mode_value)
        self.tfm_logpowerfft_win_0 = tfm.logpowerfft_win(self.samp_rate,
self.FFT_size, 2, 30)
        self.osmosdr_source_0 = osmosdr.source( args="numchan=" + str(1) + " " +
'' )
        self.osmosdr_source_0.set_sample_rate(samp_rate)
        self.osmosdr_source_0.set_center_freq(freq, 0)
        self.osmosdr_source_0.set_freq_corr(0, 0)
        self.osmosdr_source_0.set_dc_offset_mode(2, 0)
        self.osmosdr_source_0.set_iq_balance_mode(0, 0)
        self.osmosdr_source_0.set_gain_mode(False, 0)
        self.osmosdr_source_0.set_gain(0, 0)
        self.osmosdr_source_0.set_if_gain(0, 0)
        self.osmosdr_source_0.set_bb_gain(0, 0)
        self.osmosdr_source_0.set_antenna('', 0)
        self.osmosdr_source_0.set_bandwidth(0, 0)

        self._gui_time_switch_range = Range(50, 1500, 50, 250, 200)
        self._gui_time_switch_win = RangeWidget(self._gui_time_switch_range,
self.set_gui_time_switch, 'Frequency Switch Time (ms)', "counter_slider", float)

        self._gui_samp_rate_options = (10, 20, )
        self._gui_samp_rate_labels = ('10 Msps', '20 Msps', )
        self._gui_samp_rate_tool_bar = Qt.QToolBar(self)
        self._gui_samp_rate_tool_bar.addWidget(Qt.QLabel('Sample Rate'+": "))
        self._gui_samp_rate_combo_box = Qt.QComboBox()
        self._gui_samp_rate_combo_box.setEnabled(False)
        self._gui_samp_rate_tool_bar.addWidget(self._gui_samp_rate_combo_box)
        for label in self._gui_samp_rate_labels:
self._gui_samp_rate_combo_box.addItem(label)
        self._gui_samp_rate_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._gui_samp_rate_combo_box,
"setCurrentIndex", Qt.Q_ARG("int", self._gui_samp_rate_options.index(i)))
        self._gui_samp_rate_callback(self.gui_samp_rate)
        self._gui_samp_rate_combo_box.currentIndexChanged.connect(
          lambda i: self.set_gui_samp_rate(self._gui_samp_rate_options[i]))

        self._gui_mode_options = (1, 2, )
        self._gui_mode_labels = ('Percentage (%)', 'Value (dBm)', )
        self._gui_mode_tool_bar = Qt.QToolBar(self)
        self._gui_mode_tool_bar.addWidget(Qt.QLabel('Mode'+": "))
        self._gui_mode_combo_box = Qt.QComboBox()
        self._gui_mode_tool_bar.addWidget(self._gui_mode_combo_box)
        for label in self._gui_mode_labels: self._gui_mode_combo_box.addItem(label)
        self._gui_mode_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._gui_mode_combo_box, "setCurrentIndex",
Qt.Q_ARG("int", self._gui_mode_options.index(i)))
```

```
        self._gui_mode_callback(self.gui_mode)
        self._gui_mode_combo_box.currentIndexChanged.connect(
            lambda i: self.set_gui_mode(self._gui_mode_options[i]))

        self._gui_freq_max_range = Range(gui_freq_min+samp_rate/1e6, 6000,
samp_rate/1e6, 6000, 200)
        self._gui_freq_max_win = RangeWidget(self._gui_freq_max_range,
self.set_gui_freq_max, 'Higher Frequency (MHz)', "counter_slider", float)

        self._gui_FFT_size_options = (1024, 2048, )
        self._gui_FFT_size_labels = (str(self._gui_FFT_size_options[0]),
str(self._gui_FFT_size_options[1]), )
        self._gui_FFT_size_tool_bar = Qt.QToolBar(self)
        self._gui_FFT_size_tool_bar.addWidget(Qt.QLabel('FFT size'+": "))
        self._gui_FFT_size_combo_box = Qt.QComboBox()
        self._gui_FFT_size_combo_box.setEnabled(False)
        self._gui_FFT_size_tool_bar.addWidget(self._gui_FFT_size_combo_box)
        for label in self._gui_FFT_size_labels:
self._gui_FFT_size_combo_box.addItem(label)
        self._gui_FFT_size_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._gui_FFT_size_combo_box, "setCurrentIndex",
Qt.Q_ARG("int", self._gui_FFT_size_options.index(i)))
        self._gui_FFT_size_callback(self.gui_FFT_size)
        self._gui_FFT_size_combo_box.currentIndexChanged.connect(
            lambda i: self.set_gui_FFT_size(self._gui_FFT_size_options[i]))

        self._gui_directory_tool_bar = Qt.QToolBar(self)
        self._gui_directory_tool_bar.addWidget(Qt.QLabel('Directory'+": "))
        self._gui_directory_line_edit = Qt.QLineEdit(str(self.gui_directory))
        self._gui_directory_line_edit.setReadOnly(True)
        self._gui_directory_tool_bar.addWidget(self._gui_directory_line_edit)
        self._gui_directory_line_edit.returnPressed.connect(
            lambda:
self.set_gui_directory(str(str(self._gui_directory_line_edit.text().toAscii()))))

        self.top_layout.addWidget(self._gui_directory_tool_bar)
        self.top_layout.addWidget(self._gui_samp_rate_tool_bar)
        self.top_layout.addWidget(self._gui_FFT_size_tool_bar)
        self.top_layout.addWidget(self._gui_freq_min_win)
        self.top_layout.addWidget(self._gui_freq_max_win)
        self.top_layout.addWidget(self._gui_mode_tool_bar)
        self.top_layout.addWidget(self._gui_mode_value_win)
        self.top_layout.addWidget(self._gui_time_switch_win)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.osmosdr_source_0, 0), (self.tfm_logpowerfft_win_0, 0))
        self.connect((self.tfm_logpowerfft_win_0, 0), (self.tfm_power_comparator_ff_0,
0))

        self.startTimer()

    def startTimer(self):
```

```
    self.timer = QtCore.QTimer()
    self.timer.setInterval(self.time_switch)
    self.timer.timeout.connect(self.recurring_timer)
    self.timer.start()

def recurring_timer(self):
    if (self.get_freq()+self.samp_rate >= self.get_freq_max()):
        self.set_freq(self.get_freq_min()+self.samp_rate/2)
    else:
        self.set_freq(self.get_freq()+self.samp_rate)

def closeEvent(self, event):
    self.settings = Qt.QSettings("GNU Radio", "top_block")
    self.settings.setValue("geometry", self.saveGeometry())
    event.accept()

def get_gui_samp_rate(self):
    return self.gui_samp_rate

def set_gui_samp_rate(self, gui_samp_rate):
    self.gui_samp_rate = gui_samp_rate
    self.set_samp_rate(self.gui_samp_rate*1e6)
    self._gui_samp_rate_callback(self.gui_samp_rate)

def get_samp_rate(self):
    return self.samp_rate

def set_samp_rate(self, samp_rate):
    self.samp_rate = samp_rate
    self.set_freq(self.freq_min+(self.samp_rate/2))
    self.osmosdr_source_0.set_sample_rate(self.samp_rate)
    self.set_gui_freq_max(self.gui_freq_min+self.samp_rate/1e6)
    self.tfm_logpowerfft_win_0.set_sample_rate(self.samp_rate)
    self.tfm_power_comparator_ff_0.set_samp_rate(self.samp_rate)

def get_gui_time_switch(self):
    return self.gui_time_switch

def set_gui_time_switch(self, gui_time_switch):
    self.gui_time_switch = gui_time_switch
    self.set_time_switch(gui_time_switch)

def get_gui_freq_min(self):
    return self.gui_freq_min

def set_gui_freq_min(self, gui_freq_min):
    self.gui_freq_min = gui_freq_min
    self.set_freq_min(gui_freq_min)

def get_gui_FFT_size(self):
    return self.gui_FFT_size

def set_gui_FFT_size(self, gui_FFT_size):
    self.gui_FFT_size = gui_FFT_size
```

```
        self.set_FFT_size(self.gui_FFT_size)
        self._gui_FFT_size_callback(self.gui_FFT_size)

    def get_gui_directory(self):
        return self.gui_directory

    def set_gui_directory(self, gui_directory):
        self.gui_directory = gui_directory
        self.set_directory(self.gui_directory)
        Qt.QMetaObject.invokeMethod(self._gui_directory_line_edit, "setText",
Qt.Q_ARG("QString", str(self.gui_directory)))

    def get_freq_min(self):
        return self.freq_min

    def set_freq_min(self, freq_min):
        self.freq_min = freq_min
        self.set_freq(self.freq_min+(self.samp_rate/2))

    def get_time_switch(self):
        return self.time_switch

    def set_time_switch(self, time_switch):
        self.time_switch = time_switch
        self.timer.stop()
        self.startTimer()

    def get_gui_mode_value(self):
        return self.gui_mode_value

    def set_gui_mode_value(self, gui_mode_value):
        self.gui_mode_value = gui_mode_value
        if (self.gui_mode == 1):
            self.tfm_power_comparator_ff_0.set_diff_percentage(gui_mode_value)
        else:
            self.tfm_power_comparator_ff_0.set_diff_dBm(gui_mode_value)

    def get_gui_mode(self):
        return self.gui_mode

    def set_gui_mode(self, gui_mode):
        self.gui_mode = gui_mode
        self._gui_mode_callback(self.gui_mode)
        self.tfm_power_comparator_ff_0.set_mode(gui_mode)
        self.set_gui_mode_value(self.get_gui_mode_value())

    def get_gui_freq_max(self):
        return self.gui_freq_max

    def set_gui_freq_max(self, gui_freq_max):
        self.gui_freq_max = gui_freq_max
        self.set_freq_max(gui_freq_max)

    def get_freq_max(self):
```

```
        return self.freq_max

    def set_freq_max(self, freq_max):
        self.freq_max = freq_max

    def get_freq(self):
        return self.freq

    def set_freq(self, freq):
        self.freq = freq
        self.osmosdr_source_0.set_center_freq(self.freq, 0)
        self.tfm_power_comparator_ff_0.set_center_freq(self.freq)

    def get_FFT_size(self):
        return self.FFT_size

    def set_FFT_size(self, FFT_size):
        self.FFT_size = FFT_size

    def get_directory(self):
        return self.directory

    def set_directory(self, directory):
        self.directory = directory


def main(top_block_cls=top_block, options=None):

    if StrictVersion("4.5.0") <= StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
        style = gr.prefs().get_string('qtgui', 'style', 'raster')
        Qt.QApplication.setGraphicsSystem(style)
    qapp = Qt.QApplication(sys.argv)

    tb = top_block_cls()
    tb.start()
    tb.show()

    def quitting():
        tb.stop()
        tb.wait()
    qapp.aboutToQuit.connect(quitting)
    qapp.exec_()


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

Annex XIV – Band Scan script: 03-scan-band.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: Top Block
# Generated: Mon Feb  3 15:39:07 2020
##################################################

from distutils.version import StrictVersion

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"

from PyQt5 import Qt
from PyQt5 import Qt, QtCore
from PyQt5.QtCore import QObject, pyqtSlot
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio import qtgui
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from gnuradio.qtgui import Range, RangeWidget
from optparse import OptionParser
import osmosdr
import sip
import sys
import tfm
import time
from gnuradio import qtgui


class top_block(gr.top_block, Qt.QWidget):

    def __init__(self):
        gr.top_block.__init__(self, "Top Block")
        Qt.QWidget.__init__(self)
        self.setWindowTitle("Top Block")
        qtgui.util.check_set_qss()
        try:
            self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
        except:
            pass
        self.top_scroll_layout = Qt.QVBoxLayout()
```

```
        self.setLayout(self.top_scroll_layout)
        self.top_scroll = Qt.QScrollArea()
        self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
        self.top_scroll_layout.addWidget(self.top_scroll)
        self.top_scroll.setWidgetResizable(True)
        self.top_widget = Qt.QWidget()
        self.top_scroll.setWidget(self.top_widget)
        self.top_layout = Qt.QVBoxLayout(self.top_widget)
        self.top_grid_layout = Qt.QGridLayout()
        self.top_layout.addLayout(self.top_grid_layout)

        self.settings = Qt.QSettings("GNU Radio", "top_block")

        if StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
            self.restoreGeometry(self.settings.value("geometry").toByteArray())
        else:
            self.restoreGeometry(self.settings.value("geometry",
type=QtCore.QByteArray))

        ##################################################
        # Variables
        ##################################################

        arguments = sys.argv[1:]
        hasArguments = len(arguments) == 3

        self.gui_samp_rate = gui_samp_rate = 20 if not hasArguments else
int(sys.argv[2])
        self.samp_rate = samp_rate = gui_samp_rate*1e6
        self.gui_FFT_size = gui_FFT_size = 1024 if not hasArguments else
int(sys.argv[3])
        self.gui_directory = gui_directory = "/home/eamedina/Documentos/freq_docs/
FFT" if not hasArguments else sys.argv[1]
        self.freq_min = freq_min = 420e6
        self.variable_qtgui_chooser_0 = variable_qtgui_chooser_0 = 0
        self.gui_mode_value = gui_mode_value = 1
        self.gui_mode = gui_mode = 2
        self.freq_max = freq_max = 440e6
        self.freq = freq = freq_min+(samp_rate/2)
        self.FFT_size = FFT_size = gui_FFT_size
        self.directory = directory = gui_directory

        ##################################################
        # Blocks
        ##################################################
        self._gui_mode_value_range = Range(1, 100, 1, 10, 100)
        self._gui_mode_value_win = RangeWidget(self._gui_mode_value_range,
self.set_gui_mode_value, 'Mode Value (% or dBm)', "counter_slider", float)

        self._gui_frequency_range = Range(samp_rate/2e6, 6e6 - samp_rate/2e6,
samp_rate/1e6, freq/1e6, 6e6/samp_rate)
        self._gui_frequency_win = RangeWidget(self._gui_frequency_range,
self.set_gui_freq, 'Center Frequency', "counter_slider", float)
```

```
        self._variable_qtgui_chooser_0_options = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, )
        self._variable_qtgui_chooser_0_labels = ('433 MHz', '868 MHz', 'Wifi 2.4GHz
(1)',
            'Wifi 2.4GHz (2)', 'Wifi 2.4GHz (3)', 'Wifi 2.4GHz (4)', 'Wifi 2.4GHz (5)',  'Wifi
2.4GHz (6)',
            'Wifi 2.4GHz (7)*', 'Wifi 2.4GHz (8)*', 'Wifi 2.4GHz (9)*', 'Wifi 2.4GHz (10)*',
            'GPS L1', 'GPS L2', 'GPS L5', )
        self._variable_qtgui_chooser_0_tool_bar = Qt.QToolBar(self)
        self._variable_qtgui_chooser_0_tool_bar.addWidget(Qt.QLabel('Preset
Bands'+": "))
        self._variable_qtgui_chooser_0_combo_box = Qt.QComboBox()

self._variable_qtgui_chooser_0_tool_bar.addWidget(self._variable_qtgui_chooser_
0_combo_box)
        for label in self._variable_qtgui_chooser_0_labels:
self._variable_qtgui_chooser_0_combo_box.addItem(label)
        self._variable_qtgui_chooser_0_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._variable_qtgui_chooser_0_combo_box,
"setCurrentIndex", Qt.Q_ARG("int",
self._variable_qtgui_chooser_0_options.index(i)))
        self._variable_qtgui_chooser_0_callback(self.variable_qtgui_chooser_0)
        self._variable_qtgui_chooser_0_combo_box.currentIndexChanged.connect(
            lambda i:
self.set_variable_qtgui_chooser_0(self._variable_qtgui_chooser_0_options[i]))

        self.tfm_power_comparator_ff_0 = tfm.power_comparator_ff(self.samp_rate,
self.freq, self.FFT_size, self.directory, 1, gui_mode_value, gui_mode_value)
        self.tfm_logpowerfft_win_0 = tfm.logpowerfft_win(self.samp_rate,
self.FFT_size, 2, 30)
        self.qtgui_sink_x_0 = qtgui.sink_c(
            FFT_size, #fftsize
            firdes.WIN_HAMMING, #wintype
            freq, #fc
            samp_rate, #bw
            'Band Analysis', #name
            True, #plotfreq
            True, #plotwaterfall
            False, #plottime
            False, #plotconst
        )
        self.qtgui_sink_x_0.set_update_time(1.0/10)
        self._qtgui_sink_x_0_win = sip.wrapinstance(self.qtgui_sink_x_0.pyqwidget(),
Qt.QWidget)

        self.qtgui_sink_x_0.enable_rf_freq(True)

        self.osmosdr_source_0 = osmosdr.source( args="numchan=" + str(1) + " " +
'' )
        self.osmosdr_source_0.set_sample_rate(samp_rate)
        self.osmosdr_source_0.set_center_freq(freq, 0)
        self.osmosdr_source_0.set_freq_corr(0, 0)
        self.osmosdr_source_0.set_dc_offset_mode(2, 0)
        self.osmosdr_source_0.set_iq_balance_mode(0, 0)
```

```
        self.osmosdr_source_0.set_gain_mode(False, 0)
        self.osmosdr_source_0.set_gain(0, 0)
        self.osmosdr_source_0.set_if_gain(0, 0)
        self.osmosdr_source_0.set_bb_gain(0, 0)
        self.osmosdr_source_0.set_antenna('', 0)
        self.osmosdr_source_0.set_bandwidth(0, 0)

        self._gui_samp_rate_options = (10, 20, )
        self._gui_samp_rate_labels = ('10 Msps', '20 Msps', )
        self._gui_samp_rate_tool_bar = Qt.QToolBar(self)
        self._gui_samp_rate_tool_bar.addWidget(Qt.QLabel('Sample Rate'+": "))
        self._gui_samp_rate_combo_box = Qt.QComboBox()
        self._gui_samp_rate_combo_box.setEnabled(False)
        self._gui_samp_rate_tool_bar.addWidget(self._gui_samp_rate_combo_box)
        for label in self._gui_samp_rate_labels:
self._gui_samp_rate_combo_box.addItem(label)
        self._gui_samp_rate_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._gui_samp_rate_combo_box,
"setCurrentIndex", Qt.Q_ARG("int", self._gui_samp_rate_options.index(i)))
        self._gui_samp_rate_callback(self.gui_samp_rate)
        self._gui_samp_rate_combo_box.currentIndexChanged.connect(
          lambda i: self.set_gui_samp_rate(self._gui_samp_rate_options[i]))

        self._gui_mode_options = (1, 2, )
        self._gui_mode_labels = ('Percentage (%)', 'Value (dBm)', )
        self._gui_mode_tool_bar = Qt.QToolBar(self)
        self._gui_mode_tool_bar.addWidget(Qt.QLabel('Mode'+": "))
        self._gui_mode_combo_box = Qt.QComboBox()
        self._gui_mode_tool_bar.addWidget(self._gui_mode_combo_box)
        for label in self._gui_mode_labels: self._gui_mode_combo_box.addItem(label)
        self._gui_mode_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._gui_mode_combo_box, "setCurrentIndex",
Qt.Q_ARG("int", self._gui_mode_options.index(i)))
        self._gui_mode_callback(self.gui_mode)
        self._gui_mode_combo_box.currentIndexChanged.connect(
          lambda i: self.set_gui_mode(self._gui_mode_options[i]))

        self._gui_FFT_size_options = (1024, 2048, )
        self._gui_FFT_size_labels = (str(self._gui_FFT_size_options[0]),
str(self._gui_FFT_size_options[1]), )
        self._gui_FFT_size_tool_bar = Qt.QToolBar(self)
        self._gui_FFT_size_tool_bar.addWidget(Qt.QLabel('FFT size'+": "))
        self._gui_FFT_size_combo_box = Qt.QComboBox()
        self._gui_FFT_size_combo_box.setEnabled(False)
        self._gui_FFT_size_tool_bar.addWidget(self._gui_FFT_size_combo_box)
        for label in self._gui_FFT_size_labels:
self._gui_FFT_size_combo_box.addItem(label)
        self._gui_FFT_size_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._gui_FFT_size_combo_box, "setCurrentIndex",
Qt.Q_ARG("int", self._gui_FFT_size_options.index(i)))
        self._gui_FFT_size_callback(self.gui_FFT_size)
        self._gui_FFT_size_combo_box.currentIndexChanged.connect(
          lambda i: self.set_gui_FFT_size(self._gui_FFT_size_options[i]))
```

```
        self._gui_directory_tool_bar = Qt.QToolBar(self)
        self._gui_directory_tool_bar.addWidget(Qt.QLabel('Directory'+": "))
        self._gui_directory_line_edit = Qt.QLineEdit(str(self.gui_directory))
        self._gui_directory_line_edit.setReadOnly(True)
        self._gui_directory_tool_bar.addWidget(self._gui_directory_line_edit)
        self._gui_directory_line_edit.returnPressed.connect(
            lambda:
self.set_gui_directory(str(str(self._gui_directory_line_edit.text().toAscii()))))

        self.top_layout.addWidget(self._gui_directory_tool_bar)
        self.top_layout.addWidget(self._gui_samp_rate_tool_bar)
        self.top_layout.addWidget(self._gui_FFT_size_tool_bar)
        self.top_layout.addWidget(self._gui_frequency_win)
        self.top_layout.addWidget(self._variable_qtgui_chooser_0_tool_bar)
        self.top_layout.addWidget(self._gui_mode_tool_bar)
        self.top_layout.addWidget(self._gui_mode_value_win)
        self.top_layout.addWidget(self._qtgui_sink_x_0_win)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.osmosdr_source_0, 0), (self.qtgui_sink_x_0, 0))
        self.connect((self.osmosdr_source_0, 0), (self.tfm_logpowerfft_win_0, 0))
        self.connect((self.tfm_logpowerfft_win_0, 0), (self.tfm_power_comparator_ff_0,
0))

    def closeEvent(self, event):
        self.settings = Qt.QSettings("GNU Radio", "top_block")
        self.settings.setValue("geometry", self.saveGeometry())
        event.accept()

    def get_gui_samp_rate(self):
        return self.gui_samp_rate

    def set_gui_samp_rate(self, gui_samp_rate):
        self.gui_samp_rate = gui_samp_rate
        self.set_samp_rate(self.gui_samp_rate*1e6)
        self._gui_samp_rate_callback(self.gui_samp_rate)

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.set_freq(self.freq_min+(self.samp_rate/2))
        self.qtgui_sink_x_0.set_frequency_range(self.freq, self.samp_rate)
        self.osmosdr_source_0.set_sample_rate(self.samp_rate)

    def get_gui_FFT_size(self):
        return self.gui_FFT_size

    def set_gui_FFT_size(self, gui_FFT_size):
        self.gui_FFT_size = gui_FFT_size
        self.set_FFT_size(self.gui_FFT_size)
```

```
        self._gui_FFT_size_callback(self.gui_FFT_size)

    def get_gui_directory(self):
        return self.gui_directory

    def set_gui_directory(self, gui_directory):
        self.gui_directory = gui_directory
        self.set_directory(self.gui_directory)
        Qt.QMetaObject.invokeMethod(self._gui_directory_line_edit, "setText",
Qt.Q_ARG("QString", str(self.gui_directory)))

    def get_freq_min(self):
        return self.freq_min

    def set_freq_min(self, freq_min):
        self.freq_min = freq_min
        self.set_freq(self.freq_min+(self.samp_rate/2))

    def get_variable_qtgui_chooser_0(self):
        return self.variable_qtgui_chooser_0

    def set_variable_qtgui_chooser_0(self, variable_qtgui_chooser_0):
        self.variable_qtgui_chooser_0 = variable_qtgui_chooser_0
        self._variable_qtgui_chooser_0_callback(self.variable_qtgui_chooser_0)
        self.index = variable_qtgui_chooser_0
        self.rate = self.get_samp_rate()
        if (self.index == 0) :#433MHz
            self.set_freq(430e6 if self.rate == 20e6 else 435e6)
        elif (self.index == 1): #868MHz
            self.set_freq(870e6 if self.rate == 20e6 else 865e6)
        elif (self.index == 2): #Wifi 2.4 1
            self.set_freq(2410e6 if self.rate == 20e6 else 2405e6)
        elif (self.index == 3): #Wifi 2.4 2
            self.set_freq(2430e6 if self.rate == 20e6 else 2415e6)
        elif (self.index == 4): #Wifi 2.4 3
            self.set_freq(2450e6 if self.rate == 20e6 else 2425e6)
        elif (self.index == 5): #Wifi 2.4 4
            self.set_freq(2470e6 if self.rate == 20e6 else 2435e6)
        elif (self.index == 6): #Wifi 2.4 5
            self.set_freq(2490e6 if self.rate == 20e6 else 2445e6)
        elif (self.index == 7): #Wifi 2.4 6
            self.set_freq(2490e6 if self.rate == 20e6 else 2455e6)
        elif (self.index == 8): #Wifi 2.4 7
            self.set_freq(2490e6 if self.rate == 20e6 else 2465e6)
        elif (self.index == 9): #Wifi 2.4 8
            self.set_freq(2490e6 if self.rate == 20e6 else 2475e6)
        elif (self.index == 10): #Wifi 2.4 9
            self.set_freq(2490e6 if self.rate == 20e6 else 2485e6)
        elif (self.index == 11): #Wifi 2.4 10
            self.set_freq(2490e6 if self.rate == 20e6 else 2495e6)
        elif (self.index == 12): #GPS L1
            self.set_freq(1570e6 if self.rate == 20e6 else 1575e6)
        elif (self.index == 13): #GPS L2
            self.set_freq(1230e6 if self.rate == 20e6 else 1225e6)
```

```
        elif (self.index == 14): #GPS L5
            self.set_freq(1170e6 if self.rate == 20e6 else 1175e6)

    def get_gui_mode_value(self):
        return self.gui_mode_value

    def set_gui_mode_value(self, gui_mode_value):
        self.gui_mode_value = gui_mode_value
        if (self.gui_mode == 1):
            self.tfm_power_comparator_ff_0.set_diff_percentage(gui_mode_value)
        else:
            self.tfm_power_comparator_ff_0.set_diff_dBm(gui_mode_value)

    def get_gui_mode(self):
        return self.gui_mode

    def set_gui_mode(self, gui_mode):
        self.gui_mode = gui_mode
        self._gui_mode_callback(self.gui_mode)
        self.tfm_power_comparator_ff_0.set_mode(gui_mode)
        self.set_gui_mode_value(self.get_gui_mode_value())

    def get_freq_max(self):
        return self.freq_max

    def set_freq_max(self, freq_max):
        self.freq_max = freq_max

    def get_freq(self):
        return self.freq

    def set_freq(self, freq):
        self.freq = freq
        self.qtgui_sink_x_0.set_frequency_range(self.freq, self.samp_rate)
        self.osmosdr_source_0.set_center_freq(self.freq, 0)
        self.tfm_power_comparator_ff_0.set_center_freq(self.freq)

    def set_gui_freq(self, freq):
        self.set_freq(freq*1e6)

    def get_FFT_size(self):
        return self.FFT_size

    def set_FFT_size(self, FFT_size):
        self.FFT_size = FFT_size

    def get_directory(self):
        return self.directory

    def set_directory(self, directory):
        self.directory = directory


def main(top_block_cls=top_block, options=None):
```

```
    if StrictVersion("4.5.0") <= StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
        style = gr.prefs().get_string('qtgui', 'style', 'raster')
        Qt.QApplication.setGraphicsSystem(style)
    qapp = Qt.QApplication(sys.argv)

    tb = top_block_cls()
    tb.start()
    tb.show()

    def quitting():
        tb.stop()
        tb.wait()
    qapp.aboutToQuit.connect(quitting)
    qapp.exec_()


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

Annex XV – Jammer script: 04-jammer.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: Top Block
# Generated: Mon Feb 10 23:18:28 2020
##################################################

from distutils.version import StrictVersion

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"

from PyQt5 import Qt
from PyQt5 import Qt, QtCore
from PyQt5.QtCore import QObject, pyqtSlot
from gnuradio import analog
from gnuradio import blocks
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from gnuradio.qtgui import Range, RangeWidget
from optparse import OptionParser
import osmosdr
import sys
import time
from gnuradio import qtgui


class top_block(gr.top_block, Qt.QWidget):

    def __init__(self):
        gr.top_block.__init__(self, "Jammer")
        Qt.QWidget.__init__(self)
        self.setWindowTitle("Jammer")
        qtgui.util.check_set_qss()
        try:
            self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
        except:
```

```
            pass
        self.top_scroll_layout = Qt.QVBoxLayout()
        self.setLayout(self.top_scroll_layout)
        self.top_scroll = Qt.QScrollArea()
        self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
        self.top_scroll_layout.addWidget(self.top_scroll)
        self.top_scroll.setWidgetResizable(True)
        self.top_widget = Qt.QWidget()
        self.top_scroll.setWidget(self.top_widget)
        self.top_layout = Qt.QVBoxLayout(self.top_widget)
        self.top_grid_layout = Qt.QGridLayout()
        self.top_layout.addLayout(self.top_grid_layout)

        self.settings = Qt.QSettings("GNU Radio", "top_block")

        if StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
            self.restoreGeometry(self.settings.value("geometry").toByteArray())
        else:
            self.restoreGeometry(self.settings.value("geometry",
type=QtCore.QByteArray))

        ##################################################
        # Variables
        ##################################################
        self.gui_samp_rate = gui_samp_rate = 20
        self.samp_rate = samp_rate = gui_samp_rate*1e6
        self.gui_freq_min = gui_freq_min = 2400
        self.gui_freq_max = gui_freq_max = 2500
        self.freq_min = freq_min = gui_freq_min*1e6
        self.variable_qtgui_chooser_0 = variable_qtgui_chooser_0 = 0
        self.jammer_amp = jammer_amp = 50
        self.gui_time_switch = gui_time_switch = 50
        self.gui_rf_gain = gui_rf_gain = 14
        self.gui_jam_mode = gui_jam_mode = 1
        self.gui_if_gain = gui_if_gain = 47
        self.freq_max = freq_max = gui_freq_max*1e6
        self.center_freq = center_freq = freq_min+samp_rate/2

        ##################################################
        # Blocks
        ##################################################
        self._jammer_amp_range = Range(1, 100, 1, 50, 200)
        self._jammer_amp_win = RangeWidget(self._jammer_amp_range,
self.set_jammer_amp, "jammer_amp", "counter_slider", float)

        self._gui_rf_gain_options = (0, 14, )
        self._gui_rf_gain_labels = ('0 dBm', '14 dBm', )
        self._gui_rf_gain_group_box = Qt.QGroupBox('RF Gain (dBm)')
        self._gui_rf_gain_box = Qt.QHBoxLayout()
        class variable_chooser_button_group(Qt.QButtonGroup):
            def __init__(self, parent=None):
                Qt.QButtonGroup.__init__(self, parent)
            @pyqtSlot(int)
            def updateButtonChecked(self, button_id):
```

```
            self.button(button_id).setChecked(True)
        self._gui_rf_gain_button_group = variable_chooser_button_group()
        self._gui_rf_gain_group_box.setLayout(self._gui_rf_gain_box)
        for i, label in enumerate(self._gui_rf_gain_labels):
            radio_button = Qt.QRadioButton(label)
            self._gui_rf_gain_box.addWidget(radio_button)
            self._gui_rf_gain_button_group.addButton(radio_button, i)
        self._gui_rf_gain_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._gui_rf_gain_button_group,
"updateButtonChecked", Qt.Q_ARG("int", self._gui_rf_gain_options.index(i)))
        self._gui_rf_gain_callback(self.gui_rf_gain)
        self._gui_rf_gain_button_group.buttonClicked[int].connect(
            lambda i: self.set_gui_rf_gain(self._gui_rf_gain_options[i]))

        self._gui_if_gain_range = Range(0, 47, 1, 47, 47)
        self._gui_if_gain_win = RangeWidget(self._gui_if_gain_range,
self.set_gui_if_gain, 'IF Gain (dBm)', "counter_slider", float)

        self._variable_qtgui_chooser_0_options = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
13, 14, 15, )
        self._variable_qtgui_chooser_0_labels = ('None', '433 MHz', '868 MHz', 'Wifi
2.4GHz (1)',
            'Wifi 2.4GHz (2)', 'Wifi 2.4GHz (3)', 'Wifi 2.4GHz (4)', 'Wifi 2.4GHz (5)',  'Wifi
2.4GHz (6)',
            'Wifi 2.4GHz (7)*', 'Wifi 2.4GHz (8)*', 'Wifi 2.4GHz (9)*', 'Wifi 2.4GHz (10)*',
            'GPS L1', 'GPS L2', 'GPS L5', )
        self._variable_qtgui_chooser_0_tool_bar = Qt.QToolBar(self)
        self._variable_qtgui_chooser_0_tool_bar.addWidget(Qt.QLabel('Preset
Bands'+": "))
        self._variable_qtgui_chooser_0_combo_box = Qt.QComboBox()

self._variable_qtgui_chooser_0_tool_bar.addWidget(self._variable_qtgui_chooser_
0_combo_box)
        for label in self._variable_qtgui_chooser_0_labels:
self._variable_qtgui_chooser_0_combo_box.addItem(label)
        self._variable_qtgui_chooser_0_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._variable_qtgui_chooser_0_combo_box,
"setCurrentIndex", Qt.Q_ARG("int",
self._variable_qtgui_chooser_0_options.index(i)))
        self._variable_qtgui_chooser_0_callback(self.variable_qtgui_chooser_0)
        self._variable_qtgui_chooser_0_combo_box.currentIndexChanged.connect(
            lambda i:
self.set_variable_qtgui_chooser_0(self._variable_qtgui_chooser_0_options[i]))

        self.osmosdr_sink_0 = osmosdr.sink( args="numchan=" + str(1) + " " + '' )
        self.osmosdr_sink_0.set_sample_rate(samp_rate)
        self.osmosdr_sink_0.set_center_freq(center_freq, 0)
        self.osmosdr_sink_0.set_freq_corr(0, 0)
        self.osmosdr_sink_0.set_gain(gui_rf_gain, 0)
        self.osmosdr_sink_0.set_if_gain(gui_if_gain, 0)
        self.osmosdr_sink_0.set_bb_gain(0, 0)
        self.osmosdr_sink_0.set_antenna('', 0)
        self.osmosdr_sink_0.set_bandwidth(samp_rate, 0)
```

```
        self._gui_time_switch_range = Range(50, 1500, 50, 50, 200)
        self._gui_time_switch_win = RangeWidget(self._gui_time_switch_range,
self.set_gui_time_switch, 'Frequency Switch Time (ms)', "counter_slider", float)

        self._gui_samp_rate_options = (10, 20, )
        self._gui_samp_rate_labels = ('10 Msps', '20 Msps', )
        self._gui_samp_rate_tool_bar = Qt.QToolBar(self)
        self._gui_samp_rate_tool_bar.addWidget(Qt.QLabel('Sample Rate'+": "))
        self._gui_samp_rate_combo_box = Qt.QComboBox()
        self._gui_samp_rate_tool_bar.addWidget(self._gui_samp_rate_combo_box)
        for label in self._gui_samp_rate_labels:
self._gui_samp_rate_combo_box.addItem(label)
        self._gui_samp_rate_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._gui_samp_rate_combo_box,
"setCurrentIndex", Qt.Q_ARG("int", self._gui_samp_rate_options.index(i)))
        self._gui_samp_rate_callback(self.gui_samp_rate)
        self._gui_samp_rate_combo_box.currentIndexChanged.connect(
          lambda i: self.set_gui_samp_rate(self._gui_samp_rate_options[i]))

        self._gui_jam_mode_options = (1, 2, )
        self._gui_jam_mode_labels = ('Fixed Band (20 MHz)', 'Continuous ', )
        self._gui_jam_mode_tool_bar = Qt.QToolBar(self)
        self._gui_jam_mode_tool_bar.addWidget(Qt.QLabel('Jammer Mode'+": "))
        self._gui_jam_mode_combo_box = Qt.QComboBox()
        self._gui_jam_mode_tool_bar.addWidget(self._gui_jam_mode_combo_box)
        for label in self._gui_jam_mode_labels:
self._gui_jam_mode_combo_box.addItem(label)
        self._gui_jam_mode_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._gui_jam_mode_combo_box,
"setCurrentIndex", Qt.Q_ARG("int", self._gui_jam_mode_options.index(i)))
        self._gui_jam_mode_callback(self.gui_jam_mode)
        self._gui_jam_mode_combo_box.currentIndexChanged.connect(
          lambda i: self.set_gui_jam_mode(self._gui_jam_mode_options[i]))

        self._gui_freq_min_range = Range(0, 6000-samp_rate/1e6, samp_rate/1e6,
2400, 200)
        self._gui_freq_min_win = RangeWidget(self._gui_freq_min_range,
self.set_gui_freq_min, 'Lower Frequency (MHz)', "counter_slider", float)

        self._gui_freq_max_range = Range(10, 6000, samp_rate/1e6, 2500, 200)
        self._gui_freq_max_win = RangeWidget(self._gui_freq_max_range,
self.set_gui_freq_max, 'Higher Frequency (MHz)', "counter_slider", float)

        self.blocks_throttle_0 = blocks.throttle(gr.sizeof_gr_complex*1,
samp_rate,True)
        self.blocks_multiply_const_vxx_0 = blocks.multiply_const_vcc((6, ))
        self.analog_noise_source_x_0 =
analog.noise_source_c(analog.GR_UNIFORM, jammer_amp, 0)

        self.top_layout.addWidget(self._gui_samp_rate_tool_bar)
        self.top_layout.addWidget(self._jammer_amp_win)
        self.top_layout.addWidget(self._gui_rf_gain_group_box)
        self.top_layout.addWidget(self._gui_if_gain_win)
        self.top_layout.addWidget(self._gui_jam_mode_tool_bar)
```

```
        self.top_layout.addWidget(self._variable_qtgui_chooser_0_tool_bar)
        self.top_layout.addWidget(self._gui_freq_min_win)
        self.top_layout.addWidget(self._gui_freq_max_win)
        self.top_layout.addWidget(self._gui_time_switch_win)

        ##################################################
        # Connections
        ##################################################
        self.connect((self.analog_noise_source_x_0, 0),
(self.blocks_multiply_const_vxx_0, 0))
        self.connect((self.blocks_multiply_const_vxx_0, 0), (self.blocks_throttle_0, 0))
        self.connect((self.blocks_throttle_0, 0), (self.osmosdr_sink_0, 0))

    def closeEvent(self, event):
        self.settings = Qt.QSettings("GNU Radio", "top_block")
        self.settings.setValue("geometry", self.saveGeometry())
        event.accept()

    def get_gui_samp_rate(self):
        return self.gui_samp_rate

    def set_gui_samp_rate(self, gui_samp_rate):
        self.gui_samp_rate = gui_samp_rate
        self.set_samp_rate(self.gui_samp_rate*1e6)
        self._gui_samp_rate_callback(self.gui_samp_rate)

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.set_center_freq(self.freq_min+self.samp_rate/2)
        self.osmosdr_sink_0.set_sample_rate(self.samp_rate)
        self.osmosdr_sink_0.set_bandwidth(self.samp_rate, 0)
        self.set_gui_freq_max(self.gui_freq_min+self.samp_rate/1e6)
        self.blocks_throttle_0.set_sample_rate(self.samp_rate)

    def get_gui_freq_min(self):
        return self.gui_freq_min

    def set_gui_freq_min(self, gui_freq_min):
        self.gui_freq_min = gui_freq_min
        self.set_freq_min(self.gui_freq_min*1e6)

    def get_gui_freq_max(self):
        return self.gui_freq_max

    def set_gui_freq_max(self, gui_freq_max):
        self.gui_freq_max = gui_freq_max
        self.set_freq_max(self.gui_freq_max*1e6)

    def get_variable_qtgui_chooser_0(self):
        return self.variable_qtgui_chooser_0
```

```
def set_variable_qtgui_chooser_0(self, variable_qtgui_chooser_0):
    self.variable_qtgui_chooser_0 = variable_qtgui_chooser_0
    self._variable_qtgui_chooser_0_callback(self.variable_qtgui_chooser_0)
    self.index = variable_qtgui_chooser_0
    self.rate = self.get_samp_rate()
    if (self.index == 0) :#None
        self.set_gui_freq_min(0)
        self.set_gui_freq_max(self.samp_rate/1e6)
        self.set_center_freq(10e6 if self.rate == 20e6 else 5e6)
    elif (self.index == 1): #433MHz
        self.set_center_freq(430e6 if self.rate == 20e6 else 435e6)
    elif (self.index == 2): #868MHz
        self.set_center_freq(870e6 if self.rate == 20e6 else 865e6)
    elif (self.index == 3): #Wifi 2.4 1
        self.set_center_freq(2410e6 if self.rate == 20e6 else 2405e6)
    elif (self.index == 4): #Wifi 2.4 2
        self.set_center_freq(2430e6 if self.rate == 20e6 else 2415e6)
    elif (self.index == 5): #Wifi 2.4 3
        self.set_center_freq(2450e6 if self.rate == 20e6 else 2425e6)
    elif (self.index == 6): #Wifi 2.4 4
        self.set_center_freq(2470e6 if self.rate == 20e6 else 2435e6)
    elif (self.index == 7): #Wifi 2.4 5
        self.set_center_freq(2490e6 if self.rate == 20e6 else 2445e6)
    elif (self.index == 8): #Wifi 2.4 6
        self.set_center_freq(2490e6 if self.rate == 20e6 else 2455e6)
    elif (self.index == 9): #Wifi 2.4 7
        self.set_center_freq(2490e6 if self.rate == 20e6 else 2465e6)
    elif (self.index == 10): #Wifi 2.4 8
        self.set_center_freq(2490e6 if self.rate == 20e6 else 2475e6)
    elif (self.index == 11): #Wifi 2.4 9
        self.set_center_freq(2490e6 if self.rate == 20e6 else 2485e6)
    elif (self.index == 12): #Wifi 2.4 10
        self.set_center_freq(2490e6 if self.rate == 20e6 else 2495e6)
    elif (self.index == 13): #GPS L1
        self.set_center_freq(1570e6 if self.rate == 20e6 else 1575e6)
    elif (self.index == 14): #GPS L2
        self.set_center_freq(1230e6 if self.rate == 20e6 else 1225e6)
    elif (self.index == 15): #GPS L5
        self.set_center_freq(1170e6 if self.rate == 20e6 else 1175e6)

def get_jammer_amp(self):
    return self.jammer_amp

def set_jammer_amp(self, jammer_amp):
    self.jammer_amp = jammer_amp
    self.analog_noise_source_x_0.set_amplitude(self.jammer_amp)

def get_gui_time_switch(self):
    return self.gui_time_switch

def set_gui_time_switch(self, gui_time_switch):
    self.gui_time_switch = gui_time_switch
    self.startTimer()
```

```
def get_gui_rf_gain(self):
    return self.gui_rf_gain

def set_gui_rf_gain(self, gui_rf_gain):
    self.gui_rf_gain = gui_rf_gain
    self._gui_rf_gain_callback(self.gui_rf_gain)
    self.osmosdr_sink_0.set_gain(self.gui_rf_gain, 0)

def get_gui_jam_mode(self):
    return self.gui_jam_mode

def set_gui_jam_mode(self, gui_jam_mode):
    self.gui_jam_mode = gui_jam_mode
    self._gui_jam_mode_callback(self.gui_jam_mode)
    if (gui_jam_mode == 2):
        self.startTimer()
    else:
        self.stopTimer()


def get_gui_if_gain(self):
    return self.gui_if_gain

def set_gui_if_gain(self, gui_if_gain):
    self.gui_if_gain = gui_if_gain
    self.osmosdr_sink_0.set_if_gain(self.gui_if_gain, 0)

def get_freq_min(self):
    return self.freq_min

def set_freq_min(self, freq_min):
    print("Freq  Min")
    print(freq_min)
    self.freq_min = freq_min
    self.set_center_freq(self.freq_min+self.samp_rate/2)

def get_freq_max(self):
    return self.freq_max

def set_freq_max(self, freq_max):
    print("Freq Max")
    print(freq_max)
    self.freq_max = freq_max

def get_center_freq(self):
    return self.center_freq

def set_center_freq(self, center_freq):
    print("Center Freq")
    print(center_freq)
    self.center_freq = center_freq
    self.osmosdr_sink_0.set_center_freq(self.center_freq, 0)

def startTimer(self):
```

```
        self.stopTimer()
        self.timer = QtCore.QTimer()
        self.timer.setInterval(self.gui_time_switch)
        self.timer.timeout.connect(self.recurring_timer)
        self.timer.start()

    def stopTimer(self):
        try:
            self.timer.stop()
        except Exception as e:
            print("Timer not initialized")

    def recurring_timer(self):
        if (self.get_center_freq()+self.samp_rate >= self.get_freq_max()):
            self.set_center_freq(self.get_freq_min()+self.samp_rate/2)
        else:
            self.set_center_freq(self.get_center_freq()+self.samp_rate)

def main(top_block_cls=top_block, options=None):

    if StrictVersion("4.5.0") <= StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
        style = gr.prefs().get_string('qtgui', 'style', 'raster')
        Qt.QApplication.setGraphicsSystem(style)
    qapp = Qt.QApplication(sys.argv)

    tb = top_block_cls()
    tb.start()
    tb.show()

    def quitting():
        tb.stop()
        tb.wait()
    qapp.aboutToQuit.connect(quitting)
    qapp.exec_()


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

Annex XVI – Main script: 00-main.py

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
##################################################
# GNU Radio Python Flow Graph
# Title: Drone Detection
# Author: Erick Medina Moreno
# Description: Pool of scripts that run different processes to detect  drones
# Generated: Wed Feb 12 12:57:33 2020
##################################################

from distutils.version import StrictVersion

if __name__ == '__main__':
    import ctypes
    import sys
    if sys.platform.startswith('linux'):
        try:
            x11 = ctypes.cdll.LoadLibrary('libX11.so')
            x11.XInitThreads()
        except:
            print "Warning: failed to XInitThreads()"

from PyQt5 import Qt
from PyQt5 import Qt, QtCore
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio.eng_option import eng_option
from gnuradio.filter import firdes
from optparse import OptionParser
from PyQt5.QtCore import pyqtSlot
import sys
from gnuradio import qtgui
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.backends.qt_compat import QtCore, QtWidgets, is_pyqt5
if is_pyqt5():
    from matplotlib.backends.backend_qt5agg import (
        FigureCanvas, NavigationToolbar2QT as NavigationToolbar)
else:
    from matplotlib.backends.backend_qt4agg import (
        FigureCanvas, NavigationToolbar2QT as NavigationToolbar)
from matplotlib.figure import Figure
import sys
import time
import functools
from PyQt5.QtWidgets import QTableWidget,QTableWidgetItem, QFileDialog
from gnuradio.qtgui import Range, RangeWidget
import subprocess
```

```
class top_block(gr.top_block, Qt.QWidget):

    def __init__(self):
        gr.top_block.__init__(self, "Drone Detection")
        Qt.QWidget.__init__(self)
        self.setWindowTitle("Drone Detection")
        qtgui.util.check_set_qss()
        try:
            self.setWindowIcon(Qt.QIcon.fromTheme('gnuradio-grc'))
        except:
            pass
        self.top_scroll_layout = Qt.QVBoxLayout()
        self.setLayout(self.top_scroll_layout)
        self.top_scroll = Qt.QScrollArea()
        self.top_scroll.setFrameStyle(Qt.QFrame.NoFrame)
        self.top_scroll_layout.addWidget(self.top_scroll)
        self.top_scroll.setWidgetResizable(True)
        self.top_widget = Qt.QWidget()
        self.top_scroll.setWidget(self.top_widget)
        self.top_layout = Qt.QHBoxLayout(self.top_widget)

        self.data_params_layout = Qt.QHBoxLayout()
        self.directory_params_layout = Qt.QHBoxLayout()

        self.top_left_layout = Qt.QVBoxLayout()
        self.top_right_layout = Qt.QVBoxLayout()

        self.settings = Qt.QSettings("GNU Radio", "top_block")

        if StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
            self.restoreGeometry(self.settings.value("geometry").toByteArray())
        else:
            self.restoreGeometry(self.settings.value("geometry",
type=QtCore.QByteArray))

        ##################################################
        # Variables
        ##################################################
        self.directory = directory = "/home/eamedina/Documentos/freq_docs"
        self.spectrum_scan_button = spectrum_scan_button = 0
        self.jammer_button = jammer_button = 0
        self.base_scan_button = base_scan_button = 0
        self.band_scan_button = band_scan_button = 0
        self.graphic_band_choose = graphic_band_choose = 0
        self.samp_rate = 20e6
        self.FFT_size = 1024
        self.freq_max = 6000e6
        self.center_freq = self.samp_rate/2
        self.table_sort_index = 1
        self.table_sort_reverse = True
        self.bandwidth_range = bandwidth_range = 20
        self.samp_rate_chooser = samp_rate_chooser = 20
        self.center_freq_range = center_freq_range = 3000
```

```
        self.FFT_size_chooser = FFT_size_chooser = 1024
        self.update_graph_button = update_graph_button = 0
        self.update_params_button = update_params_button = 0
        self.update_diretory_button = update_directory_button = 0
        self.loop_min_freq = self.samp_rate/2
        self.loop_max_freq = self.freq_max

        ##################################################
        # Blocks
        ##################################################

        #SCRIPT BUTTONS
        _spectrum_scan_button_push_button = Qt.QPushButton('Spectrum Scan')
        self._spectrum_scan_button_choices = {'Pressed': 1, 'Released': 0}
        _spectrum_scan_button_push_button.pressed.connect(lambda:
self.set_spectrum_scan_button(self._spectrum_scan_button_choices['Pressed']))
        _spectrum_scan_button_push_button.released.connect(lambda:
self.set_spectrum_scan_button(self._spectrum_scan_button_choices['Released']))

        _jammer_button_push_button = Qt.QPushButton('Jammer')
        self._jammer_button_choices = {'Pressed': 1, 'Released': 0}
        _jammer_button_push_button.pressed.connect(lambda:
self.set_jammer_button(self._jammer_button_choices['Pressed']))
        _jammer_button_push_button.released.connect(lambda:
self.set_jammer_button(self._jammer_button_choices['Released']))

        _base_scan_button_push_button = Qt.QPushButton('Base Scan')
        self._base_scan_button_choices = {'Pressed': 1, 'Released': 0}
        _base_scan_button_push_button.pressed.connect(lambda:
self.set_base_scan_button(self._base_scan_button_choices['Pressed']))
        _base_scan_button_push_button.released.connect(lambda:
self.set_base_scan_button(self._base_scan_button_choices['Released']))

        _band_scan_button_push_button = Qt.QPushButton('Band Scan')
        self._band_scan_button_choices = {'Pressed': 1, 'Released': 0}
        _band_scan_button_push_button.pressed.connect(lambda:
self.set_band_scan_button(self._band_scan_button_choices['Pressed']))
        _band_scan_button_push_button.released.connect(lambda:
self.set_band_scan_button(self._band_scan_button_choices['Released']))

        #GRAPH PARAMS
        self._directory_entry_tool_bar = Qt.QToolBar(self)
        self._directory_entry_tool_bar.addWidget(Qt.QLabel('Directory'+": "))
        self._directory_entry_line_edit = Qt.QLineEdit(str(self.directory))
        self._directory_entry_line_edit.setReadOnly(True)
        self._directory_entry_tool_bar.addWidget(self._directory_entry_line_edit)
        self._directory_entry_line_edit.returnPressed.connect(
          lambda:
self.set_directory_entry(str(str(self._directory_entry_line_edit.text().toAscii()))))

        _update_directory_button_push_button = Qt.QPushButton('Select Directory')
        self._update_directory_button_choices = {'Pressed': 1, 'Released': 0}
```

```
        _update_directory_button_push_button.pressed.connect(lambda:
self.set_update_directory_button(self._update_directory_button_choices['Pressed'
]))
        _update_directory_button_push_button.released.connect(lambda:
self.set_update_directory_button(self._update_directory_button_choices['Release
d']))

        self._graphic_band_choose_options = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, )
        self._graphic_band_choose_labels = ('CONTINUOUS', 'ALL', '433 MHz', '868
MHz', 'Wifi 2.4GHz (1)',
            'Wifi 2.4GHz (2)', 'Wifi 2.4GHz (3)', 'Wifi 2.4GHz (4)', 'Wifi 2.4GHz (5)',  'Wifi
2.4GHz (6)*',
            'Wifi 2.4GHz (7)*', 'Wifi 2.4GHz (8)*', 'Wifi 2.4GHz (9)*', 'Wifi 2.4GHz (10)*')
        self._graphic_band_choose_tool_bar = Qt.QToolBar(self)
        self._graphic_band_choose_tool_bar.addWidget(Qt.QLabel("Choose band"+":
"))
        self._graphic_band_choose_combo_box = Qt.QComboBox()

self._graphic_band_choose_tool_bar.addWidget(self._graphic_band_choose_com
bo_box)
        for label in self._graphic_band_choose_labels:
self._graphic_band_choose_combo_box.addItem(label)
        self._graphic_band_choose_callback = lambda a i:
Qt.QMetaObject.invokeMethod(self._graphic_band_choose_combo_box,
"setCurrentIndex", Qt.Q_ARG("int", self._graphic_band_choose_options.index(i)))
        self._graphic_band_choose_callback(self.graphic_band_choose)
        self._graphic_band_choose_combo_box.currentIndexChanged.connect(
            lambda a i:
self.set_graphic_band_choose(self._graphic_band_choose_options[i]))

        self._samp_rate_chooser_options = (10, 20, )
        self._samp_rate_chooser_labels = ('10 Msps', '20 Msps', )
        self._samp_rate_chooser_tool_bar = Qt.QToolBar(self)
        self._samp_rate_chooser_tool_bar.addWidget(Qt.QLabel('Sample Rate'+": "))
        self._samp_rate_chooser_combo_box = Qt.QComboBox()

self._samp_rate_chooser_tool_bar.addWidget(self._samp_rate_chooser_combo_b
ox)
        for label in self._samp_rate_chooser_labels:
self._samp_rate_chooser_combo_box.addItem(label)
        self._samp_rate_chooser_callback = lambda a i:
Qt.QMetaObject.invokeMethod(self._samp_rate_chooser_combo_box,
"setCurrentIndex", Qt.Q_ARG("int", self._samp_rate_chooser_options.index(i)))
        self._samp_rate_chooser_callback(self.samp_rate_chooser)
        self._samp_rate_chooser_combo_box.currentIndexChanged.connect(
            lambda a i: self.set_samp_rate_chooser(self._samp_rate_chooser_options[i]))

        self._FFT_size_chooser_options = (1024, 2048, )
        self._FFT_size_chooser_labels = (str(self._FFT_size_chooser_options[0]),
str(self._FFT_size_chooser_options[1]), )
        self._FFT_size_chooser_tool_bar = Qt.QToolBar(self)
        self._FFT_size_chooser_tool_bar.addWidget(Qt.QLabel('FFT Size'+": "))
        self._FFT_size_chooser_combo_box = Qt.QComboBox()
```

```
self._FFT_size_chooser_tool_bar.addWidget(self._FFT_size_chooser_combo_box)
        for label in self._FFT_size_chooser_labels:
self._FFT_size_chooser_combo_box.addItem(label)
        self._FFT_size_chooser_callback = lambda i:
Qt.QMetaObject.invokeMethod(self._FFT_size_chooser_combo_box,
"setCurrentIndex", Qt.Q_ARG("int", self._FFT_size_chooser_options.index(i)))
        self._FFT_size_chooser_callback(self.FFT_size_chooser)
        self._FFT_size_chooser_combo_box.currentIndexChanged.connect(
          lambda i: self.set_FFT_size_chooser(self._FFT_size_chooser_options[i]))

        _update_params_button_push_button = Qt.QPushButton('Update Params.')
        self._update_params_button_choices = {'Pressed': 1, 'Released': 0}
        _update_params_button_push_button.pressed.connect(lambda:
self.set_update_params_button(self._update_params_button_choices['Pressed']))
        _update_params_button_push_button.released.connect(lambda:
self.set_update_params_button(self._update_params_button_choices['Released']))

        #FREQUENCY/RANGE CONTROLS
        self._center_freq_range_range = Range(5, 6000, 5, 3000, 200)
        self._center_freq_range_win = RangeWidget(self._center_freq_range_range,
self.set_center_freq_range, 'Freq. (MHz)', "counter_slider", float)

        self._bandwidth_range_range = Range(10, 6000, 10, 20, 200)
        self._bandwidth_range_win = RangeWidget(self._bandwidth_range_range,
self.set_bandwidth_range, 'Bandwidth (MHz)', "counter_slider", float)

        _update_graph_button_push_button = Qt.QPushButton('Update Graph')
        self._update_graph_button_choices = {'Pressed': 1, 'Released': 0}
        _update_graph_button_push_button.pressed.connect(lambda:
self.set_update_graph_button(self._update_graph_button_choices['Pressed']))
        _update_graph_button_push_button.released.connect(lambda:
self.set_update_graph_button(self._update_graph_button_choices['Released']))

        #GRAPHIC/PLOT
        self.dynamic_canvas = FigureCanvas(Figure(figsize=(5, 3)))
        self._dynamic_ax = self.dynamic_canvas.figure.subplots()

        #TABLE
        self.tableWidget = QTableWidget()
        self.tableWidget.horizontalHeader().sectionClicked.connect(self.tableClicked)

        #LEFT LAYOUT
        self.top_left_layout.addWidget(_base_scan_button_push_button)
        self.top_left_layout.addWidget(_spectrum_scan_button_push_button)
        self.top_left_layout.addWidget(_band_scan_button_push_button)
        self.top_left_layout.addWidget(_jammer_button_push_button)

        #DIRECTORY LAYOUT
        self.directory_params_layout.addWidget(self._directory_entry_tool_bar)

self.directory_params_layout.addWidget(_update_directory_button_push_button)

        #DATA PARAMS LAYOUT
```

```
        self.data_params_layout.addWidget(self._samp_rate_chooser_tool_bar)
        self.data_params_layout.addWidget(self._FFT_size_chooser_tool_bar)
        self.data_params_layout.addWidget(_update_params_button_push_button)

        #FREQUENCY PARAMS LAYOUT
        self.freq_container = Qt.QWidget()
        self.freq_params_v_layout = Qt.QVBoxLayout(self.freq_container)
        self.freq_params_h_layout = Qt.QHBoxLayout()
        self.freq_params_h_layout.addWidget(self._bandwidth_range_win)
        self.freq_params_h_layout.addWidget(_update_graph_button_push_button)
        self.freq_params_v_layout.addWidget(self._center_freq_range_win)
        self.freq_params_v_layout.addLayout(self.freq_params_h_layout)
        self.freq_container.setVisible(False)

        #RIGHT LAYOUT
        self.top_right_layout.addLayout(self.directory_params_layout)
        self.top_right_layout.addLayout(self.data_params_layout)
        self.top_right_layout.addWidget(self._graphic_band_choose_tool_bar)
        self.top_right_layout.addWidget(self.freq_container)
        self.top_right_layout.addWidget(self.dynamic_canvas)
        self.top_right_layout.addWidget(self.tableWidget)

        self.top_layout.addLayout(self.top_left_layout)
        self.top_layout.addLayout(self.top_right_layout)

        self.updateScanDataForFreq()
        self.startContinuosBandTimer()
        self.updateTableData()
        #self.startUpdateTableTimer()

    def chooseDirectory(self):
        file = str(QFileDialog.getExistingDirectory(self, "Select Directory",
self.directory))
        if len(file) > 0:
            self.set_directory_entry(file)
            self.clearGraph()
            self.clearTable()
            self.cancelUpdateTableTimer()
            self.updateTableData()
            #self.startUpdateTableTimer()

    def startUpdateTableTimer(self):
        self.updateTimer = QtCore.QTimer()
        self.updateTimer.setInterval(13000)
        timerCallback = functools.partial(self.updateTableData)
        self.updateTimer.timeout.connect(timerCallback)
        self.updateTimer.start()

    def cancelUpdateTableTimer(self):
        try:
            self.updateTimer.stop()
        except:
            print("All timer not initialized yet")
```

```
    def startUpdateAllTimer(self):
        self.timer = QtCore.QTimer()
        self.timer.setInterval(5000)
        timerCallback = functools.partial(self.updateScanData)
        self.timer.timeout.connect(timerCallback)
        self.timer.start()

    def cancelUpdateAllTimer(self):
        try:
            self.timer.stop()
        except:
            print("All timer not initialized yet")

    def startContinuosBandTimer(self):
        self.band_timer = QtCore.QTimer()
        self.band_timer.setInterval(2000)
        timerCallback = functools.partial(self.updateFreqAndScanData)
        self.band_timer.timeout.connect(timerCallback)
        self.band_timer.start()

    def cancelContinuousBandTimer(self):
        try:
            self.band_timer.stop()
        except:
            print("Continuous timer not initialized yet")

    def startUpdateBandTimer(self):
        self.band_timer = QtCore.QTimer()
        self.band_timer.setInterval(5000)
        timerCallback = functools.partial(self.updateScanDataForFreq)
        self.band_timer.timeout.connect(timerCallback)
        self.band_timer.start()

    def cancelUpdateBandTimer(self):
        try:
            self.band_timer.stop()
        except:
            print("Band timer not initialized yet")

    def updateFreqAndScanData(self):
        if (self.center_freq+self.samp_rate >= self.freq_max):
            self.center_freq = self.samp_rate/2
        else:
            self.center_freq += self.samp_rate
        self.updateScanDataForFreq()

    def updateTableData(self):
        powers = []
        freqs = []
        compare_powers = []
        compare_freqs = []
        value_list = []
        for center_freq in
range(int(self.samp_rate/2),int(self.freq_max),int(self.samp_rate)):
```

```
            self.readFilesForFreq(center_freq, self.samp_rate, self.FFT_size, powers,
freqs, compare_powers, compare_freqs, value_list)
        self.addValuesToTable(value_list)

    def updateScanData(self):
        powers = []
        freqs = []
        compare_powers = []
        compare_freqs = []
        value_list = []
        for center_freq in
range(int(self.loop_min_freq),int(self.loop_max_freq),int(self.samp_rate)):
            self.readFilesForFreq(center_freq, self.samp_rate, self.FFT_size, powers,
freqs, compare_powers, compare_freqs, value_list)
        self.plotNewValues(freqs, powers, compare_freqs, compare_powers)

    def updateScanDataForFreq(self):
        powers = []
        freqs = []
        compare_powers = []
        compare_freqs = []
        value_list = []
        self.readFilesForFreq(self.center_freq, self.samp_rate, self.FFT_size, powers,
freqs, compare_powers, compare_freqs, value_list)
        self.plotNewValues(freqs, powers, compare_freqs, compare_powers)

    def addValuesToTable(self, value_list):
        if len(value_list) == 0:
            return
        _list = sorted(value_list, key=self.getKey, reverse=self.table_sort_reverse)
        self.tableWidget.setRowCount(len(value_list))
        self.tableWidget.setColumnCount(len(value_list[0]))
        self.tableWidget.setHorizontalHeaderLabels(['Freq. (MHz)', 'Max. Diff.(dBm)',
'Min Diff.(dBm)', 'Avg. Diff.(dBm)', '% > Thr.'])
        for index in range(0, len(value_list), 1):
            current_value = _list[index]
            self.tableWidget.setItem(index,0, QTableWidgetItem(str(current_value[0])))
            self.tableWidget.setItem(index,1, QTableWidgetItem(str(current_value[1])))
            self.tableWidget.setItem(index,2, QTableWidgetItem(str(current_value[2])))
            self.tableWidget.setItem(index,3, QTableWidgetItem(str(current_value[3])))
            self.tableWidget.setItem(index,4,
QTableWidgetItem(str(current_value[4]*100)))

    def clearTable(self):
        self.tableWidget.clear()

    def clearGraph(self):
        self._dynamic_ax.clear()
        self._dynamic_ax.figure.canvas.draw()

    def check_graph_params(self):
        self.loop_min_freq = (self.center_freq_range - self.bandwidth_range/2) * 1e6
        self.loop_max_freq = (self.center_freq_range + self.bandwidth_range/2) * 1e6
        self.updateScanData()
```

```
    def getKey(self, item):
        return item[self.table_sort_index]

    def tableClicked(self, item):
        if item != self.table_sort_index:
            self.table_sort_index = item
        else:
            self.table_sort_reverse = not self.table_sort_reverse
        self.updateTableData()

    def plotNewValues(self, freqs, powers, compare_freqs, compare_powers):
        if (len(powers) > 0):
            self.clearGraph()
            self._dynamic_ax.plot(compare_freqs, compare_powers, color='red')
            self._dynamic_ax.plot(freqs, powers)
            self._dynamic_ax.figure.canvas.draw()

    def readFilesForFreq(self, center_freq, samp_rate, FFT_size, powers, freqs,
compare_powers, compare_freqs, _list):
        file_base_power = "power_%.0fMHz_%.0fMsps_%dFFT" % (center_freq // 1e6,
samp_rate // 1e6, FFT_size)
        filename_power = "{dir}/{file}.txt".format(dir=self.directory,
file=file_base_power)
        file_base_compare = "compare_%.0fMHz_%.0fMsps_%dFFT" % (center_freq //
1e6, samp_rate // 1e6, FFT_size)
        filename_compare = "{dir}/{file}.txt".format(dir=self.directory,
file=file_base_compare)
        compare_exists = False
        try:
            file_power = open(filename_power, 'r')
            file_power_index = float(file_power.readline()) #read number of values per
row of powers
            try:
                file_compare = open(filename_compare, 'r')
                file_compare_index = float(file_compare.readline())
                compare_exists = True
            except Exception:
                print("No compare file found {file}".format(file=file_base_compare))
            for power_line in file_power.readlines():
                power = float(power_line.split("@")[0])
                powers.append(power)
                freqs.append(float(power_line.split("@")[1]))
                if compare_exists:
                    line = file_compare.readline()
                    values = line.split("@")[0]
                    freq = float(line.split("@")[1])
                    values_array = values.split(";")
                    exceeded_number = float(values_array[0])
                    exceeded_average = float(values_array[1])
                    diff_min = float(values_array[2])
                    diff_average = float(values_array[3])
                    diff_max = float(values_array[4])
                    if freq > 1: #HackRF One supports values from 1MHz to 6GHz
```

```
                    _list.append((freq, diff_max, diff_min, diff_average,
exceeded_average))
                compare_powers.append(power + diff_max)
                compare_freqs.append(freq)
        except Exception:
            print("Exception reading file {file} or {file2}\n".format(file=file_base_power,
file2=file_base_compare))
            return 0
        file_power.close()
        if compare_exists:
            file_compare.close()

    def closeEvent(self, event):
        self.settings = Qt.QSettings("GNU Radio", "top_block")
        self.settings.setValue("geometry", self.saveGeometry())
        event.accept()

    def get_directory_entry(self):
        return self.directory

    def set_directory_entry(self, directory_entry):
        self.directory = directory_entry
        Qt.QMetaObject.invokeMethod(self._directory_entry_line_edit, "setText",
Qt.Q_ARG("QString", str(self.directory)))

    def get_base_scan_button(self):
        return self.base_scan_button

    def set_base_scan_button(self, base_scan_button):
        self.base_scan_button = base_scan_button
        if (base_scan_button == 1):
            subprocess.call("./01-scan_base.py {arg1} {arg2}
{arg3}".format(arg1=self.directory, arg2=int(self.samp_rate//1e6),
arg3=self.FFT_size),shell=True)

    def get_spectrum_scan_button(self):
        return self.spectrum_scan_button

    def set_spectrum_scan_button(self, spectrum_scan_button):
        self.spectrum_scan_button = spectrum_scan_button
        if (spectrum_scan_button == 1):
            subprocess.call("./02-scan_spectrum.py {arg1} {arg2}
{arg3}".format(arg1=self.directory, arg2=int(self.samp_rate//1e6),
arg3=self.FFT_size),shell=True)

    def get_band_scan_button(self):
        return self.band_scan_button

    def set_band_scan_button(self, band_scan_button):
        self.band_scan_button = band_scan_button
        if (band_scan_button == 1):
            subprocess.call("./03-scan_band.py {arg1} {arg2}
{arg3}".format(arg1=self.directory, arg2=int(self.samp_rate//1e6),
arg3=self.FFT_size),shell=True)
```

```
def get_jammer_button(self):
    return self.jammer_button

def set_jammer_button(self, jammer_button):
    self.jammer_button = jammer_button
    if (jammer_button == 1):
        subprocess.call("./04-jammer.py")

def get_graphic_band_choose(self):
    return self.graphic_band_choose

def set_graphic_band_choose(self, graphic_band_choose):
    self.graphic_band_choose = graphic_band_choose
    self._graphic_band_choose_callback(self.graphic_band_choose)

def get_update_directory_button(self):
    return self.update_directory_button

def set_update_directory_button(self, update_directory_button):
    self.update_directory_button = update_directory_button
    if update_directory_button == 1:
        self.chooseDirectory()

def get_update_graph_button(self):
    return self.update_graph_button

def set_update_graph_button(self, update_graph_button):
    self.update_graph_button = update_graph_button

def get_samp_rate_chooser(self):
    return self.samp_rate_chooser

def set_samp_rate_chooser(self, samp_rate_chooser):
    self.samp_rate_chooser = samp_rate_chooser
    self._samp_rate_chooser_callback(self.samp_rate_chooser)
    self.samp_rate = samp_rate_chooser * 1e6

def get_FFT_size_chooser(self):
    return self.FFT_size_chooser

def set_FFT_size_chooser(self, FFT_size_chooser):
    self.FFT_size_chooser = FFT_size_chooser
    self._FFT_size_chooser_callback(self.FFT_size_chooser)
    self.FFT_size = FFT_size_chooser

def get_center_freq_range(self):
    return self.center_freq_range

def set_center_freq_range(self, center_freq_range):
    self.center_freq_range = center_freq_range

def get_bandwidth_range(self):
    return self.bandwidth_range
```

```
def set_bandwidth_range(self, bandwidth_range):
    self.bandwidth_range = bandwidth_range

def get_update_graph_button(self):
    return self.update_graph_button

def set_update_graph_button(self, update_graph_button):
    self.update_graph_button = update_graph_button
    if update_graph_button == 1:
        self.check_graph_params()

def get_update_params_button(self):
    return self.update_params_button

def set_update_params_button(self, update_params_button):
    self.update_params_button = update_params_button

def get_graphic_band_choose(self):
    return self.graphic_band_choose

def set_graphic_band_choose(self, graphic_band_choose):
    self.cancelUpdateAllTimer()
    self.cancelUpdateBandTimer()
    self.cancelContinuousBandTimer()
    self.graphic_band_choose = graphic_band_choose
    self._graphic_band_choose_callback(self.graphic_band_choose)
    self.index = graphic_band_choose
    self.freq_container.setVisible(False)
    if (self.index == 0) :#CONTINUOUS
        self.center_freq = self.samp_rate / 2
        self.updateScanDataForFreq()
        self.startContinuosBandTimer()
        return
    elif (self.index == 1) :#ALL
        self.updateScanData()
        self.startUpdateAllTimer()
        self.freq_container.setVisible(True)
        return
    elif (self.index == 2) :#433MHz
        self.center_freq = 430e6 if self.samp_rate == 20e6 else 435e6
    elif (self.index == 3): #868MHz
        self.center_freq = 870e6 if self.samp_rate == 20e6 else 865e6
    elif (self.index == 4): #Wifi 2.4 1
        self.center_freq = 2410e6 if self.samp_rate == 20e6 else 2405e6
    elif (self.index == 5): #Wifi 2.4 2
        self.center_freq = 2430e6 if self.samp_rate == 20e6 else 2415e6
    elif (self.index == 6): #Wifi 2.4 3
        self.center_freq = 2450e6 if self.samp_rate == 20e6 else 2425e6
    elif (self.index == 7): #Wifi 2.4 4
        self.center_freq = 2470e6 if self.samp_rate == 20e6 else 2435e6
    elif (self.index == 8): #Wifi 2.4 5
        self.center_freq = 2490e6 if self.samp_rate == 20e6 else 2445e6
    elif (self.index == 9): #Wifi 2.4 6
```

```
            self.center_freq = 2490e6 if self.samp_rate == 20e6 else 2455e6
        elif (self.index == 10): #Wifi 2.4 7
            self.center_freq = 2490e6 if self.samp_rate == 20e6 else 2465e6
        elif (self.index == 11): #Wifi 2.4 8
            self.center_freq = 2490e6 if self.samp_rate == 20e6 else 2475e6
        elif (self.index == 12): #Wifi 2.4 9
            self.center_freq = 2490e6 if self.samp_rate == 20e6 else 2485e6
        elif (self.index == 13): #Wifi 2.4 10
            self.center_freq = 2490e6 if self.samp_rate == 20e6 else 2495e6
        self.updateScanDataForFreq()
        self.startUpdateBandTimer()

def main(top_block_cls=top_block, options=None):

    if StrictVersion("4.5.0") <= StrictVersion(Qt.qVersion()) < StrictVersion("5.0.0"):
        style = gr.prefs().get_string('qtgui', 'style', 'raster')
        Qt.QApplication.setGraphicsSystem(style)
    qapp = Qt.QApplication(sys.argv)

    tb = top_block_cls()
    tb.start()
    tb.show()

    def quitting():
        tb.stop()
        tb.wait()
    qapp.aboutToQuit.connect(quitting)
    qapp.exec_()


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

Glossary

AI: Artificial Intelligence

API: Application Programming Interface

DSP: Digital Signal Processing

FFT: Fast Fourier Transform

FPV: First Person View

GB: GigaByte

GNSS: Global Navigation Satellite Systems

IDE: Integrated Development Environment

IQ: In-Phase and Quadrature

kB: KiloByte

MB: MegaByte

ML: Machine Learning

OS: Operative System

OSMOCOM: Open Source Mobile Communications

RF: Radio frequency

SDK: Software Development Kit

SDR: Software Defined Radio

UAV: Unmanned Aerial Vehicles