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 fulfillment

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 public areas putting in risk the integrity of people. In order to address these issues, we propose the creation of a software tool that can help detect drones and block both its communications with the remote controller, and it's global navigation satellite system (GNSS) capabilities.

This solution runs in a Raspberry Pi under an Ubuntu Server installation, and use an external software defined radio (SDR) named HackRF One for both reception (Rx) and transmission (Tx) of radio frequency (RF) signals. This software tool has been developed using Python as the programming language, and GNURadio as the software development kit (SDK) that gives the system the capacbility to execute telecommunications tasks.

The solution consists of five Python scripts with its respective user interface for a manual operation. The main script works as a controller for the four remaining RF reception and transmission scripts, and is in charge of providing them with its main execution parameters. It also offers visual tools to explore the RF data generated by the others scripts. The base script is is in charge of obtaining our reference power values for frequencies between 1MHz and 6 GHz. The spectrum scan and band scan scripts are in charge of comparing the real time power values against the reference values, and generate statistics that can help us identify the frequencies in which there is an unusual RF activity. The last script is in charge of generating a noisy signal that can block or interfere communications in different frequency bands where we expect drone RF activity.

It was proved that despite the computation limits of a Raspberry Pi, this software can be installed and executed in this computer, scanning the RF spectrum between 1MHz and 6 GHz, and identifying unusual RF activity in frequencies used by drones. It was also proved that a noisy signal generated by our system can indeed interfere or block completely the communications of a drone with its remote controller, and with the GNSS. The solution proposed here is a starting point for a more robust UAV detection and jamming system.

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 people's 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 face this problem, we must follow two lines of work in order to countermeasure the drone threat. The first one is the detection, that can be handled with different alternatives such as sound recognition, image recognition, radar detection and radio detection. The other one is to handicap the drone's normal functioning, which can include radio interference, radio control takeover, 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 handicap the drone's communications.

To accomplish this task we will use a Raspberry Pi as the solution's host computer, together with an SDR peripheral called HackRF One, the GNU Radio SDK as our telecommunications framework, and Python as the programming language of our software.

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. In this project we will take into consideration only civil-use UAVs.

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 fly this type of aircraft.

Aeronautical regulations in Spain state the requirement of a license to pilot remotely a drone professionally, they also indicate that in areas near airports, nor 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 airplane's pilots.

Some other 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 soccer 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 its quantity, number of blades, length and material. 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. These characteristics 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 acceleration, 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 autonomous 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 UAV it can also show telemetry data from the drone and multimedia [4].

The previously described components are the most common found in civil-use 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 can be equipped with heat camera sensor to control the cattle from the air, 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 its heat camera [6].

1.3. Drone footprints and detection

Since the components of drones have been already explained in the previous section, we can start discussing about the traces or footprints that a UAV leaves while it is in operation. 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 receiving and transmitting RF signals.

1.4. Drone communications

In the scope of this work we will focus on drones that have the capability to communicate with a remote controller or with GNSS, therefore, we will use the RF footprint detection approach. For this reason it is fundamental to understand how communications work in UAVs.

There are at least four purposes to establish communications in a drone, and not all of them are addressed to a remote controller: command and control, telemetry, multimedia and orientation or navigation.

Command and control, and telemetry refer to all communications between the remote controller and the UAV to send flight orders and receive back data about the flight conditions and sensors 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 communication with GNSS systems to obtain the drone's position for its orientation in both user controlled and autonomous flights.

Different types of communications can be established at different frequency bands and the used transmission system varies according to the manufacturer and drone model as we can see in Table 1.1.

Table 1.1.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 – 1.610 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	

Drones communications are performed in different frequency bands of the spectrum going from as low as 27 MHz up to 5.8 GHz with several distinct transmission systems.

1.5. Adopted Solution

The main objective of this work is to build a software tool that can help detect drones and jam its communications and navigation capabilities using a HackRF One SDR peripheral, and GNURadio as the telecommunications SDK with Python as the programming language. The built software must be able to be executed and proved in a Raspberry Pi.

The approach to perform the detection is by initially measuring the power levels across the spectrum from 1MHz to 6 GHz, establishing a base level for each frequency as our reference. We choose these limits both because it is the HackRF One range, and because as we saw in Table 1, UAVs communications are established from frequencies as low as 27 MHz up to 5.8 GHz, which are included in this range.

Once the previous step is complete, we can perform a real time scan in which we compare these values against the reference levels, in order to detect unusual RF activity and determine the frequencies were this is happening, so that with the help of a graphical tool we can analyze frequencies of interest and determine visually the existance of drone RF activity.

To proceed with the jamming of the drone's communication or GNSS capabilities, we give the user the option to generate a noisy signal that can produce interference or interrupt definitively the communications between the drone and its remote controller, or with the orientation satellites.

This solution is intended to be the base of a more robust drone detection and inhibition system. Further work can be focused on the implementation of DSP techniques to obtain cleaner data from the source; focus on the use of amplifiers for a better Rx and TX; or focus on automatic detection techniques based on machine learning (ML) algorithms or artificial intelligence (AI) using neural networks.

2. State of the art

2.1. Drone Detection and jamming

Drone detection and jamming has become in the recent years a field of study due to several security problems caused by civil UAVs as we explained in the introduction. This situation has led to a scenario in which governments are investing in anti-drone tools to face this menace, with the police forces acquiring drone disabling equipment and cities with airports deploying drone detection systems to prevent unauthorized UAV activity in restricted air space.

Spain's Police has publicly announced in social networks the use of anti-drone guns in massive events (Figure 2.1), specifically the UAV-D04JAI model 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 systems 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 emergencies service a system called *Caelus*, which is in charge of detecting drones in the city's aerospace, specially in heights between 250 to 400 meters where the UAVs become a threat to airplanes flying over the city. The solution counts with antennas distributed across the city and an artificial intelligence system in charge of identifying the drone's flight information and 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 get 3D position of the UAVs 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 provided by USA's Raytheon, which offers an infrared sensor to detect, identify and track rogue drones. And once a UAV is 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 the jamming of drones. With our focus in the RF detection, we use a passive approach in which we eavesdrop the communications in specific frequency bands, and according to the detected signal's characteristics it can be determined if it belongs to a drone [20]. One way to detect the presence of drones is by analyzing its RF characteristics such as bandwidth, modulation, its frequency hopping mechanism, and also by decoding its information. But in practice with a vast number of protocols and different bandwidths used for the transmissions, it doesn't seem feasible to identify a drone without full knowledge of all these RF characteristics for all drones wanted to be detected [21].

Even though having information on all available drones seems like an unfeasible task, there is an initiative 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 noisy signal that can be transmitted in frequencies that depend on the communication mechanism used by the drone. If the drone is using the WiFi operation mode, then we can generate the noise in a specific WiFi channel [24]; or if it using a different mode with frequency hopping we should generate the noise along the whole 2.4 GHz band [25].

A different approach to countermeasure the drone presence, is by hijacking the communications exploiting protocol security issues. A researcher exploited failures in the DSMX [11] protocol, and with a device called *Icarus* has proven the feasibility of hijacking drones by injecting malicious control packets in its communications, replacing the control information source and leaving the original remote controller unable to maneuver 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 and its orientation capabilities. This will result in the creation of five different Python scripts. The scripts will give the user the possibility to scan the RF spectrum from 1 MHz to 6 GHz and establish reference power levels. Also the scripts will detect unusual RF activity comparing real time power values against the reference values. It will also provide with graphic tools to see the spectrum in a determined bandwidth. Another script will allow the user to generate a jamming signal that can interfere or block drone communications with the remote controller or orientation satellites. 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 this 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
- Huawei Y7 Mobile Smartphone

Software used for implementation

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

Software used for testing

- Htop
- DJI GO 4 Mobile Application

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 and execute the software, and the peripheral SDR equipment to receive and transmit RF signals.

3.1.1. HackRF One

HackRF One is a 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 base-band filter, and the antenna 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 base-band ("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 Table 3.1.

Table 3.1.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 maximum transmission power values, the TX power can't be set to a specific value in the device and we expect the output power to be in the ranges specified. The device doesn't provide certifications that indicate that transmissions comply with government regulations, so the user is the sole responsible of using it adequately.

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

The HackRF One can be used as a standalone device with its own built-in functions and APIs that give control over the device and that can perform a scan of up to 8 GHz per second [31]. However using these functions for the implementation of a project will only work for this device and will not be compatible with other SDR devices.

HackRF One is a quadrature sampling system, which means that all samples are IQ samples. As a consequence in the frequency 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. Also, since we are working with complex sampling, the sample rate in Msps indicates the bandwidth in 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 1; also due to its half-duplex capability, which allows us to both receive and transmit, even though not simultaneously; and finally because of its affordable price.

For simplicity, in the remaining sections of this document we will refer to this device only as the HackRF, excluding its model name.

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 64 bit quadcore processor and with 1 GB SD RAM memory. For networking purposes 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 (Figure 3.2), 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 in this computer. Its setup steps can be found in the Annex I – Raspberry setup.

The Raspberry Pi 3B+ board 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 Pi, 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.

For simplicity, in the remaining sections of this document we will refer to this device only as the Raspberry, excluding its full name and model name.

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 1.90 GHz 64 bit 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 signal generator to verify that our project can identify RF activity in 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. Regarding communications, 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 the European region. In this region the drone is authorized to use the 2.4-2.4835 GHz band with a power up to 20 dBm, and the 5.725-5.850 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 (Table 3.2). It doesn't have a built-in GPS so it doesn't offer autonomous flight mode [33].

When it works in WiFi mode, the drone acts as an Access Point (AP) and creates its own WiFi network. A mobile device 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 UAV. There are led lights in the drone that indicate with a green color 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 we can identify the channel in which the UAV has established its communications. 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).

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

Table 3.2.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

Table 3.3.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

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 transmission system [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 center frequencies. The Table 3.3 summarizes all the information of the RC operation mode.

In the mobile app we have options to see the 2.4 GHz band RF activity in real time (Figure 3.7), and we find there are two drone RF operation modes: automatic and fixed. The automatic mode selects the channel which is cleaner to establish the communications, and selects by default the 20 MHz bandwidth and the upper 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 test drone, it allowed us only to configure transmission in 10 MHz or 20 MHz bandwidth, so we could choose among 136 different channels for our communications.

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

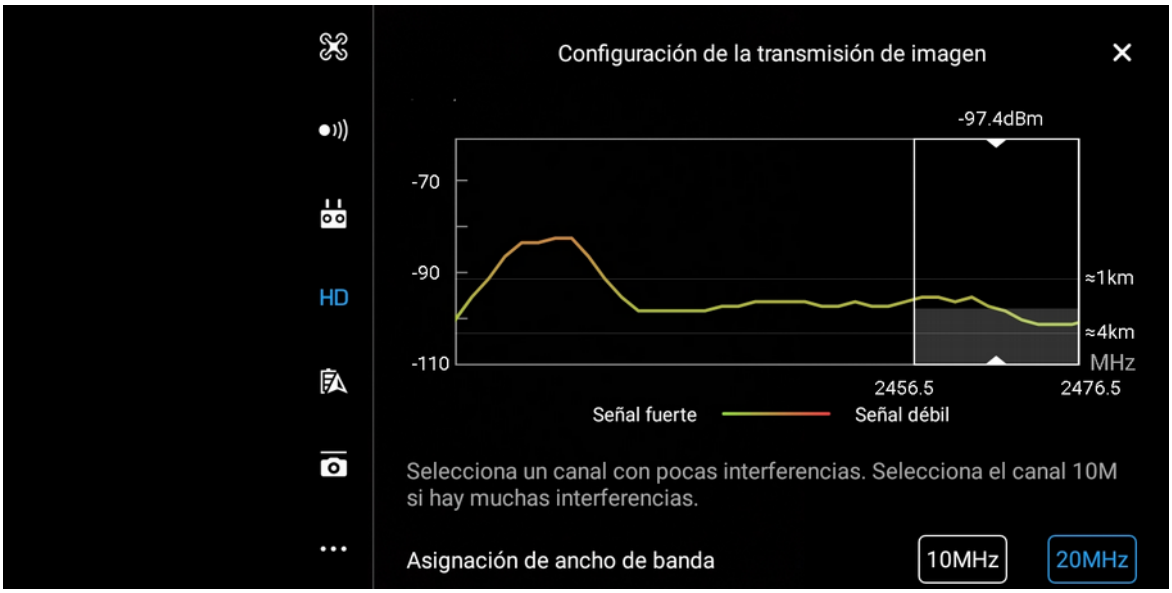


Figure 3.7: Screenshot of the DJI Go 4 app in the RC communications option. Communication is established in in 2466.6 MHz with 20 MHz bandwidth.

3.2.4. Huawei Y7 Mobile Smartphone

It is a mobile smartphone with a Qualcomm Snapdragon 430 processor, with 3.0 GB of RAM Memory and 32 GB of disk space. It offers WiFi and Bluetooth connectivity in 2.4 GHz. It comes equipped with the Android operative system at its 8.0 version.

It is used to install the DJI GO 4 application provided by the DJI manufacturer as a remote controller app for the DJI Mavic Pro drone.

It can work as a standalone remote controller in charge of the transmission and reception of data to and from the drone in the WiFi operation mode. It can also work as a screen for FPV connected to the GL200A remote controller in the RC mode.

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 that will allow the user to control the different execution parameters available.

3.3.1. GNURadio

GNU Radio is an 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].

GNURadio is a telecommunications framework that is in charge of all the signal processing by means of software. It can be used to develop applications that receive data from digital streams or that push data into digital streams, which are then transmitted by means of hardware.

The SDK comes equipped with readily available blocks that have their specific function just like in hardware radios, and among them we have filters, channel codes, synchronization elements, equalizers, demodulators, vocoders and decoders. There is also an option to create custom blocks in case that a function is missing or if a specific implementation is needed. The blocks connect to each other passing data between them and creating bigger systems for data processing. The data can be in the form of bits, bytes, vectors, bursts or more complex data types.

GNURadio has its own integrated development environment (IDE) named GNURadio Companion, which provides a drag and drop graphical interface to easily create telecommunications systems using all the available blocks. Files are created under the *grc* extension, and its execution generates a Python script which is the final executable file. The source code of the scripts reflects the GNURadio blocks that compose the system, and also user interface components under the QT framework. Even though the IDE creates automatically Python scripts, applications can also be developed using C++ as the programming language.

Even though we can use GNURadio Companion to create complex telecommunications systems, there are some limitations using it when creating more complex software solutions.

We have found that some native blocks provide no flexibility to change certain parameters that we might need to alter. One example is the LogPowerFFT block, which is in charge of converting the complex data stream coming from the SDR into power values in dBm using the Blackman-Harris window. If we want to change the used window, there is no option given to do it, so a new custom block must be created just for this purpose.

Even though it is not a GNURadio specific issue, it is a limitation to our project the default process of saving information generated by the SDR. These files consist of a large register of data in form of bytes or numbers, and don't contain information about the parameters such as the sample rate or frequencies. And in the case of our project, in which we are going to obtain data from frequencies between 1 MHz and 6 GHz, it is desirable that all data generated reflects information about the frequencies and the different execution parameters. Additionally, these files' size grow indiscriminately depending on the sampling rate and the time during which the data is obtained, because it is common that the new samples get appended to the file, and its size can easily reach the order of the MB in a few seconds, and the order of the GB after a few minutes. This is

an important limitation because we want to execute our solution in a low processing power and low disk space computer like the Raspberry Pi. And also, we have to be able to check if the information that we are gathering in the files is consequent with what we expect, thus the format in which we store the data has to be different and easy to read. As we will see later, with the use of custom blocks we can create our own file format to overcome the file format, readability and size issues.

The most important limitation found for our project is that automatic tasks that change the blocks parameters cannot be implemented directly from the IDE. Although it does provide blocks that allow a user to change manually block parameters, for the objective of this project of creating an automatic spectrum scanner and analyzer, user interaction is discouraged, and no native blocks are found to fulfill this requisite of creating automated tasks. In our case, we must automate frequency shifting in order to scan from 10 MHz up to 6 GHz without requiring the user to perform each change. This major limitation can be overcome creating these tasks by code, modifying manually the Python script and adding these features as we will explain in later sections.

Despite the limitations that GNURadio Companion has for our project, the use of GNURadio as the SDK provides us with benefits that surpass the setbacks. Since this is an open source project, we can have access to the source code of all the blocks available in the IDE and use it to generate new blocks with different characteristics adjusted to our needs. Also it has a growing community of supporters that has built software packages with new blocks and functionalities, as is the case of the Open Source Mobile Communications (OSMOCOM). This group offers the *gr-osmosdr* package which allows to use any OSMOCOM compatible hardware as a source or sink, allowing the execution of an application with several devices, giving us the flexibility to replace the SDR equipment without the need to adapt our project's source code.

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€.

GNURadio can be installed in Windows, MacOS or Linux, and for this project we tried with both MacOS and Linux Ubuntu. In the MacOS installation process we faced several difficulties, like the need to have dependencies on other applications such as XCode and X11. The setup wasn't an straight process and presented several setbacks, but we could get the tool running and working. Its execution wasn't smooth and it presented a screen blink while a GNURadio script was running. The final issue was generated when an OS update was performed, making the scripts unexecutable for an unkown reason.

After all the MacOS difficulties, we tried with a Linux Ubuntu installation, and it showed to be a more straightforward process and provided a stable development environment along the whole project duration. In order to install GNURadio in Linux Ubuntu, we must follow the instructions found in the Annex II - GNURadio setup (Linux-Ubuntu).

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.

3.4.2. DJI GO 4 mobile application

It is a mobile application available to download for the Android operative system directly from the Google Play Store (Figure 3.8). It works as a remote controller app for several DJI drone models including our Mavic Pro.

When working in WiFi mode, it works as a standalone remote controller that sends and receive the control data and multimedia via WiFi channels using the mobile smartphone as the transmitter and receptor (Figure 3.9).

When working in RC mode it works as an FPV screen were the multimedia and telemetry data is displayed, but the command and control data transmission and reception are performed by the Mavic Pro's remote controller.

Figure 3.8: Screenshot of the DJI Go 4 app

Figure 3.9: Screenshot of the DJI Go 4 app working in WiFi mode

3.5. The outcome

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 jammer solution, and we will focus our work on implementing the basis of a functional software. We'll provide flexibility for its execution, making it possible to adapt to the different SDR hardware characteristics and to the user's needs, by means of several controls to customize the execution parameters. At the same time, we leave the door open for new add-ons related to DSP or tools like ML or AI.

For this project we have programmed five executable Python scripts, initially generated by the GNURadio Companion IDE, but later modified manually to adjust its execution to our needs. Additionally we have created three custom GNURadio blocks under our own *gr-tfm* package, which will be in charge of processing the data and creating the file databases.

The main executable program is one script containing four buttons that open other four scripts, and a graphic tool to compare the measured power values and a table were we can see and filter the obtained information. The base script is in charge of obtaining the base values of the spectrum power without drone activity. Spectrum and band scan scripts are in charge of scanning in real time the spectrum and compare the powers with the base values obtained previously. The jamming script is in charge of generating jamming signals to interfere or block communications in a given frequency band.

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 complex data from the HackRF source into power values in dBm with a Hamming window. Another is the *power_analyzer*, which is in charge of creating the base powers file database with a custom 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 blocks we must follow the steps indicated in the Annex III - Custom GNURadio block setup, and for the execution of the scripts we must follow the Annex IV - Scripts setup. 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 which we explain all actions that a user can take with our software. It can be found in the Annex V – User Manual.

The process consists on developing this software project in the laptop computer and then port it to the Raspberry Pi, being fundamental that both devices have installed the same

versions of our development tools like Python and GNURadio, to guarantee a correct compatibility.

3.5.1. Custom Block: logpowerfft_hamming

This GNURadio block is in charge of converting the stream of complex 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 3.10) which does the same operations but with a Blackman-Harris window and it can't be changed (Figure 3.11). 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's source code 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 is the power in dBm. The significant parameters to be configured are the sample rate, the vector length or FFT size and the frame rate.

First, the block decimates the samples stream into vectors with a rate and a length specified in the parameters. These values which are in the time domain are transformed to the frequency domain with the FFT using a Hamming window. We obtain the squared magnitude of this vector, and its respective logarithmic value with decimal base obtaining the power in dBm (Code 3.1).

In an scenario in which we are using 20 Msps as the sample rate and 1024 as the FFT size, what we do here is to transform the complex values stream into a 1024 points vector that occupy 20 MHz.

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

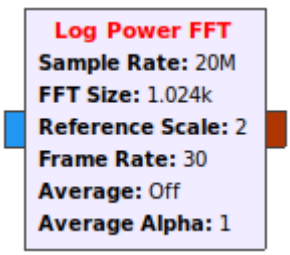


Figure 3.10: Default Log Power FFT block

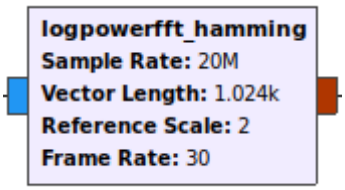


Figure 3.11: Custom Log Power FFT block with Hamming window

```
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 GNURadio 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 by the vector length parameter, the output is a document created in a directory, and its name depends on the parameters: sample rate, center frequency and vector length (Figure 3.12).

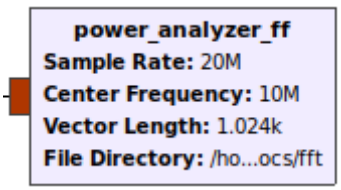


Figure 3.12: Custom power analyzer block

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 very top of it indicating the number of input vectors that have been used. This value is know as the file index and is completely important since this document works with averages.

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 file will have all the new calculated values and is temporary. 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 approach of editing line by line an already created file.

To calculate the new power values, we simply multiply the file index per the current 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 exploited in the base scan script, where we will create a powers file for every frequency analyzed in the range between 1 MHz and 6 GHz.

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. To handle files of these sizes in the data analysis phase, we would need a processing unit with high computation power, and with sufficient space in the disk to store these big 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 genrated 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, making it a it very efficient information storage method,

because the file size will be almost the same for a scan of a few seconds, or a scan of minutes or hours.

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

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

The input is a numeric vector of size specified by the vector length parameter, the output is a document created in a 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 is 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 3.13) or Fixed Value (Figure 3.14). Both modes require a numeric value for them to work.

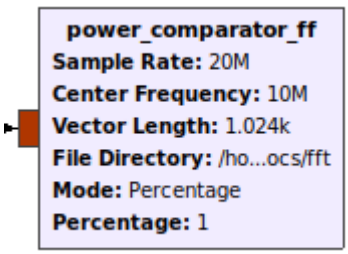


Figure 3.13: Custom power comparator block in mode fixed value

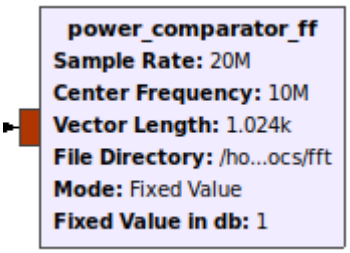


Figure 3.14: Custom power comparator block in mode percentage

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, 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, 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 new input value. If the value exceeds the threshold, then we compare the values to the minimum (value 3), 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)

Therefore each frequency has its own threshold, which gives us flexibility in the comparison, so we don't have an unique threshold value for all frequencies. The comparison values help us spot easily unusual RF activity peaks, and with the help of averages we can see how often values are above threshold for each frequency, indicating id this RF activity is frequent or just an isolated case.

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 – GNURadio custom block: tfm_power_comparator_ff.xml) and *power_comparator_ff.py* (Annex XI – GNURadio custom block: power_comparator_ff.py).

```
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 comparison values calculations in the power_comparator block

3.5.4. Script: Scan Base

This script is in charge of obtaining the power values for all frequencies between 1 MHz and 6 GHz, storing them in a file database.

The script is initially created using the GNURadio Companion IDE, and is composed mainly by three blocks which are the OSMOCOM source, our custom *logpowerfft_hamming* and *power_analyzer* blocks (Figure 3.15). There are also several variables and user interface components. The variables hold the values for the parameters of the lower, central and upper frequencies that limit the scan, the FFT size, the sample rate, the directory where the files will be saved and the frequency switch time. The FFT size, sample rate and the directory are provided by the main script and can't be changed by the user from within this script. The lower and upper frequencies can neither be changed to guarantee a base scan involving all frequencies between 1 MHz and 6 GHz. The only value allowed to be changed by the user is the frequency switch time.

The user interface components display the values of the FFT size, sample rate and directory, and additionally we provide a slider component that allows the user to change the time in which the central frequency changes (Figure 3.16).

The structure of our system in GNURadio is pretty straightforward: the source (HackRF) is configured with a central frequency and sample rate, and feeds all the received data to the *logpowerfft_hamming* block, which converts the complex values stream into vectors with power values in dBm. These vectors are passed to the *power_analyzer* block which creates with them a file for the corresponding central frequency, sample rate and FFT size, averaging the power every time a new vector arrives.

Since we need to obtain the values for all frequencies between 1 MHz and 6GHz, we need to create a task in which the central frequency changes for both the HackRF and the *power_analyzer*. As we explained previously in the GNURadio section, this feature can't be implemented straight from the IDE, and must be written directly in Python code.

To perform this task we choose to use a Timer from the QT framework, which makes a call to a given function every certain amount of time. This amount of time is controlled by the user with a slider, can be altered at any time during the script's execution and is set to 50 ms. by default. The function is in charge of changing the frequency between a lower and upper limit, and depending on the sample rate (Code 3.6). When we set a new frequency value, this value is passed to both the source and the *power_analyzer* so they both know which frequencies they are dealing with.

Using a sample rate of 20 Msps we must perform 300 frequency hops to sweep the spectrum from 1 MHz to 6 GHz, and with a frequency jump time of 50 ms. we can perform a full scan in a time of 15 seconds. Increasing the frequency switch time will result in the obtainment of more data for every frequency, but will also increase the time for a total scan.

In the *power_analyzer* block, each frequency change produces a different file. And for the configuration mentioned in the previous paragraph, 300 files would be generated.

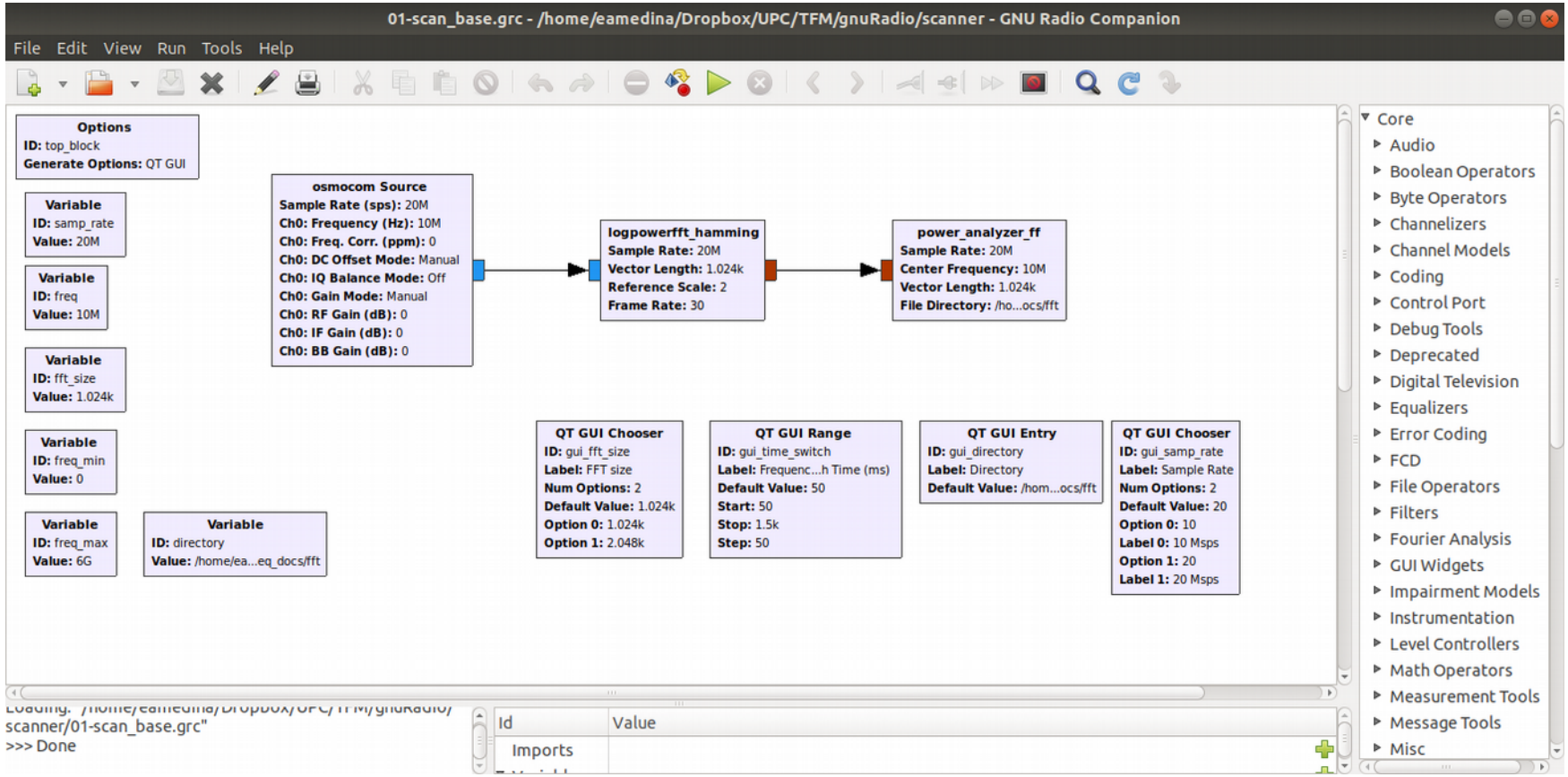


Figure 3.15: GNURadio Companion block structure for base scan

All the files generated by this script are fundamental to obtain the power comparison values in the spectrum and band scan scripts, and to display the comparison graphics and comparison values table in the main script. Therefore, it is mandatory that this script is the first executed script when using this project's software.

The files involved in this script are *01-scan_base.grc* and *01-scan_base.py* (Annex XII – Base Scanner script: 01-scan-base.py).

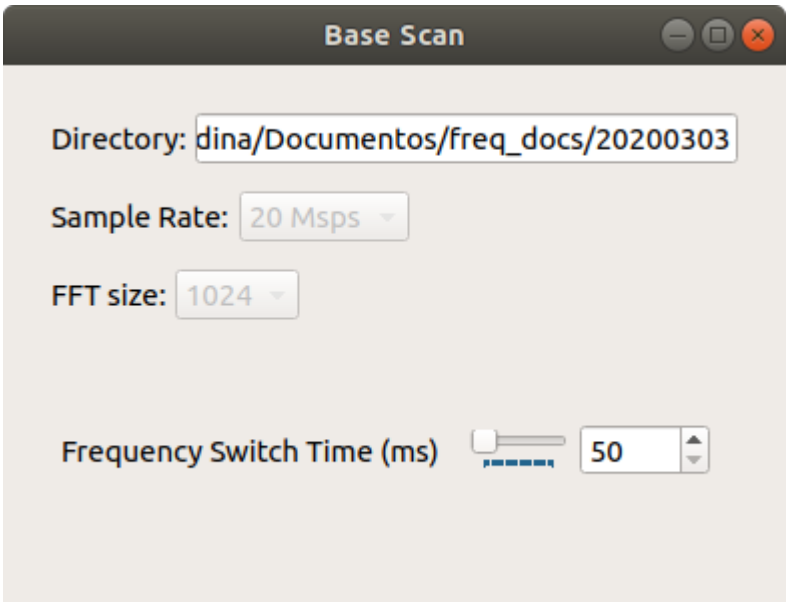


Figure 3.16: User interface components in base scan script

```
def start_timer(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

3.5.5. Script: Spectrum Scan

This script is in charge of comparing the real time power values against the base power values for all the frequencies from 1 MHz to 6 GHz, generating a file database with all the resulting values.

The script is initially created using the GNURadio Companion IDE, and is composed mainly by three blocks which are the OSMOCOM source, our custom *logpowerfft_hamming* and *power_comparator* blocks (Figure 3.18). There are also several variables and user interface components. Same as in the previous script, the variables hold the values for the parameters of the lower, central and upper frequencies that limit the scan, the FFT size, the sample rate, the directory where the files will be saved and the frequency switch time. Additionally we have variables related to the operation mode and the value to calculate the threshold. The FFT size, sample rate and the directory are provided by the main script and can't be changed by the user from within this script.

This script allows the user to control more parameters than in the base scan script. The lower and upper frequencies can be modified, so the comparison process can be done for specific bands instead of the whole spectrum between 1 MHz and 6 GHz. The operation mode and the mode value of the *power_comparator* block can be modified by the user during the execution of the script, in order to modify the power threshold value. Finally, the frequency switch time can also be modified as in the base scan script.

The user interface components display the values of the FFT size, sample rate and directory. We provide slider component to modify the lower and upper frequencies boundaries. There is a selector to choose the operation mode between *Percentage* and *Fixed* modes, and a slider to set the mode value. The frequency switch time is also modified by a slider (Figure 3.17).

The structure of our system is similar to the one of the base scan script. The only difference is that at the end of our flow we have a *power_comparator* block instead of a *power_analyzer*. First we have the source (HackRF), which is configured with a central frequency and sample rate, and feeds all the received data to the *logpowerfft_hamming* block, which is in charge of converting the complex values stream into vectors with power values in dBm. These vectors are passed to the *power_comparator* block which compares the input against the base power values and creates a file for the corresponding central frequency, sample rate and FFT size, calculating differences and averages. For the comparison to work correctly, the base values files must have been created previously in the specified directory with the same frequency, sample rate and FFT size parameters.

Just like in the base scan script, we must create a process that changes the frequency periodically. We reuse the base scan script's solution, by implementing a QT Timer which calls the frequency changing function every certain amount of time that is set by the user, and is set to 250 ms by default. The frequency hops between values limited by the lower and upper frequencies, and its value also depends on the sample rate. The frequency limits can be changed by the user at any time during the execution of the script. When the

frequency changes, the both the source and the *power_comparator* block are informed, so they know with which frequencies they are dealing with.

Using a sample rate of 20 Msps we must perform 300 frequency jumps to sweep the spectrum from 1 MHz to 6 GHz, and with a frequency jump time of 250 ms. we can perform a full scan in 75 seconds. This time will vary if the user defines different frequency boundaries or switch time.

In the *power_comparator* block, each frequency change produces a different file. And for the configuration mentioned in the previous paragraph, 300 files would be generated.

The files involved in this script are *02-scan_spectrum.grc* and *02-scan_spectrum.py* (Annex XIII – Spectrum Scan Script: 02-scan-spectrum.py).

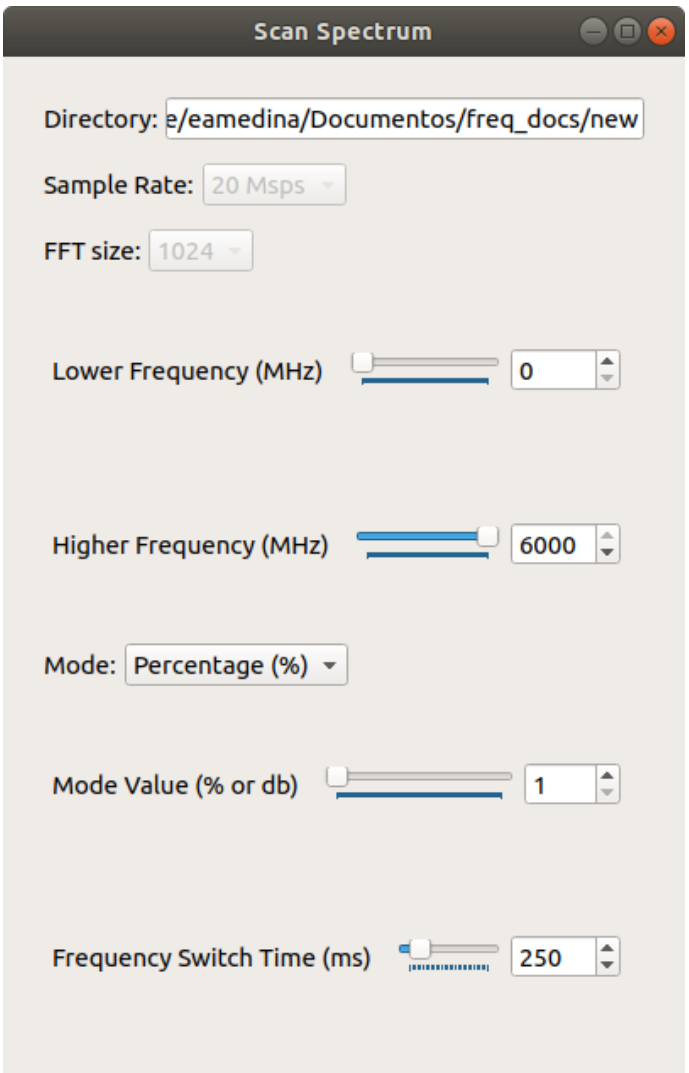


Figure 3.17: User interface components in spectrum scan script

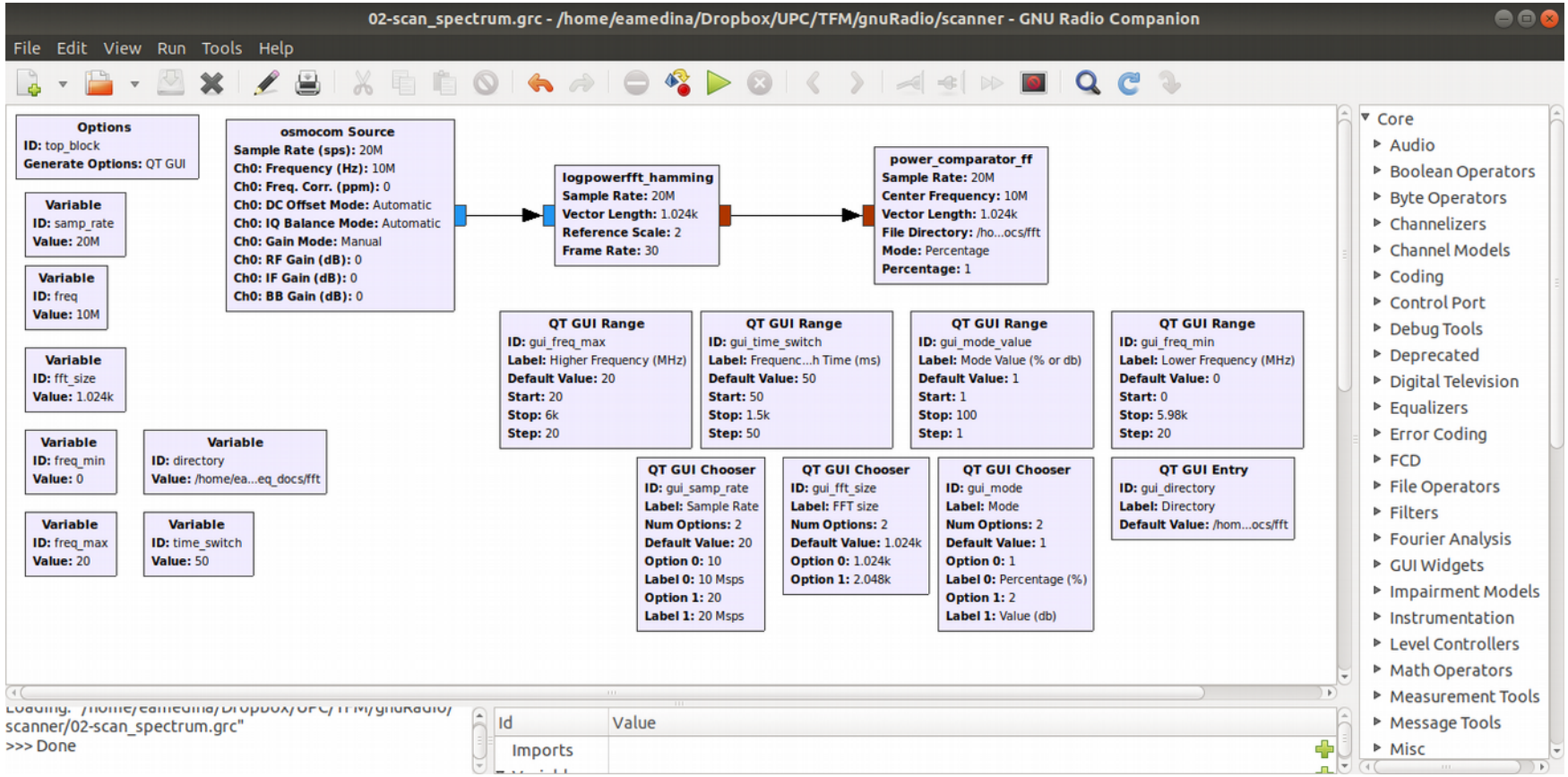


Figure 3.18: GNURadio Companion block structure for spectrum scan

3.5.6. Script: Band Scan

This script is in charge of comparing the real time power values against the base power values for a limited bandwidth of up to 20 MHz, generating a file database with all the resulting values. This script also provides tools to analyze graphically the signals in real time.

The script is entirely created in GNURadio Companion, and is composed mainly by four components which are the OSMOCOM source, a QT GUI Sink and our custom *logpowerfft_hamming* and *power_comparator* block (Figure 3.19). We also have several variables and user interface components. Same as in the previous scripts, the variables are hold the values of the center frequency, the FFT size, the sample rate, the directory, the frequency switch time, the operation mode and values of the *power_comparator* custom block. The FFT size, sample rate and the directory are provided by the main script and can't be changed by the user from within this script. The center frequency, operation mode and value can be modified by the user at any time during the execution of the script.

The user interface components display the values of the FFT size, sample rate and directory. We provide two components that modify the center frequency. One is a slider component where we can choose frequencies between 1 MHz and 6 GHz. And the other one is a selector with a list of frequency bands in which drone communications are performed. These frequency bands are 433 MHz, 868 MHz and the 2.4 GHz band. There is a selector to choose the operation mode between *Percentage* and *Fixed* modes, and a slider to set the mode value to calculate the power threshold. Additionally we have the QT GUI Sink, who provides two graphical modes. This a native GNURadio block which provides options to analyze visually the spectrum with a frequency display (Figure 3.20) or with a waterfall display (Figure 3.21).

The structure of our system is similar to the one of the spectrum scan script. The only difference is that we add the QT GUI Sink. First we have the source (HackRF), which is configured with a central frequency and sample rate, and feeds all the received data to the QT GUI Sink and the *logpowerfft_hamming* block, which is in charge of converting the complex values stream into vectors with power values in dBm. These vectors are passed to the *power_comparator* block which compares the input against the base power values and creates a file for the corresponding central frequency, sample rate and FFT size, calculating differences and averages. For the comparison to work correctly, the base values files must have been created previously in the specified directory with the same frequency, sample rate and FFT size parameters.

The files involved in this script are *03-scan_band.grc* and *03-scan_band.py* (Annex XIV – Band Scan script: 03-scan-band.py).

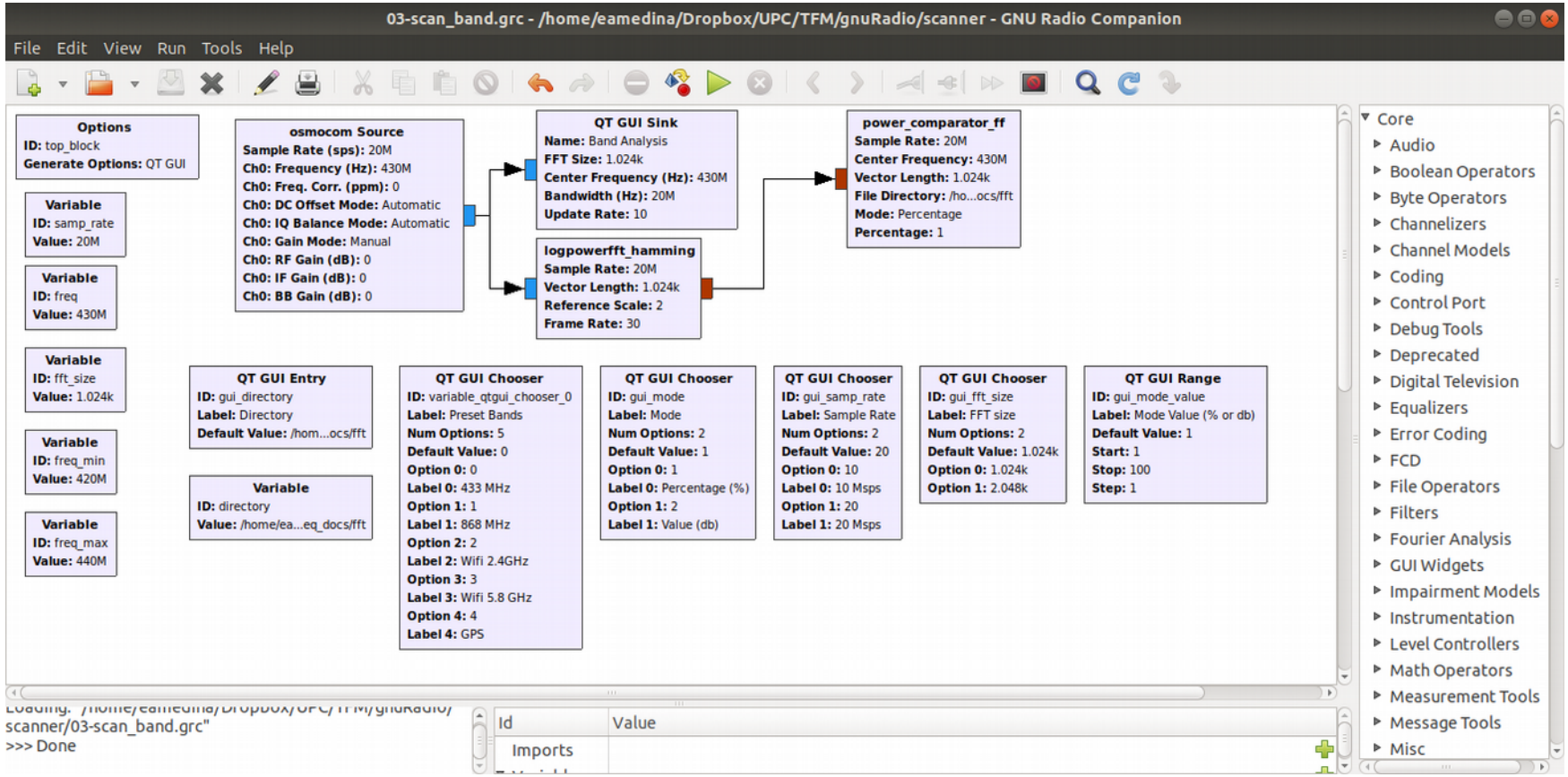


Figure 3.19: GNURadio Companion block structure for band scan

Figure 3.20: User interface components in band scan script with frequency display

Figure 3.21: User interface components in band scan script with waterfall display

3.5.7. Script: Jammer

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

The script is initially created in GNURadio Companion, and is composed mainly by four blocks which are the noise source, a multiply block, a throttle and the OSMOCOM sink which in this case will be the HackRF (Figure 3.22). We also have several variables and user interface components. The variables hold the values for the parameters of the lower, central and upper frequencies, the sample rate/bandwidth of the noise signal, the amplitude of the signal, the RF and IF gains, the operation mode and the frequency switch time. All variables can be modified by the user at any time during the execution of the script.

The user interface components allow to change all the execution parameters of this script. We have a sample rate selector with values of 10 Msps or 20 Msps, which defines the bandwidth of the noise signal. We have controls for the amplitude of the signal, and for the RF and IF gains. We have a selector for the operation mode, which can be *Fixed Band* or *Continuous*. There is also a selector for preset bands configurations when working in *Fixed Band* mode, which includes frequencies in which it is common to find communications or navigation signals such as 433 MHz, 868 MHz, 2.4 GHz or GNSS bands (Figure 3.23). There is a slider for the lower frequency, which works in both modes; other slider for the upper frequency and the frequency switch time, which work only in the *Continuous* operation mode.

The structure of our system consists of a software based noise source, connected to a constant multiply block, and a throttle which limits the generated data to the sample rate, and finally we have the OSMOCOM sink, which transmits the noise data with the HackRF.

We give the user the chance to transmit the noise in a band greater than 20 MHz by means of the *Continuous* operation mode. This mode follows the same principle of the base scan and spectrum scan scripts, where we create a process that changes the frequency periodically. We reuse the solution of the previous scripts and use a QT Timer that calls the frequency changing function at a time specified by the user. The frequency changes its value between the lower and upper boundaries set by the user. This mode is useful if the user needs to generate a noise signal in the whole 2.4 GHz band if needed. The noise won't be present all the time in all frequencies, but this is an option that can be used in case of drones with frequency hopping protocols.

In the contrary, the *Fixed Mode* transmits the noise at a specific frequency given by the value of the lower frequency variable or when selecting one of the preset bands options.

The files involved in this script are *04-jammer.grc* and *04-jammer.py* (Annex XV – Jammer script: 04-jammer.py).

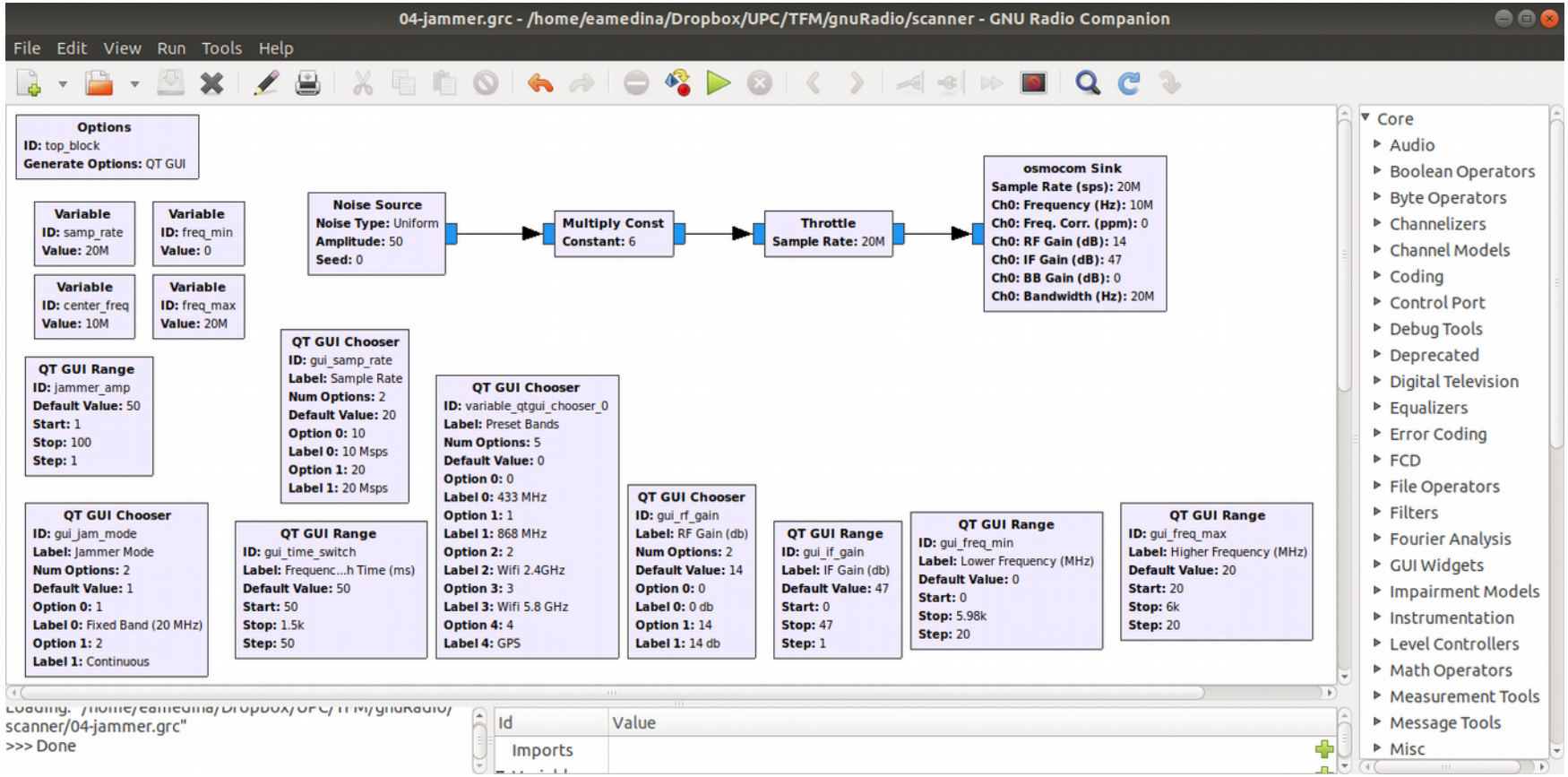


Figure 3.22: GNURadio Companion block structure for jammer script

Figure 3.23: User interface components in jammer script

3.5.8. Script: Main Program

The main script is the controller script which launches the previously explained scripts and provides the user with graphical tools to explore the obtained data.

It has four buttons that launch the base scan, spectrum scan, band scan and jammer scripts. Here we choose the directory where the files generated by our scripts are saved. Also we choose the sample rate and FFT size, allowing for different combinations of these values. These three parameters are passed to the other scripts and can only be modified here. This script also use this parameters to get the data that is displayed in the frequency plot and in the comparisons table. When one of these values changes, the data in the graphic and the table changes too.

If there is no data for a given parameter combination, neither the graphic nor the table displays data (Figure 3.24), but when there is at least data for the base values, the graphic will be displayed.

The frequency plot displays the base power values in blue color and the maximum difference in red color, which helps us identify the peaks of the unusual RF activity detected (Figure 3.25). If there is no comparison data, only the blue data will be displayed. This graphic has an auxiliary component which determines the operation mode. By default it works in the *Continuous* mode, which sets the bandwidth of the graphic to the value of the sample rate, and changes the center frequency every three seconds. There are also preset bands like 433 MHz, 868 MHz and the 2.4 GHz band (separated in chunks), so we can easily configure the graphic to display data in frequencies where we expect drone RF activity. We also give the user an option to customize its graphic with the *All* mode, where the user can set the frequency and the bandwidth of the plot (Figure 3.26). It is useful when we want to display in the graphic a band whose width is greater than 20 MHz, such as all the 2.4 GHz band.

For the table to show data, the comparison values are mandatory. It shows the frequencies, the minimum, maximum and average differences from the base values, and the percentage of values that are above the threshold. All the values are displayed in a different column, and by default the data is sorted in descending order by the maximum difference parameter. With a click on the respective table header, the sorting order can be changed, working as a data filter so the user can easily explore the information by the parameter that is most convenient to analyze.

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 main script and launch all the scripts that are part of this software, we need to follow the indications of the Annex IV - Scripts setup.

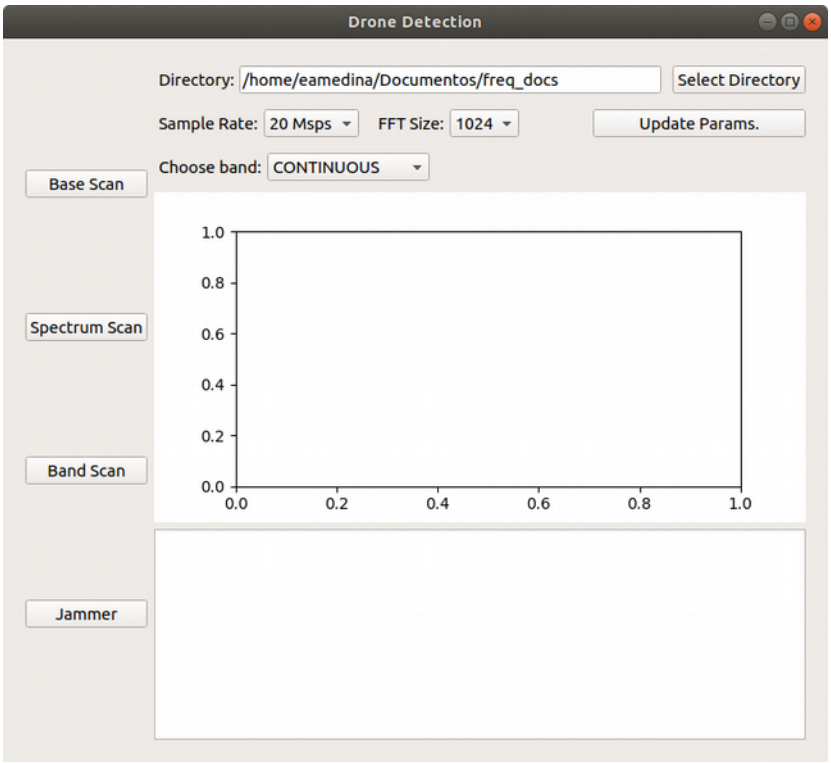


Figure 3.24: Main script with no data

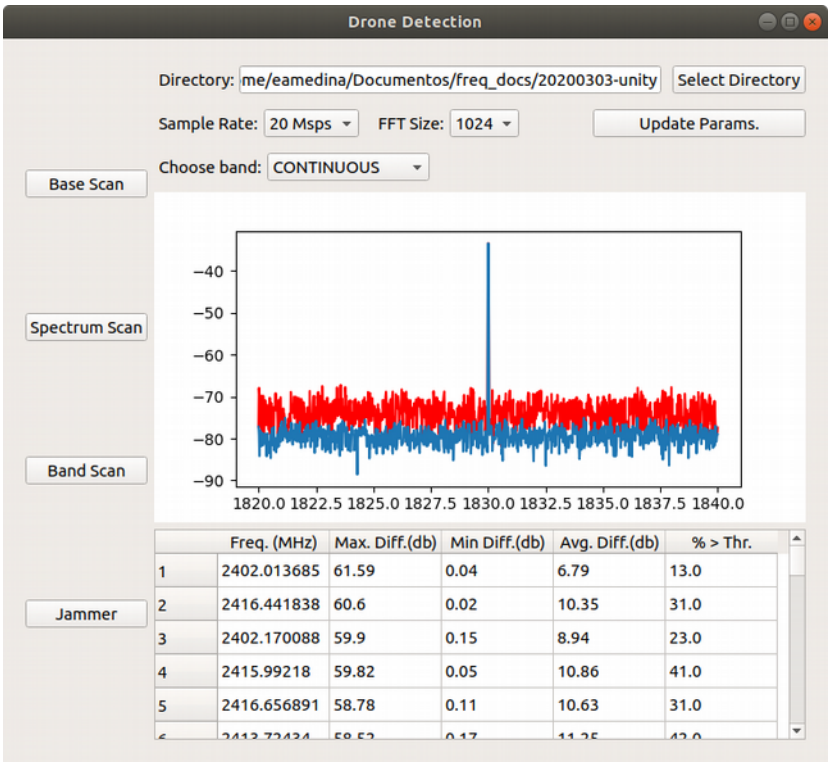


Figure 3.25: Main script with graphic in Continuous mode

Figure 3.26: Main script with graphic in All mode displaying data for the 2.4 GHz band

4. Tests and Results

The tests will be made for the base scan, spectrum scan, band scan and jammer scripts. The equipment used for the test are the ones described in Section 3. For each script we will perform tests with different parameters combinations, and the results will be stored in distinct folders in order to verify easily that all the expected files are created, and to measure the disk space that they occupy.

A total of 28 tests will be performed, where some of them will be done on both the laptop and the Raspberry and others only on the Raspberry. The summary for all tests that are part of this section is found in the Table 4.1.

To verify that our detection solution woks for different parts of the RF spectrum, we will generate signals in frequencies at 27 MHz, 433 MHz and 2.4 GHz, and check if they have been detected.

The computer performance while running the tests will be monitored using the *htop* tool, and is mandatory in all laptop tests.

All tests will be performed in an urban indoor environment in the city of Barcelona, Spain.

Table 4.1.Tests summary

SCRIPT	TEST 1	TEST 2	TEST 3	TEST 4
Base Scan (Laptop, Raspberry)	10 Msps 1024 FFT	10 Msps 2048 FFT	20 Msps 1024 FFT	20 Msps 2048 FFT
Spectrum Scan (Laptop, Raspberry)	10 Msps 1024 FFT	10 Msps 2048 FFT	20 Msps 1024 FFT	20 Msps 2048 FFT
Band Scan (Laptop, Raspberry)	20 Msps 1024 FFT 27 MHz	20 Msps 1024 FFT 433 MHz	20 Msps 1024 FFT 2.4 GHz (WiFi Mode)	20 Msps 1024 FFT 2.4 GHz (RC Mode)
Jammer Script (Raspberry)	20 Msps 27 MHz	20 Msps 433 MHz	20 Msps 1.5 GHz	20 Msps 2.4 GHz

4.1. Base Scan Script Tests

This test's purpose is to verify that this script creates the corresponding file database with the base power values respecting the file name format and the file data structure. This test will be performed in both laptop and Raspberry computers, with different parameters and time configurations (Table 4.2).

For the laptop tests we will use a frequency switch time of 50 ms., since this time assures that a file is created for each frequency hop performed by our system. A total time of five minutes has been assigned for each laptop test.

For the Raspberry tests we will use a frequency switch time of 500 ms., since this time assures that a file is created for each frequency hop performed by our system. A total time of fifteen minutes has been assigned for each Raspberry test.

The setup for this test consist of the laptop (Figure 4.1) or Raspberry (Figure 4.2) running the script and *htop*, with the HackRF One connected to it via a USB 2.0 cable. All tests will have the same setup, since the only difference among them will be the parameters configured in the script.

For the 10 Msps tests we expect the creation of 600 files, since the RF spectrum up to 6 GHz will be divided into chunks of 10 MHz. For the 20 Msps tests, we expect the creation of 300 files, since the 6 GHz will be divide into chunks of 20 MHz. For the 1024 FFT size tests we expect that files contain 1024 frequencies with its respective power and a correct frequency separation, and for 2048 FFT size tests we expect that files contain 2048 power values with its respective frequency separation.

In Table 4.3 we establish the expected results in terms of files, its structure and the expected spectrum sweep time and number. Expected spectrum sweep time refers to the time it takes to scan all frequencies between 1 MHz and 6 GHz, and total spectrum sweeps refers to the number of times a complete spectrum scan is performed during the test.

Table 4.2.Time configuration for base scan script tests

COMPUTER	FREQUENCY SWITCH TIME	TOTAL TEST TIME
Laptop	50 ms	5 minutes
Raspberry	500 ms	15 minutes

Figure 4.1: Laptop base scan script test setup

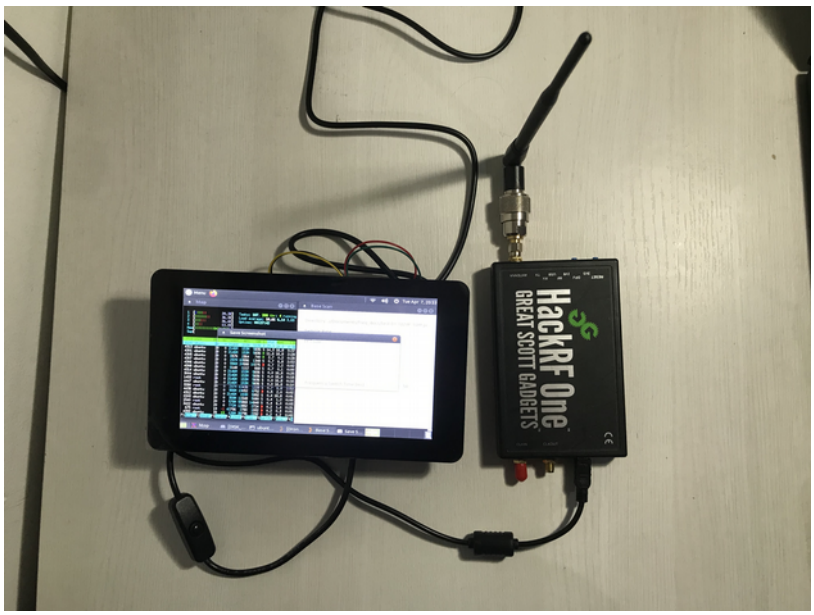


Figure 4.2: Raspberry base scan script setup

Table 4.3.Expected results for base scan tests

EXPECTED RESULT	10 Msps 1024 FFT	10 Msps 2048 FFT	20 Msps 1024 FFT	20 Msps 2048 FFT
NUMBER OF FILES	600	600	300	300
FILE NAME FORMAT	power_*{frequency}*MHz_10Msps_1024FFT	power_*{frequency}*MHz_10Msps_2048FFT	power_*{frequency}*MHz_20Msps_1024FFT	power_*{frequency}*MHz_20Msps_2048FFT
FREQUENCY VALUES	1024	2048	1024	2048
FREQUENCY SEPARATION	9.775 kHz	4.885 kHz	19.550 kHz	9.770 kHz
SPECTRUM SWEEP TIME (LAPTOP)	30 s.	30 s.	15 s.	15 s.
TOTAL SPECTRUM SWEEPS (LAPTOP)	10	10	20	20
SPECTRUM SWEEP TIME (RASPBERRY)	300 s. (5 min.)	300 s. (5 min.)	150 s. (2.5 min.)	150 s. (2.5 min.)
TOTAL SPECTRUM SWEEPS (RASPBERRY)	3	3	6	6

4.1.1. Laptop Test: 10Msps 1024FFT 50ms

For this configuration, the script creates 600 files that occupy 11.6 MB (Figure 4.5), accounting for a size of less than 20 kB each. With *htop* we see that when the script is running, the computer uses around 82% of all the CPU cores, 2.00 of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 30% for any of the cores (Figure 4.3).

When we explore the files, we can see that the naming format is correct, the number of power values is 1024, and the frequency separation is 9.775 kHz as expected (Figure 4.4). The file index is not unique for all documents, indicating that out system doesn't generate a constant number of vectors over time, making it possible that when scanning some frequencies we can receive several input vectors and for others only one vector. Even though, when exploring the files we see that the most frequent index is 15, which is greater than the number of expected total spectrum sweeps.

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 for 10 Msps-1024 FFT size in laptop

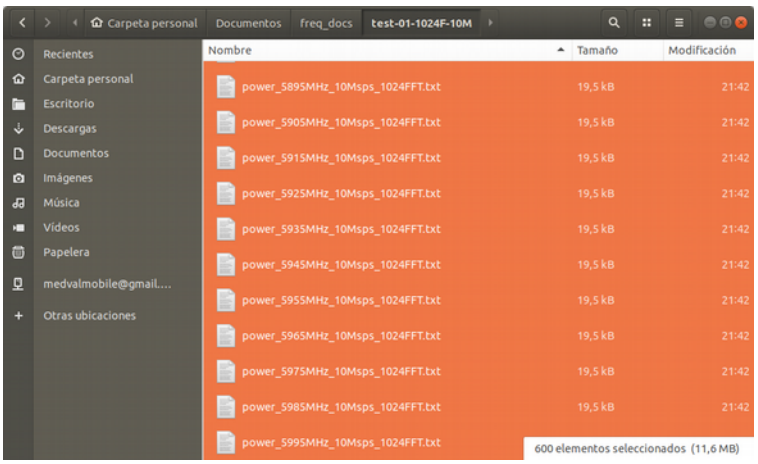


Figure 4.5: Files generated by base script for 10 Msps-1024 FFT size in laptop.

4.1.2. Laptop Test: 10Msps 2048FFT 50ms

For this configuration, the script creates 600 files that occupy 23.1 MB (Figure 4.46), accounting for a size of less than 40 kB each. With *htop* we see that when the script is running, the computer uses around 180% of all the CPU cores, 2.06 of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 55% for any of the cores (Figure 4.6).

When we explore the files, we can see that the naming format is correct, the number of power values is 2048, and the frequency separation is 4.885 kHz as expected (Figure 4.7). The most common file index is 15, which is greater than the the number of expected total spectrum sweeps.

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 for 10 Msps-2048 FFT size in laptop

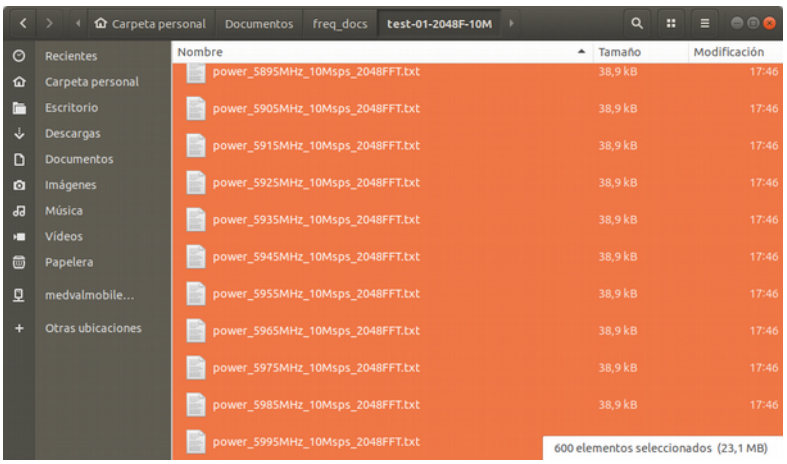


Figure 4.8: Files generated by base script for 10 Msps-2048 FFT size in laptop.

4.1.3. Laptop Test: 20Msps 1024FFT 50ms

For this configuration, the script creates 300 files that occupy 5.8 MB (Figure 4.11), accounting for a size of less than 20 kB each. With *htop* we see that when the script is running, the computer uses around 161% of all the CPU cores, 2.09 of 7.6 GB of RAM memory, and the CPU usage peak does not exceed 55% for any of the cores (Figure 4.9).

When we explore the files, we can see that the naming format is correct, the number of power values is 1024, and the frequency separation is 19.550 kHz as expected (Figure 4.10). The most common file index is 35, which is greater than the the number of expected total spectrum sweeps.

Figure 4.9: Laptop base script parameters configuration for 20 Msps-1024 FFT size-50 ms frequency jump time. CPU consumption below.

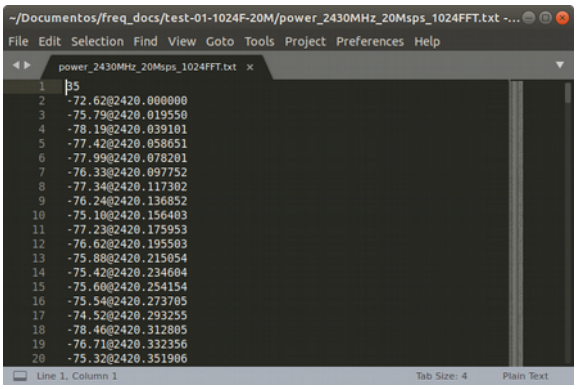


Figure 4.10: File structure created by base script for 20 Msps-1024 FFT size in laptop

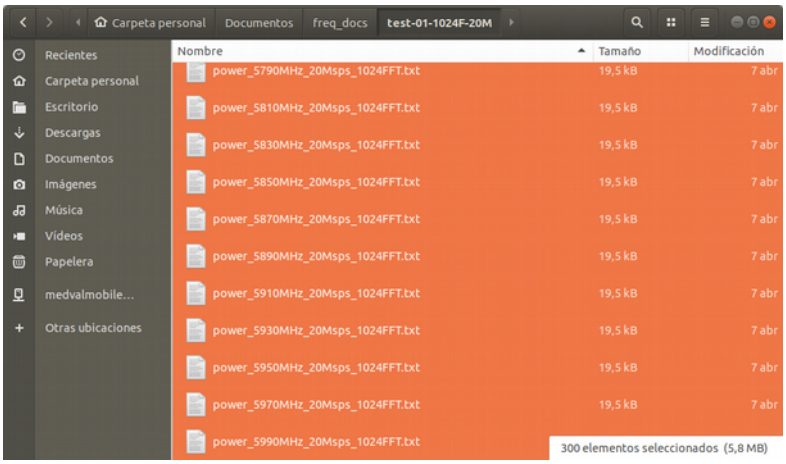


Figure 4.11: Files generated by base script for 20 Msps-1024 FFT size in laptop.

4.1.4. Laptop Test: 20Msps 2048FFT 50ms

For this configuration, the script creates 300 files that occupy 11.6 MB (Figure 4.14), accounting for a size of less than 40 kB each. With *htop* we see that when the script is running, the computer uses around 212% of all the CPU cores, 2.05 of 7.6 GHz of RAM memory, and the CPU usage peak does not exceed 65% for any of the cores (Figure 4.12).

When we explore the files, we can see that the naming format is correct, the number of power values is 2048, and the frequency separation is 9.770 kHz as expected (Figure 4.13). The most common file index is 26, which is greater than the the number of expected total spectrum sweeps.

Figure 4.12: Laptop base script parameters configuration for 20 Msps-2048 FFT size-50 ms frequency jump time. CPU consumption below.

Figure 4.13: File structure created by base script for 20 Msps-2048 FFT size in laptop

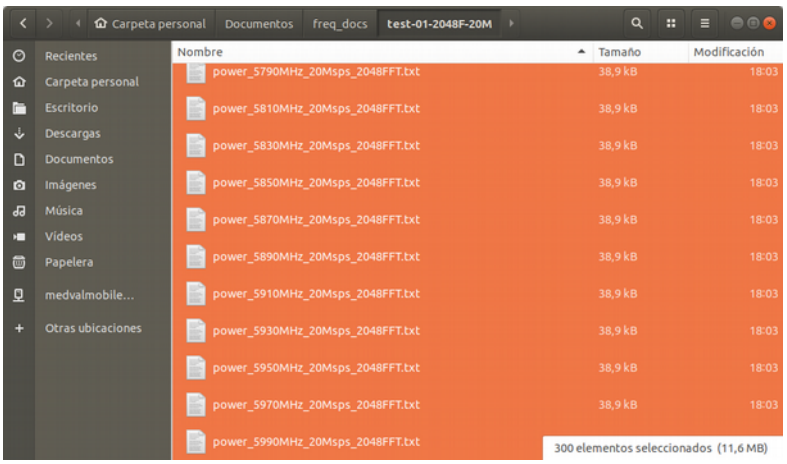


Figure 4.14: Files generated by base script for 20 Msps-2048 FFT size in laptop.

4.1.5. Base Scan Script Test Results Analysis for Laptop

The base scan script is in charge of creating a file database with averaged power values for all the spectrum from 1 MHz to 6 GHz. In the tests we have used four different parameters combinations, and the obtained results are summarized in the Table 4.4.

The number of files created in each test, correspond to the expected values. Also, the name format, the number of power values and the frequency separation matches our expected values for all tests.

In tests with a sample rate of 10 Msps, 600 files are created correctly, and for a sample rate of 20 Msps, 300 files are correctly generated. For tests with a 1024 FFT size, the files have 1024 frequency values as expected, and for tests with a 2048 FFT size, the files correctly reflect 2048 frequency values.

The frequency separation is correct for each configuration and is calculated by dividing the sample rate by the FFT size minus one, because we are working with zero indexed values. 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.

Each test configuration generates different file sizes and total disk space occupied by all files. Files with a 1024 FFT size have a maximum size of 19.5 kB, independently of the sample rate. Files with a 2048 FFT size have a maximum size of 38.9 kB, independently of the sample rate. The size of files with 2048 FFT size is the double of the files with 1024 FFT because they store the double of information.

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 power values. 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 power values. This accounts for a ratio of 4 MB per GHz. We can see that coincidentally the two remaining configurations 10 Msps-1024FFT and 20Msps-2048FFT have the same total file size value 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 represents the number of input vectors received to calculate the power values, and can be used to validate if the expected number of spectrum sweeps is accomplished. For the 10 Msps tests, we expect 10 spectrum scans, but we see that the files have an index of 15. As we explained earlier during the tests, our system can deliver more than one vector during a given frequency scan, justifying that we have received more input vectors than we expected. For the 20 Msps test, we have a 35 input vectors for 1024 FFT size and 26 for 2048 FFT size, which are greater than the 20 full spectrum scans expected.

Table 4.4.Base scan script test results in laptop

BASE SCAN SCRIPT TEST RESULTS - LAPTOP				
VALUE	**10 Msps 1024 FFT**	**10 Msps 2048 FFT**	**20 Msps 1024 FFT**	**20 Msps 2048 FFT**
NUMBER OF FILES	600	600	300	300
TOTAL FILES SIZE	11.6 MB	23.1 MB	5.8 MB	11.6 MB
MAX. FILE SIZE	19.5 kB	38.9 kB	19.5 kB	38.9 kB
TOTAL FILE SIZE RATIO	2 MB / GHz	4 MB / GHz	1 MB / GHz	2 MB / GHz
FILE NAME FORMAT	power_5995MHz_10Msps_1024FFT	power_5995MHz_10Msps_2048FFT	power_5990MHz_20Msps_1024FFT	power_5990MHz_20Msps_2048FFT
FILE INDEX	15	15	35	26
FREQUENCY VALUES	1024	2048	1024	2048
FREQUENCY SEPARATION	9.775 kHz	4.885 kHz	19.550 kHz	9.770 kHz
% CPU USED	82%	180%	161%	212%
RAM MEMORY USED (GB)	2.0	2.06	2.09	2.05

We monitored the performance of the computer while executing the tests, and even though we obtained only snapshots in a given time, this helped us identify the computation power required by each parameter configuration. It is expected that tests with 2048 FFT size, will require more computation power, since they make the double of operations than in 1024 FTT size. This assumption is proved in the results obtained, where the 10Msps-2048FFT configuration uses 180% of the CPU cores while the 10Msps-1024FFT configuration uses 80%. The same is observed in the other test where the 20Msps-2048FFT configuration uses 212% of the CPU cores against the 10Msps-1024FFT configuration 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.

With respect to the usage of RAM memory, we appreciate it doesn't vary greatly among the different tests, 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 biggest number of input vectors in five minutes with 35. It also has a better CPU performance compared to configurations with 2048 FFT sizes.

4.1.6. Raspberry Test: 10Msps 1024FFT 500ms

This script's test results are similar to the obtained in the laptop with the same parameters configuration: 600 files are created with a total size of 11.6 MB and each file's maximum size is under 20 kB. The script is run altogether with the CPU monitoring tool *htop*, but unlike what we saw in the laptop test, the CPU usage percentage varies rapidly every second, however we obtain a snapshot value of 174% and 677 MB of 1 GB of RAM memory used (Figure 4.15).

The script generates correctly the files with its corresponding name format, power values, and the frequency separation corresponds to its configuration (Figure 4.16). In this test we have seen that files have very different index values, and it wasn't possible to find a common number among all the files.

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

power_2425MHz_10Msps_1024FFT.txt

```
β6
-76.47@2420.000000
-78.22@2420.009775
-80.81@2420.019550
-81.44@2420.029326
-82.77@2420.039101
-80.42@2420.048876
-82.62@2420.058651
-81.37@2420.068426
-82.14@2420.078201
-80.06@2420.087977
-79.60@2420.097752
-81.45@2420.107527
-81.54@2420.117302
```

Figure 4.16: File structure created by base script for 10 Msps-1024 FFT size in Raspberry

4.1.7. Raspberry Test: 10Msps 2048FFT 500ms

This script's test results are similar to the obtained in the laptop under the same parameters: 600 files are created with a total size of 23.1 MB and each file's maximum size is under 40 kB. The *htop* snapshot shows a value of 163% of CPU usage and 674 MB of 1 GB of RAM memory occupied (Figure 4.17).

The script generates correctly the files with its corresponding name format, power values, and the frequency separation corresponds to its configuration (Figure 4.18). Just like in the previous test, we can't find a common file index for all the files.

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

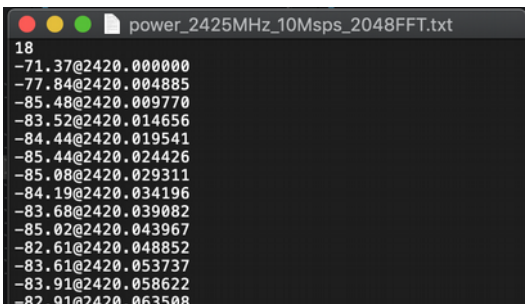


Figure 4.18: File structure created by base script for 10 Msps-2048 FFT size in Raspberry

4.1.8. Raspberry Test: 20Msps 1024FFT 500ms

This script's test results are similar to the obtained in the laptop with the same parameters configuration: 300 files are created with a total size of 5.8 MB and each file's maximum size is under 20 kB. The *htop* snapshot shows a value of 241% of CPU usage and 666 MB of 1 GB of RAM memory occupied (Figure 4.19).

The script generates correctly the files with its corresponding name format, power values, and the frequency separation corresponds to its configuration (Figure 4.20). In this test a common file index was not found.

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

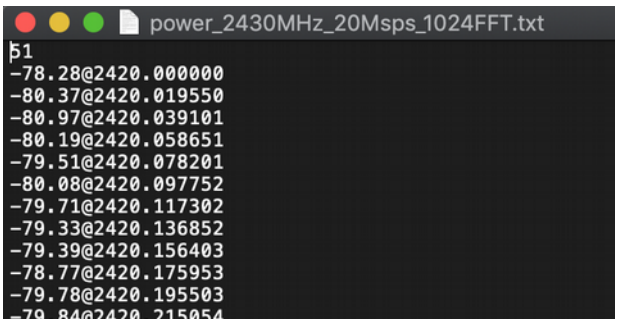


Figure 4.20: File structure created by base script for 20 Msps-10244 FFT size in Raspberry

4.1.9. Raspberry Test: 20Msps 2048FFT 500ms

This script's test results are similar to the obtained in the laptop under the same parameters: 300 files are created with a total size of 11.6 MB and each file's maximum size is under 40 kB. The *htop* snapshot shows a value of 222% of CPU usage and 673 MB of 1 GB of RAM memory occupied (Figure 4.21).

The script generates correctly the files with its corresponding name format, power values, and the frequency separation corresponds to its configuration (Figure 4.22). In this test a common file index was not found.

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

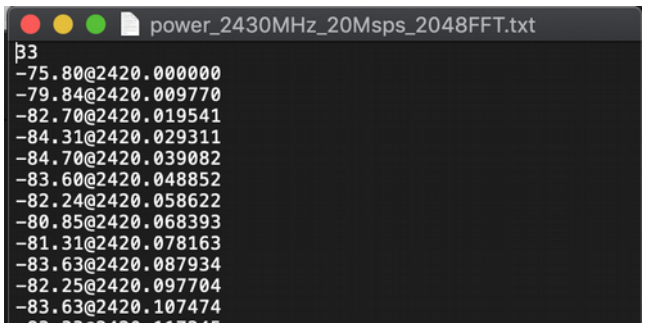


Figure 4.22: File structure created by base script for 20 Msps-2048 FFT size in Raspberry

4.1.10. Base Scan Script Test Results Analysis for Raspberry

Initially the tests for the base scan script run in Raspberry were using 50 ms. as the frequency switch time, but we could see 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 with 50 ms. as the frequency switch time. So we opted to increase this time from 50 ms. to 100 ms., then to 200 ms. and finally to 500 ms. as the time in which we saw that the files were created for all the frequencies during one total spectrum scan. Logically, since with this new frequency jump time a spectrum sweep would take five minutes, we had to increase the total time for this test up to 15 minutes.

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 we can see in Table 4.5.

While performing the test we saw that the Raspberry suffers from glitches reflected in the responsiveness of the user interface, sometimes freezing it for a few seconds. We have experienced with the Raspberry that when executing other processes like opening the internet browser or opening a text editor, usually it takes some significant time and often leaves the user interface unresponsive too. 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 CPU usage percentage varied greatly with each measure, making it an unusable parameter to evaluate the performance of the tests configurations.

The file index in these tests vary from file to file in each test, and we couldn't find a common value in all files, which makes this value an unusable parameter to verify the number of spectrum sweeps performed by the system during the test time.

The most important result we obtain from these tests is that the base scan script can indeed be executed in the Raspberry, and that the base powers database is created successfully so that it can be used for the spectrum and band scan script tests.

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.

Table 4.5.Base scan script test results in Raspberry

BASE SCAN SCRIPT TEST RESULTS - RASPBERRY				
VALUE	**10 Msps 1024 FFT**	**10 Msps 2048 FFT**	**20 Msps 1024 FFT**	**20 Msps 2048 FFT**
NUMBER OF FILES	600	600	300	300
TOTAL FILES SIZE	11.6 MB	23.1 MB	5.8 MB	11.6 MB
MAX. FILE SIZE	19.5 kB	38.9 kB	19.5 kB	38.9 kB
TOTAL FILE SIZE RATIO	2 MB / GHz	4 MB / GHz	1 MB / GHz	2 MB / GHz
FILE NAME FORMAT	power_5995MHz_10Msps_1024FFT	power_5995MHz_10Msps_2048FFT	power_5990MHz_20Msps_1024FFT	power_5990MHz_20Msps_2048FFT
FILE INDEX	36	18	51	33
FREQUENCY VALUES	1024	2048	1024	2048
FREQUENCY SEPARATION	9.775 kHz	4.885 kHz	19.550 kHz	9.770 kHz
% CPU USED	174%	163%	241%	222%
RAM MEMORY USED (MB)	677	674	666	673

4.1.11. Base Scan Script result comparison for laptop and Raspberry

We have successfully performed all tests in both laptop and Raspberry obtaining the expected results.

It was necessary to establish different test and frequency switch times for both computers, due to the different computation power of each hardware. For the laptop a frequency hop time of 50 ms. was enough for all the frequencies to be analyzed. But for Raspberry we had to set a time of 500 ms. for the system to analyze all frequencies correctly.

This different frequency switch time implies that the data received during a frequency scan in Raspberry is greater than the data received in the same frequency scan in the laptop. This means that the base values from the Raspberry tests have had more input vectors for the average, making it a more accurate value than the base power values from the laptop tests, where the averages are obtained using less input vectors making it prone to reflect peaky values.

To verify the resulting data obtained from the tests in both computers, we generated a graphic to compare the obtained averaged power values at a specific band, in this case the 433 MHz band (Figure 4.23). 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 software tool, because the data of the Raspberry and the laptop tests are found in different directories.

In the graphic we can see that results are similar in general, but we see smoother values in Raspberry and peaky values in laptop results, reflecting the expected behavior. It is also important to note that the tests done in the laptop were done in different days than those done in the Raspberry, which can lead to have different spectrum behavior in both results.

Depending on the user requirements and the CPU power, multiple combinations of frequency switch time and total scan time can be used. Smaller frequency switch time means less values in each frequency scan, but gives us a lesser total sweep time, making it possible to detect intermittent signals in the spectrum. Instead when using a bigger frequency switch time, we can get more data in a frequency scan and hence have a smoother average value, but it increases the total sweep time making it less probable to detect intermittent signals in the spectrum.

With respect to the generated files and its size, we have demonstrated that the structure of our file accomplishes its objective of having an almost constant size, that doesn't vary greatly in time. In the tests made in the Raspberry with a total time of 15 minutes we accomplish the same file size values than with the test done in the laptop with a total time of five minutes. Due to how the data is stored in the files, we assure that the file size won't vary greatly if this script is used in scans that can last for hours.

An interesting value obtained from the tests is the total file size ratio, which indicates the disk space needed to store the data per each GHz of spectrum analyzed. We use the total file size for each configuration and divide it by the 6 GHz that we scan, giving us a

MB / GHz ratio. The most efficient configuration gives us a ratio of 1 MB per each GHz scanned.

Regarding the monitoring of the CPU usage, we get a stable and confident value from the snapshots obtained in the laptop tests, but in the Raspberry test we observed an unstable behavior of this value, which changed greatly at every measure. This led us to discard the CPU usage values obtained in the Raspberry and discourage the usage of the *htop* tool while running the tests in this computer. In the laptop we see consequent results, where the CPU usage in configurations of 2048 FFT size is greater than those with 1024 FFT, since the double of operations are performed with the input data.

Finally we can conclude that the base scan script can be executed with success in a laptop or with a Raspberry, and that the most efficient parameter configuration is the 20 Msps and 1024 FFT size, which provides us the lower total file size ratio, a better CPU usage result than the configurations with 2048 FFT size, and reflected a greater file index than the other configurations. The minimum recommended time configuration for laptop would be 50 ms. for frequency switch time and 10 minutes for total scan time. For Raspberry a 500 ms. frequency switch time and 20 minutes of total scan time.

Figure 4.23: Graphic comparison of base scan script resulting power values in 433 MHz for laptop and Raspberry

4.2. Spectrum Scan Script Tests

These test's purpose is to verify that this script compares real time values with the base power values and creates the corresponding file database respecting the file name format and the file data structure. This test will be performed in both laptop and Raspberry computers, with different parameters and time configurations (Table 4.6).

For the laptop tests we will use a frequency switch time of 250 ms., since this time assures that the comparison file is created for each frequency hop performed by our system. A total time of five minutes has been assigned for each laptop test.

For the Raspberry tests we will use a frequency switch time of 1000 ms., since this time assures that a file is created for each frequency hop performed by our system. A total time of fifteen minutes has been assigned for each Raspberry test.

For all tests we will use the *power_comparator* in Fixed Mode with a value of 10 dB, that will set the threshold to 10 dB above the base power value for each frequency.

The setup for this test consist of the laptop (Figure 4.24) or Raspberry (Figure 4.25) running the script with the HackRF One connected to it via a USB 2.0 cable. The *htop* tool will only be used for laptop tests. All tests will have the same setup, since the only difference among them will be the parameters configured in the script.

Just like in the base scan script test, for the 10 Msps tests we expect the creation of 600 files, and for the 20 Msps tests, we expect the creation of 300 files. For the 1024 FFT size tests we expect that files contain 1024 frequencies with its respective comparison values and a correct frequency separation, and for 2048 FFT size tests we expect that files contain 2048 comparison values with its respective frequency separation.

In Table 4.7 we establish the expected results in terms of files, its structure and the expected spectrum sweep time and number. Expected spectrum sweep time refers to the time it takes to scan all frequencies tested.

Even though this script allows to modify the frequencies that are scanned, we will perform the test with the full spectrum available from 1 MHz to 6 GHz.

When executing the scripts, we will use in parallel the garage and the toy car remote controllers to generate signals in 27 MHz and 433 MHz respectively, with the intention to verify that our system can detect them. The remote controllers will be at a distance of up to 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.

Table 4.6.Parameter configuration for spectrum scan script tests

COMPUTER	FREQUENCY SWITCH TIME	TOTAL TEST TIME	OPERATION MODE AND VALUE
Laptop	250 ms	5 minutes	Fixed Mode – 10 dB
Raspberry	1000 ms	15 minutes	Fixed Mode – 10 dB

Table 4.7.Expected results for spectrum scan tests

EXPECTED RESULT	10 Msps 1024 FFT	10 Msps 2048 FFT	20 Msps 1024 FFT	20 Msps 2048 FFT
NUMBER OF FILES	600	600	300	300
FILE NAME FORMAT	compare_*{frequency}*MHz_10Msps_1024FFT	compare_*{frequency}*MHz_10Msps_2048FFT	compare_*{frequency}*MHz_20Msps_1024FFT	compare_*{frequency}*MHz_20Msps_2048FFT
FREQUENCY VALUES	1024	2048	1024	2048
FREQUENCY SEPARATION	9.775 kHz	4.885 kHz	19.550 kHz	9.770 kHz
SPECTRUM SWEEP TIME (LAPTOP)	150 s. (2.5 min.)	150 s. (2.5 min.)	75 s. (1.25 min.)	75 s. (1.25 min.)
TOTAL SPECTRUM SWEEPS (LAPTOP)	2	2	4	4
SPECTRUM SWEEP TIME (RASPBERRY)	600 s. (10 min.)	600 s. (10 min.)	300 s. (5 min.)	300 s. (5 min.)
TOTAL SPECTRUM SWEEPS (RASPBERRY)	1.5	1.5	3	3

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 (Figure 4.27), accounting for a size of less than 40 kB each. With *htop* we see that when the script is running, the computer uses around 110% of all CPU cores, 1.69 of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 40% for any of the cores (Figure 4.26).

When we explore the files, we can see that the naming format is correct, the number of frequency values with their corresponding calculated differences and averages is 1024, and the frequency separation is 9.775 kHz as expected. The most common file index is 16.

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 (Figure 4.28)

compare_435MHz_10Msps_1024FFT (Figure 4.29)

For the first file we see that a peak was detected around 27.15 MHz, with a value 44 dB above the base. For the second file we see a peak detected around 433.92 MHz, with a value 55 dB above the base.

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.

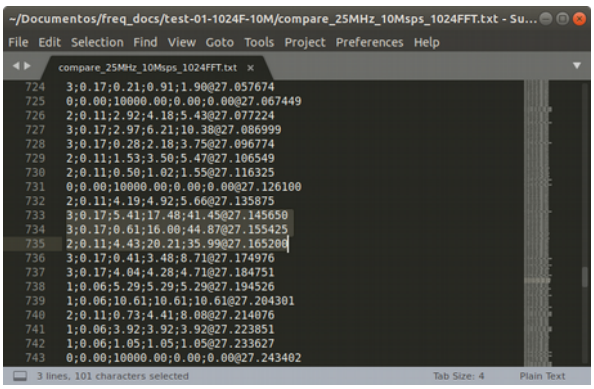


Figure 4.28: File data with values of a peak detected in 27 MHz with spectrum script for 10 Msps-1024 FFT size.

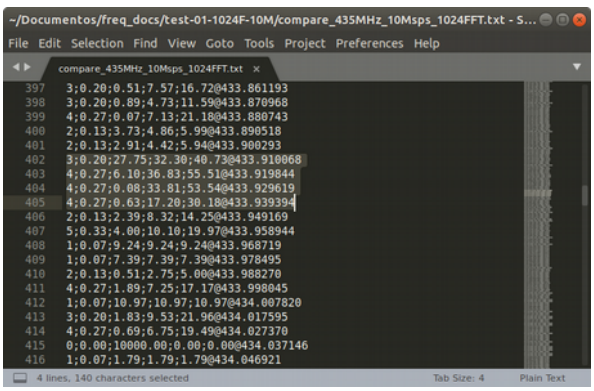


Figure 4.29: File data with values of a peak detected in 433 MHz with spectrum script for 10 Msps-1024 FFT size.

4.2.2. Laptop Test: 10Msps 2048FFT 250ms

For this configuration, the script creates 600 files that occupy 46 MB (Figure 4.31), accounting for a size of less than 80 kB each. With *htop* we see that when the script is running the computer uses around 132% of all CPU cores, 1.80 of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 40% for any of the cores (Figure 4.30).

When we explore the files, we can see that the naming format is correct, the number of frequency values with their corresponding calculated differences and averages is 2048, and the frequency separation is 4.885 kHz as expected. The most common file index is 16.

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 (Figure 4.32)

compare_435MHz_10Msps_2048FFT (Figure 4.33)

For the first file we see that a peak was detected around 27.22 MHz, with a value 32 dB above the base. For the second file we see a peak detected around 433.92 MHz, with a value 60 dB above the base.

Figure 4.30: Laptop spectrum script parameters configuration for 10 Msps-2048 FFT size. CPU consumption in the right.

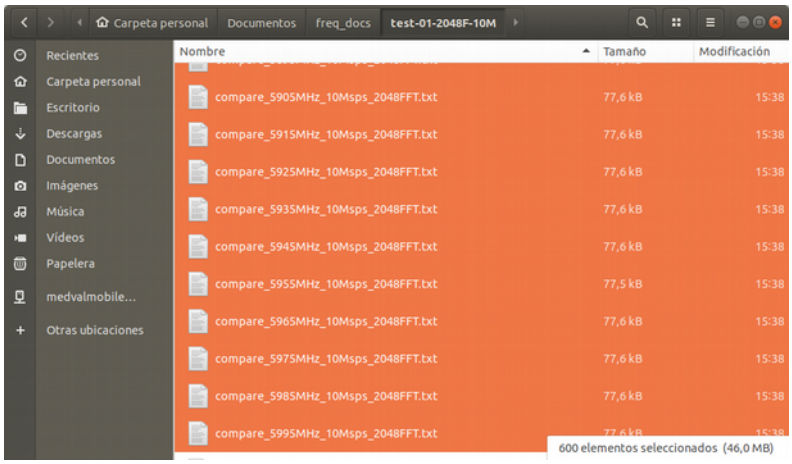


Figure 4.31: Files generated by spectrum script with its names and sizes for 10 Msps-2048 FFT size.

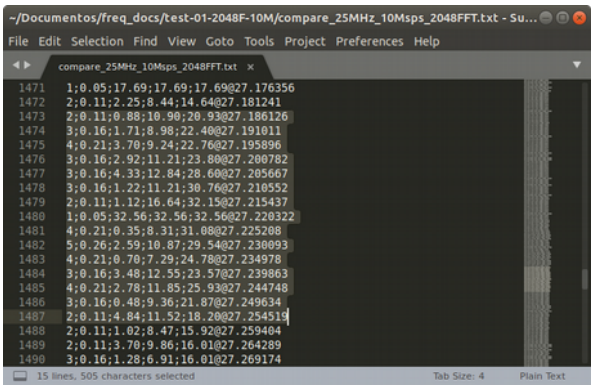


Figure 4.32: File data with values of a peak detected in 27 MHz with spectrum script for 10 Msps-2048 FFT size.

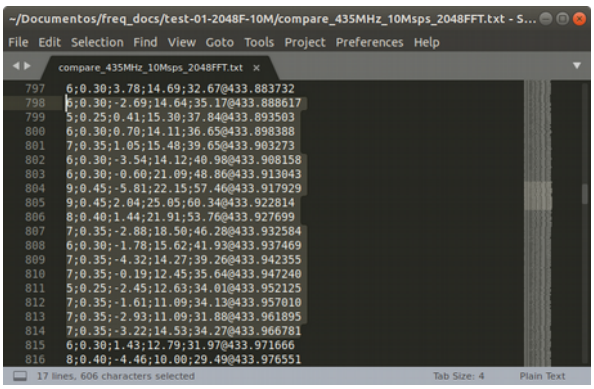


Figure 4.33: File data with values of a peak detected in 433 MHz with spectrum script for 10 Msps-2048 FFT size.

4.2.3. Laptop Test: 20Msps 1024FFT 250ms

For this configuration, the script creates 300 files that occupy 11.4 MB (Figure 4.35), accounting for a size of less than 40 kB each. With *htop* we see that when the script is running the computer uses around 123% of all CPU cores, 1.81 of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 35% for any of the cores (Figure 4.34).

When we explore the files, we can see that the naming format is correct, the number of frequency values with their corresponding calculated differences and averages is 1024, and the frequency separation is 19.550 kHz as expected. The most common file index is 31.

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 (Figure 4.36)

compare_430MHz_20Msps_1024FFT (Figure 4.37)

For the first file we see that a peak was detected around 27.17 MHz, with a value 31 dB above the base. For the second file we see a peak detected around 433.93 MHz, with a value 50 dB above the base.

Figure 4.34: Laptop spectrum script parameters configuration for 20 Msps-1024 FFT size. CPU consumption in the right.

Figure 4.35: Files generated by spectrum script with its names and sizes for 20 Msps-1024 FFT size.

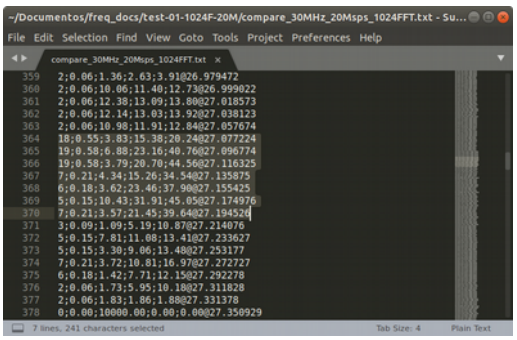


Figure 4.36: File data with values of a peak detected in 27 MHz with spectrum script for 20 Msps-2048 FFT size.

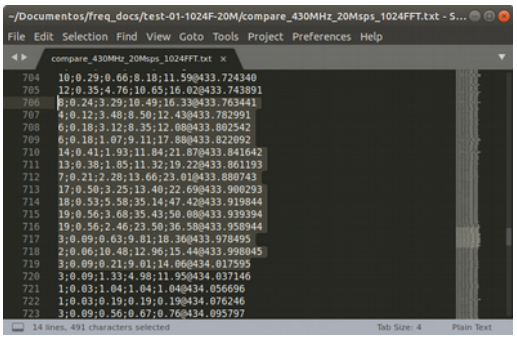


Figure 4.37: File data with values of a peak detected in 433 MHz with spectrum script for 20 Msps-2048 FFT size.

4.2.4. Laptop Test: 20Msps 2048FFT 250ms

For this configuration, the script creates 300 files that occupy 22.9 MB (Figure 4.39), accounting for a size of less than 80 kB each. With *htop* we see that when the script is running the computer uses around 215% of all CPU cores, 2.05 of 7.69 GB of RAM memory, and the CPU usage peak does not exceed 65% for any of the cores (Figure 4.38).

When we explore the files, we can see that the naming format is correct, the number of frequency values with their corresponding calculated differences and averages is 1024,and the frequency separation is 19.550 kHz as expected. The most common file index is 31.

When we explore the files, we can see that the naming format is correct, the number of frequency values with their corresponding calculated differences and averages is 2048, and the frequency separation is 9.770 kHz as expected. The most common file index is 30.

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 (Figure 4.40)

compare_430MHz_20Msps_2048FFT (Figure 4.41)

For the first file we see that a peak was detected around 27.09 MHz, with a value 26 dB above the base. For the second file we see a peak detected around 433.92 MHz, with a value 53 dB above the base.

Figure 4.38: Laptop spectrum script parameters configuration for 20 Msps-2048 FFT size. CPU consumption in the right.

Figure 4.39: Files generated by spectrum script with its names and sizes for 20 Msps-2048 FFT size.

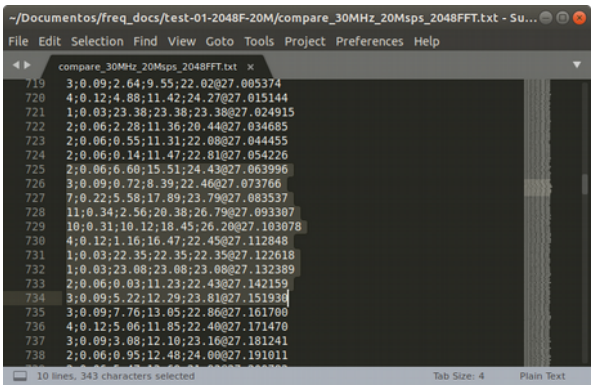


Figure 4.40: File data with values of a peak detected in 27 MHz with spectrum script for 20 Msps-2048 FFT size.

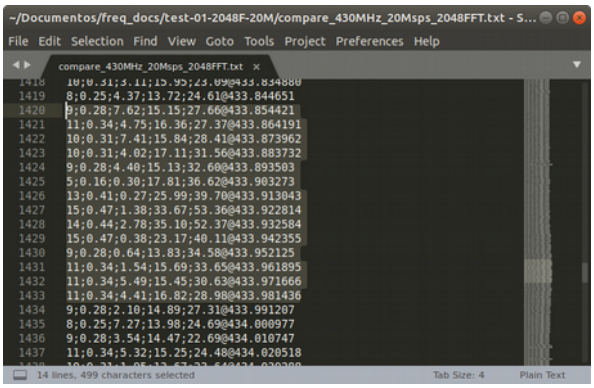


Figure 4.41: File data with values of a peak detected in 27 MHz with spectrum script for 20 Msps-2048 FFT size.

4.2.5. Spectrum Scan Script Test Results Analysis for Laptop

The spectrum scan script is in charge of comparing the real time power values against the averaged base power values, and create a file database with the obtained results. In the tests we have used four different parameters combinations, and the obtained results are summarized in the Table 4.8. All tests were done using the Fixed Value mode to establish the power threshold 10 dB above the base.

The number of files created in each test, correspond to the expected values. Also, the name format, the number of frequency values and the frequency separation matches our expected values for all tests.

In tests with a sample rate of 10 Msps, 600 files are created correctly, and for a sample rate of 20 Msps, 300 files are correctly generated. For tests with a 1024 FFT size, the files have 1024 frequency values as expected, and for tests with a 2048 FFT size, the files correctly reflect 2048 frequency values.

The comparison data for each frequency follows correctly the proposed format, and we could verify that the obtained values make sense, where the minimum difference is always smaller than the maximum difference and its average is between those values. The value above threshold is always smaller than the file index, and the average of values above threshold is always smaller than 100%. We will use Figure 4.41 to make a quick check that the data obtained in these files have sense. In the line 1427 we can see at frequency 433.922814 MHz, 15 values have been above the threshold, accounting for a 47% of all the values analyzed. The minimum value that exceeded the base was 1.38 dB above it, the maximum was 53.36 dB higher than the base, and all values that exceeded the threshold average 33.67 dB higher than the base. Here we find that the minimum is smaller than the 10 dB threshold, and it can be explained by the fact that the script default operation mode is *Percentage* and is set to 1%, and to set the operation mode to Fixed Value and the value to 10 dB, we must do it manually, and several seconds pass until we can configure it correctly.

The frequency separation is correct for each configuration and is calculated by dividing the sample rate by the FFT size minus one, because we are working with zero indexed values. 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.

Each test configuration generates different file sizes and total disk space occupied by all files. Files with a 1024 FFT size have a maximum size of 38.8 kB, independently of the sample rate. Files with a 2048 FFT size have a maximum size of 77.6 kB, independently of the sample rate. The size of files with 2048 FFT size is the double of the files with 1024 FFT because they store the double of information.

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 12 MB, where all the spectrum information can be contained in 300 files with 1024 comparison values. With this configuration we obtain a ratio of under 2 MB of information per each GHz analyzed. The biggest total file size is 46 MB, for the 10 Msps and 2048 FFT size configuration, which consists of 600 files containing 2048 power values. This accounts for a ratio of

under 8 MB per GHz. We can see that coincidentally the two remaining configurations 10 Msps-1024FFT and 20Msps-2048FFT have approximately the same total file size, 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 under 4 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.

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% of the CPU cores while 10Msps-1024FFT uses 110%. The same is observed in the other tests where the 20Msps-2048FFT configuration uses 215% of the CPU cores against the 10Msps-1024FFT configuration 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 in part by the fact that in this script we work with up to three files simultaneously for each frequency, make more operations and store data in memory before 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 files for each configuration at frequencies 27 MHz and 433 MHz. All files reflect that peaks have been detected in the corresponding frequencies, with values of maximum difference from the base that go from 26 dB to 60 dB.

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 gave us the biggest number of input vectors in five minutes with 31. It is also has a better CPU and RAM memory performance compared to configurations with 2048 FFT sizes.

Table 4.8.Spectrum script test results in laptop

SPECTRUM SCAN SCRIPT TEST RESULTS - LAPTOP				
VALUE	**10 Msps 1024 FFT**	**10 Msps 2048 FFT**	**20 Msps 1024 FFT**	**20 Msps 2048 FFT**
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
TOTAL FILE SIZE RATIO	4 MB / GHz	8 MB / GHz	2 MB / GHz	4 MB / GHz
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	16	14	31	30
FREQUENCY VALUES	1024	2048	1024	2048
FREQUENCY SEPARATION	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 / MAX. DIFF.	27.15 MHz 44 dB	27.22 MHz 32 dB	27.17 MHz 31 dB	27.09 MHz 26 dB
433 MHZ PEAK FREQUENCY / MAX. DIFF.	433.92 MHz 55 dB	433.92 MHz 60 dB	433.93 MHz 50 dB	433.92 MHz 53 dB

4.2.6. Raspberry Test: 10Msps 1024FFT 1000ms

This script's test results are similar to the obtained in the laptop with the same parameters configuration: 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 name format, power values, and the frequency separation corresponds to its configuration. In this test we have seen that files have very different index values, and it wasn't possible to find a common number among all the files.

Figure 4.42: 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 with the same parameters configuration: 600 files are created with a total size of 46 MB and each file's maximum size is under 80 kB.

The script generates correctly the files with its corresponding name format, power values, and the frequency separation corresponds to its configuration. In this test we have seen that files have very different index values, and it wasn't possible to find a common number among all the files.

Figure 4.43: 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 with the same parameters configuration: 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 name format, power values, and the frequency separation corresponds to its configuration. In this test we have seen that files have very different index values, and it wasn't possible to find a common number among all the files.

Figure 4.44: 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 with the same parameters configuration: 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 name format, power values, and the frequency separation corresponds to its configuration. In this test we have seen that files have very different index values, and it wasn't possible to find a common number among all the files.

Figure 4.45: Raspberry spectrum script parameters configuration for 20 Msps-2048 FFT size.

4.2.10. Spectrum Scan Script Test Results Analysis for Raspberry

Initially the tests for the base scan script run in Raspberry were using 250 ms. as the frequency switch time, but we could see that due to the lower CPU power not all frequency files were generated. We increased the time to 1000 ms., as the time that we saw was needed by the system to create the files for all the frequencies during one total spectrum scan. Logically, since with this new frequency jump time a spectrum sweep would take five minutes, we had to increase the total time for this test up to 15 minutes.

For these tests we have decided to stop using the *htop* tool, so that all the computation power is dedicated to the execution of the script, trying to avoid glitches and blocks in the responsiveness of the Raspberry. This means that we won't have data about CPU and RAM memory usage.

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 we can see in Table 4.9. All tests were done using the Fixed Value mode to establish the power threshold 10 dB above the base.

The comparison data for each frequency follows correctly the proposed format, and the maximum, minimum and average values are consequent. For the Raspberry tests no screenshot of the files were added, since in the laptops tests they were provided and the file format is exactly the same.

To verify that the script indeed detects unusual signals generated during the tests, we have verified the files for each configuration at frequencies 27 MHz and 433 MHz. For the 27 MHz signal, only in one of the four tests we can see a peak detected with a value of 49 dB above the base. In two other tests, values of 11 dB and 15 dB above the base were found but they aren't big enough to indicate that our generated signal was detected. In the remaining test, no value above the threshold was detected in 27 MHz. For 433 MHz, we can see that all four tests have detected a peak with values of up to 33 dB above the base. Since each test lasted 15 minutes, it was more difficult that our system detect in all cases the signals that were generated manually.

The most important result we obtain from these tests is that the spectrum scan script can indeed be executed in the Raspberry, and that the comparison values database is created successfully helping in the detection of unusual RF signal peaks.

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.

Table 4.9.Spectrum scan script results in Raspberry

SPECTRUM SCAN SCRIPT TEST RESULTS - RASPBERRY				
VALUE	**10 Msps 1024 FFT**	**10 Msps 2048 FFT**	**20 Msps 1024 FFT**	**20 Msps 2048 FFT**
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	77.8 kB	38.9 kB	77.8 kB
TOTAL FILE SIZE RATIO	4 MB / GHz	8 MB / GHz	2 MB / GHz	4 MB / GHz
FILE NAMING FORMAT	compare_5995 MHz_10Msps_1024FFT	compare_5995 MHz_10Msps_2048FFT	compare_5990 MHz_20Msps_1024FFT	compare_5990 MHz_20Msps_2048FFT
FREQUENCY VALUES	1024	2048	1024	2048
FREQUENCY SEPARATION	9.775 kHz	4.885 kHz	19.550 kHz	9.770 kHz
27 MHZ PEAK FREQUENCY / MAX. DIFF.	No value above threshold detected	27.23 MHz 11 dB	27.19 MHz 15 dB	27.15 MHz 49 dB
433 MHZ PEAK FREQUENCY / MAX. DIFF.	433.90 MHz 32 dB	433.33 MHz 33 dB	433.95 MHz 27 dB	433.51 MHz 24 dB

4.2.11. Spectrum Scan Script result comparison for laptop and Raspberry

We have successfully performed all tests in both laptop and Raspberry obtaining the expected results.

It was necessary to establish different test and frequency switch times for both computers, due to the different computation power of each hardware. For the laptop a frequency hop time of 250 ms. was enough for all the frequencies to be analyzed. But for Raspberry we had to set a time of 1000 ms. for the system to analyze all frequencies correctly.

With the resulting data obtained from the tests in laptop and Raspberry we generate a graphic to compare the obtained values at the frequencies at which we generated manually a signal, 27 MHz (Figure 4.46) and 433 MHz (Figure 4.47). 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.

The graphic shows a peaky behavior for all tests, and it is justified because we are displaying the base power value where the threshold has not been passed, and the maximum difference from the base where the threshold has been exceeded.

For 27 MHz in laptop, all test could detect a peak, but only one could do so in Raspberry. For 433 MHz a peak was detected in all test, with greater power values in the laptop.

It is possible that for the tests done in Raspberry, the remote controller signal was generated while the scanner was analyzing other frequencies, so it couldn't be detected, and when the scanner was analyzing the signal's frequency, it wasn't being generated. This determines that it can be a more difficult task to detect unusual activity with this script in Raspberry, due to the greater frequency switch time.

With respect to the generated files and its size, we have demonstrated that the proposed file structure accomplishes its objective of having an almost constant size, that doesn't vary greatly in time. The files in the Raspberry tests have the same size than those generated in the laptop tests, even though in the first the total test time is 15 minutes and in the second it is five minutes. Due to how the data is stored in the files, we assure that the file size won't vary greatly if this script is used in scans that can last for hours. For these files we can see that the most efficient total file size ratio is 2 MB per each GHz.

Regarding the monitoring of the CPU usage, we get a stable and confident value from the snapshots obtained in the laptop tests, where we see consequent results, in which the CPU usage in configurations of 2048 FFT is greater than those with 1024 FFT, since the double of operations are performed with the input data.

Finally we can conclude that the spectrum scan script can be executed with success in a laptop or with a Raspberry, and that the most efficient parameter configuration is the 20 Msps and 1024 FFT size, which provides us the lower total file size ratio, a better CPU usage result than the configurations with 2048 FFT size. The minimum recommended time configuration for laptop would be 250 ms. for frequency switch time and 10 minutes for total scan time. For Raspberry a 1000 ms. frequency switch time and 20 minutes of total scan time.

Figure 4.46: Graphic comparison of spectrum scan script resulting power values in 27 MHz for laptop and Raspberry

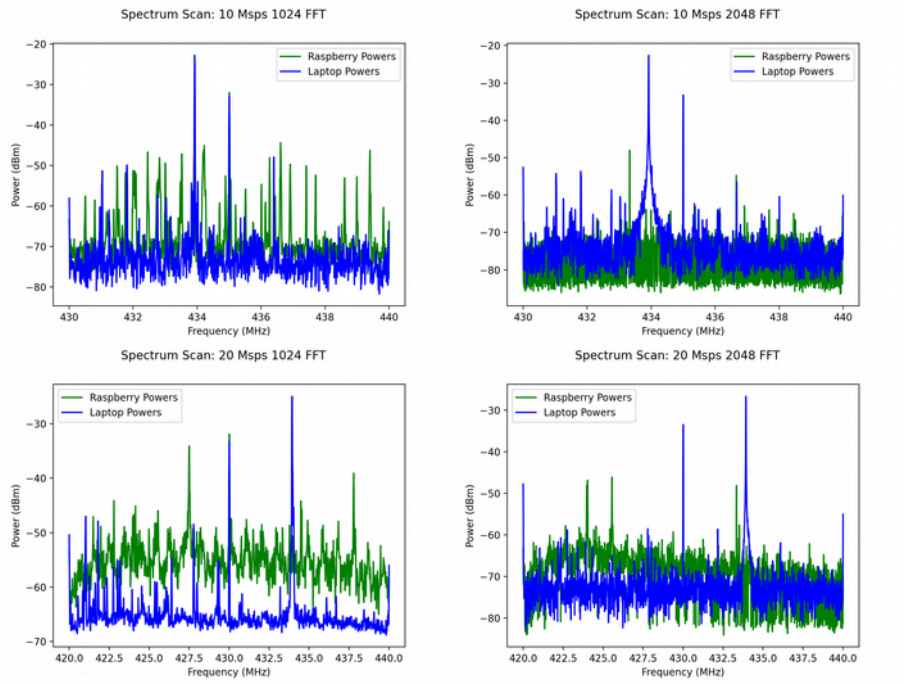


Figure 4.47: Graphic comparison of spectrum scan script resulting power values in 433 MHz for laptop and Raspberry

4.3. Band Scan Script Tests

These test's purpose is to verify that this script can help in the detection of RF signals in a specific band both by creating the comparisons file database and by visually identifying the signals that are going to be generated for this test.

For all tests we will use the *power_comparator* block in Fixed Mode with a value of 10 dB, that will set the threshold to 10 dB above the base power value for each frequency.

Only the 20 Msps and 1024 FFT size parameter configuration will be used, since it has been proved in the previous test that is the most efficient parameter combination. This indicates that the maximum bandwidth analyzed by this script is 20 MHz.

The tests will be performed both in laptop and in the Raspberry, and there are no time configurations in this script.

Four tests per computer will be performed, and each scanning a given band, where we will generate an RF signal. We will use the toy car remote controller to analyze in 27 MHz, the garage remote controller in 433 MHz and the drone for the 2.4 GHz band, in WiFi and RC modes.

To verify that our script can indeed detect the signals, we will first make a scan in the given bands with no RF signals generated, so that at the end of all the tests we can make comparison graphics between data with and without the remote controller activity.

The comparison graphics will be generated using a specifically created custom Python function that allows us to compare results from different tests.

The base values for these tests are the obtained in the previously performed base scan script test.

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

The setup for this test is the following: the laptop, the HackRF and the toy car with its remote controller (Figure 4.48). The remote controller will be activated during all the duration of the test.

Figure 4.48: Band scan script test setup for detection at 27 MHz in laptop

In our script we must manually set the center frequency to 30 MHz, so that in the frequency display we can observe the spectrum from 20 MHz up to 40 MHz (Figure 4.49).

When the remote controller is transmitting we can observe in the graphic a peak at 27 MHz. While this visual analysis is being held, our system is generating the corresponding file database with the comparison done against the base power values.

The graphic gives us the option to hold the maximum and minimum values, in case it is required. It is important to note that in this script no graphical comparison are performed, since they are 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 (Figure 4.50).

To verify that the data gathered during the test reflects the activity that was detected visually we generate a comparison graphic between the base power values, values of the band with no remote controller activity, and the data gathered during the test. In the generated graphic we can spot the previously identified peak at 27 MHz, and that there was no activity in this frequency before the test (Figure 4.51).

Figure 4.49: Band scan script graphic detection of an active signal in 27 MHz.

Figure 4.50: Band scan script file with detection of a peak at 27 MHz

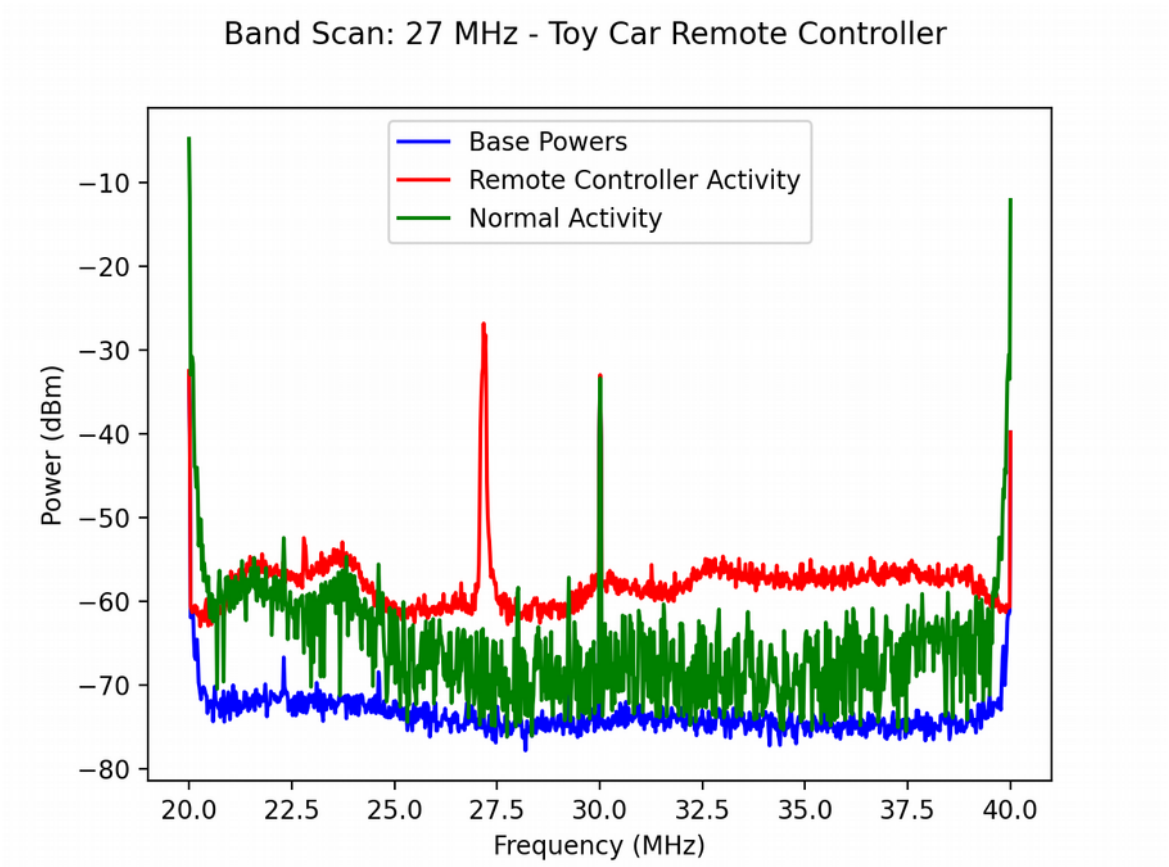


Figure 4.51: Graphic comparison of band scan script test results for 27 MHz in laptop

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* garage remote controller (Figure 4.52). The remote controller will be activated during all the duration of the test.

Figure 4.52: Band scan script test setup for detection at 433 MHz in laptop

Our script has configured 433 MHz as the default center frequency, but it can also be set manually with the slider controller or with the *preset band* selector. The frequency display shows the spectrum from 420 MHz up to 440 MHz (Figure 4.53).

When the remote controller is transmitting we can observe in the graphic a peak at 433 MHz. While this visual analysis is being held, our system is generating the corresponding file database with the comparison done against the base power values.

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 (Figure 4.54).

To verify that the data gathered during the test reflects the activity that was detected visually we generate a comparison graphic between the base power values, values of the band with no remote controller activity, and the data gathered during the test. In the generated graphic we can spot the previously identified peak at 433 MHz, and that there was no activity in this frequency before the test (Figure 4.55).

Figure 4.53: Band scan script graphic detection of an active signal in 433 MHz.

Figure 4.54: Band scan script file with detection of a peak at 433 MHz

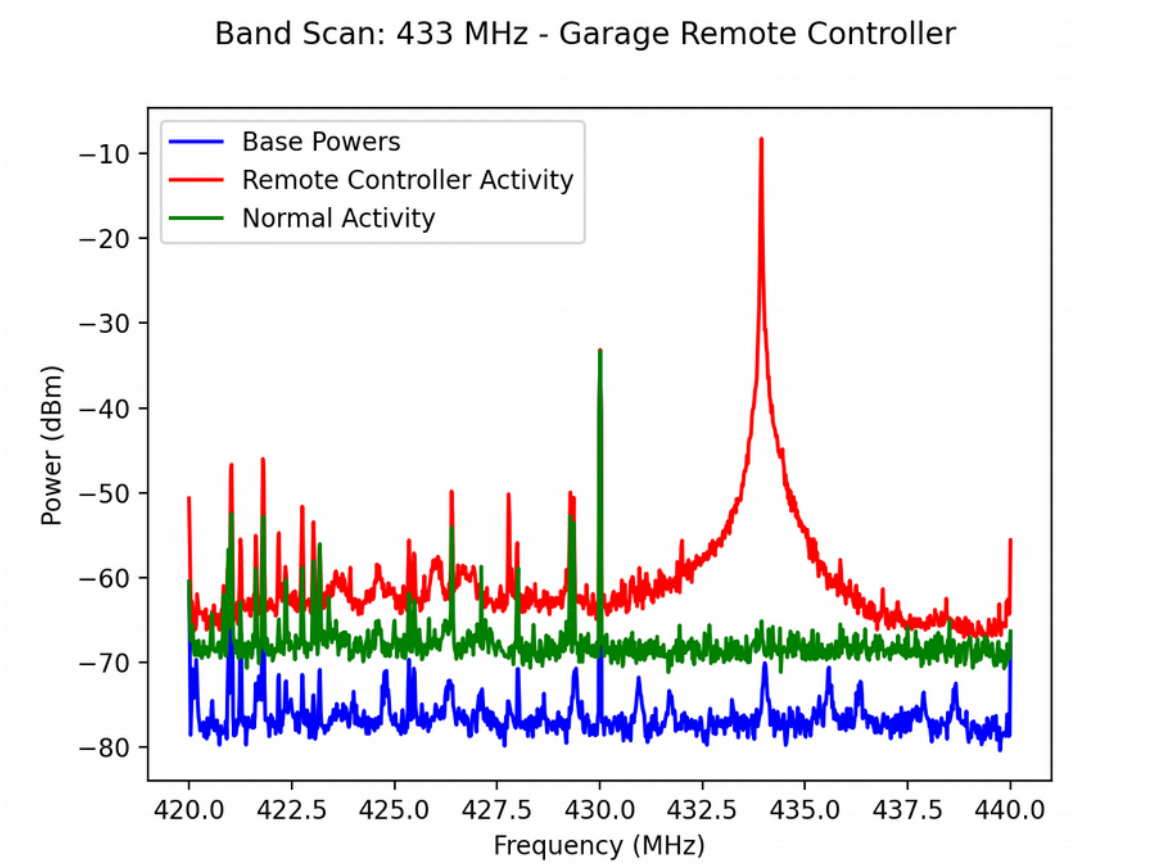


Figure 4.55: Graphic comparison of band scan script test results for 433 MHz in laptop

4.3.3. Laptop Test: Drone WiFi Mode (2.4 GHz)

The setup for both drone detection test is the following: the laptop, the HackRF, the DJI Mavic Pro drone, the remote controller and the Huawei Y7 smartphone with the DJI GO 4 app (Figure 4.56). In this test we will set the drone in the WiFi mode, connect the smartphone to the WiFi network generated by the drone, and use the mobile app to control the UAV.

Figure 4.56: Laptop band scan script test setup for drone (2.4 GHz)

For the 2.4 GHz band tests, the approach is completely different than the others tests. Since the toy car and garage 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 bandwidth signal in RC mode or normal WiFi signals in WiFi mode, and also the frequency at which it appears can vary all across the 2400-2480 MHz spectrum. Another difficulty is that our HackRF scans only up to 20 MHz, and we need to verify at least 80 MHz in search of drone RF activity.

Since the location for this test is an indoor location in an urban area, we also expect a constant RF activity in the 2.4 GHz band due to home WiFi networks and Bluetooth

mainly. Even though this could complicate our detection tasks, it can prove that our solution works in real life scenarios.

Once the connection between the smartphone and the drone is established, we start monitoring the 2.4 GHz band in chunks of 20 MHz. Since it is using WiFi communications, we see carriers raise and disappear just like in normal WiFi transmissions, which complicates the task of visually detecting the drone's RF footprint along the whole band.

From the DJI Go 4 app, we could determine that communications during the test were performed using the WiFi channel 7 (2431–2453 MHz) (Figure 4.58).

To explore the file database searching for peaks like we did in the previous tests, is not an option in the 2.4 GHz band since WiFi uses wide-band transmissions.

So in this case we will use an "off" approach, in which we will analyze the data after the test to detect unusual RF activity in the band, between the data gathered prior to the test and the data obtained while using the drone. In the graphic we can observe that the peaks data of both scans is mostly similar, except for the 2430-2440 MHz portion of the band, which difference could be attributed to the drone's communications since they were performed in channel 7. Even though for this test we know the frequencies at which transmission are performed, cannot reach to the conclusion that the RF activity spotted in that portion of the 2.4 GHz band correspond to drone's communications.

Figure 4.57: Band scan script configuration for 2.430 GHz.

Figure 4.58: Screenshot of the DJI Go 4 app indicating the WiFi channel used during the test. Current channel in blue (CH7).

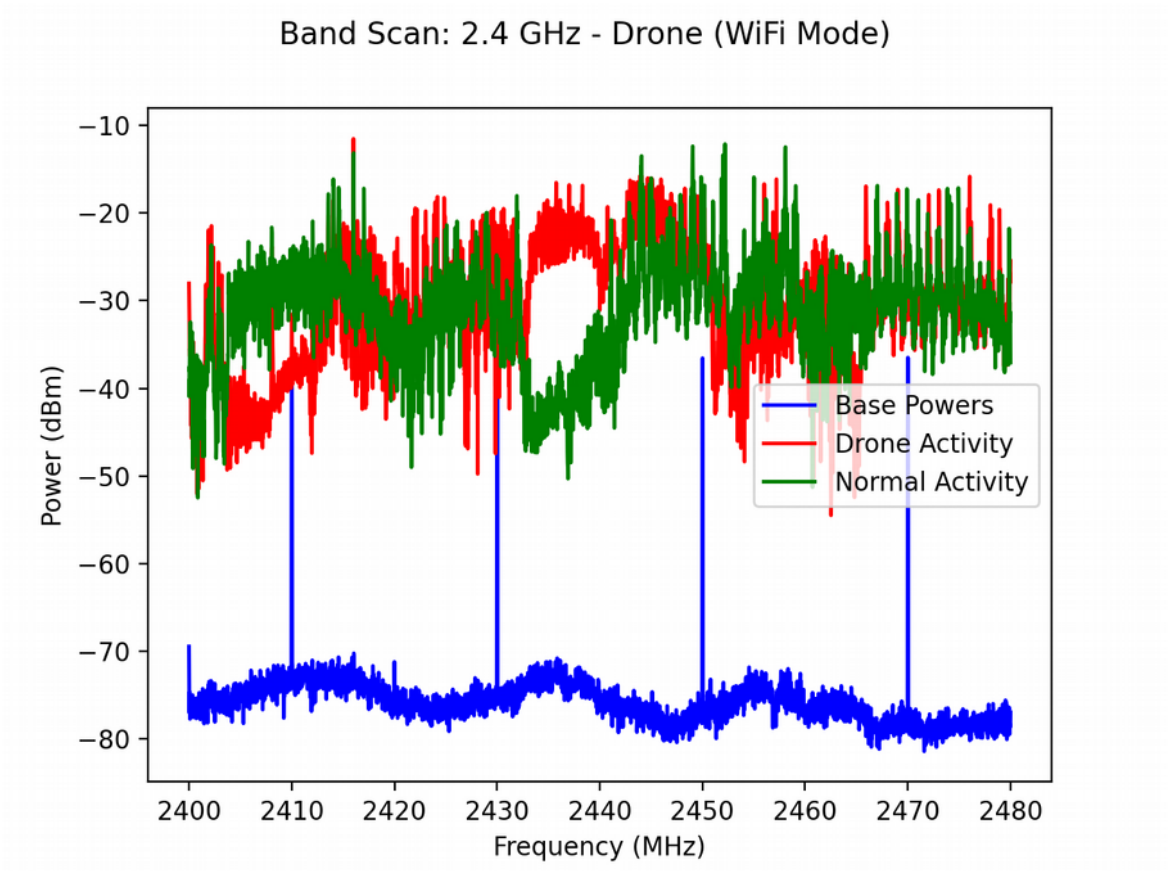


Figure 4.59: Graphic comparison of band scan script test results for 2.4 GHz in WiFi mode in laptop

4.3.4. Laptop Test: Drone RC Mode (2.4 GHz)

The setup for this test is the specified in Figure 4.56. In this test we will set the drone in the RC mode, connect the smartphone to the drone's remote controller, use the mobile app as a FPV screen and the remote controller to transmit and receive the data to and from the UAV.

Once the connection between the remote controller and the drone is established, we start monitoring the 2.4 GHz band in chunks of 20 MHz. In the 2460-2480 MHz part of the the band we observe a wide-band transmission that remains active all the time while the drone communications are being held (Figure 4.60). In the visual analysis we can identify that this signal has a bandwidth of approximately 18 MHz and its center frequency is around 2470 MHz. Contrary to what we saw in the WiFi mode test, this wide-band signal is permanent and has a completely different behavior than the WiFi transmissions. Once the drone's communications are dropped, the signal disappears from the spectrum graphic.

Figure 4.60: Band scan script graphic detection of an active signal in 2.4 GHz.

Using the FCC ID found in the drone's remote controller, we can obtain in the FCC's official website [34] the compliance reports where we can get the RF tests done to the drone's communications. In these reports we find a graphic of the 20 MHz High Channel operation mode, which has its center frequency at 2472.5 MHz and its bandwidth is 17.92 MHz. This official data confirms that the signal we have graphically detected corresponds to the drone communication in RC mode.

Figure 4.61: DJI Mavic Pro's remote controller FCC's RF test for 20 MHz high channel operation mode.

Similar to what we experimented in the previous test, detecting the drone activity by reading the file database seems like an unsuccessful method. We will find WiFi, Bluetooth and other RF activity performed in the 2.4 GHz, with maximum values that seem similar across neighboring frequencies. The only approach that we can use to identify a drone signal in this band is the visual one.

We also perform the database data analysis by generating a graphic which compares the values obtained prior to the test with the data generated while the drone was in use (Figure 4.62). In the graphic 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 upper 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 data with its respective graphic we can't reach to a final conclusion.

Figure 4.62: Laptop scan band script test results comparison for 2.4 GHz band

4.3.5. Band Scan Script Test Results Analysis for Laptop

The band scan script has the same file database creation functionality than the spectrum scan script, but limited to a bandwidth of up to 20 MHz, and with a real time frequency graphic with extra controls that allow the user to explore visually the spectrum searching for unusual RF activity. The results for these tests are summarized in the Table 4.10.

In the four tests, the comparison files were generated successfully respecting the central frequencies, sample rate and FFT size.

In three of the four tests we have been able to detect visually the RF activity corresponding to that generated by the remote controllers in 27 MHz, 433 MHz and 2.4 GHz in RC mode. By analyzing the data obtained during the tests and generating comparison graphics with it, we could identify the RF activity by identifying narrow-band peaks in the tests performed in 27 MHz and 433 MHz. Since the signals generated in the 2.4 GHz are wide-band this approach doesn't seem to be conclusive.

Since the tests have been performed in an urban environment, the data obtained in the 2.4 GHz band contains the RF activity of the WiFi networks and Bluetooth communications performed in the location. For scenarios in which the WiFi and

Bluetooth activity is negligible or nonexistent, such as in rural environments or specific laboratory settings, the peak data analysis can lead to better results than the obtained in these tests.

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.

Table 4.10.Band scan script test results in laptop

BAND SCAN SCRIPT TEST RESULTS - LAPTOP				
TEST	**VISUAL DETECTION**	**DATA DETECTION**	**PEAK FREQUENCY**	**MAX. DIFF. FROM BASE**
27 MHZ	YES	YES	27.17 MHz	46 dB
433 MHZ	YES	YES	433.93 MHz	66 dB
2.4 GHZ (WiFi MODE)	NO	NO		
2.4 GHZ (RC MODE)	YES	NO		

4.3.6. Raspberry: Toy Remote Controller (27 MHz)

The setup for this test is the following: the Raspberry, the HackRF and the toy car with its remote controller (Figure 4.65). The remote controller will be activated during all the duration of the test.

Figure 4.63: Raspberry band scan script test setup for remote controller (27 MHz)

In our script we must set the center frequency manually to 30 MHz with the slider, so that the frequency display shows the spectrum from 20 MHz up to 40 MHz (Figure 4.64).

When the remote controller is transmitting we can observe in the graphic a peak at 27 MHz. The comparison file is generated while we perform this test, comparing the real time powers with the base power values.

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 44 dB above the base, and with 81% of all the received power values in this frequency above the threshold.

In the comparison graphic with the base power values, power values obtained before and during the test, we can spot the peak at 27 MHz, and that there was no activity in this frequency before the test (Figure 4.65).

Figure 4.64: Band scan script graphic detection of an active signal in 27 MHz.

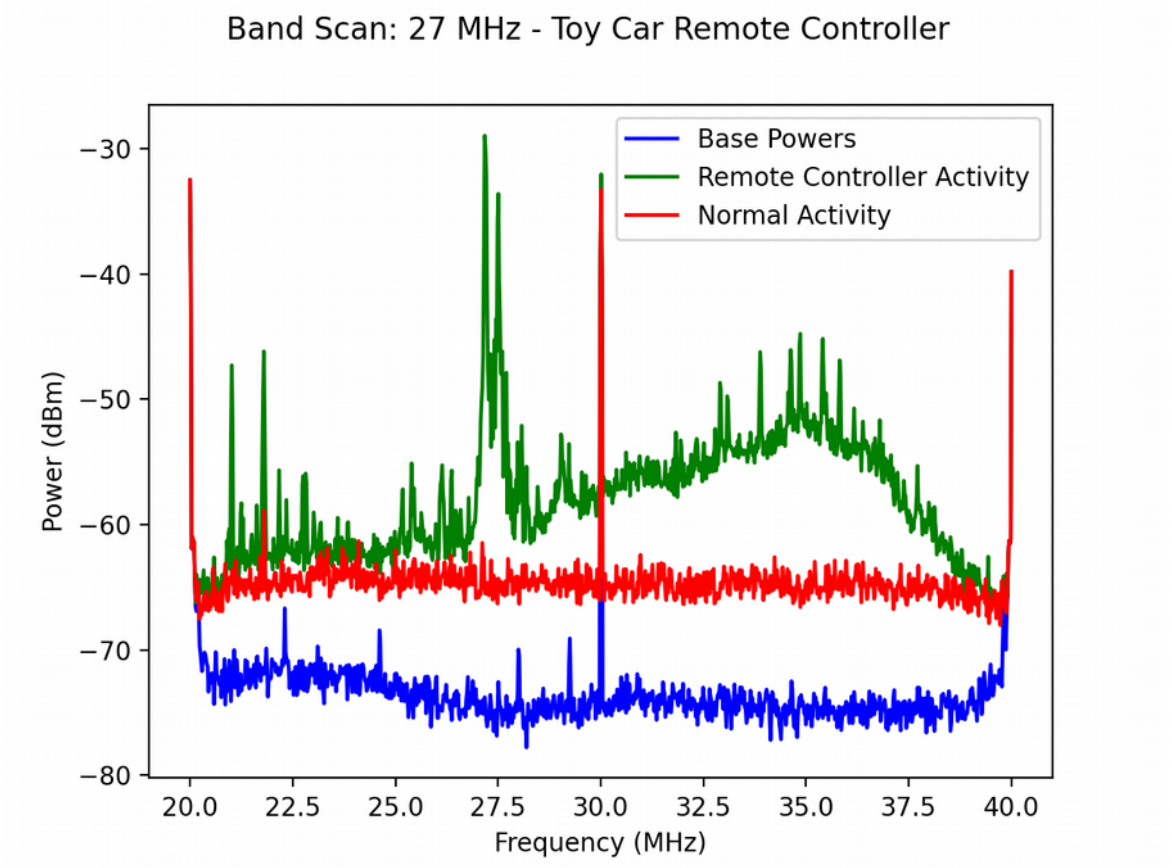


Figure 4.65: Graphic comparison of band scan script test results for 27 MHz in Raspberry

4.3.7. 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 4.66).

Figure 4.66: Raspberry band scan script test setup for remote controller (433 MHz)

When the remote controller is transmitting we can observe in the graphic a peak at 433 MHz (Figure 4.67). While this visual analysis is being held, our system is generating the corresponding file database with the comparison done against the base power values.

Exploring the file created during this test, we can see that the peak of the generated signal is located at 433.91 MHz, with a power value 55 dB above the base. The other data on this frequency shows that 54% of all the received power values in this frequency were above the threshold.

To verify that the data gathered during the test reflects the activity that was detected visually we generate a comparison graphic between the base power values, values of the band with no remote controller activity, and the data gathered during the test. In the generated graphic we can spot the previously identified peak at 433 MHz, and that there was no activity in this frequency before the test (Figure 4.68).

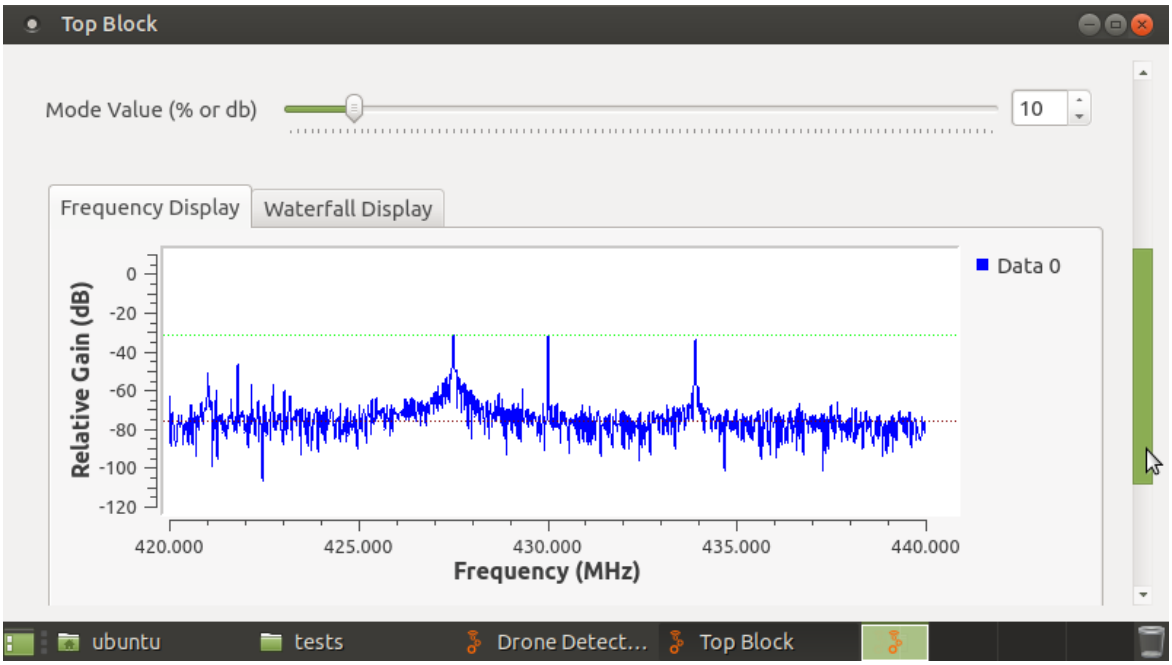


Figure 4.67: Band scan script graphic detection of an active signal in 433 MHz.

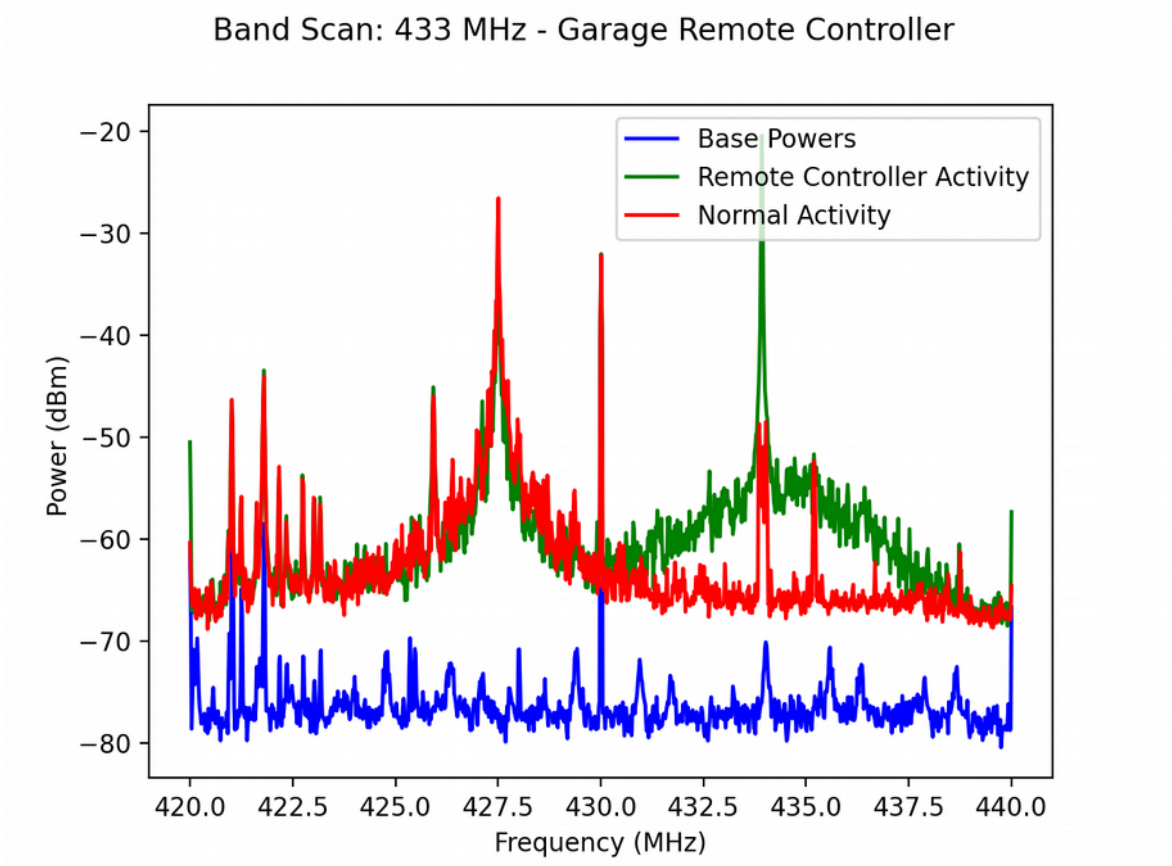


Figure 4.68: Graphic comparison of band scan script test results for 433 MHz in Raspberry

4.3.8. Raspberry: Drone RC Mode (2.4 GHz)

The setup for this test is the following: the Raspberry, the HackRF, the DJI Mavic Pro drone in RC mode, the remote controller and the Huawei Y7 smartphone with the DJI Go 4 app (Figure 4.69).

Figure 4.69: Raspberry band scan script test setup for drone (2.4 GHz)

Once the connection between the remote controller and the drone is established, we start monitoring the 2.4 GHz band in chunks of 20 MHz. We detect a wide-band signal that is in the 2460-2480 MHz band,just like in the laptop test (Figure 4.70). This signal occupies around 18 MHz and its center is around 2470 MHz. The behavior is the same as in the laptop test, remaining active while the communication lasts between the drone and the remote controller and disappearing once the drone's communications are 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 4.61.

To verify that the data gathered during the test reflects the activity that was detected visually we generate a comparison graphic between the base power values, values of the band with no drone activity, and the data gathered during the test. As we saw in the laptop test's graphic, we can observe an almost constant maximum value at the end of the 2.4 GHz band which correspond to the drone's RF activity. However only with the analysis of this graphic we cannot reach to a final conclusion.

Figure 4.70: Band scan script graphic detection of an active signal in 2.4 GHz.

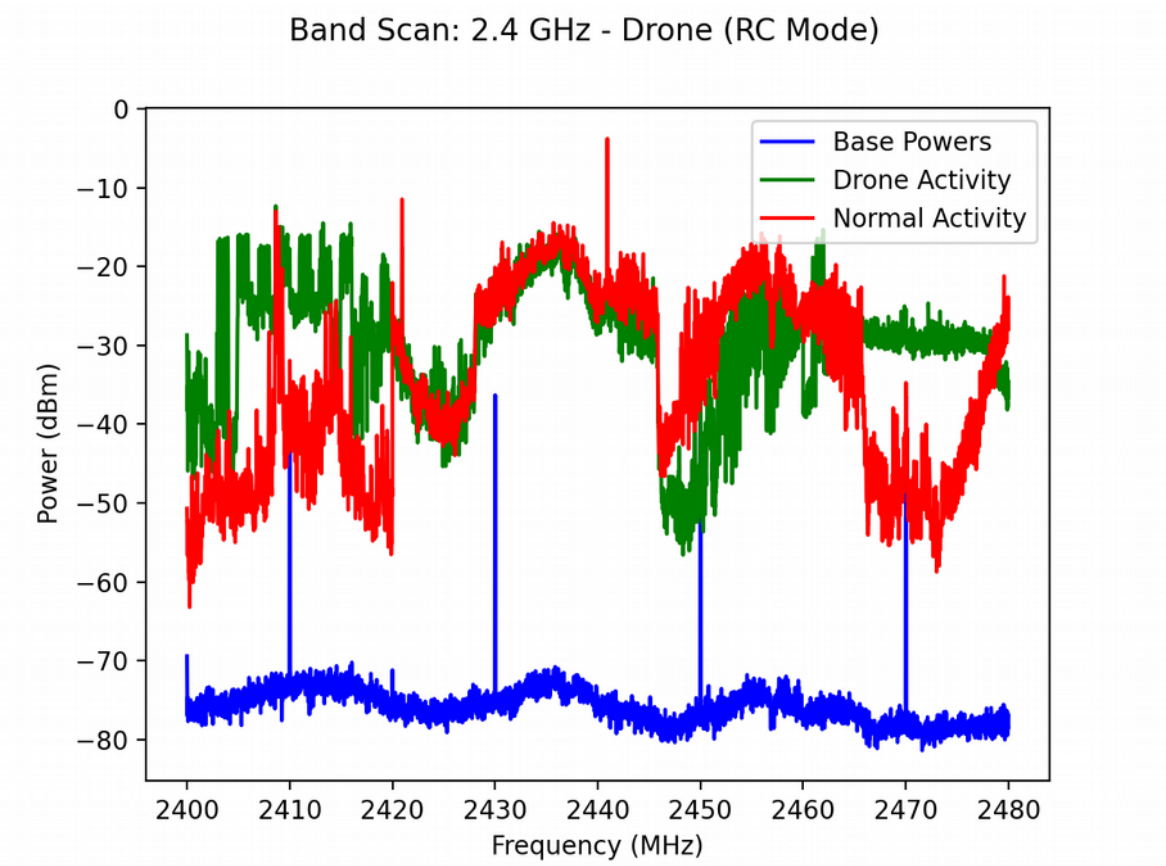


Figure 4.71: Graphic comparison of band scan script test results for 2.4 GHz in Raspberry

4.3.9. Raspberry Test Analysis for Band Scan Script

For the Raspberry computer we have chosen to execute the tests for 27 MHz, 433 MHz and 2.4 GHz with the drone in RC mode only. Since in the laptop test, no drone activity could be identified visually nor by data analysis, there was no need to execute the same test in the Raspberry. The results of these tests are summarized in the Table 4.11.

Similar to the laptop results, in the three test performed we have been able to detect visually the RF activity corresponding to that generated by the remote controllers in 27 MHz, 433 MHz and 2.4 GHz. By analyzing the obtained data we could only identify the peaky behaviors of the narrow-band signals in 27 MHz and 433 MHz, but for the 2.4 GHz drone wide-band signals this approach is not conclusive.

The script has proven to be successful in the detection of unusual RF activity, and has been successfully executed in the Raspberry with no setbacks.

Table 4.11.Band scan scrip test results in Raspberry

BAND SCAN SCRIPT TEST RESULTS - RASPBERRY				
TEST	**VISUAL DETECTION**	**DATA DETECTION**	**PEAK FREQUENCY**	**MAX. DIFF. FROM BASE**
27 MHZ	YES	YES	27.17 MHz	44 dB
433 MHZ	YES	YES	433.91 MHz	55 dB
2.4 GHZ (RC MODE)	YES	NO		

4.3.10. Band Scan Script result comparison for laptop and Raspberry

We have successfully performed the band scan script tests in both laptop and Raspberry successfully detecting in a visual form the generated signals by the remote controllers in 27 MHz, 433 MHz, and 2.4 GHz with the drone in RC mode.

Our approach of obtaining the peaks by calculating the difference of the real time power with respect to the base power seems to work without problem when we are working with narrow-band signals in bands that usually don't have a constant RF activity, such as the 27 MHz and 433 MHz band. With wide-band signals like the used in the 2.4 GHz our approach is not the optimal, and other techniques should be used like ML algorithms or neural networks to determine if a received signal may correspond to drone's transmission.

We have failed to identify the RF footprint of the drone while being used in the WiFi mode, since it uses the WiFi transmission system and in a place with high WiFi activity like in the urban environment of these tests, transmissions from other WiFi networks have a similar behavior to the drone's communications making it difficult to differentiate between them. On the other hand, when the drone is configured in RC mode, it leaves an unmistakable RF footprint in the 2.4 GHz band with a different behavior than the WiFi signals, making it easier to detect visually.

The use of the graphic tool to visualize the spectrum behavior in real time has proven to be an effective instrument to help detect unusual RF activity in a given band. And together with the file database they complement to be a useful method for RF signals detection.

It is recommended that this script is used with the 20 Msps configuration since it provides the greater bandwidth available making it possible to detect wide-band signals specially in the 2.4 GHz band. The FFT size configuration can be any, but it is recommended to use the 1024 size configuration since it was proven in previous tests that it consumes less CPU resources than the 2048 size configurations. It is important to note that previous to the use of this script, a base scan must be performed with the configuration chosen by the user.

4.4. Jammer Script Tests

These test's purpose is to verify that this script can transmit a signal noise in given frequency bands with the intention to interfere or block communications.

In order to test this script we will use the following setup: the Raspberry connected to a HackRF One executing the jammer script, and the laptop with another HackRF One executing the band scan script. The Raspberry will be used as the transmission system and the laptop will be used as an spectrum analyzer. We will use the 20 Msps and 1024 FFT configuration parameters. Previously to the test we will use the base scan 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, so we can compare the different power levels involved in each scenario. The graphics will be generated using a created custom Python function that allows us to compare results from different tests.

Figure 4.72: Jammer script setup test with Raspberry as the noise generator and the laptop as the spectrum analyzer.

4.4.1. Jammer test at 27 MHz

First we will execute the jammer script with 30 MHz as the center frequency, generating a noisy signal of 20 MHz of bandwidth. The objective would be to block the communication between the toy remote control and the toy car. Our attemp 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 noise signal with a power enough of being capable to interfere in the communications (Figure 4.73).

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 (Figure 4.74). 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 (Table 3.1), 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.

Figure 4.73: Jammer signal at 30 MHz

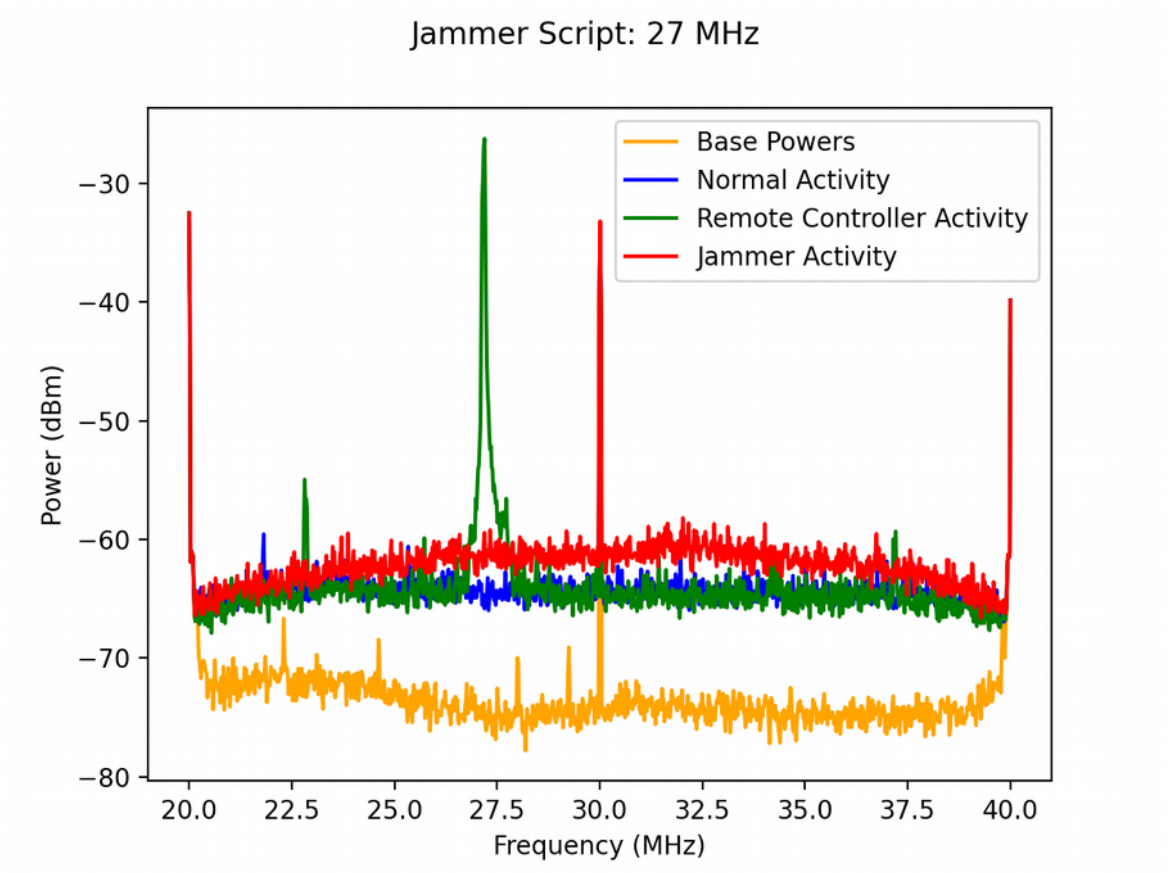


Figure 4.74: Comparison graphic of power values with jammer activity for 27 MHz

4.4.2. Jammer test at 433 MHz

For this test 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 on the graphics only.

This time we can see graphically in the laptop, that the HackRF generates a significant noise signal that could interfere in the communications (Figure 4.75). We see that it has a power around 20 dB greater than the power under normal circumstances.

With the comparison graphic, we can see a considerable difference in the power levels with and without the jammer signal (Figure 4.76). Even though, the noise power isn't greater than the peak of the signal generated by the remote controller.

Figure 4.75: Jammer signal at 430 MHz

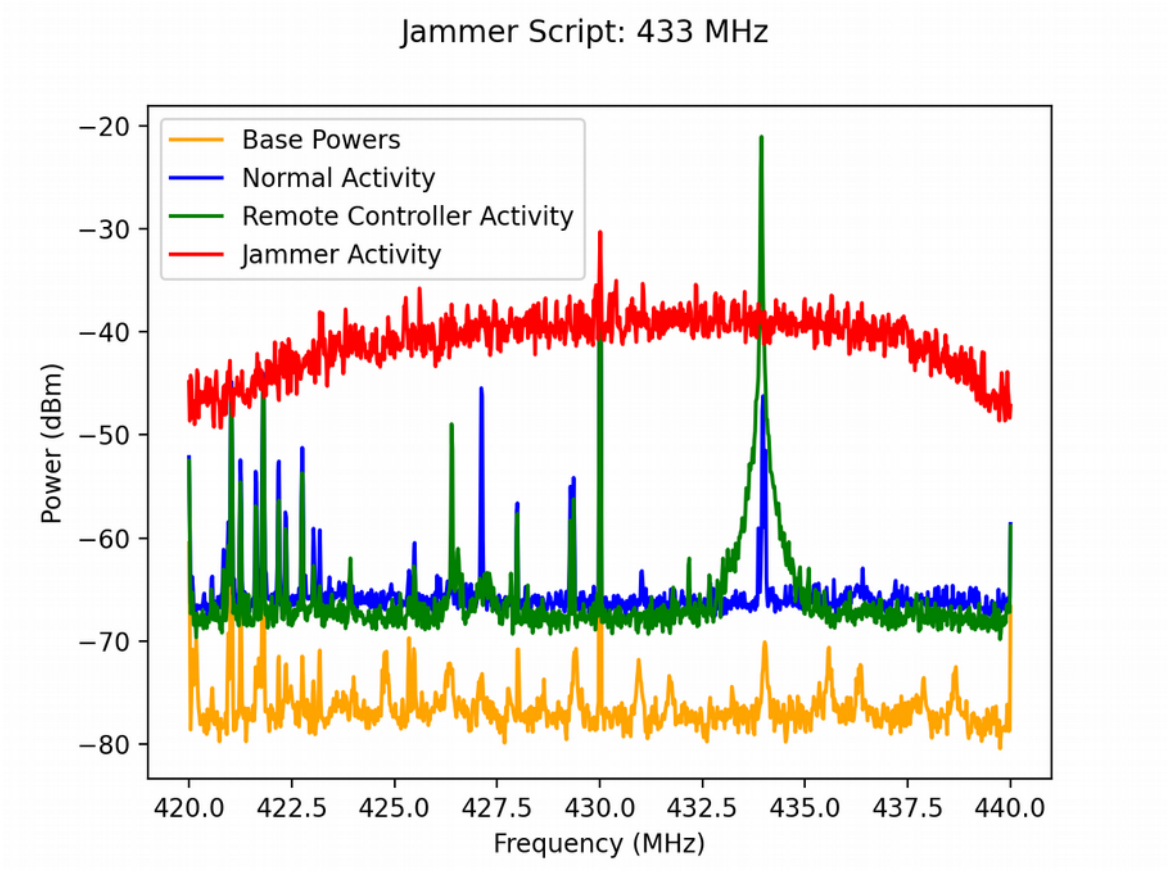


Figure 4.76: Comparison graphic of power values with jammer activity for 433 MHz

4.4.3. Jammer test at 1.5 GHz (GNSS)

For this test we will execute the jammer with different center frequencies at which we will generate a noisy signal of 20 MHz bandwidth. For this test we will use the drone in the RC mode connected with the remote controller and the smartphone with the DJI Go 4 app.

As we saw in Table 1.1, there are several frequencies in which we find the GNSS transmissions, Our system has preset band options for some GNSS frequencies, but it can also be manually set with the help of the slider component.

We use the 1570 MHz and 1590 MHz as center frequencies to jam the L1, G1 and E1 bands transmissions (Figure 4.77). To block communications for the L2, G2 and E6 bands we set the frequency to 1230 MHz, 1270 MHz and 1290 MHz. And to interfere the communications in the L5, G3, and E5a/b bands we set the center frequency to 1150 MHz, 1170 MHz and 1190 MHz.

Even though the DJI Mavic Pro drone doesn't have a built-in GNSS module, it obtains the navigation data from the mobile smartphone. And in the mobile app it indicates the number of, in this case GPS, satellites to which the device is connected. In Figure 4.78 we can see that the smartphone's position is calculated with 5 satellites, in the top center of the image. When we activate the jammer in 1570 MHz, which includes the L1 band, the GPS connection is lost, and the app let us know by disabling the satellite icon (Figure 4.79).

With the comparison graphic, we can see a considerable difference in the power levels with and without the jammer signal (Figure 4.80) in all frequencies that were set, accomplishing the interference of GNSS communications.

Figure 4.77: Jammer script configuration at 1.570 GHz

Figure 4.78: Screenshot of the DJI Go 4 app with GPS connection

Figure 4.79: Screenshot of the DJI Go 4 app with no GPS connection

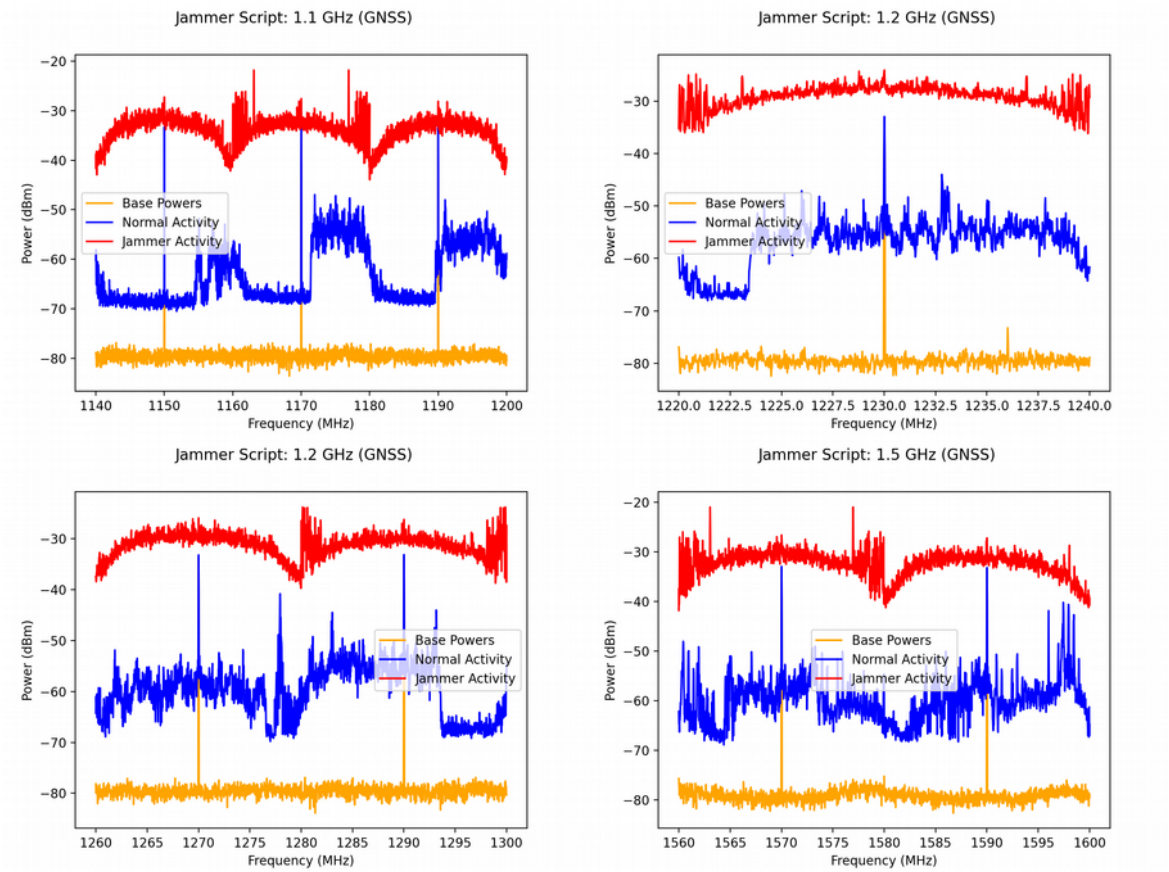


Figure 4.80: Comparison graphic of power values with jammer activity for GNSS frequencies

4.4.4. Jammer test at 2.4 GHz

For this test we will execute the jammer in the 2.4 GHz band with the drone in both WiFi and RC mode. First we will start with tests for RC mode.

Using the 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 4.81).

Figure 4.81: 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 (Figure 4.82). But we perceive that this signal is intermittent and doesn't remain all the time raised. 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 4.83).

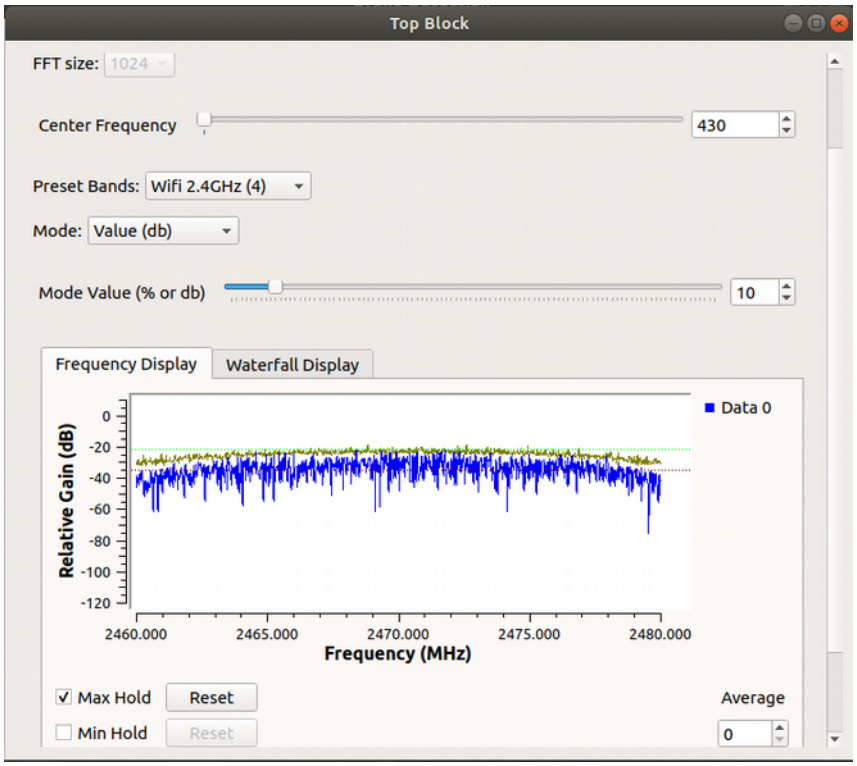


Figure 4.82: Jammer signal at 2470 MHz

Figure 4.83: Warning of interference in the drone remote controller app, caused by the jammer at 2470 MHz.

The drone communication, 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, when 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, and at 10 MHz bandwidth 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 4.84: Drone operating at 2428.5 MHz – 20 MHz BW

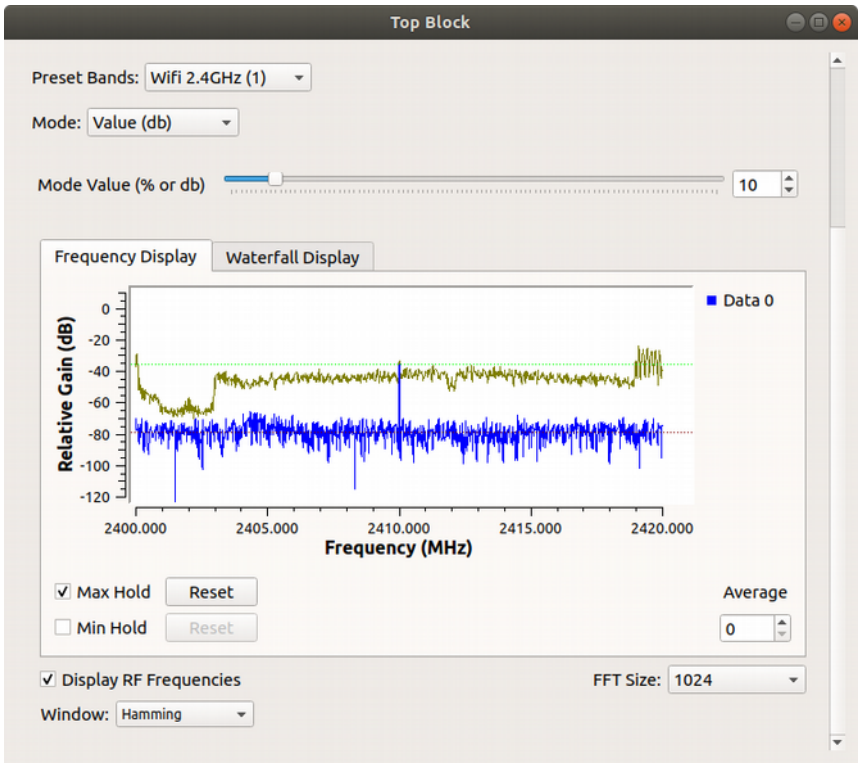


Figure 4.85: Drone operating at 2410.5MHz – 20 MHz BW

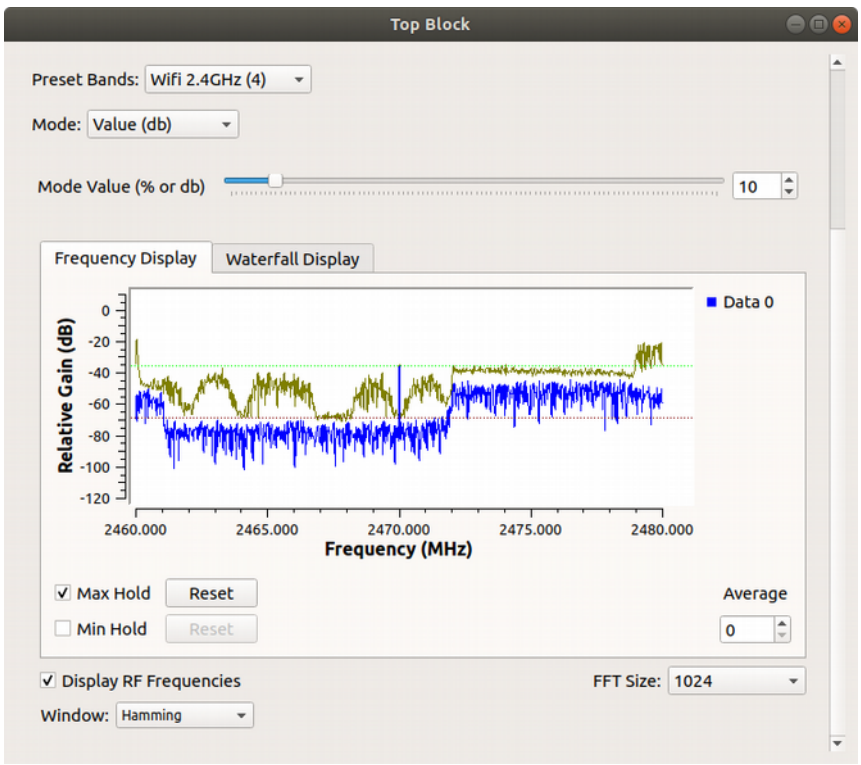


Figure 4.86: Drone operating at 2476.5MHz – 10 MHz BW

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 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, but in this case the result is immediate. The communication is lost, from the mobile app we can see that 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.

Figure 4.87: 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.

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

Drone Detector and Jammer

This software solution is a tool used to detect and block communications in all the spectrum from 1 MHz to 6 GHz. Even though it can be used for any purpose, it is specially designed to detect and jam communications in frequencies where drone activity is expected such as 433 MHz, 868 MHz and the 2.4 GHz band.

It consists of five executable Python scripts, which have different functions that will be explained further, and they use GNURadio as the telecommunications SDK. It is necessary to have an OSMOCOM compatible hardware such as the HackRF One or USRPs for reception and transmission of RF signals.

The scripts can be executed in any computer with the requisites previously installed, and is proven to work in a Raspberry Pi 3 B+.

The main script is the principal script that controls the execution parameters and the launch of the other scripts and is meant to be the first executed script. It is mandatory that a Base Scan is performed first before launching the Spectrum Scan or Band Scan scripts. Base Scans must be performed for any parameter combination (Sample Rate-FFT size) that is going to be used.

Software Requisites:

Python 2.7

GNURadio

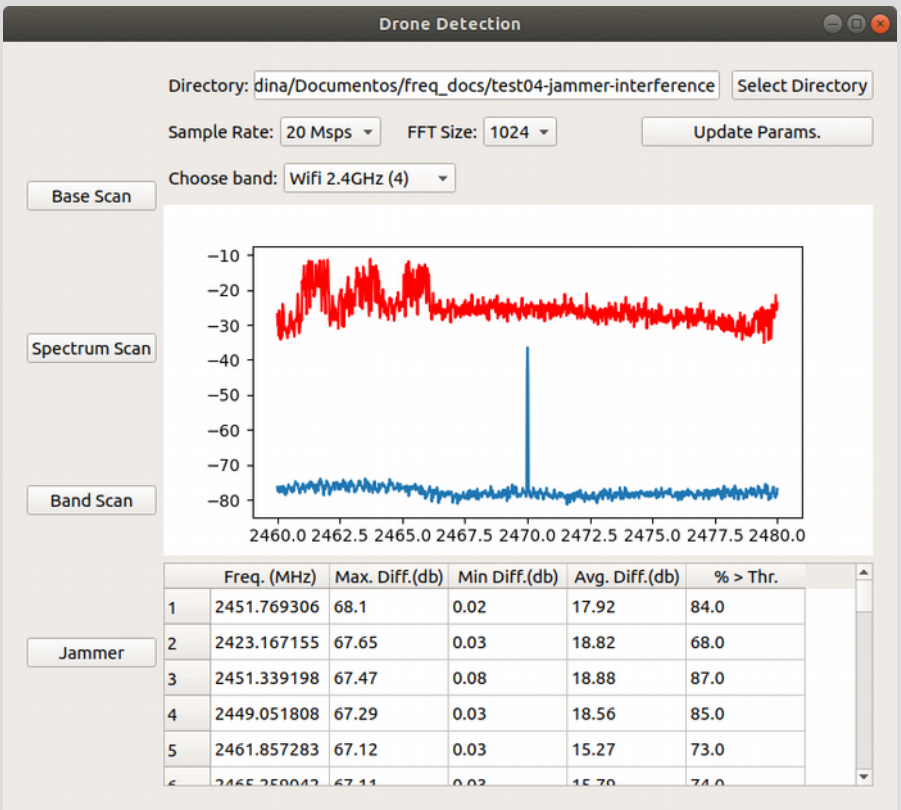
Hardware Requisites:

Computer with at least a 1.4 GHz 64 bit quadcore processor and 1 GB of RAM memory.

An OSMOCOM compatible SDR such as the HackRF One.

Main Script

This script is the controller script which launches the scanner and jammer scripts with the user selected parameters, and also works as a graphic visualize of the results obtained.



	Freq. (MHz)	Max. Diff.(db)	Min Diff.(db)	Avg. Diff.(db)	% > Thr.
1	2451.769306	68.1	0.02	17.92	84.0
2	2423.167155	67.65	0.03	18.82	68.0
3	2451.339198	67.47	0.08	18.88	87.0
4	2449.051808	67.29	0.03	18.56	85.0
5	2461.857283	67.12	0.03	15.27	73.0

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.

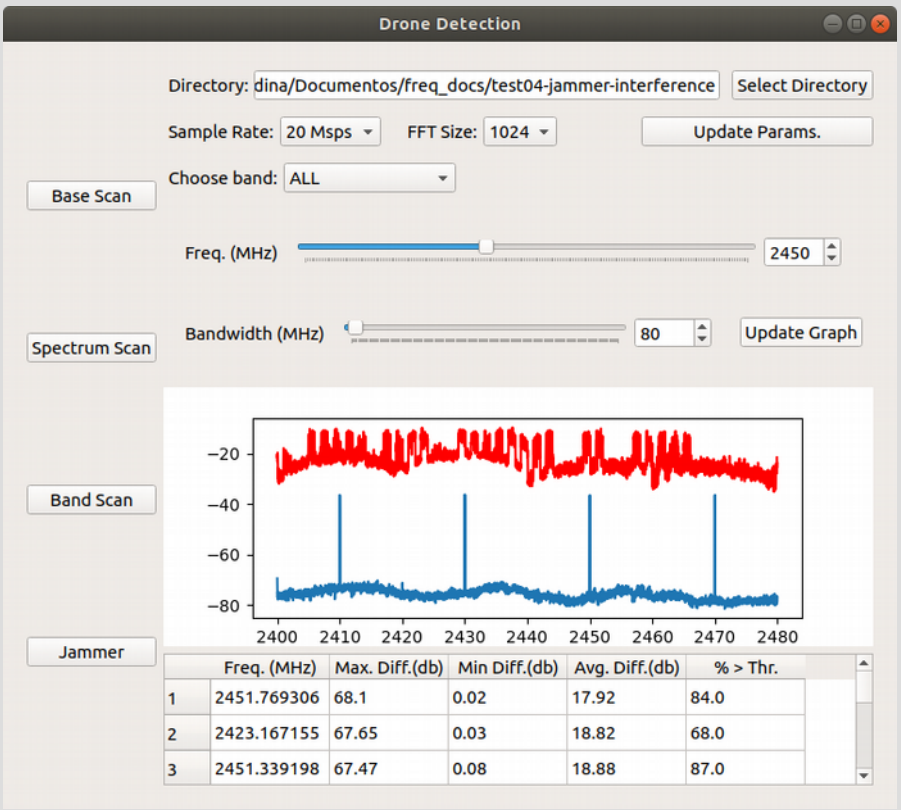
Table: The table displays all the data obtained from the spectrum scan script with the use of the data obtained from the base scan script. Here we find the frequency and the values that have exceede the threshold, with their respective maximum difference from the base, the minimum difference and the 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.

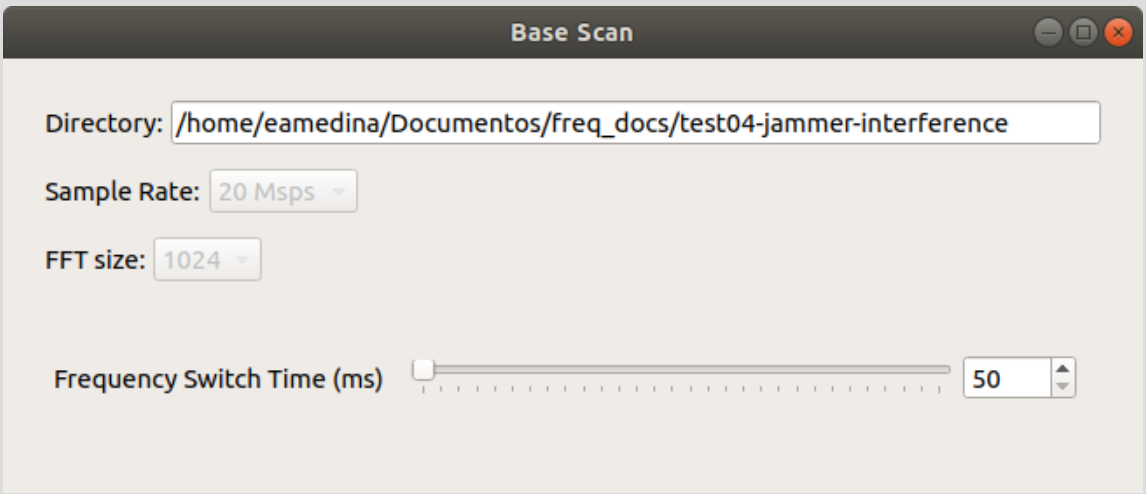
Jammer: launches the execution of the jammer script, which is in charge of generating a noise signal within a bandwidth determined by the user. This script is used to interfer or block the communications between a drone and its remote controller.



Base Scan

This script is in charge of obtaining the averaged base power values for all frequencies between 1 MHz and 6 GHz, storing them in a file database.

If this script is run in a laptop we recommend using 50 ms. as the frequency switch time and a total acquisition time of at least 10 minutes. If the script is executed in a Raspberry, we recommend a frequency switch time of 500 ms. and a total acquisition time of at least 20 minutes.



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.

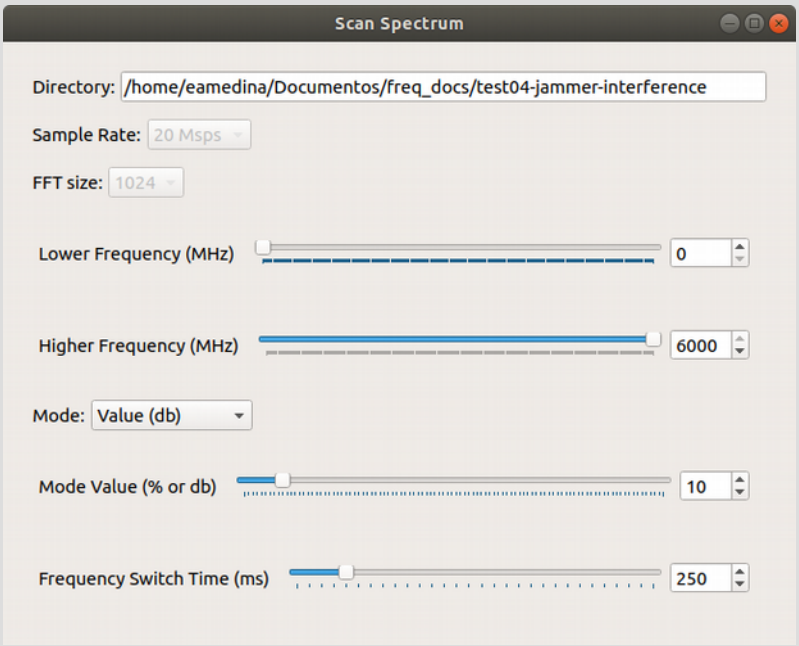
FFT Size: the size of bins in which the signal will be divided after FFT. This parameter is passed by the main script.

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

This script is in charge of comparing the real time power values against the base power values for all frequencies between 1 MHz and 6 GHz, storing the comparison results in a file database.

If this script is run in a laptop we recommend using 250 ms. as the frequency switch time and a total acquisition time of at least 10 minutes. If the script is executed in a Raspberry, we recommend a frequency switch time of 1000 ms. and a total acquisition time of at least 20 minutes.



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.

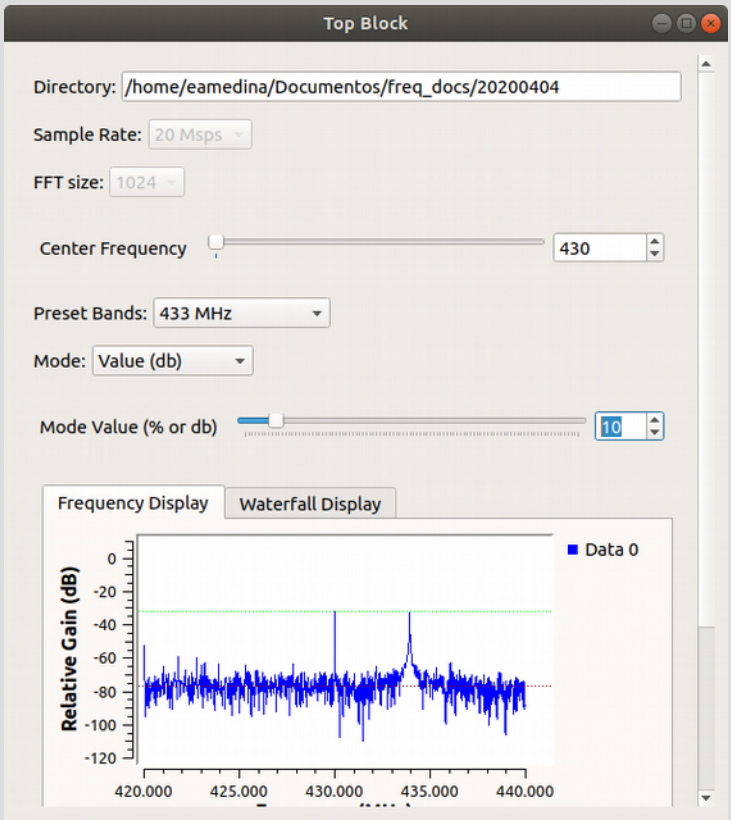
Fixed mode establishes the threshold as a fixed dB value above the base for each frequency.

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

This script has the same functionality than the spectrum scan script but limited to a bandwidth given by the sample rate and with a graphic tool to visualize the spectrum in real time.



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.

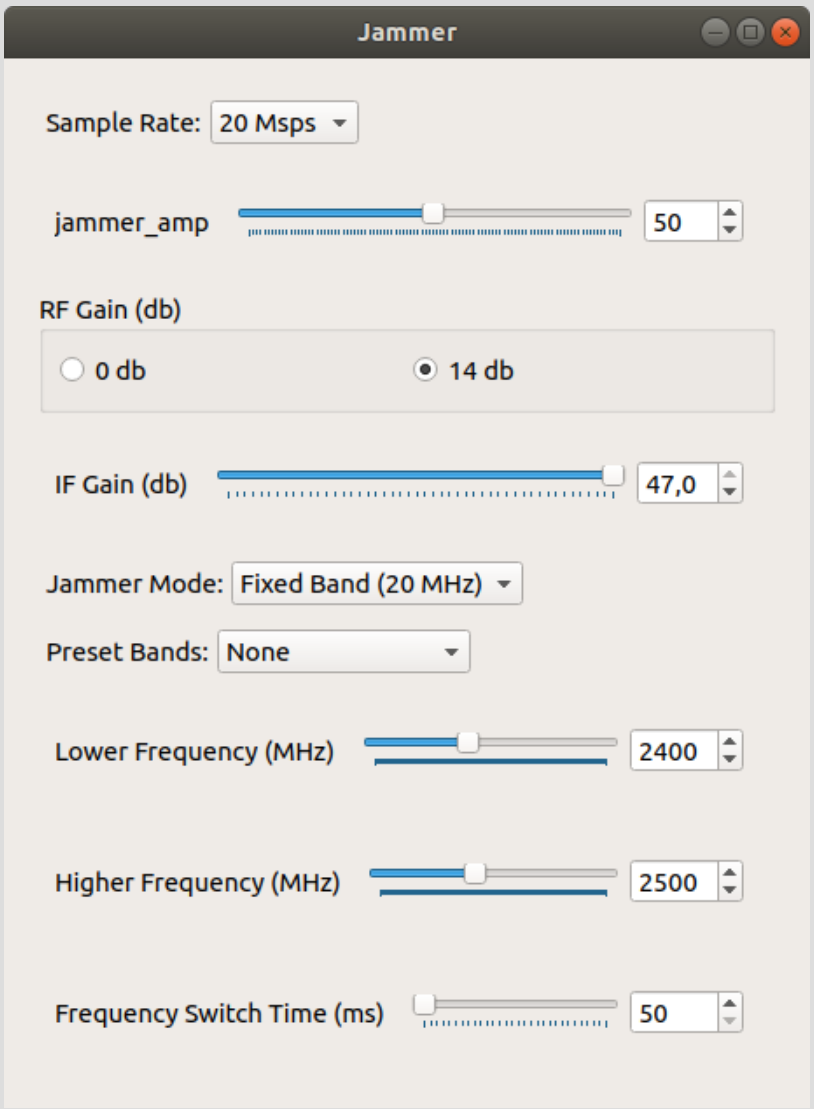
Fixed mode establishes the threshold as a fixed dB value above the base for each frequency.

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

This script is in charge of generating a noise signal of up to 20 MHz of bandwidth in a given frequency.



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.start_timer()

    def start_timer(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.start_timer()

    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.start_timer()

    def start_timer(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.start_timer()

    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.start_timer()
```

```
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.start_timer()
    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 start_timer(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 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 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 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 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

FCC: Federal Communications Commission

FFT: Fast Fourier Transform

FHSS: Frequency Hopping Spread Spectrum

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