**BACHELOR'S THESIS**

DEGREE IN COMPUTING ENGINEERING

# *"Design and implementation of a GUI to manage a particle accelerator module"*


AUTHOR:

**Salvador Jesús Megías Andreu**

SUPERVISED BY:

**Prof. Andrés Roldán Aranda**

DEPARTMENT:

**Electronics and Computers Technologies**


Granada, July 2022

***"Design and implementation of a GUI to manage a particle accelerator module"***

AUTHOR:

**Salvador Jesús Megías Andreu**

SUPERVISED BY:

**Prof. Andrés Roldán Aranda**

D. Andrés María Roldán Aranda, Profesor del departamento de Electrónica y Tecnología de los Computadores de la Universidad de Granada, como director del Trabajo Fin de Grado de D. Salvador Jesús Megías Andreu,

Informa:

Que el presente trabajo, titulado:

***"Design and implementation of a GUI to manage a particle accelerator module"***

ha sido realizado y redactado por el mencionado alumno bajo mi dirección, y con esta fecha autorizo a su presentación.

Granada, a 06 de Julio de 2022

Fdo. Prof. Andrés María Roldán Aranda

Los abajo firmantes autorizan a que la presente copia de Trabajo Fin de Grado se ubique en la Biblioteca del Centro y/o departamento para ser libremente consultada por las personas que lo deseen.

Granada, a 06 de Julio de 2022

Fdo. Salvador Jesús Megías Andreu

DNI: 75578537P

Fdo. Prof. Andrés María Roldán Aranda

# ”Design and implementation of a GUI to manage a particle accelerator module”

**Salvador Jesús Megías Andreu**

**KEYWORDS:**

EPICS, SCPI, Ethernet, GUI, BTESA, API, Informatic, OpenAPI, TCP, UDP, SSPA, BLAS, Resonance Cavity, Python, PyQt, IOC, THD.

**ABSTRACT:**

The main purpose of this project is developing a GUI in PyQt that is capable of controlling and monitoring various oscilloscopes (Anritsu MS2830A and Agilent N9020A in our case) remotely, as well as the input and output signals of a BLAS, and that is able to establish a connection with all the records of our EPICS IOC in order to have access to real-time data from our entire system from any device on our subnet quickly.

This Bachelor's Thesis is approached from an ambitious perspective, since the purpose of this project as a whole is to prepare a BLAS manufactured by the company BTESA for the educational field, that is, It is intended to test and study this technology in addition to developing software capable of interacting with it in order to be able to sell these devices to universities in the future, and in this way, that future students can obtain theoretical and practical knowledge related to particle accelerators such as RF amplifiers.

Due to the great complexity involved in this project, several engineers from other areas of engineering (telecommunications engineers and electronics engineers) who are doing their TFM share part of the project with me. This fact allowed me to interact with experts in other areas of knowledge other than mine in a professional way, and to face together problems that arose throughout the project, and that we have solved as a team, thus learning new concepts, which otherwise, it would have been difficult to learn.

The result of the exposed culminates with the obtention of a complete and functional monitoring and control system for the equipment, which complies the requirements defined in the preliminary stages, and supposes the finalization of the Degree.

# ”Design and implementation of a GUI to manage a particle accelerator module”

**Salvador Jesús Megías Andreu**

**PALABRAS CLAVE:**

EPICS, SCPI, Ethernet, GUI, BTESA, API, Informática, OpenAPI, TCP, UDP, SSPA, BLAS, Cavidad de Resonancia, Python, PyQt, IOC, THD.

**RESUMEN:**

El objetivo principal del presente proyecto es desarrollar una GUI en PyQt que sea capaz de controlar y monitorizar diversos osciloscopios (Anritsu MS2830A y Agilent N9020A en nuestro caso) de forma remota, así como las señales de entrada y salida de un BLAS, y que sea capaz de establecer una conexión con todos los registros de nuestro EPICS IOC para poder así tener acceso a los datos en tiempo real de todo nuestro sistema desde cualquier dispositivo de nuestra subred rápidamente.

Este Trabajo Fin de Grado se aborda desde una ambiciosa perspectiva, puesto que la finalidad de este proyecto en su conjunto es preparar un BLAS fabricado por la empresa BTESA para el ámbito educacional, es decir, se pretende probar y estudiar esta tecnología además de desarrollar software capaz de interactuar con la misma para poder así vender estos dispositivos a universidades en un futuro, y de este modo, que futuros estudiantes puedan obtener conocimientos teóricos y prácticos relativos a aceleradores de partículas tales como los amplificadores RF.

Debido a la gran complejidad que atañe este proyecto, diversos ingenieros de otras áreas de la ingeniería (ingenieros en telecomunicaciones e ingenieros en electrónica) que están realizando su TFM, se reparten parte del proyecto conmigo. Este hecho, me permitió interactuar con expertos en otras áreas de conocimiento distintas a la mía de forma profesional, y afrontar juntos problemas que iban surgiendo a lo largo del proyecto, y que en equipo hemos resuelto aprendiendo así nuevos conceptos, que de otra forma, hubiese sido difícil aprender.

El resultado de todo lo expuesto culmina con la obtención de un sistema de monitoreo y control sobre el equipo completo y funcional, que cumple con los requisitos definidos en etapas iniciales, y con el cual se cierra la etapa universitaria de Grado.

*‘There is a driving force more powerful than steam, electricity and atomic energy: the will.*

*–Albert Einstein–’*

# *Agradecimientos:*

Me gustaría mostrar mi más sincero agradecimiento a todas aquellas personas que a lo largo de estos años me han acompañado en la carrera y, a día de hoy, en este proyecto final, gracias a mis padres por hacer posible mi formación, a mis amigos que nunca han dudado que lo conseguiría y, especialmente, a Annys Andreina Palacios Marcano, por su inmensurable fe en mí, gracias por tu valioso apoyo y por haber visto en mí potencial y cualidades de las que yo creía carecer, gracias por animarme siempre a mejorar cada día y a no conformarme sin haber luchado por dar lo mejor de mí, tanto a nivel profesional como personal.

Gracias, también, a aquellos que se convirtieron en un referente profesional, personas extraordinarias que me demostraron la amplitud y diversidad de posibilidades que existen en el campo de la informática, gracias por ayudarme a explorar y encontrar mi camino, gracias por la humildad y paciencia con la que me habéis enseñado, tendréis mi admiración por siempre.

Y por supuesto, gracias a mi tutor Andrés Roldán Aranda, por brindarme la oportunidad de participar en este proyecto tan ambicioso, un desafío que hoy orgullosamente puedo dar por finalizado; si bien es cierto que se encuentra muy alejado de cualquier cosa que haya visto en la carrera, me alegra que fuese así pues me permitió salir de mi zona de confort, hacer cosas muy interesantes y, sobre todo, aprender.

A todos vosotros... Gracias, este proyecto también es vuestro.

# Contents

# List of Figures

# List of Tables

# List of Videos

# Glossary

**Adobe Acrobat DC** Acrobat DC is the latest version of Acrobat subscription software. It is the most productive and collaborative mobile PDF solution offered by Adobe; combines Acrobat desktop software and mobile scanning app, signature app, and Acrobat Reader mobile app; enhanced with premium mobile features and premium Document Cloud services..

**API** An application programming interface or API is the set of functions and procedures (or methods, in object-oriented programming) that a certain library offers to be used by other software as an abstraction layer. They are generally used in libraries..

**ASGI** (Asynchronous Server Gateway Interface). ASGI consists of two different components:

- A protocol server, which terminates sockets and translates them into connections and per connection event messages.
- An application, which lives inside a protocol server, is called once per connection, and handles event messages as they happen, emitting its own event messages back when necessary.

.

**BTESA** BTESA is the Spanish leading provider of TV Transmitters and high power Solid State Amplifier systems: BTESA website..

**CubeSat** Miniaturized satellite normally for space research, with dimensions of $1\,\mathrm{dm}^3$ and mass lower than 1.33 kg per unit.

**DHCP** DHCP (Dynamic Host Configuration Protocol) is a protocol that provides quick, automatic, and central management for the distribution of IP addresses within a network.

**Ethernet** Ethernet is the traditional technology for connecting devices on a wired local area network (LAN) or wide area network (WAN), allowing them to communicate with each other through a protocol..

**GranaSAT** GranaSAT is an academic project from the University of Granada originally consisting in designing and developing a picosatellite (CubeSat). Coordinated by the Professor Andrés María Roldán Aranda, GranaSAT is a multidisciplinary project with students from different degrees, where they can acquire and enlarge the knowledge necessary to face an actual aerospace project.

**OpenAPI** OpenAPI is a standard for the description of application programming interfaces (). In particular, OpenAPI can be used to describe, develop, test, and document REST-compliant APIs..

**Switch** A network switch is a small device that centralizes communications among several connected devices in one local area network (LAN)..

**THD** Total Harmonic Distortion is a useful technique to analyze any non-linear behavior of a system..

**ToolTips** A ToolTip is a short description, usually just a few words, that appears when the user holds the mouse pointer briefly over a control or another part of the user interface without clicking.

# Acronyms

**ATE** Automatic Test Equipment.

**BLAS** Beam Loading Advanced Simulator.

**EPICS** Experimental Physics and Industrial Control System.

**GUI** Graphical User Interface.

**IOC** Input/Output Controller.

**IP** Internet Protocol.

**PDF** Portable Document Format.

**PV** Process Variable.

**PyQt** Python QT..

**Python** Python main Website..

**RF** Radio Frequency.

**SCPI** Standard Commands for Programmable Instruments.

**SSPA** Solid State Power Amplifier.

**TCP** Transmission Control Protocol.

**UDP** User Datagram Protocol.

**UGR** University of Granada.

# Chapter 1

# Introduction

## 1.1 Context

The following Bachelor's thesis has been carried out as the culmination of my computer engineering career. This project was offered by Professor Andrés María Roldán Aranda and carried out by me in *Granasat's* laboratory.

Granasat is an aerospace development group from the University of Granada (UGR), which is made up entirely for students and under the supervision of the professor Dr. Andrés María Roldán Aranda.

**Figure 1.1** – *Granasat logo*

The Granasat project, which logo is shown in the Figure 1.1, began in 2013 as a student initiative composed by several students who were interested in aerospace engineering, and wanted to focus their technical studies into the aerospace scope, so they decided to participate in the BEXUS/REXUS programme, defined as [10].

Nowadays, multiple projects of multiple branches and careers are carried out in Granasat. In my case, my thesis project (TFG) titled: "Design and implementation of a GUI to manage a particle accelerator module", which is part of a development and cooperation initiative between Professor Andrés Roldán and the company BTESA [8]. This initiative consists of 2 parts, the first is to check the operation by performing various tests on the hardware (a non-commercial BLAS prototype), this part is in charge of my colleague Andoni, a telecommunications engineering master's student. While the second part, which I am in charge of, consists of the design, development and implementation of an interface (GUI) and several Python libraries to be able to monitor, control and visualize several machines in addition to the BLAS.

## 1.2 Motivation

The main motivation that has led me to choose this project as the TFG is that it is something totally different from what I learned at university, since it is a case of research and development directly applicable to real life, being able to interact directly with real machines and prototypes (BLAS) without using any simulator.

In addition, being a job in which machines have been used, I have had the opportunity to learn electronic concepts and technical knowledge in order to understand how to operate these machines, which otherwise would have been impossible for me to learn.

Another reason for my motivation is the fact that my work on this project is just the beginning of a job that will have further development after me, so I am proud to be the first computer science student in contributing to this project and in laying the foundations to be able to continue advancing and improving without having to deal from scratch with the control of the machines.

## 1.3 Project Goals

The main objectives are:

- **Carry out an analysis and investigation of the machines and tools that we will use in the project.**
- **Design and develop the necessary software to carry out the project (Python libraries, as well as the GUI and integration in EPICS).**
- **Check if the implemented system works correctly, verifying that its purpose is fulfilled.**
- **Grow as a professional facing a research project as complex as this one, showing what was acquired during the degree, and above all, showing the knowledge acquired throughout its development.**

## 1.4 Project Requirements



The requirements that were given to me when creating the project are the following:

| Ref. | Formal Requirements |
|---|---|
| FoR.1 | Use EPICS to share in real time the data obtained and managed in the system between the different devices of the system, regardless of the operating system they have. |
| FoR.2 | Create libraries in Python to be able to remotely control and manage Anritsu MS2830A (2.12) and Agilent N9020A (2.12) machines via Ethernet. |
| FoR.3 | Create an API through OpenApi that simulates the input and output data of the BLAS to be able to integrate this functionality in the GUI as well. |
| FoR.4 | Create a modern and interactive GUI with QT that integrates everything created previously |

**Table 1.1** – *Formal Requirements*

### 1.4.1 GUI Requirements

| Ref. | Formal Requirements |
|---|---|
| GUI.FoR.1 | Must be capable of communicating with the Anritsu MS2830A (2.12) and Agilent N9020A (2.12) machines via Ethernet by using the Python developed Libraries. |
| GUI.FoR.2 | Must be capable of collecting BLAS data (in our case from an API) |
| GUI.FoR.3 | Must be capable of transmitting all this information through EPICS in order to have real-time monitoring throughout of all devices on our subnet. |
| GUI.FoR.4 | Must be Responsive. |
| GUI.FoR.5 | Must be Multilanguage. |
| GUI.FoR.6 | Must use Multithreading to avoid GUI hang or freeze. |
| GUI.FoR.7 | The GUI should be as interactive as possible. |

**Table 1.2** – *Formal Requirements for GUI*

1

## 1.5 Chapter Description

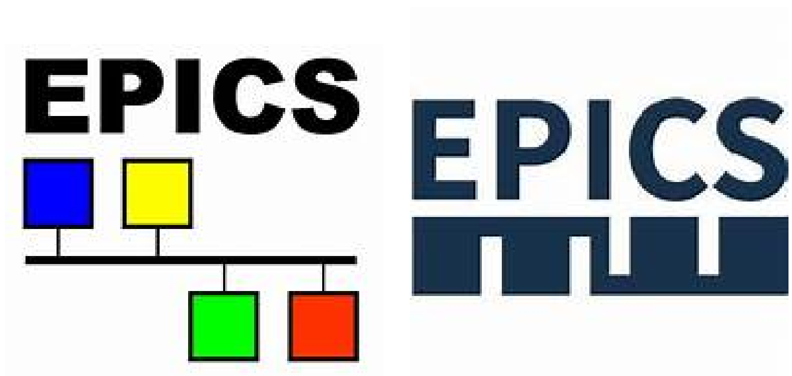
Following the objectives listed above , the project was developed after planning the next structure:

- **Chapter 2: Background study.**
  This chapter will contain all the previous knowledge needed in order to understand the work described in the following chapters.
- **Chapter 3: Project planning.**
  In this chapter the basis of the project management will be described. The Gantt Diagram (3.1) will be shown with which the development of the project as a whole has been planned, and the time that each stage of the project should last (said diagram was fulfilled).
  Likewise, an estimated budget (3.2) of what would be the value of replicating said project in a company will be shown.
- **Chapter 4: System description and design.**
  In this chapter, an overview of the system as a whole and its operation will be made, as well as the design and development of all the parts of the project that concern me (these parts are exposed in previous chapters), which are : the Python libraries (4.3), the GUI (4.4), the API (4.5), and the EPICS system (4.6).
- **Chapter 5: System testing.**
  In this chapter, the operation of the entire GUI as a whole will be shown through various videos (5).
- **Chapter 6: Conclusions, future work and lessons learned.**
  In the last chapter the conclusions obtained throughout the project will be presented(6.1), as well as future improvements that can be applied to it (6.2).

# Chapter 2

# Background study

## 2.1 EPICS Study




**(a)** *Traditional EPICS logo [2]* **(b)** *Modern EPICS logo [1]*

**Figure 2.1** – *EPICS logos*

EPICS [9] is a software environment used to develop and implement distributed control systems (such as particle accelerators, large telescopes...). Said environment is conceived and designed to develop control systems that usually contain a large network of computers.
EPICS provides control and data collection tools for the experimental physics community, currently being used by up to 70 universities and institutions, while making contributions to the development and evolution of the system.

The EPICS toolset enables the creation of server and client applications. Servers provide data access (read or write) locally or over a network.

Clients can display, store, and manipulate data. Client software ranges from user interface tools to powerful data management services.

EPICS can be installed as explained in the following annex: EPICS Installation in Linux OS

### 2.1.1 EPICS basic system structure

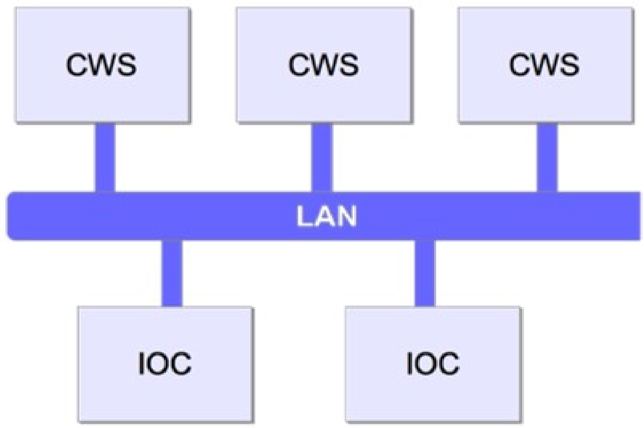



**Figure 2.2** – *EPICS Structure*

**Basic components:**

Now we are going to make a brief description of each of the elements of the previous image:

- **IOC:** It is the input/output server component of EPICS. It can be any platform that can support EPICS runtime databases along with the other software components (Examples: regular desktop, realtime OS based systems like vxWorks and even EPICS IOC can run on low cost hardware). cost as a RaspberryPi or similar).
- **CWS (Client WorkStation):** Workstation on which EPICS tools and client applications can be used. Each CWS station is the equivalent of an EPICS system client (Examples: user interfaces tools and the data file => server machine or similar with a "normal" OS, ie Linux, Windows or MacOS).
- **LAN (Local Area Network):** Standard communication network based on Ethernet that allows communication between the IOCs and the CWS (local area network).

An EPICS control system may be made up of multiple IOCs and CWSs, all of which communicate over a LAN. This separation of servers and clients makes it easier to set up such systems and also makes the system more robust, since servers and clients can be removed or added without shutting down.

### 2.1.2 Software components of an IOC

An IOC is a process made up of the following software elements:

- **IOC database:** Database containing a set of named records of various types. These registers house the PV (Process variables).
- **Record support:** Each record type has an associated set of record support routines to implement the record type's functionality.
- **Scanners:** Mechanisms to process records in the IOC database.
- **Device support:** Device support routines link I/O data to database records.
- **Device drivers:** They handle access to external devices.
- **Sequencer:** Finite State Machine (this module is external, so it is not included in the main EPICS software distribution)
- **Channel Access or pvAccess:** Interface between the outside world and the IOC to access the EPICS database through the network.



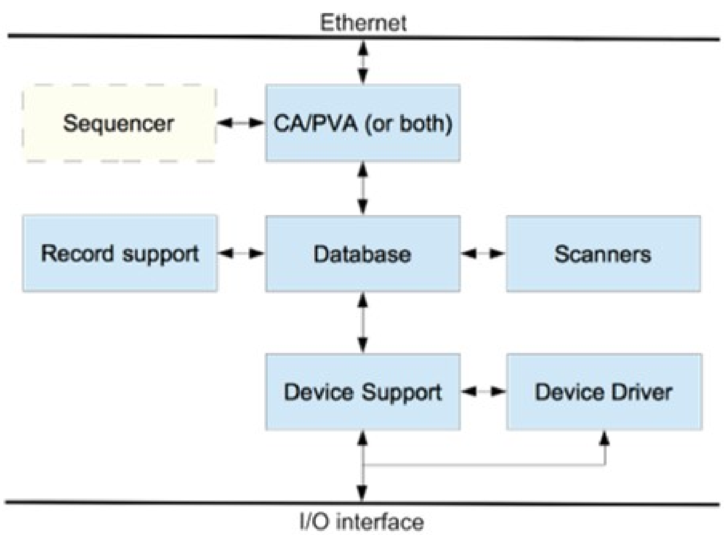



**Figure 2.3** – *IOC Software*

The reference manual for record types can be found collected by official sources in the following link (from the latest version of EPICS => 7.0.4): EPICS Records.

### 2.1.3 SMALL EXAMPLE TO TEST EPICS (file: TFG_SalvadorEPICS.txt)



```

record(ai, "temperature:water")
{
     field(DESC, "Water temperature in the fish tank")
}

record(aai,"array:temperatures")  {
     field(DESC,"array of analog inputs")

}
```

**Listado 2.1** – *Example of an easy IOC definition (his records)*

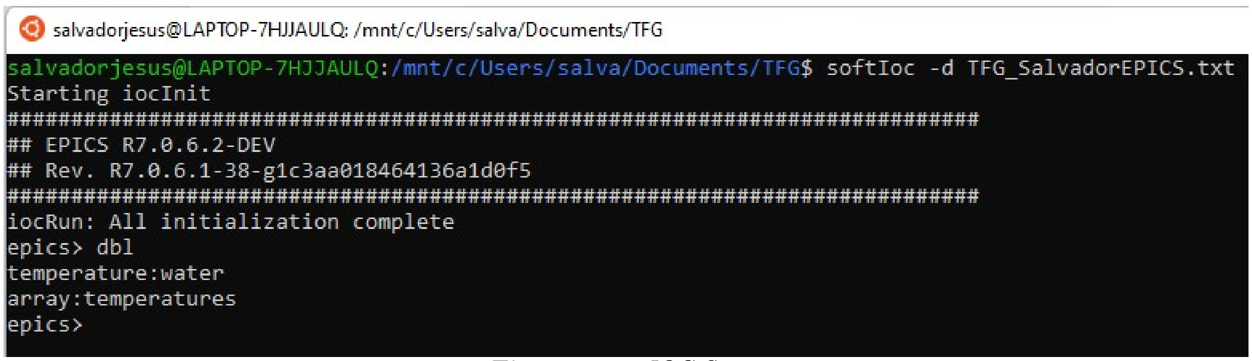
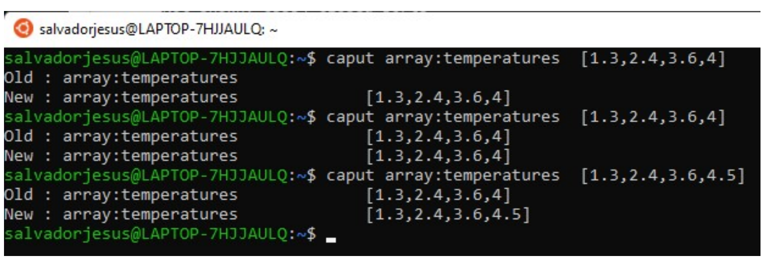
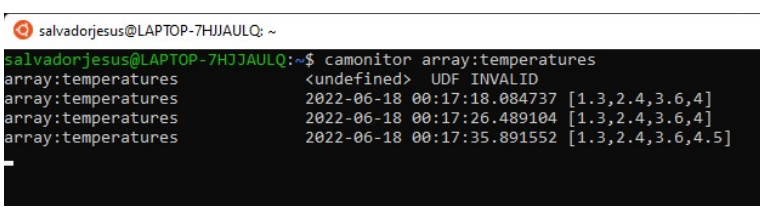



**Figure 2.4** – *IOC Setup*




**(a)** *Changing records values in EPICS*




**(b)** *Monitoring changes*

**Figure 2.5** – *EPICS IOC example*

As we can see we have created a simple IOC and we have deployed it, making changes and obtaining results with the basic functions of EPICS: camonitor, caget and caput:

- **camonitor:** Used to monitor a record in real time.
- **caput:** Used to modify the value or values of a record.
- **caget:** It is used to obtain the value of the record at the moment this request is made.



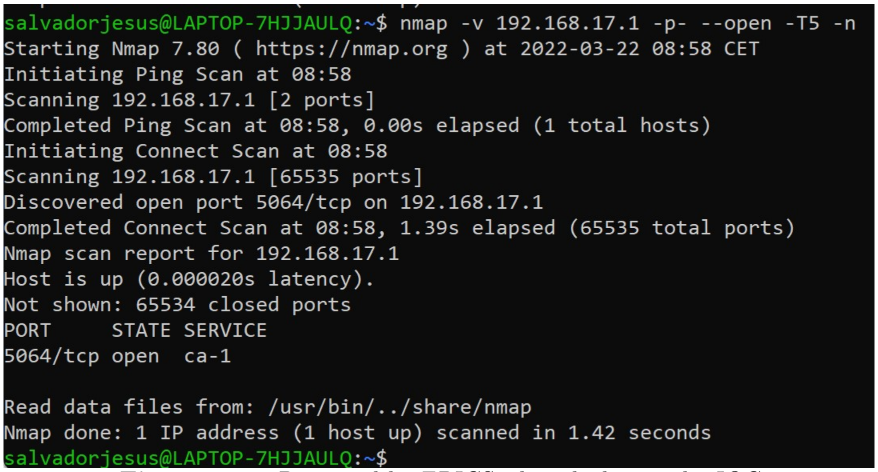


**Figure 2.6** – *Port used by EPICS when deploying the IOC*

As we can see, EPICS uses port 5064 for its operation. EPICS searches for connections using the UDP protocol, and once the connection is established with the device in question, EPICS proceeds to perform a data transfer or communication with the connected device using the TCP protocol.
This is done in this way so that the connection process (using the UDP protocol) is simpler and faster, and once the connection is established, proceed with the TCP protocol, which, although slower, is much safer in terms of security in the data transferred or communicated.

### 2.1.4 Interact with EPICS using Python

Doing some research, I found a package with which we would be able to communicate with EPICS through Python. This package is PyEpics [14]:

**Figure 2.7** – *PyEpics logo*

The Python epics package provides several function, modules, and classes to interact with EPICS Channel Access.

A simple way to interact with the EPICS Records of our EPICS IOC would be just to use the caput(),caget() or camonitor() functions offered by this Python package:

**Procedural Approach: caget(), caput()**

To get values from PVs, you can use the `caget()` function:

```
>>> from epics import caget, caput, cainfo
>>> m1 = caget('XXX:m1.VAL')
>>> print(m1)
1.2001
```

To set PV values, you can use the `caput()` function:

```
>>> caput('XXX:m1.VAL', 1.90)
>>> print(caget('XXX:m1.VAL'))
1.9000
```

**Figure 2.8** – *Normal functions in PyEpics*

But, there is another more efficient way if you need to reference the same PV (Record value) more than once throughout the program, which is to create PV objects from where we can access and modify the PV at any point in the program by calling created object.

**Creating and Using PV Objects**

If you are repeatedly referencing the same PV, you

```
>>> from epics import PV
>>> pv1 = PV('XXX:m1.VAL')
```

**Figure 2.9** – *Creating a PV object*

With these PV objects, the task of interacting with the PVs by accessing or modifying their content becomes an easier task:

To set a PV's value, you can either use

```
>>> pv1.put(1.9)
```

or assign the `value` attribute

```
>>> pv1.value = 1.9
```

**Figure 2.10** – *Functions that we will use to set PVs values*



```
>>> print(pv1.get())
1.90
```

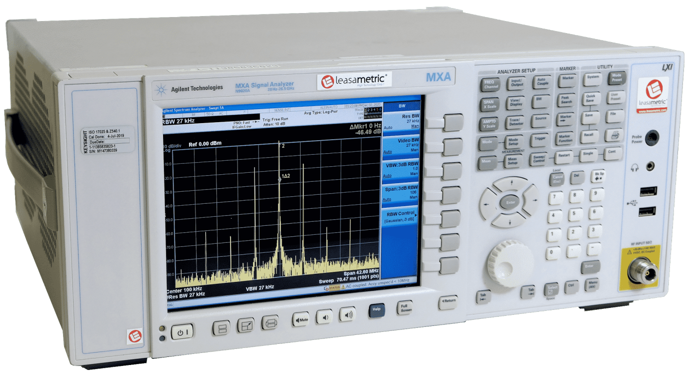
or

```
>>> print(pv1.value)
1.90
```

**Figure 2.11** – *Functions that we will use to get PVs values*

## 2.2 Python Libraries via Ethernet

One of the requirements of the project was to create 2 libraries in Python to be able to control the following machines via Ethernet:




**(a)** *Agilent N9020A machine [5]*


**(b)** *Anritsu MS2830A machine [7]*

**Figure 2.12** – *Machines*

First of all, say that the only thing these machines understand are the SCPI commands.

SCPI defines a standard intended for instrumentation control. The SCPI describes a language that is useful for controlling test instruments. SCPI offers a standard syntax, data interchange format and command structure.

The key objective of SCPI is to minimize the development time of an ATE program. The objective is accomplished through providing a reliable programming environment for data usage and instrument control. This reliable programming environment is gained using defined data formats, program messages and instrument responses across every SCPI equipment irrespective of the designer.

So we have to find a Python package with which we can send SCPI commands to the machines, although the SCPI commands used in both machines are similar, there are some changes from one machine to another, so we will have to look in the respective manuals [4][6] of each machine the SCPI commands recognized by each of said machines.

Investigating, I found a package with which I could perform this task, PyVISA [15]:

**Figure 2.13** – *Python package to communicate*

Pyvisa is the package that I am going to use to communicate with machines, since it is a simple library to use in Python that allows Ethernet communication (which is what interests me) with devices through.

```

import pyvisa as visa
import numpy as np
from struct import unpack
import pylab
import time
import math
from matplotlib import pyplot as plot

# Establish Connection
rm = visa.ResourceManager('@py') # Calling PyVisaPy library
#scope = rm.open_resource('USB0::0x0699::0x0409::C010730::INSTR') # Connecting via USB
# 192.168.1.200 => IP de la máquina Agilent en la subred
scope = rm.open_resource('TCPIP::192.168.1.200::INSTR') # Connecting via LAN
print(scope)
```

**Listado 2.2** – *Agilent connection example via Ethernet*

2

Pyvisa consists of two main functions once we have established a connection:

- **write():** Is a function with which we can send the SCPI commands (the commands or language that the machine understands) to the machine, this function is used to change parameters in the machine, that is, when you use this function it is to modify something, not to wait for a response from the machine, since this function does not collect any data from the machine except for the successful status of sending the command.
- **query():** This function is the one we will use to request data or information from the machine, through this function we can request data from the machine and most importantly, the machine returns this data, so when we use this function it will be to collect the information that we Give the machine in a variable or print it directly, since all the machine returns are strings.

## 2.3 Python QT

**Figure 2.14** – *PyQT logo*



PyQt [3] connects the Qt C++ cross-platform framework with the Python language, it is a GUI module.

The principle on which a Qt class functions is related to a slot mechanism responsible for offering communication between items with the purpose of designing re-usable software components with ease.

Also, Qt comes with Qt Designer, a tool that acts as a graphical user interface. PyQt can design Python code from Qt Designer, while adding new GUI controls when both Qt Designer and Python programming language are used.

I will use QT designer to create the GUI we talked about in the project requirements. Now I am going to describe part of the study that I carried out to achieve certain functionalities in the GUI that we are going to develop.

Now I am going to describe part of the study that I carried out to achieve certain functionalities in the GUI that we are going to develop.

2

### 2.3.1 MultiThreading in PyQt

Multithreading can be useful in case you need your QT GUI to hold Widgets or elements that use functions with infinite loops or conditional loops, of which you don't know when they are going to end.
In those cases, Multithreading would need to be implemented to prevent the GUI from freezing and inoperative.

Here we can find an easy code to find out how multithreading works in Python QT: Example to show Multithreading in PyQt

- **run():** The starting point for the thread. After calling start(), the newly created thread calls this function.
- **super():** The idea of super() is that you don't have to bother calling both superclasses "___init___()" methods separately – super() will take care of it.
- **We will use "Qtcore.pyqtSignal()" so that the threads can communicate with our class sharing data through these signals declared with Python QT, we can define the type of signals that we want to treat and handle as ints, floats, lists...**

This is how looks like the example to explain Multithreading in PythonQT, and it is the one in which I noticed and studied to be able to extrapolate Multithreading to the elements or widgets of my GUI that needed it.

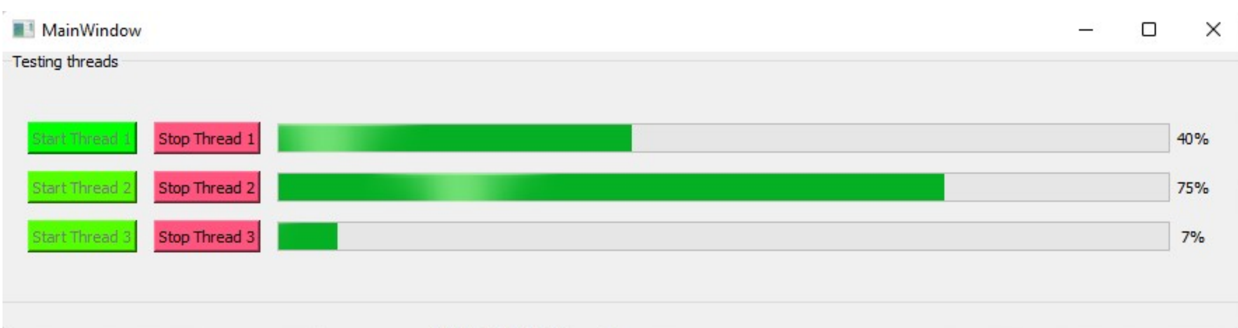



**Figure 2.15** – *Multithreading example*

### 2.3.2 Multilanguage GUI

We are going to see two ways to get multi-language support in Python:

- **One of the ways is going to be using the Google translator (ONLINE):**
  - Advantages:
    * It's fast
    * It is versatile (supports many languages)
    * It is robust
    * It's simple to manage
  - Disadvantages
    * Depends on internet connection
- **The other way will be using Python's argostranslate module (OFFLINE):**
  - Advantages:
    * It is robust
    * It is very simple
    * It's reasonably fast
    * Does not depend on the internet to work
  - Disadvantages
    * You need to download dictionary files for it to work (takes up memory)

#### 2.3.2.1 Google Translate in Python

```python
from googletrans import Translator, constants
from pprint import pprint

translator = Translator()

# Vamos a realizar la traducción al chino:
translation = translator.translate("Hola Mundo", dest="zh-cn")

prueba = translation.text
print(prueba)
print(type(prueba))

# Ahora en alemán:
translation = translator.translate("Hola Mundo", dest="de")

prueba = translation.text
print(prueba)
```

**Listado 2.3** – *Translation OFFLINE in Python with Google Translator*


**Figure 2.16** – *Google translator in Python example*

As we can see, it returns the translation in a string, so it is very easy to handle.

#### 2.3.2.2 ArgosTranslate in Python

We must download the dictionary file that we want to use, specifically I am going to download the file: en_es.argosmodel

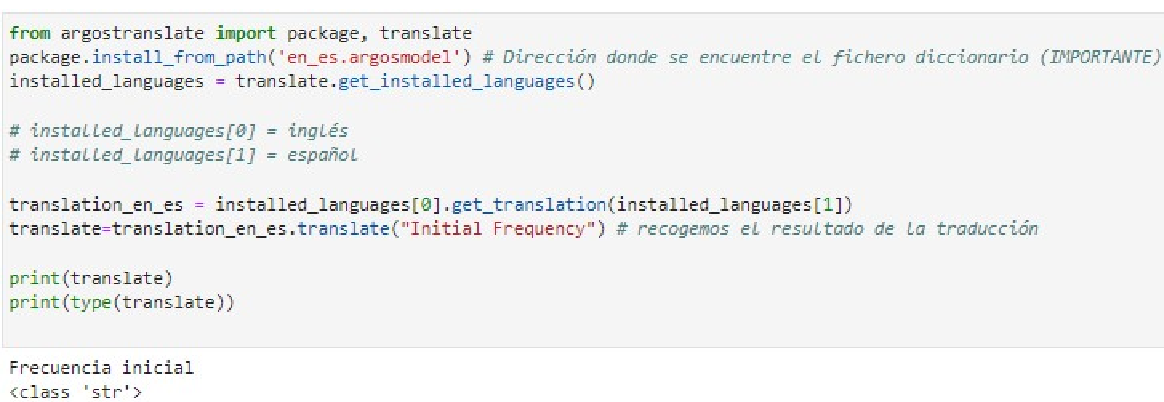
Which is a file with which we will be able to translate OFFLINE from English to Spanish (there are more languages available: from Spanish to English, from English to German, from English to Italian, from English to French... etc. )

Link where to download the dictionary files: Dictionary Files for argostranslate

Once we have the file downloaded, we are going to carry out an EXAMPLE OF USE:

```
from argostranslate import package, translate
package.install_from_path('en_es.argosmodel') # Dirección donde se encuentre el fichero diccionario (IMPORTANTE)
installed_languages = translate.get_installed_languages()

# installed_languages[0] = inglés
# installed_languages[1] = español

translation_en_es = installed_languages[0].get_translation(installed_languages[1])
translate=translation_en_es.translate("Initial Frequency") # recogemos el resultado de la traducción

print(translate)
print(type(translate))
```

**Listado 2.4** – *Translation OFFLINE in Python with argostranslate*



```
from argostranslate import package, translate
package.install_from_path('en_es.argosmodel') # Dirección donde se encuentre el fichero diccionario (IMPORTANTE)
installed_languages = translate.get_installed_languages()

# installed_languages[0] = inglés
# installed_languages[1] = español

translation_en_es = installed_languages[0].get_translation(installed_languages[1])
translate=translation_en_es.translate("Initial Frequency") # recogemos el resultado de la traducción

print(translate)
print(type(translate))

Frecuencia inicial
<class 'str'>
```

**Figure 2.17** – *Argostranslate translator in Python example*

As we can see, it returns the translation in a string, so it is very easy to handle.

#### 2.3.2.3 Proposal => Why not?

What if we make a hybrid system so that when we have access to the internet, Google's translator is used (to take advantage of its enormous versatility), and if not, argostranslate is used?

```python
from googletrans import Translator, constants
from argostranslate import package, translate

try:
    translator = Translator()

    # Vamos a realizar la traducción al español:
    translation = translator.translate("This is an online traduction made by Google translator", dest="es")

    prueba = translation.text
    print(prueba)
    print(type(prueba))



except:
    from argostranslate import package, translate
    package.install_from_path('en_es.argosmodel') # Dirección donde se encuentre el fichero diccionario (IMPORTANTE)
    installed_languages = translate.get_installed_languages()

    # installed_languages[0] = inglés
    # installed_languages[1] = español

    translation_en_es = installed_languages[0].get_translation(installed_languages[1])
    translate=translation_en_es.translate("This is an offline traduction made by Argostranslate translator") # recogemos el resultado de la traducción

    print(translate)
    print(type(translate))
```

**Listado 2.5** – *Translation OFFLINE in Python with Google Translator*

```
from googletrans import Translator, constants
from argostranslate import package, translate

try:
    translator = Translator()

    # Vamos a realizar la traducción al español:
    translation = translator.translate("This is an online traduction made by Google translator", dest="es")

    prueba = translation.text
    print(prueba)
    print(type(prueba))



except:
    from argostranslate import package, translate
    package.install_from_path('en_es.argosmodel') # Dirección donde se encuentre el fichero diccionario (IMPORTANTE)
    installed_languages = translate.get_installed_languages()

    # installed_languages[0] = inglés
    # installed_languages[1] = español

    translation_en_es = installed_languages[0].get_translation(installed_languages[1])
    translate=translation_en_es.translate("This is an offline traduction made by Argostranslate translator") # recogemos el resultado de la traducción

    print(translate)
    print(type(translate))
```

```
Esta es una traducción offline hecha por el traductor Argostranslate
<class 'str'>
```

**Figure 2.18** – *hibrid translator in Python example (online and offline)*

This is the idea that we are going to apply to our project to create a multilanguage GUI.

### 2.3.3 Responsive GUI

The key elements to make our GUI responsive are the Layouts. Layouts are, so to speak, containers built into QT Designer with intrinsic responsive properties. In such a way that if you place a widget (such as a Dial, a Label...etc) inside a layout, it will acquire the responsive properties of the container (layout) in which it is contained.



#### 2.3.3.1 Easy Example of responsive GUI

We are going to make a simple example to put the aforementioned into practice, in order to get an idea of how it works and as a result of that example you will be able to extrapolate this base to a larger and more complex responsive GUI project.

First of all I am going to create a MainWindow in QT Designer,then we will insert a Grid Layout into the MainWindow and next we will insert a Tab widget (When you drop the Tab on the Grid Layout, the Tab automatically adjusts to the size of the window):

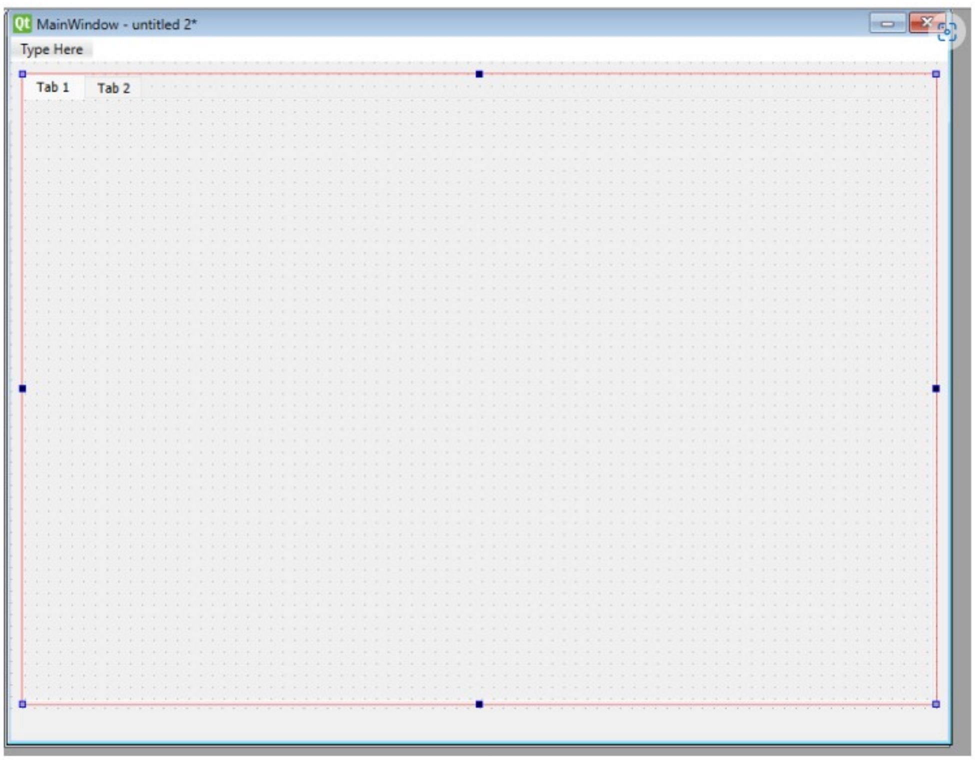



**Figure 2.19** – *Tab Widget into an Grid Layout*

Then we put a Grid Layout in the Tab, select the Tab with the right mouse button and go to: -> Lay out -> Lay Out in a Grid. Obtaining as a result a Layout integrated in the complete Tab. With this we get that all the widgets that we put in the Tab are totally responsive with respect to the size of the GUI window.

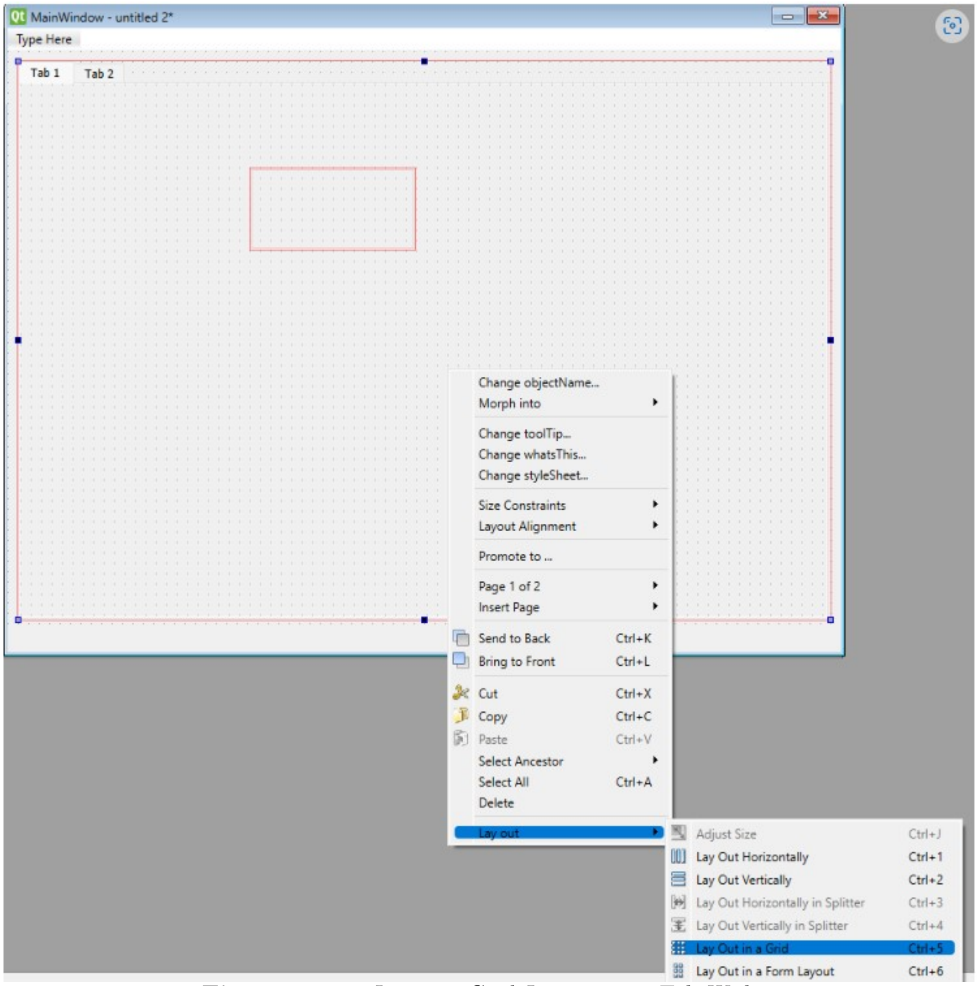



**Figure 2.20** – *Insert a Grid Layout in a Tab Widget*

2

Result of the previous step:

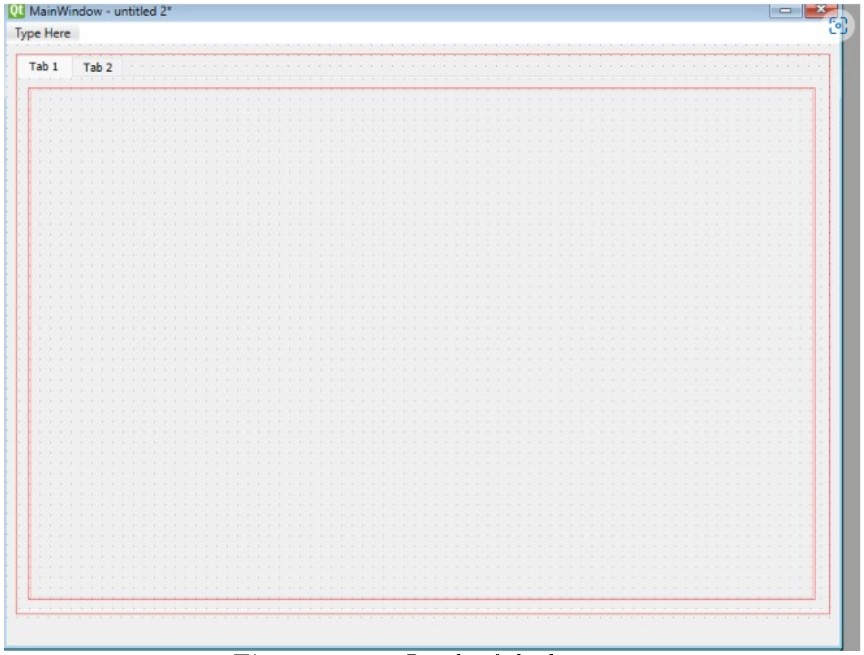



**Figure 2.21** – *Result of the last step*

Then we are going to incorporate elements or widgets to the Tab Layout (Dials, calendar....etc):

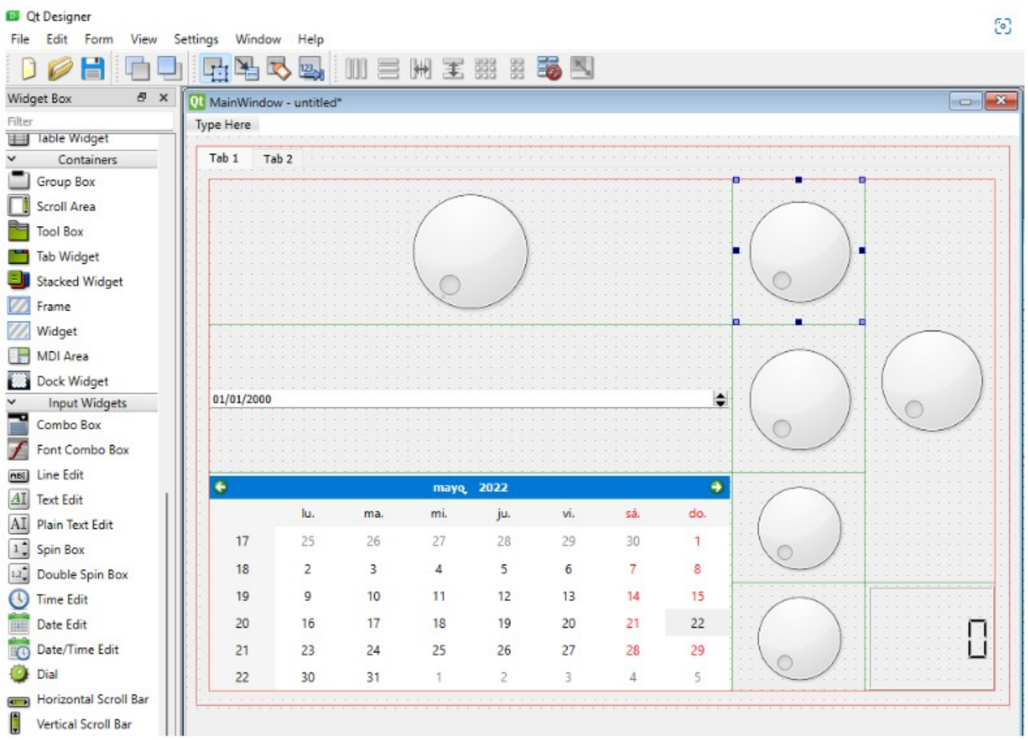



**Figure 2.22** – *Final responsive GUI result example*

You can also make use of other types of Layouts (vertical, horizontal...) I recommend that you use them within a Grid Layout for less hassle, these are used when placing widgets, which are placed vertically or horizontally, and play around with all this so that the elements that are in the GUI are placed more or less where you want them to be placed and that it is in a responsive way.

2

## 2.4 BLAS simulation with API

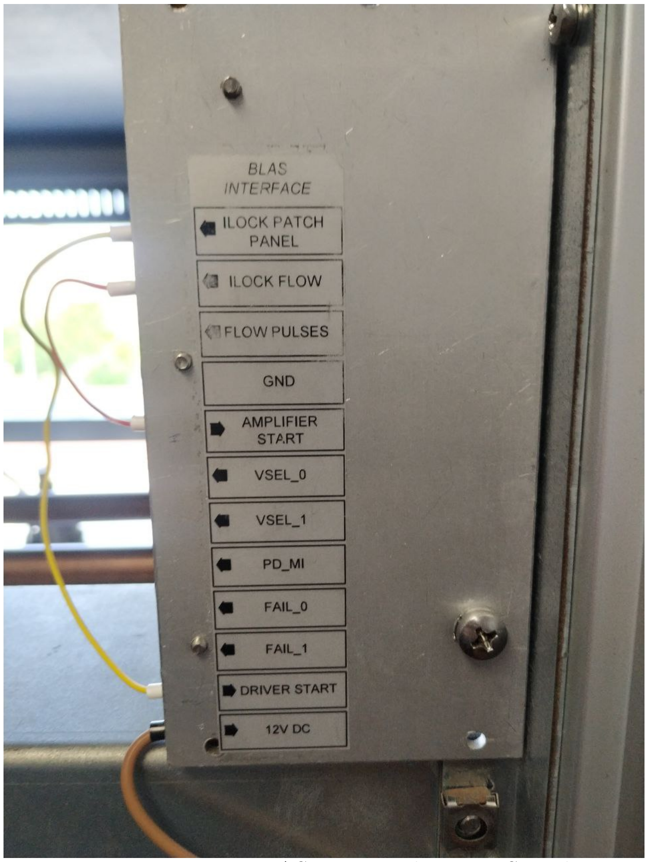


**Figure 2.23** – *BLAS input and output Signals*

Due to the fact that my project partner (Andoni) did not finish the part of the BLAS signals by the time I am doing my TFG, it has not been possible to connect the raspberry tablet with the BLAS input and output signals.
So instead of doing that, my Project tutor (Andrés Roldán) told me to create a simple API to simulate the values of the BLAS input and output signals under normal conditions.

I will use OpenAPI Standard [13]. OpenAPI [17] is a standard for the description of programming interfaces, or API. The OpenAPI specification defines an open and vendor-neutral description format for API services. In particular, OpenAPI can be used to describe, develop, test, and document REST-compliant APIs.

**Figure 2.24** – *OpenAPI logo*

Researching long and hard, I found two Python modules that allowed me to create an API according to the OpenAPI standard and deploy said API to be able to make requests to it:

- **FastApi:** FastAPI is a modern and fast (high performance) web framework for building APIs with Python 3.6+ based on the standard Python type annotations.
- **Uvicorn:** Uvicorn is an ASGI web server implementation for Python.

**(a)** *FastAPI logo* **(b)** *Uvicorn logo*

**Figure 2.25** – *Tools used for API creation in Python*

FastAPI [12] provides automatic documentation that follows the OpenAPI specification, we can just access it as soon as our API has been deployed by accessing the "addresAPI/docs" address of the API.

we need Uvicorn to deploy our API created using FastApi.

### 2.4.1 API example to understand tools operation

```python
# AUTOR: Salvador Jesús Megías Andreu

# file name = apiExample.py

# HOW TO USE TO EXECUTE AND LAUNCH: uvicorn apiExample:api --reload

# If you don't find the uvicorn module installed, use the following:
# HOW TO USE TO EXECUTE AND LAUNCH: python -m uvicorn apiExample:api --reload

# The OPENAPI documentation generated by FastAPI can be accessed directly
# at the following link (as long as you have launched uvicorn with the command above): http://127.0.0.1:8000/docs

from fastapi import FastAPI
import random

api = FastAPI(title= "TFG Salvador Jesús Megías Andreu")

@api.get("/getTemperature")
async def getTemperature():
    return float(random.randint(22*10,50*10)/10)

@api.get("/getFlow")
async def getFlow():
    return float(random.randint(1*10,7*10)/10)
```

**Listado 2.6** – *Example of API to prove Tools (FastApi and Uvicorn)*

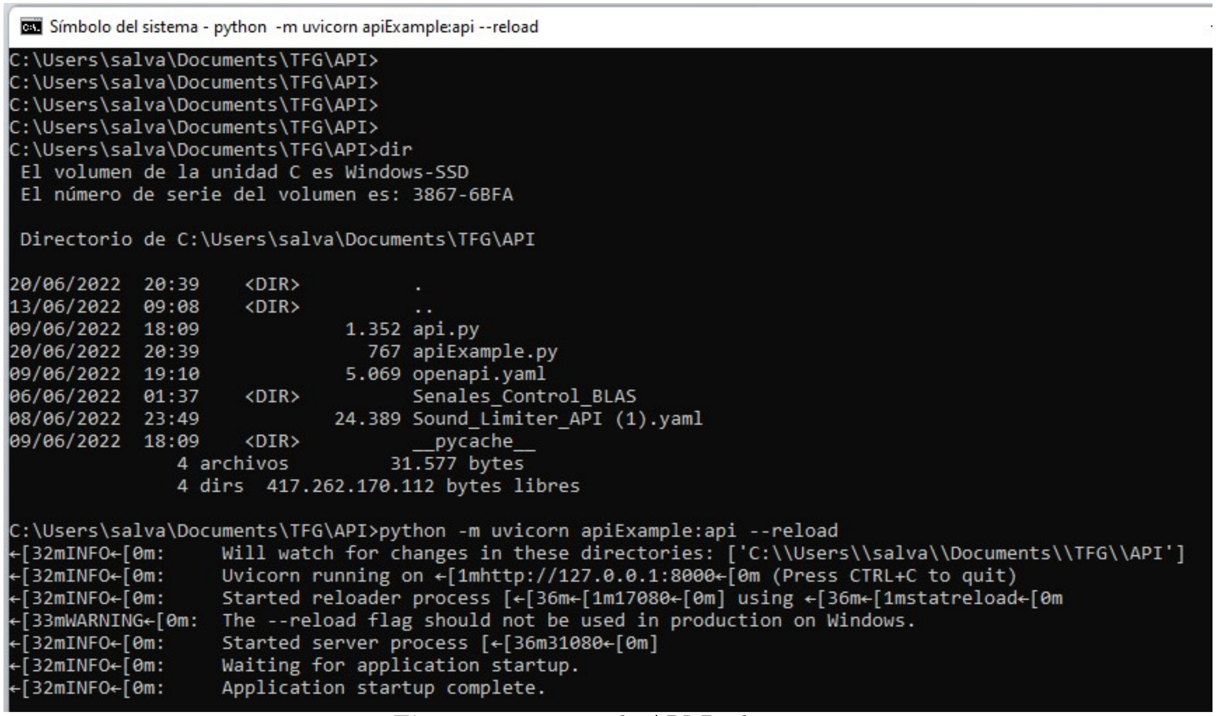



**Figure 2.26** – *sample API Deployment*

Once we have deployed the API, we review the API documentation automatically created by FastApi (with the OpenAPi standard) in our browser and verify that the API works correctly:

In the part rounded with a red circle, you can download the .json file that describes our API:

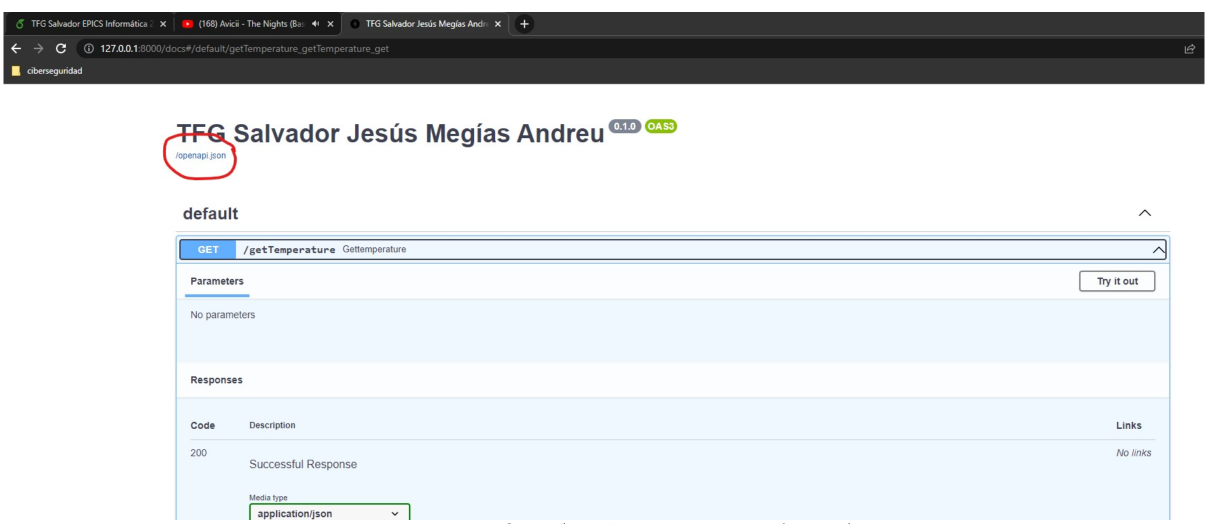



**Figure 2.27** – *OpenApi documentation of our API*

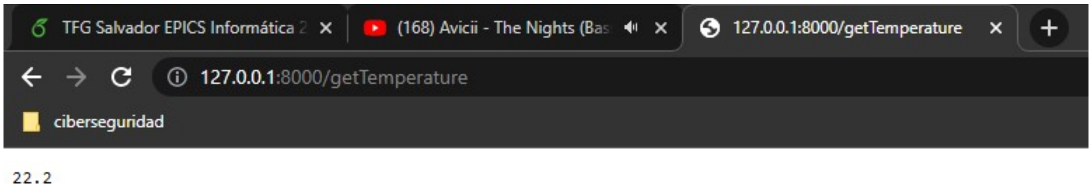



**Figure 2.28** – *API function check*

As we can see, the API works properly.

# Chapter 3

# Project planning

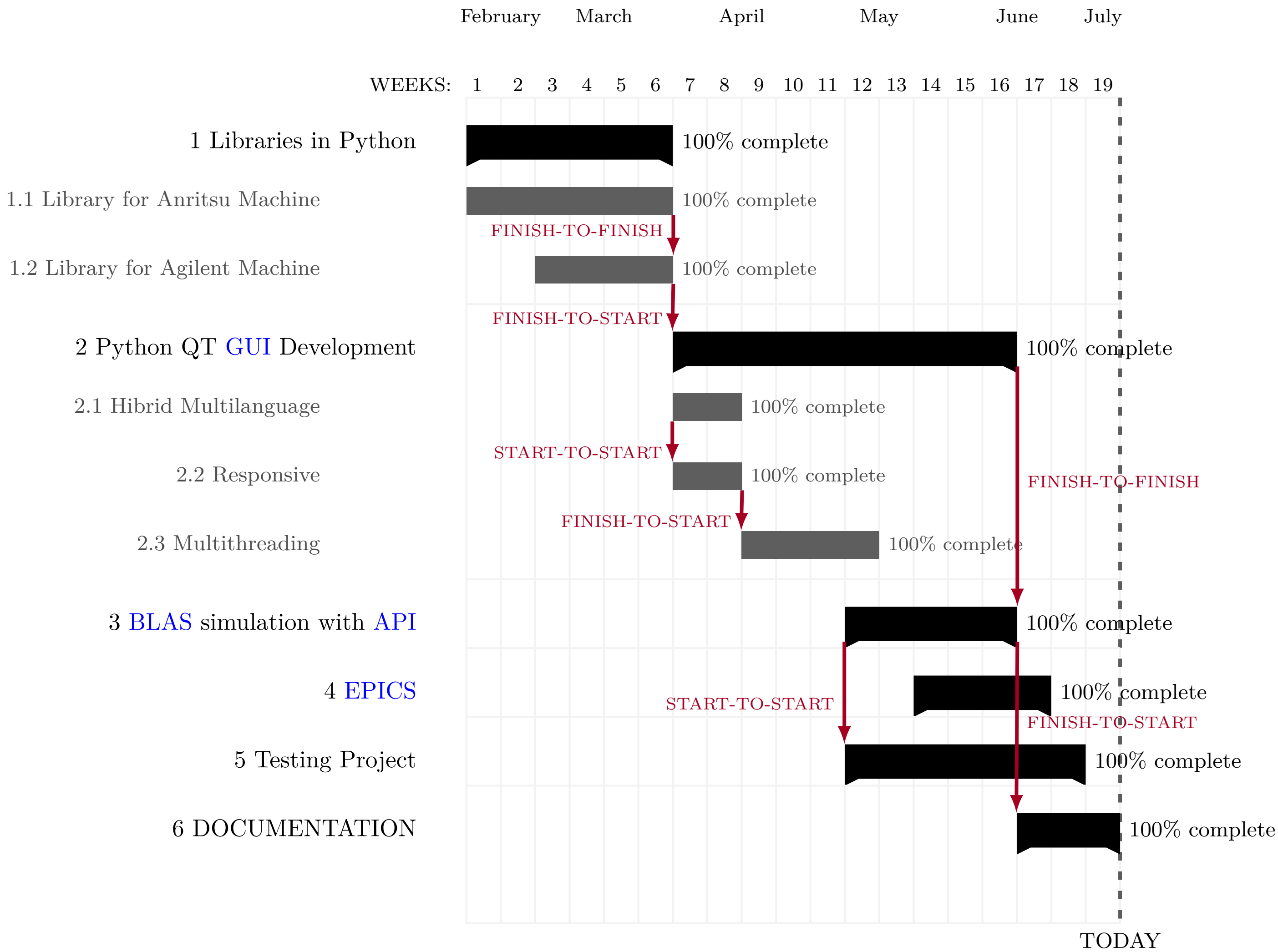

**Figure 3.1** – *Project Planning image*

Once known the background study [2] of the project and before to start it, it's necessary to identify and set at what parts of the system are going to work and for how long we can do it.

To know these details will help us to manage efficiently our work during the design phase of the Communication System, its implementation and the test phase.

To carry out this planning task, we will use the Gantt Diagram to divide the design, development and test time in the most efficient and orderly way possible. We will also make a realistic budget for the total preparation of this project.

## 3.1 Gantt Diagram



## 3.2 Budget

This section will analyse the investment made in terms of costs of material and manpower that has been required in order to perform this master's thesis. This estimation should give an idea of how much it could cost to carry out the same project in an actual engineering company.

For each individual item in the budget, the price of the goods before taxes has been considered. In addition, they have been rounded to the nearest ten. Moreover, manpower has been calculated based on average salary for a junior computer engineer in Granada.

| Thesis Total Budget | |
|---|---|
| Item | Total cost [€] |
| Raspberry Pi LCD Touch Screen 7" | 75 |
| Anritsu MS2830A machine | 28.000 |
| Agilent N9020A machine | 14.250 |
| BLAS non-commercial prototype | 30.000 |
| Switch (24 ports) | 600 |
| Ethernet cable | 300 |
| 3 month junior informatic salary | 3.500 |
| used monitors and towers | 1.000 |
| BUDGET | 77.725 |

Table 3.1 – *GranaSAT – Budget expenses for Thesis*

Chapter 4

# System description and design

## 4.1 System overview

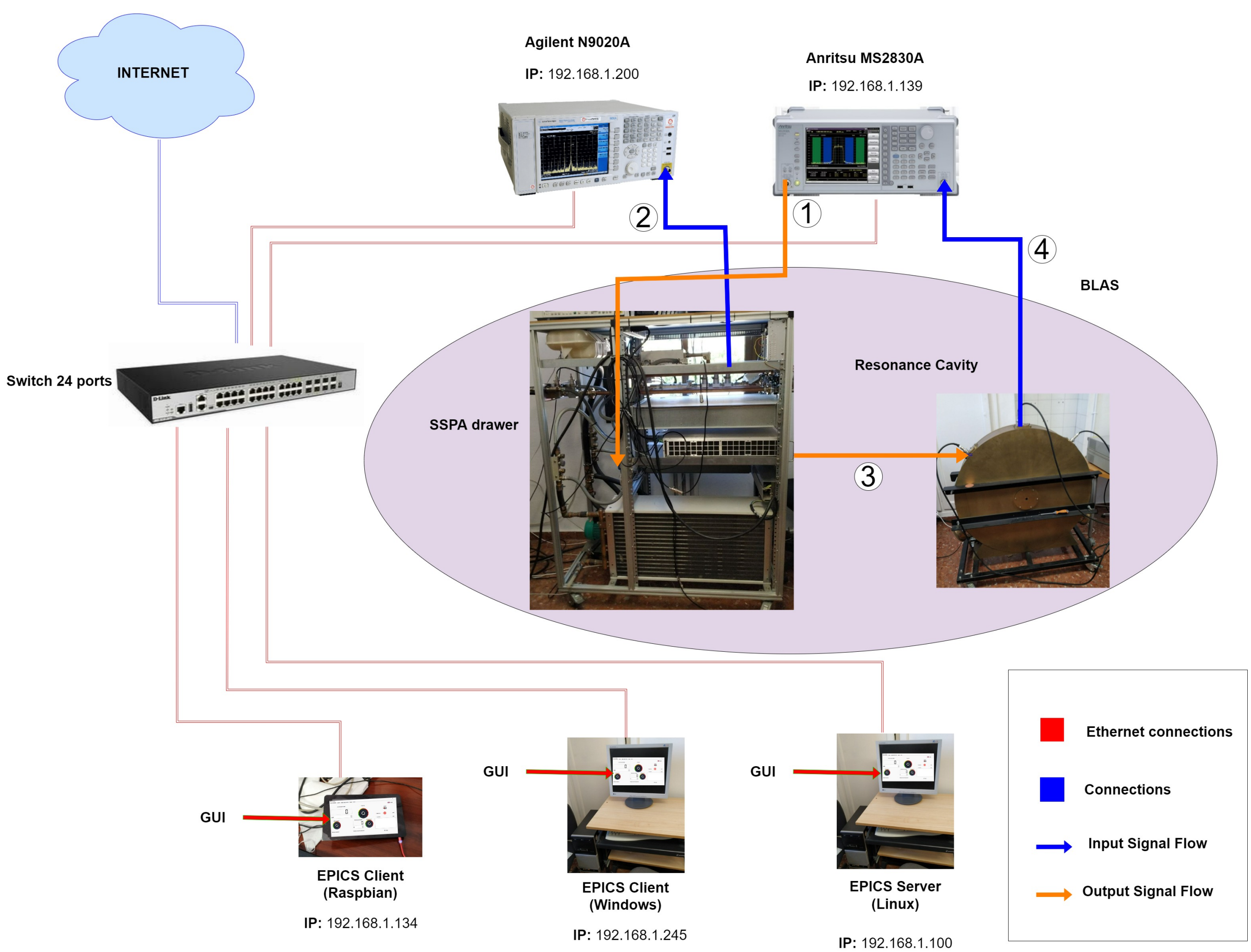



Figure 4.1 – *System overview*

### 4.1.1 System Components

In the previous image (4.1) we can see the distribution of our system as a whole. As we can see, our system is interconnected by Ethernet cables or RF cables, and consists of the following elements:

- Agilent N9020A machine (2.12): It contains the spectrum analyzer mode, which we will use to study and monitor different signals.
- Anritsu MS28030A machine (2.12): It contains the spectrum analyzer mode and the signal generator mode, which we will use respectively to study and monitor different signals and generate RF signals.
- BLAS: It is a prototype designed for learning in the area of particle accelerators and RF amplifiers, currently not marketable, which is composed of the SSPA drawer and the resonance cavity.
  - SSPA drawer: Here the initial signal introduced by the Anritsu MS2830A machine (in signal generator mode) will be amplified, so that once amplified, said signal is conducted to the resonance cavity through an RF cable. It can be monitored by input and output signals (2.4).
  - Resonance cavity: Here the amplified RF signal in the SSPA drawer circulates in a circular direction creating an electric field perpendicular to the magnetic field.
- port-24 Switch: It allows the creation of a sub-network that interconnects and allows all devices in the sub-network to communicate with each other remotely via Ethernet.
- Devices (computers and raspberry tablet): The computer with the Linux operating system has EPICS installed and will be from where we launch our IOC. All the devices have the developed GUI installed to be able to control and monitor the system.

### 4.1.2 System Behaviour

The general behavior of the system is described below in steps:

#### 4.1.2.1 First step

Through the GUI developed using the Python library (D) (2.2) created to manage the Anritsu MS2830A machine, we modify at will the outgoing power created by the machine in signal generator mode, which will be directed to the SSPA drawer.

#### 4.1.2.2 Second step

The RF signal coming into the SSPA drawer is amplified by the two amplifiers it contains.

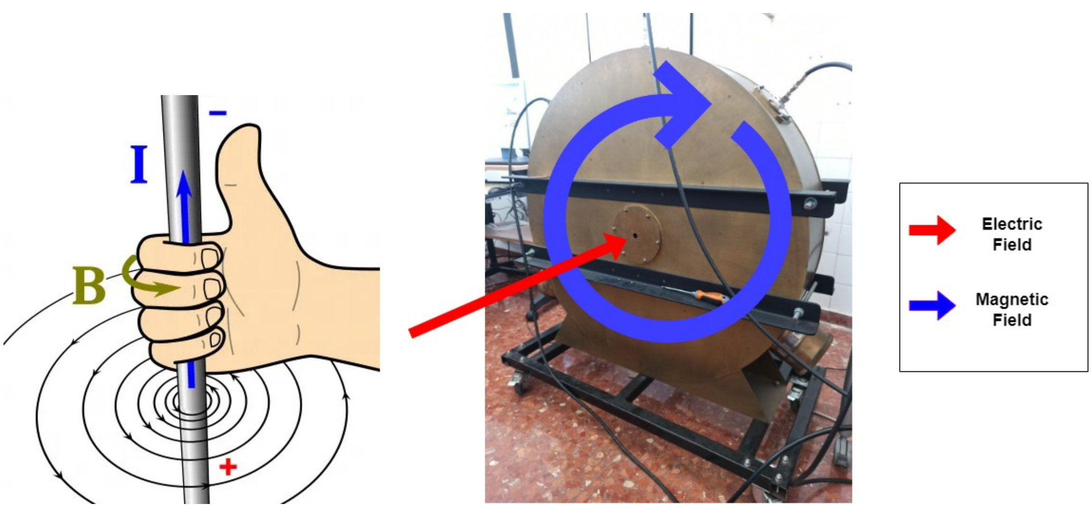
#### 4.1.2.3 Third step

Once the RF signal has been amplified by the SSPA drawer, we can monitor said signal with our GUI using the Python library (C) created to manage the Agilent N9020A machine in spectrum analyzer mode (being able to view the signal itself in our GUI).



#### 4.1.2.4 Fourth step

Simultaneously, while analyzing the outgoing amplified signal with our developed GUI, said RF signal is directed to the resonance cavity, where said circular magnetic field generates an electric field perpendicular to it (this is due to the right hand rule).




(a) *Maxwell's Right Hand Rule (corkscrew rule)* [16]   (b) *Resonance Cavity Functioning*

Figure 4.2 – *Resonance Cavity Explanation*

This generated electric field is the cause of accelerating the positive ions through the small hole located in the central part of the resonance cavity.

#### 4.1.2.5 Fifth step

Finally, with the magnetic field running through the resonance cavity, we make measurements monitoring said signal or magnetic field through our GUI, using the library created in Python (D) to control the Anritsu MS2830A machine in spectrum analyzer mode.

#### 4.1.2.6 BLAS I/0

The idea to handle this was shown in the next section: 2.23

#### 4.1.2.7 EPICS Background Functioning

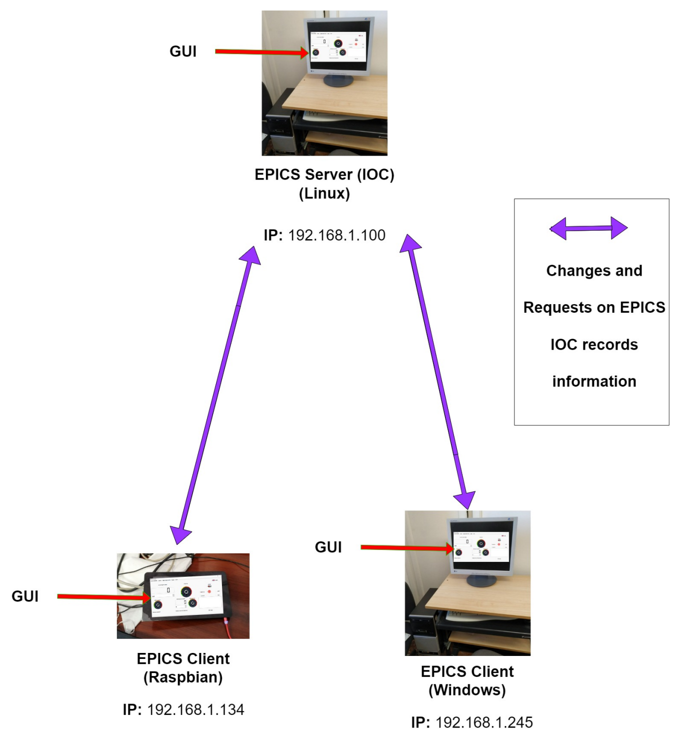



Figure 4.3 – *EPICS (2.1) Flow*

The main idea is that all the GUIs of the system devices are connected to the IOC of the EPICS (2.1) server (installed on the computer with Linux as the operating system), in such a way that when there is a change in the values of any of the GUIs, these changes in said GUI modify the records of the IOC of our EPICS (2.1) server, and that eventually these changes in the IOC records of our EPICS (2.1) server propagate to the rest of the GUIs of the other devices.
With all the above, I intend that the entire system developed as a whole be updated in real time and that said information can be accessed from any part of the subnet.



4

## 4.2 System setup problems

### 4.2.1 Agilent DHCP Problem

Checking the connection of the system devices to the subnet, I realized that the Agilent N9020A machine did not have the DHCP service active, so it could not connect to our subnet and consequently could not obtain an IP from it:

Figure 4.4 – *DHCP Problem*

4

However, the only way to configure the DHCP service is with administrator permissions, so we would have to restart the computer in administrator mode.

Luckily, the username and password of the machine to be able to access in administrator mode are the ones that come by default from the factory: Password for Agilent N9020A machine in administrator mode

Figure 4.5 – *Login Administrator Mode*

Once we are as administrators in the system, we activate the DHCP service:

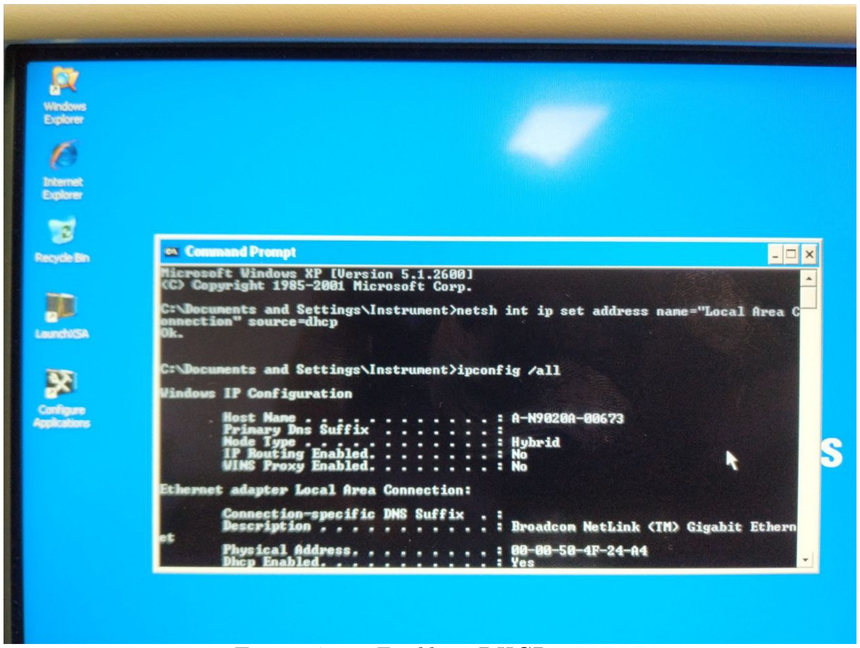



Figure 4.6 – *Enabling DHCP service*

Obtaining an IP within our subnet, and thus, being able to connect to that machine remotely via Ethernet:


Figure 4.7 – *DHCP Problem Solved*

## 4.3 Python Libraries Development

As we have seen in the chapter "Background study", specifically in the section "Python Libraries via Ethernet", we are going to use the PyVISA Python package to write and query SCPI commands to the Agilent N9020A (2.12) and Anritsu MS2830A (2.12) machines, since these machines only understand these SCPI commands if we want to communicate with them remotely.

### 4.3.1 Anritsu MS2830A Python Library

Making use of the programming manual [6] of said machine (2.12) through SCPI commands, I have created the following library in Python capable of handling the signal generator and the spectrum analyzer in the same: Library Code in Python to communicate with Anritsu MS2830A machine

This library in Python is pretty well commented, so any doubt that may arise from it should be resolved with the comments made on the code.

### 4.3.2 Agilent N9020A Python Library

Making use of the programming manual [4] of said machine (2.12) through SCPI commands, I have created the following library in Python capable of handling the spectrum analyzer in the same: Library Code in Python to communicate with Agilent N9020A machine

This library in Python is pretty well commented, so any doubt that may arise from it should be resolved with the comments made on the code.

4

## 4.4 GUI's Development with PyQT

With the knowledge acquired in the section "Python QT", I have developed a GUI with all the requirements (1.4) defined at the beginning of the project.

Due to the great extension that the final version of the development of said GUI supposes (more than 2400 lines of code), I am going to explain small sectors of code to be able to understand this project or part of this project.

These instructions will allow you to get a working copy of the project on your local machine for development and testing purposes: GUI's Installation and Deployment

The entire code of my Bachelor's Thesis can be found here => Salvador Bachelor's Thesis

### 4.4.1 Multilanguage GUI Development

With the idea in mind that we saw in the section "Proposal => Why not?", I have created a hybrid translation system that translates all the text of the GUI when changing languages.
It translates both the text and the ToolTips of the widgets, as well as the messages of the warning windows.



```
# Lists where we will store the widgets and their respective ToolTips so that we can attribute them quickly and not one by one
        # This will allow the scalability of the GUI to be greater
        self.toolTipsObjects = [self.comboBox,self.comboBox_2,self.checkBox, self.radioButton, self.radioButton_2,self.pushButton,self.pushButton_2,self.pushButton_3,self.pushButton_4,self.checkBox_2,self.checkBox_3,self.comboBox_3,self.checkBox_4,self.radioButton_3,self.pushButton_5,self.pushButton_6,self.pushButton_7]
        self.toolTips = ["It changes the language of the GUI","It changes the language of the GUI","It connects and disconnect from the machine","It turns on generator mode","It turns on spectrum mode","It plots machine's data in an image","It sends Generator's data to the machine","It sends Spectrum's data to the machine","It plots machine's data between 100 MHz to 1.5 GHz in an image","turn on and off the Generator","It connects and disconnect from BLAS","It changes the language of the GUI","It connects and disconnect from the machine","It turns on spectrum mode","It sends Spectrum's data to the machine","It plots machine's data in an image","It plots machine's data between 100 MHz to 1.5 GHz in an image"]

        # We store the widgets (labels, buttons and radioButtons) from which we will take their text and thus be able to translate them quickly
        self.allLabels =[self.label_2,self.label_4,self.label_5,self.label_6,self.label_7,self.label_8,self.label_9,self.label_10,self.label_11,self.label_12,self.label_13,self.label_14,self.label_17,self.label_24,self.label_25,self.label_27,self.label_28,self.label_29,self.label_30,self.label_31,self.label_32,self.label_33,self.label_35]
```

```
        self.allPushButtons= [self.pushButton, self.pushButton_2, self
.pushButton_3, self.pushButton_4, self.pushButton_5, self.pushButton_6
, self.pushButton_7]
        self.allRadioButtons = [self.radioButton, self.radioButton_2,
self.radioButton_3]
```

Listado 4.1 – *Lists and objects used for translation*

I have decided to put all the texts, tooltips and messages for popup warning windows in lists, as well as all the widgets to assign all these texts to.
This is done to be able to assign said texts, whether in English or translated into Spanish, to the widgets in loops more quickly.

```
##############################################################################################################################

    # Function that collects the text of all the Widgets of the GUI to be translated in a list
    def getCompleteText(self):

        text = []

        # we collect in text of all the labels of the list "allLabels"
        for i in self.allLabels:
            text.append(i.text())

        # we collect in text of all the pushButtons of the list "allPushButtons"
        for i in self.allPushButtons:
            text.append(i.text())

        # we collect in text of all the radioButtons of the list "allRadioButtons"
        for i in self.allRadioButtons:
            text.append(i.text())

        return text

##############################################################################################################################

    # Function that is responsible for modifying the text of all the Widgets of the GUI to be translated
    def setCompleteText(self, traduccion):

        # we copy de translation list to handle it more easily
        trad = traduccion.copy()

        # we set the text (with the translated text) of all the labels of the list "allLabels"
        for i in self.allLabels:
            i.setText(trad[0])
            trad.pop(0)

        # we set the text (with the translated text) of all the pushButtons of the list "allPushButtons"
        for i in self.allPushButtons:
            i.setText(trad[0])
            trad.pop(0)

        # we set the text (with the translated text) of all the radioButtons of the list "allRadioButtons"
        for i in self.allRadioButtons:
            i.setText(trad[0])
            trad.pop(0)
```

Figure 4.8 – *Functions to get text from GUI's widgets and put text to GUI's widgets*

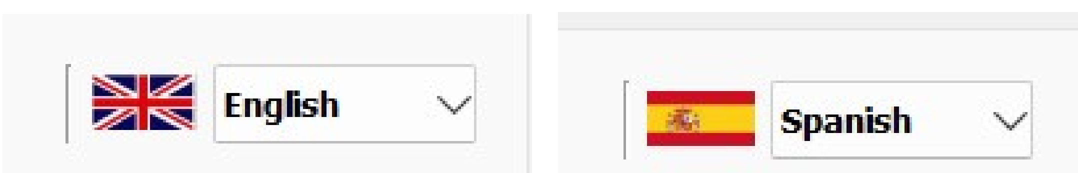
With those functions I can collect all the texts of the widgets that interest me and are in the lists previously shown and also assign new text to said widgets with the translation that can be carried out.

```
#############################################################################################################################

    # Function which declares the id of the comboBox to 1 and calls the "translate" function to perform the full GUI translation
    def idComboBox1(self):
        self.indexComboBox = 1
        self.traduce(self.comboBox)

#############################################################################################################################

    # Function which declares the id of the comboBox to 2 and calls the "translate" function to perform the full GUI translation
    def idComboBox2(self):
        self.indexComboBox = 2
        self.traduce(self.comboBox_2)

#############################################################################################################################

    # Function which declares the id of the comboBox to 3 and calls the "translate" function to perform the full GUI translation
    def idComboBox3(self):
        self.indexComboBox = 3
        self.traduce(self.comboBox_3)
```

Figure 4.9 – *Set the id of the ComboBox from where the translation action was requested*



Because we have 3 comboBox (widgets) in the GUI from where we can change the language of the entire GUI, I have created these functions to set the id of the comboBox from which the request to translate the GUI is being launched, and then call the "traduce" function, passing the id as an attribute.




(a) *comboBox in English* (b) *comboBox in Spanish*

Figure 4.10 – *GUI's comboBox before and after translation*

Finally, we are going to see "traduce" function. This function ensures that, if there is internet, the translation is done online, and if there is no internet, it is done offline.
In addition to that, it is in charge of keeping the 3 comboBoxes of the GUI coordinated together with the images of the corresponding flags, and finally it is in charge of displaying in the GUI all the translated text along all the widgets whose text has been translated:

```
# Function that translates all the text of the GUI widgets (including Tooltips and warning popup messages)
def traduce(self,comboBox):

    traduccion =[]
    tooltips =[]

    # If Spanish has been selected in the comboBox passed as atribute to the function
    if comboBox.currentIndex() == 1:# Let's do the translation into Spanish:
        # If we have internet, the translation will be done with the Google translator
        self.traducedMessagesWindows.clear()
        try:

            translator = Translator()

            # the text of the GUI widgets is translated
            for i in self.text:
                translation = translator.translate(i, dest="es")
                traduccion.append(translation.text)

            # ToolTips are translated
            for i in self.toolTips:
                translation = translator.translate(i, dest="es")
                tooltips.append(translation.text)

            # warning popup messages are translated
            for i in self.messagesWindows:
                translation = translator.translate(i, dest="es")
                self.traducedMessagesWindows.append(translation.text)




        # If not, it will be done offline with argosmodel
        except:


            from argostranslate import package, translate

            package.install_from_path('./language/en_es.argosmodel') # Address where the dictionary file is located (IMPORTANT)
            installed_languages = translate.get_installed_languages
```

```python
()

            # installed_languages[0] = inglés
            # installed_languages[1] = español

            translation_en_es = installed_languages[0].get_translation(installed_languages[1])

            # the text of the GUI widgets is translated
            for i in self.text:
                translate=translation_en_es.translate(i) # we collect the result of the translation
                traduccion.append(translate)

            # ToolTips are translated
            for i in self.toolTips:
                translate=translation_en_es.translate(i) # we collect the result of the translation
                tooltips.append(translate)

            # warning popup messages are translated
            for i in self.messagesWindows:
                translate=translation_en_es.translate(i) # we collect the result of the translation
                self.traducedMessagesWindows.append(translate)


        # Because comboBox widgets react to "currentIndexChanged" signals

        # To keep the 3 GUI comboBoxes coordinated, we must block the signals of the other 2 comboBoxes (in order to modify the index of the other 2 comboBoxes)
        # while modifying the index of the comboBox we are changing. If we don't do it like this,
        # the functions to which the other 2 comboBoxes are connected will be executed, making the translation process redundant and very slow.

        # If the index of the comboBox that has been modified is 1, the signals of the other 2 comboBoxes (2 and 3) are blocked,
        # then the indexes of these comboBoxes are modified and then the signals of these 2 comboBoxes are activated again to continue to be pending the signals to changes.
        if self.indexComboBox == 1:
            self.comboBox_2.blockSignals(True)
            self.comboBox_3.blockSignals(True)

            self.comboBox_2.setCurrentIndex(1)
            self.comboBox_3.setCurrentIndex(1)

            self.comboBox_2.blockSignals(False)
            self.comboBox_3.blockSignals(False)
```

```
        # If the index of the comboBox that has been modified is 2,
the signals of the other 2 comboBoxes (1 and 3) are blocked,
        # then the indexes of these comboBoxes are modified and then
 the signals of these 2 comboBoxes are activated again to continue
to be pending the signals to changes.
        elif self.indexComboBox == 2:
            self.comboBox.blockSignals(True)
            self.comboBox_3.blockSignals(True)

            self.comboBox.setCurrentIndex(1)
            self.comboBox_3.setCurrentIndex(1)

            self.comboBox.blockSignals(False)
            self.comboBox_3.blockSignals(False)

        # If the index of the comboBox that has been modified is 3,
the signals of the other 2 comboBoxes (1 and 2) are blocked,
        # then the indexes of these comboBoxes are modified and then
 the signals of these 2 comboBoxes are activated again to continue
to be pending the signals to changes.
        else:
            self.comboBox.blockSignals(True)
            self.comboBox_2.blockSignals(True)

            self.comboBox.setCurrentIndex(1)
            self.comboBox_2.setCurrentIndex(1)

            self.comboBox.blockSignals(False)
            self.comboBox_2.blockSignals(False)

        # finally we change the flags that are located in front of
the comboBox widgets for Spanish flags
        self.label.setPixmap(QtGui.QPixmap('./images/spanish.png'))
        self.label_23.setPixmap(QtGui.QPixmap('./images/spanish.png'
))
        self.label_34.setPixmap(QtGui.QPixmap('./images/spanish.png'
))

        # Executing these functions, all texts, Tooltips and warning
 popup messages will be changed to Spanish in the GUI.
        self.setCompleteText(traduccion)
        self.setToolTips(tooltips)

    # If Eglish has been selected in the comboBox passed as atribute
 to the function
    else:

        # If the index of the comboBox that has been modified is 1,
the signals of the other 2 comboBoxes (2 and 3) are blocked,
        # then the indexes of these comboBoxes are modified and then
 the signals of these 2 comboBoxes are activated again to continue
to be pending the signals to changes.
```

```
        if self.indexComboBox == 1:
            self.comboBox_2.blockSignals(True)
            self.comboBox_3.blockSignals(True)

            self.comboBox_2.setCurrentIndex(0)
            self.comboBox_3.setCurrentIndex(0)

            self.comboBox_2.blockSignals(False)
            self.comboBox_3.blockSignals(False)

        # If the index of the comboBox that has been modified is 2, the signals of the other 2 comboBoxes (1 and 3) are blocked,
        # then the indexes of these comboBoxes are modified and then the signals of these 2 comboBoxes are activated again to continue to be pending the signals to changes.
        elif self.indexComboBox == 2:
            self.comboBox.blockSignals(True)
            self.comboBox_3.blockSignals(True)

            self.comboBox.setCurrentIndex(0)
            self.comboBox_3.setCurrentIndex(0)

            self.comboBox.blockSignals(False)
            self.comboBox_3.blockSignals(False)

        # If the index of the comboBox that has been modified is 3, the signals of the other 2 comboBoxes (1 and 2) are blocked,
        # then the indexes of these comboBoxes are modified and then the signals of these 2 comboBoxes are activated again to continue to be pending the signals to changes.
        else:
            self.comboBox.blockSignals(True)
            self.comboBox_2.blockSignals(True)

            self.comboBox.setCurrentIndex(0)
            self.comboBox_2.setCurrentIndex(0)

            self.comboBox.blockSignals(False)
            self.comboBox_2.blockSignals(False)


         # finally we change the flags that are located in front of the comboBox widgets for British flags
        self.label.setPixmap(QtGui.QPixmap('./images/uk.png'))
        self.label_23.setPixmap(QtGui.QPixmap('./images/uk.png'))
        self.label_34.setPixmap(QtGui.QPixmap('./images/uk.png'))

        # Executing these functions, all texts, Tooltips and warning popup messages will be changed to English in the GUI.
        self.setCompleteText(self.text)
        self.setToolTips(self.toolTips)
```

Listado 4.2 – *"traduce" function (main function for GUIS's translation)*

### 4.4.2 MultiThreading GUI Development

With the idea in mind that we saw in the section "MultiThreading in PyQt", I have created different classes and functions to handle different tasks for Anritsu (2.12), Agilent (2.12), BLAS and EPICS that require multithreading to prevent the GUI from freezing.
In this section I will explain the development of Multithreading for Agilent, to get an idea of how it works.

```python
##########################################################################################################
########### CLASS FOR MANAGEMENT OF MULTITHREADING FOR AGILENT ##########################################################
##########################################################################################################

# class that is responsible for creating the threads for managing and monitoring the maximum power of the agilent machine (in spectrum analyzer mode)

class ThreadClassAgilent(QtCore.QThread):

    # we define a signal that will handle a float number (the maximum power each time)
    any_signal = QtCore.pyqtSignal(float)

    # in the constructor we define the object of the class that will point through a pointer to the object of the AgilentN9020A class
    # (object created in the GUI class that handles all the functions of the library)
    def __init__(self, parent=None, agilent=None):
        super(ThreadClassAgilent, self).__init__(parent)
        self.is_running = True
        self.agilent = agilent

##########################################################################################################

    # Function that is responsible for calling the getMaxFreqPower function to calculate said value
    # and emits the value of said maximum power through the signal defined at the beginning of the class
    # all this is done indefinitely every 30 seconds
    def run(self):
        while (True):
            power=self.agilent.getMaxFreqPower()
            time.sleep(0.01)
            self.any_signal.emit(power)
            time.sleep(30)

##########################################################################################################

    # Function that is responsible for stopping the thread of the class
    def stop(self):
        self.is_running = False
        #print('Stopping thread...',self.index)
        self.terminate()
```

Figure 4.11 – *Class for Multithreading management in Agilent*

As we can see in this class, we import the object of the agilent class to be able to use its functions, and what I do is that when the thread (object of this class) is created and launched, the maximum power is calculated at that moment via the "getMaxFreqPower()" function of the Agilent class and is emitted as a signal, performing this task in an infinite loop every 30 seconds.

```
##########################################################################################
########### FUNCTIONS FOR HANDLING MULTITHREADING IN AGILENT ##############################
##########################################################################################

    # Function that creates and runs the thread that will monitor the maximum power of the Agilent machine every 30 seconds
    def startAgilent(self):
        self.threadAgilent = ThreadClassAgilent(parent=None, agilent=self.agilent)
        self.threadAgilent.start()

        # The power measured at each instant of time is sent to the function "plotMaxFreqPowerTimeAnritsu"
        self.threadAgilent.any_signal.connect(self.plotMaxFreqPowerTimeAgilent)


#######     #########       #############       ##########          ##########      ############      ##########

    # Function that stops the created thread
    # In principle we do not need a stop of the thread because we want it to work continuously
    def stopAgilent(self):
        self.threadAgilent.stop()

##########################################################################################
##########################################################################################
```

Figure 4.12 – *Functions for Multithreading management in Agilent*

These are the functions with which we handle the previously seen class, these functions are defined within the class of the GUI itself. As we can see, what is done is to create an object of the previous class and we pass the object of the Agilent class as an argument and then we start the thread, and finally we tell it that any signal (a float with the maximum power at each instant in our case) to be issued must be redirected to the "plotMaxFreqPowerTimeAgilent" function as an argument.

```python
# Function that plots the image of the monitoring of the maximum
 power in time (every 30 seconds)
# It receives the parameter of the power from the thread that is
 performing said measurement in the background

def plotMaxFreqPowerTimeAgilent(self,power):

    # we modify the list of units measured so far in time (each unit
  of measurement represents 30 seconds)
    self.maxFreqTimeAgilent.append(self.numeroAgilent)
    self.numeroAgilent += 1

    # we modify the list of powers measured so far in time we modify
  the list of units measured so far adding the power measured at
 that instant to the end of the list
    self.maxPowerTimeAgilent.append(power)

    # and we show in real time the maximum power at that moment and
 the frequency where said power is located in the following labels
    maxfreq=self.agilent.maxfreq

    self.label_39.setText(str(maxfreq)+"MHz")

    self.label_40.setText(str(power) + "dBm")

    # we create an image with the information previously collected
 in lists (as we saw before)

    plot.clf()
    # Label for x-axis
    plot.xlabel("Time Progress (each unit is 30 seconds)")

    # Label for y-axis
    plot.ylabel("Maximum power (dBm)")

    # title of the plot
    plot.title("Monitoring maximun power value progress")

    # we mark the points on the graph
    plot2.scatter(self.maxFreqTimeAgilent, self.maxPowerTimeAgilent)

    #I generate the image with the units of time and the powers
 offered by the machine

    plot.plot(self.maxFreqTimeAgilent, self.maxPowerTimeAgilent)
    plot.savefig('./images/graphAgilentTimeMax.jpg') # Relative
 address where you want the created image to be saved
    #plot.show()

    # If we are connected to EPICS server (to EPICS IOC)
    if self.EPICS_connected:
```

```
            # we modify the registers with the lists of the samples
taken (units of time and list of maximum frequencies over time)
            self.Agilent_SA_unitsTime.put(self.maxFreqTimeAgilent)
            self.Agilent_SA_MaximumPowers.put(self.maxPowerTimeAgilent)



        # And plot it into the graphicsView widget
        self.scene = QtWidgets.QGraphicsScene()
        self.pixmap = QtGui.QPixmap("./images/graphAgilentTimeMax.jpg")
        self.scene.addPixmap(self.pixmap)
        self.graphicsView_4.setScene(self.scene)
```

Listado 4.3 – *function "plotMaxFreqPowerTimeAgilent"*

In this function, what I basically do is that I collect the signals that it is receiving (the maximum powers) and I save them in a list to later create a graph with them and plot it on the screen in the GUI.

Example of "plotMaxFreqPowerTimeAgilent" function output:



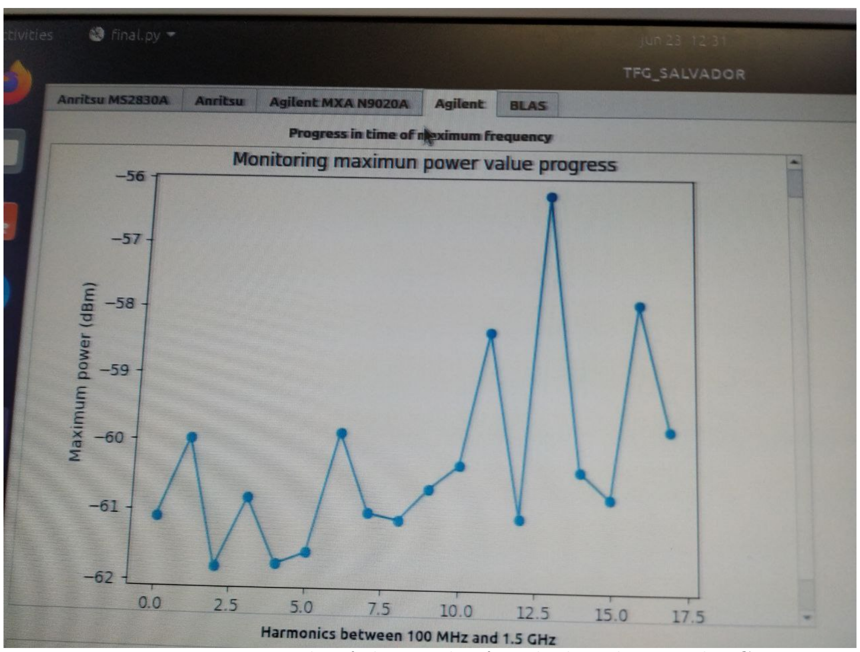



Figure 4.13 – *Example of the result of Multithreading in the GUI*

### 4.4.3 GUI parts

As we saw in the "Responsive GUI" section, I have designed and developed the GUI in several parts using a Tab, among these parts is the section of the GUI that controls the Anritsu machine (2.12) (both the signal generator mode and the spectrum analyzer), also the section of the GUI that controls the Agilent machine (2.12) (the spectrum analyzer mode) and the section that monitors the BLAS input and output signals (2.23) simulated by the developed API:

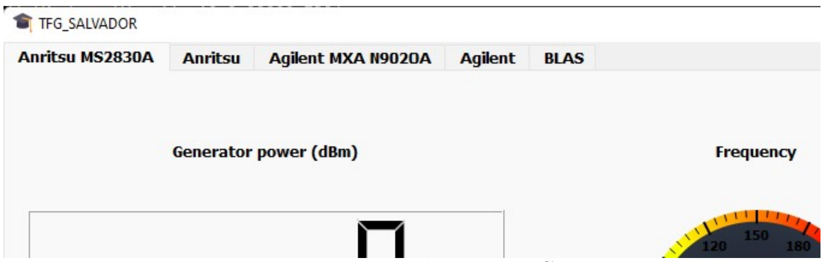


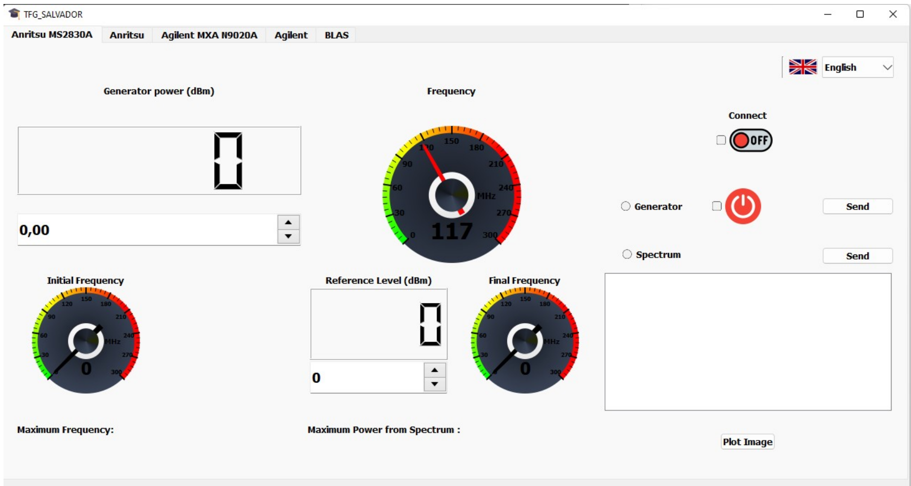

Figure 4.14 – *Tab parts in GUI*

Now we are going to see the part of the constructor of the main class of the GUI where the connections of the functions developed for the control of the GUI with the different widgets that make up the GUI are declared through signals (it is event-driven programming):

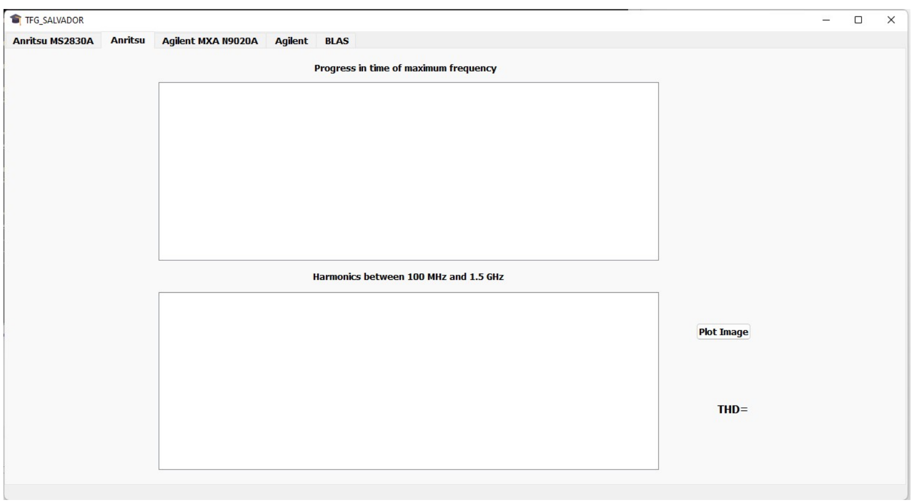
```
# The signals to understand it are like Threads that are waiting for an event to occur to execute the functio
# That is, it's like event-driven programming

# Widget connections with functions for Anritsu using signals

self.comboBox.currentIndexChanged.connect(lambda:self.idComboBox1())
self.checkBox.clicked.connect(lambda:self.conectarAnritsu())
self.radioButton.clicked.connect(lambda:self.setGenerator())
self.radioButton_2.clicked.connect(lambda:self.setSpectrum())
self.pushButton_2.clicked.connect(lambda:self.setParamsGenerator())
self.pushButton_3.clicked.connect(lambda:self.setParamsSpectrumAnritsu())
self.pushButton.clicked.connect(lambda:self.plotImageAnritsu())
self.pushButton_4.clicked.connect(lambda:self.plotLargeSpectrumAnritsu())
self.checkBox_2.clicked.connect(lambda:self.turnOnGenerator())
self.doubleSpinBox.valueChanged.connect(lambda:self.moveDoubleSpinBox())
self.spinBox.valueChanged.connect(lambda:self.moveSpinBox())

# Widget connections with functions for BLAS using signals

self.checkBox_3.clicked.connect(lambda:self.connectBlas())
self.comboBox_2.currentIndexChanged.connect(lambda:self.idComboBox2())

# Widget connections with functions for Agilent using signals

self.pushButton_6.clicked.connect(lambda:self.plotImageAgilent())
self.spinBox_2.valueChanged.connect(lambda:self.moveSpinBoxAgilent())
self.radioButton_3.clicked.connect(lambda:self.setSpectrumAgilent())
self.pushButton_5.clicked.connect(lambda:self.setParamsSpectrumAgilent())
self.checkBox_4.clicked.connect(lambda:self.conectarAgilent())
self.pushButton_7.clicked.connect(lambda:self.plotLargeSpectrumAgilent())
self.comboBox_3.currentIndexChanged.connect(lambda:self.idComboBox3())
```

Figure 4.15 – *Connect widgets to functions using signals*

#### 4.4.3.1 GUI development for Anritsu




(a) *Part 1 for Anritsu in GUI*




(b) *Part 2 for Anritsu in GUI*

Figure 4.16 – *part of the GUI developed for the control and management of the Anritsu machine*

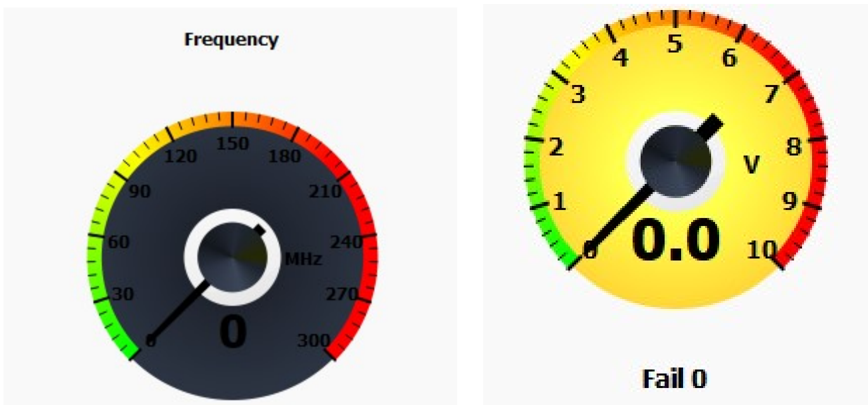
In this part, what has been done is to develop functions that in turn call the functions developed in the library that we have created to remotely control the Anritsu MS2830A machine (D), and that are capable of collecting information from the GUI widgets and modifying the state of the GUI widgets accordingly.

An example of this could be the following function, which collects the values of the widgets related to the Anritsu generator in the GUI and sends them to the machine, modifying its parameters remotely (if the signal generator mode is not selected, a warning window will appear notifying it to you in the language selected, this has been done in all functions to make the GUI as robust as possible to failures).

And if we are connected to the EPICS IOC, the values of the registers are modified.

```python
########################################################################################################################

    # Function that establishes the selected parameters of the signal generator in the
    # GUI sending them to the machine modifying the parameters of said machine

    def setParamsGenerator(self):

        # If the signal generator mode is selected
        if self.radioButton.isChecked():
            # we send the parameters established in the GUI to the Anritsu machine
            self.anritsu.setParamsGenerator(self.doubleSpinBox.value(),int(self.dial_2.value))

            # If we are connected to EPICS server (to EPICS IOC)
            if self.EPICS_connected:

                # we modify the registers of the power and frequency of the signal generator
                self.Anritsu_SG_Power.put(self.doubleSpinBox.value())
                self.Anritsu_SG_Frequency.put(int(self.dial_2.value))
                # and we tell the record that monitors all changes in Anritsu, that there has been a change
                self.Anritsu_SomeValueChanged.put(1)

        # If the signal generator mode is not selected, it is notified by a pop-up window, a warning message.
        else:
            if self.comboBox.currentIndex() == 1:
                QtWidgets.QMessageBox.warning(None,self.traducedMessagesWindows[0],self.traducedMessagesWindows[3])
            else:
                QtWidgets.QMessageBox.warning(None,self.messagesWindows[0],self.messagesWindows[3])
```

Figure 4.17 – *Example of function developed for Anritsu in the GUI*




(a) *Gauge type 1* (b) *Gauge type 2*

Figure 4.18 – *Gauges used in the GUI*

The dials or Gauges used in the GUI [11] have not been developed by me, I have only made use of them, modifying them slightly to make my GUI as modern as possible.
The real author of these Gauges is Stefan Holstein.
The code for it can be found at the following link: Stefan Holstein Code (@KhamisiKibet)

4

#### 4.4.3.2 (THD) Total Harmonic Distortion

In the second image of (4.16), We can see the section for harmonics between 100 MHz and 1.5 GHz (This part is developed in Anritsu and Agilent):

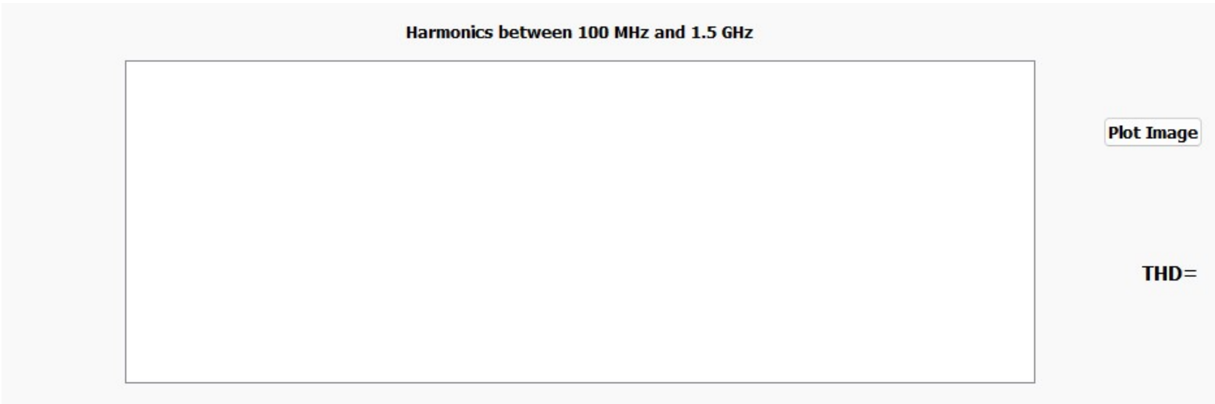



Figure 4.19 – *harmonics part in GUI*

Here we know a relevant concept, the THD [18], which is a useful technique to analyze any non-linear behavior of a system.

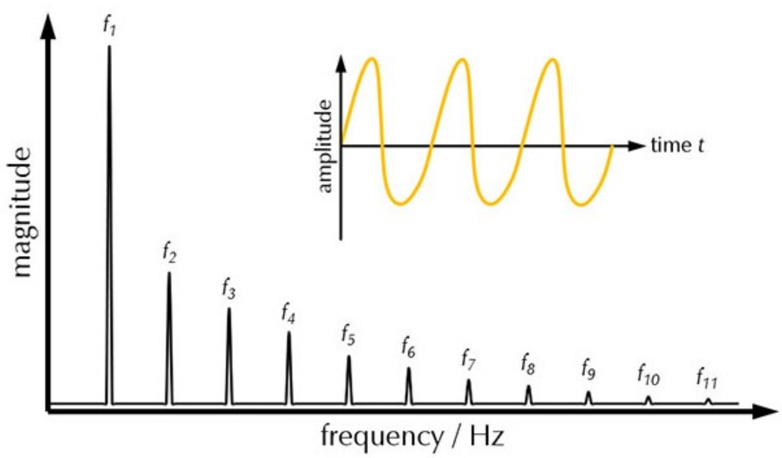



Figure 4.20 – *THD concept*

Making use of the following formulas (both are actually the same) we can calculate the THD:

$$\text{THD} = \frac{\sum \text{Potencia de los armonicos}}{\text{Potencia de la frecuencia fundamental}} = \frac{P_0 + P_1 + P_2 + P_3 + \cdots + P_N}{P_1}$$

Figure 4.21 – *THD formula in spanish*

$$THD(w) = \frac{\sum_{n=2}^{10} Y_n^2(w)}{Y1}.100\% \tag{4.4.1}$$

Basically we have to find throughout the entire signal (from 100 MHz to 1.5 GHz in our case) all the Harmonics of it.
Once we have found all the harmonics, we must pass them from dBm to W and apply the formula seen above.

For this we will use the following functions developed:

```
# Function that is responsible for converting all the elements of a list from Dbm to W
def convertDbmToW(self,datos):

    datosW=[]
    datosMW=[]

    # here we convert from Dbm to mW
    for i in datos:
        datosMW.append(float(10**(i/10)))

    # here we convert from mW to W
    for i in datosMW:
        datosW.append(float(i)/1000)

    #print(datosW)
    return datosW

##################################################################################################

# Function that calculates the THD value
def calculateTHD(self,datos):
    return float(sum(datos)/max(datos))
```

Figure 4.22 – *Functions in python to calculate the THD*

Giving as a final result in the GUI if we press the "Plot Image" button the following:

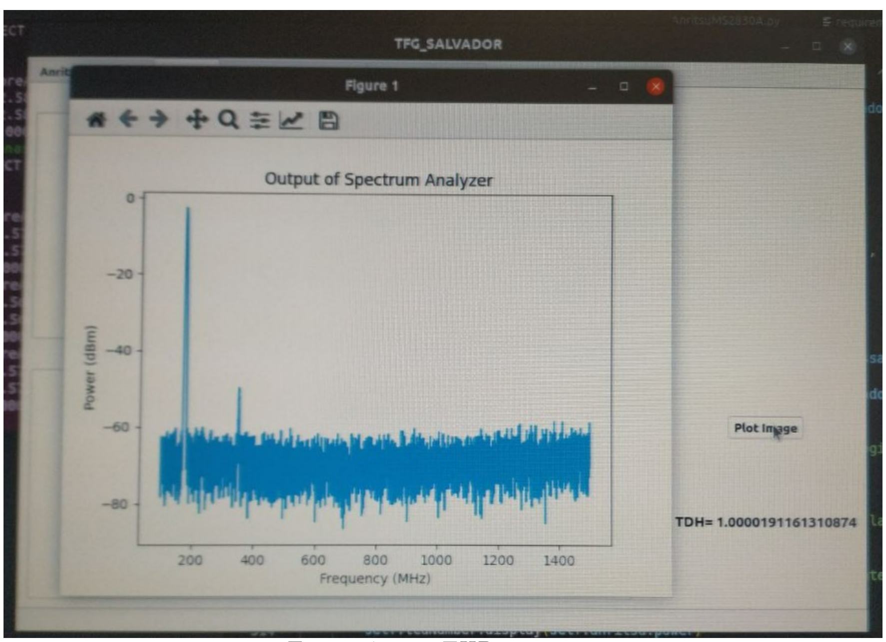
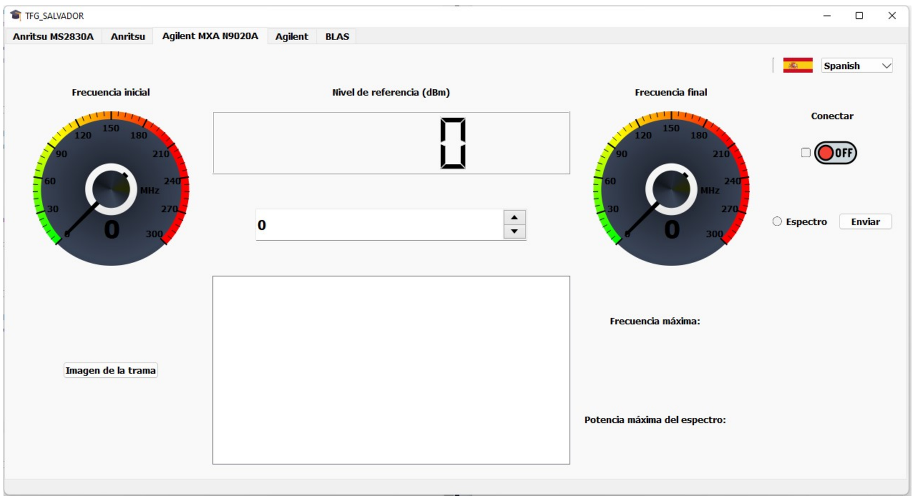
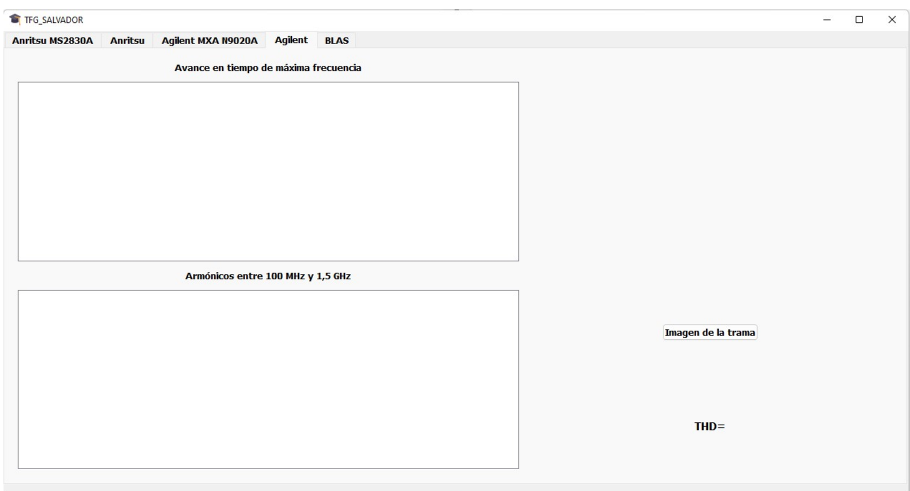



Figure 4.23 – *THD executing*

#### 4.4.3.3 GUI development for Agilent




(a) *Part 1 for Agilent in GUI*




(b) *Part 2 for Agilent in GUI*

Figure 4.24 – *part of the GUI developed for the control and management of the Agilent machine*

In this part, what has been done is to develop functions that in turn call the functions developed in the library that we have created to remotely control the Agilent N9020A machine (C), and that are capable of collecting information from the GUI widgets and modifying the state of the GUI widgets accordingly.

An example of this could be the following function, which selects the spectrum analyzer mode of the Agilent machine remotely (if we are not connected to the Agilent machine, a warning window will appear notifying it to you in the language selected, this has been done in all functions to make the GUI as robust as possible to failures).

And if we are connected to the EPICS IOC, the value of the register related to the instrument choosed in Agilent machine is modified.



```
###################################################################################################################

    # Function that selects the spectrum analyzer mode in the machine

    def setSpectrumAgilent(self):

        # if we are connected to the machine
        if self.conectadoAgilent:
            # we select the spectrum analyzer mode
            self.agilent.setSpectrum()

            # If we are connected to EPICS server (to EPICS IOC)
            if self.EPICS_connected:
                # we modify the register of the chosen mode to "SA" (spectrum analyzer)
                self.Agilent_Instrument_Choosed.put('SA')
                # and we tell the record that monitors all changes in Agilent, that there has been a change
                self.Agilent_SomeValueChanged.put(1)

        # if we are not connected to the machine, it is notified by a pop-up window, a warning message.
        else:
            if self.comboBox.currentIndex() == 1:
                QtWidgets.QMessageBox.warning(None,self.traducedMessagesWindows[0],self.traducedMessagesWindows[1])
            else:
                QtWidgets.QMessageBox.warning(None,self.messagesWindows[0],self.messagesWindows[1])

            # and we reset the radioButton
            self.radioButton_3.setCheckable(False)
            self.radioButton_3.setCheckable(True)
```

Figure 4.25 – *Example of function developed for Agilent in the GUI*

#### 4.4.3.4 GUI development for BLAS Simulation

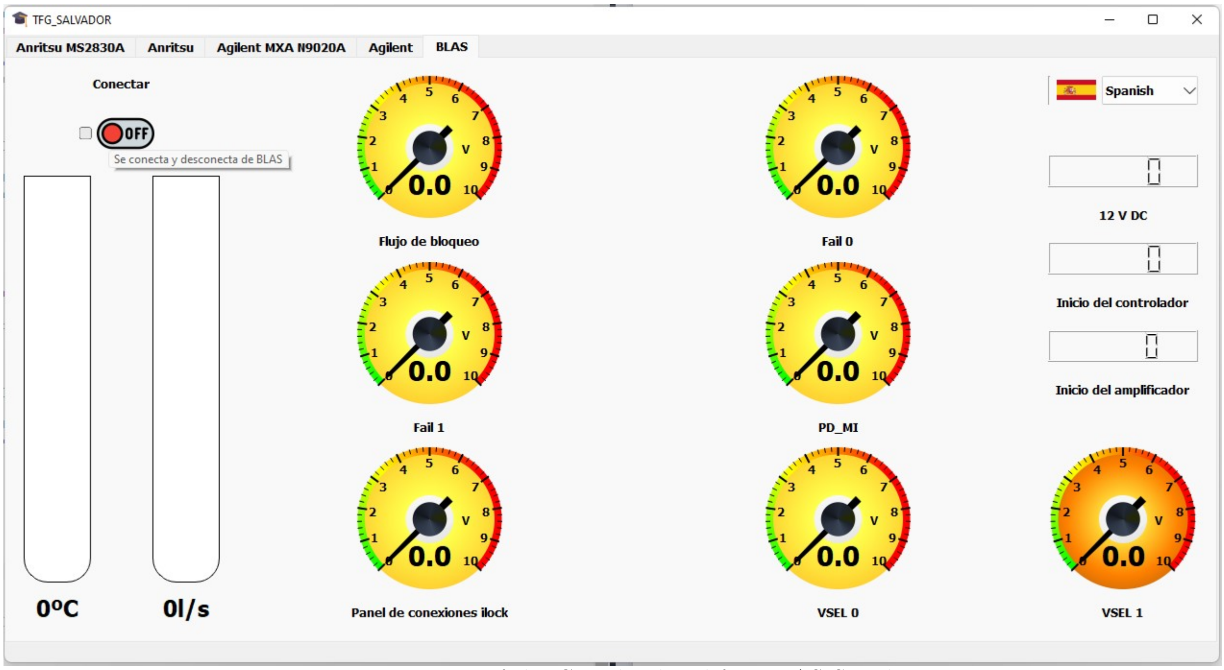



Figure 4.26 – *part of the GUI developed for BLAS Simulation*

Part of the development regarding the BLAS in the GUI is explained here: API Development.

Now, we will see the main function associated with BLAS in the GUI:

```
# Function that receives the information (monitored by the created thread) of the simulated BLAS input and output signals through an API.
def setBlas(self,datos):

    # If the length of the list that we receive with the information is 0, it means that it has not connected with the API, so we show a warning popup
    if len(datos)==0:
        if self.comboBox.currentIndex() == 1:
            QtWidgets.QMessageBox.warning(None, self.traducedMessagesWindows[0], self.traducedMessagesWindows[5])
        else:
            QtWidgets.QMessageBox.warning(None, self.messagesWindows[0], self.messagesWindows[5])

        # we stop the thread that should be monitoring the API
```

```
            self.stopBlas()
            # we uncheck the checkBox
            self.checkBox_3.setChecked(False)

            # and we change the icon back to OFF
            self.checkBox_3.setIcon(QtGui.QIcon("./images/switch-off.png
"))

        # if we have successfully connected to the API and received
information
        else:

            # we transform the received information to the corresponding
 data types

            temp = float(datos[0])
            #flujo = float(datos[1])
            ilockFlow= float(datos[2])
            fail0= float(datos[3])
            fail1= float(datos[4])
            pd_mi= float(datos[5])
            ilockPatchPanel=float(datos[6])
            vsel0 = float(datos[7])
            vsel1 = float(datos[8])

            # and we send this information to the GUI by modifying the
value and state of the GUI widgets

            self.progressBar.setValue(int(float(datos[0])))
            self.label_3.setText(str(datos[0])+"Celsius Degrees")
            self.progressBar_2.setValue(int(float(datos[1])))
            self.label_16.setText(str(datos[1])+"l/s")
            self.dial_4.updateValue(ilockFlow)
            self.dial_5.updateValue(fail0)
            self.dial_6.updateValue(fail1)
            self.dial_7.updateValue(pd_mi)
            self.dial_10.updateValue(ilockPatchPanel)
            self.dial_9.updateValue(vsel0)
            self.dial_8.updateValue(vsel1)

            # If we are connected to EPICS server (to EPICS IOC)
            if self.EPICS_connected:

                ## we modify the registers referring to the values
offered by the BLAS API
                self.BLAS_waterTemperature.put(temp)
                self.BLAS_waterFlow.put(float(datos[1]))
                self.BLAS_IlockFlow.put(ilockFlow)
                self.BLAS_Fail0.put(fail0)
                self.BLAS_Fail1.put(fail1)
                self.BLAS_PD_MI.put(pd_mi)
                self.BLAS_IlockPatchPanel.put(ilockPatchPanel)
                self.BLAS_VSEL0.put(vsel0)
```

```
            self.BLAS_VSEL1.put(vsel1)



        # HERE I DEFINE DIFFERENT SITUATIONS THAT MAY OCCUR DEPENDING ON THE VALUES OF THE INFORMATION RECEIVED, SHOWING POP-UP WARNING WINDOWS IF THERE IS ANY PROBLEM IN THE BLAS

        if temp< 65 and ilockFlow== 12 and fail0==0 and fail1==0 and pd_mi==7.6 and ilockPatchPanel==12 and vsel0 >= 0 and vsel0 <= 5 and vsel1 >= 0 and vsel1 <= 5:
            self.lcdNumber_3.display(12)
            self.lcdNumber_4.display(12)
            self.lcdNumber_5.display(12)

        elif temp< 65 and ilockFlow== 12 and fail0==5 and fail1==5 and pd_mi==0 and ilockPatchPanel==0 and vsel0 >= 0 and vsel0 <= 5 and vsel1 >= 0 and vsel1 <= 5:
            self.lcdNumber_3.display(12)
            self.lcdNumber_4.display(0)
            self.lcdNumber_5.display(12)

            if self.comboBox.currentIndex() == 1:
                QtWidgets.QMessageBox.warning(None, self.traducedMessagesWindows[0], self.traducedMessagesWindows[6])
            else:
                QtWidgets.QMessageBox.warning(None, self.messagesWindows[0], self.messagesWindows[6])

        elif temp< 65 and ilockFlow== 0 and fail0==5 and fail1==5 and pd_mi==7.6 and ilockPatchPanel==12 and vsel0 >= 0 and vsel0 <= 5 and vsel1 >= 0 and vsel1 <= 5:
            self.lcdNumber_3.display(12)
            self.lcdNumber_4.display(12)
            self.lcdNumber_5.display(0)

            if self.comboBox.currentIndex() == 1:
                QtWidgets.QMessageBox.warning(None, self.traducedMessagesWindows[0], self.traducedMessagesWindows[7])
            else:
                QtWidgets.QMessageBox.warning(None, self.messagesWindows[0], self.messagesWindows[7])

        elif temp< 65 and ilockFlow== 12 and fail0==5 and fail1==5 and pd_mi==0 and ilockPatchPanel==12 and vsel0 >= 0 and vsel0 <= 5 and vsel1 >= 0 and vsel1 <= 5:
            self.lcdNumber_3.display(12)
            self.lcdNumber_4.display(12)
            self.lcdNumber_5.display(12)

            if self.comboBox.currentIndex() == 1:
                QtWidgets.QMessageBox.warning(None, self.
```

```
traducedMessagesWindows[0], self.traducedMessagesWindows[8])
            else:
                QtWidgets.QMessageBox.warning(None, self.
messagesWindows[0], self.messagesWindows[8])

        elif temp > 65 and ilockFlow== 12 and fail0==0 and fail1==0
and pd_mi==7.6 and ilockPatchPanel==12 and vsel0 >= 0 and vsel0 <=
5 and vsel1 >= 0 and vsel1 <= 5:
            self.lcdNumber_3.display(12)
            self.lcdNumber_4.display(12)
            self.lcdNumber_5.display(12)

            if self.comboBox.currentIndex() == 1:
                QtWidgets.QMessageBox.warning(None, self.
traducedMessagesWindows[0], self.traducedMessagesWindows[9])
            else:
                QtWidgets.QMessageBox.warning(None, self.
messagesWindows[0], self.messagesWindows[9])
```

Listado 4.4 – *Function that receives the data from the API and modifies the state of the GUI widgets*

## 4.5 API Development

As we saw in the section "BLAS simulation with API", I have used the OpenAPI standard to develop through the Python FastApi package, the API with which I will simulate the values offered by the BLAS under normal conditions.
Code: API developed to simulate BLAS data under normal conditions.

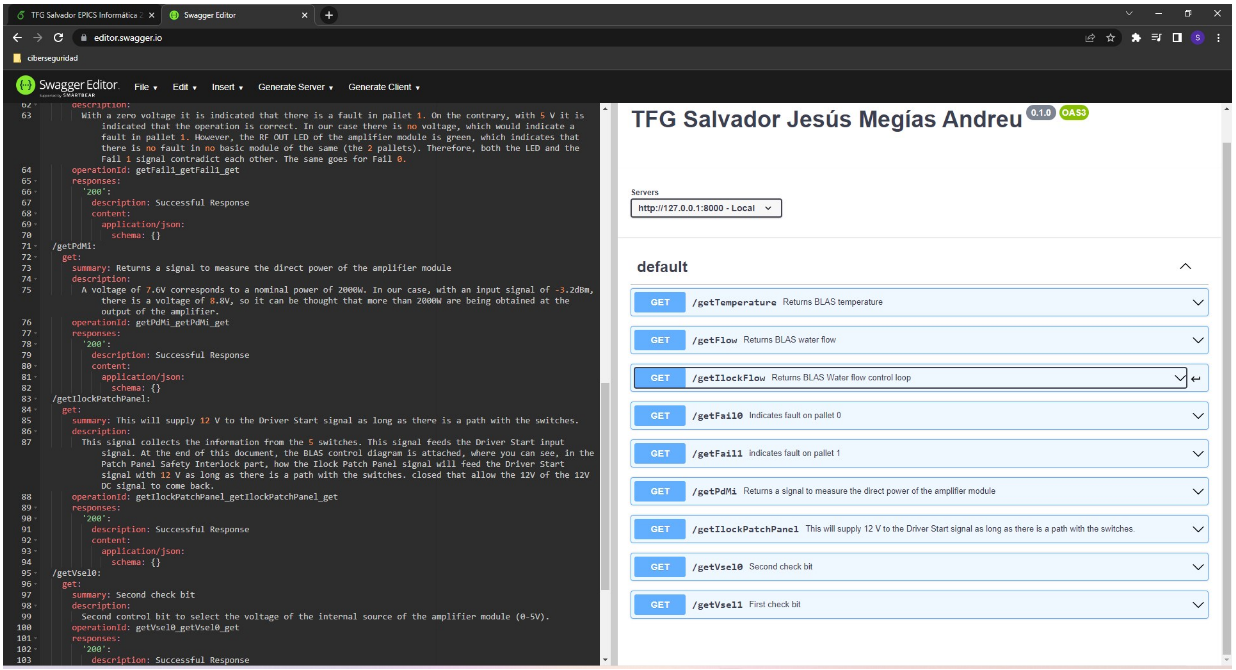


Figure 4.27 – *API defined with OpenAPI standard in Swagger "openapi.yaml"*

And I have developed a class that uses multithreading (4.4.2) to make requests to the API every 5 seconds, and the values of the widgets referring to the BLAS are updated:

```



# class that is responsible for creating the threads for managing and monitoring the values offered by the API that simulates the input and output signals of the BLAS
class ThreadClassBlas(QtCore.QThread):

    # we define a signal that will handle a list (with the values of the BLAS)
    any_signal = QtCore.pyqtSignal(list)

    # in the constructor we are going to define the addresses where we can extract each of the BLAS values from the API
    def __init__(self, parent=None):
        super(ThreadClassBlas, self).__init__(parent)
        self.is_running = True
        self.apiURL = "http://127.0.0.1:8000/"
        self.temperature = self.apiURL + "getTemperature"
        self.flow = self.apiURL + "getFlow"
        self.iLockFlow = self.apiURL + "getIlockFlow"
        self.fail0 = self.apiURL + "getFail0"
        self.fail1 = self.apiURL + "getFail1"
        self.PdMi = self.apiURL + "getPdMi"
        self.ilockPatchPanel = self.apiURL + "getIlockPatchPanel"
        self.vsel0 = self.apiURL + "getVsel0"
        self.vsel1 = self.apiURL + "getVsel1"

#
##########################################################################################


    # Function that is responsible, through the Python REQUESTS package, for taking the values directly from the API and saving them in a list,
    # to finally emit said list through the signal defined at the beginning of the class.
    # all this is done indefinitely every 5 seconds
    def run(self):
        while (True):
            datosBlas=[]
            try:
                r = requests.get(self.temperature)
                datosBlas.append(r.text)

                r = requests.get(self.flow)
                datosBlas.append(r.text)

```

```
                r = requests.get(self.iLockFlow)
                datosBlas.append(r.text)

                r = requests.get(self.fail0)
                datosBlas.append(r.text)

                r = requests.get(self.fail1)
                datosBlas.append(r.text)

                r = requests.get(self.PdMi)
                datosBlas.append(r.text)

                r = requests.get(self.ilockPatchPanel)
                datosBlas.append(r.text)

                r = requests.get(self.vsel0)
                datosBlas.append(r.text)

                r = requests.get(self.vsel1)
                datosBlas.append(r.text)



                time.sleep(0.01)
                self.any_signal.emit(datosBlas)
                time.sleep(5)

            # in case of any failure, an empty list is issued
            except:
                self.any_signal.emit(datosBlas)
                time.sleep(5)

# ##########################################################################################


    # Function that is responsible for stopping the thread of the class
    def stop(self):
        self.is_running = False
        #print('Stopping thread...',self.index)
        self.terminate()
```

Listado 4.5 – *Class that handles BLAS Multithreading*

```
#########################################################################################################################
############ FUNCTIONS FOR HANDLING MULTITHREADING IN THE BLAS ##########################################################
#########################################################################################################################

    # Function that creates and runs the thread that will monitor the values collected from the
    # API that simulates the input and output signals of the BLAS every 5 seconds
    def startBlas(self):
        self.threadBlas = ThreadClassBlas(parent=None)
        self.threadBlas.start()

        # The values (in a list) measured at each instant of time is sent to the function "setBlas"
        self.threadBlas.any_signal.connect(self.setBlas)

#######     #########       #############       ##########          ##########      ############      ##########


    # Function that stops the created thread
    def stopBlas(self):
        self.threadBlas.stop()


#########################################################################################################################
#########################################################################################################################
```
Figure 4.28 – *Functions that handle the class that handles BLAS Multithreading*

## 4.6 EPICS Development

With the knowledge acquired in the section EPICS Study, I have created an EPICS IOC that when deployed on the EPICS server, will contain all the records (with a little description of each one) used in the project and those records that may be useful or interesting to use in future versions of the project carried out by other students in Granasat: Final EPICS IOC Records.

Once we have the EPICS server deploying the EPICS IOC seen above, we can make use of all the EPICS IOC registers through Python, being able to modify or consult the register values from anywhere in the subnet, which we are going to take advantage of to ensure that all the GUIs that are running from any device from any point of the subnet are synchronized in real time by using the updated values of the EPICS IOC registers.

From the GUI as soon as it is displayed, it tries to connect through Python to all the EPICS IOC Records seen above, if it is successful, the variable "self.EPICS_connected" will be True and we can continue to keep the GUI synchronized with EPICS, if not, the GUI will function normally as a single individual out of sync with EPICS:

```
#Function that is responsible for connecting to all the PVs (process
  variables), that is, to all the records of our EPICS IOC
def connectEPICS(self):

    #try to connect to them
    try:

        # connects with the EPICS IOC records referring to the
 Anritsu MS2830A machine in the GUI
        self.Anritsu_SomeValueChanged = epics.PV('Anritsu:
 SomeValueChanged')

        self.Anritsu_SPECT_InitialFrequency = epics.PV('Anritsu:
 SPECT_InitialFrequency')
        self.Anritsu_SPECT_FinalFrequency= epics.PV('Anritsu:
 SPECT_FinalFrequency')
        self.Anritsu_SPECT_ReferenceLevel= epics.PV('Anritsu:
 SPECT_ReferenceLevel')
        self.Anritsu_SPECT_MaximumFrequency = epics.PV('Anritsu:
 SPECT_MaximumFrequency')
        self.Anritsu_SPECT_MaximumPower = epics.PV('Anritsu:
 SPECT_MaximumPower')
        self.Anritsu_SPECT_THD = epics.PV('Anritsu:SPECT_THD')
        self.Anritsu_SPECT_Frequencies = epics.PV('Anritsu:
 SPECT_Frequencies')
        self.Anritsu_SPECT_Powers = epics.PV('Anritsu:SPECT_Powers')
        self.Anritsu_SPECT_unitsTime = epics.PV('Anritsu:
 SPECT_unitsTime')
        self.Anritsu_SPECT_MaximumPowers = epics.PV('Anritsu:
 SPECT_MaximumPowers')
        self.Anritsu_SG_Power = epics.PV('Anritsu:SG_Power')
        self.Anritsu_SG_Frequency = epics.PV('Anritsu:SG_Frequency')
        self.Anritsu_SG_State = epics.PV('Anritsu:SG_State')
```

```
        self.Anritsu_Instrument_Choosed = epics.PV('Anritsu:Instrument_Choosed')



        # connects with the EPICS IOC records referring to the Agilent N9020A machine in the GUI
        self.Agilent_SomeValueChanged = epics.PV('Agilent:SomeValueChanged')

        self.Agilent_InitialFrequency = epics.PV('Agilent:InitialFrequency')
        self.Agilent_FinalFrequency = epics.PV('Agilent:FinalFrequency')
        self.Agilent_ReferenceLevel = epics.PV('Agilent:ReferenceLevel')
        self.Agilent_MaximumFrequency = epics.PV('Agilent:MaximumFrequency')
        self.Agilent_MaximumPower = epics.PV('Agilent:MaximumPower')
        self.Agilent_THD = epics.PV('Agilent:THD')
        self.Agilent_SA_Frequencies = epics.PV('Agilent:SA_Frequencies')
        self.Agilent_SA_Powers = epics.PV('Agilent:SA_Powers')
        self.Agilent_SA_unitsTime = epics.PV('Agilent:SA_unitsTime')
        self.Agilent_SA_MaximumPowers = epics.PV('Agilent:SA_MaximumPowers')
        self.Agilent_Instrument_Choosed = epics.PV('Agilent:Instrument_Choosed')


        # connects with the EPICS IOC records referring to the BLAS simulated values in the GUI
        self.BLAS_SomeValueChanged = epics.PV('BLAS:SomeValueChanged')

        self.BLAS_waterTemperature = epics.PV('BLAS:waterTemperature')
        self.BLAS_waterFlow = epics.PV('BLAS:waterFlow')
        self.BLAS_IlockFlow = epics.PV('BLAS:IlockFlow')
        self.BLAS_Fail0 = epics.PV('BLAS:Fail0')
        self.BLAS_Fail1 = epics.PV('BLAS:Fail1')
        self.BLAS_PD_MI = epics.PV('BLAS:PD_MI')
        self.BLAS_IlockPatchPanel = epics.PV('BLAS:IlockPatchPanel')
        self.BLAS_VSEL0 = epics.PV('BLAS:VSEL0')
        self.BLAS_VSEL1 = epics.PV('BLAS:VSEL1')
        self.BLAS_12V_DC = epics.PV('BLAS:12V_DC')
        self.BLAS_DriverStart = epics.PV('BLAS:DriverStart')
        self.BLAS_AmplifierStart = epics.PV('BLAS:AmplifierStart')

        # we initialize the records that monitor changes in all records to 0
        # 0 = there are no changes in IOC records
        # 1 = there are changes in IOC records
```

```
        self.Anritsu_SomeValueChanged.put(0)
        self.Agilent_SomeValueChanged.put(0)
        self.BLAS_SomeValueChanged.put(0)

        # EPICS has been correctly connected
        self.EPICS_connected = True


        # If those record has value None, it means that we have not connected with the EPICS IOC
        # so we generate an exception so that the code of the exception is executed
        if self.Anritsu_SomeValueChanged.value == None or self.Agilent_SomeValueChanged.value == None:
            raise

    # if there are any exceptions or errors, nothing is done and the GUI is run by itself, without connecting to any EPICS server. This is done so that it can work anyway.
    except:
        self.EPICS_connected = False
```

Listado 4.6 – *Connecting from the GUI to EPICS IOC Records*

4

Once connected to EPICS, I have developed classes and functions that use Multithreading (4.4.2) to keep the following registers monitored at all times:

- self.Anritsu_SomeValueChanged
- self.Agilent_SomeValueChanged
- self.BLAS_SomeValueChanged

```python
# class that is responsible for creating the threads for managing and monitoring any changes to the EPICS IOC records
# in real time. A 1 will be issued if there is any change in any record of the Agilent machine
# If there is no change, nothing will be issued

class ThreadClassEPICSAgilent(QtCore.QThread):
    any_signal = QtCore.pyqtSignal(int)

    # In the constructor we define a variable that will point through a pointer
    # to the PV (record) "Agilent_SomeValueChanged" to monitor if any EPICS IOC record modifies its value.
    def __init__(self, parent=None, Agilent_SomeValueChanged=None):
        super(ThreadClassEPICSAgilent, self).__init__(parent)
        self.is_running = True
        self.Agilent_SomeValueChanged = Agilent_SomeValueChanged

#######     #########       #############       ##########          ##########      ############      ##########

    # Function that is responsible for emitting a 1 through the signal defined at the beginning of the class
    # if the value of the PV (record) "Agilent_SomeValueChanged" of our EPICS IOCS is 1
    # (that is, if there has been any record related to the Agilent machine has been modified)
    # all this is done indefinitely all the time
    def run(self):
        while (True):
            if int(self.Agilent_SomeValueChanged.value) == 1:
                recordAgilentChanged=1
                self.any_signal.emit(recordAgilentChanged)


#######     #########       #############       ##########          ##########      ############      ##########

    # Function that is responsible for stopping the thread of the class
    def stop(self):
        self.is_running = False
        #print('Stopping thread...',self.index)
        self.terminate()
```

Figure 4.29 – *Class that controls Agilent Multithreading with EPICS*

```python
##########################################################################################################################

    # Function that creates and executes the thread that will monitor if there are changes or not in the EPICS IOC records referring to Agilent
    def startEPICS_Agilent(self):
        self.threadEPICS_Agilent = ThreadClassEPICSAgilent(parent=None , Agilent_SomeValueChanged=self.Agilent_SomeValueChanged)
        self.threadEPICS_Agilent.start()

        # If there have been changes in any Agilent record, a signal with value 1 is sent to the "setEpicsAgilent" function so that it is executed
        self.threadEPICS_Agilent.any_signal.connect(self.setEpicsAgilent)

#######     #########       #############       ##########          ##########      ############      ##########

    # Function that stops the created thread
    def stopEPICS_Agilent(self):
        self.threadEPICS_Agilent.stop()
```

Figure 4.30 – *Functions that handle the class that handles Agilent Multithreading with EPICS*

If, for example, the register self.Agilent_SomeValueChanged changes to 1, it will mean that there have been changes in the values of the registers referring to Agilent in the GUI (for example, the value of the final frequency has changed), and therefore after this, the values of the registers referring to Agilent from the EPICS IOC will be taken and they will be assumed as their own in the GUI in the part referring to Agilent, thus ensuring that the GUI is always synchronized with EPICS (all this will be done by the thread created through the class developed and assigned to this mission to prevent the GUI from freezing).

```python
# Function that if it receives a signal with value 1 (meaning that there have been changes in the IOC records referring to the Agilent machine),
# it modifies in the GUI those records that have changed in the EPICS IOC
def setEpicsAgilent(self,changed):

    # If the value of the signal it receives is 1 (some record's value changed)
    if changed == 1:

        # we check what values have changed in the EPICS with respect to those of Agilent's attributes in the GUI
        # and those records that have changed, we modify the values of the Agilent attributes in the GUI with these new values from the EPICS IOC
        # and we also modify the state of the corresponding widgets in the GUI accordingly

        if int(self.Agilent_InitialFrequency.value) != int(self.agilent.inicialFreq):
            self.agilent.inicialFreq = int(self.Agilent_InitialFrequency.value)
            self.dial_11.value = self.agilent.inicialFreq

        if int(self.Agilent_FinalFrequency.value) != int(self.agilent.finalFreq):
            self.agilent.finalFreq = int(self.Agilent_FinalFrequency.value)
            self.dial_12.value = self.agilent.finalFreq

        if int(self.Agilent_ReferenceLevel.value) != int(self.agilent.referenceLevel):
            self.agilent.referenceLevel = int(self.Agilent_ReferenceLevel.value)
            self.spinBox_2.setValue(self.agilent.referenceLevel)

        if str(self.agilent.instAgilent) not in str(self.Agilent_Instrument_Choosed.value) :

            if "SA" in str(self.Agilent_Instrument_Choosed.value):
                self.agilent.instAgilent = 'SA'
                self.radioButton_3.setChecked(True)

        # When we finish checking everything and changing what corresponds
        # we put the record that monitors changes in Agilent's records back to 0 (that is, there are no pending changes anymore)
        self.Agilent_SomeValueChanged.put(0)
```

Figure 4.31 – *Function that is executed when the value emitted by the thread is 1 (that is, there were values that changed in EPICS)*

And just like this part explained about EPICS with Agilent, the rest of the GUI works to achieve the goal of being in sync with other GUIs deployed on other devices on the same subnet through EPICS.

4

# Chapter 5

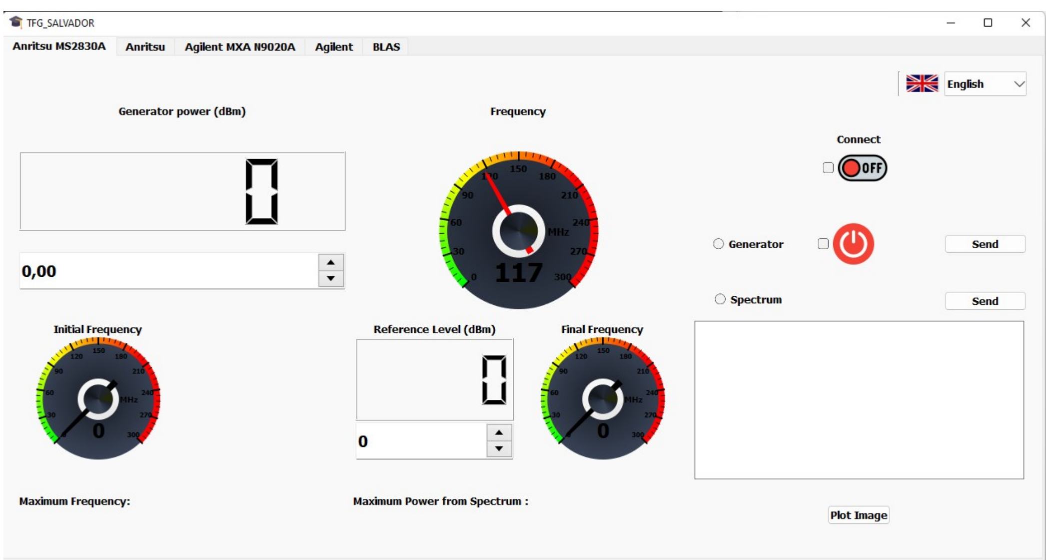
# System testing

This chapter shows various videos showing the operation, as a whole, of all the parts of the GUI developed throughout the project, including a video of the GUI running on a Raspberry tablet.

To be able to display the videos in this PDF, please use Adobe Acrobat DC and click the images of this chapter.
Another option to display the videos is visiting my github: Project Material.

## 5.1 Testing GUI's translation

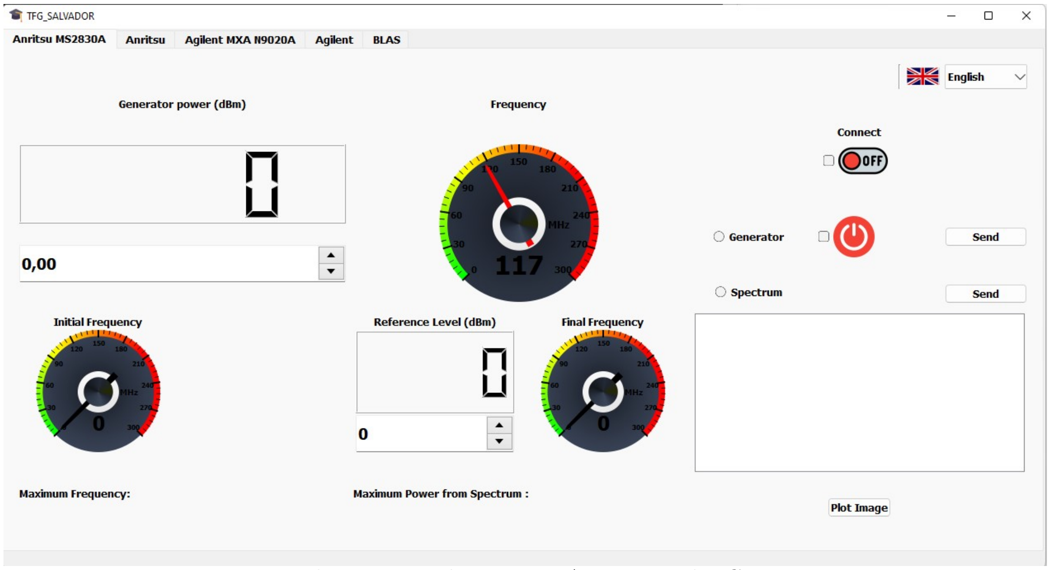
It can be found also in YouTube: GUI's translation video




Video 5.1 – *Video testing GUI's translation*

## 5.2 Testing Anritsu in the GUI

It can be found also in YouTube: Anritsu Working in the GUI



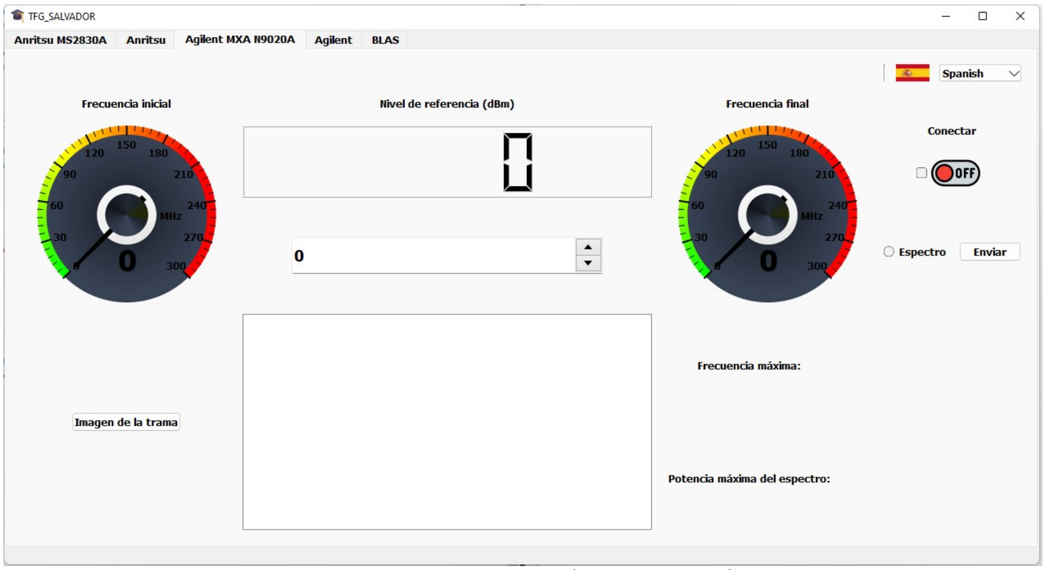

Video 5.2 – *Video testing Anritsu in the GUI*

It can be found also in YouTube: Anritsu machine working remotely

Video 5.3 – *Video showing the Anritsu machine being remotely controlled*

## 5.3 Testing Agilent in the GUI

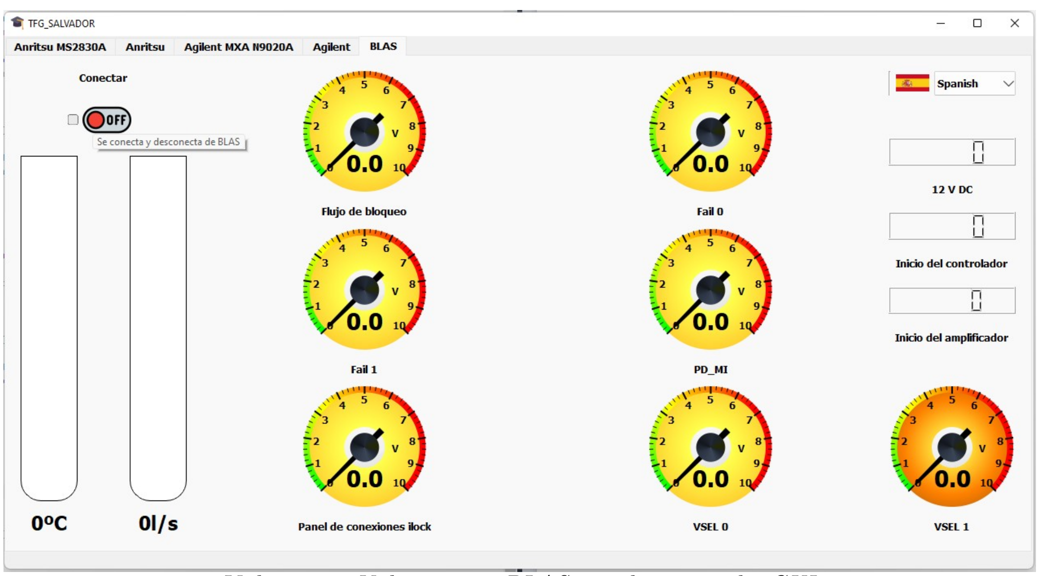
It can be found also in YouTube: Agilent Working in the GUI




Video 5.4 – *Video testing Agilent in the GUI*

## 5.4 Testing BLAS simulation in the GUI

It can be found also in YouTube: BLAS Simulation Working in the GUI




Video 5.5 – *Video testing BLAS simulation in the GUI*

## 5.5 Testing the GUI in Raspberry Tablet

It can be found also in YouTube: GUI working in Tablet Raspberry

Video 5.6 – *Video testing the GUI in Raspberry Tablet*

5

# Chapter 6

# Conclusions and future work

## 6.1 Conclusions

Once we have finished the entire process and design of the system and we have verified that it really works, we can conclude that the expectations and requirements imposed from the beginning of the project are met, and we can conclude that the result obtained is an intuitive and functional system, capable of to communicate with various machines in the system and able to be synchronized through EPICS with other GUIs deployed in the system (on the same subnet).

Now I will give a personal conclusion or assessment of the project, from the perspective of a student.

This project is a project that has required a lot of time and effort and a great breadth of knowledge acquired to carry it out. Much of this knowledge has been acquired throughout the development of the project, and this fact is a factor that I value very much, since I love to learn, and above all, I love to learn useful things, and I consider that these concepts are useful in face my future.

The fact that it was a project that has a real and marketable projection for the future is and was undoubtedly an incentive to put all my enthusiasm and commitment into its correct development.

## 6.2 Future work

This project was born already bearing in mind that it would not be possible to finish it completely due to the high complexity of the project as a whole, of which mine is a part of it. The nature of the remaining work is varied, because it requires, apart from computer science, from other areas of engineering different from mine.

So I will try to list things that, in my opinion, could be done or should be done to improve the system:

- The multilanguage factor of the GUI could be improved, since the offline translation depends on the system architecture being 64 bits (it is a requirement for the installation of the Python translation package argostranslate).
- As soon as the study on the input and output signals of the BLAS (2.23) is definitively finished by my colleagues in the laboratory, they should be connected by wiring to the Raspberry tablet (5.6) in order to monitor the signals of the BLAS itself, and thus not have to simulate these signals through an API as if has had to be done in the present project
- Improve the EPICS system to take into account the signals collected directly from the BLAS, the code referring to this task would have to be modified.
- It could be tried to put this entire system in a Docker or Kubernete to avoid the heavy task of solving possible version incompatibilities when downloading dependencies and forgetting what operating system we are using. It would simply be to download the Docker or Kubernete with this system developed and deploy it.

# Bibliography


[1] Epics new website. https://epics-controls.org/.

[2] Epics old website. https://epics.anl.gov/.

[3] Harwani, B. M. *Qt5 Python GUI Programming Cookbook: Building responsive and powerful cross-platform applications with PyQt.* Packt Publishing Ltd, 2018.

[4] Agilent n9020a programming manual. http://www.ue.eti.pg.gda.pl/~bpa/lab_klb/pdfs/n9000a-6.pdf.

[5] Agilent website. https://www.agilent.com/.

[6] Anritsu ms2830a programming manual. https://dl.cdn-anritsu.com/en-au/test-measurement/files/Manuals/Operation-Manual/MS269xA/MS269xA_2830A_40A_50A_SpectrumAnalyzer_Remote_Manual_e_57_0.pdf.

[7] Anritsu website. https://www.anritsu.com.

[8] Btesa website. https://www.btesa.com/.

[9] Porto, A. Desarrollo de un sistema de control remoto utilizando epics. Bachelor's thesis, Universidad del País Vasco, 2020.

[10] REXUS/BEXUS. Rocket & balloon experiments for university students, 2013. http://www.rexusbexus.net/ Accessed July-2016.

[11] Code developed by stefan holstein for the creation of the gui gauges. https://github.com/KhamisiKibet/QT-PyQt-PySide-Custom-Widgets/blob/main/Custom_Widgets/AnalogGaugeWidget.py.

[12] Documentation for fastapi introduction and use. https://fastapi.tiangolo.com/.

[13] Documentation for openapi introduction. https://rapidapi.com/blog/api-glossary/openapi/.

[14] Documentation for pyepics introduction and use. https://pyepics.github.io/pyepics/overview.html.

[15] Documentation for pyvisa introduction and use. https://pyvisa.readthedocs.io/en/latest/.

[16] Maxwell corkscrew rule theory. https://electricalvoice.com/right-hand-grip-cork-screw-rule/.

[17] Openapi initiative official website. https://www.openapis.org/.

[18] Thd (total harmonic distortion) explanation theory. https://www.gamry.com/application-notes/EIS/total-harmonic-distortion/#:~:text=The%20largest%20peak%20can%20be,linear%20behavior%20of%20a%20signal.

# Appendix A

# EPICS Installation in Linux OS

In order to install EPICS you need to install make, c++ and libreadline on the machine. With sudo apt-get install you can install all these packages very quickly:

- sudo apt-get install make
- sudo apt-get install g++
- sudo apt-get install libreadline-dev

We create the EPICS folder, where our program will be installed, and execute the following git command to start the installation. This will download the necessary installation files from the EPICS GitHub account. Initially, the version 7.0 files are downloaded:

- mkdir $HOME/EPICS
- cd $HOME/EPICS
- git clone –recursive https://github.com/epics-base/epics-base.git

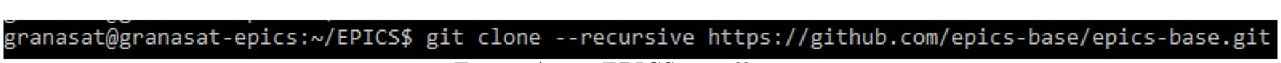


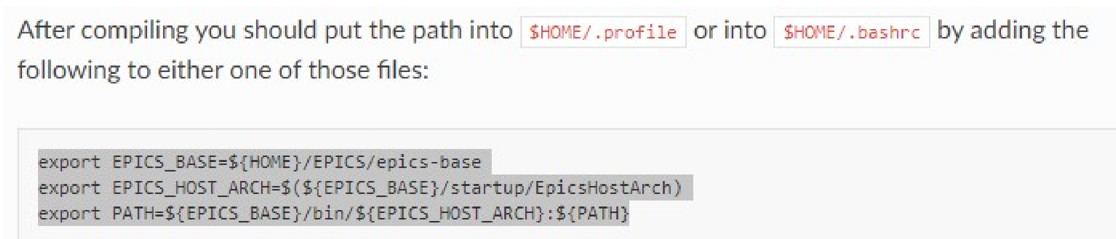
Figure A.1 – *EPICS installation*

The downloaded files are saved in a folder called epics_base. Inside it, you have to execute the make command, so that the installation begins using the Makefile file that is downloaded in the previous step:

- cd epics-base
- make

(Make is a Unix utility that is designed to start execution of a makefile. A makefile is a special file, containing shell commands, that you create and name makefile (or Makefile depending upon the system). While in the directory containing this makefile, you will type make and the commands in the makefile will be executed.)

Next, you have to enter the following text in one of the two indicated files (.profile and .bashrc). I have introduced it in both:


After compiling you should put the path into `$HOME/.profile` or into `$HOME/.bashrc` by adding the following to either one of those files:

```
export EPICS_BASE=${HOME}/EPICS/epics-base
export EPICS_HOST_ARCH=$(${EPICS_BASE}/startup/EpicsHostArch)
export PATH=${EPICS_BASE}/bin/${EPICS_HOST_ARCH}:${PATH}
```

Figure A.2 – *Environment Variables*

These environment variables are absolutely necessary for EPICS to work correctly, declaring where the previously downloaded program is located, the host and the path.

After this, we put in terminal one of the following commands, depending on the file that you have modified (those commands are to reload the changes in said files and that the environment variables defined in said files take effect):

- source $HOME/.profile
- source $HOME/.bashrc

From here EPICS is already installed. To verify that it actually works, we will put the following command:

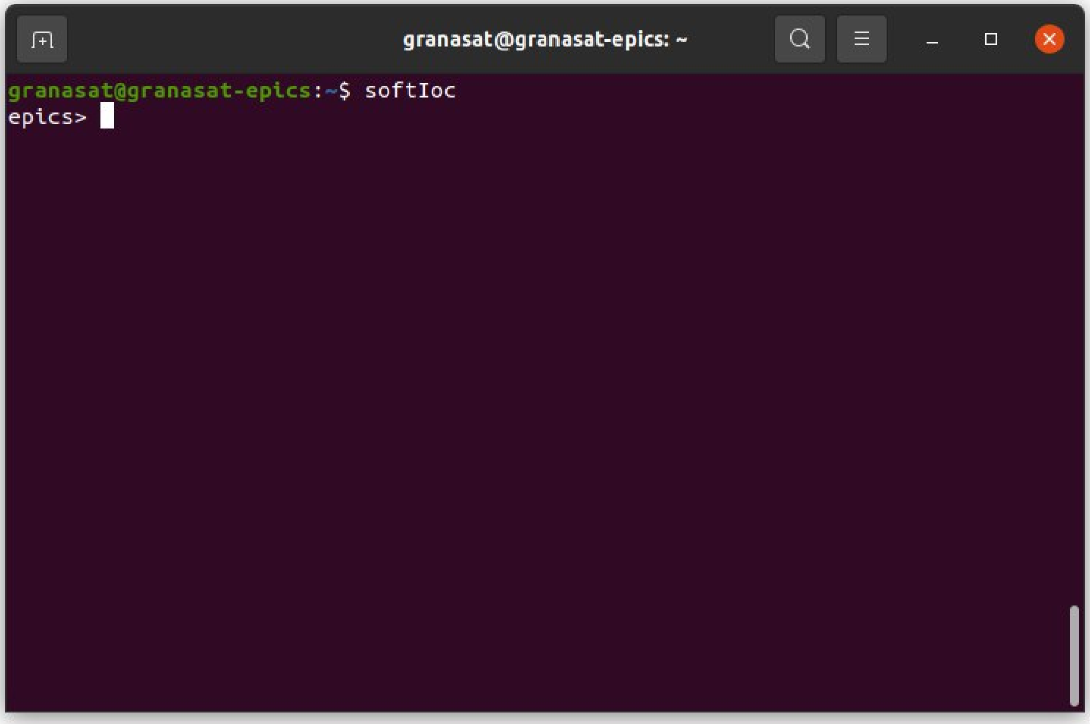



Figure A.3 – *EPICS Terminal*

If the EPICS terminal is displayed, it means that it is installed and working.

# Appendix B

# GUI's Installation and Deployment

Because Python is an interpreted language, the code written for this project will work on any machine that already has the Python interpreter installed.

We will use pip (which is Python's package management system) to install all the necessary software for the proper functioning of the project.

```

# Author: Salvador Jesus Megías Andreu

# Possible modes of use depending on the version of pip you have

# HOW TO USE: pip install -r requirements.txt
# HOW TO USE: pip3 install -r requirements.txt

# Versions used on my computer (to avoid version conflict issues)
# Python version used during development : 3.9.13
# Version of pip used during development: 22.1.2


PyVISA==1.11.3     # to communicate with hosts via ethernet
PyVISA-py==0.5.2     # to communicate with hosts over ethernet
matplotlib==3.5.0    # to generate plots with data
PyQt5==5.15.4      # to create the GUI
PyQt5-Qt5==5.15.2    # to create the GUI
pyqt5-tools==5.15.4.3.2 # to create the GUI
googletrans==3.1.0a0  # to translate the online GUI
argostranslate==1.6.1    # to translate the offline GUI
requests==2.23.0     # to make requests to the api
numpy==1.21.4      # to handle arrays more efficiently when searching for maxima
scipy==1.8.1     # to perform maxima calculations on dot plots
pyepics==3.5.1     # to communicate with epics
fastapi==0.78.0      # to create an openapi api
uvicorn==0.17.6      # to launch the created api openapi
```

Listado B.1 – *requirements.txt file*

For Project Deployment:

- API Deployment: python -m uvicorn api:api –reload
- EPICS IOC Deployment: softIoc -d TFG_EPICS_IOC.txt
- python TFG_SALVADOR.py

# Appendix C

# Library Code in Python to communicate with Agilent N9020A machine

```

# ##############################################################################
# ##############################################################################
# ##############################################################################
# ##############################################################################
######## ##############
######## AUTHOR: SALVADOR JESÚS MEGÍAS ANDREU ##############
######## EMAIL: salvadorjmegias@gmail.com ##############
######## UNIVERSITY EMAIL: salvadorjesus@correo.ugr.es ##############
######## ##############
# ##############################################################################
# ##############################################################################
# ##############################################################################
# ##############################################################################
```

```
# Librerías o Modulos necesarios a importar
import pyvisa as visa
import numpy as np
from struct import unpack
import pylab
import time
from matplotlib import pyplot as plot

#
 ##################################################################################
############## CLASS AGILENT_N9020A
 #####################################
############## SPECTRUM ANALYZER CONTROLLED
 #####################################
############## BY ETHERNET CONTROL
 #####################################
#
 ##################################################################################

 class Agilent_MXA_N9020A:

    # La medida con la que vamos a trabajar van a ser los MHz
    medida ='MHZ'

#
 ##################################################################################
############## CONSTRUCTOR
 #################################################################
#
 ##################################################################################

    # Cuando creemos un objeto de la clase, ejecutaremos el setup()
 conectándose automáticamente a la máquina
    # mediante conexión TCP

    def __init__(self):

        self.scope= self.setup()
        #sel.nPoints = int(self.scope.query('SWE:POIN?'))

#
 ##################################################################################
```

```
############## IDENTITY & SETUP ##################################################################
#
##############################################################################################

    # Muestra la información propia ofrecida por la máquina

    def identity(self):
        info= self.scope.query('*IDN?')
        info = info.split(",")
        print("Fabricante: ",info[0])
        print("Modelo: ",info[1])
        print("Número de serie: ",info[2])
        print("Firmware: ",info[3])

    # Establece una conexión TCP con la máquina mediante su IP devolviendo el objeto conectado para poder manejarlo

    def setup(self):
        # 192.168.1.200 IP AGILENT MACHINE
        rm = visa.ResourceManager('@py') # Calling PyVisaPy library
        scope = rm.open_resource('TCPIP::192.168.1.200::INSTR') # Connecting via LAN
        return scope


    # Finaliza la conexión con la máquina
    def disconnect(self):
        self.scope.close()

#
##############################################################################################
# AL CONTRARIO QUE LA MÁQUINA ANRITSU, AGILENT SOLO TIENE MODO SPECTRUM ANALYZER, POR LO QUE NO ES NECESARIO ACTIVAR NINGÚN MODO
# EL MODO SPECTRUM ANALYZER VIENE POR DEFECTO
#
##############################################################################################

# Activa el modo Spectrum en la máquina (por si acaso)

    def setSpectrum(self):
        self.scope.write('INST SA')
        self.instAgilent = 'SA'
        #print("Ha seleccionado el Spectrum Analyzer")


    # Función con la que recogemos todos los datos de la máquina una vez nos conectamos a esta (para tener los datos y no tener que reescribirlos si no es necesario)
```

```
    # Dejamos la máquina finalmente en el modo en el que estaba
 cuando nos conectamos, no modificando así nada
    def getInitialParamsAgilent(self):

        self.instAgilent= str(self.scope.query('INST?'))

        # Seleccionamos el spectrum para poder recoger los datos del
 spectrum
        self.setSpectrum()
        self.inicialFreq= float(self.scope.query('FREQ:START?'))/1e6
        self.finalFreq= float(self.scope.query('FREQ:STOP?'))/1e6
        self.centralFreq= float(self.scope.query('FREQ:CENT?'))/1e6
        self.referenceLevel = float(self.scope.query('DISP:WIND:TRAC
:Y:RLEV?'))
        self.nPoints = int(self.scope.query('SWE:POIN?'))
        self.span = float(self.scope.query('FREQ:SPAN?'))/1e6

        self.scope.write('INST '+ self.instAgilent)

#
 ##########################################################################################
######### FUNCTIONS FOR SPECTRUM ANALYZER
 ######################################################
#
 ##########################################################################################


    # Muestra todos los parámetros del Espectro en ese momento

    def getParamsSpectrum(self):
        print("Frecuencia central: ",self.centralFreq ,"MHz") #
 Muestra la frecuencia central
        print("Frecuencia inicial: ",self.inicialFreq,"MHz") #
 Muestra la frecuencia inicial
        print("Frecuencia final: ",self.finalFreq,"MHz") # Muestra
 la frecuencia final
        print("Nivel de referencia: ",self.referenceLevel,"dBm") #
 Muestra el nivel de referencia

    # Modifica todos los parámetros del Spectrum Analyzer (la medida
 de las frecuencias está en MHz)

    def setParamsSpectrum(self,inicialFreq , finalFreq ,
 referenceLevel):
        # Guarda todos los datos en variables del objeto de la clase
        self.inicialFreq = float(inicialFreq)
        self.finalFreq = float(finalFreq)
        self.referenceLevel = float(referenceLevel)
        self.centralFreq = (self.finalFreq + self.inicialFreq)/2.0

        self.scope.write('FREQ:START '+ str(inicialFreq) + self.
 medida) # Modifica la frecuencia inicial
```

```
        self.scope.write('FREQ:STOP '+ str(finalFreq) + self.medida) # Modifica la frecuencia final
        self.scope.write('FREQ:CENT '+ str(self.centralFreq)+ self.medida) # Modifica la frecuencia central
        self.scope.write('DISP:WIND:TRAC:Y:RLEV '+ str(referenceLevel)) # Modifica el nivel de referencia
        # Se define el número de puntos observables y medibles al valor que tenga la máquina en ese momento (por defecto son 10001)
        self.nPoints = int(self.scope.query('SWE:POIN?'))



    # Modifica todos los parámetros del Spectrum Analyzer mediante el uso de span (la medida de las frecuencias está en MHz)

    def setParamsSpectrumSpan(self, centralFreq , span , referenceLevel):
            self.setCentralFreqMHz(centralFreq) # Modifica la frecuencia central y su atributo de la clase
            self.setSpanMHz(span) # llama a la función modificando el valor de span inicialFreq y finalFreq y los atributos de la clase
            self.setReferenceLevelDBM(referenceLevel) # llama a la función modificando el valor de referencelevel y el atributo de la clase
            # Se define el número de puntos observables y medibles al valor que tenga la máquina en ese momento (por defecto son 10001)
            self.nPoints = int(self.scope.query('SWE:POIN?'))



    # Modifica el valor del span, inicialFreq y finalFreq y aparte los atributos de la clase correspondientes a la frecuencia inicial y final y el span
    # (Hace falta haber definido la frecuencia central)

    def setSpanMHz(self, span):
        self.span = float(span)
        mitad = self.span / 2.0
        self.scope.write('FREQ:SPAN '+ str(span)+ self.medida)

        self.inicialFreq = self.centralFreq - mitad
        self.scope.write('FREQ:START '+ str(self.inicialFreq) + self.medida)
        self.finalFreq = self.centralFreq + mitad
        self.scope.write('FREQ:STOP '+ str(self.finalFreq) + self.medida)

    # Muestra el Span en ese momento

    def getSpanMHz(self):
        print("Span: ", self.span , " MHz")
```

```
    # Modifica el valor de la frecuencia central y el atributo del objeto de la clase correspondiente

    def setCentralFreqMHz(self,centralFreq):
        self.centralFreq = float(centralFreq)
        self.scope.write('FREQ:CENT '+ str(centralFreq)+ self.medida)

    # Muestra la frecuencia central en ese momento

    def getCentralFreqMHz(self):
        print("Frecuencia central: ",self.centralFreq," MHz")

    # Modifica la frecuencia incial y el atributo del objeto de la clase correspondiente

    def setInicialFreqMHz(self,inicialFreq):
        self.inicialFreq = float(inicialFreq)
        self.scope.write('FREQ:START '+ str(inicialFreq) + self.medida)
        self.centralFreq = (self.inicialFreq + self.finalFreq)/2.0

    # Muestra la frecuencia inicial en ese momento

    def getInicialFreqMHz(self):
        print("Frecuencia inicial: ",self.inicialFreq," MHz")

    # Modifica la frecuencia final y el atributo del objeto de la clase correspondiente

    def setFinalFreqMHz(self,finalFreq):
        self.finalFreq = float(finalFreq)
        self.scope.write('FREQ:STOP '+ str(finalFreq) + self.medida)
        self.centralFreq = (self.inicialFreq + self.finalFreq)/2.0

    # Muestra la frecuencia final en ese momento

    def getFinalFreqMHz(self):
        print("Frecuencia final: ",self.finalFreq," MHz")

    # Modifica el nivel de referencia y el atributo del objeto de la clase correspondiente

    def setReferenceLevelDBM(self,referenceLevel):
        self.referenceLevel = float(referenceLevel)
        self.scope.write('DISP:WIND:TRAC:Y:RLEV '+ str(referenceLevel))

    # Muestra el nivel de referencia en ese momento

    def getReferenceLevelDBM(self):
        print("Nivel de referencia: ",self.referenceLevel," dBm")
```

```

    # Muestra y devuelve el número de puntos observables y medibles
 en ese momento

    def getNumPoints(self):
        puntos = int(self.scope.query('SWE:POIN?'))
        #print("Número de puntos: ",puntos)
        return puntos

    # Modifica el número de puntos observables y medibles y el
 atributo del objeto de la clase correspondiente

    def setNumPoints(self,npoints):
        self.nPoints = int(npoints)
        self.scope.write('SWE:POIN '+ str(npoints))

    # Muestra y devuelve la frecuencia donde se encuentra la
 potencia máquina y la potencia máxima

    def getMaxFreqPower(self):
        self.scope.write('CALC:MARK:MAX')
        #print("Frecuencia donde se encuentra la potencia máxima: ",
 self.scope.query('CALC:MARK:X?')," MHz")
        #print("Potencia máxima: ",self.scope.query('CALC:MARK:Y?')
 ," dBm")
        self.maxfreq = float(self.scope.query('CALC:MARK:X?'))/1e6
        power = float(self.scope.query('CALC:MARK:Y?'))
        return power

    # Función que devuelve la potencia referente a una frecuencia
 dada

    def getPowerDBM(self, freq):
        self.scope.write('CALC:MARK:X '+ str(freq)+ self.medida)
        print("Potencia asociada a la frecuencia dada: ",self.scope.
 query('CALC:MARK:Y?'), " dBm")

#
 ##########################################################################################
############## PLOT INFORMATION
 ############################################################
#
 ##########################################################################################

    # Función que guarda una imagen png y la muestra en pantalla de
 la señal del Spectrum completa en ese momento

    def plotInfoAgilent(self):

        puntos = self.getNumPoints() # guardamos el número de puntos
  a representar con plot
```

```
        self.scope.write('FORM ASC') # Pide a la máquina que lo que se le pide lo devuelva en formato ASCII
        datos = self.scope.query('TRAC? TRACE1') # Pide a la máquina los datos del Spectrum (los datos son las potencias de los 10001 puntos observables y medibles)
        datosManipulables = datos.split(",") # separa todos los datos separados por comas y los guarda en una lista
        datosManipulables = [float(i) for i in datosManipulables] # transformo los datos de string a float para poder trabajar con ellos


        self.datosCapturados = datosManipulables.copy() # Para poder hacer uso de ellos en caso de querer buscar máximos, mínimos...

        freq = self.finalFreq - self.inicialFreq # definimos la amplitud del intervalo de frecuencias a representar
        pointWidth = freq / float(puntos) # defino la anchura que debe ocupar cada punto en la imagen (para saber donde colocar cada frecuencia en la imagen)


        # Creo una lista de todas las frecuencias a representar
        frequencies = []
        count = 0
        while len(frequencies) != puntos:
            frequencies.append(self.inicialFreq+(pointWidth*count))
            count+=1

        plot.clf()

        # Label for x-axis
        plot.xlabel("Frequency (MHz)")

        # Label for y-axis
        plot.ylabel("Power (dBm)")

        # title of the plot
        plot.title("Output of Spectrum Analyzer")

        # Add grid lines
        plot.grid()

        #Genero la imagen con las frecuencias calculadas y las potencias ofrecidas por la máquina

        plot.plot(frequencies,datosManipulables)
        plot.savefig('./images/graphAgilent.png') # Dirección relativa donde se quiere que se guarde la imagen creada
        plot.show()
```

Listado C.1 – *Agilent N9020A Library in Python*

# Appendix D

# Library Code in Python to communicate with Anritsu MS2830A machine

```
# ###############################################################################
# ###############################################################################
# ###############################################################################
# ###############################################################################
######## ##############
######## AUTHOR: SALVADOR JESÚS MEGÍAS ANDREU ##############
######## EMAIL: salvadorjmegias@gmail.com ##############
######## UNIVERSITY EMAIL: salvadorjesus@correo.ugr.es ##############
######## ##############
# ###############################################################################
# ###############################################################################
# ###############################################################################
# ###############################################################################
```

```
# Librerías o Modulos necesarios a importar
import pyvisa as visa
import numpy as np
from struct import unpack
import pylab
import time
from matplotlib import pyplot as plot


#
 ##########################################################################################
############### CLASS ANRITSUMS2830A
 ########################################
############### SPECTRUM ANALYZER CONTROLLED
 ########################################
############### AND SIGNAL GENERATOR CONTROLLED
 ########################################
############### BY ETHERNET CONTROL
 ########################################
#
 ##########################################################################################


class AnritsuMS2830A:

    # La medida con la que vamos a trabajar van a ser los MHz
    medida ='MHZ'

#
 ##########################################################################################
############### CONSTRUCTOR
 ######################################################################
#
 ##########################################################################################


    # Cuando creemos un objeto de la clase, ejecutaremos el setup()
 conectándose automáticamente a la máquina
    # mediante conexión TCP

    def __init__(self):

        self.scope= self.setup()

        #self.nPoints = int(self.scope.query('SWE:POIN?'))# reogemos
  el número de puntos de la máquina
```

```
        #if int(mode)==0:
            #self.setSignalGen()
        #else:
            #self.setSpectrum()

#
 ###############################################################################################################
############## IDENTITY & SETUP
 ##############################################################################
#
 ###############################################################################################################


    # Muestra la información propia ofrecida por la máquina

    def identity(self):
        info= self.scope.query('*IDN?')
        info = info.split(",")
        print("Fabricante: ",info[0])
        print("Modelo: ",info[1])
        print("Número de serie: ",info[2])
        print("Firmware: ",info[3])

    # Establece una conexión TCP con la máquina mediante su IP
 devolviendo el objeto conectado para poder manejarlo

    def setup(self):
        # 192.168.1.139 IP MÁQUINA ANRITSU
        rm = visa.ResourceManager('@py') # Calling PyVisaPy library
        scope = rm.open_resource('TCPIP::192.168.1.139::INSTR') #
 Connecting via LAN
        #scope.write('*RST;*CLS') # resetea la máquina
        return scope

    # Finaliza la conexión con la máquina
    def disconnect(self):
        self.scope.close()
#
 ###############################################################################################################
############## GETTERS & SETTERS
 #############################################################################
#
 ###############################################################################################################


    # Activa el modo Spectrum en la máquina

    def setSpectrum(self):
        self.scope.write('INST SPECT')
        self.instAnritsu = 'SPECT'
        #print("Ha seleccionado el Spectrum Analyzer")
```

```python
    # Activa el modo Generador en la máquina

    def setSignalGen(self):
        self.scope.write('INST SG')
        self.instAnritsu = 'SG'
        #print("Ha seleccionado el Signal Generator")


    # Función con la que recogemos todos los datos de la máquina una vez nos conectamos a esta (para tener los datos y no tener que reescribirlos si no es necesario)
    # Dejamos la máquina finalmente en el modo en el que estaba cuando nos conectamos, no modificando así nada
    def getInitialParamsAnritsu(self):

        self.instAnritsu= str(self.scope.query('INST?'))

        # Seleccionamos el generador para poder obtener los datos del generador
        self.setSignalGen()
        self.frequency = float(self.scope.query('FREQ?'))
        self.state = int(self.scope.query('OUTP?'))
        self.power = float(self.scope.query('POW?'))

        # Seleccionamos el spectrum para poder recoger los datos del spectrum
        self.setSpectrum()
        self.inicialFreq= float(self.scope.query('FREQ:START?'))/1e6
        self.finalFreq= float(self.scope.query('FREQ:STOP?'))/1e6
        self.centralFreq= float(self.scope.query('FREQ:CENT?'))/1e6
        self.referenceLevel = float(self.scope.query('DISP:WIND:TRAC:Y:RLEV?'))
        self.nPoints = int(self.scope.query('SWE:POIN?'))
        self.span = float(self.scope.query('FREQ:SPAN?'))/1e6

        self.scope.write('INST '+ self.instAnritsu)
# ##########################################################################################
######### FUNCTIONS FOR SIGNAL GENERATOR ##################################################
# ##########################################################################################


    # Muestra en terminal la información relativa al generador: la frecuencia, la potencia y si está o no encendido

    def getParamsGenerator(self):
        print("Frecuencia del generador: ",self.frequency,"MHz")
        print("Potencia del generador: ",self.power,"dBm")
        if self.state == 1:
```

```python
            print("Estado del generador: Activado")
        else:
            print("Estado del generador: Desactivado")

    # Modifica todos los parámetros del generador de una sola vez

    def setParamsGenerator(self, frequency , power):
        self.scope.write('FREQ '+ str(frequency) + self.medida) # Modifica la frecuencia
        self.scope.write('UNIT.POW DBM') # Modifica las unidades de la potencia DBM
        self.scope.write('POW ' + str(power)) # Modifica la potencia
        self.scope.write('OUTP 1') # Enciende el Generador
        # Guarda todos los datos en variables del objeto de la clase
        self.frequency = float(frequency)
        self.power = float(power)
        self.state = 1

    # Modifica la frecuencia y el atributo del objeto de la clase correspondiente

    def setFrequencyMHz(self, frequency):
        self.scope.write('FREQ '+ str(frequency) + self.medida)
        self.frequency = float(frequency)

    # Muestra la frecuencia en ese instante

    def getFrequencyMHz(self):
        print("Frecuencia del generador: ",self.frequency," MHz")

    # Modifica la potencia y el atributo del objeto de la clase correspondiente

    def setPowerGeneratordBm(self,power):
        self.scope.write('POW ' + str(power))
        self.power = float(power)

    # Muestra la potencia en ese instante
    def getPowerGeneratordBm(self):
        print("Potencia del generador: ",self.power," dBm")

    # Función capaz de encender o apagar el generador (1 enciende, 0 apaga) y guarda el estado en un atributo del objeto de la clase correspondiente

    def setStateGenerator(self,state):
        self.scope.write('OUTP '+str(state))
        self.state = int(state)

    # Muestra el estado del generador (1 => encendido , cualquier otro => apagado (0))

    def getStateGenerator(self):
```

```
        if self.state == 1:
            print("El Generador está activado | estado: ", self.state)
        else:
            print("El Generador está desactivado | estado: ", self.state)



# ####################################################################################
######### FUNCTIONS FOR SPECTRUM ANALYZER ####################################################
# ####################################################################################

    # Muestra todos los parámetros del Espectro en ese momento

    def getParamsSpectrum(self):
        print("Frecuencia central: ",self.centralFreq ,"MHz") # Muestra la frecuencia central
        print("Frecuencia inicial: ",self.inicialFreq,"MHz") # Muestra la frecuencia inicial
        print("Frecuencia final: ",self.finalFreq,"MHz") # Muestra la frecuencia final
        print("Nivel de referencia: ",self.referenceLevel,"dBm") # Muestra el nivel de referencia

    # Modifica todos los parámetros del Spectrum Analyzer (la medida de las frecuencias está en MHz)

    def setParamsSpectrum(self,inicialFreq , finalFreq , referenceLevel):
        # Guarda todos los datos en variables del objeto de la clase
        self.inicialFreq = float(inicialFreq)
        self.finalFreq = float(finalFreq)
        self.referenceLevel = float(referenceLevel)
        self.centralFreq = (self.finalFreq + self.inicialFreq)/2.0

        self.scope.write('FREQ:START '+ str(inicialFreq) + self.medida) # Modifica la frecuencia inicial
        self.scope.write('FREQ:STOP '+ str(finalFreq) + self.medida) # Modifica la frecuencia final
        self.scope.write('FREQ:CENT '+ str(self.centralFreq)+ self.medida) # Modifica la frecuencia central
        self.scope.write('DISP:WIND:TRAC:Y:RLEV '+ str(referenceLevel)) # Modifica el nivel de referencia
        # Se define el número de puntos observables y medibles al valor que tenga la máquina en ese momento (por defecto son 10001)
        self.nPoints = int(self.scope.query('SWE:POIN?'))
```

```python


    # Modifica todos los parámetros del Spectrum Analyzer mediante el uso de span (la medida de las frecuencias está en MHz)

    def setParamsSpectrumSpan(self, centralFreq , span , referenceLevel):
            self.setCentralFreqMHz(centralFreq) # Modifica la frecuencia central y su atributo de la clase
            self.setSpanMHz(span) # llama a la función modificando el valor de span inicialFreq y finalFreq y los atributos de la clase
            self.setReferenceLevelDBM(referenceLevel) # llama a la función modificando el valor de referencelevel y el atributo de la clase
            # Se define el número de puntos observables y medibles al valor que tenga la máquina en ese momento (por defecto son 10001)
            self.nPoints = int(self.scope.query('SWE:POIN?'))



    # Modifica el valor del span, inicialFreq y finalFreq y aparte los atributos de la clase correspondientes a la frecuencia inicial y final y el span
    # (Hace falta haber definido la frecuencia central)

    def setSpanMHz(self, span):
        self.span = float(span)
        mitad = self.span / 2.0
        self.scope.write('FREQ:SPAN '+ str(span)+ self.medida)

        self.inicialFreq = self.centralFreq - mitad
        self.scope.write('FREQ:START '+ str(self.inicialFreq) + self.medida)
        self.finalFreq = self.centralFreq + mitad
        self.scope.write('FREQ:STOP '+ str(self.finalFreq) + self.medida)

    # Muestra el Span en ese momento

    def getSpanMHz(self):
        print("Span: ", self.span , " MHz")



    # Modifica el valor de la frecuencia central y el atributo del objeto de la clase correspondiente

    def setCentralFreqMHz(self,centralFreq):
        self.centralFreq = float(centralFreq)
```

```
        self.scope.write('FREQ:CENT '+ str(centralFreq)+ self.medida)

    # Muestra la frecuencia central en ese momento

    def getCentralFreqMHz(self):
        print("Frecuencia central: ",self.centralFreq," MHz")

    # Modifica la frecuencia incial y el atributo del objeto de la clase correspondiente

    def setInicialFreqMHz(self,inicialFreq):
        self.inicialFreq = float(inicialFreq)
        self.scope.write('FREQ:START '+ str(inicialFreq) + self.medida)

    # Muestra la frecuencia inicial en ese momento

    def getInicialFreqMHz(self):
        print("Frecuencia inicial: ",self.inicialFreq," MHz")

    # Modifica la frecuencia final y el atributo del objeto de la clase correspondiente

    def setFinalFreqMHz(self,finalFreq):
        self.finalFreq = float(finalFreq)
        self.scope.write('FREQ:STOP '+ str(finalFreq) + self.medida)

    # Muestra la frecuencia final en ese momento

    def getFinalFreqMHz(self):
        print("Frecuencia final: ",self.finalFreq," MHz")

    # Modifica el nivel de referencia y el atributo del objeto de la clase correspondiente

    def setReferenceLeveldBm(self,referenceLevel):
        self.referenceLevel = float(referenceLevel)
        self.scope.write('DISP:WIND:TRAC:Y:RLEV '+ str(referenceLevel))

    # Muestra el nivel de referencia en ese momento

    def getReferenceLeveldBm(self):
        print("Nivel de referencia: ",self.referenceLevel," dBm")

    # Muestra y devuelve el número de puntos observables y medibles en ese momento

    def getNumPoints(self):
        puntos = int(self.scope.query('SWE:POIN?'))
        #print("Número de puntos: ",puntos)
        return puntos
```

```python

    # Modifica el número de puntos observables y medibles y el atributo del objeto de la clase correspondiente

    def setNumPoints(self,npoints):
        self.nPoints = int(npoints)
        self.scope.write('SWE:POIN '+ str(npoints))

    # Muestra y devuelve la frecuencia donde se encuentra la potencia máquina y la potencia máxima

    def getMaxFreqPower(self):
        self.scope.write('CALC:MARK:ACT ON; CALC:MARK:RES PEAK; CALC:MARK:MAX')
        #print("Frecuencia donde se encuentra la potencia máxima: ", self.scope.query('CALC:MARK:X?'), " MHz")
        #print("Potencia máxima: ",self.scope.query('CALC:MARK:Y?'), " dBm")
        self.maxfreq = float(self.scope.query('CALC:MARK:X?'))/1e6
        power = float(self.scope.query('CALC:MARK:Y?'))
        return power

    # Función que devuelve la potencia referente a una frecuencia dada

    def getPowerdBm(self, freq):
        self.scope.write('CALC:MARK:ACT OFF; CALC:MARK:ACT ON; CALC:MARK:X '+ str(freq)+ self.medida)

        power = self.scope.query('CALC:MARK:Y?')
        print("Potencia asociada a la frecuencia dada: ",power, " dBm")
        return power

# ##########################################################################################
############### PLOT INFORMATION ###############################################################
# ##########################################################################################


    # Función que guarda una imagen png y la muestra en pantalla de la señal del Spectrum completa en ese momento

    def plotInfoAnritsu(self):

        puntos = self.getNumPoints() # guardamos el número de puntos a representar con plot

        self.scope.write('FORM ASC') # Pide a la máquina que lo que se le pide lo devuelva en formato ASCII
```

```
        datos = self.scope.query('TRAC? TRAC1') # Pide a la máquina los datos del Spectrum (los datos son las potencias de los 10001 puntos observables y medibles)
        datosManipulables = datos.split(",") # separa todos los datos separados por comas y los guarda en una lista
        datosManipulables = [float(i) for i in datosManipulables] # transformo los datos de string a float para poder trabajar con ellos

        self.datosCapturados = datosManipulables.copy() # Para poder hacer uso de ellos en caso de querer buscar máximos, mínimos...

        freq = self.finalFreq - self.inicialFreq # definimos la amplitud del intervalo de frecuencias a representar
        pointWidth = freq / float(puntos) # defino la anchura que debe ocupar cada punto en la imagen (para saber donde colocar cada frecuencia en la imagen)


        # Creo una lista de todas las frecuencias a representar
        frequencies = []
        count = 0
        while len(frequencies) != puntos:
            frequencies.append(self.inicialFreq+(pointWidth*count))
            count+=1

        plot.clf()
        # Label for x-axis
        plot.xlabel("Frequency (MHz)")

        # Label for y-axis
        plot.ylabel("Power (dBm)")

        # title of the plot
        plot.title("Output of Spectrum Analyzer")

        # Add grid lines
        plot.grid()

        #Genero la imagen con las frecuencias calculadas y las potencias ofrecidas por la máquina

        plot.plot(frequencies,datosManipulables)
        plot.savefig('./images/graphAnritsu.png') # Dirección relativa donde se quiere que se guarde la imagen creada
        plot.show()
```

Listado D.1 – *Anritsu MS2830A Library in Python*

# Appendix E

# Example to show Multithreading in PyQt

This example was taken and modified from the following website: Multithreading example base

```
from PyQt5 import QtCore, QtWidgets,QtGui
from PyQt5 import uic
import sys, time

# Clase de la GUI

class PyShine_THREADS_APP(QtWidgets.QMainWindow):
    def __init__(self):
        QtWidgets.QMainWindow.__init__(self)
        self.ui = uic.loadUi('threads.ui',self) # Cargamos la GUI directamente del archivo .ui
        self.resize(888, 200)
        # Esto de abajo no hará nada puesto que no tenemos la imagen del logo dle ejemplo
        icon = QtGui.QIcon()
        icon.addPixmap(QtGui.QPixmap("PyShine.png"), QtGui.QIcon.Normal, QtGui.QIcon.Off)
        self.setWindowIcon(icon)

        # Deshabilito los botones de Stop inicialmente para evitar fallas en el programa
        self.pushButton_4.setEnabled(False)
        self.pushButton_5.setEnabled(False)
        self.pushButton_6.setEnabled(False)

        # Se define un conjunto vacío donde guardaremos los threads que se vayan a crear a lo largo del programa para manejarlos facilmente
        self.thread={}

        # Se conectan todos los botones a las funciones correspondientes para que estas se ejecuten al pulsar el botón en la GUI (programación de eventos)
```

```
        self.pushButton.clicked.connect(self.start_worker_1)
        self.pushButton_2.clicked.connect(self.start_worker_2)
        self.pushButton_3.clicked.connect(self.start_worker_3)
        self.pushButton_4.clicked.connect(self.stop_worker_1)
        self.pushButton_5.clicked.connect(self.stop_worker_2)
        self.pushButton_6.clicked.connect(self.stop_worker_3)


    # Funciones encargadas de iniciar los threads

    def start_worker_1(self):
        self.thread[1] = ThreadClass(parent=None,index=1) # Creamos en la posición 1 del conjunto thread un objeto de la clase ThreadClass para poder manejar dicho objeto facilmente (otorgándole al thread creado el id=1)
        self.thread[1].start() # Llamamos a la función run() de la clase ThreadClass para iniciar el Thread 1 (por defecto start() llama a la función run()) VER CAP DESPUÉS DEL CÓDIGO
        self.thread[1].any_signal.connect(self.my_function) # conforme van llegando las señales (datos del 0 al 99) se conecta con la función my_function, la cual irá actualizando el progressBar
        self.pushButton.setEnabled(False) # Cuando le doy al botón de start del Thread 1, deshabilito el botón de start del Thread 1
        self.pushButton_4.setEnabled(True) # Cuando le doy al botón de start del Thread 1, habilito el botón de stop del Thread 1

    def start_worker_2(self):
        self.thread[2] = ThreadClass(parent=None,index=2) # Creamos en la posición 2 del conjunto thread un objeto de la clase ThreadClass para poder manejar dicho objeto facilmente (otorgándole al thread creado el id=2)
        self.thread[2].start() # Llamamos a la función run() de la clase ThreadClass para iniciar el Thread 2 (por defecto start() llama a la función run()) VER CAP DESPUÉS DEL CÓDIGO
        self.thread[2].any_signal.connect(self.my_function) # conforme van llegando las señales (datos del 0 al 99) se conecta con la función my_function, la cual irá actualizando el progressBar
        self.pushButton_2.setEnabled(False) # Cuando le doy al botón de start del Thread 2, deshabilito el botón de start del Thread 2
        self.pushButton_5.setEnabled(True) # Cuando le doy al botón de start del Thread 2, habilito el botón de stop del Thread 2

    def start_worker_3(self):
        self.thread[3] = ThreadClass(parent=None,index=3) # Creamos en la posición 3 del conjunto thread un objeto de la clase ThreadClass para poder manejar dicho objeto facilmente (otorgándole al thread creado el id=3)
        self.thread[3].start() # Llamamos a la función run() de la clase ThreadClass para iniciar el Thread 3 (por defecto start() llama a la función run()) VER CAP DESPUÉS DEL CÓDIGO
        self.thread[3].any_signal.connect(self.my_function) #
```

```
conforme van llegando las señales (datos del 0 al 99) se conecta
con la función my_function, la cual irá actualizando el progressBar
        self.pushButton_3.setEnabled(False) # Cuando le doy al botón de start del Thread 3, deshabilito el botón de start del Thread 3
        self.pushButton_6.setEnabled(True) # Cuando le doy al botón de start del Thread 3, habilito el botón de stop del Thread 3

    # Funciones encargadas de para los threads

    def stop_worker_1(self):
        self.thread[1].stop() # Llamamos a la función stop() de la clase ThreadClass para parar el Thread 1
        self.pushButton.setEnabled(True) # Cuando le doy al botón de stop del Thread 1, habilito el botón de start del Thread 1
        self.pushButton_4.setEnabled(False) # Cuando le doy al botón de stop del Thread 1, deshabilito el botón de stop del Thread 1

    def stop_worker_2(self):
        self.thread[2].stop() # Llamamos a la función stop() de la clase ThreadClass para parar el Thread 2
        self.pushButton_2.setEnabled(True) # Cuando le doy al botón de stop del Thread 2, habilito el botón de start del Thread 2
        self.pushButton_5.setEnabled(False) # Cuando le doy al botón de stop del Thread 2, deshabilito el botón de stop del Thread 2

    def stop_worker_3(self):
        self.thread[3].stop() # Llamamos a la función stop() de la clase ThreadClass para parar el Thread 3
        self.pushButton_3.setEnabled(True) # Cuando le doy al botón de stop del Thread 3, habilito el botón de start del Thread 3
        self.pushButton_6.setEnabled(False) # Cuando le doy al botón de stop del Thread 3, deshabilito el botón de stop del Thread 3


    # Función que va modificando el valor del progressBar correspondiente al id del thread asociado a dicho progressBar

    def my_function(self,counter):

        #cnt=counter
        index = self.sender().index
        if index==1:
            self.progressBar.setValue(counter)
        if index==2:
            self.progressBar_2.setValue(counter)
        if index==3:
            self.progressBar_3.setValue(counter)



# Clase para el control del Multithreading en la GUI

class ThreadClass(QtCore.QThread):
```

```
    any_signal = QtCore.pyqtSignal(int) # Creamos un Objeto de
 pyqtSignal el cual va a manejar enteros

    # Constructor para definir la variable index e is_running
    def __init__(self,parent=None,index=0):
        #super(ThreadClass, self).__init__(parent) # Carga el
 constructor del padre que en nuestro caso es QThread
        super(QtCore.QThread,self).__init__(parent) # la declaración
  de arriba y esta hacen exactamente lo mismo
        self.index=index
        self.is_running = True

    # Función (a la que recurre start()) con bucle infinito que va
 emitiendo los datos (del 0 al 99) una y otra vez mediante el objeto
  creado any_signal de pyqtSignal

    def run(self):
        print('Starting thread...',self.index) # Se muestra el index
  del Thread que llama a esta función
        cnt=0
        while (True):
            cnt+=1
            if cnt==99: cnt=0 # Si el valor de cnt a llegado a 99,
 se inicializa a 0 de nuevo
            time.sleep(0.01)
            self.any_signal.emit(cnt) # emit() ==> función que emite
  los datos

    # Función que se encarga de parar o finalizar el proceso del
 thread que llama esta función

    def stop(self):
        self.is_running = False
        print('Stopping thread...',self.index) # Se muestra el index
  del Thread que llama a esta función
        self.terminate() # terminate() ==> función que finaliza el
 proceso



app = QtWidgets.QApplication(sys.argv)
mainWindow = PyShine_THREADS_APP()
mainWindow.show()
sys.exit(app.exec_())
```
Listado E.1 – *Multithreading example in PyQt*

# Appendix F

# Final EPICS IOC Records

Below we can see all the records of our EPICS IOC used in the project or that could be useful and interesting for future developments and evolutions of the project:

The field called "NELM" (Number of Elements) in the records is to declare, when displaying the IOC, the number of elements that these records can contain, that is, we declare that they are arrays or lists and the number of elements that they can have.
Otherwise, the Python EPICS Client will only fetch an element from the list or array on request for the value of the list or array.

```
record(ai, "Anritsu:SPECT_InitialFrequency")
{
    field(DESC, "Initial Frequency in Anritsu (MHz)")
}

record(ai, "Anritsu:SPECT_FinalFrequency")
{
    field(DESC, "Final Frequency in Anritsu (MHz)")
}

record(ai, "Anritsu:SPECT_ReferenceLevel")
{
    field(DESC, "Reference Level in Anritsu (dBm)")
}

record(ai, "Anritsu:SPECT_MaximumFrequency")
{
    field(DESC, "Maximum Frequency in Anritsu (MHz)")
}

record(ai, "Anritsu:SPECT_MaximumPower")
{
    field(DESC, "Maximum Power in Anritsu (dBm)")
}
```

```
record(ai, "Anritsu:SPECT_THD")
{
    field(DESC, "THD in Anritsu  (100 MHz - 1.5 GHz)")
}


record(aai,"Anritsu:SPECT_Frequencies")  {
    field(DESC,"Frequencies in Anritsu  (MHz)")
    field(NELM,"30000")

}

record(aai,"Anritsu:SPECT_Powers")  {
    field(DESC,"Powers in Anritsu  (dBm)")
    field(NELM,"30000")

}

record(aai,"Anritsu:SPECT_unitsTime")  {
    field(DESC,"units time in Anritsu  (30s)")
    field(NELM,"30000")

}

record(aai,"Anritsu:SPECT_MaximumPowers")  {
    field(DESC,"Maximum Powers in Anritsu  (dBm)")
    field(NELM,"30000")

}


record(ai, "Anritsu:SG_Power")
{
    field(DESC, "Power in Anritsu SG (dBm)")
}

record(ai, "Anritsu:SG_Frequency")
{
    field(DESC, "Frequency in Anritsu SG (MHz)")
}

record(ai, "Anritsu:SG_State")
{
    field(DESC, "State of Generator in Anritsu (1 or 0)")
}

record(ai, "Anritsu:Instrument_Choosed")
{
    field(DESC, "Instrument choosed")
}

```

```
record(ai, "Anritsu:SomeValueChanged")
{
    field(DESC, "Value changed (1 or 0)")
}












record(ai, "Agilent:InitialFrequency")
{
    field(DESC, "Initial Frequency in Agilent SA (MHz)")
}

record(ai, "Agilent:FinalFrequency")
{
    field(DESC, "Final Frequency in Agilent SA (MHz)")
}

record(ai, "Agilent:ReferenceLevel")
{
    field(DESC, "Reference Level in Agilent SA (dBm)")
}

record(ai, "Agilent:MaximumFrequency")
{
    field(DESC, "Maximum Frequency in Agilent SA (MHz)")
}

record(ai, "Agilent:MaximumPower")
{
    field(DESC, "Maximum Power in Agilent SA (dBm)")
}

record(ai, "Agilent:THD")
{
    field(DESC, "THD in Agilent SA (100 MHz - 1.5 GHz)")
}


record(aai,"Agilent:SA_Frequencies")  {
    field(DESC,"Frequencies in Agilent SA (MHz)")
    field(NELM,"30000")

}

```

```
record(aai,"Agilent:SA_Powers")   {
    field(DESC,"Powers in Agilent SA (dBm)")
    field(NELM,"30000")

}

record(aai,"Agilent:SA_unitsTime")   {
    field(DESC,"units time in Agilent SA (30s)")
    field(NELM,"30000")

}

record(aai,"Agilent:SA_MaximumPowers")   {
    field(DESC,"Maximum Powers in Agilent SA (dBm)")
    field(NELM,"30000")

}

record(ai, "Agilent:Instrument_Choosed")
{
    field(DESC, "Instrument choosed")
}

record(ai, "Agilent:SomeValueChanged")
{
    field(DESC, "Value changed (1 or 0)")
}








record(ai, "BLAS:waterTemperature")
{
    field(DESC, "Water temperature in the circuit (Celsius degrees)")
}

record(ai, "BLAS:waterFlow")
{
    field(DESC, "Water Flow in the circuit (l/s)")
}

record(ai, "BLAS:IlockFlow")
{
    field(DESC, "IlockFlow output signal (V)")
}

record(ai, "BLAS:Fail0")
{
```

```
    field(DESC, "Fail0 output signal (V)")
}

record(ai, "BLAS:Fail1")
{
    field(DESC, "Fail1 output signal  (V)")
}

record(ai, "BLAS:PD_MI")
{
    field(DESC, "PD_MI output signal (V)")
}

record(ai, "BLAS:IlockPatchPanel")
{
    field(DESC, "IlockPatchPanel output signal (V)")
}

record(ai, "BLAS:VSEL0")
{
    field(DESC, "VSEL0 output signal (V)")
}

record(ai, "BLAS:VSEL1")
{
    field(DESC, "VSEL1 output signal (V)")
}

record(ai, "BLAS:12V_DC")
{
    field(DESC, "12V_DC input signal (V)")
}

record(ai, "BLAS:DriverStart")
{
    field(DESC, "DriverStart input signal (V)")
}

record(ai, "BLAS:AmplifierStart")
{
    field(DESC, "AmplifierStart input signal (V)")
}

record(ai, "BLAS:SomeValueChanged")
{
    field(DESC, "Value changed (1 or 0)")
}
```

Listado F.1 – *Records that make up our EPICS IOC "TFG_EPICS_IOC.txt File"*

# Appendix G

# API developed to simulate BLAS data under normal conditions

```

# ########################################################################
# ########################################################################
# ########################################################################
# ########################################################################
######## ##############
######## AUTHOR: SALVADOR JESÚS MEGÍAS ANDREU ##############
######## EMAIL: salvadorjmegias@gmail.com ##############
######## UNIVERSITY EMAIL: salvadorjesus@correo.ugr.es ##############
######## ##############
# ########################################################################
# ########################################################################
# ########################################################################
# ########################################################################
```

```
# HOW TO USE TO EXECUTE AND LAUNCH: uvicorn api:api --reload

# If you don't find the uvicorn module installed, use the following:
# HOW TO USE TO EXECUTE AND LAUNCH: python -m uvicorn api:api --reload

# The OPENAPI documentation generated by FastAPI can be accessed directly
# automatically at the following link (provided you have launched uvicorn with the command above): http://127.0.0.1:8000/docs


from fastapi import FastAPI
import random

api = FastAPI(title= "TFG Salvador Jesús Megías Andreu")

@api.get("/getTemperature")
async def getTemperature():
    return float(random.randint(22*10,50*10)/10)

@api.get("/getFlow")
async def getFlow():
    return float(random.randint(1*10,7*10)/10)


@api.get("/getIlockFlow")
async def getIlockFlow():
    return int(12)

@api.get("/getFail0")
async def getFail0():
    return int(0)

@api.get("/getFail1")
async def getFail1():
    return int(0)

@api.get("/getPdMi")
async def getPdMi():
    return float(7.6)

@api.get("/getIlockPatchPanel")
async def getIlockPatchPanel():
    return int(12)

@api.get("/getVsel0")
async def getVsel0():
    return float(random.choice([5.0,0.0]))
```

```
@api.get("/getVsel1")
async def getVsel1():
    return float(random.choice([5.0,0.0]))
```

Listado G.1 – *API developed to simulate BLAS data under normal conditions*