Universidad de Alcalá
Escuela Politécnica Superior

INGENIERÍA EN TECNOLOGÍAS DE TELECOMUNICACIÓN

Trabajo Fin de Grado

Radioespectrómetro solar basado en Software Defined Radio (SDR)

Autor: Diego Javier Abuelo Suárez

Tutor y cotutor (en su caso): Manuel Prieto Mateo

2022

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

Grado en Ingeniería en Tecnologías de Telecomunicación

Trabajo Fin de Grado

Radioespectrómetro solar basado en Software Defined Radio (SDR)

Autor: Diego Javier Abuelo Suárez

Tutor/es: Manuel Prieto Mateo

TRIBUNAL:

Presidente: Álvaro Perales Eceiza

Vocal 1º: Agustín Martínez Hellín

Vocal 2º: Manuel Prieto Mateo

F**ECHA**: 13/09/2022

Índice

1 Índice de figuras

2 Glosario

Abreviatura	Significado
TCP	Transmision Control Protocol (Protocolo de control de transmisión)
PCB	Printed Circuit Board (Placa de Circuito Impreso)
USRP	Universal Software Radio Peripheral (Softare Universal de Radio Periférica)
UMTS	Universal Mobile Telecommunications System (Sistema universal de telecomunicaciones móviles)
LTE	Long Term Evolution
GSM	Global System for Mobile communications
TFID	Radio-Frecuency IDentification (Identificación por Radio Frecuencia)
BTS	Base Transceiver Station (Estación base transmisora)
ARM	Advanced RISC Machine (Maquina RISC Avanzada)
COM	Communicaction Port (Puerto de comunicación)
SSH	Secure Shell Protocol (Protocolo de Shell seguro
ASCII	American Standard Code for Information Interchange
AGC	Automatic Gain Control (Control de Ganancia Automática)
UWB	Ultra Wideband (Ultra banda ancha)
DSP	Digital Signal Processor (Procesador Digital de Señal)
CME	Coronal Mass Ejections (Eyección de masa coronal)
ADC	Analog-to-Digital Converter
DAC	Digital-to-Analog Converter
SDR	Software Defined Radio
ONU	Organización de las Naciones Unidas
ISWI	International Space Weather Initiative

3 Resumen en español

El trabajo trata sobre realizar un lector de espectro solar utilizando un dispositivo de bajo coste basado en los dispositivos SDR (Software Defined Radio). Estos dispositivos procesan digitalmente la señal que leen a través del puerto de conexión de la antena. La idea final es sustituir un dispositivo que hay actualmente en la universidad de Alcalá por otro basado en esta tecnología.
Por ello se realiza un estudio de los dispositivos SDR que hay en el mercado y se termina eligiendo el basado en RTL-SDR. Se trata del dispositivo más barato y eficiente del mercado con estas características. Una de las premisas de este trabajo es que tiene que ser de bajo coste, por ello se elige también este dispositivo de precio reducido. Este SDR junto con una Raspberry Pi serán los encargados de leer el espectro electromagnético. Aparte de estos dos dispositivos, también será necesario tener un ordenador con sistema operativo Windows para poder ejecutar ahí el programa e-callisto y procesar todo lo leído con el SDR y la Raspberry.
El objetivo del trabajo es desarrollar un código que realice las medidas utilizando el SDR. Hay varias versiones del trabajo que se irán estudiando más en detalle. A continuación, se expone para qué sirve cada versión:

- `v0.0.0`. Esta versión inicial es para leer los datos que envía e-callisto utilizando un programa de terceros. Aquí no se programa nada en Python.
- `v0.0.1`. Esta es la primera versión que se ejecuta en la Raspberry Pi. En esta versión se realiza una lectura de un pequeño espectro de 4,8 MHz de ancho de banda y se verifica que se ha leído correctamente.
- `v0.0.2`. Esta es la primera versión en Python que se ejecuta en el ordenador Windows. Este ordenador será el encargado de realizar una conexión serie con el programa y procesar los datos que el programa e-callisto envíe a través del puerto serie. En esta versión simplemente se realiza lo mismo que la versión v0.0.0, pero realizado todo en Python.
- `v0.0.3`. Esta versión se vuelve a ejecutar en la Raspberry Pi. Es de las versiones más importantes que va a tener este trabajo ya que es la primera que se encarga de leer el espectro desde 45 hasta 870 MHz. Este es el espectro completo que vamos a tener que leer en la versión final. Es por ello por lo que aquí se lee y se dibuja en una gráfica para ver que los datos leídos concuerdan con los datos reales y esperados. Esta versión tiene una gran contrapartida y es el tiempo que tarda en ejecutarse. Mientras que el tiempo ideal anda en torno a los pocos segundos, esta versión tarda en ejecutarse más de dos minutos.
- `v0.0.4`. Esta versión se ejecuta en el ordenador central. Es el primer intento para leer y procesar las muestras que envía e-callisto a través del puerto serie. En la versión `v0.0.2`, simplemente creamos la conexión serie y probamos que se había creado correctamente, nada más. Pero en esta versión ya se intenta leer lo que se envía.
- `v0.1`. Hay un gran cambio en esta versión con respecto a las versiones anteriores. Se trata de la primera versión que lee e interpreta los datos que se envían a través del puerto serie. Es la versión más compleja hasta el momento y se encarga de procesar todo lo que leemos y nos interesa utilizar en este trabajo. Para realizar esta ha habido que analizar registros reales para ver cómo procesaban estos datos. A esta versión solo le falta poder ver los datos que se envían a través del puerto serie en la gráfica, cosa que se verá en futuras versiones. La ejecución de este script hace que e-callisto arranque sin problemas.
- `v0.2`. Esta versión es una evolución de la anterior. En este caso se le añade la funcionalidad de poder ver las gráficas en el programa e-callisto. Los datos que se envían son datos estándar o datos de prueba para ver que se está representando lo que el usuario desea. Una vez tenemos esta versión funcional, ya el último paso es realizar esta versión, pero leyendo datos reales a través del SDR.
- `v0.2.1`. Es la última versión que se ejecuta únicamente en la Raspberry Pi. Esta versión trata de leer de manera eficiente los datos del SDR. La gran contrapartida de utilizar un RTL-SDR es que no tiene una gran capacidad de procesamiento. Es por ello por lo que hay que buscar una manera eficiente de leer los datos, cosa que se realiza en esta versión. Estos datos se leerán y se representarán para comprobar que se están leyendo los datos de manera eficiente y rápida.

- `v0.3.0`. Esta versión se utiliza simplemente para comprobar la conexión TCP entre el ordenador central y la Raspberry donde se aloja el SDR. Se realiza la conexión y se mide un pequeño espectro para ver que todo funciona correctamente.
- `v0.3.1`. Se trata de la versión final. Esta versión es capaz de leer el puerto serie, comunicarse con e-callisto, procesar todo lo que se envía y enviar datos reales leídos con el SDR en tiempo real. Se trata del objetivo del trabajo y se llega en esta versión.

4 Resumen en inglés

The project deals with the realization of a solar spectrum reader using a low-cost device based on SDR (Software Defined Radio) devices. These devices digitally process the signal they read through the antenna connection port. The final idea is to replace a device currently available at the University of Alcalá with another one based on this technology.
Therefore, a study of the SDR devices on the market was carried out and the RTL-SDR based device was chosen. This is the cheapest and most efficient device on the market with these characteristics. One of the premises of this project is that it has to be low cost. This SDR together with a Raspberry Pi will be in charge of reading the electromagnetic spectrum. Apart from these two devices, it will also be necessary to have a computer with Windows operating system to run the e-callisto program there and process everything read with the SDR and the Raspberry.
The objective of the project is to develop a code that performs the measurements using the SDR. There are several versions of the project that will be studied in more detail. The following is a description of what each version is used for:

- `v0.0.0`. This initial version is for reading the data sent by e-callisto using a third-party program. Nothing is programmed here in Python.
- `v0.0.1`. This is the first version that runs on the Raspberry Pi. In this version, a small 4.8 MHz bandwidth spectrum is read and verified to be read correctly.
- `v0.0.2`. This is the first Python version to run on the Windows computer. This computer will be in charge of making a serial connection with the program and process the data that the e-callisto program sends through the serial port. This version simply does the same as version v0.0.0.0, but all done in Python.
- `v0.0.3`. This version runs again on the Raspberry Pi. It is one of the most important versions that this project is going to have since it is the first one that is responsible for reading the spectrum from 45 to 870 MHz. This is the full spectrum that we are going to have to read in the final version. That is why here it is read and plotted on a graph to see that the data read agrees with the real and expected data. This version has a big trade-off and that is the time it takes to run. While the ideal time is around a few seconds, this version takes more than two minutes to run.
- `v0.0.4`. This version runs on the host computer. It is the first attempt to read and process the samples sent by e-callisto through the serial port. In version `v0.0.2`, we simply created the serial connection and tested that it was created correctly, nothing more. In this version we already try to read what is sent.
- `v0.1`. There is a big change in this version with respect to previous versions. It is the first version that reads and interprets the data sent through the serial port. It is the most complex version so far and is responsible for processing everything we read and we are interested in using in this work. To make this one we have had to analyze real registers to see how they processed this data. This version only needs to be able to see the data sent through the serial port on the graph, something that will be seen in future versions. The execution of this script makes e-callisto start up without problems.
- `v0.2`. This version is an evolution of the previous one. In this case the functionality of being able to see the graphs in the e-callisto program is added. The data that are sent are standard data or test data to see that it is representing what the user wants. Once we have this functional version, the last step is to perform this version, but reading real data through the SDR.
- `v0.2.1`. It is the latest version that runs only on the Raspberry Pi. This version tries to efficiently read data from the SDR. The big downside of using an RTL-SDR is that it does not have a large processing power. That is why we have to find an efficient way to read the data, which is done in this version. This data will be read and plotted to verify that the data is being read efficiently and quickly.
- `v0.3.0`. This version is simply used to test the TCP connection between the host computer and the Raspberry where the SDR is hosted. The connection is made and a small spectrum is measured to see that everything is working properly.

- `v0.3.1`. This is the final version. This version can read the serial port, communicate with e-callisto, process everything that is sent and send real data read with the SDR in real time. This is the goal of the project and it is reached in this version.

5 Introducción

Este proyecto consiste en desarrollar un script de Python basándose en el uso de SDR. En concreto, se enmarca en el ámbito de los llamados radiotelescopios solares terrestres, que persiguen registrar las emisiones de radio procedentes de nuestro Sol en un rango concreto de frecuencias desde estaciones ubicadas en la superficie terrestre.

Es conocido que las emisiones de radio solares pueden ser precursoras o indicadoras de sucesos violentos en el Sol, como las llamadas CME (Coronal Mass Ejections) o fulguraciones solares. Las CMEs son espectaculares erupciones de plasma y campos magnéticos desde la superficie del Sol hacia la heliosfera[1].Las emisiones de radio registradas en las frecuencias decamétricas y métricas son el primer signo de una CME[2] y se clasifican, entre otras, en emisiones de tipo II o de tipo III[3]. Por otro lado, las fulguraciones solares también llevan asociadas emisiones de radio. Kundu et al.[4] informaron de ráfagas o estallidos de emisiones de radio de baja amplitud a bajas frecuencias (38,5, 50 y 73,8 MHz), asociadas a liberaciones de energía de baja intensidad en la corona solar.

La mejor manera de detectar las emisiones de radio solares es empleando espectrómetros, como demostró Wild[5]. Estas emisiones de radio son producto de haces de electrones, frentes de choque, posiblemente de electrones atrapados, y de ondas de alta frecuencia en el plasma. Por tanto, el estudio de estas emisiones de radio nos ayuda a comprender estos fenómenos. Los interferómetros no tienen el número suficiente de canales de frecuencia para reemplazar a un espectrómetro de alta resolución en frecuencia, ya que una emisión de radio de esta naturaleza puede tener un ancho de banda de un 1% con respecto a la frecuencia central.

e-callisto (http://www.e-callisto.org) es una cadena de estaciones de observación solar en ondas de radio repartidas por todo el mundo para poder seguir el Sol 24 horas al día. Forma parte del programa de cooperación internacional de la ONU ISWI (International Space Weather Initiative) cuyo objetivo es el desarrollo del conocimiento científico necesario para entender y predecir la meteorología espacial cercana al planeta Tierra. Este desarrollo incluye la instrumentación, el análisis de datos, modelización, educación, formación de expertos y divulgación

A pesar de que existen redes de detectores de radioemisiones solares como la de la Fuerza Aérea de los EE.UU., su cobertura no es completa y los datos no están disponibles en tiempo real para los investigadores. Para llenar este vacío, a raíz del Año Heliofísico Internacional (IHY2007), que recibió fondos de las Naciones Unidas, la NASA y la Swiss National Science Foundation, ingenieros de la Escuela Politécnica Federal (ETH) de Zürich, coordinados por el Prof. Monstein, comenzaron a distribuir un espectrómetro diseñado por él, llamado Callisto (Compound Astronomical Low cost Low frequency Instrument for Spectroscopy and Transportable Observatory), con la idea de que su bajo coste y portabilidad permitiera instalarlos en cualquier país, por pocos recursos que tuviera, en algunos casos por los institutos nacionales de astronomía y en otros por ingenieros voluntarios o profesores de secundaria. A pesar de su bajo coste, e-CALLISTO es especialmente útil para estudiar las grandes explosiones en la atmósfera del Sol conocidas como erupciones solares. Los datos científicos aportados por e-CALLISTO complementan además a las medidas registradas por instrumentos en el espacio como los que se encuentran a bordo de las misiones STEREO o WIND. Las emisiones de radio de estos sucesos son importantes para entender la dinámica de la corona solar. Las llamaradas solares son

[1] M. Aschwanden, «Physics of the Solar Corona. An Introduction,» *Physics of the Solar Corona,* 01 08 2004.

[2] D. McLean, Metrewave solar radio burst, Reino Unido: Cambridge University Press, 1985.

[3] H. Ratcliffe, E. P. Kontar y H. Reid, «Large-scale simulations of solar type III radio bursts: flux density, drift rate, duration, and bandwidth,» *A&A,* p. 572, 2014

[4] M. Kundu, S. White y P. Jackson, «Microwave observations of red dwarf flare stars,» *Advances in Space Research,* 01 01 1986.

[5] J. Wild y L. McCready, «bservatioas of the Spectrum of High-Intensity Solar Radiation at Metre Wavelengths. I. The Apparatus and Spectral Types of Solar Burst Observed,» *Australian Journal of Chemistry,* pp. 387-398, 3 3 1950.

también a menudo asociadas a eyecciones de masa coronal, enormes flujos de partículas cargadas del Sol que son un peligro para los satélites en órbita, las redes de distribución eléctrica y pueden interrumpir las señales de televisión o de los dispositivos móviles. Como las señales de radio viajan más rápido que las partículas, e-callisto también funciona como un sistema de alerta temprana para las explosiones de radio, alertando a los centros de control de misiones espaciales de las perturbaciones causadas por las próximas eyecciones de masa coronal del sol. La figura 1 muestra un ejemplo de espectrograma producido por un nodo de la red e- callisto. En ella se puede apreciar su distribución en tiempo y en frecuencia.

La Universidad de Alcalá, en colaboración con la Junta de Comunidades de Castilla-La Mancha y su Parque Científico y Tecnológico, participa en la red e- callisto aportando tres nodos. Uno se encuentra ubicado en el Edificio Politécnico de la Universidad de Alcalá, otro en la ciudad de Sigüenza y, finalmente, otro en el municipio de Peralejos de las Truchas.

5.1 SDR (Software Defined Radio)

5.1.1 Introducción

La radio definida por software (SDR) es un sistema de radiocomunicación en el que los componentes que tradicionalmente se han implementado en hardware (por ejemplo, mezcladores, filtros, amplificadores, moduladores/demoduladores, detectores, etc.) se implementan mediante software en un ordenador personal o sistema integrado[6]. Este concepto no es nuevo, pero es ahora con la mejora de los dispositivos cuando comienzan a ser un sistema para tener en cuenta en el ámbito de la radiocomunicación.
Realizar un sistema SDR puede ser tan sencillo como utilizar un ordenador o un sistema empotrado equipado con un conversor analógico digital y una antena receptora. La gran diferencia que hay entre una radio definida por software y un sistema de procesamiento por hardware, es que el procesamiento de esta señal que se recibe se puede procesar con un procesador de propósito general y no se necesita un procesador específico para realizar esa tarea (como es el caso del DSP). Importante remarcar que no es necesario de un procesador específico, pero es mejor utilizarlo al ser más rápido en las tareas de procesado.

5.1.2 Ventajas de la radio definida por software

A continuación, paso a enumerar algunas de las ventajas de los SDR frente a los sistemas de procesamiento tradicionales:

- Transmitir en el mismo lugar y a la misma frecuencia. Mediante algunas técnicas (UWB y espectro ensanchado) se puede transmitir desde un mismo sitio y a la misma frecuencia. Esto se logra gracias a algunas técnicas de corrección de error que evitan los errores causados por las interferencias que se derivan.
- Permiten detectar transmisiones más débiles. Gracias a que las antenas definidas por software "fijan" de forma adaptativa una señal direccional, de modo que los receptores pueden rechazar mejor las interferencias procedentes de otras direcciones, lo que permite detectar las transmisiones más débiles.

Las técnicas de espectro ensanchado y banda ultra ancha permiten que varios transmisores transmitan en el mismo lugar y en la misma frecuencia con muy pocas interferencias, normalmente combinadas con una o más técnicas de detección y corrección de errores para solucionar todos los errores causados por esas interferencias.
Las antenas definidas por software "fijan" de forma adaptativa una señal direccional, de modo que los receptores pueden rechazar mejor las interferencias procedentes de otras direcciones, lo que permite detectar las transmisiones más débiles.
Técnicas de radio cognitiva: cada radio mide el espectro en uso y comunica esa información a otras radios que cooperan, de modo que los transmisores pueden evitar las interferencias mutuas seleccionando las frecuencias no utilizadas. Alternativamente, cada radio se conecta a una base de datos de geolocalización para obtener información sobre la ocupación del espectro en su ubicación y, de forma flexible, ajusta su frecuencia de funcionamiento y/o su potencia de transmisión para no causar interferencias a otros servicios inalámbricos.
El ajuste dinámico de la potencia de transmisión, basado en la información comunicada por los receptores, reduce la potencia de transmisión al mínimo necesario, lo que disminuye el problema de las distancias cortas y reduce las interferencias a otros, y alarga la vida de la batería en los equipos portátiles.
Red de malla inalámbrica en la que cada radio añadida aumenta la capacidad total y reduce la potencia necesaria en cualquier nodo [3]. Cada nodo transmite utilizando sólo la potencia necesaria para que el mensaje salte hasta el nodo más cercano en esa dirección, reduciendo el problema de las distancias cortas y las interferencias con otros.

[6] M. Dillinger, K. Madani y N. Alonistoti, Software Defined Radio: Architectures, Systems and Functions, Wiley & Sons, 2003.

5.1.3 Estudio de las herramientas basadas en SDR disponibles en el mercado

En el mercado hay disponibles distintas opciones para utilizar receptores SDR. En el caso del trabajo, una de sus premisas es que sea un sistema barato y eficiente. A continuación, se explorarán distintos dispositivos SDR que hay en el mercado y sus ventajas e inconvenientes:

5.1.3.1 RTL-SDR

Figura 1: Ejemplo de dispositivo RTL-SDR

Se trata de la solución más barata del mercado. Es un dispositivo cuyo propósito cuando se creó era ver la televisión a bajo coste en medio del campo. Un ingeniero descubrió que estos dispositivos podían ser modificados para leer diferentes espectros de una forma barata y eficiente.

Ventajas

- Muy barato. Un dispositivo de estas características lo puedes obtener por unos 50€.
- Es muy usado por lo que hay una gran comunidad detrás que se encarga de dar soporte.

Inconvenientes

- Al ser tan barato, sus características son muy limitadas.

5.1.3.2 HackRF

Figura 2: Ejemplo de dispositivo HackRF

HackRF es una gran opción para los principiantes. Es de código abierto, incluyendo su diagrama esquemático, el diagrama de PCB, el código del controlador y el firmware del chip. También soporta frecuencias de 1MHz-6Ghz. HackRF sólo es capaz de transmitir y recibir en semidúplex, lo que es un gran inconveniente para los sistemas de alto rendimiento.

5.1.3.3 USRP

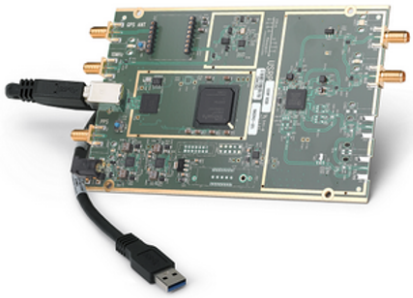


Figura 3: Ejemplo de dispositivo USRP

El hardware, el firmware y el código de USRP son de código abierto, lo que lo convierte en una excelente opción para los desarrolladores. USRP tiene varios modelos con diferentes interfaces y tamaños:

- La serie USRP X utiliza una interfaz Ethernet 10g.
- La serie USRP N utiliza Ethernet iG.
- La serie USRP B utiliza una interfaz USB 2.0 (antigua) y USB 3.0 (nueva).
- Por último, la serie USRP E tiene un procesador ARM integrado y no necesita de otro ordenador.

5.1.3.4 BladeRF

Figura 4: Ejemplo de dispositivo BladeRF

BladeRF es un hardware de alto rendimiento para el SDR para Hackers. A diferencia de HackRF, es full-duplex, lo que lo hace ideal para aplicaciones de alto rendimiento como OpenBTS (OpenBTS es una estación base móvil de código abierto). Su único inconveniente es su rango de frecuencias ya que BladeRF sólo es capaz de operar en radiofrecuencias de hasta 3,8Ghz.

5.1.3.5 LimeSDR

Figura 5: Ejemplo de dispositivo LimeSDR

LimeSDR es una plataforma SDR de código abierto, habilitada para aplicaciones. Es capaz de enviar y recibir transmitir UMTS, LTE, GSM, LoRa, Bluetooth, Ziggbee, RFID, Transmisión Digital y más.
Uno de los puntos fuertes de LimeSDR es que está habilitado para aplicaciones.
LimeSDR está integrado en el núcleo de Snappy Ubuntu y cualquiera que sea capaz de descargar y utilizar una aplicación puede utilizarlo. Esto hace que a un público mucho más amplio. Las aplicaciones disponibles para el LimeSDR incluyen:

- Radioastronomía

- RADAR
- Estación base móvil de 2G a 4G
- Streaming
- Pasarela IoT
- Radio HAM
- Emulación y detección de teclados y ratones inalámbricos
- Sistemas de control de la presión de los neumáticos
- Transpondedores de aviación
- Contadores de servicios públicos
- Mando y control de drones
- Pruebas y mediciones

5.1.4 Hardware elegido

El hardware elegido para el trabajo es:

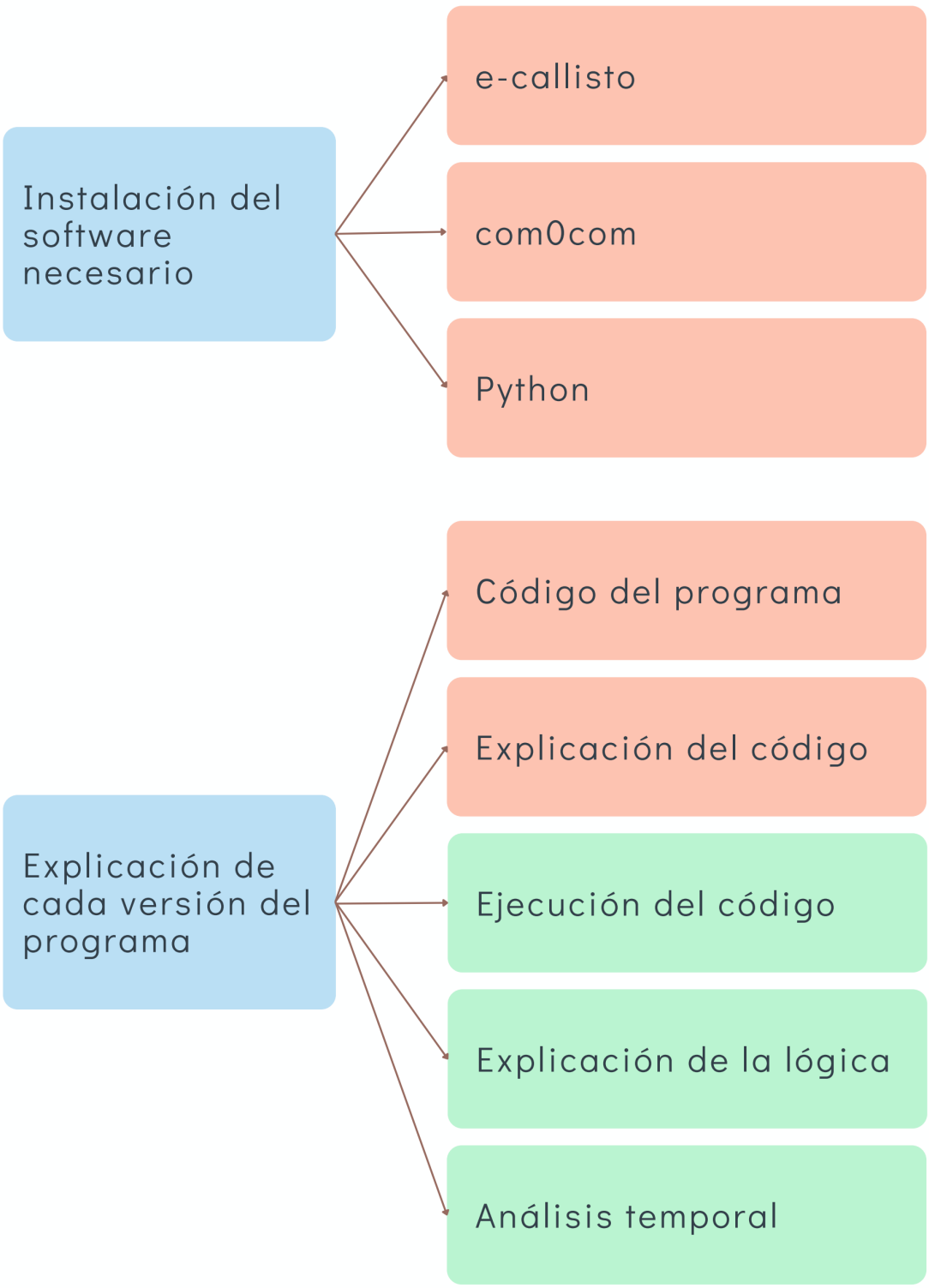
- Dispositivo RDL-SDR. Se utiliza este al ser de bajo coste, ya que es una de las premisas del trabajo
- Raspberry Pi 3 Model B. Ordenador pequeño basado en ARM que se utilizará para leer los datos que nos entregue el SDR.
- Ordenador Windows. Aquí se ejecutará e-callisto y se comunicará con la Raspberry para leer las muestras.

5.2 Objetivos y estructuración del trabajo

El objetivo de este trabajo es desarrollar un script que sea capaz de leer las muestras que se leen en un SDR para posteriormente enviarlas a e-callisto para su representación.
El trabajo se va a organizar de la siguiente forma:

1. Explicación de cómo se instalan y configuran los programas que se van a utilizar en el trabajo.
2. Explicación versión por versión del programa que se va a desarrollar para este trabajo. Se realizará así para que el lector vea cual ha sido el desarrollo completo del programa. Cada versión tendrá los siguientes puntos:
    a. Código del programa. Este punto lo tienen todas las versiones. Se trata del código del script de esa versión. Si tuviéramos más de un script, se mostrarán ambos.
    b. Explicación del código. Esta es, probablemente, la parte más importante de cada uno de los puntos del trabajo. Se trata de la explicación de qué hace cada parte del código del programa. Se explica punto a punto todo lo que hace cada una de las funciones.
    c. Ejecución del Código. Este punto no está en todas las versiones. Aquí se muestra la ejecución del código de la versión. No aparece en todas las versiones puesto que no todas tienen un código ejecutable al ser versiones de configuración y no mostrar nada visible al ejecutarse (sobre todo las primeras versiones que son simplemente de prueba).
    d. Explicación añadida de alguna lógica. En algunas versiones, es necesario añadir una explicación de la lógica de funcionamiento de ciertas partes del código. Por ello se añade este punto, para esclarecer el porqué de ciertas funciones dentro del código.
    e. Análisis de tiempo. En las versiones que se lee el espectro, es necesario realizar análisis temporales para ver la eficiencia de la lectura de las muestras con el SDR.

He decidido hacerlo así para que el lector pueda ver el proceso que he seguido al desarrollar el trabajo. De esta forma se puede ver como poco a poco se han ido añadiendo cosas a un script inicial muy simple, para llegar a una versión final funcional donde se leen las muestras en el SDR y se envían al programa e-callisto.
Toda esta estructuración se ve mucho más clara en el diagrama de bloques de a continuación:



- En rosa, los campos que aparecen en todas las versiones
- En verde, los campos que solo aparecen en algunas versiones

6 Descripción del trabajo

6.1 Instalación de los programas necesarios

6.1.1 Instalación del Programa e-callisto

En este punto, ya podemos instalar el programa e-callisto y comenzar a trabajar con él. Lo primero es descargarlo desde la página web del programa (http://www.e-callisto.org/Software/Software/Callisto-Software.html). La versión que descargaremos será la primera, el archivo `CallistoInstaller.zip`

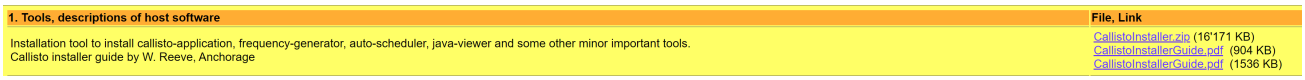


Figura 6: Descripción del enlace en la web de e-callisto

Se puede descargar también desde este enlace directamente:
http://www.e-callisto.org/Software/CallistoInstaller.zip
Con todo esto, se nos descargará un archivo `.zip`, el cual debemos descomprimir y ejecutar el instalador llamado `CallistoInstaller.exe` (como se muestra en la imagen de a continuación).

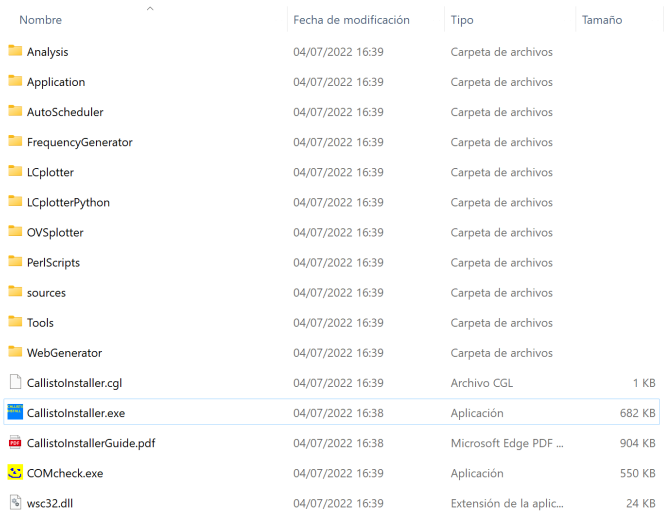


Figura 7: Captura de pantalla de la carpeta de e-callisto

6.1.1.1 Instalación de e-callisto

Para instalar el programa simplemente seleccionamos la opción que dice *Install Callisto software and configuration files.* Dejamos todas las opciones por defecto ya que editaremos lo necesario más adelante modificando el archivo de configuración.

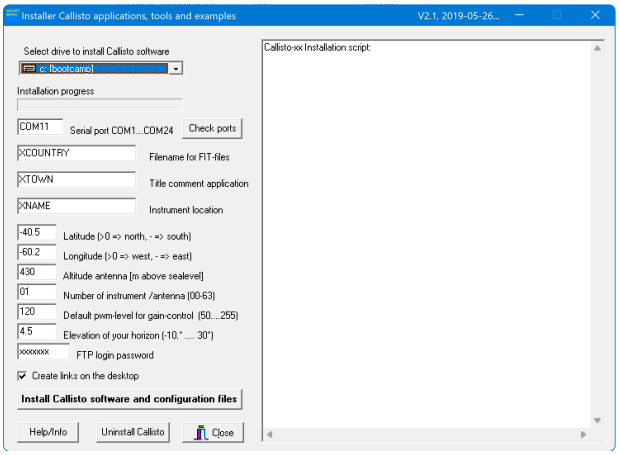


Figura 8: Captura de pantalla de la instalación de e-callisto

La configuración que viene por defecto no es la que se usará en el programa final. Por ello iremos al archivo de configuración localizado en `C://CALLISTO-01/Application` y cambiaremos el archivo llamado `callisto.cfg` por lo siguiente:

`callisto.cfg`

```
/* Callisto configuration file generated by CallistoInstaller.exe */
/* Automatically generated at 2022-01-29, 16:20:35                */
/* Author: Christian Monstein, 2017-07-28, Version 1.1            */
/*                                                                */
[rxcomport]=COM3   // Serial communication port COM1...COM18
[observatory]=12   // Instrument CALLISTO=12, please don't change
[instrument]=SPAIN-ALCALA   // Filename for FIT-file -> instrument code in the data archive
[titlecomment]=EPS  // title of the application and keyword in OVS-files
[origin]=UAH  // place of the instrument (town, village or name)
[longitude]=W,3.34             // default geographical longitude in decimal degree
[latitude]=N,40.51             // default geographical latitude in decimal degree
[height]=580.0                // default altitude [m] above sealevel
[clocksource]=1   // 1=internal clock, 2=external 1MHz 5V TTL
[filetime]=900   // observation time for one FIT-file = 15minutes
[frqfile]=frq14450.cfg   // default frequency file/program
[focuscode]=01   // focus-code = number of instrument per location
[mmode]=3   // recording mode: 2=calibrated data, 3=raw data
[fitsenable]=1   // 0=no FIT-files, 1=write FIT-files to disc
[datapath]=c:\CALLISTO-01\FITfiles\   // default data path
[logpath]=c:\CALLISTO-01\LogFiles\   // default logfile path
[lcpath]=c:\CALLISTO-01\LightCurves\   // default light curve path (LC*)
[ovspath]=c:\CALLISTO-01\OVSfiles\   // default spectral overview path (OVS*)
[chargepump]=1   // 0=pump off, 1=pump on
[agclevel]=120   // PWM level for tuner gain controil 50...255, default 120
[detector_sens]=25.4   // sensitivity of log. detector in mV/dB, default 25.4
[db_scale]=5   // dB per divison in XY_plot (1...10)
[autostart]=0   // 0=no autostart, 1=autostart after power fail
[priority]=1   // 0=NORMAL, 1=ABOVE, 2=HIGH, 3=REALTIME PRIORITY
```

Antes de instalar hay que realizar una serie de pasos para que el programa funcione, ya que no tenemos el aparato que el programa necesita. Por ello, es necesario “engañarlo” para que el programa lea correctamente la información que solicite.

6.1.1.2 Simular puertos serie

Es hora de simular puertos serie. Para ello, hay que instalar dos programas:

- El primero es *Null Modem Emulator* o *com0com*. Se puede descargar de la siguiente url: Null-modem emulator (com0com) - virtual serial port driver for Windows (sourceforge.net).
  Se trata de un programa de código abierto donde podremos simular puertos series para conectar el programa e-callisto a uno de los puertos creados y leer los datos en el otro de los puertos creados. Es

decir, si creo los puertos COM5 y COM6, con conectar el programa e-callisto al COM5 podría leer los datos conectándome al COM6 (así como enviar datos).

- El otro programa que hay que instalar es el *COM Port data Emulator*. Se puede descargar desde aquí: COM Port Data Emulator
  Este programa sirve para leer los datos que se reciben a través del puerto serie creado con el programa antes descrito. Se trata de un programa que se usará temporalmente en el desarrollo de este trabajo, ya que la idea final es hacerlo todo desde un script de Python.

6.1.1.3 Configuración Com0Com

Para configurar el programa *Com0Com* hay que crear dos puertos virtuales conectados (en mi caso COM3 y COM4). Para ello hay que abrir el programa y añadir un nuevo par de puertos (*add pair*). Esto creará dos puertos nuevos que llamaremos COM3 y COM4.
De todas las opciones que aparecen para seleccionar, solo dejaremos seleccionadas las dos primeras en ambos puertos (*use Port Class* y *Emulate baudrate*).
En la captura de a continuación se muestra como debe quedar la configuración del programa.

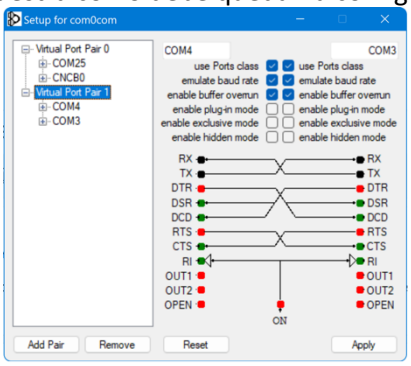


Figura 9: Captura de pantalla de la configuración de com0com

Estos dos puertos creados son los que usaremos en un futuro para conectar el programa e-callisto y el programa para leer los datos.

6.1.1.4 Configuración COM Port Data Emulator

La configuración de este programa va a depender de como hayamos configurado el Com0Com. Tomaré en cuenta la configuración creada con los puertos COM3 y COM4.

1. Abrimos el programa y nos dirigimos a la pestaña *Device*. Una vez aquí, seleccionamos la primera opción que dice *Serial Port* y, en la primera columna ponemos el número de puerto al que nos conectamos, que en mi caso es el 4.
2. El resto de la configuración es la paridad y el *baudrate* que lo dejaremos como 9600,8,N,1.
3. El desplegable de la derecha lo dejaremos en *None*.

Debería quedar tal que así:

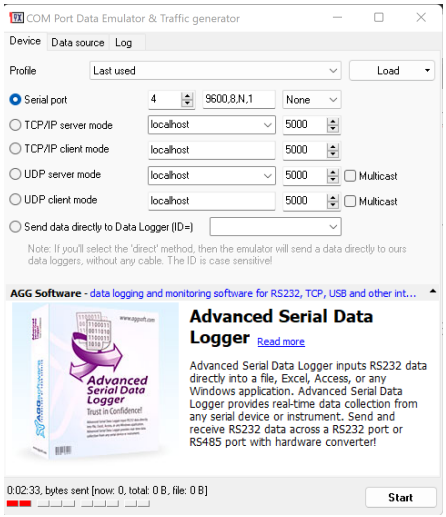


Figura 10: Captura de pantalla de la configuración de COM Port Data Emulator

Con todo esto ya debería estar configurado correctamente. Solo nos queda configurar el programa e-callisto y probar que la comunicación entre los puertos creados es correcta.

6.1.1.5 Configuración e-Callisto

Para acceder a la configuración de e-callisto, hay que ir a la carpeta de instalación (`C://CALLISTO-01/Application`) y acceder al archivo de configuración `callisto.cfg`.

Aquí habrá que cambiar el puerto de comunicación por uno de los dos puertos que hemos creado en el apartado anterior. Es decir, si hemos creado los puertos COM3 y COM4 debemos poner el puerto COM4. Dejaremos el resto de la configuración tal y como está. debería de quedar algo como lo siguiente:

`callisto.cfg`

```
/* Callisto configuration file generated by CallistoInstaller.exe */
/* Automatically generated at 2022-01-29, 16:20:35                */
/* Author: Christian Monstein, 2017-07-28, Version 1.1            */
/*                                                                */
[rxcomport]=COM3   // Serial communication port COM1...COM18
[observatory]=12   // Instrument CALLISTO=12, please don't change
[instrument]=SPAIN-ALCALA   // Filename for FIT-file -> instrument code in the data archive
[titlecomment]=EPS  // title of the application and keyword in OVS-files
[origin]=UAH  // place of the instrument (town, village or name)
[longitude]=W,3.34             // default geographical longitude in decimal degree
[latitude]=N,40.51             // default geographical latitude in decimal degree
[height]=580.0                // default altitude [m] above sealevel
[clocksource]=1   // 1=internal clock, 2=external 1MHz 5V TTL
[filetime]=900   // observation time for one FIT-file = 15minutes
[frqfile]=frq14450.cfg   // default frequency file/program
[focuscode]=01   // focus-code = number of instrument per location
[mmode]=3   // recording mode: 2=calibrated data, 3=raw data
[fitsenable]=1   // 0=no FIT-files, 1=write FIT-files to disc
[datapath]=c:\CALLISTO-01\FITfiles\   // default data path
[logpath]=c:\CALLISTO-01\LogFiles\   // default logfile path
[lcpath]=c:\CALLISTO-01\LightCurves\   // default light curve path (LC*)
[ovspath]=c:\CALLISTO-01\OVSfiles\   // default spectral overview path (OVS*)
[chargepump]=1   // 0=pump off, 1=pump on
[agclevel]=120   // PWM level for tuner gain controil 50...255, default 120
[detector_sens]=25.4   // sensitivity of log. detector in mV/dB, default 25.4
[db_scale]=5   // dB per divison in XY_plot (1...10)
[autostart]=0   // 0=no autostart, 1=autostart after power fail
[priority]=1   // 0=NORMAL, 1=ABOVE, 2=HIGH, 3=REALTIME PRIORITY
```

Con todo esto ya deberíamos tener todo configurado para comenzar a trabajar con los puertos serie creados anteriormente.

6.2 Version 0.0.0 (v0.0.0): Leer los datos enviados por el programa e-callisto

La primera versión no es una versión como tal, es simplemente una forma de asegurarnos que los puertos creados con el programa com0com están correctamente configurados junto con el programa e-callisto. Sabiendo esto, debemos seguir los siguientes pasos:

1. Abrir el programa COM Port Data Emulator.
2. Una vez abierto, ir a la pestaña log y clickar el botón de start.
3. Con el programa COM Port Data Emulator abierto, abrimos e-callisto y lo dejamos arrancar.

Si todo ha ido bien, mientras el programa e-callisto se está abriendo deberían empezar a aparecer datos en el COM Port Data Emulator. Debería verse algo como lo de a continuación:

Figura 11: Captura de pantalla de COM Port Data Emulator antes de abrir e-callisto

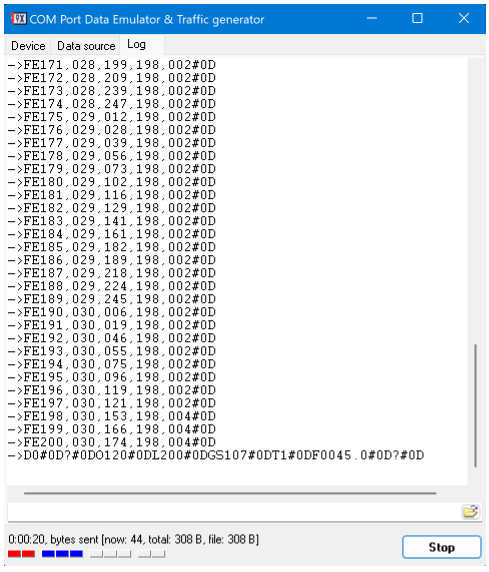


Figura 12: Captura de pantalla de COM Port Data Emulator cuando e-callisto ha terminado de arrancar

Lo que mostrará el programa e-callisto no es importante de momento, pero debería mostrar algo como lo siguiente:

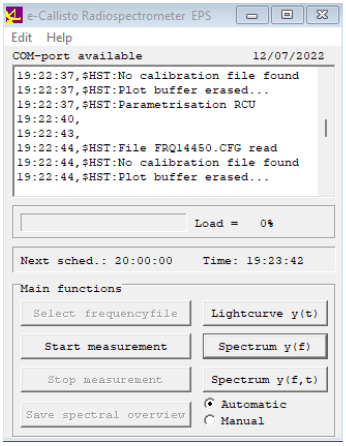


Figura 13: Captura de pantalla de e-callisto durante el funcionamieno de COM Port Data Emulator

Con todo esto ya sabemos que los puertos han sido creados correctamente y se comunican entre ellos sin problema. Lo siguiente ya será comenzar a trabajar en Python.

6.3 Versión 0.0.1 (v0.0.1): Primera lectura con el SDR

Lo primero que debemos hacer antes de comenzar la conexión entre e-callisto y el script de Python es probar que somos capaces de leer datos a través del SDR.
Hay varias opciones de hacerlo, como instalar un programa de SDR (de los muchos que se pueden encontrar en internet) o probarlo a través de Python. En este trabajo se optará por la segunda opción.

6.3.1 Desarrollo en Python

Para la lectura de datos del SDR es necesario un hardware específico para ello. El hardware utilizado en este trabajo está basado en componentes low-cost entre los que encontramos:

- Raspberry Pi 3 Model B. Este miniordenador basado en ARM se puede encontrar a un bajo precio en el mercado. Se trata de un producto que ronda los 60€ en cualquier tienda especializada.
- RTL-SDR. El RTL-SDR es un dongle que será el encargado de leer el espectro y procesarlo con ayuda de la Raspberry Pi 3. Es otro producto low-cost ya que se puede encontrar por unos 55€ en el mercado.

Con este hardware nos queda desarrollar un script básico y ejecutarlo en la Raspberry Pi para probarlo.

6.3.1.1 Conexión a la Raspberry

La conexión a la Raspberry Pi se realiza a través de un túnel seguro utilizando el protocolo SSH. En mi caso, utilizaré el programa MobaXTerm para conectarme a la Raspberry.
El programa se puede descargar desde la siguiente página web (MobaXterm) y es completamente gratuito.
La forma de conectarse a la Raspberry es la siguiente:

1. Abrir el programa MobaXterm
2. Seleccionar la opción session situada arriba a la izquierda.

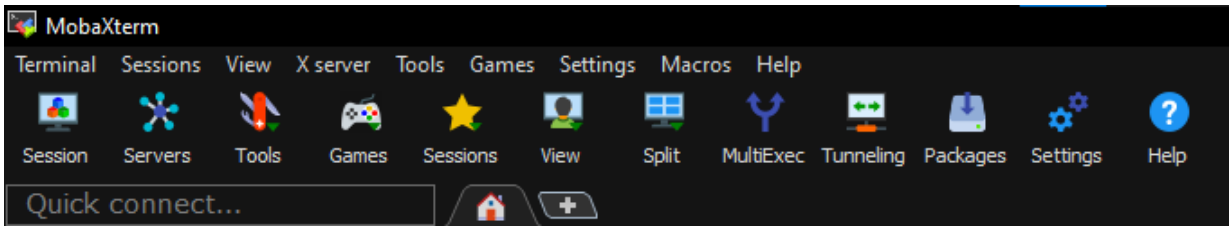


Figura 14: Captura de pantalla de la barra superior de MobaXTerm

3. Una vez seleccionada, elegir la opción SSH y rellenar los campos requeridos. En mi caso son los siguientes:
    - Remote Host: 192.168.1.23
    - Specify Username: pi
    - Port: 22

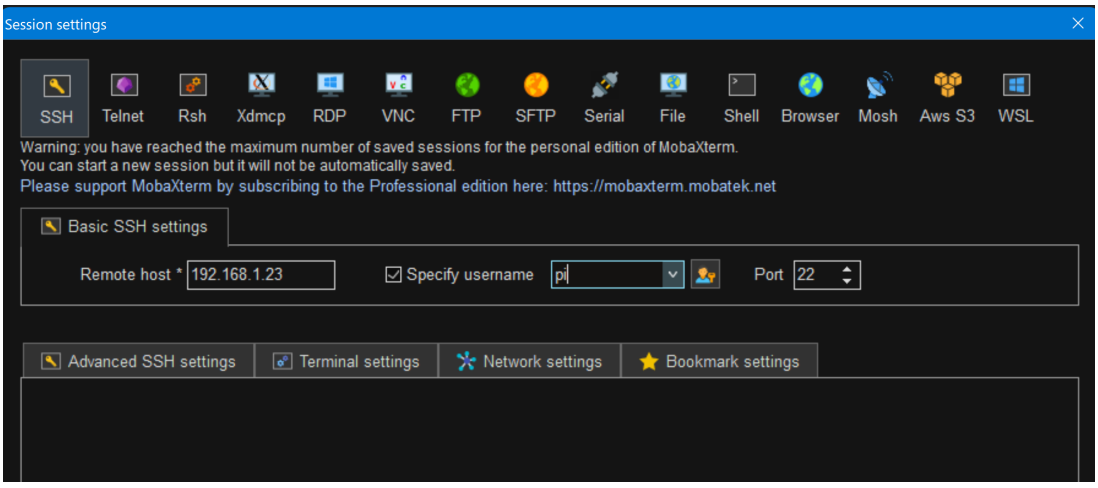


Figura 15: Captura de pantalla de MobaXTerm antes de realizar la conexión

4. Y seleccionar el botón de ok de abajo. Esto abrirá una pestaña y establecerá la conexión con el dispositivo remoto.

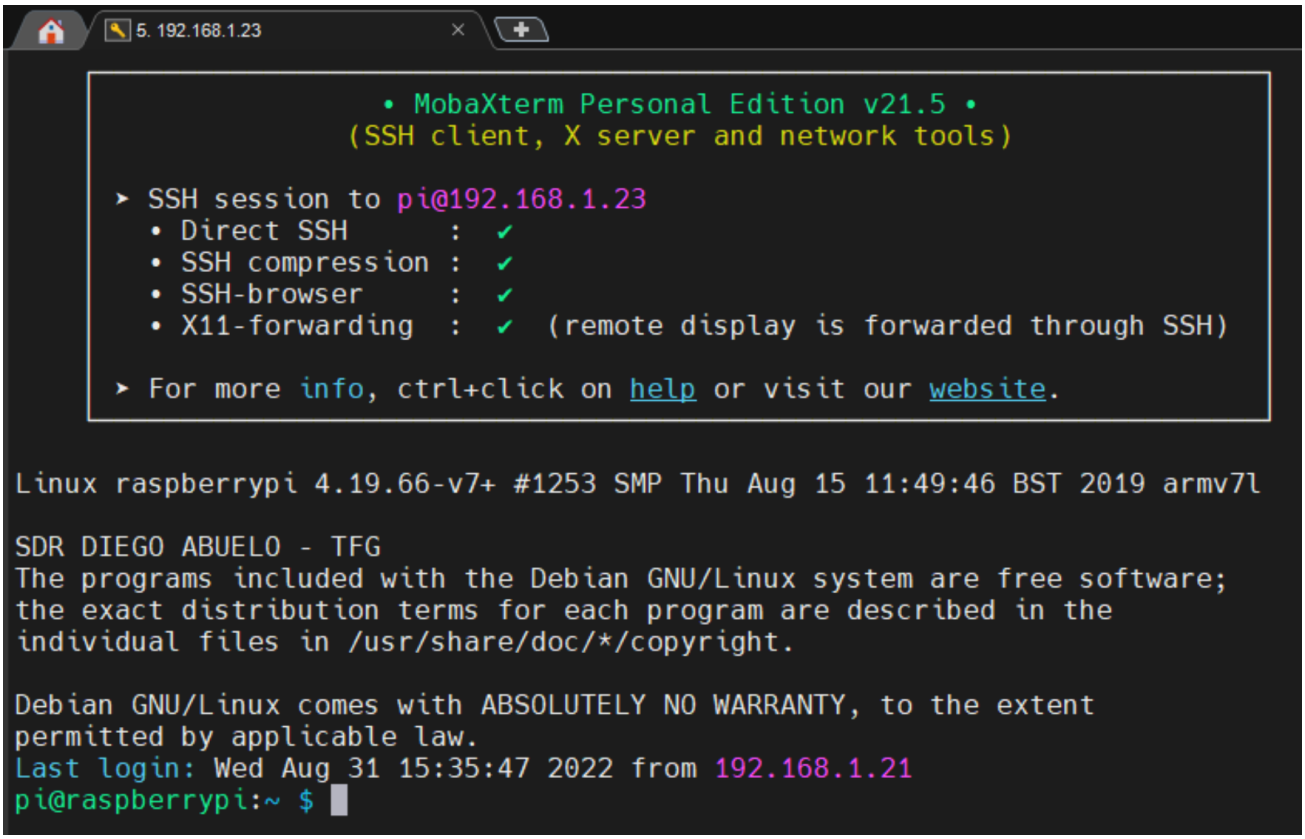


Figura 16: Captura de pantalla de MobaXTerm una vez se ha realizado la conexión SSH

Una vez dentro del dispositivo, ya lo siguiente sería ejecutar el script de Python.

6.3.2 Instalación de Python en Linux

Lo primero es instalar Python. Al estar en un Sistema Operativo basado en Linux, la instalación difiere bastante de un dispositivo Windows. La explicación de como instalar Python en Windows está explicada unas líneas más abajo (`Instalación de Python`)
Para instalar en Linux es tan sencillo como abrir una ventana de comandos del dispositivo y escribir los siguientes dos comandos:

```
$ sudo apt update
$ sudo apt install python3 idle3
```

Con esto ya debería quedar instalado Python. Para comprobar que se ha instalado correctamente escribimos el siguiente comando:

```
$ python --version
```

Y nos debería devolver algo como lo siguiente:

```
Python 3.10.5
```

Esto que devuelve depende de la versión que se instale, que en mi caso es la 3.10.5.
Con Python instalado, ya solo queda comenzar a escribir el código.

6.3.3 Instalar bibliotecas

Durante el desarrollo del programa en Python va a ser necesario instalar diferentes bibliotecas. Todas ellas se instalan de la siguiente manera:

```
$ pip3 install [nombredelabiblioteca]
```

Especificaré durante el trabajo el nombre de la biblioteca para su instalación. Asimismo, al final del trabajo dejaré una lista con todas las bibliotecas juntas para mayor comodidad.

6.3.4 Desarrollo del código en Python

Lo primero siempre al desarrollar código en Python es importar las bibliotecas necesarias. En este caso se van a necesitar tres:

- `Pyrtlsdr.` Esta biblioteca es la necesaria para comunicarse con el SDR. Es la biblioteca más importante de este trabajo junto con la que realiza la conexión serie.
- `pylab-sdk.` Esta biblioteca sirve para realizar operaciones con los datos leídos del SDR.
- `Matplotlib.` En este caso, esta biblioteca sirve para representar los datos antes leídos en el SDR.

Estas tres bibliotecas se instalan como se ha especificado unas líneas más arriba. Por ejemplo, para instalar `pyrtlsdr` habría que utilizar el comando:

```
$ pip3 install pyrtlsdr
```

Una vez están instaladas las bibliotecas, se escribe el código. En este caso, es un código estándar obtenido de la documentación de la extensión desarrollada para Python que es de la siguiente forma:

`v0.0.1.py`

```
from rtlsdr import RtlSdr
from pylab import *
from matplotlib import pyplot as plt

sdr = RtlSdr()

# configure device
sdr.sample_rate = 2.4e6
sdr.center_freq = 95e6
sdr.gain = 4

samples = sdr.read_samples(256*1024)
sdr.close()

# use matplotlib to estimate and plot the PSD
plt.psd(samples, NFFT=1024, Fs=sdr.sample_rate/1e6, Fc=sdr.center_freq/1e6)
plt.xlabel('Frequency (MHz)')
plt.ylabel('Relative power (dB)')
plt.savefig("SDR.png")
```

Teniendo este código, lo siguiente es ejecutarlo.

Para ello habrá que navegar hasta la carpeta donde se encuentra el archivo Python y escribir:

```
$ python3 v0.0.1.py
```

Al escribir eso estaremos lanzando la ejecución del script.

Al finalizar la ejecución se creará una imagen en la misma carpeta donde se encuentra el archivo `.py` que tendrá dibujado un espectro de una forma similar a la siguiente:

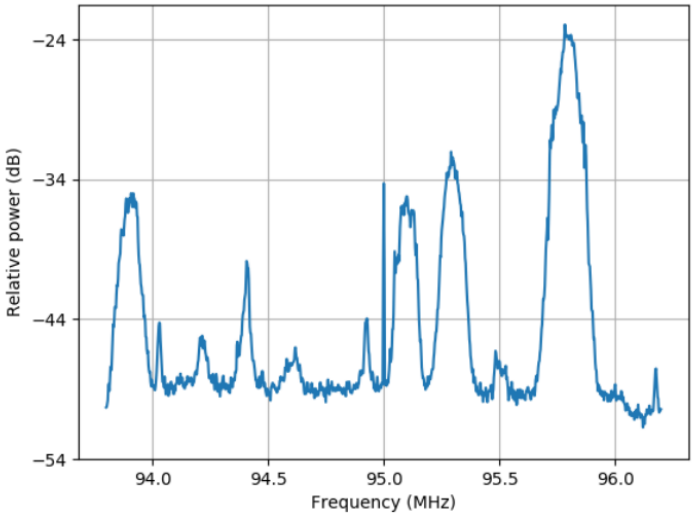


Figura 17: Espectro generado en la ejecución de la versión v0.0.1

En este caso, simplemente hemos representado un espectro con frecuencia central en 95 MHz y un ancho de banda de 2 veces la frecuencia de muestreo, o lo que es lo mismo, 4,8 MHz.
Se observa que es un espectro real y, con ello, ya podemos comenzar a trabajar en el desarrollo del script que se comunicará entre Python y el programa e-callisto.

6.3.5 Análisis de tiempo

Una de las cosas importantes para este trabajo es el tiempo que tarda en leer el espectro el SDR. Se trata de algo que hay que ir mejorando con el paso de las versiones hasta tener una lectura del espectro eficiente y rápida.
El tiempo que se tarda en leer el espectro en esta versión es de:

$$t \approx 1s$$

Se trata de un tiempo bajo, pero también es cierto que solo estamos leyendo un espectro muy pequeño. El tiempo de esta versión es algo anecdótico ya que no es el espectro final que el SDR va a leer.

6.3.6 Explicación del código

A continuación, voy a explicar paso a paso que realiza el código.
La explicación hay que dividirla en tres bloques: importar las bibliotecas, configurar el SDR y configurar la representación gráfica.

- Importar las bibliotecas. Aquí se importan las 3 bibliotecas que se van a utilizar en el script. Se usarán las bibliotecas de `rtlsdr`, `pylab` y `matplotlib`. La primera es para el SDR mientras que las otras dos son para la representación del espectro.

```
from rtlsdr import RtlSdr
from pylab import *
from matplotlib import pyplot as plt
```

- Configurar el SDR. Para configurar el SDR hay que configurar ciertos parámetros mínimos antes de ejecutar la lectura de muestras.
  Lo primero es ejecutar el comando `rtlsdr()` y asignarlo a la variable SDR. Con esto lo siguiente es configurar los parámetros necesarios para la lectura de muestras:
  - `sample_rate`. Frecuencia de muestreo. Aquí se configura la frecuencia de muestreo, que en nuestro caso es de 2,4 MHz.
  - `center_freq`. Frecuencia central. Esta es la frecuencia central de medición. En este caso de ejemplo va a ser de 95 MHz.
  - `gain`. Ganancia. La dejaremos en el valor por defecto, que es 4.

  Lo siguiente ya es ejecutar la lectura de muestras. Lo óptimo es que el número de muestras sea múltiplo de 2. En este caso se leerán 256*1024 muestras. Este valor irá cambiando a lo largo del desarrollo del trabajo en busca de una medición óptima.
  Ya lo último es cerrar el SDR. Con esto ya habríamos configurado y leído las muestras necesarias de nuestro SDR.

```
sdr = RtlSdr()

# configure device
sdr.sample_rate = 2.4e6
sdr.center_freq = 95e6
sdr.gain = 4

samples = sdr.read_samples(256*1024)
sdr.close()
```

- **Configurar la representación gráfica**. Por último, queda representar las muestras antes leídas. Para ello se utiliza la biblioteca `matplotlib`.
  Se utiliza la función `psd`, que está programada para representar las muestras en formato complejo antes leídas. Hay que especificar 4 parametros:
  - `muestras`. Aquí se especifican los parámetros a representar.

- `nfft`. Parámetro que necesita la función. Lo dejaremos en 1024, que es valor que viene por defecto.
- `fs`. Este parámetro es la frecuencia de muestreo. La definiremos en MHz, por lo que debemos dividirla entre 10^6.
- `fc`. Es la frecuencia central. Al igual que con la frecuencia de muestreo hay que representarla en MHz.

Las siguientes dos funciones son para poner texto en los ejes a representar:

- `xlabel`. Como su nombre hace intuir, se trata de la etiqueta en el eje x.
- `ylabel`. En este caso es la etiqueta en el eje y.

Por último, la función `savefig` es para guardar lo antes representado en un archivo. En este caso se generará un archivo en formato png llamado `sdr.png`.

```
# use matplotlib to estimate and plot the PSD
plt.psd(samples, NFFT=1024, Fs=sdr.sample_rate/1e6, Fc=sdr.center_freq/1e6)
plt.xlabel('Frequency (MHz)')
plt.ylabel('Relative power (dB)')
plt.savefig("SDR.png")
```

6.4 Versión 0.0.2 (v0.0.2): Conexión Python-callisto

A partir de aquí ya solo se trabajará con Python para leer y enviar los datos al programa. Antes de nada, hay que instalar alguna versión de Python en el ordenador.

6.4.1 Instalación de Python en Windows

Voy a explicar cómo instalar Python en un ordenador con sistema operativo Windows.
Lo primero es ir a la Microsoft Store.
Una vez ahí buscar 'Python' en el buscador, seleccionar la opción de 'Python 3.10' e instalar.
Con todo esto deberíamos tener Python instalado correctamente.

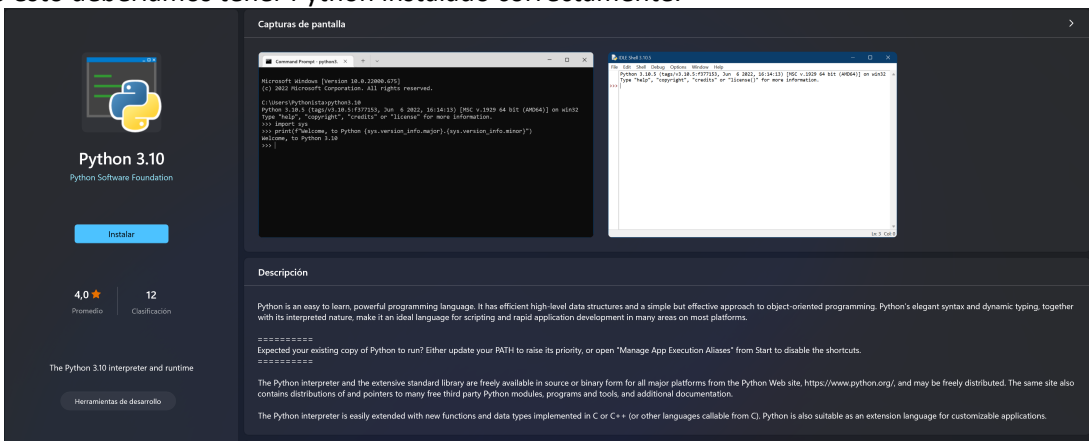


Figura 18: Captura de pantalla de la versión de Python en la tienda de Windows

Para asegurarnos que está instalado correctamente, nos abrimos una ventana de `powershell` y tecleamos lo siguiente:

```
$ python --version
```

Y nos debería devolver algo como lo siguiente:

```
Python 3.10.5
```

Esto significa que el Python se ha instalado correctamente, por lo que podremos comenzar a trabajar con él.

6.4.2 Código del programa

Ahora vamos a replicar lo hecho en la versión inicial (`v0.00`) en código de Python. Para ello lo primero es abrirnos un editor de código. En mi caso utilizaré VS Code.
Una vez abierto, comenzamos a escribir código:

6.4.2.1 Instalar bibliotecas

```
import serial
import time
```

Estas dos librerías son `pyserial` y `time`. La librería time no es necesario instalarla (viene por defecto en Python), pero la de `pyserial` es necesario instalarla. Para ello hay que escribir:

```
$ pip3 install pyserial
```

Con esto ya tendríamos las bibliotecas instaladas y podemos comenzar con el código para crear la conexión con el programa e-callisto.

6.4.2.2 Creación de la conexión con e-callisto

Lo siguiente sería conectarse con el programa e-calisto. El código es el que se muestra a continuación. Se trata de un código simple donde se crea la conexión con e-callisto. Está versión simplemente hace eso, crear la conexión y nada más. No se envían ni se leen datos a través del puerto serie.

`v0.0.2.py`

```
import time
import serial
```

```
# configure the serial connections (the parameters differs on the device you are
connecting to)
ser = serial.Serial(
    port='COM23',
    baudrate=9600,
    parity=serial.PARITY_ODD,
    stopbits=serial.STOPBITS_TWO,
    bytesize=serial.SEVENBITS
)

ser.isOpen()
```

6.4.3 Explicación del código

El código de la versión v0.0.2 viene explicado a continuación. Tiene tres partes bien diferenciadas: importar las bibliotecas, configurar el puerto serie y leer el puerto serie.

- Importar las bibliotecas. En este caso se importan 2 bibliotecas:
    - `time`. Esta biblioteca se utiliza para hacer esperas en el programa.
    - `serial`. Esta es la biblioteca más importante de este script. Es la necesaria para crear una conexión serie en Python.

```
import time
import serial
```

- Configurar el puerto serie. La configuración del puerto serie se hacen con la función `serial.serial`. Esta función tiene diferentes parámetros:
    - `port`. Esto es el puerto de conexión. Es el puerto que hayamos configurado previamente con el com0com. En este caso es el COM23 pero si cambiamos la configuración en com0com habría que cambiarlo también aquí.
    - `baudrate`. Es la tasa de envío de datos a través del puerto serie. Se mide en baudios por segundo. De momento la dejaremos en 9600.
    - `parity`. Se trata del bit de paridad. Lo dejaremos como paridad impar.
    - `stopbits`. Se especifica el/los bit/s con el que se diferencian los bytes que se envían. Los bits de stop son 2.
    - `bytesize`. Lo último a especificar es el tamaño de los bytes. En este caso lo dejaremos en 7 bits.

  Todos estos parámetros se irán modificando con el paso de las versiones del programa. No son definitivos.
  La última función que se utilizará es la de `isopen()`. Es para comprobar que el puerto está abierto. Con esto ya estaría configurado el puerto y se podría empezar a enviar y recibir datos a través de él.


```
# configure the serial connections (the parameters differs on the device you are
connecting to)
ser = serial.Serial(
    port='COM23',
    baudrate=9600,
    parity=serial.PARITY_ODD,
    stopbits=serial.STOPBITS_TWO,
    bytesize=serial.SEVENBITS
)

ser.isOpen()
```

6.5 Versión 0.0.3 (v0.0.3): Representación del espectro completo

Esta versión es una de las más importantes para el desarrollo completo del trabajo. Se trata de generar el espectro completo pedido. Para esta versión lo importante es que se lea el espectro y se represente correctamente. No importa la eficiencia ni el tiempo de lectura, solo que se lea.
Es por ello por lo que esta versión tarda en ejecutarse algunos minutos para solo mostrar una lectura del espectro.

6.5.1 Código del programa

`v0.0.3.py`

```
from rtlsdr import *
import matplotlib.pyplot as plt
import numpy as np

# configure device
SAMPLERATE = 2.4e6  # Hz

powe = np.array([])
freq = np.array([])


for i in np.arange(45, 871, 2.5):
    sdr = RtlSdr()

    # configure device
    sdr.sample_rate = SAMPLERATE
    sdr.center_freq = i*1e6  # Hz
    sdr.freq_correction = 60   # PPM
    sdr.gain = 'auto'  # 4
    samples = sdr.read_samples(8*1024)
    sdr.close()
    sdr = None
    # use matplotlib to estimate and plot the PSD
    power, psd_freq = plt.psd(samples, NFFT=1024, Fs=SAMPLERATE /
                              1e6)
    psd_freq = psd_freq + i
    powe = np.concatenate((powe, np.array(power)))
    freq = np.concatenate((freq, np.array(psd_freq)))
    print(i)


plt.clf()
plt.figure(dpi=1200)
plt.figure(figsize=(25, 10))
plt.plot(freq, powe)
plt.yscale('log')
plt.xlim(45, 870)
plt.xlabel('Frequency (MHz)')
plt.ylabel('Relative power (dB)')
plt.savefig('v0.0.3.png')
```

Es una evolución de la versión `v0.0.1`. La idea al desarrollar esta versión es la de realizar lo mismo que en la versión inicial para una frecuencia central, pero desplazándolo a través de todo el espectro.

6.5.2 Análisis de tiempo

En este caso, el tiempo va a ser sustancialmente mayor a la versión anterior. En este caso el tiempo de ejecución total es de:

$$t \approx 150s$$

Como se observa, el tiempo que tarda en ejecutarse esta versión es desproporcionado. Estamos ante una lectura total del espectro y se hace sin pensar en el tiempo de ejecución. Se observa que el espectro se lee correctamente por medio del SDR y nada más. La mejora de la eficiencia vendrá en futuras versiones.

6.5.3 Explicación del código

Esta versión tiene tres partes bien diferenciadas: importar las bibliotecas, lectura de las muestras y representación gráfica.

- Importar las bibliotecas. Se importan las mismas bibliotecas que para la versión `v0.0.1`. Esto significa que tendremos las bibliotecas `rtlsdr`, `pylab` y `matplotlib`.

```
from rtlsdr import *
import matplotlib.pyplot as plt
import numpy as np
```

- Lectura de las muestras. Lo siguiente será leer las muestras. Para ello configuraremos la frecuencia de muestreo en 2,4 MHz.
  Las dos variables que se crean, `power` y `freq` servirán para almacenar las muestras tanto de potencia como sus correspondientes frecuencias para una correcta representación.
  A partir de ahí hay que realizar un bucle, iterando desde 45 MHz hasta 870 MHz (el 1 extra en el código es para que lea también el valor para 870). Se va a realizar la medición en saltos de 2,5 MHz. Lo óptimo seria realizarlo en saltos del doble de la frecuencia de muestreo, pero para esta versión no es necesario optimizar nada todavía.
  Una vez dentro del bucle, se realiza lo que se hacía en la versión `v0.0.1`: se inicia el SDR, se configura una frecuencia de muestreo, una frecuencia central, una frecuencia de corrección y la ganancia. En este caso, la ganancia se cambia a 'auto' en vez de dejarlo en 4. También cambia el número de muestras que se leen por frecuencia. En este caso lo dejaremos en 8*1024 muestras.
  La clave aquí es que la frecuencia central va a ser $i*10^6$ o lo que es lo mismo, desde 45MHz a 870 a saltos de 2,5 MHz. No se realiza como en el fichero de configuración de e-callisto de momento. Simplemente se hace un barrido mas genérico para comprobar que se puede leer bien. Como se dice unas líneas mas arriba, la optimización no es importante para esta versión.

```
# configure device
SAMPLERATE = 2.4e6  # Hz

powe = np.array([])
freq = np.array([])


for i in np.arange(45, 871, 2.5):
    sdr = RtlSdr()

    # configure device
    sdr.sample_rate = SAMPLERATE
    sdr.center_freq = i*1e6  # Hz
    sdr.freq_correction = 60   # PPM
    sdr.gain = 'auto'  # 4
    samples = sdr.read_samples(8*1024)
    sdr.close()
    sdr = None
    # use matplotlib to estimate and plot the PSD
    power, psd_freq = plt.psd(samples, NFFT=1024, Fs=SAMPLERATE /
                              1e6)
    psd_freq = psd_freq + i
    powe = np.concatenate((powe, np.array(power)))
    freq = np.concatenate((freq, np.array(psd_freq)))
    print(i)
```

- Representación gráfica. Lo último que habría que hacer es la representación gráfica. Es más compleja que en la primera versión ya que hay que ir guardando en dos variables todo lo leído.
  Lo primero es guardar las muestras a representar en dos variables: `power` y `psd_freq`. Habrá que sumar la frecuencia que estamos midiendo.

Lo siguiente es usar otras dos variables auxiliares llamadas `powe` y `freq`. Habrá que ir añadiendo los elementos leídos y guardarlos en estas variables auxiliares.
Ya fuera del bucle y una vez leído todo, lo siguiente es representar lo leído en una gráfica.
Las funciones utilizadas nos son más que la configuración para realizar una correcta representación de los datos leídos previamente. Las diferentes funciones son:

- `clf()`. Sirve para limpiar la gráfica de posibles representaciones anteriores.
- `figure(dpi=1200)`. Se trata de la densidad de pixeles de la gráfica. Lo dejaremos en 1200 que es un tamaño relativamente grande.
- `figure(figsize=(25,10))`. Se trata del tamaño de la gráfica.
- `plot(freq,powe)`. Esta función es la que se encarga de representar los datos. En el eje x representaremos las frecuencias leídas mientras que en el eje y aparecerá la potencia leída en cada una de ellas.
- `yscale('log')`. En este caso, al ser datos representados en unidades logarítmicas, hay que cambiar la escala a logarítmica.
- `xlim(45,870)`. Ponemos un límite a la gráfica para que no sea excesivamente grande.
- `xlabel()`. La etiqueta que aparecerá en el eje x.
- `ylabel()`. La etiqueta que va a aparecer en el eje y.
- `savefig()`. Esta función se utiliza para guardar la figura antes representada en un archivo, en este caso `.png`.

Con todo esto ya estaría explicado el script.

```
    # use matplotlib to estimate and plot the PSD
    power, psd_freq = plt.psd(samples, NFFT=1024, Fs=SAMPLERATE /
                              1e6)
    psd_freq = psd_freq + i
    powe = np.concatenate((powe, np.array(power)))
    freq = np.concatenate((freq, np.array(psd_freq)))
    print(i)


plt.clf()
plt.figure(dpi=1200)
plt.figure(figsize=(25, 10))
plt.plot(freq, powe)
plt.yscale('log')
plt.xlim(45, 870)
plt.xlabel('Frequency (MHz)')
plt.ylabel('Relative power (dB)')
plt.savefig('v0.0.3.png')
```

6.5.4 Ejecución del código

La ejecución crea un archivo llamado `v0.0.3.png`. La imagen es la representación gráfica del espectro.

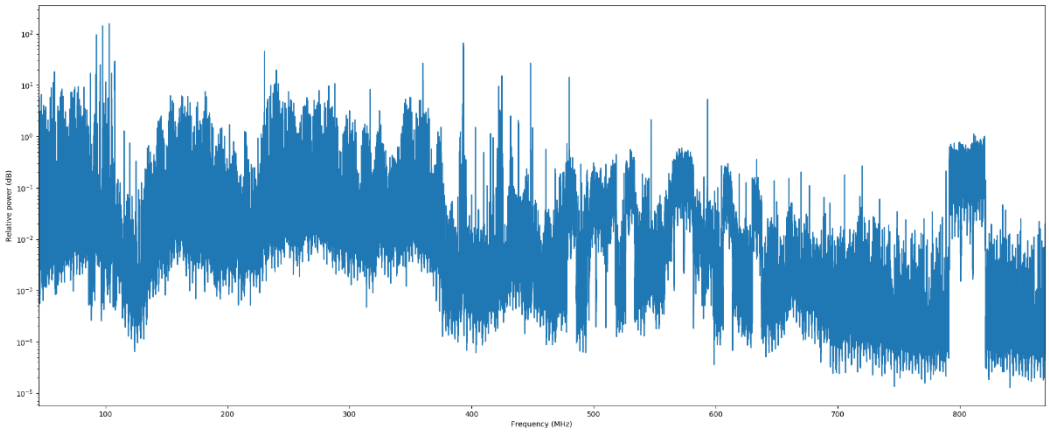


Figura 19: Espectro leído en la ejecución de la versión v0.0.3

6.6 Versión 0.0.4 (v0.0.4): Primera comunicación con e-callisto

En esta versión, el script de Python hará algo similar a lo que hacía el programa en la versión `v0.0.0` pero todo desarrollado ya en Python.
Esta versión es capaz de leer los datos enviados por el programa e-callisto, imprimirlos por pantalla y guardarlos en un archivo de texto para poder analizar con más detenimiento los registros que el programa entregue.
También es capaz de comenzar a interpretar ciertos comandos. En este caso solo interpreta uno y es el de cerrar. Esto significa que al cerrar el usuario el programa, el script se parará y no continuará ejecutándose indefinidamente.

6.6.1 Código del programa

`v0.0.4.py`

```
import time
import serial

# Configure the serial connections (the parameters differs on the device you are
connecting to)
# COMMANDS SENT FROM ECALLISTO
# U2 -> Gain control tuner, set voltage by command <Oxxx> (0120)
# U4 ->
# U6 ->
# O120 ->
# D0 -> Stop debugging mode
# D1 -> Start debugging mode
# FEx, y, z, a, b -> x = channel number, y and z are the frecuency and a and b are the
spare code.

# L200 -> Number of samples to be taken
# GS107 ->


PORT = 'COM3'

try:
    ser = serial.Serial(
        port=PORT,
        baudrate=9600,
        parity=serial.PARITY_ODD,
        stopbits=serial.STOPBITS_TWO,
        bytesize=serial.SEVENBITS
    )
except:
    print("Error al abrir el puerto " + PORT)
    exit()

ser.isOpen()



# #this will store the line
line = []
file = open('log.txt', 'w')

while True:
    for c in ser.read():
        line.append(c)
        ser.write(b'\x06')
        if c == 13:
            string_ints = [chr(int) for int in line]
            hex_ints = [str(int) for int in line]
```

```
            print("Calisto>> " + ''.join(string_ints))
            file.write("Calisto>> " + ''.join(string_ints) + '          ' + '
'.join(hex_ints) + '\r')
            if ''.join(hex_ints) == '834813':
                file.close()
                exit()
            line = []
            break
```

Con este código ejecutándose y abriendo el programa se puede comenzar a tener una comunicación entre e-callisto y el script.

6.6.2 Explicación del código

Esta versión tiene tres partes bien diferenciadas: importar las bibliotecas, configuración del puerto e interpretación de los datos.

- Importar las bibliotecas. En este caso se siguen utilizando las dos mismas bibliotecas que en versiones anteriores: `time` y `serial`.

```
import time
import serial
```

- Configuración del puerto. Lo siguiente es configurar el puerto para su comunicación. La configuración en sí es la misma que para la versión `v0.0.2`. Lo que se ha añadido en esta versión es el control de errores utilizando un `try-except`. Esto se utiliza para que, si no se puede crear el puerto por cualquier motivo, salte el `except` mostrando un mensaje de error por pantalla y cerrando el script.
  Por lo demás es todo igual, simplemente se ha creado una variable global `PORT` para configurar el puerto de una manera más cómoda.

```
PORT = 'COM3'

try:
    ser = serial.Serial(
        port=PORT,
        baudrate=9600,
        parity=serial.PARITY_ODD,
        stopbits=serial.STOPBITS_TWO,
        bytesize=serial.SEVENBITS
    )
except:
    print("Error al abrir el puerto " + PORT)
    exit()

ser.isOpen()
```

- Interpretación de los datos. Ahora llegamos a la parte importante de esta versión y donde se produce el mayor cambio con respecto a versiones anteriores.
  - Lo primero que observamos es que se crea una variable line. En esta variable se guardará el valor que vaya tomando cada 'línea' enviada por e-callisto. Se interpreta como 'línea' al código enviado separado por un retorno de carro. Este código en ASCII es el 13. Unas líneas más abajo se utilizará una comparación para ver si hemos llegado a ese final de línea e interpretar lo enviado línea a línea.
  - Lo siguiente que se observa es que se crea un archivo. Este archivo se crea en modo escritura y se titula `log.txt`. Aquí se guardará el registro de todo lo que ocurra en el programa.

  Con estas dos variables creadas previo al envío de datos, ya podemos comenzar con el programa en sí.
  - Se pondrá el programa en un bucle infinito para que esté continuamente leyendo datos. Solo parará cuando se cierre el programa. Dentro de este bucle infinito, aparece un bucle `for`, que irá leyendo los caracteres ASCII que envía e-callisto y los ira guardando en la variable line (`line.append(c)`). Cuando se encuentre el carácter 13, se puede comenzar a interpretar los datos del programa. En este programa se envia un carácter en hexadecimal `0x06`. Esto fue un

intento de comunicación con el programa para enviar datos. Si e-callisto no recibe nada, comienza a quedarse en modo espera hasta recibir algún tipo de datos. Enviando el `0x06`, el programa interpreta que hay alguien al otro lado de la comunicación y comienza a enviar datos. Lo que se envía no tiene ninguna lógica a nivel del programa en sí, pero sirve para comenzar a ver lo que se envía a través del puerto serie.

- Lo primero que se hace es convertirlo a texto legible y a hexadecimal. Se imprime por pantalla (`print()`) y se guarda en el archivo de registro (`file.write()`).
- Ya lo último es el `if` para detectar el cerrado del programa. Observando se llega a la conclusión que el código `834813` (`S0`) es lo que envía e-callisto para parar la comunicación. Simplemente interpretamos eso como fin del programa y cerramos el archivo y salimos del script. Si no se cierra, se resetea el valor de la línea (`line =[]`) y se vuelve a comenzar a leer la siguiente línea.

```
# #this will store the line
line = []

file = open('log.txt', 'w')

while True:
    for c in ser.read():
        line.append(c)
        ser.write(b'\x06')
        if c == 13:
            string_ints = [chr(int) for int in line]
            hex_ints = [str(int) for int in line]
            print("Calisto>> " + ''.join(string_ints))
            file.write("Calisto>> " + ''.join(string_ints) + '          ' +
''.join(hex_ints) + '\r')
            if ''.join(hex_ints) == '834813':
                file.close()
                exit()
            line = []
            break
```

Se trata de un código relativamente simple pero que ya comienza a mostrar lo que se envía a través del puerto serie. Esta versión será una de las patas sobre la que se forjará todo el trabajo y, sobre todo, la versión `0.1`, la que considero la más importante del trabajo y que pasaré a hablar a continuación.

6.6.3 Ejecución del código

En esta versión simplemente se responde al programa y se cierra correctamente. A continuación, se muestran un par de capturas del programa e-callisto cuando se ejecuta con este código.

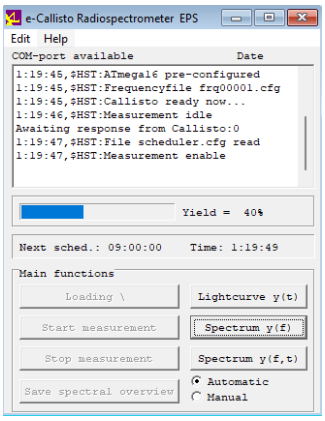


Figura 20: Captura de pantalla mientras está iniciando el programa

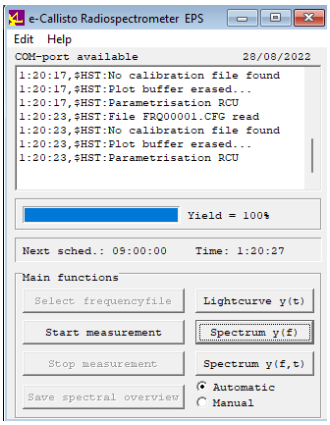


Figura 21: Captura de pantalla al terminar de iniciarse y quedarse en espera

6.7 Versión 0.1 (v0.1): Interpretación de los datos

Esta versión es la más larga y la más importante de todo el trabajo. No es la más compleja, pero si la que más tiempo me llevó desarrollar. Se trata de la primera versión capaz de comunicarse con e-callisto y de interpretar los mensajes que el programa enviaba.
Para que esta interpretación fuera posible me tuve que guiar por unos registros previamente grabados por mi tutor del TFG con un medidor real para ver qué mensajes se enviaban entre el programa y el receptor.

6.7.1 Interpretación de los registros

Como dije antes, una de las partes más importantes ha sido interpretar los registros. Ahora que somos capaces de comunicarnos con e-calisto hay que ver qué le enviamos para que el programa haga lo que deseamos.
Lo primero es irnos a la documentación oficial, que dejaré en la bibliografía. Dentro de esta documentación hay un archivo llamado `ecallistomanual.pdf` donde se encuentran todas las especificaciones del software del programa.
Este archivo explica que significa cada comando que envía e-callisto, lo que me será útil para el desarrollo de esta versión. Con todo esto, lo primero es ver que nos envía e-callisto al arrancar el programa.

6.7.1.1 Arranque de e-callisto sin enviar nada

Si lanzamos una ejecución normal, sin enviar nada desde el script de Python, recibiremos algo como lo siguiente:

```
Calisto (19:31:46)>> FE1,011,027,198,001
Calisto (19:31:46)>> FE2,011,027,198,001
Calisto (19:31:46)>> FE3,011,027,198,001
Calisto (19:31:46)>> FE4,011,027,198,001
Calisto (19:31:46)>> FE5,011,027,198,001
Calisto (19:31:46)>> FE6,011,027,198,001
Calisto (19:31:46)>> U2
Calisto (19:31:46)>> U4
Calisto (19:31:46)>> U6
Calisto (19:31:46)>> FE7,011,027,198,001
Calisto (19:31:46)>> O120
Calisto (19:31:46)>> FE8,011,027,198,001
Calisto (19:31:46)>> FE9,011,033,198,001
Calisto (19:31:46)>> FE10,011,053,198,001
```

Se observa que e-callisto nos está enviando un comando repetido muchas veces y otros comandos más cortos únicamente cuatro veces. Esta ejecución está cortada, pero la longitud de todo lo que envía llega hasta el comando `FE200…`
Con esta primera ejecución es hora de analizar el significado de todo esto:

- `FE1,011,027,198,001`. Este comando se habría que separarlo en varias partes. Si vamos al manual antes nombrado, y nos vamos a la página 19 de este observamos una tabla donde vienen explicados diferentes comandos. Si lo leemos detenidamente se puede observar lo siguiente (lo pongo traducido para que sea más claro. En el manual viene explicado en inglés):

Comando	Ex.	Descripción
Guardar la frecuencia en la EEPROM	FEx,y,z	x=número de canal, $0 \leq x < 500$, y=frecuencia [MHz], $045.000 < y < 870.000$ [TBC], z=código de repuesto $0 \leq x \leq 63$, donde z se obtiene del programa.

  Con esto podemos compararlo con lo que nos envía el programa y tenemos:
  - `FE1`. En este caso, el 1 correspondería a la x, lo que nos diría el canal. Tenemos 200 canales, predefinidos en el archivo de frecuencia, lo que correspondería correctamente con las 200 entradas que nos envía e-callisto al ejecutar.
  - `011,027`. Aunque se puede pensar que esto es la y y la z, se trata únicamente de la y, por lo que estaríamos hablando de la frecuencia. Explicar cómo se calcula la frecuencia.
  - `198,001`. Esto es el spare code o código de repuesto. Es un código que genera e-callisto y que no será importante en el desarrollo del trabajo.

- `U2`. Como antes, podemos ir a la tabla del manual en las paginas 19-21 y buscar este código. Encontraremos lo siguiente:

Comando	Ex.	Descripción
Medir voltaje AGC	U2	Control de ganancia. Ajusta el voltaje con el comando <Oxxx>

  Como pone en la descripción, hay que ajustar el voltaje con el comando `<Oxxx>`. Este comando aparecerá unos mensajes más abajo.
- `U4`. Viendo en la tabla tenemos:

Comando	Ex.	Descripción
Medir voltaje del emisor	U4	Testear el voltaje del emisor BF199

  Es simplemente un comando para ver el voltaje al que trabajar. En lo que respecta al trabajo, no es un comando importante.
- `U6`. En este caso se observa:

Comando	Ex.	Descripción
Medir voltaje de entrada	U6	10/37 del voltaje de entrada después del diodo y el fuse

  Al igual que el anterior, es otro comando para medir el voltaje y no será importante para el desarrollo del trabajo.
- `O120`. Este es el comando para ajustar el control de ganancia. Si nos vamos a la tabla tenemos:

Comando	Ex.	Descripción
Ganancia del tuner	OXXX	Voltaje PWM expresado en dígitos de 0…255 Para ajustar el valor PWM, ver la gráfica de sensibilidad. Para ver el voltaje AGC del sistema, pulsar <U2>

  Aquí se ajusta la ganancia del sistema a 120 sobre 255. Lo que es lo mismo, se deja en un valor intermedio. En este comando tampoco se entrará más en profundidad.

Hasta aquí tenemos un análisis de los primeros comandos que se envían. Si no enviamos nada, los comandos `FEx,y,z` no llegarían a enviarse completamente al no recibir respuesta de un receptor.

6.7.1.2 Arranque de e-callisto enviando byte de asentimiento

Lo siguiente es conseguir que e-callisto arranque correctamente. Para ello hay que responder a los comandos antes explicados de forma que e-callisto los interprete como respuesta.

Para saber qué enviar, el tutor del TFG me facilitó un registro de la comunicación que tenían un receptor ya funcional y el programa. El registro esta completo en el apéndice, pero a continuación se observa lo importante en este tema:

```
IRP_MJ_WRITE  DOWN   FE2,005,043,198,001.
IRP_MJ_READ   UP     $CRX:AGC gain = 2.6…
IRP_MJ_WRITE  DOWN   FE3,005,043,198,001.
IRP_MJ_READ   UP     ]
IRP_MJ_WRITE  DOWN   FE4,005,043,198,001.
IRP_MJ_READ   UP     ]
IRP_MJ_WRITE  DOWN   FE5,005,043,198,001.
IRP_MJ_READ   UP     ]
```

En este caso, DOWN es del programa e-callisto al receptor y UP del receptor al programa. Se observa que el receptor envía un `]` como respuesta de asentimiento ante lo que envía e-callisto.

Si se observa todo el registro, se ve que lo hace hasta que el programa termina de arrancar y se queda esperando a realizar alguna medición. Es justo en ese momento que e-callisto envía más comandos que se pasaran a analizar a continuación.

```
Calisto>> D0
Calisto>> ?
Calisto>> O120
Calisto>> L200
Calisto>> GS107
Calisto>> T1
Calisto>> F0045.0
```

```
Calisto>> ?
Calisto>> S0
```

Esto son los últimos mensajes que se envían antes de entrar el programa en modo espera exceptuando el último, el S0. Los comandos que aparecen son:

- `D0`. Si nos vamos a la tabla del manual nos encontramos:

Comando	Ex.	Descripción
Parar el modo depuración	D0	No enviar información adiciona a la máquina controladora.

  Esto nos indica que aquí paran de enviarse los mensajes que se envían al iniciar el programa. Aquí se queda en espera e-callisto hasta que el usuario interactúe con el programa de alguna forma.

- `L200`. Este comando se envía ya que nosotros como usuarios hemos seleccionado el botón de start measurement.

Comando	Ex.	Descripción
Longitud del barrido o numero de canales	LX	Numero de canales que se medirán en un barrido

  En este caso, medimos 200 muestras en el espectro 45-870 MHz.

- `GS107`. Otro comando que se envia después de elegir la opción de start measurement.

Comando	Ex.	Descripción
Establecer frecuencia de repetición de la máquina utilizando el reloj interno.	GSx	Establecer la frecuencia en modo asíncrono x=0-5 (elegir 1 Hz, 50 Hz, 200 Hz, 400 Hz o 800 Hz)

- `T1`. La explicación de este comando es la siguiente:

Comando	Ex.	Descripción
Desencadenar a través del temporizador	T1	El temporizador interno comienza las mediciones.

  Como dice en el manual, este comando es para que el temporizador comience a medir.

- `F0045.0`. Este comando es el último que se envía desde e-callisto al receptor antes de empezar a recibir datos.

Comando	Ex.	Descripción
Frecuencia del receptor	F0x	Establecer la frecuencia del receptor con el valor x, donde x está expresada en MHz

  Al comenzar la medición en 45 MHz, es coherente que la frecuencia a la que se comienza la medición sea esa.

- `S0`. Este comando se envía al cerrar el programa. Se puede intuir el significado de este.

Comando	Ex.	Descripción
Parar la medición	S0	Parar el modo de medición.

  Como se intuía, este comando es el que finaliza la ejecución del programa.

Una vez se ha analizado en registro que se recibe a través del puerto serie al conectarse al programa e-callisto, es momento de pasar al código.

6.7.2 Código del programa

Este es el código de esta versión. Se explicará en la siguiente sección el significado de cada uno de los puntos.

`V0.1.0.py`

```
import serial
from datetime import datetime

flag = True

def init_serial():
    try:
        ser = serial.Serial(
            port='COM4',
```

```
            baudrate=115200,
            parity=serial.PARITY_NONE,
            bytesize=serial.EIGHTBITS,
        )
    except:
        print("Error al abrir el puerto " + ser.port)
        exit()
    ser.isOpen()
    return ser


def write_serial(ser, data):
    ser.write(data)
    logfile.write("SDR    (" + datetime.now().strftime("%H:%M:%S") + ")>> " + str(data) +
'\n')
    print("SDR    >> " + str(data))

def check_close(string):
    if string == "834813":
        logfile.close()
        return True
    return False


def check_start_previous(string):
    if string == "834913": # Equivale a S1
        return True
    return False


def check_start(string):
    if string == "716913": # Equivale a GE
        return True
    return False


def check_stop(string):
    if string == "716813":
        logfile.close()
        return True
    return False


ser = init_serial()
# this will store the line
line = []

logfile = open("log.txt", "w")




while True:
    for c in ser.read():
        line.append(c)
        if c == 13:
            string_ints = [chr(int) for int in line]
            string_hex = [str(int) for int in line]
            print("Calisto>> " + ''.join(string_ints))
            print("       >> " + ''.join(string_hex) + '\n')
            logfile.write("Calisto (" + datetime.now().strftime("%H:%M:%S") + ")>> " +
''.join(string_ints))
            logfile.write("                    " + ''.join(string_hex) + '\n')
            if flag:
```

```
                write_serial(ser, bytes('$Frequency\r', encoding='utf8'))
                flag = False
            write_serial(ser, bytes([0x5d]))
            if check_close(''.join(string_hex)):
                ser.close()
                exit()
            if check_start_previous(''.join(string_hex)):
                write_serial(ser, bytes('$CRX:Started.2', encoding='utf8'))
            if check_start(''.join(string_hex)):
                f = open('string.txt', 'r')
                for sampleString in f:
                    write_serial(ser, bytes(sampleString, encoding='utf8'))
                f.close()
            line = []
            break
```

Este código así en crudo es difícil de entender, por lo que pasamos a explicarlo a continuación.

6.7.3 Explicación del código

Este código es más complejo que las versiones anteriores. Es por ello por lo que está dividido en 4 secciones en lugar de 3 como los anteriores. Las 4 secciones son: importar las bibliotecas, creación del puerto serie, funciones para interpretar los datos y bucle de comunicación.

- **Importar las bibliotecas**. En este caso hay una biblioteca más que en las otras versiones. Las bibliotecas utilizadas son:
    - `serial`. Se utiliza para escribir por el puerto serie.
    - `datetime`. Esta función es simplemente para los registros. Con esto se puede añadir formato de fecha y hora para un mejor análisis posterior.

  En este programa no se usar la biblioteca time como en versiones anteriores ya que no hay esperas controladas.

```
import serial
from datetime import datetime
```

- **Creación del puerto serie**. La principal diferencia con respecto a la versión anterior es que ahora está todo dentro de una función para una mejor estructuración del programa.
  También se observa que la paridad ahora está ajustada a `none` y el tamaño de byte se configura con tamaño de 8. El `baudrate` también se aumenta hasta los 115200 para tener más velocidad de envío. Esta es la configuración final que se utiliza en la comunicación y se ajusta en esta versión ya funcional.

```
def init_serial():
    try:
        ser = serial.Serial(
            port='COM4',
            baudrate=115200,
            parity=serial.PARITY_NONE,
            bytesize=serial.EIGHTBITS,
        )
    except:
        print("Error al abrir el puerto " + ser.port)
        exit()
    ser.isOpen()
    return ser
```

- **Funciones para interpretar los datos.** Aquí es donde viene uno de los grandes cambios con respecto a versiones anteriores. He añadido diferentes funciones para limpiar el código principal y poder reutilizar ciertos trozos de código que paso a explicar a continuación:
    - `write_serial`**.** Esta función es para enviar datos a través del puerto serie y para añadir su línea correspondiente en el registro.
    - `check_close()`**.** Esta función se utiliza para ver cuando se recibe el comando de fin de programa. El valor que compara está en ASCII.

- `check_start_previous()`**.** Está función es para poder enviar un mensaje inicial que es necesario para comenzar a leer datos. Se utiliza para ver cuando se envía un comando que significa el inicio de la lectura de datos.
- `check_start()`**.** En esta función sí que se lee cuando el usuario desea que se comience la lectura de datos.
- `check_close()`**.** Esta función es la última, pero no la menos importante. Se trata de la función que lee cuando se para la lectura de datos.

Con todo esto ya tendríamos las funciones listas para ser ejecutadas en el bucle de comunicación.

```
def write_serial(ser, data):
    ser.write(data)
    logfile.write("SDR    (" + datetime.now().strftime("%H:%M:%S") + ")>> " + str(data) +
'\n')
    print("SDR    >> " + str(data))

def check_close(string):
    if string == "834813":
        logfile.close()
        return True
    return False


def check_start_previous(string):
    if string == "834913": # Equivale a S1
        return True
    return False


def check_start(string):
    if string == "716913": # Equivale a GE
        return True
    return False


def check_stop(string):
    if string == "716813":
        logfile.close()
        return True
    return False
```

- **Bucle de comunicación.** En este bucle es donde se realiza la comunicación entre e-callisto y el script. Al igual que en la versión anterior, se lee línea a línea separando una de ellas al leer el carácter con valor ASCII 13. A partir de aquí es donde cambia todo con respecto a la versión previa.
  - Lo primero que se hace es interpretar el mensaje tanto en texto claro (`string_ints`) como en hexadecimal (`string_hex`) y se imprime por pantalla aparte de guardarlo en el registro.
  - Después se ve que hay un `flag`. Esto es para enviar un mensaje nada más empezar distinto del `0x5D` de asentimiento. Este mensaje es necesario para que el programa detecte el receptor como conectado y aparezca arriba a la izquierda un mensaje de 'receiver connected'. Una vez enviado se desactiva el `flag` y ya no se utiliza más.
  - Debajo se ve que se envía el `0x5D` de asentimiento cada vez que e-callisto envía algo. Esto ya lo hacia la versión `v0.0.4`.

  Hasta aquí es todo lo relativo al inicio del programa. A partir de esta línea, el programa consiste en comprobar si se inicia o no la lectura de datos.
  - Si se lee que el comando de fin del programa, se finaliza la ejecución.
  - Si se lee el `check_start_previous()` se envía la cadena de texto que es necesaria al leer ese comando. Esta cadena la he obtenido mirando los registros enviados por mi tutor.
  - Por último, si se lee el `check_start()` se abre un archivo de datos crudos y se empiezan a enviar.

Hay que entender un punto muy importante de esta versión y es que no es capaz de observarse ninguna grafica en el programa. Si se abre cualquiera de las 3 opciones que implementa este programa no se observa nada. Esto será solucionado en la versión `v0.3`.

```
while True:
    for c in ser.read():
        line.append(c)
        if c == 13:
            string_ints = [chr(int) for int in line]
            string_hex = [str(int) for int in line]
            print("Calisto>> " + ''.join(string_ints))
            print("       >> " + ''.join(string_hex) + '\n')
            logfile.write("Calisto (" + datetime.now().strftime("%H:%M:%S") + ")>> " +
''.join(string_ints))
            logfile.write("                      " + ''.join(string_hex) + '\n')
            if flag:
                write_serial(ser, bytes('$Frequency\r', encoding='utf8'))
                flag = False
            write_serial(ser, bytes([0x5d]))
            if check_close(''.join(string_hex)):
                ser.close()
                exit()
            if check_start_previous(''.join(string_hex)):
                write_serial(ser, bytes('$CRX:Started.2', encoding='utf8'))
            if check_start(''.join(string_hex)):
                f = open('string.txt', 'r')
                for sampleString in f:
                    write_serial(ser, bytes(sampleString, encoding='utf8'))
                f.close()
            line = []
            break
```

6.7.4 Ejecución del código

La ejecución del código hace que el programa e-callisto arranque sin problemas. Y se cierre sin problemas. A continuación, se muestran unas capturas del programa al ejecutar el script y, después, abrir el programa.

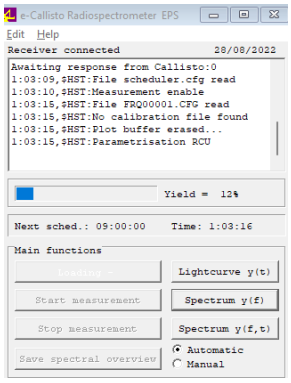


Figura 22: Captura de pantalla mientras está iniciando el programa

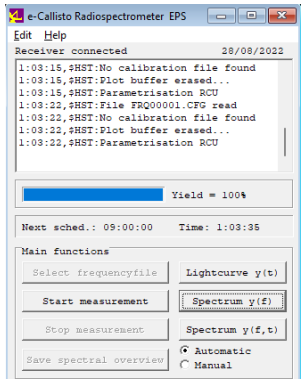


Figura 23: Captura de pantalla al terminar de iniciarse y quedarse en espera

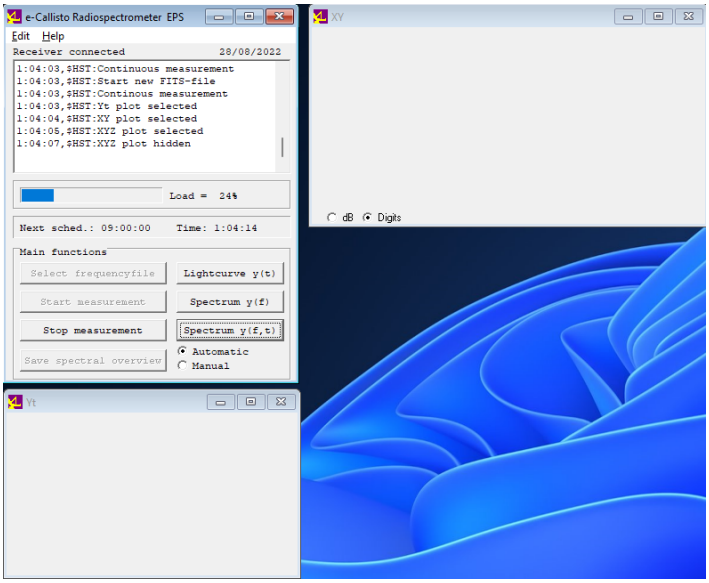


Figura 24: Captura de pantalla al iniciar la medición de datos y abrir las gráficas

Con esto se ve que el programa se está ejecutando de acuerdo con lo explicado.

6.8 Versión 0.2 (v0.2): Representación gráfica en e-callisto

La principal diferencia entre esta versión y la anterior es la posibilidad de ver una representación gráfica de los datos. Para poder ver la representación gráfica no solo es necesario enviar datos (eso ya se hacía en la versión anterior y no se visualizaba nada), también hay que añadir un carácter al mensaje de inicio para que e-callisto lo interprete correctamente. Este carácter es un `\r`.
Simplemente añadiendo esto, el programa interpreta que lo que se envía a partir de ese punto son datos de mediciones de la antena.

6.8.1 Formato de envío de datos

La forma de enviar datos a e-callisto es un tema que no es fácil de interpretar. Viendo diferentes logs (registros) se observa que antes de enviar cada muestra se envían unos caracteres especiales que es necesario comprender.
Viendo los registros se observa que antes de cada muestra se envían la siguiente secuencia:

```
'\x00\x01\x02'
```

Enviando esto y después la muestra en formato HEX y con un tamaño de 10 bits se observa que se representa lo que se envía. Si no se envía esa cadena por delante, el programa interpreta los datos de una forma errónea.
Por ejemplificar y probar el funcionamiento he probado a enviar el mismo valor en todas las frecuencias y así probar que el funcionamiento es el correcto.

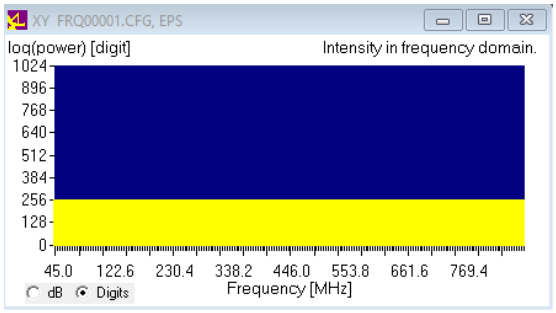


Figura 25: Captura de pantalla de la representación grafica de e-callisto con un valor constante

Si enviamos el mismo valor de forma constante (en este caso el 255) se observa que se envía de forma correcta.
Lo siguiente ya será hacerlo con valores reales, pero eso ya pertenece a la versión final, que es la versión `v0.3.1`.

6.8.2 Código del programa

El código de esta versión es el siguiente:

`v0.2.0.py`

```
import serial
from datetime import datetime
import random

flag = True

def init_serial():
    try:
        ser = serial.Serial(
            port='COM4',
            baudrate=115200,
            parity=serial.PARITY_NONE,
            bytesize=serial.EIGHTBITS,
        )
    except:
        print("Error al abrir el puerto " + ser.port)
        exit()
    ser.isOpen()
```

```
    return ser


def write_serial(ser, data):
    ser.write(data)
    logfile.write("SDR    (" + datetime.now().strftime("%H:%M:%S") + ")>> " + str(data) +
'\n')
    print("SDR    >> " + str(data))

def check_close(string):
    if string == "834813":
        logfile.close()
        return True
    return False


def check_start_previous(string):
    if string == "834913": # Equivale a S1
        return True
    return False


def check_start(string):
    if string == "716913": # Equivale a GE
        return True
    return False


def check_stop(string):
    if string == "716813":
        logfile.close()
        return True
    return False

def starts_FE(string):
    if string[0:4] == "7069":
        return True
    return False



ser = init_serial()
# this will store the line
line = []

logfile = open("log.txt", "w")
samples = []






while True:
    for c in ser.read():
        line.append(c)
        if c == 13:
            string_ints = [chr(int) for int in line]
            string_hex = [str(int) for int in line]
            print("Calisto>> " + ''.join(string_ints))
            print("       >> " + ''.join(string_hex) + '\n')
            logfile.write("Calisto (" + datetime.now().strftime("%H:%M:%S") + ")>> " +
''.join(string_ints))
            logfile.write("                      " + ''.join(string_hex) + '\n')
            if starts_FE(''.join(string_hex)):
```

```
                write_serial(ser, bytes([0x5d]))
                if flag:
                    write_serial(ser, bytes('$Frequency\r', encoding='utf8'))
                    flag = False
            elif check_start_previous(''.join(string_hex)):
                write_serial(ser, bytes('$CRX:Started.2\r', encoding='utf8'))
            elif check_start(''.join(string_hex)):
                while True:
                    var = ''
                    samples.clear()
                    for i in range (0,200):
                        samples.append(255)
                    for i in range (200):
                        var += '\x00\x01\x02' + format(samples[i], 'x')
                    ser.write(var.encode("utf-8"))
            elif check_close(''.join(string_hex)):
                ser.close()
                exit()
            line = []
            break
```

6.8.3 Explicación del código

En este código hay 4 partes bien diferenciadas: importar las bibliotecas, creación del puerto serie, funciones para interpretar los datos y bucle de comunicación. Son las mismas que la versión anterior ya que las diferencias solo aparecen en una de ellas (en el resto también hay, pero son muy pequeñas).

- **Importar las bibliotecas**. Aquí hay que añadir una biblioteca con respecto a la versión anterior:
    - `Random`. Se utiliza para generar números aleatorios. En el código que está unas líneas más arriba no aparece ninguna función donde se utilice, pero para probar el correcto funcionamiento, aparte de enviar datos fijos, también he tenido que hacer pruebas mandando datos aleatorios.

```
import serial
from datetime import datetime
import random
```

- **Creación del puerto serie**. No hay ninguna diferencia con respecto a la versión anterior. Para ver el significado ir a `v0.1.0`.

```
def init_serial():
    try:
        ser = serial.Serial(
            port='COM4',
            baudrate=115200,
            parity=serial.PARITY_NONE,
            bytesize=serial.EIGHTBITS,
        )
    except:
        print("Error al abrir el puerto " + ser.port)
        exit()
    ser.isOpen()
    return ser
```

- Funciones para interpretar los datos. La mayoría de las funciones son las mismas que en la versión anterior. Simplemente se ha añadido una:
    - `starts_fe()`. Esta función sirve para identificar si se está enviando una cadena que empieza por FE. Esto se hace para enviar solo el carácter de asentimiento `0x5D` cuando se recibe una cadena que inicia con FE y no en otro momento.

  El resto de las funciones no paso a explicarlas ya que están explicadas ya en la versión anterior.

```
def write_serial(ser, data):
    ser.write(data)
    logfile.write("SDR    (" + datetime.now().strftime("%H:%M:%S") + ")>> " + str(data) +
'\n')
    print("SDR    >> " + str(data))
```

```
def check_close(string):
    if string == "834813":
        logfile.close()
        return True
    return False


def check_start_previous(string):
    if string == "834913": # Equivale a S1
        return True
    return False


def check_start(string):
    if string == "716913": # Equivale a GE
        return True
    return False


def check_stop(string):
    if string == "716813":
        logfile.close()
        return True
    return False

def starts_FE(string):
    if string[0:4] == "7069":
        return True
    return False
```

- Bucle de comunicación. Aquí se ha añadido lo explicado en el punto anterior. Cuando se detecta el inicio del bucle se añade el carácter `\r` para que se pueda visualizar la gráfica.
  Añadido a eso, aquí se ve que se añade una cadena de texto de 200 muestras donde cada muestra tiene el valor constante de 255 (`samples.append(255)`).
  Luego a eso se le añaden los caracteres que hacen que e-callisto interpretara las muestras correctamente y se envía todo por el puerto serie codificándolo.

```
ser = init_serial()
# this will store the line
line = []

logfile = open("log.txt", "w")
samples = []


while True:
    for c in ser.read():
        line.append(c)
        if c == 13:
            string_ints = [chr(int) for int in line]
            string_hex = [str(int) for int in line]
            print("Calisto>> " + ''.join(string_ints))
            print("       >> " + ''.join(string_hex) + '\n')
            logfile.write("Calisto (" + datetime.now().strftime("%H:%M:%S") + ")>> " +
''.join(string_ints))
            logfile.write("                      " + ''.join(string_hex) + '\n')
            if starts_FE(''.join(string_hex)):
                write_serial(ser, bytes([0x5d]))
                if flag:
                    write_serial(ser, bytes('$Frequency\r', encoding='utf8'))
                    flag = False
            elif check_start_previous(''.join(string_hex)):
                write_serial(ser, bytes('$CRX:Started.2\r', encoding='utf8'))
            elif check_start(''.join(string_hex)):
```

```
            while True:
                var = ''
                samples.clear()
                for i in range (0,200):
                    samples.append(255)
                for i in range (200):
                    var += '\x00\x01\x02' + format(samples[i], 'x')
                ser.write(var.encode("utf-8"))
        elif check_close(''.join(string_hex)):
            ser.close()
            exit()
        line = []
        break
```

6.8.4 Ejecución del código

La captura de la ejecución del código es la que se ve en la explicación del formato de los datos:

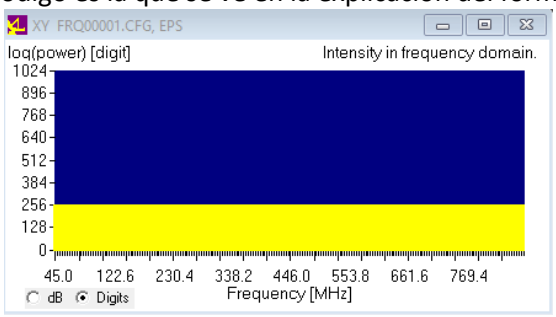


Figura 26: Captura de pantalla de e-callisto con un valor constante de 255

Aparece una recta constante en 255.

6.9 Versión 0.2.1 (v0.2.1): Lectura de datos eficiente

Esta versión ya es la versión casi definitiva del módulo SDR. La única diferencia entre esta versión y la versión final es que esta se ejecuta en la Raspberry Pi y la versión final se ejecuta desde la maquina donde está el programa e-callisto a través de una conexión TCP.
Antes de mostrar el código hay que explicar la lógica que me ha llevado a elegir cierta frecuencia de muestreo.

6.9.1 Lógica lectura de datos

El tiempo de ejecución de la versión `v0.0.3` sobrepasaba el minuto. Esto no es algo factible para el desarrollo del programa, por lo que habría que disminuir el tiempo de ejecución de alguna forma.

1. La primera forma que desarrollé fue iniciando el SDR y finalizándolo fuera del bucle. Es decir, que dentro del bucle solo hubiera que especificar la nueva frecuencia. Haciendo esto se consigue bajar el tiempo de ejecución a unos 30 segundos. Sigue siendo mucho, pero es algo más factible.
2. La segunda opción y que es definitiva es leyendo selectivamente solo las muestras necesarias. Tenemos que leer 200 muestras en el espectro 45-200. La opción que he encontrado más eficiente es realizar 100 mediciones cogiendo la primera muestra y la última.
   Si se ejemplifica se entiende mejor. Cogemos las 4 primeras muestras a leer:

```
[0001]=0045.000,0
[0002]=0049.250,0
[0003]=0053.563,0
[0004]=0057.875,0
```

   Lo que he decidido hacer es coger la frecuencia intermedia de las muestras 1 y 2 y ponerla como frecuencia central. El ancho de banda será la mitad de la distancia entre muestras. Por ello al realizar una medición, la primera y la última muestra son las que queremos. Si realizas esto 100 veces tienes el espectro completo.
   Con este método, el tiempo barrido baja a menos de 5 segundos. Sigue siendo un tiempo alto, pero es algo factible.
   En el caso de las muestras de más arriba, las frecuencias centrales serían:

$$f_1 = 45 + \frac{49{,}250 - 45}{2} = 47{,}125$$

$$f_2 = 53{,}563 + \frac{57{,}875 - 53{,}563}{2} = 55{,}719$$

   Con esas dos frecuencias y cogiendo el inicio y el fin tendríamos las 4 muestras pedidas.
   Para simplificar y realizar un barrido constante (ya que las frecuencias del archivo de e-callisto no tienen unas diferencias entre frecuencias constante) he realizado el barrido con una diferencia entre muestras de:

$$\frac{870 - 45}{200} = 4{,}125 \rightarrow 8{,}25\ entre\ frecuencias$$

   Ahí está la explicación del código de esta versión.

6.9.2 Análisis de tiempo

Esta es la lectura de datos más eficiente de todo el trabajo. Se trata de la lectura más rápida de todas. El tiempo medio que se tarda en realizar el barrido es de:

$$t \approx 5s$$

Con la fórmula de leer 100 veces diferentes espectros se consigue reducir el tiempo de lectura del espectro completo a unos 5s. Se trata de un tiempo bastante aceptable sabiendo que estamos ante un dispositivo de bajo coste. Con este tiempo se pueden empezar a realizar ya mediciones del espectro con e-callisto de forma eficaz.

6.9.3 Código del programa

El código de esta versión es el siguiente:

`V0.2.1.py`

```
from rtlsdr import *
import matplotlib.pyplot as plt
import numpy as np
import time
import math

#optimizar

powe = np.array([])
freqdb = np.array([])
freq = []
savesamples = []

sdr = RtlSdr()
sdr.sample_rate = 2.0625e6
sdr.center_freq = 47.0625e6
sdr.freq_correction = 60   # PPM
sdr.gain = 'auto'  # 4

start = time.time()
for i in np.arange(47.0625, 871, 8.25):

    # configure device
    sdr.center_freq = i*1e6  # Hz
    samples = sdr.read_samples(256)
    savesamples.append(int(np.linalg.norm(samples[0])*1024/3))
    savesamples.append(int(np.linalg.norm(samples[255])*1024/3))
    #sdr = None
    # use matplotlib to estimate and plot the PSD
    power, psd_freq = plt.psd(samples, NFFT=1024, Fs=2.0625e6/1e6)
    psd_freq = psd_freq + i
    powe = np.concatenate((powe, np.array(power)))
    freqdb = np.concatenate((freqdb, np.array(psd_freq)))
    #savefreq.append(i)

print(time.time()-start)
print(savesamples, len(savesamples))
#print(savefreq, len(savefreq))
sdr.close()

for i in np.arange(45,870, 4.125):
    freq.append(i)

print(freq, len(freq))

plt.clf()
plt.figure(dpi=1200)
plt.figure(figsize=(25, 10))
plt.plot(freqdb, powe)
plt.yscale('log')
plt.xlim(45, 870)
plt.xlabel('Frequency (MHz)')
plt.ylabel('Relative power (dB)')
plt.savefig('v0.2.0(dB).png')
print("Figura dB creada")

plt.clf()
plt.figure(dpi=1200)
plt.figure(figsize=(25, 10))
plt.plot(freq, savesamples)
plt.xlim(45, 870)
plt.ylim(0,1024)
plt.xlabel('Frequency (MHz)')
plt.ylabel('Power')
plt.savefig('v0.2.0.png')
print("Figura unidades naturales creada")
```

6.9.4 Explicación del código

En este código hay 3 partes: importar las bibliotecas, lectura de las muestras y representación de las muestras.

- Importar las bibliotecas. Las bibliotecas que se utilizan en esta versión son:
  - `rtlsdr`. Se usa desde la primera versión y se utiliza para la lectura de datos con el SDR.
  - `pyplot`. Igual, esta función se utiliza desde la primera versión y es para representar los datos.
  - `numpy`. En este caso es para el cálculo de datos.
  - `time`. Esta versión es la primera que utiliza esta librería y se utiliza para calcular los tiempos de ejecución de manera precisa.
  - `math`. Por último, esta biblioteca la utilizamos para realizar ciertas operaciones matemáticas que no vienen por defecto.

  Con todo esto ya tenemos todas las bibliotecas que se utilizaran en esta versión.

```
from rtlsdr import *
import matplotlib.pyplot as plt
import numpy as np
import time
import math
```

- Lectura de las muestras. Aquí se produce la lectura de muestras de la manera que se explica en el punto de la lógica de lectura de datos.
  - Se crean varias variables para guardar los datos leídos y se configura el SDR con la frecuencia de muestreo y la frecuencia central inicial. Después de esto iniciamos el contador.
  - En el bucle `for` es donde se produce la lectura de datos. Ya directamente después de leerlas se añaden a la variable `savesamples` las dos muestras necesarias (la primera y la última). A partir de aquí es la lógica para la representación clásica de la medida. Esto es porque quiero representar los espectros de las dos formas (forma clásica y forma eficiente) para comprobar que se está leyendo bien todo.
  - Fuera ya del bucle se imprime el tiempo que tarda y se cierra el SDR.

  Esto es todo lo referente a la lectura de muestras. En el punto de ejecución del código se verán las capturas que diferencian las dos formas de representación del espectro.

```
powe = np.array([])
freqdb = np.array([])
freq = []
savesamples = []

sdr = RtlSdr()
sdr.sample_rate = 2.0625e6
sdr.center_freq = 47.0625e6
sdr.freq_correction = 60   # PPM
sdr.gain = 'auto'  # 4

start = time.time()
for i in np.arange(47.0625, 871, 8.25):

    # configure device
    sdr.center_freq = i*1e6  # Hz
    samples = sdr.read_samples(256)
    savesamples.append(int(np.linalg.norm(samples[0])*1024/3))
    savesamples.append(int(np.linalg.norm(samples[255])*1024/3))
    #sdr = None
    # use matplotlib to estimate and plot the PSD
    power, psd_freq = plt.psd(samples, NFFT=1024, Fs=2.0625e6/1e6)
    psd_freq = psd_freq + i
    powe = np.concatenate((powe, np.array(power)))
    freqdb = np.concatenate((freqdb, np.array(psd_freq)))
    #savefreq.append(i)

print(time.time()-start)
print(savesamples, len(savesamples))
#print(savefreq, len(savefreq))
```

```
sdr.close()
```

- Representación de las muestras. Hay dos representaciones graficas en este script. La primera es la representación clásica y la segunda la representación de solo 200 muestras.
    - La representación clásica es igual a las versiones anteriores. Se cogen las variables de frecuencia y de potencia leída y se representan con el eje y en formato logarítmico.
    - La representación de 200 muestras es algo más compleja. Lo primero es crear la variable de frecuencia, que es el bucle de la primera línea.
      Lo segundo es que el eje y no es logarítmico. Esto hace que los dos espectros que se visualizaran tienen distinta escala. También hay que añadir que el valor de la amplitud está escalado para que sea lo más acorde a la realidad.

  Con todo esto ya tendríamos representados los espectros de las dos formas.

```
for i in np.arange(45,870, 4.125):
    freq.append(i)

print(freq, len(freq))

plt.clf()
plt.figure(dpi=1200)
plt.figure(figsize=(25, 10))
plt.plot(freqdb, powe)
plt.yscale('log')
plt.xlim(45, 870)
plt.xlabel('Frequency (MHz)')
plt.ylabel('Relative power (dB)')
plt.savefig('v0.2.0(dB).png')
print("Figura dB creada")

plt.clf()
plt.figure(dpi=1200)
plt.figure(figsize=(25, 10))
plt.plot(freq, savesamples)
plt.xlim(45, 870)
plt.ylim(0,1024)
plt.xlabel('Frequency (MHz)')
plt.ylabel('Power')
plt.savefig('v0.2.0.png')
print("Figura unidades naturales creada")
```

6.9.5 Ejecución del código

La ejecución del código, como se explica en el punto de explicación del código, genera dos graficas: la gráfica clásica y la gráfica con el método eficiente.

6.9.5.1 Grafica clásica

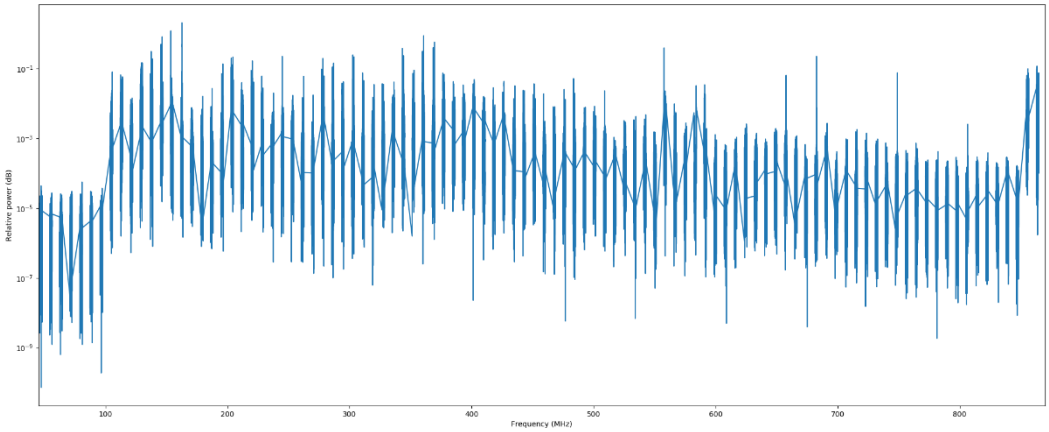


Figura 27: Representación gráfica utilizando el mismo método de dibujo que en versiones anteriores

Se observa en la gráfica que hay un cumulo de muestras y, de golpe, ninguna muestra. Esto es porque se leen 256 muestras cada 4,125 MHz para ser eficiente. Se leen tantas porque es el mínimo para que el SDR no genere problemas.

6.9.5.2 Grafica de 200 muestras

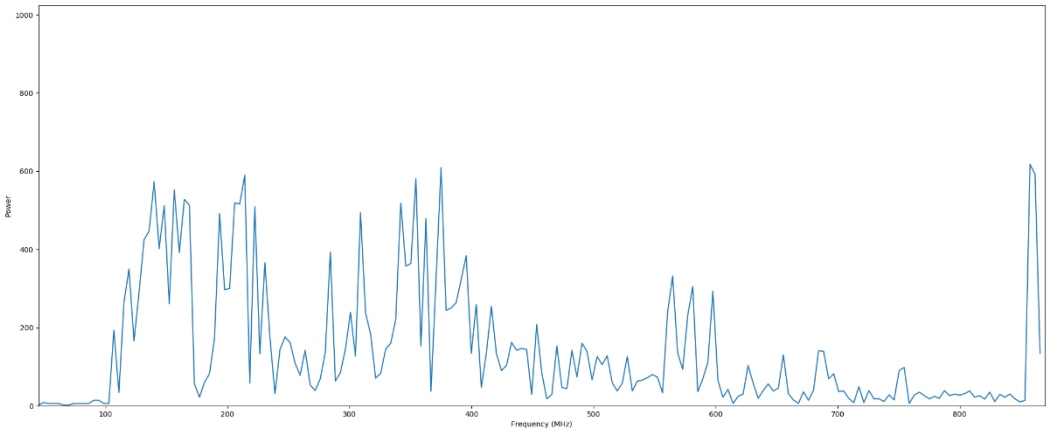


Figura 28: Representación gráfica de las 200 muestras

Aquí ya tendríamos la gráfica con las 200 muestras. Esto es lo que se enviaría a e-callisto. Se ve que las pendientes son mucho mayores, pero es algo más que lógico al tener un eje y con una escala real y no logarítmica como en la primera gráfica.

6.10 Versión 0.3.0 (v0.3.0): Prueba de la lectura a través de TCP

Esta versión es una versión de 'transición'. Y digo esto porque en esta versión simplemente se prueba que se pueden leer datos a través de la conexión TCP. El código es muy simple y no es nada complejo.

6.10.1 Código del programa

En esta versión hay dos archivos que se deben ejecutar simultáneamente. Uno debe hacerlo en la maquina Windows donde se ejecuta e-callisto y otro en la Raspberry Pi donde tenemos el SDR que se encargará de leer los datos.

6.10.1.1 Código Windows

`V0.3.0_win.py`

```
from rtlsdr import *
import numpy as np


sdr = RtlSdrTcpClient(hostname='192.168.1.38', port=12345)

samples_mod = []

sdr.sample_rate = 2.4e6  # Hz
sdr.center_freq = 800e6     # Hz
sdr.freq_correction = 60   # PPM
sdr.gain = 'auto'

freq = np.arange(sdr.center_freq - sdr.sample_rate/2, sdr.center_freq + sdr.sample_rate/2,
sdr.sample_rate/512)

samples = sdr.read_samples(512)
for i in samples:
    samples_mod.append(int(np.linalg.norm(i)*1024))
print(samples_mod)
```

6.10.1.2 Código Raspberry Pi

`V0.3.0_rpi.py`

```
from rtlsdr import *
import socket

server = RtlSdrTcpServer(hostname='192.168.1.38', port=12345)
server.run_forever()
```

6.10.2 Explicación del código

La gran diferencia entre utilizar la conexión TCP y hacerlo desde la propia máquina es que ahora se pueden ejecutar los comandos de medida (`sdr.readsamples()`) desde la maquina donde se encuentra Windows. Para que esto funcione hay que abrir un puerto en la Raspberry Pi (en el caso del código el 12345) y conectarse desde el ordenador con Windows.
El código se explica a continuación:

6.10.2.1 Código Windows

Tiene dos partes diferenciadas: importar las bibliotecas y lectura de muestras.

- Las bibliotecas para importar son:
    - `pyrtlsdr`. Esta biblioteca es para leer los datos del SDR. La instalación en Linux era muy sencilla, solo había que utilizar el código:

```
pip3 install pyrtlsdr
```

Pero en Windows hay que instalar más cosas aparte de esto. Todo esto se explica detallado en el manual del anexo para que el usuario sepa que hacer si el script no funciona correctamente.

  - `numpy`. Esta biblioteca es para operaciones matemáticas. Ya la estaba utilizando en las versiones que se ejecutaban desde la Raspberry Pi.

```
from rtlsdr import *
import numpy as np
```

- Lectura de muestras. En este caso he utilizado una parte del código de la versión `v0.2.1` para probar que la lectura se hacía de forma correcta. La explicación de cada cosa del siguiente código aparece ya explicada en la versión `v0.2.1`.

```
sdr = RtlSdrTcpClient(hostname='192.168.1.38', port=12345)


samples_mod = []

sdr.sample_rate = 2.4e6  # Hz
sdr.center_freq = 800e6     # Hz
sdr.freq_correction = 60   # PPM
sdr.gain = 'auto'

freq = np.arange(sdr.center_freq - sdr.sample_rate/2, sdr.center_freq + sdr.sample_rate/2,
sdr.sample_rate/512)

samples = sdr.read_samples(512)
for i in samples:
    samples_mod.append(int(np.linalg.norm(i)*1024))
print(samples_mod)
```

6.10.2.2 Código Raspberry Pi

Este código también tiene dos partes diferenciadas, que son la importación de las bibliotecas y la apertura del puerto de conexión.

- Importar las bibliotecas. Hay dos bibliotecas:
  - `pyrtlsdr`. Biblioteca para leer los datos del SDR.
  - `socket`. Esta biblioteca es para poder abrir un socket de conexión TCP.

```
from rtlsdr import *
import socket
```

- Apertura puerto de conexión. Simplemente son dos instrucciones:
  - La primera de ellas define la IP de la Raspberry Pi y el puerto que desea abrir. Estos datos deben ser iguales en el script de Windows.
  - La segunda simplemente indica que el servidor va a estar ejecutándose infinitamente hasta que se pulse el comando `Ctrl + C`.

```
server = RtlSdrTcpServer(hostname='192.168.1.38', port=12345)
server.run_forever()
```

6.11 Versión 0.3.1 (v0.3.1): versión final con dos dispositivos

Después de 9 versiones, llegamos a la versión final. Se trata de la primera versión capaz de leer los datos del SDR y mostrarlos en e-callisto.

6.11.1 Código del programa

Como en la versión anterior, hay dos códigos: el que se ejecuta en el ordenador Windows y el que se ejecuta en la Raspberry Pi.

6.11.1.1 Código Windows

`v0.3.1_win.py`

```
# write data to serial port
import serial
from datetime import datetime
from rtlsdr import *
import numpy as np
import threading
import sys

# Calculando el baudrate
# Se envian 65535 bytes en 2.05s -> 524280/2.05 = 255746.34 -> 256000 bits/s
if len(sys.argv) < 3:
    print("Escriba la IP y el puerto después del script. Por ejemplo:\n
Python3 v0.3.1_win.py 192.168.1.32 12345")
    exit()


flag = True
var = ''
flagstop = False

sdr = RtlSdrTcpClient(hostname='192.168.1.23', port=12344)

samples_mod = []

sdr.sample_rate = 2.0625e6
sdr.center_freq = 47.0625e6
sdr.freq_correction = 60   # PPM
sdr.gain = 'auto'  # 4



def init_serial():
    try:
        ser = serial.Serial(
            port='COM4',
            baudrate=115200,
            parity=serial.PARITY_NONE,
            bytesize=serial.EIGHTBITS,
        )
    except:
        print("Error al abrir el puerto " + ser.port)
        exit()
    ser.isOpen()
    return ser



def write_serial(ser, data):
    ser.write(data)
    logfile.write("SDR    (" + datetime.now().strftime("%H:%M:%S") + ")>> " + str(data) +
'\n')
    print("SDR    >> " + str(data))
```

```
def check_close(string):
    if string == "834813":
        logfile.close()
        return True
    return False


def check_start_previous(string):
    if string == "834913": # Equivale a S1
        return True
    return False


def check_start(string):
    if string == "716913": # Equivale a GE
        return True
    return False


def check_stop(string):
    if string == "716813":
        return True
    return False

def starts_FE(string):
    if string[0:4] == "7069":
        return True
    return False


def send_samples():
    global var
    while True:
        ser.write(var.encode("utf-8"))
        if flagstop:
            print("flagstop send_samples")
            break

def read_samples():
    global var
    while True:
        samples_mod.clear()
        for i in np.arange(47.0625, 871, 8.25):
            # configure device
            sdr.center_freq = i*1e6  # Hz
            samples = sdr.read_samples(256)
            samples_mod.append(int(np.linalg.norm(samples[0])*1024/2))
            samples_mod.append(int(np.linalg.norm(samples[255])*1024/2))
        var = ''
        for i in range (200):
            if (samples_mod[i] > 255):
                var += "\x00\x01\x02" + format(samples_mod[i], 'x' )
            elif (samples_mod[i] > 15):
                var += "\x00\x01\x020" + format(samples_mod[i], 'x' )
            else:
                var += "\x00\x01\x0200" + format(samples_mod[i], 'x' )
        if flagstop:
            print("flagstop read_samples")
            break

ser = init_serial()
# this will store the line
line = []
```

```
logfile = open("log.txt", "w")
samples = []



while True:
    for c in ser.read():
        line.append(c)
        if c == 13:
            string_ints = [chr(int) for int in line]
            string_hex = [str(int) for int in line]
            print("Calisto>> " + ''.join(string_ints))
            print("       >> " + ''.join(string_hex) + '\n')
            logfile.write("Calisto (" + datetime.now().strftime("%H:%M:%S") + ")>> " +
''.join(string_ints))
            logfile.write("                      " + ''.join(string_hex) + '\n')
            if starts_FE(''.join(string_hex)):
                write_serial(ser, bytes([0x5d]))
                if flag:
                    write_serial(ser, bytes('$Frequency\r', encoding='utf8'))
                    flag = False
            elif check_start_previous(''.join(string_hex)):
                write_serial(ser, bytes('$CRX:Started.2\r', encoding='utf8'))
            elif check_start(''.join(string_hex)):
                flagstop = False
                flagS0 = True
                var = ''
                samples.clear()
                for i in range(0,200):
                    samples_mod.append(255)
                for i in range (200):
                    var += "\x00\x01\x02" + format(samples_mod[i], 'x')
                ser.write(var.encode('utf-8'))
                sdr_thread = threading.Thread(target=send_samples)
                sdr_readsamples = threading.Thread(target=read_samples)
                sdr_thread.start()
                sdr_readsamples.start()
            elif check_stop(''.join(string_hex)):
                flagstop = True
            elif check_close(''.join(string_hex)):
                if flagS0:
                    flagS0 = False
                    break
                ser.close()
                logfile.close()
                exit()
            line = []
            break
```

6.11.1.2 Código Raspberry Pi

`v0.3.1_rpi.py`

```
from rtlsdr import *
import socket
import sys

def get_ip():
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.settimeout(0)
    try:
        # doesn't even have to be reachable
        s.connect(('10.254.254.254', 1))
        IP = s.getsockname()[0]
    except Exception:
        IP = '127.0.0.1'
```

```
    finally:
        s.close()
    return IP

print("La IP de la Raspberry Pi es: " + get_ip())
if len(sys.argv) < 2:
    print("Escriba el puerto despues del script. Por ejemplo:\nPython3 v0.3.1_rpi.py
12345")
    exit()
print("El puerto de conexión es el: " + sys.argv[1])

server = RtlSdrTcpServer(hostname=get_ip(), port=int(sys.argv[1]))
server.run_forever()
```

6.11.2 Explicación del código

Al igual que en la versión anterior, hay dos códigos que explicar.

6.11.2.1 Código Windows

El código de Windows se divide en 5 partes: importar las bibliotecas, configuración del puerto, funciones de interpretación de datos, lectura del puerto serie y lectura de muestras.

- **Importar las bibliotecas**. Se utilizan 5 bibliotecas:
    - `pyserial`. Se utiliza para leer el puerto serie
    - `datetime`. Para dar formato a las fechas.
    - `pyrtlsdr`. Para la lectura de datos en el SDR.
    - `numpy`. Se utiliza para operaciones de datos leídos del SDR.
    - `threading`. Es una biblioteca nueva y se utiliza para poder ejecutar en paralelo la lectura de datos en el SDR y el envío de los mismos a e-callisto.
    - `sys`. Segunda nueva biblioteca que se utiliza. En este caso es para poder leer los argumentos que se escriben al ejecutar el programa.

```
import serial
from datetime import datetime
from rtlsdr import *
import numpy as np
import threading
import sys
```

- **Configuración del puerto**. Es exactamente la misma configuración que en la versión anterior `v0.2`.

```
def init_serial():
    try:
        ser = serial.Serial(
            port='COM4',
            baudrate=115200,
            parity=serial.PARITY_NONE,
            bytesize=serial.EIGHTBITS,
        )
    except:
        print("Error al abrir el puerto " + ser.port)
        exit()
    ser.isOpen()
    return ser
```

- **Funciones de interpretación de datos**. Aquí la mayoría de las funciones son iguales a la versión `v0.2`. Paso a explicar las nuevas. Para saber el funcionamiento del resto remitirse a la `v0.2`.
    - `check_stop()`. En esta función solo se ha borrado una instrucción que hacia cerrar el log. Esto creaba un error al parar de leer las muestras.
    - `send_samples()`. Esta función manda datos a través del puerto serie. Lo hace infinitamente hasta que un flag cambie de estado. Esto lo hará al nosotros parar la lectura de datos.

```
def write_serial(ser, data):
    ser.write(data)
    logfile.write("SDR    (" + datetime.now().strftime("%H:%M:%S") + ")>> " + str(data) +
'\n')
```

```
    print("SDR    >> " + str(data))

def check_close(string):
    if string == "834813":
        logfile.close()
        return True
    return False


def check_start_previous(string):
    if string == "834913": # Equivale a S1
        return True
    return False


def check_start(string):
    if string == "716913": # Equivale a GE
        return True
    return False


def check_stop(string):
    if string == "716813":
        return True
    return False

def starts_FE(string):
    if string[0:4] == "7069":
        return True
    return False

def send_samples():
    global var
    while True:
        ser.write(var.encode("utf-8"))
        if flagstop:
            print("flagstop send_samples")
            break
```

- Lectura del puerto serie. Casi todo es igual a la versión anterior a excepción de cuando se pulsa el botón de leer muestras, cuando se pulsa el de parar y cuando se cierra el programa:
    - Botón de leer muestras pulsado. Aquí se actualiza el flag de salida de los bucles de envío y lectura de datos y se crea una línea continua como primeros datos que se envían a e-callisto. Una vez se ha hecho esto, crean dos hilos para ejecutarlos en segundo plano mientras se siguen leyendo datos que envíe e-callisto.
    - Botón de parar de leer muestras pulsado. Al parar de leer muestras se actualiza el flag y se espera a que terminen los dos hilos iniciados en el botón de empezar a leer muestras.
    - Botón de cerrar e-callisto. Aquí hay que filtrar si se trata del botón de cerrar o no. Esto es porque al darle a parar de leer se envía el mismo comando que al cerrar. Filtrando por el flag de antes es como se comprueba que se viene o no del botón de parar de leer muestras.

```
ser = init_serial()
# this will store the line
line = []

logfile = open("log.txt", "w")
samples = []

while True:
    for c in ser.read():
        line.append(c)
        if c == 13:
            string_ints = [chr(int) for int in line]
            string_hex = [str(int) for int in line]
```

```
            print("Calisto>> " + ''.join(string_ints))
            print("       >> " + ''.join(string_hex) + '\n')
            logfile.write("Calisto (" + datetime.now().strftime("%H:%M:%S") + ")>> " +
''.join(string_ints))
            logfile.write("                     " + ''.join(string_hex) + '\n')
            if starts_FE(''.join(string_hex)):
                write_serial(ser, bytes([0x5d]))
                if flag:
                    write_serial(ser, bytes('$Frequency\r', encoding='utf8'))
                    flag = False
            elif check_start_previous(''.join(string_hex)):
                write_serial(ser, bytes('$CRX:Started.2\r', encoding='utf8'))
            elif check_start(''.join(string_hex)):
                flagstop = False
                flagS0 = True
                var = ''
                samples.clear()
                for i in range(0,200):
                    samples_mod.append(255)
                for i in range (200):
                    var += "\x00\x01\x02" + format(samples_mod[i], 'x')
                ser.write(var.encode('utf-8'))
                sdr_thread = threading.Thread(target=send_samples)
                sdr_readsamples = threading.Thread(target=read_samples)
                sdr_thread.start()
                sdr_readsamples.start()
            elif check_stop(''.join(string_hex)):
                flagstop = True
            elif check_close(''.join(string_hex)):
                if flagS0:
                    flagS0 = False
                    break
                ser.close()
                logfile.close()
                exit()
            line = []
            break
```

- Lectura de muestras. La lectura de muestras es prácticamente coger el código de la versión `v0.2.1` que se ejecutaba en la Raspberry Pi y pasar a ejecutarlo en el ordenador donde se ejecuta e-callisto. Tiene dos partes:
    - Configurar el SDR. Esto incluye crear la conexión TCP y definir la frecuencia de muestreo, frecuencia central, etc.
    - Lectura de datos. Esto se hace en la función `read_samples()`. Hace el bucle de lectura al igual que en la versión `v0.2.1` y a eso le añade la conversión a una cadena de texto que e-callisto sepa interpretar.
      Al final del bucle hay un flag para poder parar la lectura de datos cuando se solicite.

```
sdr = RtlSdrTcpClient(hostname='192.168.1.23', port=12345)

samples_mod = []

sdr.sample_rate = 2.0625e6
sdr.center_freq = 47.0625e6
sdr.freq_correction = 60   # PPM
sdr.gain = 'auto'  # 4


def read_samples():
    global var
    while True:
        samples_mod.clear()
        for i in np.arange(47.0625, 871, 8.25):
            # configure device
            sdr.center_freq = i*1e6  # Hz
```

```
            samples = sdr.read_samples(256)
            samples_mod.append(int(np.linalg.norm(samples[0])*1024/2))
            samples_mod.append(int(np.linalg.norm(samples[255])*1024/2))
        var = ''
        for i in range (200):
            if (samples_mod[i] > 255):
                var += "\x00\x01\x02" + format(samples_mod[i], 'x' )
            elif (samples_mod[i] > 15):
                var += "\x00\x01\x020" + format(samples_mod[i], 'x' )
            else:
                var += "\x00\x01\x0200" + format(samples_mod[i], 'x' )
        if flagstop:
            print("flagstop read_samples")
            break
```

6.11.2.2 Código Raspberry Pi

El código de la Raspberry Pi tiene 3 partes: importación de las bibliotecas, función de lectura de IP y la apertura del puerto de conexión.

- Importar las bibliotecas. Hay dos bibliotecas:
  - `pyrtlsdr`. Biblioteca para leer los datos del SDR.
  - `socket`. Esta biblioteca es para poder abrir un socket de conexión TCP.

```
from rtlsdr import *
import socket
import sys
```

- Función de lectura de IP. Se trata de una función que lee la IP del dispositivo para poder configurar de manera automáticamente la apertura del puerto. Para ejecutar este script es necesario pasar por argumento el puerto.

```
def get_ip():
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.settimeout(0)
    try:
        # doesn't even have to be reachable
        s.connect(('10.254.254.254', 1))
        IP = s.getsockname()[0]
    except Exception:
        IP = '127.0.0.1'
    finally:
        s.close()
    return IP

print("La IP de la Raspberry Pi es: " + get_ip())
if len(sys.argv) < 2:
    print("Escriba el puerto despues del script. Por ejemplo:\nPython3 v0.3.1_rpi.py
12345")
    exit()
print("El puerto de conexión es el: " + sys.argv[1])
```

- **Apertura puerto de conexión**. Simplemente son dos instrucciones:
  - La primera de ellas define la IP de la Raspberry Pi y el puerto que desea abrir. Estos datos deben ser iguales en el script de Windows.
  - La segunda simplemente indica que el servidor va a estar ejecutándose infinitamente hasta que se pulse el comando `Ctrl + C`.

```
server = RtlSdrTcpServer(hostname=get_ip(), port=int(sys.argv[1]))
server.run_forever()
```

6.11.3 Ejecución del Código

Antes de exponer las capturas hay que explicar cómo se ejecutan ambos scripts ya que hay que pasar por argumento ciertos datos.

- Para ejecutar el script de la Raspberry hay que pasar por argumento el puerto de la siguiente forma:

```
python3 v0.3.1_rpi.py 12345
```

Donde el 12345 es el puerto por el que se establecerá la conexión.

- Por otro lado, para ejecutar el script de Windows hay que escribir lo siguiente:

```
python3 v0.3.1_win.py 192.168.1.32 12345
```

En este caso, se pasa primero la IP de la Raspberry Pi a la que se tiene que conectar y seguidamente el puerto. Es importante que el puerto sea el mismo o no se podrá realizar la conexión.

A continuación, pongo dos capturas del código ejecutándose:

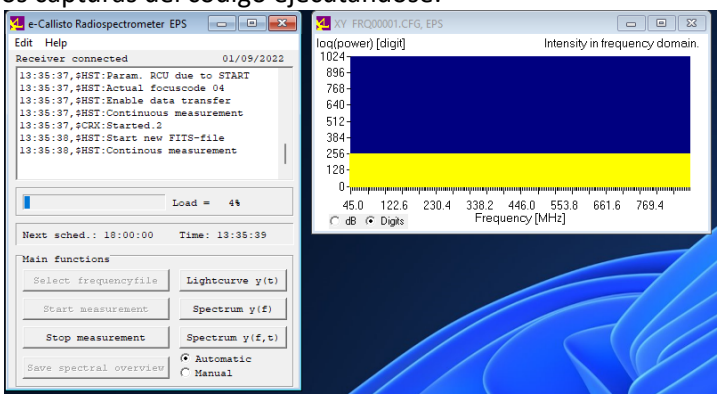


Figura 29: Captura mientras se está leyendo por primera vez el espectro

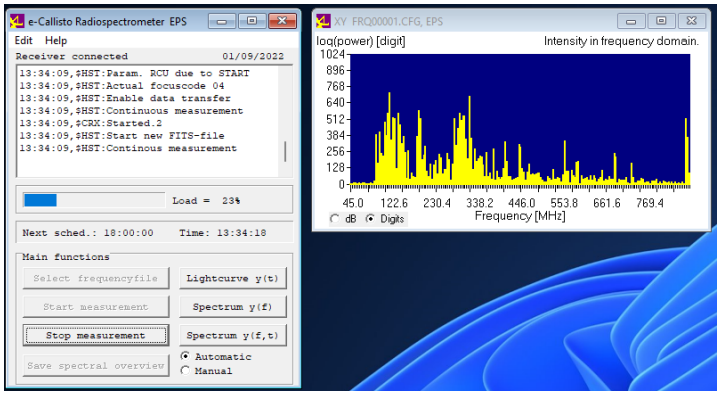


Figura 30: Captura del espectro leído y representado

6.12 Versión 0.3.2 (v0.3.2): versión final con un único dispositivo

Se trata de una versión similar a la anterior, pero en este caso solo se utiliza un dispositivo. Para ejecutar esta versión se necesita un ordenador Windows y el SDR conectado a este ordenador. No se necesita la Raspberry Pi para esta versión.

6.12.1 Código del programa

Aquí hay únicamente un script con el código que se ejecutará en un ordenador basado en Windows y con el e-callisto instalado.

v0.3.2.py

```
# write data to serial port
import serial
from datetime import datetime
from rtlsdr import *
import numpy as np
import threading
import sys

# Calculando el baudrate
# Se envian 65535 bytes en 2.05s -> 524280/2.05 = 255746.34 -> 256000 bits/s

flag = True
var = ''
flagstop = False

sdr = RtlSdr()

samples_mod = []

sdr.sample_rate = 2.0625e6
sdr.center_freq = 47.0625e6
sdr.freq_correction = 60   # PPM
sdr.gain = 'auto'  # 4



def init_serial():
    try:
        ser = serial.Serial(
            port='COM4',
            baudrate=115200,
            parity=serial.PARITY_NONE,
            bytesize=serial.EIGHTBITS,
        )
    except:
        print("Error al abrir el puerto " + ser.port)
        exit()
    ser.isOpen()
    return ser


def write_serial(ser, data):
    ser.write(data)
    logfile.write("SDR    (" + datetime.now().strftime("%H:%M:%S") + ")>> " + str(data) +
'\n')
    print("SDR    >> " + str(data))

def check_close(string):
    if string == "834813":
        logfile.close()
        return True
```

```
    return False


def check_start_previous(string):
    if string == "834913": # Equivale a S1
        return True
    return False


def check_start(string):
    if string == "716913": # Equivale a GE
        return True
    return False


def check_stop(string):
    if string == "716813":
        return True
    return False

def starts_FE(string):
    if string[0:4] == "7069":
        return True
    return False


def send_samples():
    global var
    while True:
        ser.write(var.encode("utf-8"))
        if flagstop:
            print("flagstop send_samples")
            break

def read_samples():
    global var
    while True:
        samples_mod.clear()
        for i in np.arange(47.0625, 871, 8.25):
            # configure device
            sdr.center_freq = i*1e6  # Hz
            samples = sdr.read_samples(256)
            samples_mod.append(int(np.linalg.norm(samples[0])*1024/2))
            samples_mod.append(int(np.linalg.norm(samples[255])*1024/2))
        var = ''
        for i in range (200):
            if (samples_mod[i] > 255):
                var += "\x00\x01\x02" + format(samples_mod[i], 'x' )
            elif (samples_mod[i] > 15):
                var += "\x00\x01\x020" + format(samples_mod[i], 'x' )
            else:
                var += "\x00\x01\x0200" + format(samples_mod[i], 'x' )
        if flagstop:
            print("flagstop read_samples")
            break

ser = init_serial()
# this will store the line
line = []

logfile = open("log.txt", "w")
samples = []
```

```
while True:
    for c in ser.read():
        line.append(c)
        if c == 13:
            string_ints = [chr(int) for int in line]
            string_hex = [str(int) for int in line]
            print("Calisto>> " + ''.join(string_ints))
            print("       >> " + ''.join(string_hex) + '\n')
            logfile.write("Calisto (" + datetime.now().strftime("%H:%M:%S") + ")>> " +
''.join(string_ints))
            logfile.write("                      " + ''.join(string_hex) + '\n')
            if starts_FE(''.join(string_hex)):
                write_serial(ser, bytes([0x5d]))
                if flag:
                    write_serial(ser, bytes('$Frequency\r', encoding='utf8'))
                    flag = False
            elif check_start_previous(''.join(string_hex)):
                write_serial(ser, bytes('$CRX:Started.2\r', encoding='utf8'))
            elif check_start(''.join(string_hex)):
                flagstop = False
                flagS0 = True
                var = ''
                samples.clear()
                for i in range(0,200):
                    samples_mod.append(255)
                for i in range (200):
                    var += "\x00\x01\x02" + format(samples_mod[i], 'x')
                ser.write(var.encode('utf-8'))
                sdr_thread = threading.Thread(target=send_samples)
                sdr_readsamples = threading.Thread(target=read_samples)
                sdr_thread.start()
                sdr_readsamples.start()
            elif check_stop(''.join(string_hex)):
                flagstop = True
            elif check_close(''.join(string_hex)):
                if flagS0:
                    flagS0 = False
                    break
                ser.close()
                logfile.close()
                exit()
            line = []
            break
```

6.12.2 Explicación del código

En esta versión solo hay una diferencia mínima con la versión `v0.3.1`. Es por ello por lo que pasaré a explicar esa mínima diferencia.
Si se desea saber que hace el resto del código, consulte la versión anterior.

- La diferencia es la siguiente línea:

```
sdr = RtlSdr()
```

  Se trata de cambiar la función del SDR de una conexión TCP a conectarse directamente al dongle.
- Hay otra diferencia y es que no es necesario pasar por argumento ningún dato ya que no hay ningún tipo de conexión con ningún dispositivo.

6.12.3 Ejecución del código

Para ejecutar este código no es necesario pasar ningún valor por referencia como si ocurría en la versión anterior.
A continuación, muestro dos capturas del código ejecutándose:

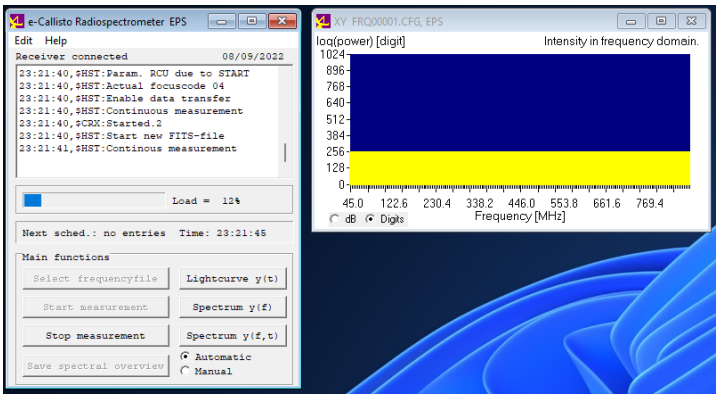


Figura 31: Captura mientras se está leyendo por primera vez el espectro

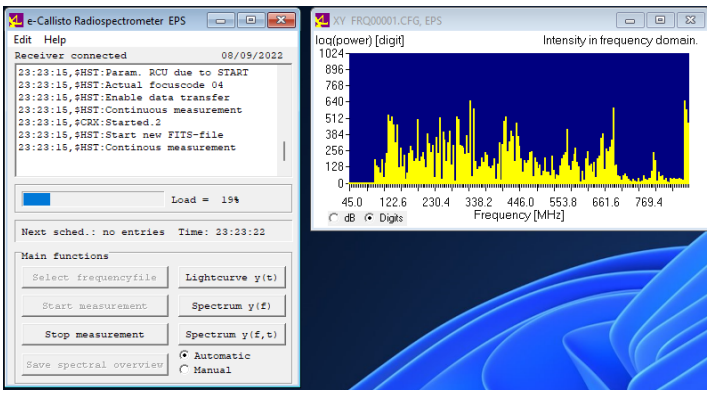


Figura 32: Captura del espectro leído y representado

7 Conclusiones y trabajo futuro

Como conclusión a este trabajo podemos decir que es una muy buena primera aproximación para empezar a utilizar el SDR junto con el programa e-callisto. Se trata de un trabajo de bajo coste con dispositivos muy baratos pero que pueden servir como inicio para futuras versiones tener un mejor dispositivo con una mejor lectura de datos y con un mejor entendimiento del programa.
La idea de este trabajo era tener un punto de partida con un SDR low cost para así poder mejorar esta versión de este en futuros trabajos de fin de grado o de fin de máster.

8 Bibliografía

- Aschwanden, M. (01 de 08 de 2004). Physics of the Solar Corona. An Introduction. *Physics of the Solar Corona*.
- *Comparación SDRs*. (s.f.). Obtenido de https://www.hackers-arise.com/post/software-defined-radio-sdr-for-hackers-hardware-comparison-for-sdr.
- Dillinger, M., Madani, K., & Alonistoti, N. (2003). *Software Defined Radio: Architectures, Systems and Functions.* Wiley & Sons.
- *Documentación e-callisto*. (s.f.). Obtenido de http://www.e-callisto.org.
- *Documentación Math*. (s.f.). Obtenido de https://docs.python.org/3/library/math.html.
- *Documentación matplotlib*. (s.f.). Obtenido de https://claudiovz.github.io/scipy-lecture-notes-ES/intro/matplotlib/matplotlib.html.
- *Documentación Numpy*. (s.f.).Obtenido de https://numpy.org/.
- *Documentación Pyrtlsdr*. (s.f.).Obtenido de https://pyrtlsdr.readthedocs.io/en/latest/.
- *Documentación Pyserial*. (s.f.).Obtenido de https://pyserial.readthedocs.io/en/latest/pyserial_api.html.
- *Documentación Python*. (s.f.).Obtenido de https://www.python.org/doc/.
- *Documentación Random*. (s.f.).Obtenido de https://docs.python.org/3/library/random.html.
- *Documentación Sys*. (s.f.).Obtenido de https://docs.python.org/3/library/sys.html.
- *Documentación Time*. (s.f.).Obtenido de https://docs.python.org/3/library/time.html.
- Kundu, M., White, S., & Jackson, P. (01 de 01 de 1986). Microwave observations of red dwarf flare stars. *Advances in Space Research*.
- McLean, D. (1985). *Metrewave solar radio burst.* United Kingdom: Cambridge University Press.
- *Página de descarga COM Port Data Emulator*. (s.f.).Obtenido de https://www.aggsoft.com/com-port-emulator.htm.
- *Página de descarga com0com*. (s.f.).Obtenido de http://com0com.sourceforge.net/.
- *Página de descarga Visual Studio Code*. (s.f.).Obtenido de https://code.visualstudio.com/.
- Ratcliffe, H., Kontar, E. P., & Reid, H. (2014). Large-scale simulations of solar type III radio bursts: flux density, drift rate, duration, and bandwidth. *A&A*, pág. 572.
- Wild, J., & McCready, L. (3 de 3 de 1950). bservatioas of the Spectrum of High-Intensity Solar Radiation at Metre Wavelengths. I. The Apparatus and Spectral Types of Solar Burst Observed. *Australian Journal of Chemistry*, págs. 387-398.

9 Anexos/apéndices:

9.1 Presupuesto

Para el presupuesto del trabajo hay que contar dos puntos: el coste del material y el coste de la mano de obra.

9.1.1.1 Material

El material utilizado para este trabajo es:

- Raspberry Pi 3 Model B. Miniordenador basado en ARM con un coste aproximado de 60€.
- RTL-SDR. Dongle USB que se puede encontrar por unos 55€.

La antena la se tenia antes de comenzar el trabajo, por lo que no se cuenta dentro del presupuesto.

9.1.1.2 Mano de obra

En este caso se trata de un precio estándar por todas las horas dedicadas al desarrollo del programa. Al no haber distintas tareas y ser todo simplemente desarrollo de código, el precio estándar será el mismo.

Desglose de mano de obra

Tarea	Horas	Precio/hora	Precio total
Lectura documentación	10	30 €	300 €
Desarrollo código	50	30 €	1500€
Testing de la aplicación	10	30 €	300€
Precio total	70		2100 €

9.1.2 Precio total del trabajo

Con esto, el precio total del trabajo es de:

Desglose	Precio
Material • **Raspberry Pi 4** • **SDR**	 60€ 55€
Mano de obra total	2100 €
Total	2215 €

9.2 Manual de usuario

A continuación, paso a explicar cómo instalar la última versión en cualquier equipo Windows para que funcione correctamente.

9.2.1 Versión Raspberry Pi y Windows

Para esta versión se necesita tener una Raspberry Pi y un ordenador Windows. Si lo que desea es ejecutar todo únicamente en Windows, esta no es la versión.

9.2.1.1 Puntos previos

IMPORTANTE: Para que el programa funcione correctamente es necesario ejecutar los dos scripts. Primero lanzar la ejecución del script de la Raspberry Pi y luego hacer lo propio con el script de Windows. **Si uno de los dos no se ejecuta el programa dará un error y no funcionará.**

1. Lo primero es tener instalado Python en el ordenador. El punto `Instalación de Python` explica paso a paso como hacerlo.
2. Lo segundo es tener instalado e-callisto en el ordenador. También hay un punto en el trabajo que explica cómo se hace. El punto `Instalación del programa e-callisto`.
3. También es necesario tener instalado el com0com. Al igual que los dos puntos anteriores, también hay un punto que lo explica. Se trata del punto `Simular puertos Serie`.
4. Es también necesario abrir el script de Windows y cambiar el puerto serie por el que se haya configurado en com0com.
5. Es necesario saber la IP de la Raspberry Pi. Esto se puede obtener de varias formas. Para comodidad del lector, al ejecutar el script en la Raspberry Pi aparecerá la IP de la máquina. Nos la apuntamos porque habrá que añadirla en el script de Windows.

9.2.1.2 Instalación de bibliotecas

Con esto, ya podemos instalar las bibliotecas necesarias para el funcionamiento correcto del programa.

1. Para instalar las bibliotecas en Python hay un punto que lo explica (`Instalar bibliotecas`). Paso a enumerar todas las bibliotecas que hay que instalar:
    a. `Pyrtlsdr`
    b. `Numpy`
    c. `Pyserial`

9.2.1.3 Problema instalando `pyrtlsdr` en Windows

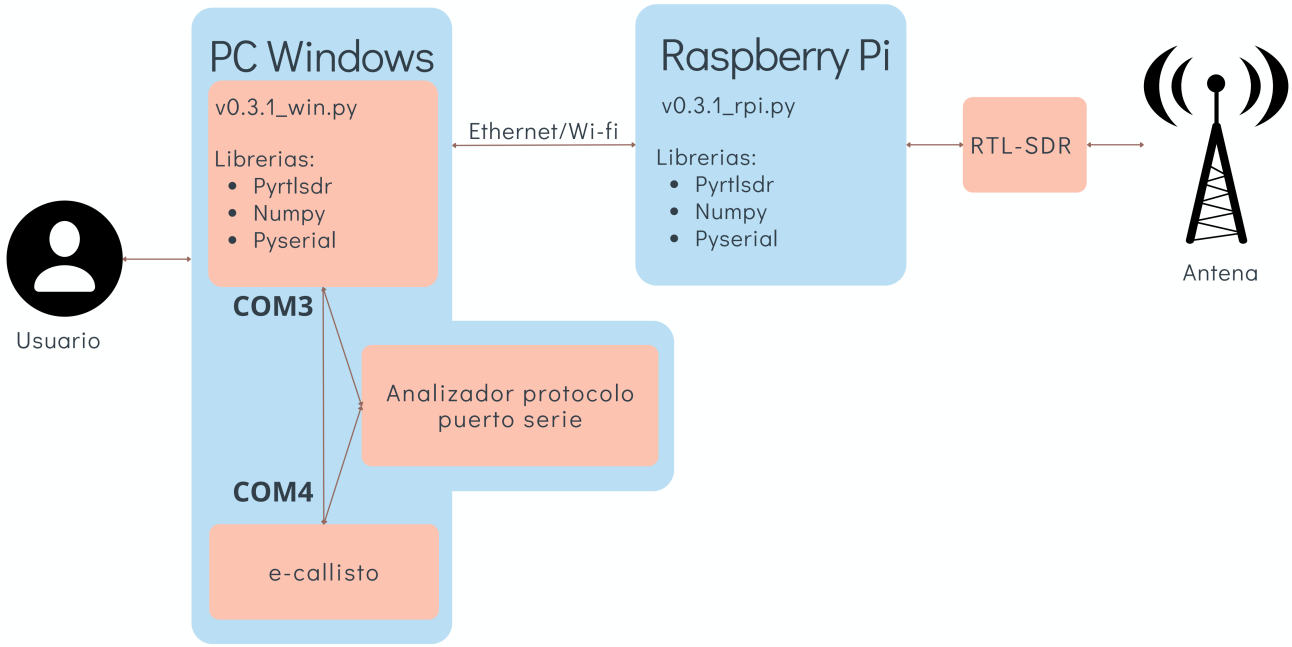
En Windows no sirve solo con utilizar el comando `pip3 install pyrtlsdr`, es necesario instalar ciertas bibliotecas. Hay que copiar los 3 archivos que dejo en la carpeta `v0.3` dentro de la carpeta `dll`. Esa carpeta `dll` yo la tengo copiada en el directorio de `CALLISTO` (en C:).

El punto importante aquí es que hay que editar el archivo librtlsdr.py. Si lo ejecutas y te sale un error, lo mas probable es que sea de este archivo. Vamos a él (depende de cada PC la ruta donde se encuentre) y comentamos las líneas 29-33 y añadimos nosotros las nuestras que quede tal que así:

```
# driver_files += ['librtlsdr.so', 'rtlsdr/librtlsdr.so']
# driver_files += ['rtlsdr.dll', 'librtlsdr.so']
# driver_files += ['..//rtlsdr.dll', '..//librtlsdr.so']
# driver_files += ['rtlsdr//rtlsdr.dll', 'rtlsdr//librtlsdr.so']
# driver_files += [lambda : find_library('rtlsdr'), lambda :find_library('librtlsdr')]
driver_files += ['C:\\CALLISTO-01\\dll\\librtlsdr.dll']
driver_files += ['C:\\CALLISTO-01\\dll\\libusb-1.0.dll']
driver_files += ['C:\\CALLISTO-01\\dll\\libwinpthread-1.dll']
```

Lo que escribimos en las 3 ultimas rutas son donde hemos puesto los 3 archivos de la carpeta dll. Con esto ya debería funcionar correctamente.

9.2.1.4 Diagrama de bloques del modelo



9.2.1.5 Funcionamiento del programa

Para hacer funcionar el programa hay que realizar 3 pasos:

1. Abrir una conexión `ssh` con la Raspberry Pi, irse a la carpeta donde se encuentre el script y ejecutarlo añadiéndole el puerto de la siguiente forma:

```
Python3 v0.3.1_rpi.py 12345
```

   Debería salir algo como lo de a continuación:

```
pi@raspberrypi:~/TFG/v0/v0.3 $ python3 v0.3.1_rpi.py 12345
La IP de la Raspberry Pi es: 192.168.1.32
El puerto de conexión es el: 12345
Detached kernel driver
Found Rafael Micro R820T tuner
[R82XX] PLL not locked!
```

2. Lo siguiente es abrir una terminal en Windows, irse a la carpeta del script y hacer algo parecido a lo que hacíamos en el script de la Raspberry. En este caso hay que añadir la IP y el puerto. La IP debe ser la de la Raspberry, mientras que el puerto debe ser el mismo que hemos añadido al ejecutar el script de la Raspberry.

```
Python3 v0.3.1_win.py 192.168.1.32 12345
```

   Debería salir algo como lo de a continuación.

```
PS C:\Users\famil\Dropbox\Universidad\tfg\v0\v0.3\v0.3.1> python3 v0.3.1_win.py
192.168.1.32 12345
Closing socket connection
Closing socket connection
Closing socket connection
Closing socket connection
```

3. Lo último ya es abrir e-callisto y utilizar el programa como siempre.

9.2.2 Versión Windows

9.2.2.1 Puntos previos

1. Lo primero es tener instalado Python en el ordenador. El punto `Instalación de Python` explica paso a paso como hacerlo.
2. Lo segundo es tener instalado e-callisto en el ordenador. También hay un punto en el trabajo que explica cómo se hace. El punto `Instalación del programa e-callisto`.
3. También es necesario tener instalado el com0com. Al igual que los dos puntos anteriores, también hay un punto que lo explica. Se trata del punto `Simular puertos Serie`.

4. Es también necesario abrir el script de Windows y cambiar el puerto serie por el que se haya configurado en com0com.

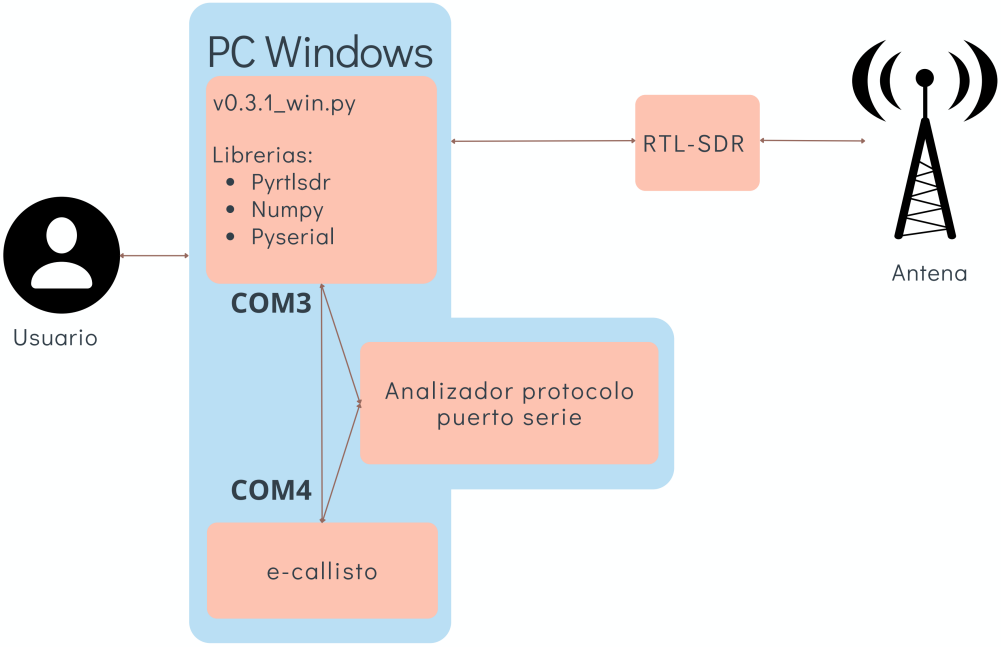
9.2.2.2 Instalación de bibliotecas

Con esto, ya podemos instalar las bibliotecas necesarias para el funcionamiento correcto del programa.

1. Para instalar las bibliotecas en Python hay un punto que lo explica (`Instalar bibliotecas`). Paso a enumerar todas las bibliotecas que hay que instalar:
   a. `Pyrtlsdr`
   b. `Numpy`
   c. `Pyserial`

Es posible que aparezca un error al instalar la biblioteca de `pyrtlsdr`. Para solucionarlo ir al punto del apartado anterior `Problema instalando pyrtlsdr en Windows`.

9.2.2.3 Diagrama de bloques del modelo



9.2.2.4 Funcionamiento del programa

Para hacer funcionar el programa hay que realizar 3 pasos:

1. Abrir una terminal en Windows, irse a la carpeta del script y ejecutar el script de la siguiente forma

```
Python3 v0.3.2.py
```

Debería salir algo como lo de a continuación.

```
PS C:\Users\famil\Dropbox\Universidad\tfg\v0\v0.3\v0.3.2> python3 v0.3.2.py
Found Rafael Micro R820T tuner
[R82XX] PLL not locked!
Exact sample rate is: 2062500.135740 Hz
```

2. Lo siguiente es abrir e-callisto y utilizarlo para leer el espectro.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR