Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de la Telecomunicación

Desarrollo de backend para control remoto de instrumentación electrónica

Autor: Manuel Jesús Toledano Mariscal
Tutores: Fernando Muñoz Chavero
Enrique Lopez Morillo

Departamento de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo de backend para control remoto de instrumentación electrónica

Autor:
Manuel Jesús Toledano Mariscal

Tutor:
Fernando Muñoz Chavero
Profesor titular
Cotutor:
Enrique López Morillo

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Grado: Desarrollo de backend para control remoto de instrumentación electrónica

Autor: Manuel Jesús Toledano Mariscal

Tutores: Fernando Muñoz Chavero

Enrique López Morillo

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de

Sevilla, 2020

El Secretario del Tribunal

A mis padres

A mi hermana

A mi abuelo

Agradecimientos

Con la entrega de este trabajo culmina una etapa en la que será por siempre una de las etapas más importantes y apasionantes de mi vida.

Agradecer en primer lugar a mi tutor Fernando Muñoz Chavero, por aceptar, y debido a la pandemia, poder haber continuado con la ejecución de este proyecto, en el que ha costado, pero al final todo acaba saliendo adelante.

Nunca será suficiente el agradecimiento a mis padres y a mi hermana, por el apoyo enorme que he tenido por parte de ellos, gracias al gran esfuerzo que han realizado, he podido finalizar el grado con valor y mucha constancia.

También agradecer a todos mis amigos que he hecho en la carrera, que como ya saben, vine desde un pueblo de Jaén solo, por lo que mi vida empezaba de nuevo haciendo nuevas amistades aquí en Sevilla, y ahora estoy súper agradecido con las personas que me he encontrado en este camino.

Seguidamente, también recordar a mis compañeros de trabajo en Everis, que en este tiempo hemos generado una amistad muy buena, en el que hemos organizado de todo. Espero continuar en contacto con todos vosotros, y que logremos todo lo que se nos proponga.

Por último, este trabajo se lo quería dedicar en especial a mi abuela, el cual estaba deseando verme con la carrera terminada, pero no pudo ser. Te echo mucho de menos abuelo. Va por ti.

Resumen

Actualmente, el análisis de señales y comprensión de estas es muy importante en el ámbito de las Telecomunicaciones, ya que, gracias a la globalización y digitalización del mundo, con las nuevas tecnologías, hace crecer una demanda de especialización en este tema.

Por otro lado, para especializarse en este tema, en la carrera es necesario conocer cómo analizar, desarrollar y ejecutar estas, ya que, para el Grado en Ingeniería de las Tecnologías de la Telecomunicación, una buena parte está destinado al uso de estas herramientas y estudio de señales que vengan de distintos dispositivos electrónicos.

Para ello, y gracias al planteamiento del Grado, las clases se dividen en partes teóricas y prácticas, para luego, en los laboratorios, estudiar en profundidad estas herramientas de medida de señales, entre otras.

De ahí surge la necesidad actual de realizar la misma parte práctica estando en el laboratorio que en remoto, o incluso sin disponer de estas herramientas de medida. Por lo que se pretende es acercar, a través de un Portal Web, este manejo de estas herramientas "a distancia", como pueden ser osciloscopios, generadores de señal, analizador de espectro, etc.

En conclusión, el objetivo de este proyecto es poder analizar una arquitectura web que satisfaga estos requerimientos de poder conectar las herramientas de medida a un portal, para poder trabajar con ellos de forma remota.

Abstract

Currently, analysis of signals and understanding of these is very important in Telecommunications field, since, thanks to the globalization and digitalization of the world, with new technologies, makes grow a demand for specialization in this topic.

On the other hand, to specialize in this topic, in the degree it's necessary to know how to analyze, develop and execute these, since, for the Degree in Engineering of Telecommunications Technologies, a good part is intended for the use of these tools and the study of signals coming from different electronic devices.

To do this, and thanks to the approach of the Degree, the classes are divided into theoretical and practical parts, and then, in the laboratories, study in depth these measurement tools of signals, among others.

Hence the current need to perform the same practical part being in the laboratory as in remote, or even without having these measuring tools. What is intended is to bring, through a Web Portal, this management of these tools "remotely", such as oscilloscopes, signal generators, spectrum analyzers, etc.

Índice

Agradecimientos *vii*
Resumen *viii*
Abstract *ix*
Índice *x*
Índice de Tablas *xii*
Índice de Ilustraciones *xiii*
Índice de Códigos *xiv*
1 Introducción *1*
2 Estado del arte *3*
2.1 Buses de instrumentación electrónica **3**
2.1.1 VISA 4
2.1.2 IVI-C 4
2.2 Servidores backend **4**
2.2.1 Lenguaje C 4
2.2.2 Lenguage Python 5
2.3 Mecanismos de comunicación entre backend y frontend **6**
2.4 Tareas de medición con los dispositivos **6**
2.4.1 Medidas en el dominio del tiempo 7
2.4.2 Medidas en el dominio de la frecuencia 7
3 Componentes del sistema *11*
3.1 Fuente de alimentación **11**
3.2 Generador de señal Agilent N8241A **12**
3.3 Osciloscopio Keysight Infinium MSO9104A **14**
3.4 Servidor Backend **14**
3.5 Servidor Frontend **15**
3.6 Base de datos **16**
4 Solución técnica *17*
4.1 Conexión Ethernet **18**
4.2 Comunicación y pruebas con los dispositivos **18**
4.3 Programación con Python **20**
4.3.1 Importar librerías 20
4.3.2 Creación del dispositivo electrónico como variable en la lógica 20
4.3.3 Establecimiento de conexión y configuración de parámetros del dispositivo 21

4.3.4 Lectura y escritura de commandos 21
4.3.5 Comandos de los dispositivos de medida 22

4.4 Arquitectura Backend 24
4.4.1 Aplicación IFFTSignals 25
4.4.2 Modelo de datos 26
4.4.3 Funciones de la aplicación 27
4.4.4 Comunicación entre frontend (Angular) y backend (Django) 29

5 Conclusiones y mejoras 33

5.1 Mejoras del sistema 34

6 Referencias 35

7 Anexo I: Código python 36

7.1 Example_origin_signal.py 36

7.2 Obtain_ifft_psd.py 38

8 Anexo II: Codigo Backend 43

1.1 Models.py 43

1.2 Views.py 44

1.3 Urls.py 61

1.4 Serializers.py 62

9 Anexo III: Otros códigos 65

9.1 CommandsKeysigth.ipsx 65

ÍNDICE DE TABLAS

Tabla 1. Funciones PyVisa 22
Tabla 2. Comandos Keysight 22

ÍNDICE DE ILUSTRACIONES

Ilustración 1. Comunicación entre frontend (Angular) y backend (Django) 6
Ilustración 2. Señal en el dominio del tiempo. 7
Ilustración 3. Señal discreta (DFT) 8
Ilustración 4. FFT de la señal anterior 8
Ilustración 5. PSD de la señal anterior 9
Ilustración 6. Componentes del sistema creado. 11
Ilustración 7. Fuente de Alimentación Keythley 2036B 12
Ilustración 8. Generador de señal Agilent N8241A 13
Ilustración 9. N8241A Control Utility 13
Ilustración 10. Osciloscopio Infinium MSO9104A 14
Ilustración 11. Aplicación web completa 15
Ilustración 12. Pantalla inicial de la web 15
Ilustración 13. Arquitectura completa de la solución técnica. (modificar imagen) 17
Ilustración 14. Keysight Connection Expert 18
Ilustración 15. Keysight Command Expert 19
Ilustración 16. Pantalla del osciloscopio 20
Ilustración 17. Resultado de la ejecución 24
Ilustración 18. Directorio de la aplicación IFFTSignals 25
Ilustración 19. Modelo de datos del sistema. 26
Ilustración 20. Resultado de la persistencia en base de datos de los modelos. 27
Ilustración 21. Url de peticiones entre Angular y Django. 30
Ilustración 22. Formulario creado en Angular. 31
Ilustración 23. Señal resultada de pulsar "configurar señal osciloscopio". 31

ÍNDICE DE CÓDIGOS

Código 1. Importar librerías 20

Código 2. Creación del objeto asignado al dispositivo de medida 21

Código 3. Establecimiento de conexión y configuración de timeout 21

Código 4. Lectura y escritura de comandos 21

Código 5. Utilización de los comandos del osciloscopio 23

Código 6. Generación de una aplicación en Django. 25

Código 7. Generación del modelo de datos dentro de la aplicación Django 26

Código 8. Confirmación del modelo de datos para persistir en base de datos. 26

Código 9. Función de reconfiguración del osciloscopio dentro de Django (Python) 27

Código 10. Habilitar ip dónde se aloja la aplicación Angular 29

Código 11. Aplicaciones instaladas en Django 30

Código 12. Dirección web de la petición principal 30

1 INTRODUCCIÓN

Actualmente en el mercado existe una gran variedad de dispositivos electrónicos para la generación de señales y medición de las mismas. De ahí surge la necesidad de poder homogeneizar estos dispositivos y que sea más intuitivo su manejo.

Por otro lado, para acercar la utilización de estos dispositivos a todo el alumnado de la Escuela, y de forma remota por los tiempos actuales, se pretende generar un portal web el cual pueda tener conectados todos estos dispositivos, y poder generar señales y mediciones de forma autónoma y a distancia, como puede ser las señales FFT, PSD, etc.

Por tanto, el objetivo del proyecto será el diseño de la arquitectura del portal web, junto con la conexión de la instrumentación electrónica necesaria para la generación y medición de señales.

Comenzaremos este proyecto explicando el estado del arte de todos los dispositivos electrónicos y su vinculación con portales web. Seguidamente, analizaremos los distintos componentes que utilizaremos en este escenario. Después, se explicará la solución que se ha llevado a cabo, con todos los pasos realizados para la creación del portal web. Por último, finalizaremos con las conclusiones y mejoras futuras para el portal.

2 ESTADO DEL ARTE

Que algo no haya salido como hayas querido no
significa que sea inútil
Thomas Edison

En el mercado tenemos distintas empresas que trabajan con dispositivos de generación y medida de señales de elementos electrónicos, como por ejemplo *Keysight Tecnhologies, Tektronix, Keythley*, etc. En primer lugar, necesitamos homogeneizar el uso de estas herramientas para poder dar una solución compatible con el mayor numero de herramientas que existan en el mercado, para así poder tener disponible un portal modularizado, es decir, que sea robusto frente a cambios, en dispositivos de medida, generación de señales, etc.

Como uno de los objetivos será poder realizar mediciones de señales de amplificadores, de ondas de radio, etc., vamos a definir unos conceptos específicos para la materia para que nos sitúe en el planteamiento del problema y a su vez conducir hasta la solución.

2.1 Buses de instrumentación electrónica

Existen distintos tipos de buses para interconectar los equipos que forman la solución técnica. Las principales características que se requieren en estos entornos son:

- Poder conectar un número de instrumentos
- Ser compatibles a un estándar que esté aceptado por la mayoría de los fabricantes
- Tener una velocidad de intercambio de datos lo suficientemente alta, de forma que la transferencia de los datos se pueda hacer casi instantáneo sin percibir saltos ni errores en las medidas.

Uno de los buses más utilizados, es el bus GPIB (General Purpose Interface Bus), el cual es un estándar de bus de datos digital, desarrollado por Hewlett-Packard en 1970, y sirve para conectar dispositivos de medición electrónica con dispositivos que controlen estas herramientas, como, por ejemplo, un ordenador.

Posteriormente, fue un estándar adoptado por la organización IEEE en 1978, el cual rige la norma IEEE-488. Por lo tanto, desde 1978 hasta la actualidad, los buses GPIB rigen las siguientes normas:

1. IEEE-488.1: Definición de los parámetros mecánicos, eléctricos y protocolo de comunicación básico de GPIB
2. IEEE-488-2: Codificación, Formateo de la comunicación, nuevos protocolos, y comandos comunes en

el estándar, como ampliación de la norma anterior

A raíz de estas normas, surgió la necesidad de crear un estándar para comandos de dispositivo, SCPI (Standard Commands for Programmable Instrumentation), que fue creado en los años 90, aunque no se ha implementado universalmente. Por ello, estudiaremos dos tipos de especificaciones de este estándar que abarcan la mayoría de las fabricantes en el mundo.

2.1.1 VISA

VISA (Virtual Instrument Standar Architecture), es una especificación definida por las empresas Agilent y National Instrument. Es un conjunto de librerías que se utiliza en la mayoría de los dispositivos de medida y se usa para desarrollar aplicaciones y drivers de input/output de forma que el software de distintas empresas pueda ser compatibles en un mismo ecosistema y puedan ser instalados utilizando simultáneamente varios medios de comunicación (por GPIB, Ethernet, etc.), y en aplicaciones desarrolladas con diferentes lenguajes (C, Python, etc.)

2.1.2 IVI-C

IVI-C, también forma un conjunto de librerías para la comunicación entre dispositivos de medida y aplicaciones desarrolladas para tal fin. La principal diferencia respecto a VISA es que tiene limitaciones respecto al lenguaje de programación utilizado, que solo es posible la programación en C o LabView, por lo que limita también el uso en distintas aplicaciones.

2.2 Servidores backend

Para poder realizar conexiones entre los dispositivos electrónicos y un ordenador, debemos buscar aquellos lenguajes de programación que sean compatibles con esta funcionalidad.

Como nuestro objetivo será la creación de un servidor backend, que almacene toda la lógica, el procesado de la señal, su medición, y su envío para su representación en un frontend, necesitamos analizar los distintos lenguajes de programación los cuales son compatibles tanto para la comunicación con los dispositivos de medida a través del bus GPIB como para el procesado lógico de estos datos en un servidor.

2.2.1 Lenguaje C

En primer lugar, hemos visto la compatibilidad que tienen las herramientas de medida con el lenguaje C, tanto a través del uso de las librerías VISA como IVI-C.

Sin embargo, hemos observado varias limitaciones que tiene este lenguaje para el objetivo del proyecto.

- Normalmente, el lenguaje C se suelte utilizar para crear software de sistemas, programas estáticos dentro de un ejecutable, y en muy pocas situaciones, aplicaciones web.
- Los datos con los que trabaja normalmente son de tipo estáticos, con un lenguaje de bajo nivel.

Por otro lado, y la principal razón que hemos descartado la utilización de este lenguaje de programación, es porque no es posible generar un servidor backend en este lenguaje, ya que no dispone de frameworks ni similares que pueda generar modelos de datos, incluso una comunicación entre otras herramientas de programación, como puede ser con aplicaciones de frontend (Angular)

2.2.2 Lenguage Python

Python es un lenguaje de programación interpretado, el cual una de las mejoras respecto a cualquier lenguaje es la legibilidad de su código, resultando una programación más intuitiva, y más sencilla a la hora de interpretarlo. Es un lenguaje de código abierto, y tiene una gran variedad de usos, desde una programación orientada a objetos, hasta generación de servidores backend con complejas arquitecturas.

Debido a estas características anteriormente descritas, además de la compatibilidad de la librería PyVISA, y la posibilidad de generar un servidor backend como objetivo de este proyecto, hemos decidido partir como base este lenguaje de programación.

PyVISA es un conjunto de librerías que permite tener el control de todos los tipos de dispositivos electrónicos independientemente de su interfaz, ya se conecten vía GPIB, Ethernet, USB, RS232.

Gracias a esta librería, facilitan tareas de monitorización y control de los instrumentos que estén conectados a un equipo remoto, ya que es compatible con la mayoría de los dispositivos de medida.

2.2.2.1 Frameworks

Un framework es el esquema o estructura que se establece y que se aprovecha para desarrollar y organizar un software determinado, es decir, es un método para hacer más sencilla la programación de cualquier aplicación.

A continuación, presentaremos unos framework de Python como objetivo de seleccionar uno de ellos para montar nuestra arquitectura backend.

a) Flask

Flask es un framework escrito en Python que permite crear aplicaciones web rápidamente y con un mínimo número de líneas de código. Por lo tanto, es denominado un “microframework”, lo que significa que el “core” de su framework es simple pero extensible a cualquier aplicación.

A diferencia de otros frameworks, Flask ofrece un proyecto “limpio”, y es el usuario el que decide todo, desde qué aplicaciones por defecto utilizar, hasta la base de datos, etc., para así lograr un mayor control sobre el código realizado. Por lo tanto, la contra que tiene es que puede llevar un poco más de tiempo configurando todo el servidor que otros framework que ayudan en ese ámbito.

En conclusión, para una persona con experiencia en el campo de la programación, la curva de aprendizaje de este framework es mayor que la utilización de otros sistemas que puede componerse a través de comandos, y generación de aplicaciones dentro del core de las mismas.

b) Django

Por su parte, Django es otro framework, el cual también permite crear aplicaciones web rápidamente, y sigue el estándar del resto opciones, pudiendo configurar todo, tanto la base de datos, como las distintas aplicaciones dentro del backend, previa generación del código a través del propio Django.

Esto es, hay que adaptarse a la metodología de programación de Django, al igual que hacen otros frameworks, tema que se ha estandarizado a la hora de realizar servidores backends, ya sea en Java, Python, etc.

Así, debido a todas las mejoras que tiene este framework, la parte del servidor backend se realizará con el framework Django, el cual es un framework desarrollado de código abierto, y mantiene el patrón de diseño MVC (Model View Controller). Por desarrollar qué es el patrón MVC, es un estilo en la arquitectura software que separa todos los componentes para que puedan funcionar de forma independiente entre ellos y puedan tener una comunicación en conjunto para lograr toda la funcionalidad que se requiera.

1. El modelo contiene la representación de los datos que se maneja en el sistema, junto a la lógica de negocio y los mecanismos para almacenar datos.
2. El controlador, es el que actúa como intermediario entre el Modelo y la Vista, el cual gestiona todo el flujo de información y comunicación, adaptando los datos a las necesidades de cada uno

2.3 Mecanismos de comunicación entre backend y frontend

Para completar toda la aplicación web, necesitamos, aparte del backend que vamos a desarrollar en este proyecto, un frontend, es decir, la parte de la aplicación la cual es usada directamente por el usuario a través de un navegador web, donde podrá manejar todas las funcionalidades del portal.

El código es ejecutado directamente en el navegador, mientras se renderiza en una página, con la ayuda de *HTML*, *CSS* y *Javascript*.

El principio de comunicación en cualquier aplicación web se basa en protocolo http en el que se puede mandar peticiones (*request*) y respuestas (*response*).

Por lo tanto, a través de este protocolo, el backend normalmente introduce el contenido en forma de JSON formateado, respuestas HTML formateadas, o código simple, siendo manejable por la parte frontend y escribiendo estas respuestas en el navegador.

El frontend envía, para comunicarse con backend, entre otros: peticiones HTTP (como *GET*, *POST*, etc.), formularios de datos, o JSON formateado.

En este ámbito del proyecto, vamos a trabajar con datos en forma de JSON formateados.

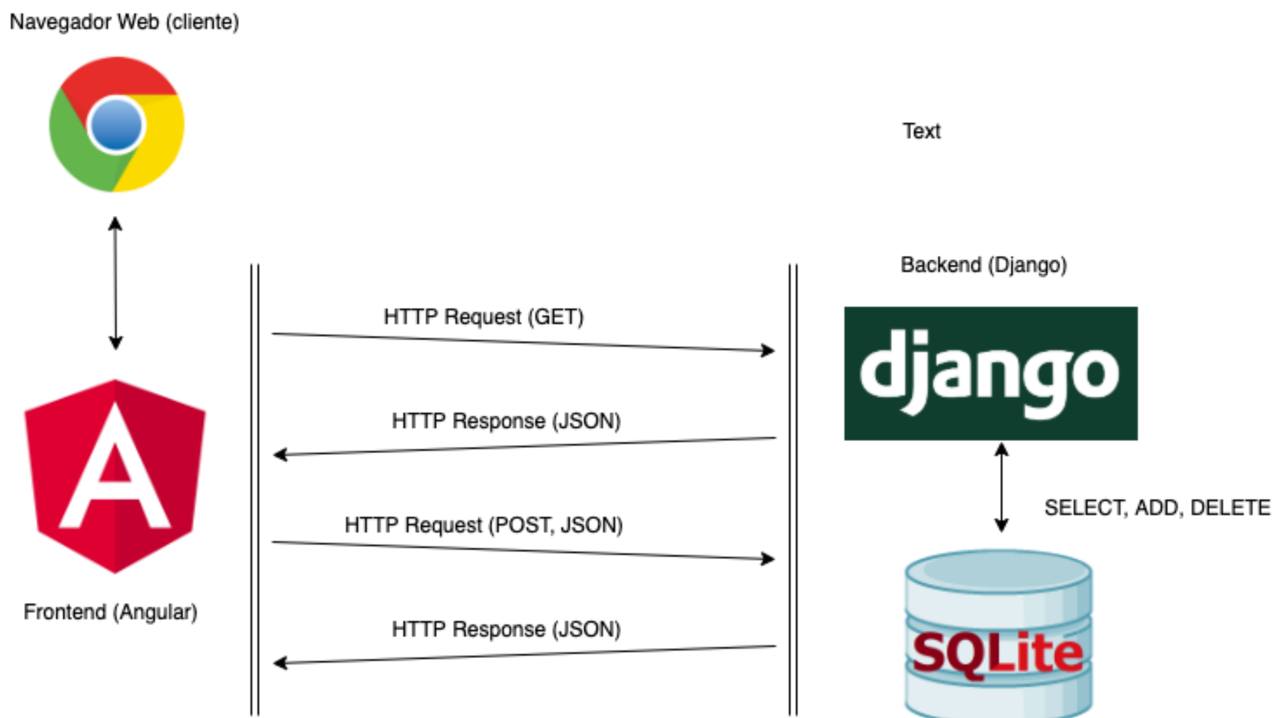

Ilustración 1. Comunicación entre frontend (Angular) y backend (Django)

2.4 Tareas de medición con los dispositivos

Por último, como objetivo de este proyecto, que será la medición de señales electrónicas de forma remota, vamos

a obtener dos tipos de mediciones: mediciones con señales en el dominio del tiempo y en el dominio de la frecuencia

2.4.1 Medidas en el dominio del tiempo

A través de los comandos comunes VISA, la medida se obtiene instantáneamente de parámetros de una señal en el dominio del tiempo, como puede ser:

- Amplitud
- Frecuencia
- Periodo de la señal
- Tensión máxima y tensión mínima
- Offset.

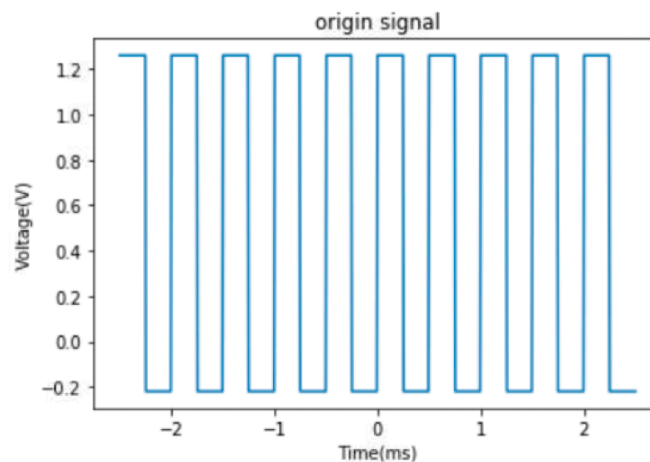

Ilustración 2. Señal en el dominio del tiempo.

2.4.2 Medidas en el dominio de la frecuencia

En el desarrollo de este proyecto, vamos a generar dos tipos de medidas en el dominio de la frecuencia principalmente: la FFT (Transformada de Fourier), y la PSD (Densidad Espectral de Potencia).

2.4.2.1 FFT (Fast Fourier Transform)

Un algoritmo FFT (Fast Fourier Transform) calcula la transformada discreta de Fourier de una secuencia (DFT), o su inversa (IFFT). Este algoritmo de Fourier convierte una señal en el dominio del tiempo, en una representación en el dominio de la frecuencia y viceversa. Por lo tanto, una FFT calcula rápidamente estas transformaciones por factorización de la matriz DFT en un producto de pocos factores.

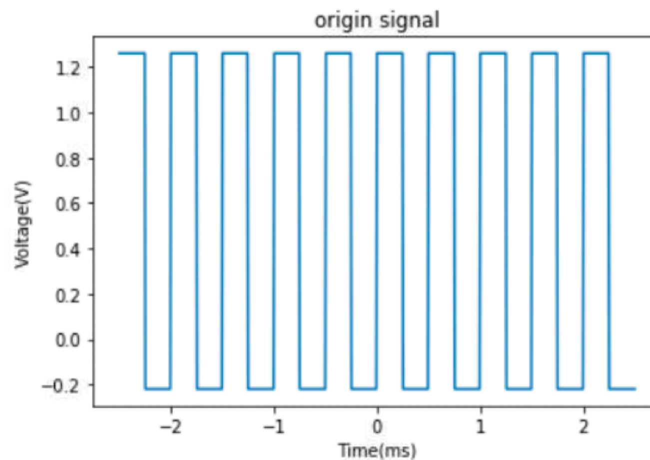

Ilustración 3. Señal discreta (DFT)

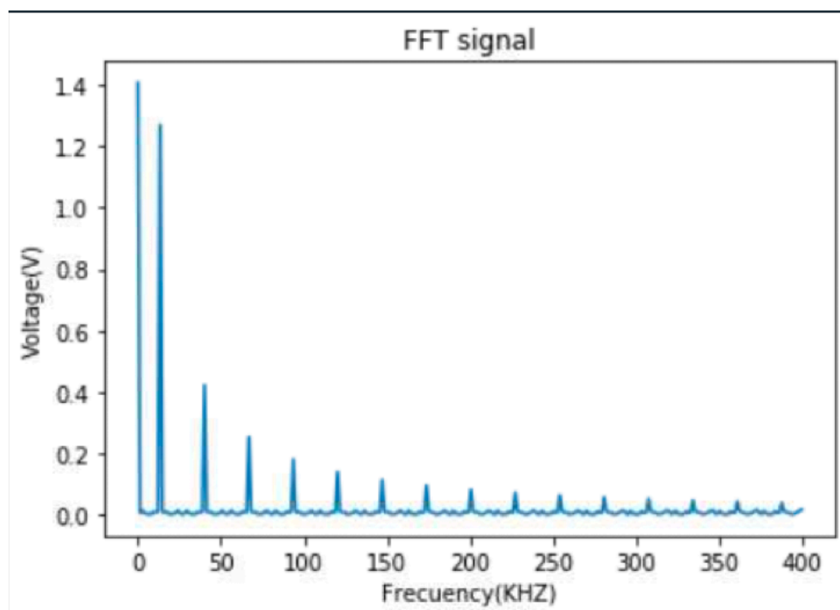

Ilustración 4. FFT de la señal anterior

Las FFT son ampliamente utilizadas para muchas aplicaciones en ingeniería, como, por ejemplo:

- Tratamiento de imágenes y audios.
- Eliminación del ruido en señales de comunicaciones
- Análisis en frecuencia de señales discretas
- Análisis de vibraciones, etc.

2.4.2.2 PSD (Power Spectral Density)

La PSD (Power Spectral Density) de una señal es una función matemática que nos informa de cómo está distribuida la potencia de dicha señal en el dominio de la frecuencia.

La Densidad Espectral de Potencia o PSD se calcula usando el Teorema de Wiener-Khinchin, el cual se relaciona la PSD con la Tranformada de Fourier o FFT.

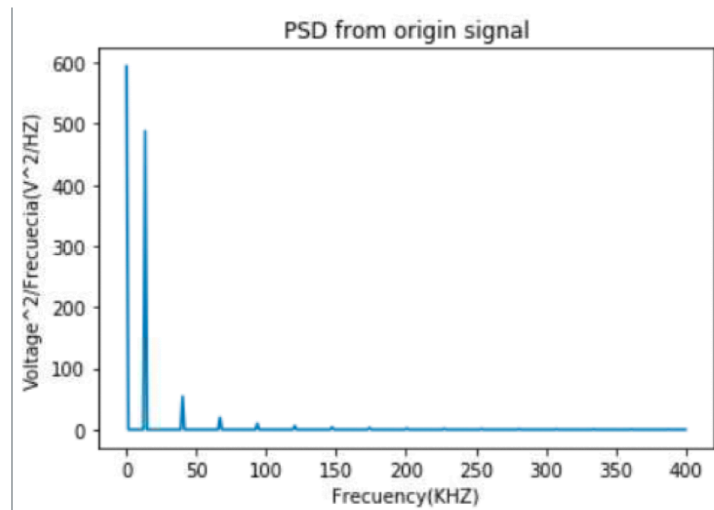

Ilustración 5. PSD de la señal anterior

La PSD sirve para dar una estimación de la Densidad Espectral de Potencia, ya que las señales no son fijas en el tiempo y no podemos determinar con absoluta precisión, a no ser que dispongamos la señal hasta el infinito, cosa que no es posible.

La función PSD la calcularemos par ver la potencia de las señales que generemos en el generador de señal y se puedan visualizar a través del osciloscopio. Esto tiene funcionalidad futura para incluir dispositivos, como amplificadores de audio o similares.

3 Componentes del sistema

La arquitectura que vamos a realizar es la siguiente: por un lado, vamos a tener la parte del portal web (frontend + backend + base de datos) y, por otro lado, vamos a conectar los diferentes dispositivos (generador de señal, osciloscopio y fuente de alimentación). Una vez realizada estas dos partes, los uniremos a través de protocolos GPIB para que tenga una comunicación bidireccional entre el portal web y los dispositivos. Por tanto, de forma esquemática, tendremos lo siguiente:

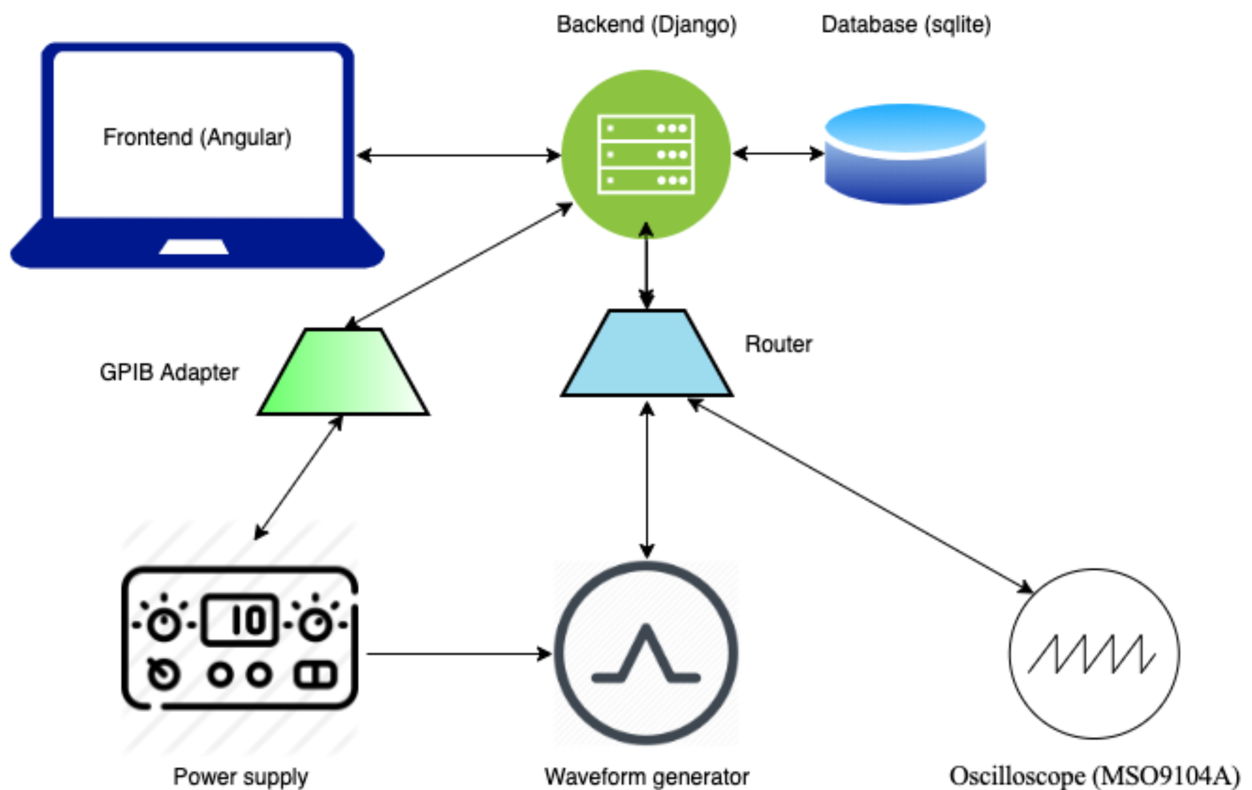

Ilustración 6. Componentes del sistema creado.

Vamos a describir todos los componentes necesarios en este escenario.

3.1 Fuente de alimentación

En primer lugar, tendremos una fuente de alimentación, trabajaremos la **2636B Series SMU**, de la empresa *Keythley*, la cual suministrará una tensión e intensidad constante al circuito. Más concretamente, tendremos siempre conectado a la corriente, para evitar que en ningún momento pueda fallar el sistema y no funcione la aplicación web.

Ilustración 7. Fuente de Alimentación Keythley 2036B

Para destacar, tenemos dos canales en los que se podrá suministrar una alimentación a los dispositivos de medida, dejando el otro canal para cualquier otro uso en el laboratorio.

Es el único equipo que no tiene comunicación ethernet como el resto de los dispositivos. Puede ir conectado mediante un adaptador GPIB-USB al equipo, a un controlador GPIB o directamente al equipo si este tiene una tarjeta **PCI-Express**. (***Peripheral Component Interconnect***)

Por otro lado, otra de las limitaciones de esta fuente de alimentación será que utiliza un protocolo denominado **TSP** (**Tunnel Setup Protocol**), que es propiedad de *Tektronix*, por lo que tiene un conjunto limitado de comandos VISA, la mayoría no responden al protocolo estándar de VISA. [1]

3.2 Generador de señal Agilent N8241A

El siguiente dispositivo es el generador de señal **Agilent N8241A**, el cual podemos generar ondas arbitrarias, de hasta 1.25GS/s y 15bits de resolución vertical, además de 500MHZ de Ancho de banda por canal, teniendo un total de 4 canales por los que podemos generar 4 señales distintas.

[1] No será objetivo del proyecto el control remoto de la fuente de alimentación, ya que vamos a tener una alimentación constante para toda la aplicación.

Ilustración 8. Generador de señal Agilent N8241A

La empresa Keysight nos proporciona un programa el cual se pueden introducir la generación de ondas arbitrarias [1].

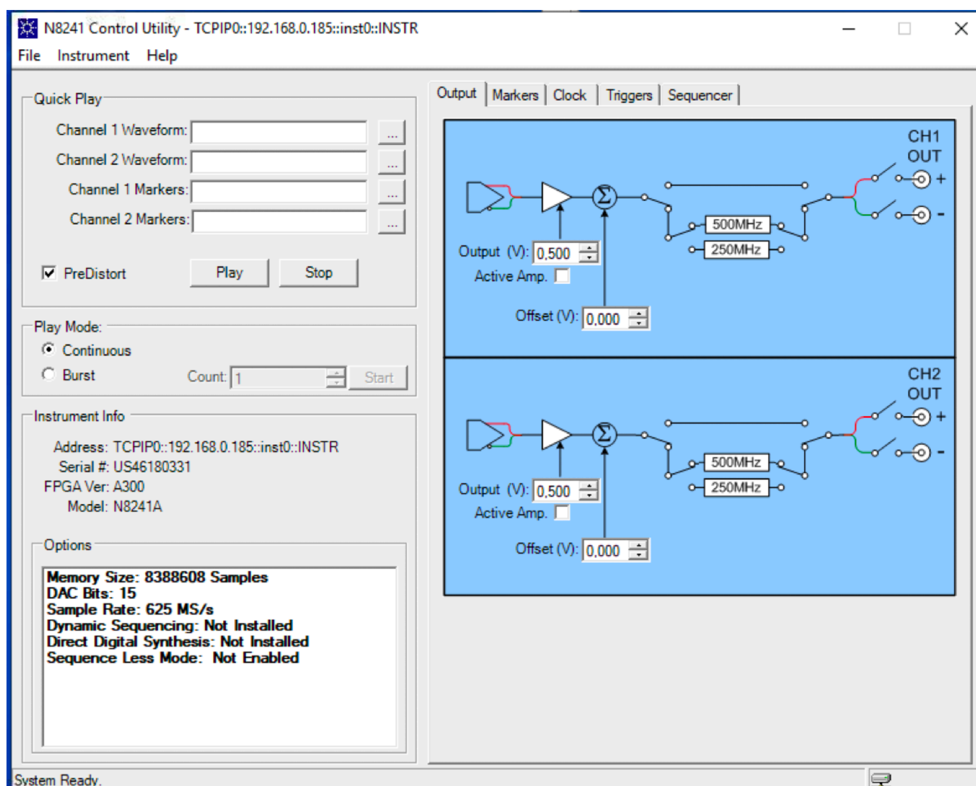

Ilustración 9. N8241A Control Utility

Para poder programarlo vía código, soporta lenguaje C, Matlab o Labview, siendo su protocolo de comunicación **SCPI** (**Standar Command Protocol Interface**), usando las librerías **IVI** (**Interchangeable Virtual Instruments**)

3.3 Osciloscopio Keysight Infinium MSO9104A

Por la parte de dispositivos de medida, vamos a utilizar el osciloscopio **Keysight Infinium MSO9104A**, el cual podemos tener señales con frecuencias de hasta 1GHz, con una frecuencia de muestreo entre los 10 GSa/s y 20 GSa/s. Tenemos 4 canales para obtener tantas señales como salidas tiene el generador de señales que hemos descrito en la sección anterior.

Ilustración 10. Osciloscopio Infinium MSO9104A

En este caso, el osciloscopio es compatible con comandos VISA. Este hecho permite que la aplicación web desarrollada en Python pueda establecer una comunicación directa con este instrumento. [2]

3.4 Servidor Backend

Para alojar toda la lógica de la aplicación web, vamos a utilizar un servidor backend denominado **Django**. Está escrito en lenguaje **Python**. Explicaremos en siguientes secciones cómo se implementará. Se ha elegido por su facilidad de escalado, donde no necesita muchos recursos hardware para funcionar correctamente. [3]

Básicamente, utilizaremos la comunicación de datos a través de peticiones ***rest***, gracias a la funcionalidad de ***Django-rest-framework***.

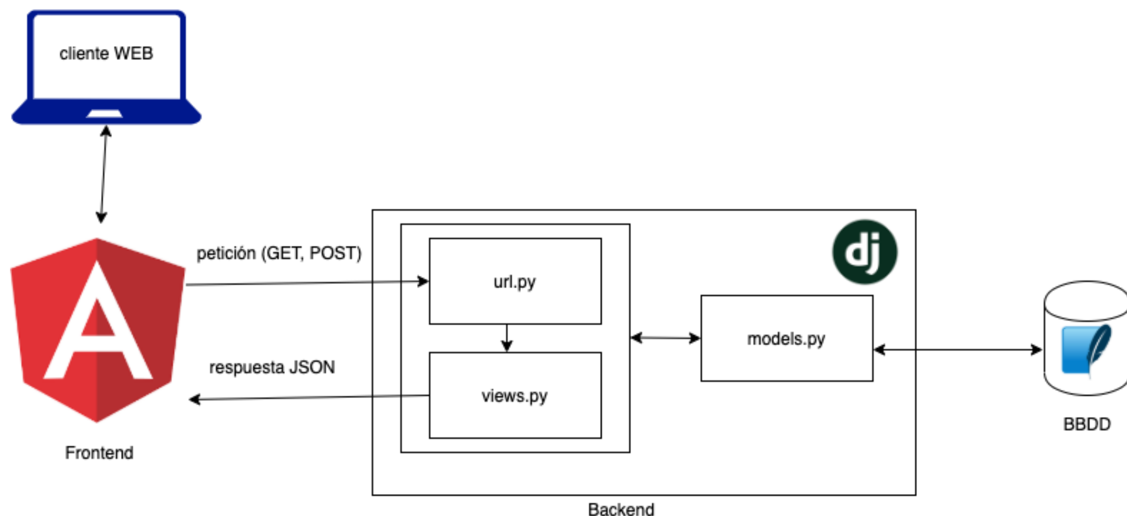

Ilustración 11. Aplicación web completa

3.5 Servidor Frontend

La parte gráfica de la aplicación web, se realizará en ***Angular*** [4], un sistema de código abierto donde se puede obtener la representación de los datos y las formas en un navegador web (Chrome, Mozilla Firefox, Edge, etc.).

Esta parte de generación del portal web en frontend no es el objetivo del proyecto, pero se incluye como parte del sistema para ofrecer una visión de la aplicación al completo.

La parte de Frontend es la encargada de incluir los distintos formularios y mandarlos por peticiones POST a backend, para así trabajarlos y devolver una respuesta JSON que pueda ser visualizada en el navegador de forma sencilla y amigable.

Un primer prototipo de la aplicación web se vería de la siguiente manera:

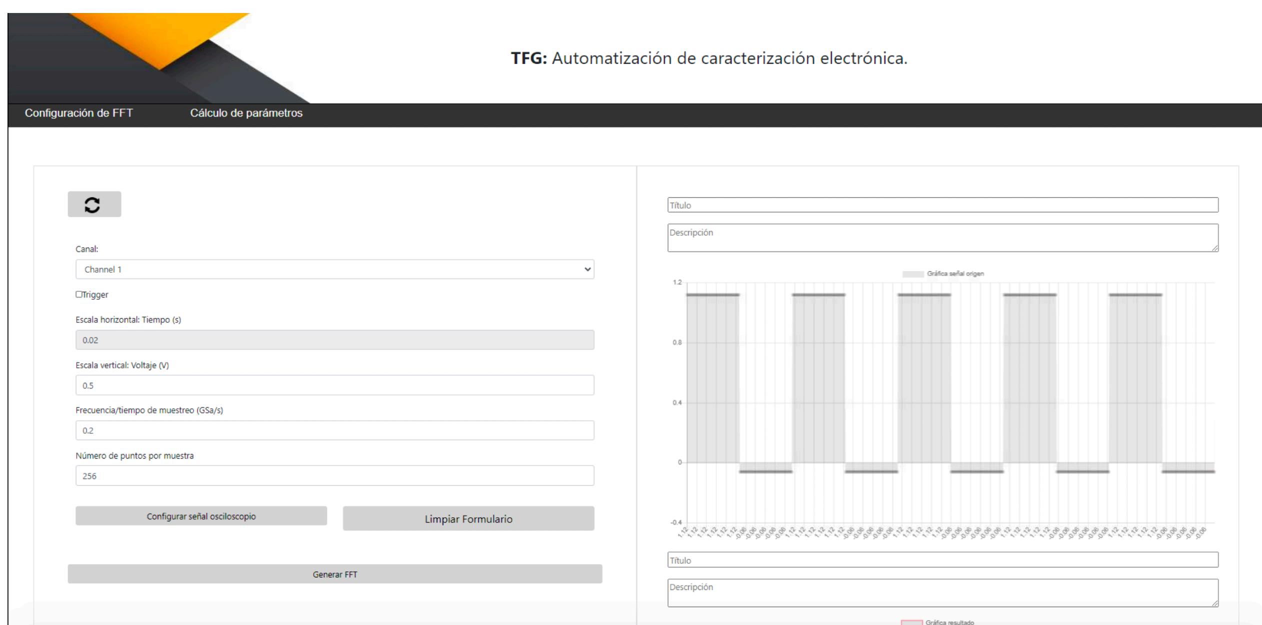

Ilustración 12. Pantalla inicial de la web

3.6 Base de datos

Para almacenar los elementos necesarios que obtengamos de los formularios web, utilizaremos una base de datos, ***sqlite*** [5], porque es una base de datos muy ligera, y permite obtener con cierta velocidad los datos persistidos. Así, podemos guardar las distintas configuraciones del osciloscopio junto a las respuestas de este, para poder enviarlas en forma de JSON a la parte de frontend.

La biblioteca implementa la mayor parte del estándar SQL-92 [6], en el cual:

- Se introdujeron nuevos tipos de datos, como DATE, TIME, VARCHAR, etc.
- Operaciones nuevas de consulta, como JOIN
- Soporte entre caracteres UTF-8, etc.

En secciones posteriores mostraremos el modelo de datos que se incluirá en la Base de datos.

4 SOLUCIÓN TÉCNICA

La arquitectura que vamos a realizar es la siguiente: por un lado vamos a tener la parte del portal web (frontend + backend + base de datos), y por otro lado vamos a conectar los diferentes dispositivos (generador de señal, osciloscopio y fuente de alimentación). Una vez realizada estas dos partes, los uniremos a través de la conexión vía Ethernet para que tenga una comunicación bidireccional entre el portal web y los dispositivos. Por tanto, de forma esquemática, tendremos lo siguiente:

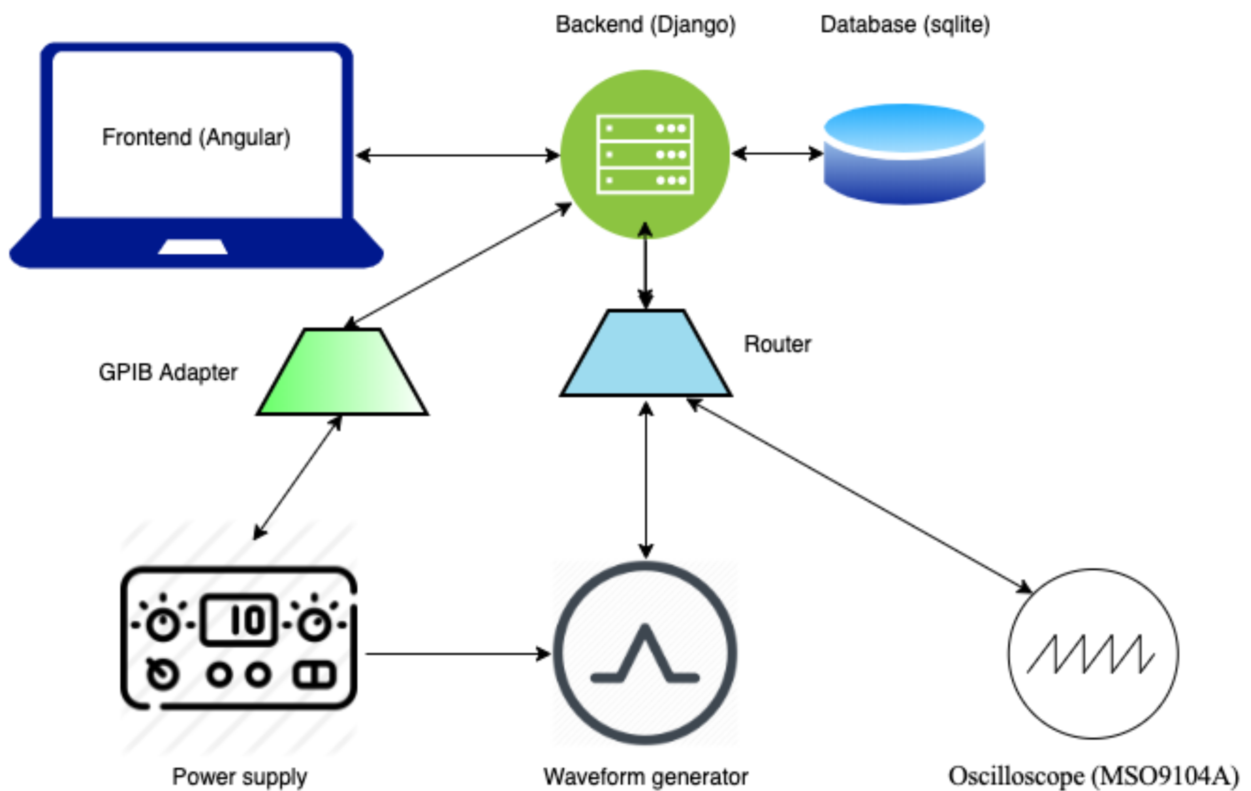

Ilustración 13. Arquitectura completa de la solución técnica. (modificar imagen)

Vamos a dividir la sección en 3 partes, desde el inicio y comunicación con los dispositivos de medida y generación de señales, hasta la comunicación en el portal web con estos dispositivos.

4.1 Conexión Ethernet

Para poder comunicarnos entre los dispositivos en un entorno automatizado, vamos a conectar tanto el osciloscopio como el generador de señal a los puertos Ethernet que tengamos disponibles en el equipo, que utilizaremos un switch para conectarlos al ordenador.. Junto a esto, necesitamos instalar en el equipo los drivers **VISA** (Virtual Instrument Estándar Architecture) y **SCPI** (Standard Commands for Programmable Instrumentation), que son librerías estándar para la comunicación entre el ordenador y los equipos.

Una vez conectados, comprobamos que el equipo detecta la conexión de ambos con la ayuda del programa *Keysight Connection Expert*, el cual nos indica las ip que se han asignado a los dispositivos, junto a toda su información (Marca, modelo, número serial, etc.).

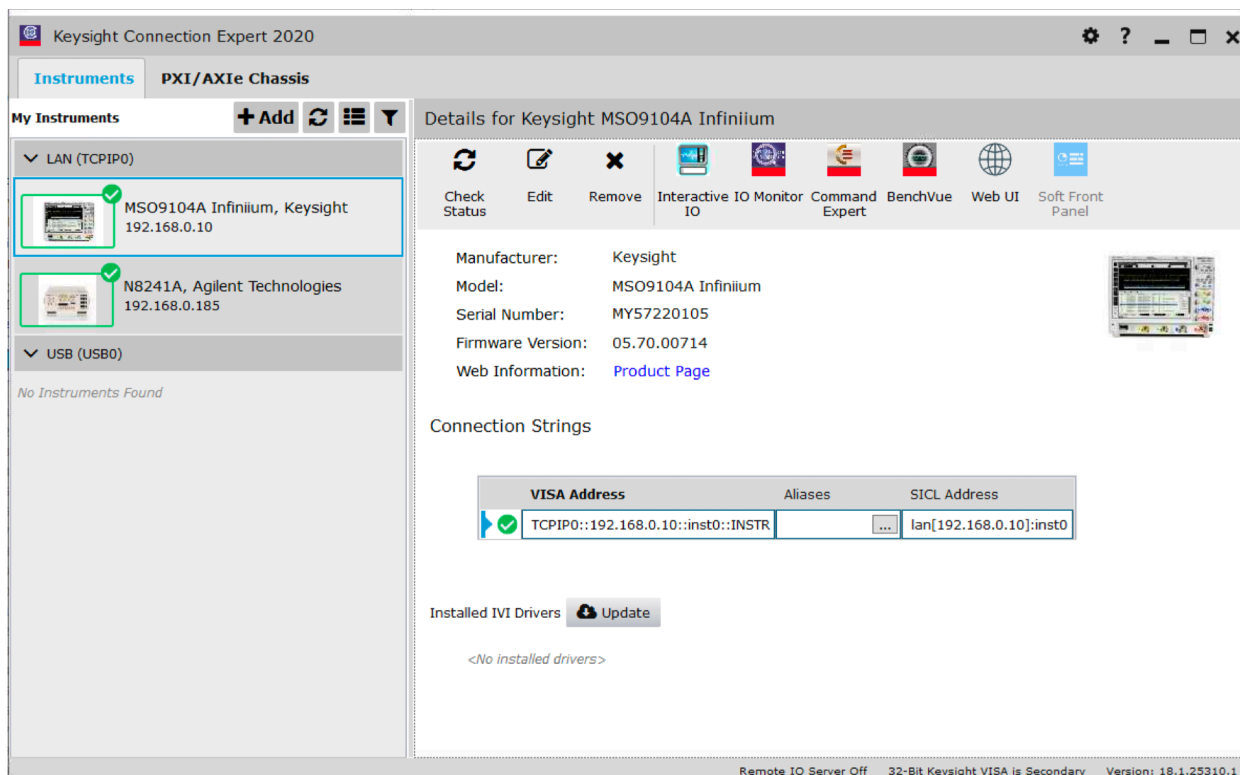

Ilustración 14. Keysight Connection Expert

Como podemos ver, la dirección LAN que nos devuelve indica que se ha conectado por Ethernet, siguiendo el protocolo TCP.

4.2 Comunicación y pruebas con los dispositivos

El siguiente paso será probar, a través de comandos SCPI, que responden correctamente y obtenemos los valores deseados. Para ello, los fabricantes disponen, por cada instrumento, un manual de programación de comandos visa disponibles en el dispositivo. Estos manuales engloban cientos de comandos que se pueden utilizar para obtener medidas, generar funciones en memoria, etc. Debido a esto, la empresa Keysight proporciona una herramienta, llamada *Keysight Command Expert*, en el que se pueden descargar las librerías de cualquier dispositivo, para que, de forma gráfica, se puedan buscar los comandos que el usuario quiera utilizar.

De esta forma, para probar, en el osciloscopio, al mandar estos comandos nos devuelve información sobre la onda que esté mostrando.

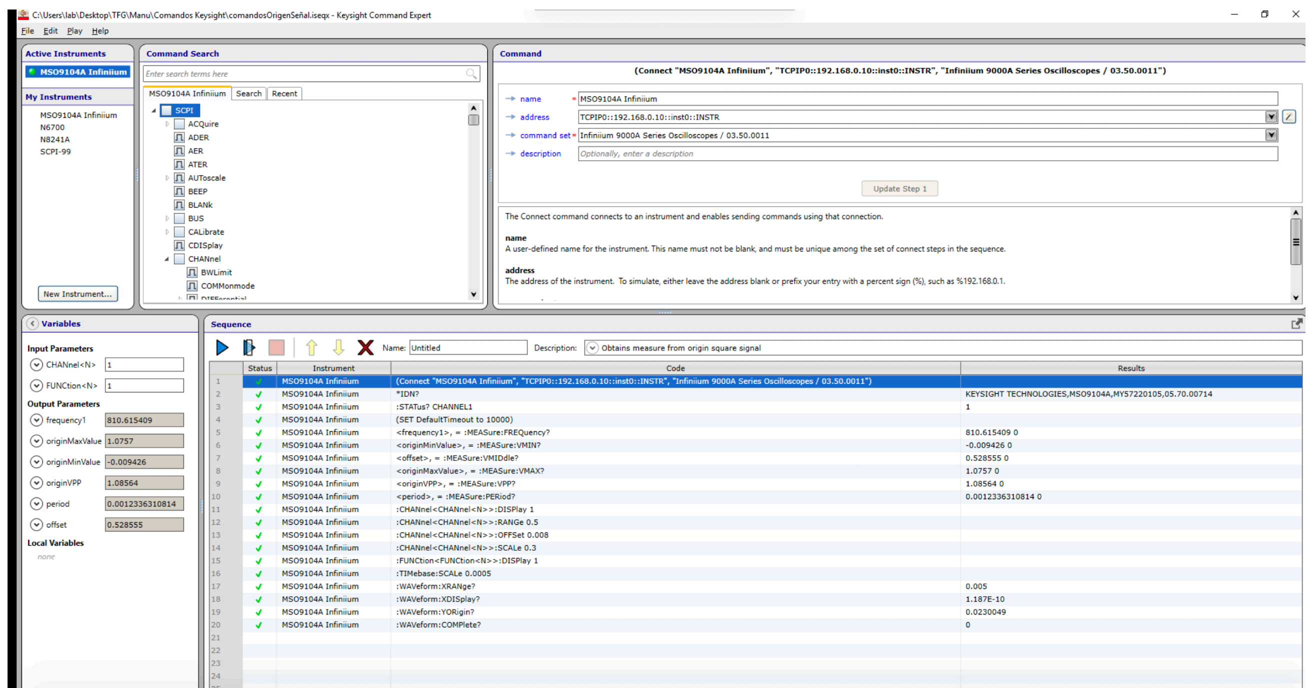

Ilustración 15. Keysight Command Expert

Tenemos las siguientes secciones en el programa:

1. Selección de comandos a buscar
2. Añadir parámetros al comando, por si necesita indicar el canal a mandar la información, un escalado específico, etc.
3. Variables que se han ido creando con los comandos, para obtener los valores alfanuméricos útiles para la medición.
4. Secuencia de comandos, donde se van incluyendo los comandos que el usuario ha seleccionado en el primer apartado, y así obtener datos del osciloscopio.

Así, para comprobar que estamos obteniendo bien los datos, nos vamos a la pantalla del osciloscopio y podemos comprobar que los datos son los mismos que reflejan dentro del osciloscopio.

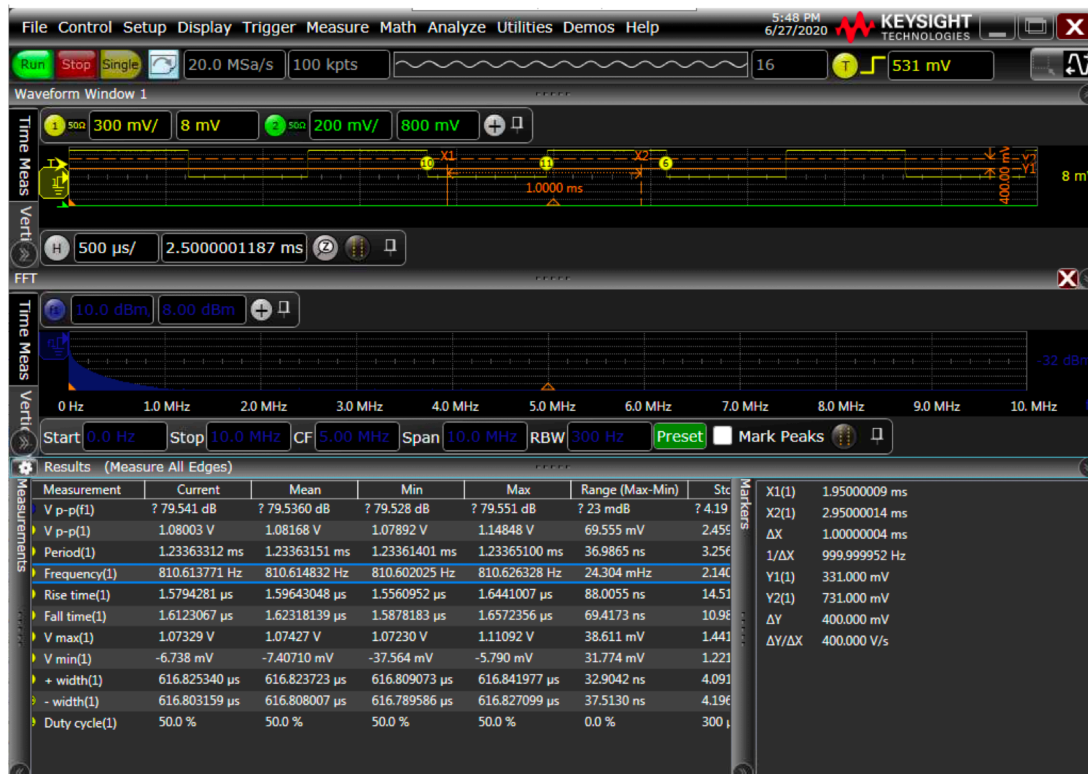

Ilustración 16. Pantalla del osciloscopio

4.3 Programación con Python

Como se ha comentado en secciones anteriores, la parte de comunicación backend del portal se va a realizar con Python. Para ello, Python dispone de las librerías VISA que se utilizan para ejecutar los comandos, similares a los que se ha visto en el apartado anterior.

A continuación, mostraremos, a través de un ejemplo, el funcionamiento de las librerías VISA:

4.3.1 Importar librerías

En primer lugar, necesitaremos importar las distintas librerías las cuales vamos a hacer uso para obtener las mediciones y realizar cálculos con estos datos:

Código 1. Importar librerías

```
import pyvisa
from scipy import signal
import numpy as np
```

En el ámbito del proyecto, utilizaremos además de las librerías ***PyVisa***, las librerías ***Numpy*** [7] y ***Scipy*** [8]. Estas últimas librerías contienen tanto funciones matemáticas para vectores y matrices de alto nivel como herramientas y algoritmos matemáticos.

4.3.2 Creación del dispositivo electrónico como variable en la lógica

Para establecer una comunicación, debemos crear el objeto que asigna al instrumento con el que se quiere realizar la misma:

Código 2. Creación del objeto asignado al dispositivo de medida

```
rm = pyvisa.ResourceManager()
MSO9104A_Infiniium = rm.open_resource('TCPIP0::192.168.0.10::inst0::INSTR')
```

En la primera línea llamamos al método ***ResourceManager*** de la librería *PyVisa.* Con ello conseguimos obtener los recursos conectados compatibles con VISA.

En la siguiente línea, se abre el instrumento que se requiera de la lista de dispositivos conectados. Se necesita los siguientes parámetros:

- *Protocolo*: tipo de protocolo utilizado para la conexión del dispositivo (GPIB, TCP/IP, etc.). En este caso es ***TCPIP0***
- *Ip*: dirección donde se encuentra el dispositivo conectado. En este caso es ***192.168.0.10::inst0***
- *Clase: tipo de clase del instrumento, viene dado por el fabricante (INSTR, INTFC, SOCKET, RAW, etc.). En este caso es* ***INSTR***

4.3.3 Establecimiento de conexión y configuración de parámetros del dispositivo

En siguiente lugar, comprobamos que el dispositivo esté conectado correctamente y devuelve una bandera en el caso que reciba los comandos.

Código 3. Establecimiento de conexión y configuración de timeout

```
identificationString = MSO9104A_Infiniium.query('*IDN?')
state = MSO9104A_Infiniium.query(':STAT? %s' % ('CHAN1'))
MSO9104A_Infiniium.timeout = 10000
```

Utilizamos el comando ***query,*** para realizar consultas al dispositivo, y nos devuelve en esta misma petición el resultado de este comando.

Por ejemplo, en la primera línea se pretende obtener el nombre del dispositivo para conocer qué dispositivo estamos estableciendo la conexión, y en la segunda línea, se consulta si está disponible (*state*) el dispositivo de medida.

Por último, se puede configurar distintos parámetros, como *timeout*, io*_protocol* (el protocolo a usar en la comunicación), etc.

4.3.4 Lectura y escritura de commandos

Una vez establecido la conexión, se procederá a leer y escribir los comandos que necesita el dispositivo de medida para comunicarse con el equipo.

Código 4. Lectura y escritura de comandos

```
temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:FREQuency?')
frequency1 = temp_values[0]
```

```
print(frequency1)


temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIN?')
originMinValue = temp_values[0]
print(originMinValue)
```

Como podemos observar, utiliza la función *query_ascii_values,* que realiza el envío del comando de medida, y obtiene, en codificación ASCII, la respuesta del dispositivo. Existen distintos comandos que se utilizan normalmente de la librería *PyVisa*, como se muestran a continuación [9]:

Tabla 1. Funciones PyVisa

Función	**Formato de salida**
query_ascii_values(*command*)	Datos en codificación ASCII
query_binary_values(*command)*	Datos en binario
write_ascii_values(*command*, *data*)	No devuelve resultados, es un método de escritura en ASCII
Write_binary_values(*command*, data)	No devuelve resultados, es un método de escritura en binario

En la mayoría de los casos de este proyecto, vamos a utilizar el primero de las funciones, para mandar comandos de consulta y recibir una serie de datos en formato ASCII para trabajar a partir de estos.

4.3.5 Comandos de los dispositivos de medida

Por último, y para cerrar los componentes que necesitamos para generar las funciones en Python son los comandos necesarios para mandar peticiones a los dispositivos de medida.

En el ejemplo anterior, estamos obteniendo distintos parámetros de una señal que se está mostrando en el oscilosocpio. Una vez consultado el manual de programación del *Keysight Infinium Oscilloscope* anteriormente explicado [2], utilizamos los siguientes comandos:

Tabla 2. Comandos Keysight

Comando	**Descripción**
:MEAS:FREQuency?	Frecuencia de la señal
:MEAS:VMIN?	Tensión máxima de la señal
:MEAS:VMAX?	Tensión mínima de la señal
:MEAS:VPP?	Tensión pico-pico de la señal
:MEAS:PERiod?	Periodo de la señal
:MEAS:VMIDdle?	Tensión media (offset) de la señal

Código 5. Utilización de los comandos del osciloscopio

```
temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:FREQuency?')
frequency1 = temp_values[0]
print("============")
print(frequency1)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIN?')
originMinValue = temp_values[0]
print("============")
print(originMinValue)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMAX?')
originMaxValue = temp_values[0]
print("============")
print(originMaxValue)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VPP?')
originVPP = temp_values[0]
print("============")
print(originVPP)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:PERiod?')
period = temp_values[0]
print("============")
print(period)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIDdle?')
originOffset = temp_values[0]
print("============")
print(originOffset)
print("============")
```

Como resultado de la ejecución, obtenemos lo siguiente:

```
====Frecuency========
810.614496
============
====MinValue========
-0.04477
============
=====MaxValue=======
1.11958
============
=====VPP=======
1.1722
============
=====Period=======
0.001233612265
============
=====Offset=======
0.52937
============
```

Ilustración 17. Resultado de la ejecución

Como se puede observar, gracias a la librería VISA de Python, se puede mandar comandos de forma usable para el usuario, y así obtener los datos que se desee. El resto de los ejemplos de scripts de Python se indicarán en el Anexo I de esta memoria.

4.4 Arquitectura Backend

Por último, para cerrar el proceso de creación de la parte backend con la comunicación con los dispositivos, vamos a incluir estos scripts en distintos métodos y funciones para que se pueda establecer la comunicación a su vez con la parte frontend.

Como hemos comentado anteriormente en el apartado 2, la arquitectura que utilizaremos será **Django** basado en **Python**. Hemos utilizado este backend porque es muy robusto y flexible a añadir nuevas funcionalidades, ya que se pueden ir añadiendo módulos y lógica a medida que cambien o añadan nuevos requerimientos.

La versión que vamos a utilizar es la 2.1, la cual tiene soporte para la versión de Python 3.7.

En este caso, como comentamos el patrón de diseño de Django, nos vamos a centrar en el modelo y el controlador de la web, siendo la Vista representada en **Angular**.

Con esto, generalmente, cuando se desarrolla una aplicación en Django, se generan distintos archivos necesarios para la correcta funcionalidad requerida. Los archivos más importantes sobre los que vamos a trabajar son los siguientes:

- ***Models.py***: aquí incluiremos el modelo de datos que describiremos en el siguiente apartado. Formaría parte del Modelo dentro del Patrón de Diseño.
- ***Views.py***: en este archivo incluiremos todos los métodos y funciones que servirán para tratar los datos y preparar los datos para mandarlos. Formaría parte tanto del modelo como del controlador, ya que disponemos de todas las funciones que servirán para enviar los datos necesarios al frontend.
- ***Urls.py***: en este archivo se localizarán los métodos de comunicación entre Angular y Django. Formaría parte del Controlador de la aplicación.

Para esta primera versión de la aplicación, vamos a utilizar el backend para fundamentalmente dos tareas:

1. Almacenar los datos de un formulario de configuración de parámetros del osciloscopio, mandar los comandos de configuración al osciloscopio para que nos devuelva la señal, y mandar a frontend los datos para generar en una gráfica la representación similar a lo que hay en el osciloscopio.
2. Generación de la FFT/PSD de la señal que hemos obtenido en el osciloscopio gracias a la librería *numpy* de Python, que sirve para generar distintos tipos de señales a raíz de los datos obtenidos en el osciloscopio.

Por lo tanto, para definir la arquitectura de la parte backend, necesitaremos los siguientes componentes:

1. Creación de la aplicación backend del sistema.
2. Modelo de datos, donde se definirán los objetos que servirán para almacenar y aplicar la lógica de la funcionalidad de la aplicación.
3. Funciones y métodos necesarios para lograr los requerimientos de la aplicación.
4. Comunicación entre la parte backend y frontend.

4.4.1 Aplicación IFFTSignals

En primer lugar, generamos la aplicación dentro del servidor Django. Para ello, nos situamos en la carpeta del proyecto Python, y desde la consola de comando escribimos

Código 6. Generación de una aplicación en Django.

```
python3 manage.py startapp IFFTSignals
```

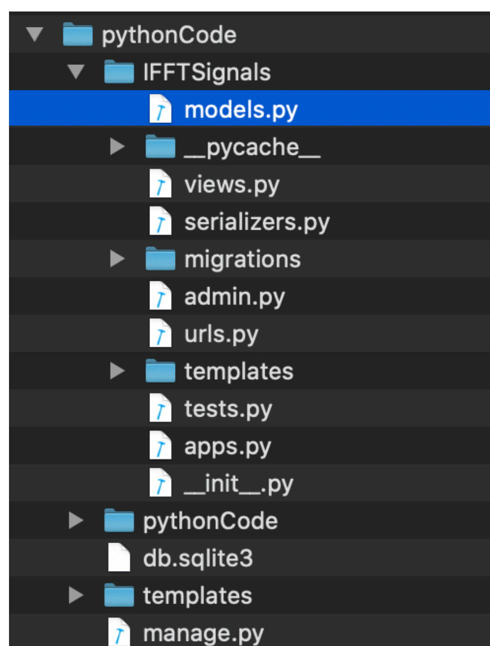

Ilustración 18. Directorio de la aplicación IFFTSignals

Con esto, automáticamente obtenemos un directorio donde tenemos que incluir todo el código de la aplicación anteriormente descrito.

4.4.2 Modelo de datos

Para montar la arquitectura, necesitamos definir unos objetos para poder manipular la información dentro del backend. Vamos a crear dos objetos: uno para la parte de entradas del formulario, y otro para la parte de la respuesta del osciloscopio.

Entradas
Canal: Canal del osciloscopio a configurar
Voltaje: Configuración del eje X del osciloscopio
Tiempo: Configuración del eje Y del osciloscopio
fMuestreo: Frecuencia de muestreo aplicado al osciloscopio
trigger: disparador del osciloscopio, para refrescar la señal en el mismo

Salida
Entrada: key que indica la entrada a la que pertenece la salida del osciloscopio
EjeX: array de datos del eje X del osciloscopio
EjeY: array de datos del eje Y del osciloscopio

Ilustración 19. Modelo de datos del sistema.

Una vez los hayamos incluido, con cada tipo de contenido, ya sea *string, number, date, etc.,* lo persistimos en la base de datos, con el siguiente comando.

Código 7. Generación del modelo de datos dentro de la aplicación Django

```
python3 manage.py makemigrations
```

Donde nos confirma qué modelos queremos persistir en la base de datos. Una vez afirmado que se van a crear los modelos deseados, realizamos un último comando para persistirlos.

Código 8. Confirmación del modelo de datos para persistir en base de datos.

```
python3 manage.py migrate
```

Con esto, se generará dos tablas en la base de datos, para almacenar cada tipo de objeto en su correspondiente tabla.

IFFTSignals_entradas	
id	integer
title	varchar(100)
canal	varchar(100)
voltaje	decimal
fMuestreo	decimal
trigger	bool
tiempo	decimal
IFFTSignals_resultados	
id	integer
title	varchar(100)
ejeX	varchar(300)
ejeY	varchar(300)
date	datetime
entrada_id	integer

Ilustración 20. Resultado de la persistencia en base de datos de los modelos.

4.4.3 Funciones de la aplicación

Esta parte es el núcleo de la aplicación, donde se van a encontrar todos los métodos disponibles tanto para tratar los datos que provengan de la parte frontend, como para mandar y recibir los datos provenientes de los dispositivos VISA.

Como código de ejemplo para mostrar una funcionalidad de las realizadas, (el resto de los métodos y funciones estarán en el Anexo II) se compone de, una vez recibido los parámetros de configuración del osciloscopio, se utiliza distintos métodos de la librería NumPy, así generar los datos correctos para mandar la señal a Angular y que pueda ser representada:

Código 9. Función de reconfiguración del osciloscopio dentro de Django (Python)

```
def reconfigureOsciloscope(entrada):
    print("reconfiguramos el osciloscopio y mandamos la senyal a Angular")
    rm = pyvisa.ResourceManager()
    canal = entrada.canal
    voltaje = entrada.voltaje
    tiempo = entrada.tiempo
    fMuestreo = entrada.fMuestreo
    CHANnel_N_ = 1

    if canal == 'CANAL 1':
        canal == 'CNAN1'
    elif canal == 'CANAL 2':
        canal == 'CNAN2'
    elif canal == 'CANAL 3':
        canal == 'CNAN3'
    else:
```

```
        canal == 'CNAN1'

    MSO9104A_Infiniium = rm.open_resource('TCPIP0::192.168.0.10::inst0::INSTR')
    identificationString = MSO9104A_Infiniium.query('*IDN?')
    state = MSO9104A_Infiniium.query(':STAT? %s' % canal)
MSO9104A_Infiniium.write(':CHANnel%d:DISPlay %d' % (CHANnel_N_, 1))
MSO9104A_Infiniium.write(':CHANnel%d:RANGe %G' % (CHANnel_N_, 0.5))
MSO9104A_Infiniium.write(':CHANnel%d:OFFSet %G' % (CHANnel_N_, 0.008))
MSO9104A_Infiniium.write(':CHANnel%d:SCALe %G' % (CHANnel_N_, 0.3))
MSO9104A_Infiniium.write(':TIMebase:SCALe %G' % (0.0005))
MSO9104A_Infiniium.timeout = 20000

    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:FREQuency?')
    frequency1 = temp_values[0]

    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIN?')
    originMinValue = temp_values[0]

    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMAX?')
    originMaxValue = temp_values[0]

    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VPP?')
    originVPP = temp_values[0]

    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:PERiod?')
    period = temp_values[0]

    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIDdle?')
    originOffset = temp_values[0]

    # Sampling rate 1000 hz / second
    # es el eje X muestreado
    t = np.linspace(-2.5, 2.5, int(frequency1), endpoint=True)

    # Plot the square wave signal
    print("============")
    print(t)
    print("============")
```

```
    # esto es el eje x
    amplitude = originVPP / 2  # Vpp/2
    offset = originOffset  # Offset
    waveform = signal.square(2 * np.pi * int(frequency1) * t)
    waveform = amplitude * waveform + offset
    print("============")
    print(waveform)
    print("============")
    # esto es el eje y

    data = {
        'ejex': np.array(t).tolist(),
        'ejey': np.array(waveform).tolist()
    }

    MSO9104A_Infiniium.close()
    rm.close()
    return data
```

Por último, la variable *data* es el conjunto de arrays que se manda al frontend para que pueda representar la señal dentro del portal.

4.4.4 Comunicación entre frontend (Angular) y backend (Django)

Para habilitar la comunicación de Django hacia Angular hay que realizar 3 pasos:

1. En el fichero ***settings.py*** del servidor Django, hay que habilitar la url dónde se aloja la aplicación de Angular, en este caso:

Código 10. Habilitar ip dónde se aloja la aplicación Angular

```
CORS_ORIGIN_WHITELIST = [
    "http://localhost:4200",
]
```

2. En el mismo fichero, incluir las aplicaciones instaladas en el servidor, además de la comunicación rest api:

Código 11. Aplicaciones instaladas en Django

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'IFFTSignals',
    'rest_framework',
    'corsheaders',
]
```

`Rest_framework` y `corsheaders`, son los protocolos de comunicación que hay que instalar en Django (`pip install djangorestframework` y `pip install corsheaders`)

3. Por último, en el archivo **urls.py**, se incluirán las peticiones que se realizarán desde Angular, se traducirán en Python, y mandará estas peticiones a los métodos que corresponda. Por ejemplo, si se quiere obtener la señal de la función descrita en el apartado 1, desde Angular tiene que realizar una petición POST a la url:

Código 12. Dirección web de la petición principal

```
$URL_DJANGO/signalConfiguration
```

```
# Create your views here.
urlpatterns = [
    url(r'^getOrigenSignal', views.init_visa),
    url(r'^signalConfiguration/(?P<pk>\d+)/edit', views.ifft_edit),
    url(r'^signalConfiguration/(?P<pk>\d+)/delete', views.ifft_delete),
    url(r'^signalConfiguration/(?P<pk>\d+)/', views.ifft_detail),
    url(r'^signalConfiguration$', views.ifft_add),
    url(r'^originSignal/(?P<pk>\d+)/', views.ifft_origin),
    url(r'^signalConfiguration/all', views.ifft_all_entries),
    url(r'^fft', views.send_ifft)
]
```

Ilustración 21. Url de peticiones entre Angular y Django.

Una vez realizado todas estas configuraciones, arrancamos el servidor con el siguiente comando:

```
python3 manage.py runserver
```

A modo de ejemplo visual, si realizamos las acciones descritas anteriormente, obtenemos la siguiente pantalla:

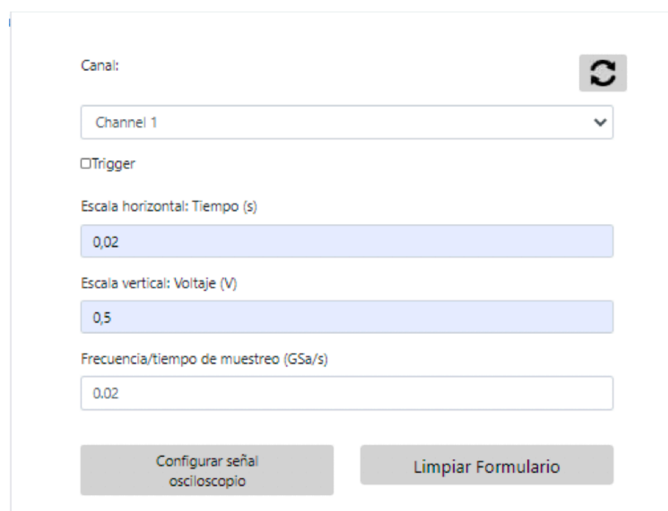

Ilustración 22. Formulario creado en Angular.

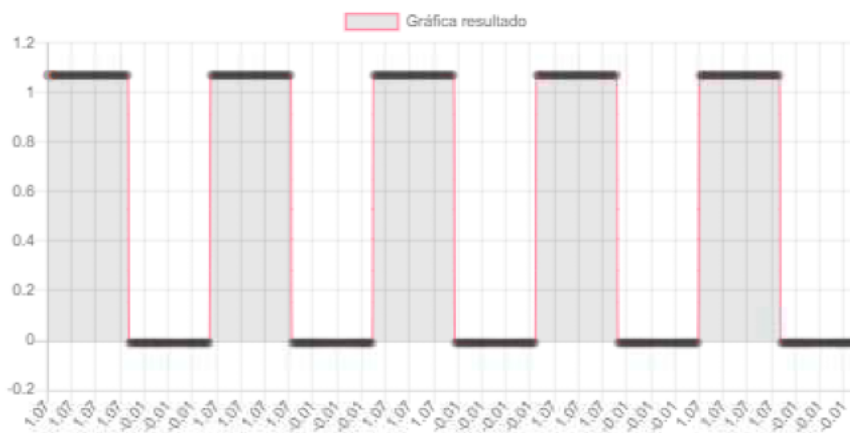

Ilustración 23. Señal resultada de pulsar "configurar señal osciloscopio".

5 CONCLUSIONES Y MEJORAS

EL objetivo del proyecto ha sido el análisis y creación de una aplicación web para poder adquirir medidas de forma remota, basándose en una arquitectura de **backend**, escrito en **Python** y utilzando un framework denominado **Django**.

Para cumplir los requerimientos del proyecto, se ha realizado una serie de funciones las cuales, permite tanto la comunicación con los equipos de medida, como el procesamiento de los datos y muestra en un portal web real.

Las ventajas de haber realizado una primera arquitectura para este requerimiento es: la modularidad del servidor, donde se pueden ir añadiendo distintos tipos de funciones para satisfacer requerimientos de medida nuevo, o generación de funciones matemáticas nuevas sin ningún tipo de problema, debido a su sencilla implementación, gracias a las librerías **PyVisa**, **NumPy** y **SciPy**.

Esta aplicación web es un primer punto para poder generar un gran Portal en el que se puedan añadir funciones prácticas para poder medir cualquier dispositivo (de radio, electromagnético), que dispongamos en los laboratorios.

Este proyecto me ha permitido desarrollarme en el ámbito de la programación web, más concretamente en la parte de programación con Python, unido a la comunicación con dispositivos electrónicos, como puede ser el osciloscopio o el generador de señal, para así facilitar y acercar el uso de estos dispositivos de forma más sencilla y autónoma.

Por otro lado, también ha servido para trabajar en equipo, junto a la otra parte del proyecto desarrollada en Angular, ya que hemos aprendido metodologías de trabajo y mecanismos para poder obtener un producto en buenas condiciones.

Gracias a este proyecto, es un primer punto de partida para el Departamento de Electrónica, poder crear una página web, accesible para todo el alumnado, con funcionalidades que antes se requería el uso presencial de estos dispositivos, poder utilizarlos y configurarlos de forma remota.

5.1 Mejoras del sistema

Una vez validada la arquitectura y adquisición de los datos propuesta, los futuros desarrollos pueden estar destinados a:

- **Conexión con las tarjetas de adquisición de datos.** Esta mejora estaba incluida en la propuesta de este desarrollo del proyecto, pero debido a la situación de no poder asistir al laboratorio presencialmente, se ha dejado como mejora en un futuro. Esto es, conexión de una placa para medir la FFT, voltaje a la salida, frecuencia, etc., para que, dada una señal de entrada, la placa, siendo un amplificador, aumente la señal a la salida, y refleje los datos en la web.
- **Modularidad para incluir cualquier dispositivo de medición**. Propondría modificar los métodos de la parte backend, para que obtengan los tipos de medida genéricos, ya sea medición de la amplitud, frecuencia, tipo de señal, etc., y tener la posibilidad, mediante la inclusión de un formulario y un nuevo modelo de datos, incluir la marca del dispositivo de medición, y los comandos necesarios para obtener las mediciones, así, la aplicación web es más versátil y escalable a cualquier dispositivo de medición que haya en el laboratorio.
- **Incluir los métodos para poder generar señales arbitrarias desde la aplicación web.** El dispositivo que hemos utilizado en este proyecto solo era compatible generar un archivo compilable en C, para generar las señales que se deseen. Existe una posibilidad de llamar, a través de un método en Python, mandar los parámetros necesarios a este archivo en código C, y que este código se encargue de generar la señal. Posteriormente, tenemos las medidas que nos devuelve el osciloscopio y se podrá visualizar y trabajar con ella desde la aplicación web
- **Sacar un dominio para la aplicación.** Almacenar el proyecto en una máquina la cual tenga un dominio específico y sea accesible remotamente desde casa, no sólo desde la escuela.

6 REFERENCIAS

[1] Keysight Control Utility: https://www.keysight.com/main/software.jspx?ckey=1670471&lc=eng&cc=US&nid=-11143.0.00&id=1670471)

[2] El manual de los comandos necesarios para trabajar con este osciloscopio se encuentra en:

https://www.keysight.com/upload/cmc_upload/All/Infiniium_prog_guide.pdf

[3] Puede verse todas sus funcionalidades y cómo trabajar con el framework Djangol en: https://www.djangoproject.com/

[4] Angular: https://angular.io

[5] Sqlite: https://www.sqlite.org/index.html

[6] Standar SQL-92: https://es.wikipedia.org/wiki/SQL-92

[7] Librería Numpy: https://numpy.org/

[8] Librería SciPy: https://www.scipy.org/

[9] Funciones PyVisa: https://pyvisa.readthedocs.io/en/latest/api/resources.html

7 Anexo I: Código python

7.1 Example_origin_signal.py

```
import pyvisa
from scipy import signal
import numpy as np
#import time
#import matplotlib.pyplot as plot



CURRENT_FREQ = 200000
FREQ_INCREMENT = 10000
ENDING_FREQ = 500000
measFreqCh1=0

# Obtains measure from origin square signal
rm = pyvisa.ResourceManager()
MSO9104A_Infiniium = rm.open_resource('TCPIP0::192.168.0.10::inst0::INSTR')
identificationString = MSO9104A_Infiniium.query('*IDN?')
state = MSO9104A_Infiniium.query(':STAT? %s' % ('CHAN1'))
MSO9104A_Infiniium.timeout = 10000
temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:FREQuency?')
frequency1 = temp_values[0]
result_state = int(temp_values[0])
print("====Frecuency========")
print(frequency1)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIN?')
originMinValue = temp_values[0]
result_state1 = int(temp_values[0])
print("====Vmin========")
print(originMinValue)
print("============")
```

```
temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMAX?')
originMaxValue = temp_values[0]
result_state2 = int(temp_values[0])
print("====Vmax========")
print(originMaxValue)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VPP?')
originVPP = temp_values[0]
result_state3 = int(temp_values[0])
print("====Vpp========")
print(originVPP)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:PERiod?')
period = temp_values[0]
result_state4 = int(temp_values[0])
print("====Period=======")
print(period)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIDdle?')
originOffset = temp_values[0]
result_state5 = int(temp_values[0])
print("=====Offset=======")
print(originOffset)
print("============")



# Sampling rate 1000 hz / second
#es el eje X muestreado
t = np.linspace(-2.5, 2.5, int(frequency1), endpoint=True)

# Plot the square wave signal
print("============")
print(t)
print("============")
```

```
#esto es el eje y
amplitude = originVPP/2 #Vpp/2
offset = originOffset #Offset
waveform = signal.square(2 * np.pi * 810 * t)
waveform = amplitude * waveform + offset
print("============")
print(waveform)
print("============")
#esto es el eje x

MSO9104A_Infiniium.close()
rm.close()
```

7.2 Obtain_ifft_psd.py

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.fftpack
import pyvisa
from scipy import signal
import numpy as np

# Number of samplepoints
N = 600
# sample spacing
T = 1.0 / 800.0

CURRENT_FREQ = 200000
FREQ_INCREMENT = 10000
ENDING_FREQ = 500000
measFreqCh1=0

# Obtains measure from origin square signal
rm = pyvisa.ResourceManager()
MSO9104A_Infiniium = rm.open_resource('TCPIP0::192.168.0.10::inst0::INSTR')
identificationString = MSO9104A_Infiniium.query('*IDN?')
```

```
state = MSO9104A_Infiniium.query(':STAT? %s' % ('CHAN1'))
MSO9104A_Infiniium.timeout = 10000
temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:FREQuency?')
frequency1 = temp_values[0]
result_state = int(temp_values[0])
print("============")
print(frequency1)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIN?')
originMinValue = temp_values[0]
result_state1 = int(temp_values[0])
print("============")
print(originMinValue)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMAX?')
originMaxValue = temp_values[0]
result_state2 = int(temp_values[0])
print("============")
print(originMaxValue)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VPP?')
originVPP = temp_values[0]
result_state3 = int(temp_values[0])
print("============")
print(originVPP)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:PERiod?')
period = temp_values[0]
result_state4 = int(temp_values[0])
print("============")
print(period)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIDdle?')
```

```
originOffset = temp_values[0]
result_state5 = int(temp_values[0])
print("============")
print(originOffset)
print("============")



# Sampling rate 1000 hz / second
#es el eje X muestreado
t = np.linspace(-2.5, 2.5, int(frequency1), endpoint=True)

# Plot the square wave signal
print("============")
print(t)
print("============")
#esto es el eje x
amplitude = originVPP/2 #Vpp/2
offset = originOffset #Offset
waveform = signal.square(2 * np.pi * 810 * t)
waveform = amplitude * waveform + offset
print("============")
print(waveform)
print("============")
#esto es el eje y

x = t
y = waveform

fig4, ax4 = plt.subplots()
ax4.plot(x,y)
plt.xlabel('Time(ms)')
plt.ylabel('Voltage(V)')
plt.title("origin signal")
plt.show()

#x = np.linspace(0.0, N*T, N)
#y = np.sin(50.0 * 2.0*np.pi*x) + 0.5*np.sin(80.0 * 2.0*np.pi*x)
yf = scipy.fftpack.fft(y)
```

```
xf = np.linspace(0.0, 1.0/(2.0*T), N//2)

fig, ax = plt.subplots()
ax.plot(xf, 2.0/N * np.abs(yf[:N//2]))
plt.xlabel('Frecuency(KHZ)')
plt.ylabel('Voltage(V)')
plt.title("FFT signal")
plt.show()



iifty = np.fft.ifft(yf);
fig2, ax2 = plt.subplots()
ax2.plot(xf, 2.0/N * np.abs(iifty[:N//2]))
plt.title("IFFT from origin signal")
plt.show()

 #Generate PSD from signal
fs = 250
t = np.arange(0, 10, 1/fs)
#xf = np.linspace(0, fs, len(t))
yf = abs(yf)**2

fig3, ax3 = plt.subplots()
ax3.plot(xf, 2.0/N * np.abs(yf[:N//2]))
plt.xlabel('Frecuency(KHZ)')
plt.ylabel('Voltage^2/Frecuecia(V^2/HZ)')
plt.title("PSD from origin signal")
plt.show()

MSO9104A_Infiniium.close()
rm.close()
```

8 ANEXO II: CODIGO BACKEND

1.1 Models.py

```
from django.db import models

class Article(models.Model):
    author = models.ForeignKey('auth.User', on_delete = models.CASCADE,)
    title = models.CharField(max_length = 100)
    body = models.TextField()
    def __str__(self):
        return self.title

class Entradas(models.Model):
    title = models.CharField(max_length=100)
    canal = models.Integerield() #es la url con guiones
    voltaje = models.IntegerField()
    fMuestreo = models.IntegerField()

    def __str__(self):
        return self.title

class Resultados(models.Model):
    entrada = models.ForeignKey(Entradas, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)
    ejeX = models.CharField(max_length=300)
    ejeY = models.CharField(max_length=300)

    def __str__(self):
        return self.title
```

1.2 Views.py

```
import json
from django.shortcuts import render
from .models import Entradas, Resultados
from .serializers import EntradasSerializer, ResultadosSerializer
from django.http import HttpResponse, JsonResponse
from rest_framework.parsers import JSONParser
from rest_framework import status, permissions
from urllib.parse import unquote
from rest_framework.decorators import api_view, permission_classes
from django.views.decorators.csrf import csrf_exempt
import pyvisa
from struct import unpack
import matplotlib.pyplot as plot
from datetime import datetime
import time
from scipy import signal
import numpy as np
import scipy.fftpack


# Create your views here.
@csrf_exempt
def ifft_list(request, format=None):
    """
    List all code snippets, or create a new snippet.
    """
    if request.method == 'GET':
        print("LOG: mandamos el listado de todas las entradas de BD")
        entradas = Entradas.objects.all()
        serializer = EntradasSerializer(entradas, many=True)
        return JsonResponse(serializer.data, safe=False)

    elif request.method == 'POST':
        print("LOG: creamos una nueva entrada en BD")
        print("===================")
```

```
        data = JSONParser().parse(request)
        print(data)
        print("===================")
        serializer = EntradasSerializer(data=data)
        if serializer.is_valid():
            return JsonResponse(serializer.data, status=201)
        print("Respuesta no valida... malformada")
        return JsonResponse(serializer.errors, status=400)



@csrf_exempt
def ifft_add(request, format=None):
    """
    List all code snippets, or create a new snippet.
    """
    if request.method == 'GET':
        print("LOG: mandamos el listado de todas las entradas de BD")
        entradas = Entradas.objects.all()
        serializer = EntradasSerializer(entradas, many=True)
        return JsonResponse(serializer.data, safe=False)

    elif request.method == 'POST':
        print("LOG: creamos una nueva entrada en BD")
        print("===================")
        data = JSONParser().parse(request)
        print(data)
        print("===================")
        serializer = EntradasSerializer(data=data)
        if serializer.is_valid():
            entrada = serializer.save()
            print("A continuacion seteamos los valores en el osciloscopio")
            reconfigured = reconfigureOsciloscope(entrada)
            return JsonResponse(reconfigured, status=201)
        print("Respuesta no valida... malformada")
        return JsonResponse(serializer.errors, status=400)



def reconfigureOsciloscope(entrada):
```

```
    print("reconfiguramos el osciloscopio y mandamos la senyal a Angular")
    rm = pyvisa.ResourceManager()
    canal = entrada.canal
    voltaje = entrada.voltaje
    tiempo = entrada.tiempo
    fMuestreo = entrada.fMuestreo

    if canal == 'CANAL 1':
        canal == 'CNAN1'
    elif canal == 'CANAL 2':
        canal == 'CNAN2'
    elif canal == 'CANAL 3':
        canal == 'CNAN3'
    else:
        canal == 'CNAN1'

    MSO9104A_Infiniium = rm.open_resource('TCPIP0::192.168.0.10::inst0::INSTR')
    identificationString = MSO9104A_Infiniium.query('*IDN?')
    state = MSO9104A_Infiniium.query(':STAT? %s' % canal)
MSO9104A_Infiniium.write(':CHANnel%d:DISPlay %d' % (CHANnel_N_, canal))
MSO9104A_Infiniium.write(':CHANnel%d:RANGe %G' % (CHANnel_N_, voltaje))
MSO9104A_Infiniium.write(':CHANnel%d:SCALe %G' % (CHANnel_N_, fMuestreo))
MSO9104A_Infiniium.write(':TIMebase:SCALe %G' % (tiempo))

    print("======State=====")
    print(state)
    print("==============")

    MSO9104A_Infiniium.timeout = 20000
    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:FREQuency?')
    frequency1 = temp_values[0]
    result_state = int(temp_values[0])
    print("============")
    print(frequency1)
    print("============")

    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIN?')
    originMinValue = temp_values[0]
```

```
result_state1 = int(temp_values[0])
print("============")
print(originMinValue)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMAX?')
originMaxValue = temp_values[0]
result_state2 = int(temp_values[0])
print("============")
print(originMaxValue)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VPP?')
originVPP = temp_values[0]
result_state3 = int(temp_values[0])
print("============")
print(originVPP)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:PERiod?')
period = temp_values[0]
result_state4 = int(temp_values[0])
print("============")
print(period)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIDdle?')
originOffset = temp_values[0]
result_state5 = int(temp_values[0])
print("============")
print(originOffset)
print("============")

# Sampling rate 1000 hz / second
# es el eje X muestreado
t = np.linspace(-2.5, 2.5, int(frequency1), endpoint=True)

# Plot the square wave signal
```

```
    print("============")
    print(t)
    print("============")
    # esto es el eje x
    amplitude = originVPP / 2  # Vpp/2
    offset = originOffset  # Offset
    waveform = signal.square(2 * np.pi * int(frequency1) * t)
    waveform = amplitude * waveform + offset
    print("============")
    print(waveform)
    print("============")
    # esto es el eje y

    data = {
        'ejex': np.array(t).tolist(),
        'ejey': np.array(waveform).tolist()
    }

    MSO9104A_Infiniium.close()
    rm.close()
    return data

@csrf_exempt
def ifft_detail(request, pk, format=None):
    """
    Retrieve, update or delete a code snippet.
    """
    try:
        entrada = Entradas.objects.get(id=pk)
    except Entradas.DoesNotExist:
        return HttpResponse(status=404)

    if request.method == 'GET':
        serializer = EntradasSerializer(entrada)
        return JsonResponse(serializer.data)

    elif request.method == 'PUT':
        data = JSONParser().parse(request)
```

```
        serializer = EntradasSerializer(entrada, data=data)
        if serializer.is_valid():
            serializer.save()
            return JsonResponse(serializer.data)
        return JsonResponse(serializer.errors, status=400)

    elif request.method == 'DELETE':
        entrada.delete()
        return HttpResponse(status=204)



@csrf_exempt
def ifft_edit(request, pk, format=None):
    """
    Update a code snippet.
    """
    try:
        entrada = Entradas.objects.get(id=pk)
    except Entradas.DoesNotExist:
        return HttpResponse(status=404)

    data = JSONParser().parse(request)
    serializer = EntradasSerializer(entrada, data=data)
    if serializer.is_valid():
        serializer.save()
        return JsonResponse(serializer.data)
    return JsonResponse(serializer.errors, status=400)



@csrf_exempt
def ifft_delete(request, pk, format=None):
    """
    Delete a code snippet
    :param request:
    :param pk:
    :param format:
    :return: 201 if enterada is deleted
    """
```

```
    try:
        entrada = Entradas.objects.get(id=pk)
    except Entradas.DoesNotExist:
        return HttpResponse(status=404)

    entrada.delete()
    return HttpResponse(status=201)



@csrf_exempt
def ifft_origin(request, pk):
    entrada = Entradas.objects.get(id=pk)
    print(entrada)
    if request.method == 'GET':
        print("LOG: mandamos el resultado en funcion de unas entradas")
        resultado = Resultados.objects.filter(entrada=entrada)
        print(resultado)
        serializer = ResultadosSerializer(resultado, many=True)
        return JsonResponse(serializer.data, safe=False)
    return HttpResponse(status=204)



@csrf_exempt
def ifft_all_entries(request):
    if request.method == 'GET':
        print("LOG: mandamos el listado de todas las entradas de BD")
        entradas = Entradas.objects.all()
        serializer = EntradasSerializer(entradas, many=True)
        return JsonResponse(serializer.data, safe=False)



@csrf_exempt
def init_visa(request):
    """
    Get Signal origin from osciloscope
    :param request:
    :return:
    """
```

```
# Obtains measure from origin square signal
rm = pyvisa.ResourceManager()
MSO9104A_Infiniium = rm.open_resource('TCPIP0::192.168.0.10::inst0::INSTR')
identificationString = MSO9104A_Infiniium.query('*IDN?')
state = MSO9104A_Infiniium.query(':STAT? %s' % ('CHAN1'))
MSO9104A_Infiniium.timeout = 20000
temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:FREQuency?')
frequency1 = temp_values[0]
result_state = int(temp_values[0])
print("====Frecuency========")
print(frequency1)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIN?')
originMinValue = temp_values[0]
result_state1 = int(temp_values[0])
print("====MinValue========")
print(originMinValue)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMAX?')
originMaxValue = temp_values[0]
result_state2 = int(temp_values[0])
print("=====MaxValue=======")
print(originMaxValue)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VPP?')
originVPP = temp_values[0]
result_state3 = int(temp_values[0])
print("=====VPP=======")
print(originVPP)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:PERiod?')
period = temp_values[0]
result_state4 = int(temp_values[0])
print("=====Period=======")
```

```
print(period)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIDdle?')
originOffset = temp_values[0]
result_state5 = int(temp_values[0])
print("=====Offset=======")
print(originOffset)
print("============")

# Sampling rate 1000 hz / second
# es el eje X muestreado
t = np.linspace(-2.5, 2.5, int(frequency1), endpoint=True)

# Plot the square wave signal
print("=====Time (EJE X)=======")
print(t)
print("============")
# esto es el eje x
amplitude = originVPP / 2  # Vpp/2
offset = originOffset  # Offset
waveform = signal.square(2 * np.pi * int(frequency1) * t)
waveform = amplitude * waveform + offset
print("====VOLTAJE (EJE Y)========")
print(waveform)
print("============")
# esto es el eje y

data = {
    'ejex': np.array(t).tolist(),
    'ejey': np.array(waveform).tolist()
}

MSO9104A_Infiniium.close()
rm.close()
return JsonResponse(data)
```

```
@csrf_exempt
def send_ifft(request):
    """
    Get IFFT from Signal origin to osciloscope
    :param request:
    :return:
    """
    if request.method == 'POST' or request.method == 'GET':
        # Obtains measure from origin square signal
        rm = pyvisa.ResourceManager()
        MSO9104A_Infiniium = rm.open_resource('TCPIP0::192.168.0.10::inst0::INSTR')
        identificationString = MSO9104A_Infiniium.query('*IDN?')
        state = MSO9104A_Infiniium.query(':STAT? %s' % ('CHAN1'))
        MSO9104A_Infiniium.timeout = 20000
        temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:FREQuency?')
        frequency1 = temp_values[0]
        result_state = int(temp_values[0])
        print("============")
        print(frequency1)
        print("============")

        temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIN?')
        originMinValue = temp_values[0]
        result_state1 = int(temp_values[0])
        print("============")
        print(originMinValue)
        print("============")

        temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMAX?')
        originMaxValue = temp_values[0]
        result_state2 = int(temp_values[0])
        print("============")
        print(originMaxValue)
        print("============")

        temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VPP?')
        originVPP = temp_values[0]
        result_state3 = int(temp_values[0])
```

```
print("============")
print(originVPP)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:PERiod?')
period = temp_values[0]
result_state4 = int(temp_values[0])
print("============")
print(period)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIDdle?')
originOffset = temp_values[0]
result_state5 = int(temp_values[0])
print("============")
print(originOffset)
print("============")

# Sampling rate 1000 hz / second
# es el eje X muestreado
t = np.linspace(-2.5, 2.5, int(frequency1), endpoint=True)

# Plot the square wave signal
print("============")
# esto es el eje x
print(t)
print("============")
amplitude = originVPP / 2  # Vpp/2
offset = originOffset  # Offset
waveform = signal.square(2 * np.pi * int(frequency1) * t)
waveform = amplitude * waveform + offset
print("============")
# esto es el eje y
print(waveform)
print("============")

# Number of samplepoints (se sustituye por lo que recibimos por parametros)
N = 600
```

```
        # sample spacing
        T = 1.0 / 800.0

        x = t
        y = waveform

        # x = np.linspace(0.0, N*T, N)
        # y = np.sin(50.0 * 2.0*np.pi*x) + 0.5*np.sin(80.0 * 2.0*np.pi*x)
        yf = scipy.fftpack.fft(y)
        xf = np.linspace(0.0, 1.0 / (2.0 * T), N // 2)

        # fig, ax = plt.subplots()
        # ax.plot(xf, 2.0 / N * np.abs(yf[:N // 2]))
        # plt.title("FFT from origin signal")
        # plt.show()

        data = {
            'ejex': np.array(xf).tolist(),
            'ejey': np.round(np.array(2.0 / N * np.abs(yf[:N // 2])), 14).tolist()
        }

        # iifty = np.fft.ifft(yf);
        # fig2, ax2 = plt.subplots()
        # ax2.plot(xf, 2.0 / N * np.abs(iifty[:N // 2]))
        # plt.title("IFFT from origin signal")
        # plt.show()

        #data2 = {
        #    'ejex': np.array(xf).tolist(),
        #    'ejeY': np.array(2.0 / N * np.abs(iifty[:N // 2])).tolist()
        #}

        MSO9104A_Infiniium.close()
        rm.close()
        return JsonResponse(data)
    return HttpResponse(status=404)

@csrf_exempt
```

```
def send_psd(request):
    """
    Get IFFT from Signal origin to osciloscope
    :param request:
    :return:
    """
    if request.method == 'POST' or request.method == 'GET':
        # Obtains measure from origin square signal
        rm = pyvisa.ResourceManager()
        MSO9104A_Infiniium = rm.open_resource('TCPIP0::192.168.0.10::inst0::INSTR')
        identificationString = MSO9104A_Infiniium.query('*IDN?')
        state = MSO9104A_Infiniium.query(':STAT? %s' % ('CHAN1'))
        MSO9104A_Infiniium.timeout = 20000
        temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:FREQuency?')
        frequency1 = temp_values[0]
        result_state = int(temp_values[0])
        print("============")
        print(frequency1)
        print("============")

        temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIN?')
        originMinValue = temp_values[0]
        result_state1 = int(temp_values[0])
        print("============")
        print(originMinValue)
        print("============")

        temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMAX?')
        originMaxValue = temp_values[0]
        result_state2 = int(temp_values[0])
        print("============")
        print(originMaxValue)
        print("============")

        temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VPP?')
        originVPP = temp_values[0]
        result_state3 = int(temp_values[0])
        print("============")
```

```
print(originVPP)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:PERiod?')
period = temp_values[0]
result_state4 = int(temp_values[0])
print("============")
print(period)
print("============")

temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIDdle?')
originOffset = temp_values[0]
result_state5 = int(temp_values[0])
print("============")
print(originOffset)
print("============")

# Sampling rate 1000 hz / second
# es el eje X muestreado
t = np.linspace(-2.5, 2.5, int(frequency1), endpoint=True)

# Plot the square wave signal
print("============")
# esto es el eje x
print(t)
print("============")
amplitude = originVPP / 2  # Vpp/2
offset = originOffset  # Offset
waveform = signal.square(2 * np.pi * int(frequency1) * t)
waveform = amplitude * waveform + offset
print("============")
# esto es el eje y
print(waveform)
print("============")

# Number of samplepoints (se sustituye por lo que recibimos por parametros)
N = 600
# sample spacing
```

```
T = 1.0 / 800.0

x = t
y = waveform

# x = np.linspace(0.0, N*T, N)
# y = np.sin(50.0 * 2.0*np.pi*x) + 0.5*np.sin(80.0 * 2.0*np.pi*x)
yf = scipy.fftpack.fft(y)
xf = np.linspace(0.0, 1.0 / (2.0 * T), N // 2)

# fig, ax = plt.subplots()
# ax.plot(xf, 2.0 / N * np.abs(yf[:N // 2]))
# plt.title("FFT from origin signal")
# plt.show()

#Generate PSD from signal
fs = 250
t = np.arange(0, 10, 1/fs)
xf = np.linspace(0, fs, len(t))
yf = abs(yf)**2

data = {
    'ejex': np.round(np.array(xf), 5).tolist(),
    'ejey': np.round(np.array(2.0 / N * np.abs(yf[:N // 2])), 5).tolist()
}

# iifty = np.fft.ifft(yf);
# fig2, ax2 = plt.subplots()
# ax2.plot(xf, 2.0 / N * np.abs(iifty[:N // 2]))
# plt.title("IFFT from origin signal")
# plt.show()

#data2 = {
#    'ejex': np.array(xf).tolist(),
#    'ejeY': np.array(2.0 / N * np.abs(iifty[:N // 2])).tolist()
#}

MSO9104A_Infiniium.close()
```

```
        rm.close()
        return JsonResponse(data)
    return HttpResponse(status=404)



@csrf_exempt
def send_parameters(request):
    """
    Get IFFT from Signal origin to osciloscope
    :param request:
    :return:
    """
    if request.method == 'POST' or request.method == 'GET':
        # Obtains measure from origin square signal
        rm = pyvisa.ResourceManager()
        MSO9104A_Infiniium = rm.open_resource('TCPIP0::192.168.0.10::inst0::INSTR')
        identificationString = MSO9104A_Infiniium.query('*IDN?')
        state = MSO9104A_Infiniium.query(':STAT? %s' % ('CHAN1'))
        MSO9104A_Infiniium.timeout = 20000
        data = JSONParser().parse(request)
        print(data)

        isAmplitud = data.get("amplitud")
        isFrecuencia = data.get("frecuencia")
        isPeriodo = data.get("periodo")
        isVmax = data.get("tensionMaxima")
        isVmin = data.get("tensionMinima")

        print(isAmplitud)
        print(isFrecuencia)
        print(isPeriodo)
        print(isVmax)

        originVpp = "" #amplitud
        frecuency1 = ""
        originMinValue = ""
        originMaxValue = ""
        period = ""
```

```
CHANnel_N_ = 1
if isFrecuencia:
    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:FREQuency?')
    frecuency1 = temp_values[0]
    result_state = int(temp_values[0])
    print("====Frecuencia========")
    print(frecuency1)
    print("============")

if isVmin:
    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIN?')
    originMinValue = temp_values[0]
    result_state1 = int(temp_values[0])
    print("====Value min========")
    print(originMinValue)
    print("============")

if isVmax:
    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMAX?')
    originMaxValue = temp_values[0]
    result_state2 = int(temp_values[0])
    print("=====Value max=======")
    print(originMaxValue)
    print("============")

if isAmplitud:
    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VPP?')
    originVpp = temp_values[0]
    result_state3 = int(temp_values[0])
    print("=====Amplitud=======")
    print(originVpp)
    print("============")

if isPeriodo:
    temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:PERiod?')
    period = temp_values[0]
    result_state4 = int(temp_values[0])
```

```
            print("=====Periodo=======")
            print(period)
            print("============")

        temp_values = MSO9104A_Infiniium.query_ascii_values(':MEAS:VMIDdle?')
        originOffset = temp_values[0]
        result_state5 = int(temp_values[0])
        print("=====Offset=======")
        print(originOffset)
        print("============")

        temp_values = MSO9104A_Infiniium.query_ascii_values(':CHANnel%d:DISPlay:SCALe?' %
(CHANnel_N_))
        scale = temp_values[0]
        print("=====SCALE=======")
        print(scale)
        print("============")

        data = {
            'amplitud': originVpp,
            'frecuencia': frecuency1,
            'periodo': period,
            'tensionMaxima': originMaxValue,
            'tensionMinima': originMinValue
        }

        MSO9104A_Infiniium.close()
        rm.close()
        return JsonResponse(data)
    return HttpResponse(status=404)
```

1.3 Urls.py

```
from django.shortcuts import render
from django.conf.urls import url
from rest_framework.urlpatterns import format_suffix_patterns
```

```
from . import views


# Create your views here.
urlpatterns = [
    url(r'^getOrigenSignal', views.init_visa),
    url(r'^signalConfiguration/(?P<pk>\d+)/edit', views.ifft_edit),
    url(r'^signalConfiguration/(?P<pk>\d+)/delete', views.ifft_delete),
    url(r'^signalConfiguration/(?P<pk>\d+)/', views.ifft_detail),
    url(r'^signalConfiguration$', views.ifft_add),
    url(r'^originSignal/(?P<pk>\d+)/', views.ifft_origin),
    url(r'^signalConfiguration/all', views.ifft_all_entries),
    url(r'^fft', views.send_ifft),
    url(r'^potencia', views.send_psd),
    url(r'^signalCalculateParams', views.send_parameters)
]
urlpatterns = format_suffix_patterns(urlpatterns)
```

1.4 Serializers.py

```
from rest_framework import serializers
from IFFTSignals.models import Entradas, Resultados



class EntradasSerializer(serializers.ModelSerializer):
    class Meta:
        model = Entradas
        fields = ('id',
                  'canal',
                  'voltaje',
                  'trigger',
                  'fMuestreo')

    def create(self, validated_data):
        """
        Create and return a new `Entrada` instance, given the validated data.
        """
```

```
        return Entradas.objects.create(**validated_data)

    def update(self, instance, validated_data):
        """
        Update and return an existing `Entrada` instance, given the validated data.
        """
        instance.title = validated_data.get('title', instance.title)
        instance.canal = validated_data.get('canal', instance.canal)
        instance.voltaje = validated_data.get('voltaje', instance.voltaje)
        instance.tiempo = validated_data.get('tiempo', instance.tiempo)
        instance.fMuestreo = validated_data.get('fMuestreo', instance.fMuestreo)
        instance.trigger = validated_data.get('trigger', instance.trigger)
        instance.save()
        return instance

class ResultadosSerializer(serializers.ModelSerializer):
    class Meta:
        model = Resultados
        fields = ('id',
                  'entrada',
                  'ejeX',
                  'ejeY')
```

9 ANEXO III: OTROS CÓDIGOS

9.1 CommandsKeysigth.ipsx

```
# Obtains measure from origin square signal
(Connect "MSO9104A Infiniium", "TCPIP0::192.168.0.10::inst0::INSTR", "Infiniium 9000A Series Oscilloscopes / 03.50.0011")
*IDN?
:STATus? CHANNEL1
(SET DefaultTimeout to 10000)
<frequency1>, = :MEASure:FREQuency?
<originMinValue>, = :MEASure:VMIN?
<offset>, = :MEASure:VMIDdle?
<originMaxValue>, = :MEASure:VMAX?
<originVPP>, = :MEASure:VPP?
<period>, = :MEASure:PERiod?
:CHANnel<CHANnel<N>>:DISPlay 1
:CHANnel<CHANnel<N>>:RANGe 0.5
:CHANnel<CHANnel<N>>:OFFSet 0.008
:CHANnel<CHANnel<N>>:SCALe 0.3
:FUNCtion<FUNCtion<N>>:DISPlay 1
:TIMebase:SCALe 0.0005
:WAVeform:XRANge?
:WAVeform:XDISplay?
:WAVeform:YORigin?
:WAVeform:COMPlete?
:CHANnel<CHANnel<N>>:DISPlay:RANGe?
:CHANnel<CHANnel<N>>:DISPlay:SCALe?
```