UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
PROGRAMA DE POSGRADO EN ASTROFÍSICA
ASTRONOMÍA

Implementación de un espectrómetro de alta resolución

TESIS
QUE PARA OPTAR POR EL GRADO DE:
Maestría en Ciencias (Astronomía)

PRESENTA:
Daniel Jacobo Díaz González

TUTOR PRINCIPAL
Dr. Stanley Eugene Kurtz Smith
Instituto de Radiastronomía y Astrofísica

COMITÉ TUTOR
Dr. Stanley Eugene Kurtz Smith
Instituto de Radiastronomía y Astrofísica
Dra. Sarah Jane Arthur Chadwick
Instituto de Radiastronomía y Astrofísica
Dr. Carlos Carrasco González
Instituto de Radiastronomía y Astrofísica
Dr. Roberto Galván Madrid
Instituto de Radiastronomía y Astrofísica

MORELIA, MICHOACÁN, JUNIO, 2019

A mis profesores en la ETSII, en la ULL y en el IRyA por mi formación.
Al Consejo de Ciencia y Tecnología (CONACYT) por los apoyos prestados.
A Stan por el apoyo y el tiempo dedicados.
A Zeben y David, por cargar con parte de lo que me tocaba.
Y especialmente a Omaira, por una paciencia infinita.
Yo.

Declaración de autenticidad

Por la presente declaro que, salvo cuando se haga referencia específica al trabajo de otras personas, el contenido de esta tesis es original y no se ha presentado total o parcialmente para su consideración para cualquier otro título o grado en esta o cualquier otra Universidad. Esta tesis es resultado de mi propio trabajo y no incluye nada que sea el resultado de algún trabajo realizado en colaboración, salvo que se indique específicamente en el texto.

Daniel Jacobo Díaz González. Morelia, Michoacán, 2019

Resumen

La espectroscopía nos ofrece información acerca de los niveles energéticos en átomos y moléculas, permitiendo, en el caso de la Astronomía, la detección y caracterización de nubes moleculares y atómicas. En este trabajo presentamos un nuevo espectrómetro de alta resolución que permitirá la obtención de resoluciones de hasta 15 kHz sobre un ancho de banda de 250MHz.

Este espectrómetro está diseñado para su integración con frontends como los existentes en el Gran Telescopio Milimétrico Alfonso Serrano (SEQUOIA y RSR), pero también podrá ser integrado fácilmente en otros radiotelescopios, siendo un instrumento portátil y de fácil configuración.

Así mismo, en este trabajo se hace entrega del código de una pipeline para el postprocesado de los datos volcados por el espectrómetro, que también estará próximamente disponible en Internet a través de un repositorio GIT.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Objetivo

Esta tesis de maestría presenta el diseño final de un espectrómetro de alta resolución (el objetivo que nos hemos fijado es obtener una resolución espectral de 15 kHz) que se podrá integrar como backend de radiostelescopios de un único plato tales como el Gran Telescopio Milimétrico Alfonso Serrando (fig. 1.1), los radiotelescopios existentes en Guanajuato o las dos antenas instaladas en Tulatcingo y que actualmente están en proceso de ser reconvertidas a radiotelescopios.

Un espectrómetro es un dispositivo que toma una señal en el dominio del tiempo y la convierte al dominio de la frecuencia, pudiendo visualizar la energía contenida en cada una de las frecuencias por las que está compuesta dicha señal. Existen diversos métodos para implemetar estos espectrómetros, tal y como comentaremos en la sección 1.3, pero nosotros nos centraremos en los espectrómetros digitales. En estos sistemas digitales (como el que nos ha ocupado a lo largo de este proyecto), esta conversión se realiza generalmente utilizando la Transformada Rápida de Fourier (FFT por sus siglas en inglés, Fast Fourier Transform).

1.2. Radiotelescopios

El primer radiotelescopio fue construido en 1937 por Grote Reber, en Chicago; este radiotelescopio artesanal era una antena parabólica de 9 metros de diámetro, y actualmente se puede visitar en el Observatorio de Green Bank, en el estado de Virginia Occidental (National Science Foundations 2018). Desde ese entonces hasta ahora la evolución de esta

herramienta ha sido espectacular, llegando a convertirse en fundamental para la historia de la astronomía. Sin embargo, los bloques funcionales que componen un radiotelescopio apenas han sufrido cambios desde entonces, pudiéndose dividir, a grosso modo, en tres fundamentales: antena, front-end y back-end. A continuación haremos una breve discusión acerca de cada uno de ellos.

1.2.1. Antena

La función primordial de la antena es la de captar las ondas en el espacio libre y convertirlas a ondas guiadas, pudiendo también en algunos casos separar la polarización de estas ondas.

Es importante destacar que, si bien la tendencia en radioastronomía es la de utilizar grandes arreglos de antenas (VLA, ALMA, SMA, por ejemplo) que pueden generar imágenes del cielo usando la técnica de interferometría de radio o síntesis de apertura, las antenas de plato único continuan siendo una herramienta de gran valor para los astrónomos, sobre todo en aquellos casos en los que no son necesarias imágenes y siendo prácticamente indispensable para los casos científicos en los que las estructuras de gran escala son relevantes. En este último caso podemos utilizar los datos de las antenas de plato único para complementar las obvervaciones interferométricas, aprovechando la mayor sensitividad a la temperatura de brillo o empleándolos para resolver el problema del "zero-spacing", la zona ciega que resulta cuando utilizamos los radiointerferómetros.

Algunas de las ventajas de las antenas de plato único frente a un interferómetro son las siguientes.

- **Sensitividad**. En general la sensitividad de una antena está directamente relacionada con la superficie colectora de la misma. Algunas antenas de plato único tienen una mayor superficie colectora que las que componen los interferómetros, por lo que la sensitividad a fuentes débiles es mayor.
- **Grandes estructuras**. Las antenas de plato único son más adecuadas para el estudio de las grandes estructuras, cuya temperatura de brillo es baja.
- **Instrumentación**. La instrumentación especializada (especialmente los receptores) es más susceptible de ser instalada en las antenas de plato único, ya que hay que construir un único instrumento.
- **Costes**. La fabricación de antenas de plato único y su instrumentación es, por lo general, menos costosa en tiempo y recursos que la fabricación de interferómetros y sus instrumentos.

Figura 1.1: Gran Telescopio Milimétrico Alfonso Serrano

1.2.2. Front-end

Una vez la señal es captada por nuestra antena, pasa al front-end. Este está compuesto por los primeros dispositivos analógicos por los que atraviesa la señal, los cuales tienen como función amplificar, filtrar y convertir la señal a una frecuencia intermedia.

Un detalle a tener en cuenta es que la potencia captada por una antena es aproximadamente

$$P = S \times A \times \Delta f \tag{1.1}$$

donde S es la densidad de flujo que llega a la tierra, A es el área de la antena y Δf es el rango de frecuencias en el que se mide la radiación. Tomando la ecuación 1.1, y sabiendo que el flujo típico de una señal astronómica se mide en Janskys (Jy) y que un Jansky es equivalente a $10^{-26}\mathrm{W}\times\mathrm{m}^{-2}\times\ \mathrm{Hz}^{-1}$, podemos concluir que una señal astronómica tendrá del orden de $10^{-26}W$, lo que supone unos -230 dBm; para contextualizar estos indicadores de potencia podemos comentar que para que una señal WI-Fi se considere idónea, su potencia debe ser de -40 dBm, mientras que una potencia de -70 dBm se considera baja y a partir de los -80 dBm nos encontramos con una señal de muy baja calidad, pudiendo producir cortes en la conexión. Teniendo en cuenta las bajas potencias que hemos dicho que caracterizan a las señales astronómicas, es necesario amplificarlas para su posterior tratamiento. Para ello utilizamos amplificadores de bajo ruido (o LNA por sus siglas en inglés, Low Noise Amplifier), que son capaces de amplificar la señal recibida, agregando muy poco ruido (generalmente utilizando enfriamiento criogénico).

Una vez la señal ha sido amplificada, se convierte a una frecuencia intermedia (IF por sus siglas en inglés, Intermediate Frequency); de esta forma se simplifica el diseño y fabricación de los dispositivos sintonizados a partir de esta etapa, con el consiguiente ahorro en costos de fabricación. Esto es debido a que dichos dispositivos estarán diseñados para trabajar siempre a dicha IF, sea cual sea la señal de entrada. Si no realizáramos esta conversión, los dispositivos sintonizados tendrían que ser diseñados para trabajar en un rango muy amplio de frecuencias, con el consiguiente incremento en complejidad y costos. Otra ventaja asociada a esta conversión es que por lo general las frecuencias elegidas para la IF presentan un mejor comportamiento frente a las atenuaciones que conlleva el proceso de transmisión, presentando menores pérdidas. Esta conversión se realiza mediante la utilización de un receptor heterodino, cuyo esquema general podemos ver en la fig. 1.2.

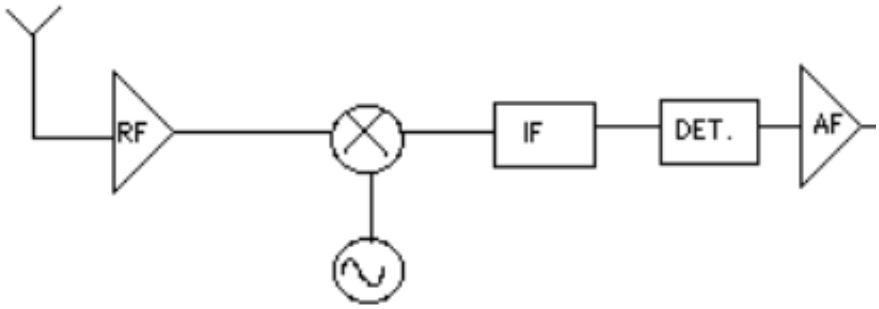

Figura 1.2: Receptor Heterodino (Wikipedia.org 2018b)

1.2.3. Back-end

Tras pasar por el front-end y ser convertida a la IF, la señal llega al back-end, siendo dicho back-end la parte del procesado en la que se centra nuestro proyecto. El objetivo del back-end no es otro que procesar la señal obtenida, obteniendo datos que después puedan ser interpretados por los astrónomos. Generalmente podemos clasificar estos back-end en tres tipos:

- **De continuo.** Este tipo de back-end estima la potencia recibida en toda la banda de paso del receptor.
- **Espectral.** En este caso, el back-end estima la densidad espectral de potencia de la señal de IF dividiendo el ancho de banda del receptor en canales adyacentes y midiendo la potencia en cada tramo.

- **Púlsares.** Por último los back-end diseñados para la búsqueda de púlsares están optimizados para tener una muy alta resolución temporal, siendo capaces de dispersar la señal en tiempo real.

Hoy por hoy cabe destacar que estos tres back-ends se pueden integrar en un único back-end capaz de cumplir los requerimientos necesarios para los tres tipos de procesado.

En el caso que ocupa esta tesis, tal y como comentamos anteriormente, hemos implementado un back-end de tipo **espectral** mediante un espectrómetro digital de alta resolución.

1.3. Espectrómetros

En astronomía, el espectrómetro es uno de los instrumentos más utilizados en las distintas regiones del espectro electromagnético (por ejemplo, radio, infrarrojo, óptico, rayos X), ya que permite obtener gran cantidad de información acerca de las distintas fuentes estudiadas, tales como velocidad, temperatura, densidad o composición química.

El cometido principal de un espectrómetro es medir la densidad espectral de energía de una señal, correspondiendo ésta a la distribución de la potencia de dicha señal sobre las frecuencias por las que está formada. Para ilustrar esto, presentamos la figura 1.3:

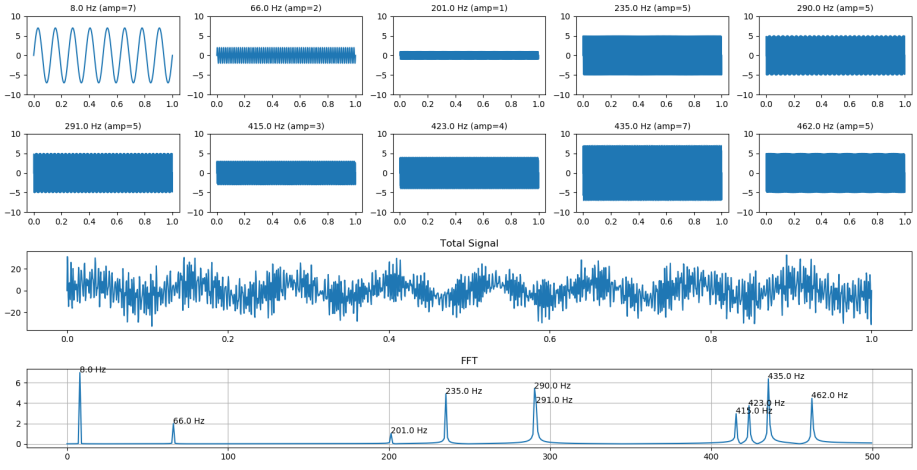

Figura 1.3: Dominio en el tiempo vs Dominio en la frecuencia

En la figura 1.3 podemos ver en las dos primeras filas un total de diez señales sinusoidales, cada una a una frecuencia dada y con una amplitud determinada (estos datos figuran sobre cada una de las gráficas); en la tercera fila vemos la señal resultado de sumar las 10 anteriories. Esta señal suma (Total Signal) es la que entra en el espectrómetro, cuya salida es el espectro que aparece en la cuarta fila, etiquetada como FFT.

Los parámetros básicos que caracterizan un espectrómetro son el ancho de banda, la resolución espectral, la resolución temporal, la sensitividad, el rango dinámico, la linealidad y la estabilidad; a continuación explicaremos brevemente cada uno de ellos.

El ancho de banda corresponde al rango de frecuencias que procesa el espectrómetro; en el caso específico de los espectrómetros digietales este ancho de banda determina la frecuencia de muestreo mínima que debe utilizar el sistema.

La resolución espectral hace referencia al Δf que es capaz de discriminar, y que viene determinado por el número de canales disponibles en el dominio de la frecuencia y el ancho de banda total que cubre, según la relación

$$\Delta f = \frac{BW}{NDC} \tag{1.2}$$

donde BW es *ancho de banda* (bandwidth) y NDC es el *número de canales*. Δf es, por tanto, una medida de la precisión que podemos alcanzar en la determinación de la frecuencia; cuanto más pequeño sea este Δf, mayor precisión. En el caso de la figura 1.3, por ejemplo, tenemos 500 puntos y 500 Hz de ancho de banda, lo que nos da una resolución espectral Δf de 1 Hz.

La resolución temporal se corresponde con la tasa a la cual el espectrómetro es capaz de obtener un espectro. En el caso de los espectrómetros digitales (caso que nos ocupa en este proyecto), esta resolución temporal es directamente proporcional a la cantidad de datos a procesar e inversamente proporcional a la potencia del hardware que utilizamos para dicho procesado.

La sensitividad se determina como la media cuadrática (RMS por sus siglas en inglés, Root Mean Square) del ruido por unidad de frecuencia. Con el fin de mejorar ese parámetro en el caso de los espectrómetros digitales el espectro que proporciona a los usuarios suele ser el resultado de promediar varios espectros, lo que conlleva una disminución en la resolución temporal, tal y como se puede deducir de la ecuación del radiómetro, de la que obtenemos que

$$\sigma \propto \frac{1}{\sqrt{\Delta\nu\Delta t}} \tag{1.3}$$

El rango dinámico hace referencia al margen que hay entre el nivel máximo y el ruido de fondo de un determinado sistema, medido en dB.

La linealidad es la propiedad que relaciona la cantidad de fotones recibidos con el voltaje producido por un sistema. Cuando se supera un número determinado de fotones, el sistema satura y no se continua incrementando el voltaje.

La estabilidad es la propiedad por la que un espectrómetro obtiene el mismo espectrómetro ante la misma señal.

A continuación haremos una breve descripción de los tipos de espectrómetros más habituales.

1.3.1. Banco de filtros

Los espectrómetros basados en los bancos de filtros fueron, históricamente hablando, los primeros en aparecer, siendo en su origen totalmente analógicos. Se basan en un concepto muy simple: la señal se hace pasar por una serie de filtros pasa-banda con la anchura que deseamos utilizar como resolución espectral y centrados en la frecuencia deseada, para a continuación medir la potencia de la señal filtrada. Aunque siguen siendo espectrómetros viables, utilizando en algunos casos filtros digitales, su poca flexibilidad los hacen poco atractivos, ya que el número de canales (N) y la resolución espectral (Δf) son siempre fijos.

1.3.2. Acústico-ópticos

Estos espectrómetros, que suelen ser complejos, costosos y no demasiado flexibles, están basados en la difracción producida por un cristal, efecto predicho por Brillouin en 1921. Una onda ultrasónica, modulada por la señal de interés, se propaga por el cristal, provocando cambios en su índice de refacción, n. El haz de un láser, cayendo sobre el cristal, sufre una difracción proporcional a estos cambios en n. Este haz modulado queda registrado en un CCD como patrones de difracción que después son analizados para obtener el espectro. Para más información, ver (Herrera-Martínez et al. 2009).

1.3.3. Autocorreladores

La densidad espectral de potencia es calculada usando el teorema de Wiener-Khinchin, que veremos más en detalle en la sección 2.1.7), y que relaciona la densidad espectral

de potencia de un proceso ergódigo aleatorio (la señal de interés) con su función de autocorrelación. El proceso que siguen estos espectrómetros es el siguiente:

1. **Muestreo** de la señal, cumpliendo con el teorema de Nyquist (ver sección 2.1.1).
2. **Cuantificación**, asignando un valor numérico a cada intervalo continuo de amplitudes (ver sección 2.1.2).
3. **Retardo y multiplicación** adecuada de las muestras, obteniendo de esta forma la función de autocorrelación.
4. Calcular la **Trasformada Rápida de Fourier** para obtener el espectro.

1.3.4. Espectrómetro de Transformada Rápida de Fourier

En los espectrómetros de Transformada Rápida de Fouerir se hace uso del teorema de Wiener-Khinchin, al igual que en los espectrómetros autocorreadores; la diferencia está en que en este caso la transformada de Fourier es calculada en tiempo real, por lo que se puede obtener el espectro y la densidad espectral de potencia sin necesidad de calcular la función de autocorrelación.

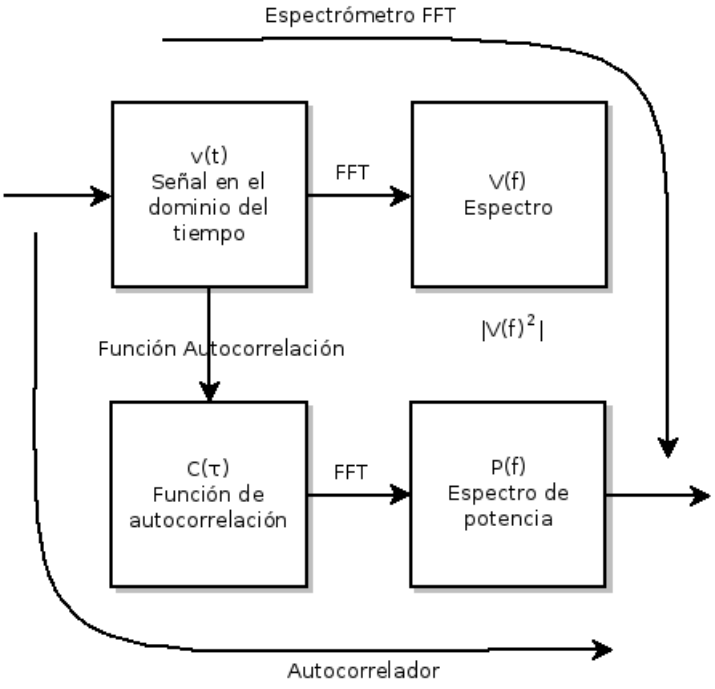

Figura 1.4: Teorema de Wiener-Khinchin

La figura 1.4 ilustra el teorema de Wiener-Khinchin, y en ella podemos apreciar esquemáticamente la diferencia entre un autocorrelador y un espectrómetro de transformada rápida de Fourier.

1.4. Líneas espectrales

Un espectrómetro, sea cual sea su tipo, tiene un objetivo principal, que no es otro que medir líneas espectrales; ¿qué son estas líneas espectrales? ¿por qué se producen ¿qué información podemos obtener de ellas?

Las líneas espectrales son el resultado de la interacción entre un sistema atómico o molecular y los fotones. Cuando un átomo o molécula se excita, pasa de un estado de energía determinado (normalmente el fundamental, conocido como estado base) a un estado más energético. Cuando este átomo o molécula vuelve al estado fundamental (o pasa a uno de menor energía), la partícula debe radiar en forma de fotones la energía sobrante que corresponde a la diferencia de energía entre los dos niveles. En las moléculas, además del cambio entre niveles de energía de los electrones, esta radiación se puede deber también a las rotaciones o vibraciones de las mismas; en cualquiera de los casos,

$$\Delta E = h\nu$$

La energía ΔE generada en cada uno de estos cambios corresponde a distintas zonas del espectro electromagnético (fig. 1.5): las ΔE producidas por el cambio entre estados electrónicos se puede detectar en el rango óptico o en el rango de radio (cuando hablamos de las RRL o Radio Recombination Lines); las ΔE debidas a las vibraciones moleculares, en el rango del infrarojo; y las ΔE que se originan en las rotaciones, en el rango de radio.

A continuación presentamos la tabla 1.1 en la que muestra un resumen con algunas de las fuentes típicas de radiación y su rango de emisión (Lumen Learning 2016):

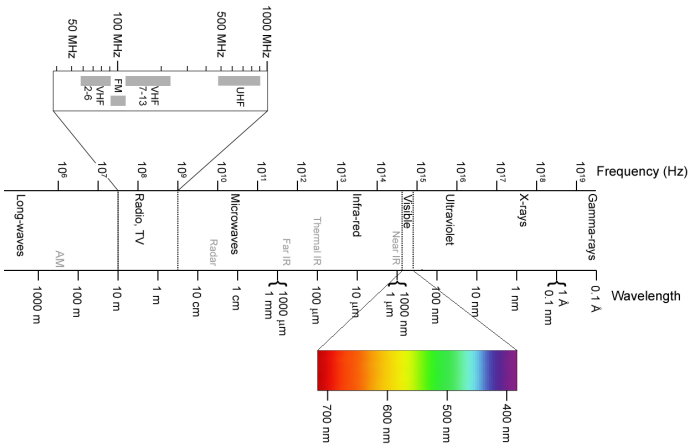

Figura 1.5: Espectro electromagnético

Radiación	**Wavelength (nm)**	**Temperatura (K)**	**Fuentes típicas**
Rayos Gamma	$< 10^{-2}$	10^{8}	Reacciones nucleares; procesos de muy alta energía
Rayos X	$10^{-2} - 20$	$10^{6} - 10^{8}$	Gas en clusters de galaxias, remanentes de supernova, corona solar
Ultravioleta	$20 - 400$	$10^{4} - 10^{6}$	Remanentes de supernova, estrellas muy calientes
Visible	$400 - 700$	$10^{3} - 10^{4}$	Estrellas
Infrarojo	$10^{3} - 10^{6}$	$10^{1} - 10^{3}$	Nubes frías de polvo y gas, planetas
Microondas	$10^{6} - 10^{9}$	< 10	Galaxias activas, púlsares, radiación de fondo cósmico
Radio	$> 10^{9}$	< 10	Remanentes de supernovas, púlsares, gas frío

Cuadro 1.1: Emisión de radiación

1.4.1. Intensidad y flujo

Un campo de radiación es descrito por la intensidad de radiación, denominada como intensidad específica o brillo. Dicha intensidad es función de la posición, de la dirección y del tiempo, pudiendo escribirse como $I_f(\vec{r}, \vec{k}, t)$, y se define como la energía por unidad de tiempo que pasa por un área unidad perpendicular a la dirección dada por el véctor $\vec{k}$, centrada en la posición $\vec{r}$, transportada por la radiación que se propaga dentro de un ángulo sólido unidad centrada en la dirección $\vec{k}$, en una unidad de intervalo de frecuencia,

$$dE = I_f \; dt \; dA \; \cos\theta \; d\Omega \; df \tag{1.4}$$

Si la intensidad procede de una fuente que subtiende un ángulo sólido $\Delta\Omega_S$, la intensidad media producida por la fuente es

$$J_f = \frac{1}{4\pi} \int I_f \Delta\Omega_S \tag{1.5}$$

La densidad de flujo (S_f) es el flujo de energía que atraviesa el área unidad por unidad de intervalo de frecuencia y por unidad de tiempo; esto implica que es la intensidad integrada para todas las direcciones, teniendo en cuenta el factor de proyección del área perpendicularmente a la dirección considerada, $\cos\theta$,

$$S_f = \int_{4\pi} I_f \; \cos\theta \; d\Omega. \tag{1.6}$$

En el caso de la astronomía θ es siempre muy pequeño, por lo que a efectos prácticos se suele tomar $\theta = 0$, por lo que el factor $\cos\theta$ queda como 1, obteniendo

$$S_f = \int_{\text{fuente}} I_f \; d\Omega \tag{1.7}$$

1.4.2. Perfiles de línea, ensanchamiento natural, colisional y Doppler

Dada una transición entre dos estados de distinta energía de una partícula, con una diferencia de energía ΔE y una frecuencia de los fotones emitidos o absorbidos $f_0 = \frac{\Delta E}{h}$, la línea observada resultante de dicha transición nunca se observará como una línea monocromática, infinitamente estrecha, sino que nos encontraremos con un perfil $\phi(f)$, al que llamaremos perfil de línea, normalizado de forma que

$$\int_{-\infty}^{+\infty} \phi(f) df = 1. \tag{1.8}$$

La función $\phi(f)$ es una función centrada en f_0, que se hace 0 rápidamente fuera de la frecuencia central. Su valor máximo es el inverso de la anchura equivalente de la línea, Δf_{eq}, puesto que

$$\phi(f_0)\Delta f_{eq} = \int_{-\infty}^{+\infty} \phi(f)df = 1. \tag{1.9}$$

Por otra parte, a efectos prácticos se toma como valor para la anchura equivalente la anchura a la altura mitad $\Delta f_{1/2}$, por lo que podemos escribir que en el caso de los perfiles de línea gaussianos $\Delta f_{1/2} \simeq \Delta f_{eq} \simeq \Delta f$.

Las causas físicas que provocan un determinado perfil de línea son varias; a continuación explicaremos las más habituales.

- **Ensanchamiento natural.** Debido al principio de incertidumbre de Heisenberg, si existe una incertidumre Δt en el tiempo en el que un sistema permanece en un estado de energía dado, entonces la energía del sistema está en un rango de energías ΔE, lo que se traduce en que los fotones emitidos no tienen todos exactamente la misma frecuencia. Esto conlleva que existe una dispersión de frecuencias Δf, que provoca un perfil de línea lorentziano

  $$\phi(f) = \frac{2}{\pi\Delta f}\ \frac{1}{1+4(f-f_0)^2/\Delta f^2} \qquad . \tag{1.10}$$

  Típicamente, este rango de frecuencias es muy pequeño, llegando a ser observacionalmente despreciable en el caso de las radiofrecuencias (caso que nos ocupa).

- **Ensanchamiento colisional.** La colisión de otras partículas puede interrumpir el proceso de emisión, cambiando el tiempo característico del mismo.

  Esto provoca un fenómeno similar al ya visto en el ensanchamiento natural; sin embargo, en este caso el efecto sí que puede llegar a ser significativo a radiofrecuencias.

- **Ensanchamiento térmico o Doppler.** Las partículas presentes en un elemento de volumen presentan una distribución de velocidades, tanto hacia el observador como alejándose del mismo. Las componentes radiales de estas velocidades (v_r) generan un corrimiento de frecuencia (en el límite no relativista) que se puede calcular como

  $$\frac{f-f_0}{f_0} = -\frac{v_r}{c}, \tag{1.11}$$

  donde f_0 es la frecuencia de la transición en el sistema de referencia de la partícula, f es la frecuencia observada, v_r es la velocidad radial y c es la velocidad de la luz.

Si suponemos que la distribución de velocidades que provoca estos corrimientos a distintas frecuencias es maxweliana, podemos entonces afirmar que el perfil de línea que observamos será un perfil gausiano, caracterizado por una anchura Δf a la altura mitad. Esta Δf se puede obtener de acuerdo a la ecuación

$$\Delta f = \left(\frac{8 \left(\ln 2 \right) kT}{m} \right)^{1/2} \frac{f_0}{c}, \tag{1.12}$$

siendo T la temperatura cinética y m la masa de las partículas. De esta forma podemos, por tanto, inferir información acerca de la temperatura del gas existente en el elemento de volumen analizado. Además, dado que las frecuencias se identifican unívocamente con determinadas transiciones, podemos también obtener información acerca de la naturaleza y estado de excitación de las partículas de dicho gas.

1.4.3. Líneas de emisión y absorción

Cuando observamos un espectro nos podemos encontrar dos clases de líneas, las de emisión y las de absorción.

Si comparamos con la emisión del continuo adyacente, las líneas de emisión tienen una intensidad mayor que el promedio de dicho continuo, mientras que las líneas de absorción tienen una intensidad menor. Por tanto, si eliminamos la emisión del continuo (o fondo), obtendremos una intensidad positiva en el caso de las líneas de emisión y una intensidad negativa en el caso de las líneas de absorción. El primer caso (emisión) es el habitual cuando la temperatura de fondo es la de la radiación cosmológica de fondo; el segundo caso, (absorción), es común cuando observamos la línea espectral sobre el fondo brillante de una fuente de continuo.

1.5. Caso científico

Tal y como ya hemos comentado, en este trabajo hemos desarrollado un espectrómetro de alta resolución. Contamos para ello con un ancho de banda de 250 MHz, con 16384 canales. Esta configuración nos permite obtener un ancho de canal de 15.259 kHz, equivalente a una velocidad de 0.0457 Km/s. Nótese que el cálculo para la conversión de frecuencia a velocidad ha sido hecho tomando como frecuencia de reposo la de 100 GHz, que es la frecuencia central del rango permitido por SEQUOIA, uno de los frontends operativos en el GTM.

Esta resolución nos permitirá trabajar con líneas muy próximas y/o angostas, tales como las que se presentan en los máseres, el gas frío, cometas, atmósferas estelares, atmósferas planetarias, desdoblamiento Zeeman, perfiles no gausianos, biomoléculas o gradientes suaves de velocidad. A continuación mostramos algunas de estas aplicaciones científicas para contextualizar la importancia de la alta resolución espectral.

1.5.1. Máseres

Los máseres pueden ser utilizados en la medición de distancias, en la determinación de la constante de Hubble, en el estudio de la dinámica de las nubes, determinación de propiedades físicas, etc. El ancho de línea de los máseres suele ser muy estrecho, debido principalmente a dos efectos. Primero, para mantener la emisión estimulada, es necesario que la velocidad del gas involucrado en el máser no varíe mucho. Si no fuera así, debido al efecto Doppler, la frecuenia de la emisión no podría estimular más radiación. Segundo, debido a la amplificación exponencial en el centro de la línea, las alas quedan más débiles, mientras que el núcleo de la línea aumenta mucho, resultando en líneas sumamente estrechas — FWHM menores que 1 km s^{-1} son comunes. Por lo tanto, es necesaria una buena resolución espectral para resolver la estructura de la línea o distinguir máseres cercanos entre sí.

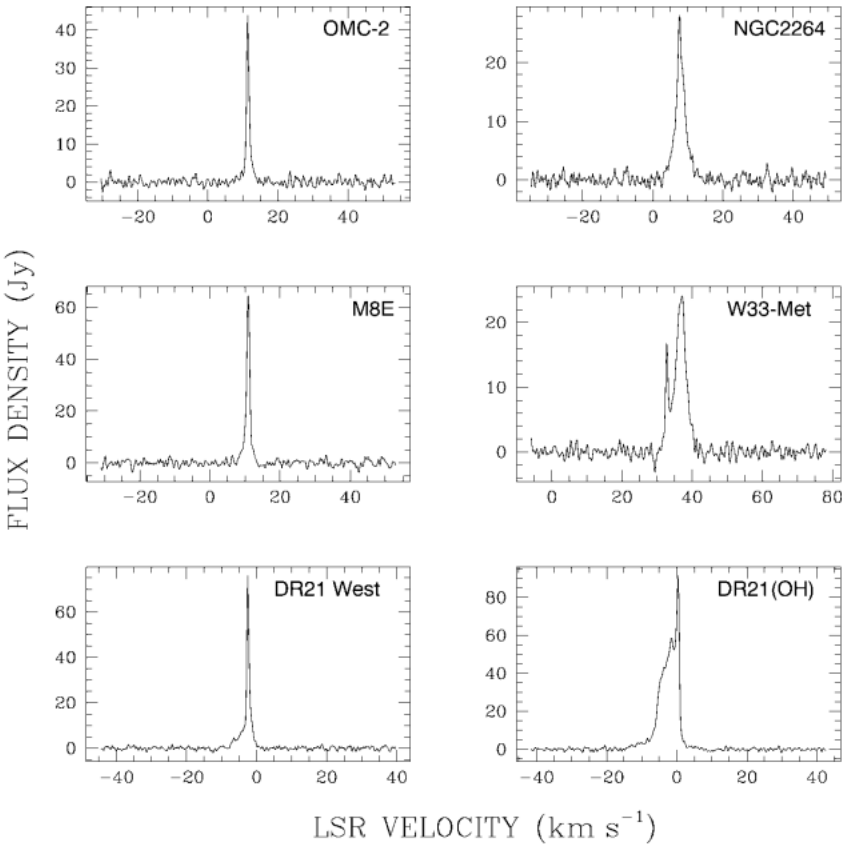

Figura 1.6: Espectros de máseres de metanol 133 GHz en Slysh et al. (1997). La resolución espectral es de 48.83 KHz

1.5.2. Efecto Zeeman

Existe un acuerdo generalizado en que el campo magnético juega un papel importante en muchos contextos astronómicos, incluyendo, entre otros, los procesos de formación estelar (Nixon & Pringle 2019). Por tanto, la observación y cuantitificación de los campos magnéticos es de vital importancia. Sin embargo, el campo magnético resulta un parḿetro relativamente dificil medir. Una de las herramientas más eficaeces que se puede utilizar para medirlo es el estudio del efecto Zeeman en los máseres.

El desdoblamiento de niveles de energía en una molécula, producido por el efecto Zeeman, resulta en tres niveles cercanos en energía. El nivel del medio no está polarizado,

mientras que el nivel superior y el inferior tienen polarizaciones opuestas. Por lo tanto, estas las líneas de los niveles superior/inferior en energí a muestran una pequeña separación en frecuencia entre las distintas polarizaciones. Como ejemplo, Momjian & Sarma (2017) reportan la detección del efecto Zeeman en la línea de 44 GHz Clase I de un maser de metanol, como se muestra en la Figura 1.8. Para medir el efecto Zeeman y así determinar el campo magético, es necesario resolver espectralmente la diferencia entre las dos polarizaciones. Como se puede apreciar en la figura, para caractarizar el "perfil S", hay que alcanzar una resolución espectra muy alta. En el caso mostrado, Momjian & Sarma (2017) observaron con canales de 3.91 kHz (0.027 km s^{-1}).

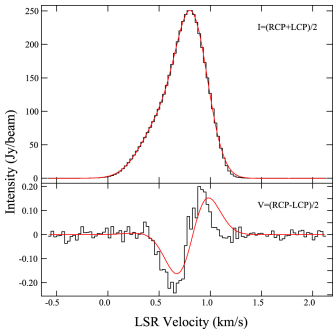

Figura 1.7: Anchos de línea en Momjian & Sarma (2017)

Table 3
Fitted and Derived Parameters of the Observed Masers in DR21(OH)

R.A. (J2000)	Decl. (J2000)	Intensity[a] (Jy beam^{-1})	Center Velocity (km s^{-1})	Velocity Linewidth[b] (km s^{-1})
20 38 59.25	42 22 48.2	219.23 ± 3.91	0.826 ± 0.002	0.365 ± 0.002
...	...	82.74 ± 2.13	0.531 ± 0.009	0.484 ± 0.010
20 38 59.29	42 22 47.0	7.11 ± 0.12	−0.855 ± 0.003	0.372 ± 0.007
20 38 59.31	42 22 49.1	16.33 ± 0.12	1.099 ± 0.001	0.352 ± 0.003
20 38 59.33	42 22 46.8	6.64 ± 0.10	−1.004 ± 0.004	0.300 ± 0.008
20 38 59.70	42 22 41.7	0.35 ± 0.01	0.010 ± 0.010	0.786 ± 0.027
20 38 59.71	42 22 45.3	0.52 ± 0.04	−0.101 ± 0.014	0.402 ± 0.034
20 38 59.71	42 22 49.1	1.03 ± 0.34	0.202 ± 0.010	0.264 ± 0.041
...	...	0.49 ± 0.15	−0.036 ± 0.128	0.475 ± 0.148
20 38 59.76	42 22 44.7	0.72 ± 0.04	−0.249 ± 0.010	0.365 ± 0.025
20 38 59.85	42 22 45.4	14.79 ± 0.09	−0.216 ± 0.001	0.219 ± 0.003
20 38 59.85	42 22 45.7	3.80 ± 0.04	−0.510 ± 0.002	0.227 ± 0.004
20 38 59.89	42 22 44.9	16.46 ± 0.12	0.312 ± 0.002	0.241 ± 0.004
20 38 59.91	42 22 44.5	18.11 ± 0.14	0.561 ± 0.001	0.203 ± 0.001
20 38 59.96	42 22 34.7	0.75 ± 0.05	−1.836 ± 0.009	0.267 ± 0.021
20 39 00.15	42 22 47.3	10.48 ± 0.07	−1.452 ± 0.001	0.387 ± 0.003
20 39 00.23	42 22 45.4	323.20 ± 1.22	0.396 ± 0.001	0.309 ± 0.002
...	...	44.30 ± 1.22	0.098 ± 0.008	0.309 ± 0.011
20 39 00.25	42 22 46.8	11.79 ± 0.10	0.095 ± 0.001	0.279 ± 0.003
20 39 00.25	42 22 46.0	0.83 ± 0.05	1.240 ± 0.016	0.605 ± 0.041
...	...	7.99 ± 0.06	0.367 ± 0.001	0.368 ± 0.003
20 39 00.30	42 22 46.6	0.90 ± 0.01	−2.049 ± 0.003	0.312 ± 0.007
20 39 00.33	42 22 47.8	1.15 ± 0.01	−0.029 ± 0.002	0.455 ± 0.004
20 39 00.51	42 22 47.1	0.72 ± 0.03	1.352 ± 0.009	0.270 ± 0.018
20 39 00.53	42 22 47.1	1.82 ± 0.04	2.524 ± 0.012	0.468 ± 0.023
...	...	1.09 ± 0.19	2.116 ± 0.025	0.300 ± 0.054
20 39 00.52	42 22 47.1	2.77 ± 0.05	1.783 ± 0.009	0.367 ± 0.022
20 39 01.01	42 22 41.1	0.81 ± 0.01	−1.017 ± 0.002	0.470 ± 0.005
20 39 01.20	42 22 40.5	0.63 ± 0.02	−2.369 ± 0.011	0.755 ± 0.031

Notes.
[a] The intensity values are primary beam corrected.
[b] The velocity linewidth was measured at FWHM.

Figura 1.8: Resultados de Momjian & Sarma (2017)

1.5.3. Moléculas Pesadas, Frías y no-Turbulentos

Una sorpresa encontrada a principios de las observaciones de moléculas en el medio interestelar fue el ancho relativamente grande de las líneas. Por ejemplo, a temperaturas de 30 K, típicas para nubes moleculares, se espera un FWHM para CO de aproximadamente 0.2 km s^{-1}. Sin embargo, los anchos observados suelen ser de varios kilómetros por segundo. Ahora se entiende que dentro de las nubes moleculares existen movimientos supersónicos que producen estos perfiles anchos.

No obstante, existen regiones "tranquilas", sin turbulencia, donde las moléculas presentan perfiles que correspondan a su ancho térmico. En una region así, moléculas típicas (como CO) tendrán anchos de décimas de kilómetro por segundo. También existen moléculas más pesadas, por ejemplo las cianopolynes (HC_3N, HC_5N, HC_7N, etc.) que

tienen FWHM muy estrechas. Usando la fórmula de Estalella y Anglada,

$$\frac{\Delta v_{th}}{\text{km s}^{-1}} = 0{,}21\left(\frac{T_k}{\text{K}}\right)^{1/2}\left(\frac{\text{m}}{\text{m}_\text{H}}\right)^{-1/2},$$

tendremos para HC_7N un peso de 99m_H, dando Δv_{th} de 0.12 km s^{-1}. Para resolver líneas tan estrechas, con un número razonable de canales sobre la línea (5 canales como mínimo, 10 canales deseables) se requiere de resoluciones del orden de 0.01 km s$^-$1 o 3.3 kHz de ancho para frecuencias de 100 GHz.

1.5.4. Líneas de Radiorecombinación

En regiones de gas ionizado, principalmente regiones HII, el proceso de recombinación puede resultar en átomos Rydberg, en los cuales el electrón se encuentra en estados de energía muy elevados, con número cuántico principal de varios cientos. Debido a su gran tanaño ($r_n \propto n^2$) estos átomos son extremadamente susceptibles a ensanchamiento, por varios mecanismos.

Los distintos mecanismos provocan perfiles distintos en las lí neas, incluyendo perfiles Gaussianos y Lorentzianos, y su combinación, el perfil de Voigt. Para distinguir entre estos perfiles hay que tener un espectrómetro capaz de trazar las alas de las lí neas, además de su núcleo. Aunque no requiere de alta resolución espectral, sí que se requiere de un espectrómetro sensible, para trazar emisión débil.

Un ejemplo es Keto et al. (2008), quienes determinan la contribución por separado del ensachamiento térmico, el ensanchamiento dinámico y el ensanchamiento por presión, llegando a la conclusión de que el ensanchamiento por presión también tiene una importante contribución, ya que el ancho de las líneas decrece con el incremento en frecuencia. Para llegar a esta conclusión, tuvieron que caracterizar con buena precisión los perfiles de las tres líneas a distintas frecuencias, como se muestra en la figura 1.9.

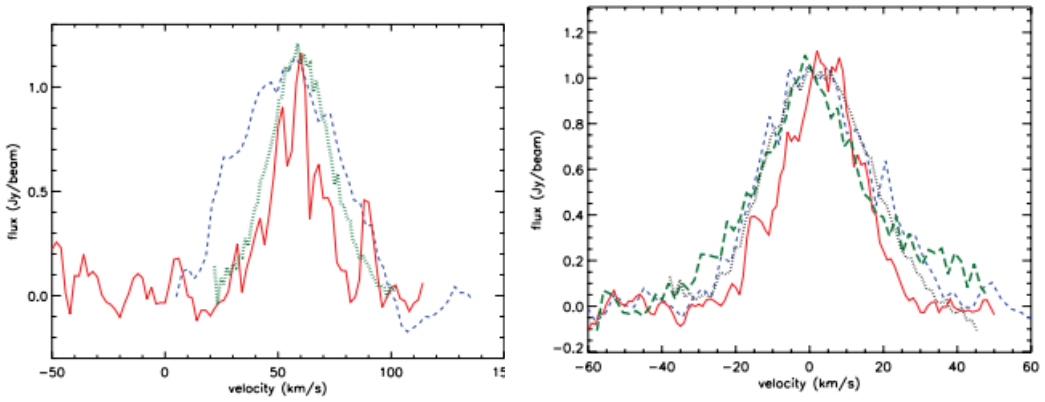

FIG. 1.—H66α (*dashed blue line*), H53α (*dotted green line*), and H5
(*solid red line*) in W51e2. The flux densities of the H66α line and the H53α li
have been multiplied by 80 and 27, respectively.

FIG. 5.—H92α (*long-dashed green line*), H66α (*short-dashed blue line*),
H53α (*dotted black line*), and H30α (*solid red line*) in G10.6−0.04. The flux
densities of the H92α, H66α, and H53α lines have been multiplied by 115, 5.5,
and 4.8, respectively.

Figura 1.9: (Keto et al. 2008)

1.6. Estructura de la tesis

Este trabajo consta de esta introducción, 5 capítulos y un apéndice.

- **Capítulo 2. Procesamiento digital de señales.** Estableceremos el marco teórico de esta tesis, explicando los conceptos fundamentales que hemos utilizado a lo largo del trabajo. Haremos una introducción a los conceptos fundamentales del **procesamiento digital de señales**, explicando los conceptos matemáticos de las transformada de Fourier y el teorema de Wiener-Khinchin. A continuación explicaremos el proceso de conversión analógico-digital de las señales, hablando de la cuantificación y el muestreo. Concluiremos este capítulo hablando de espectrómetros existentes, con el fin de ubicar al lector y darle un contexto a nuestro proyecto.

- **Capítulo 3. Implementación.** Explicaremos la implementación del espectrómetro. Para ello comenzaremos hablando del hardware utilizado, dando sus especificaciones y justificando nuestra elección. En la segunda parte del capítulo hablaremos sobre el software utilizado, explicando las funcionalidades implementadas; explicaremos en detalle los modelos que hemos implementado. En ambas secciones (hardware y sotfware) expondremos las complicaciones que hemos encontrado y las distintas soluciones que hemos implementado.

- **Capítulo 4. Pruebas de Banco.** En este capítulo explicaremos las pruebas que hemos realizado para evaluar el funcionamiento del espectrómetro, presentando los

resultados de las mismas. Explicaremos las pruebas en el laboratorio y las pruebas realizadas con pequeñas antenas.

- **Capítulo 5. Conclusiones.** Aquí expondremos las conclusiones a las que hemos llegado, y estableceremos una serie de líneas sobre las que nos gustaría continuar trabajando en un futuro.
- **Apéndice.** Incluiremos los códigos fuente de la pipeline.

Capítulo 2

Procesamiento digital de señales

2.1. Procesamiento digital de Señales.

La densidad espectral de potencia de una señal es una función matemática que nos informa de cómo está distribuida la potencia de dicha señal en función de las frecuencias por la que está formada. El reto de un espectrómetro digital es poder medir esta densidad espectral de potencia de la forma más precisa posible.

A efectos prácticos, la densidad espectral de potencia nos da el perfil de la línea espectral que queremos observar.

2.1.1. Muestreo

El procesamiento digital de cualquier señal analógica comienza con la digitalización de la misma, siendo el primer paso su muestreo digital. Durante este proceso se toman muestras de la señal a una tasa de muestreo (frecuencia) constante y se cuantifican, convirtiendo la serie infinita de valores que puede tomar la amplitud en una serie discreta de valores determinados; el valor de cuantificación se obtiene por aproximación dentro de un conjunto de niveles definidos a priori. A continuación se convierten esos valores a su representación binaria para su almacenamiento.

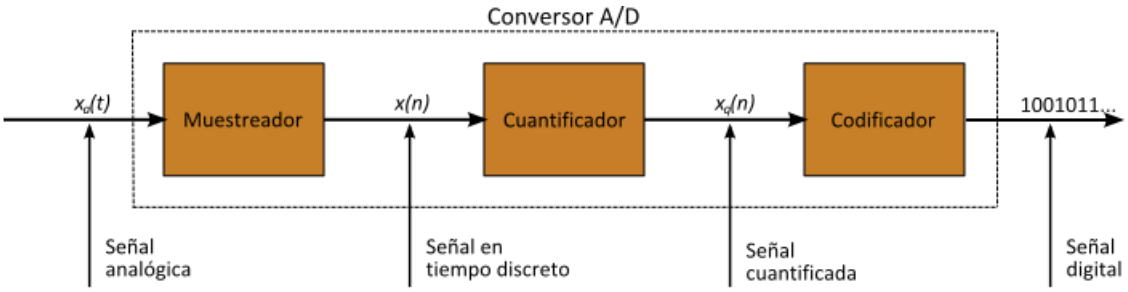

Figura 2.1: Conversor analógico-digital

Este proceso lleva implícita la pérdida de fidelidad a la señal original, debido fundamentalmente a dos motivos. En primer lugar, tenemos una serie finita de valores, por lo que no podemos representar dicha señal en cada instante (sería una serie infinita de valores). En segundo lugar, obtenemos un **error de cuantificación** debido a la aproximación que hacemos a los valores escogidos.

Dicho lo anterior, hay dos parámetros cuya importancia salta a la vista: el primero, la frecuencia o tasa de muestreo; el segundo, en nivel de precisión de la cuantificación. El primero de estos parámetros, la frecuencia, nos lleva a una explicación algo más larga que la cuantificación, por lo que pasaremos a explicar esta última en primer lugar.

2.1.2. Cuantificación

Como ya dijimos en la seccón 2.1.1, el procesamiento digital de señales analógicas requiere la conversión de dichas señales a señales digitales. Uno de los detalles a tener en cuenta en esta conversión es la decisión de cuantos bits de información se van a requerir para representar la señal digital. Este número determinará la precisión de la conversión y el rango dinámico que tendrá dicha conversión.

Vamos a definir como rango dinámico la diferencia entre el nivel medio de ruido y el nivel pico, medido en decibelios. Dado que cada vez es más frecuente que los radioastrónomos deseen observar en frecuencias que no están en bandas protegidas (Group 2006) (Bianchi et al. 2008), o que señales de origen artificial se cuelen en las bandas observadas, las interferencias provenientes de las señales de origen artificial deben ser tenidas en cuenta. Con el fin de tener en cuenta estas interferencias, los sistemas diseñados para los radiotelescopios deben incrementar su rango dinámico, buscando evitar la saturación de los mismos. Esto es debido a que las señales de naturaleza astronómica son de muy baja intensidad y las señales de origen artificial pueden (y suelen) ser mucho

más intensas.

Pongamos un ejemplo en el que se va a buscar registrar la potencia de una señal con un rango dinámico de $20dB$. Recordemos que la potencia de una señal la calculamos como

$$P \propto 10 \log \left(V_p^2\right) = 20 \log \left(V_p\right) \tag{2.1}$$

y que lo que se registra es el nivel de voltaje detectado. Dado que lo que se registra es el nivel de voltaje, por cada bit que utilicemos en nuestra cuantificación podremos doblar nuestro nivel, de tal forma que con cada bit que incrementemos vamos a poder registrar $2V_p$. Si lo aplicamos en la ecuación 2.1, y calculamos la potencia que nos da cada bit (pasamos de V_p a $2V_p$), obtenemos que

$$P = 20 \log \left(\frac{2V_p}{V_p}\right) = 20 \log \left(2\right) \approx 6dB \tag{2.2}$$

lo que quiere decir que por cada bit ganamos 6 dB. Con un simple cálculo se obtiene que para conseguir un rango dinámico de $20dB$ se necesitan 4 bits. De igual forma, con el mismo cálculo se determina que para conseguir $30db$ hacen falta 5 bits.

2.1.3. Teorema del muestreo (Nyquist)

Fue formulado por primera vez por Harry Nyquist en 1928 (Certain topics in telegraph transmission theory), pero su demostración hubo de esperar hasta 1949, fecha en la que Claude E. Shannon lo demostró formalmente (de ahí que también se conozca como teorema de muestreo de Nyquist-Shannon) (Shannon 1949).

Este teorema demuestra que para que una señal pueda ser reconstruida de forma exacta, esta debe estar limitada en banda (con ancho de banda B) y se debe muestrear, al menos, al doble de su ancho de banda ($f_s \geq 2B$) (entendemos como ancho de banda la diferencia entre la frecuencia máxima y la frecuencia mínima de la señal); si se dan estas dos condiciones, toda la información existente en la señal analógica está contenida en la serie de muestras tomadas.

2.1.4. Aliasing

En el caso de que a la hora de muestrear no se cumpla con el teorema de Nyquist-Shannon, y muestreemos por debajo de dos veces la frecuencia máxima presente en la señal, vamos a encontrarnos con el efecto conocido como aliasing. Este efecto provoca que aparezcan armónicos de frecuencias superiores a $\frac{f_s}{2}$ dentro del espectro,

afectando al mismo. En algunos casos, este efecto se usa de forma intencionada para muestrear dichas señales de frecuencia más altas que las que teóricamente podríamos muestrear.

Para minimizar los efectos de este efecto se incorporan filtros a los sistemas. Estos filtros se encargan de evitar que las frecuencias no deseadas contaminen el espectro.

2.1.5. Transformada de Fourier

La transformada de Fourier es una función matemática que nos permite obtener la representación en el dominio de la frecuencia de una función matemática en el dominio del tiempo. La transformada de Fourier de una señal no periódica viene dada por

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt \quad . \tag{2.3}$$

Como se puede ver de la ecuación 2.3, para obtener la transformada de Fourier nos hace falta una señal continua e infinita. Sin embargo, en el procesamiento digital de señales trabajamos con señales finitas y discretas (las muestreamos durante un tiempo finito y a intervalos finitos), por lo que no es aplicable la anterior ecuación. Para procesar este tipo de señales utilizamos la transformada discreta de Fourier (DFT por sus siglas en inglés), que se expresa de la siguiente forma:

$$X(\omega) = \sum_{n=0}^{\infty} x(n)e^{-j\omega n} \quad . \tag{2.4}$$

Dado que $e^{-j\omega n} = cos\ (\omega n) - j\ sin\ (\omega n)$, obtenemos que:

$$X(\omega) = \sum_{n=0}^{\infty} x(n)(cos\ (\omega n) - j\ sin\ (\omega n)) \quad . \tag{2.5}$$

Estas señales tienen un espectro continuo y periódico (período 2π), por lo que toda la información de la transformada $X(\omega)$ está contenida en el intervalo de 0 a 2π. Esto nos permite reducir la expresión anterior a un número finito de N muestras entre 0 y 2π, quedando la siguiente expresión, en la que $\omega = \frac{2\pi}{N}k$:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-\frac{2\pi j}{N}kn} = \sum_{n=0}^{N-1} x(n)(cos\ (\frac{2\pi}{N}kn) - j\ sin(\frac{2\pi}{N}kn)) \qquad k = 0, ..., N-1 \tag{2.6}$$

La DFT tal y como está indicada en la ecuación 2.6 es una función compleja, aceptando entradas con parte compleja, y produciendo espectros con una componente compleja.

En el caso de entradas reales (como las que tenemos cuando estudiamos una señal radioastronómica) la transformada $X(k)$ es simétrica, por lo que toda la información espectral queda contenida en el intervalo que va de 0 a π. Teniendo en cuenta esta simetría, introducimos el concepto de DFT real, que supone una versión simplificada de la DFT, y con la que podemos obtener la señal de salida usando únicamente $\frac{N}{2}$ muestras (las que están entre 0 y π). Utilizando esta DFT real, transformamos nuestra señal de N muestras en dos señales de salida de $N/2+1$ puntos que contienen amplitudes de ondas sinusoidales; una de ellas contiene las amplitudes del coseno, y corresponde a la parte real de la señal en el dominio de la frecuencia (Re $X(k)$), mientras que la otra contiene las amplitudes del seno, correspondiendo a la parte imaginaria (Im $X(k)$). De esta forma, al aplicar la DFT a nuestra nuestra señal obtenemos las amplitudes normalizadas de las distintas componenete que forman dicha señal, o dicho de otra forma, el espectro de nuestra señal.

2.1.6. Banco de filtros polifásicos

Tal y como comentamos en la sección 2.1.5, la utilización de la DFT presenta ciertos problemas. Los bordes abruptos que delimitan las señales finitas provocan que al calcular la transformada de Fouier discreta obtengamos funciones *sinc* (fig 2.2).

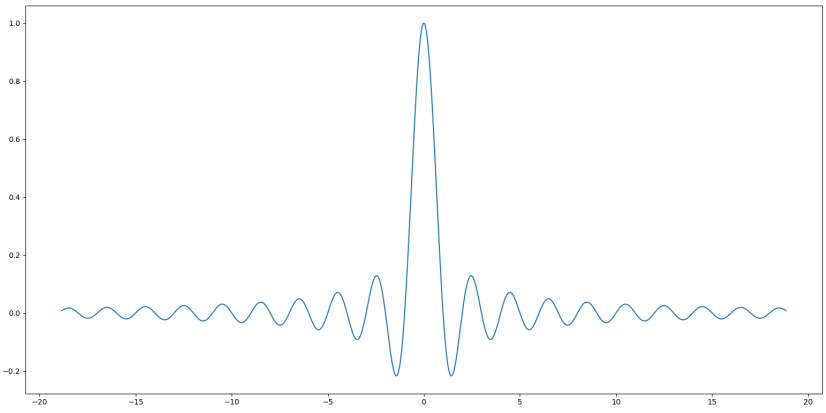

Figura 2.2: Función sinc

Estas funciones sinc provocan un efecto llamado *spectral leakage*, que consiste en la *fuga* de energía desde una frecuencia dada a las adyacentes. Esta fuga ocasiona que el espectro no sea del todo preciso, ya que está recogiendo para cada frecuencia energíax que realmente

no está presente. En la fig 2.3 se puede apreciar el resultado de una transformada de Fourier (parte superior) y de una DFT (parte inferior).

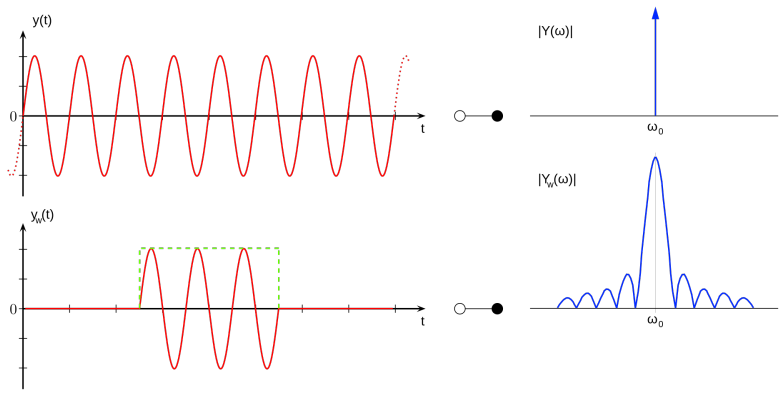

Figura 2.3: Efecto *Spectral Leakage* (Wikipedia contributors 2018)

Los bancos de filtros polifásicos (PFB de ahora en adelante) nos ofreces una forma eficiente de suavizar el *spectral leakage*, consiguiendo un espectro de mayor precisión. En la fig 2.4 (Harris & Haines 2011) se puede ver el resultado de la aplicación de un filtro polifásico, comparado con una FFT plana y una FFT con una función ventana de Hanning.

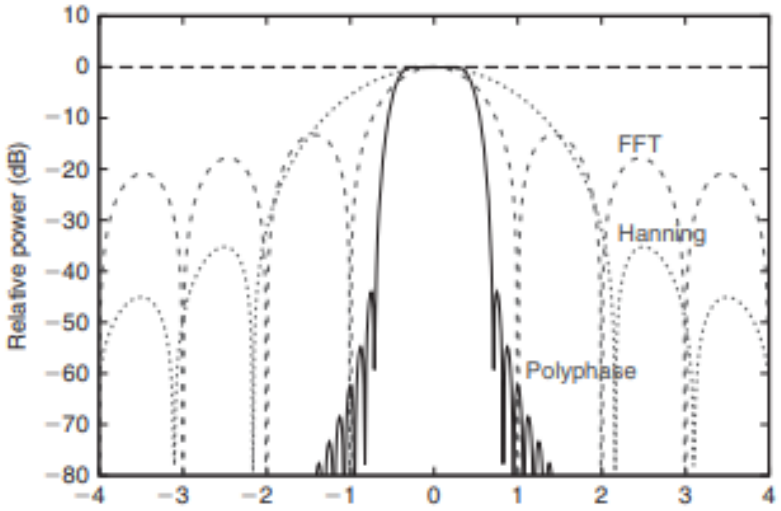

Figura 2.4: Banco de filtros polifásicos vs FFT vs FFT Hanning (Harris & Haines 2011)

2.1.7. Teoremas de la Correlación Cruzada y de Wiener-Khinchin

El teorema de correlación cruzada relaciona la transformada de Fourier de la función de correlación cruzada de dos señales con las transformadas de Fourier de las señales individuales. Supongamos $x(u)$ y $y(u)$ como dos señales integrables, que son 0 en cualquier punto fuera del intervalo $u \in [0, T)$, con valores en el intervalo $[-1, 1]$. La función de correlación de estas dos señales es

$$C_{xy}(\tau) = \int_{-\infty}^{\infty} x(u-\tau)y(u)du \tag{2.7}$$

donde τ es el intervalo temporal. El teorema de la Correlación Cruzada establece que

$$C_{xy}(\tau) = X^*(f)Y(f) \tag{2.8}$$

donde $X(f)$ y $Y(f)$ son las transformadas de Fourier de $x(t)$ y $y(t)$ y $X^*(f)$ es el conjugado de $X(f)$. El teorema de Wiener-Khinchin es un caso especial del teorema de la Correlación Cruzada en el que las dos funciones $x(t)$ y $y(t)$ son la misma función (autocorrelación); si suponemos que $x(t)$ es la función que representa el voltaje de nuestra señal y la llamamos $v(t)$, la ecuación 2.8 queda como

$$A_{vv}(\tau) = \int_{-\infty}^{\infty} v(t-\tau)v(t)dt \longleftrightarrow V^*(f)V(f) \Longrightarrow A_{vv}(\tau) = |V(f)|^2 \tag{2.9}$$

Una vez tenemos esta función de autocorrelación calculada, calculando la FFT de la autocorrelación podemos obtener el espectro de potencia de nuestra señal.

Las señales analógicas que captamos con los radiotelescopios son señales de potencia finita; para este tipo de señales es posible calcular la transformada de Fourier generalizada, pero en general no podemos calcular su módulo al cuadrado, que nos daría la potencia de la misma. Es en estos casos cuando recurrimos al teorema de Wiener-Khinchin, que acabamos de ver.

La figura 2.9 ilustra esquemáticamete dos maneras distintas para obtener el espectro de potencia de una señal $v(t)$. Si tomamos el camino superior, calculamos el espectro de potencia a través de la transformada rápida de Fourier. Si tomamos el camino inferior, utilizamos entonces el teorema de Wiener-Khinchin, calculando primero la función de autocorrelación y calculando después la transformada rápida de Fourier de esta función de autocorrelación. Esta segunda manera puede parecer absurda, dado que anãdimos un paso intermedio, agregando complejidad y siendo computacionalmente más costosa, pero hay casos en los que es necesario hacerlo de esta manera.

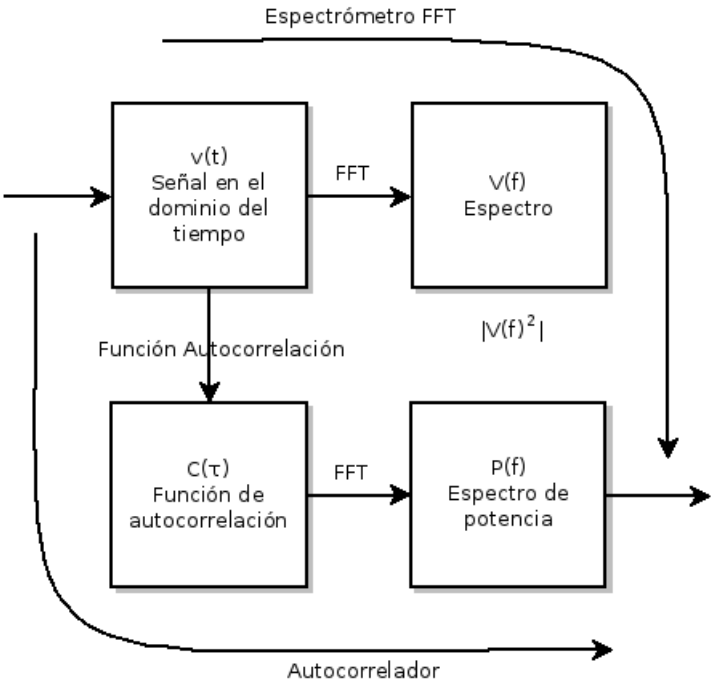

Figura 2.5: Cálculo del espectro de potencia

2.1.8. Transformada Rápida de Fourier

El cálculo de la DFT se puede abordar desde distintos enfoques (resolución de sistema lineal de N ecuaciones lineales, correlación, etc.) pero nosotros nos vamos a centrar en el que utiliza el algoritmo conocido como Transformada Rápida de Fourier (FFT, por sus siglas en inglés).

Si nos detenemos a analizar la ecuación 2.6, vemos que el algoritmo necesario para su implementación tiene una complejidad de N^2, siendo N el número de puntos de la DFT. Un detalle a tener en cuenta es que este número de puntos va a determinar el número de canales del espectro. El número de canales que tendrá el espectro será $\frac{N}{/}2$.

A continuación vamos a mostrar que la descomposición de la ecuación 2.6 permite reducir la complejidad del algoritmo a $N\ \log\ N$, correspondiendo esta descomposición al algoritmo conocido como transformada rápida de Fourier. Veamos el algoritmo.

Comenzaremos por simplificar las expresiones, reescribiendo la ecuación 2.6:

$$X(k) = \frac{1}{N} \sum_{n=0}^{N-1} x_n W_N^{kn} \tag{2.10}$$

donde

$$W_N = e^{\frac{-j2\pi}{N}} \tag{2.11}$$

y N se supone de la forma

$$N = 2^n \tag{2.12}$$

con n entero positivo. Entonces puede expresarse:

$$N = 2M \tag{2.13}$$

con M también entero positivo. Sustituyendo 2.13 en 2.10 se tiene:

$$X(k) = \frac{1}{2M} \sum_{n=0}^{2M-1} x(n) W_{2M}^{kn} =$$

$$\frac{1}{2} \left\{ \frac{1}{M} \sum_{n=0}^{M-1} x(n) W_{2M}^{k(2n)} + \frac{1}{M} \sum_{n=0}^{M-1} x(2n+1) W_{2M}^{k(2n+1)} \right\} \quad . \tag{2.14}$$

Dado que de 2.11 podemos decir que $W_{2M}^{2kn} = W_M^{kn}$, la ecuación 2.14 puede expresarse como:

$$\frac{1}{2} \left\{ \frac{1}{M} \sum_{n=0}^{M-1} x(2n) W_M^{kn} + \frac{1}{M} \sum_{n=0}^{M-1} x(2n+1) W_M^{kn} W_{2M}^{k} \right\} \quad . \tag{2.15}$$

Si ahora definimos:

$$X_{par}(k) = \frac{1}{M} \sum_{n=0}^{M-1} x(2n) W_M^{kn} \text{ para k= 0, 1, ... M-1} \tag{2.16}$$

$$X_{impar}(k) = \frac{1}{M} \sum_{n=0}^{M-1} x(2n+1) W_M^{kn} \text{ para k= 0, 1, ... M-1} \tag{2.17}$$

entonces obtenemos que la ecuación 2.15 la podemos escribir como:

$$X(k) = \frac{1}{2} \left\{ X_{par}(k) + X_{impar}(k) W_{2M}^{k} \right\} \tag{2.18}$$

Si además utilizamos $W_M^{k+M} = W_M^k$ y $W_{2M}^{k+M} = -W_{2M}^k$, obtendremos

$$X(k+M) = \frac{1}{2} \left\{ X_{par}(k) - X_{impar}(k) W_{2M}^{k} \right\} \tag{2.19}$$

De las ecuaciones 2.18 y 2.19 podemos apreciar que una transformada de N puntos puede ser calculada dividiéndola en dos partes. El cálculo de la primera mitad de $X(k)$ requiere de la evaluación de las dos transformadas de $N/2$ puntos según las ecuaciones 2.16 y 2.17; sustituimos entonces en 2.18 los valores resultantes de $X_{par}(k)$ y $X_{impar}(k)$, obteniendo $X(k)$ para k=0,1,2...,(N/2-1). De la ecuación 2.19 obtenemos que la otra mitad de $X(k)$, desde k=N/2 hasta k=N no requiere evaluaciones adicionales, con lo que obtenemos un ahorro computacional.

Si consideramos un número de muestras $N = 2^n$, con n entero positivo, podemos demostrar que el número de multiplicaciones y sumas está dado por:

$$multiplicaciones = m(n) = 2m(n-1) + 2^{n-1}, \quad n \geq 1 \tag{2.20}$$

$$sumas = a(n) = 2a(n-1) + 2^n, \quad n \geq 1 \tag{2.21}$$

siendo expresiones recursivas para las que m(0) y a(0) son iguales a 0, puesto que la transformada de un punto no requiere operación alguna.

Es posible, por tanto, determinar que el número de operaciones que se requiere para implementar el algoritmo de la FFT está dado por:

$$m(n) = \frac{1}{2}2^n \log_2 2^n = \frac{1}{2}N \log_2 N = \frac{1}{2}Nn \quad n \geq 1$$

y

$$a(n) = 2^n \log_2 2^n = N \log_2 N = Nn \quad n \geq 1 \quad ,$$

con lo que se demuestra que el algoritmo de la FFT es del orden $N \log N$.

2.2. Espectrómetros digitales. Estado del arte.

Actualmente existen varios diseños de características similares al que pretendemos desarrollar; entre ellos podemos destacar los siguientes: RUPPI (Roach Ultimate Pulsar Processing Instrument), una de las primeras aproximaciones al diseño de instrumentación astronómica con la plataforma ROACH; VEGAS (VErsatile GBT Astronomical Spectrometer), en el Green Bank Telescope; y WARES (Wideband Array Roach Enabled Spectrometer), instalado en el GTM (Gran Telescopio Milimétrico).

2.2.1. RUPPI

RUPPI s un proyecto que ha sido desarrollado utilizando la plataforma ROACH, y cuyo objetivo es facilitar el desarrollo de sistemas de procesamiento digital de señales en la astronomía. Sus características principales son las siguientes:

- Muestreo real de 8 bits.
- 2 polarizaciones.
- 800 MHz de ancho de banda.
- 2048 canales.
- Caracterización de los parámetros de Stock.
- Decimación tanto en frecuencia como en tiempo.
- Folding de pulsares on-line.

2.2.2. VEGAS

VEGAS se encuentra en operación en el Green Bank Telescope, en Virginia. Las principales características de este espectrómetro son las siguientes:

- El espectro puede medir desde 8 beams duales polarizados
- Utiliza conversores analógico digitales de 8 bits
- Tiene un ancho de banda utilizable de 1.25 GHz (1.5 GHz nominales)
- Hasta 8 ventanas espectrales sintonizables
- Puede procesar hasta 10 GHz de ancho de banda total desde un beam dual polarizado
- 64 sub-bandas por beam cuando procesa señales desde un beam dual polarizado
- 32768 canales
- 17 modos de observación con resolución espectral desde 1.5 MHz a 30 Hz y anchos de banda desde 30 MHz a 1 MHz
- Ratio de volcado de datos de 0.5 mseg.

2.2.3. UMASS WARES

- Incorporado al GTM junto con SEQUOIA y operativo desde principios de 2018.
- Una única ventana espectral activa.
- Tiene 3 modos seleccionables, de 200 MHz, 400 MHz y 800 MHz; el número de canales disponibles es 2048, 4096 y 8192 respectivamente.
- Ratio de volcado de datos de 0.5 mseg.

2.2.4. Colaboración CASPER

Ninguno de estos proyectos (el nuestro tampoco) podrían haberse llevado a cabo sin la colaboración CASPER (Berkeley University of California 2006), que lleva más de una década desarrollando herramientas hardware y software para el estudio de la radioastronomía. En la figura 2.6 (Collaboration for Astronomy Signal Processing and Electronics Research (CASPER) 2018) podemos ver la evolución del hardware desarrollado por esta colaboración.

CASPER Processing Platform Comparison Matrix						
Platform	**iBOB**	**ROACH**	**ROACH-2**	**SKARAB**	**SNAP**	**SNAP-2**
Status	***OBSOLETE***	Deprecated	Current	Current	Current	Current
FPGA	Virtex-II Pro XC2VP50	Virtex-5 V5SX95T	Virtex-6 XC6VSX475T	Virtex-7 XC7VX690T	Kintex-7 XC7K160T	Kintex Ultrascale XCKU115
Auxiliary processor	PPC	PPC	PPC	microblaze	Raspberry Pi	Zynq XC7Z010 ARM
Logic Cells	53k	94k	476k	693k	162k	1160k
DSP Slices	232	640	2016	3600	600	5520
BRAM	4.2 Mb	8.8 Mb	38 Mb	53 Mb	11 Mb	76 Mb
SRAM	2x18 Mb	2x36 Mb	4x144Mb	8x 32 Gb	-	4x36 Mb
SRAM BW	9 Gbps	43 Gbps	200 Gbps	8x30 Gbps	-	40 Gbps
DDR	-	1x8 Gb	1x16 Gb	HMC replaces	-	1x 1GB DDR3
DDR BW	-	38 Gbps	50 Gbps	SRAM and DDR	-	8 bit interface
High-speed Ethernet	2x10G	4x10G	8X10G	< 16x40G	2x10G	4x40G, 16x10G
Low-speed Ethernet	1x10 Mbps	1x100 Mbps	2*1 Gbps	1x1 Gbps	1x1G on R-PI	2x 1G
Expansion Bus	2*ZDOK	2*ZDOK	2*ZDOK	4x Mega array 16x10 Gbps	1x ZDOK	2x HPC FMC, 1 ZD+
ADCs	-	-	-	-	3x HMCAD1511 (12 inputs)	-
Xilinx Tools Required	ISE	ISE	ISE	Vivado	Vivado	Vivado

Figura 2.6: Hardware desarrollado por CASPER

Capítulo 3

Implementación

La implementación de un espectrómetro digital se puede abordar, fundamentalmente, desde tres perspectivas distintas: utilizar un circuito integrado de aplicación específica o ASIC (por sus siglas en inglés Application Specific Integrated Circuit); utilizar una plataforma basada en arreglos de compuertas lógicas programables o FPGA (por sus siglas en inglés Field Programmable Gate Array); o utilizar una plataforma basada en unidades de procesamiento gráfico o GPU's (por sus siglas en inglés Graphics Processing Unit).

Para implementar el espectrómetro digital de alta resolución hemos optado por utilizar una plataforma basada en FPGA, ya que nos ofrece la posibilidad de prototipado y reprogramación, un coste de desarrollo y adquisición muchos menores y un tiempo de diseño y manufacturación más ajustados.

3.1. Hardware

A Para nuestro desarrollo hemos optado por la utilización de la primaera versión de una plataforma basada en FPGA llamada ROACH (Reconfigurable Open Arquitecture Computing Hardware), hardware desarrollado por CASPER. CASPER es una colaboración abierta cuyo objetivo fundamental es el desarrollo de nuevas herramientas para el procesamiento digital de señales, tanto a nivel de hardware como de sotfware.

El concepto sobre el que se fundamenta la colaboración CASPER es el del diseño modular, usando protcolos de comunicación estándar y hardware comercial. Con ello lo que se pretende es acortar los tiempos de desarrollo de los nuevos instrumento, así como facilitar su replicación y reaprovechamiento. Por citar un ejemplo, se utiliza ethernet y TCP/IP para la transmisión de datos, en lugar de estar definiendo un protocolo y diseñando un bus

para cada nueva aplicación.

CASPER es una comunidad muy activa, lo que nos ha ofrecido unas enormes ventajas a la hora de desarrollar este proyecto. Hay un gran nivel de participación y una enorme predisposición para ayudar con cualquier problema que pueda surgir a la hora de utilizar cualquiera de las herramientas que desarrollan.

3.1.1. FPGA's

Una FPGA o matriz de puertas programables es un dispositivo programable que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada en el momento mediante un lenguaje de descripción especializado. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica hasta complejos sistemas en un chip.

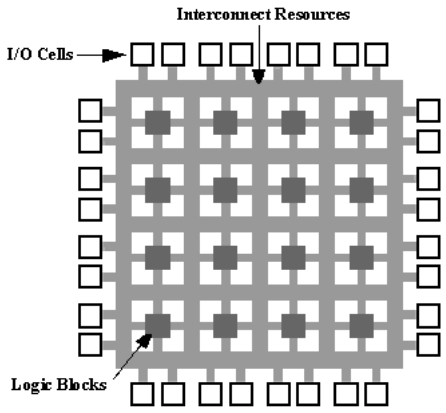

Figura 3.1: Diagrama arquitectura interna FPGA

Desde hace años el diseño de FPGAs ha evolucionado hacia la creación de plataformas que combinan los bloques lógicos con microprocesadores y periféricos para formar una plataforma programable, pudiéndose encontrar en estas plataformas funciones de alto nivel, tales como sumadores o multiplicadores, así como bloques de memoria a los que se puede acceder como registros. Actualmente la estructura básica de una FPGA consiste en un conjunto de bloques lógicos, hardware para conexión de red, bloques de entrada/salida, bloques de memoria, bloques para el procesamiento digital de señales, bloques de sincronización y bloques para la transmisión de datos a alta velocidad, tal y como se explica en (Yang et al. 2014). Los recursos utilizados por las FPGA's para ofrecernos plataformas programables, optimizadas y de fácil reutilización fueron analizados

en (Rodriguez-Andina et al. 2007).

3.1.2. ROACH v1

Para la implementación del espectrómetro hemos utilizado la plataforma ROACH v1 (fig. 3.2). Esta plataforma está equipada con una FPGA fabricada por Xilinx, más concretamente la Xilinx Virtex-5 SX95T (fig. 3.3). Las principales características de la plataforma ROACH v1 son las siguientes:

- Un PowerPC 440eEPX integrado en la placa que se encarga de las funciones de control.
- 1 tarjeta Multi-Gigabit para comunicaciones.
- 4× conectores CX4 a 10Gbps.
- 16× GPIO
- 4× SMA IO
- 2× 2M×18-bits QDR II+ SRAMs conectados a la FPGA.
- Una ranura para un módulo de memoria DDR2 DIMM.
- 2 conectores ZDOK a los que van conectados nuestros ADC.
- Puertos de comunicaciones RSD232 DB9, USB 2.0, y una ranura para tarjetas MMC/SD

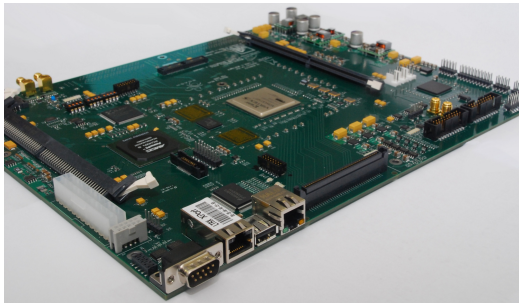

Figura 3.2: Visión trasera de la ROACH

Figura 3.3: Virtex5

En la figura 3.4 vemos el diagrama de bloques de la ROACH v1, en la que podemos apreciar las distintas partes funcionales de la misma.

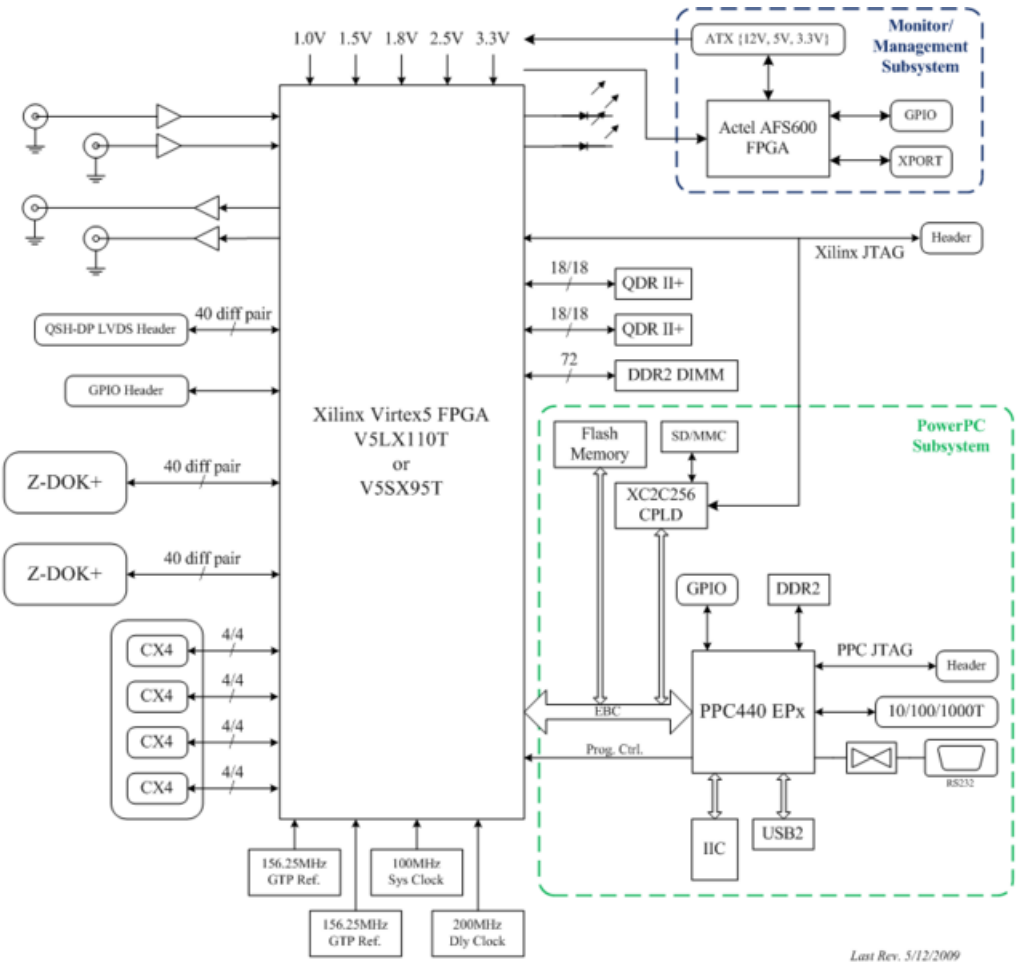

Figura 3.4: Diagrama de bloques de la ROACH

La PowerPC está dotada de una memoria DRAM propia para ejecución de procesos, y está gobernada por una versiõn especial de Linux llamada BORPH (por sus siglas en inglés, Berkeley Operating system for ReProgrammable Hardware). BORPH es un sistema operativo diseñado especialmente para su ejecución en sistemas reconfigurables basados en FPGA's, utilizando éstas como si fueran CPU's. Una de sus principales caracaterísticas es que introduce el concepto de *"hardware process"*, consistente en un diseño de hardware

(usualmente generado con HDL) que corre como un programa de usuario normal.

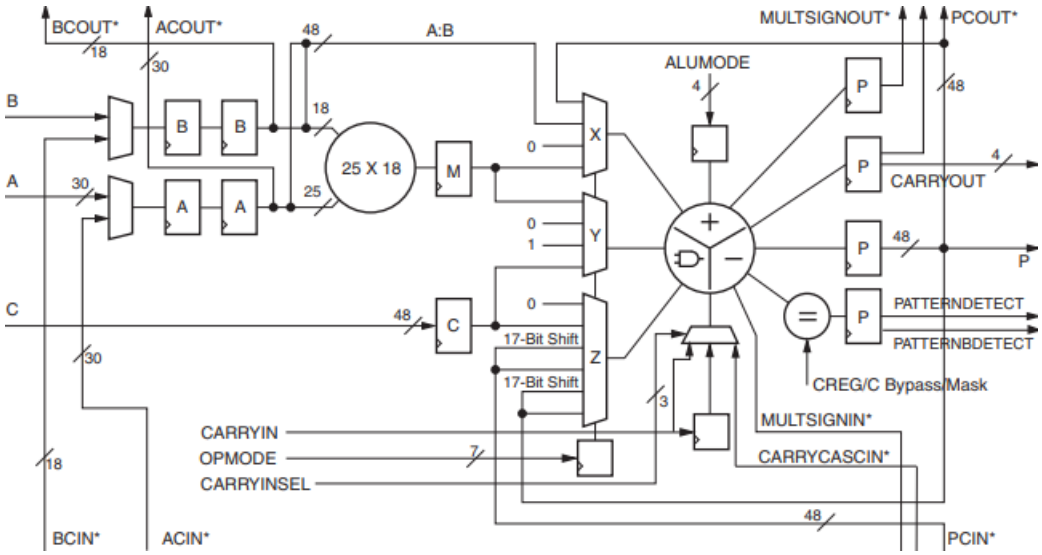

Figura 3.5: Diagrama de bloques de la unidad DSP48E

La potencia de las FPGA Virtex 5 de Xilinx está proporcionada por los recursos que nos ofrece. Según (Xilinx 2015) la unidad básica de memoria del chip consiste en un BRAM que puede ser utilizado para el almacenamiento eficiente de datos, implementación de máquinas de estado, registros, look-up tables y ROMs. Una BRAM puede almacenar hasta 36 kbits y puede ser configurada como dos RAMs independientes de 18 kbits o como una RAM de 36 kbits. Además, posee una unidad de lógica específicamente destinada a la realización de las operaciones DSP, la DSP48E. Con esta unidad lógica podemos implementar funciones tales como multiplicación, multiplicación con acumulación (MACC), suma de hasta 3 entradas, desplazamientos de bits (shifts), multiplexación, comparación de magnitud y otras. La Virtex 5 está diseñada para utilizar múltiples DSP48E en cascada, permitiendo implementar funciones matemáticas complejas, filtros y aritmética de número complejos de manera dedicada, sin recurrir a la lógica de propósito general de la FPGA, lo que supone una mejora de rendimiento importante. En la figura 3.5 podemos ver el diagrama de bloques de esta unidad.

3.1.3. Conversor analógico digital ADC1x3000-8

El conversor analógico digital (ADC por sus siglas en inglés, Analog to Digital Converter) es, como su nombre indica, el dispositivo físico encargado de muestrear la señal

analógica y convertirla a una serie digital de bits para su posterior procesado.

La tarjeta ADC escogida ha sido la ADC1x3000-8 (fig 3.6). Esta tarjeta tiene tres entradas: la de la señal, la del reloj y la de sincronización; ésta última es utilizada cuando tenemos dos tarjetas ADC1x3000-8 trabajando de forma simultánea, que no es nuestro caso. La frecuencia máxima para conversión es 1.5 GHz, mientras que la mínima es 500 MHz (Texas Instrument 2009); con esta configuración, obtenemos 3 gigamuestras cuando establecemos la frecuencia de muestreo a 1.5 GHz y 1 gigamuestras por segundo cuando establecemos la frecuencia de muestreo a 500 MHz; cada una de las muestras es de 8 bits. De estas características es de donde se deriva el nombre de la tarjeta, ADC1x3000-8:

- **ADC**. Analog to Digital Converter.
- **1**. 1 señal de entrada.
- **3000**. Hasta 3000 megamuestras por segundo.
- **8**. Cada muestra es de 8 bits.

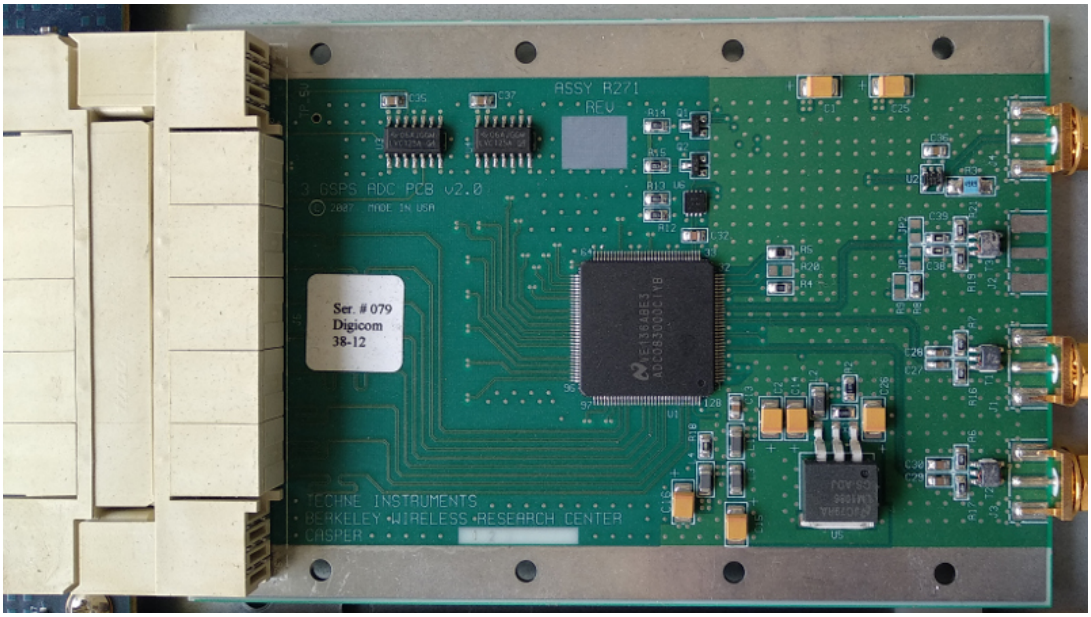

Figura 3.6: Tarjeta ADC1x3000-8. A la izquierda, el conector ZDOK; a la derecha, las entradas de reloj, señal y sincronización, con SMA

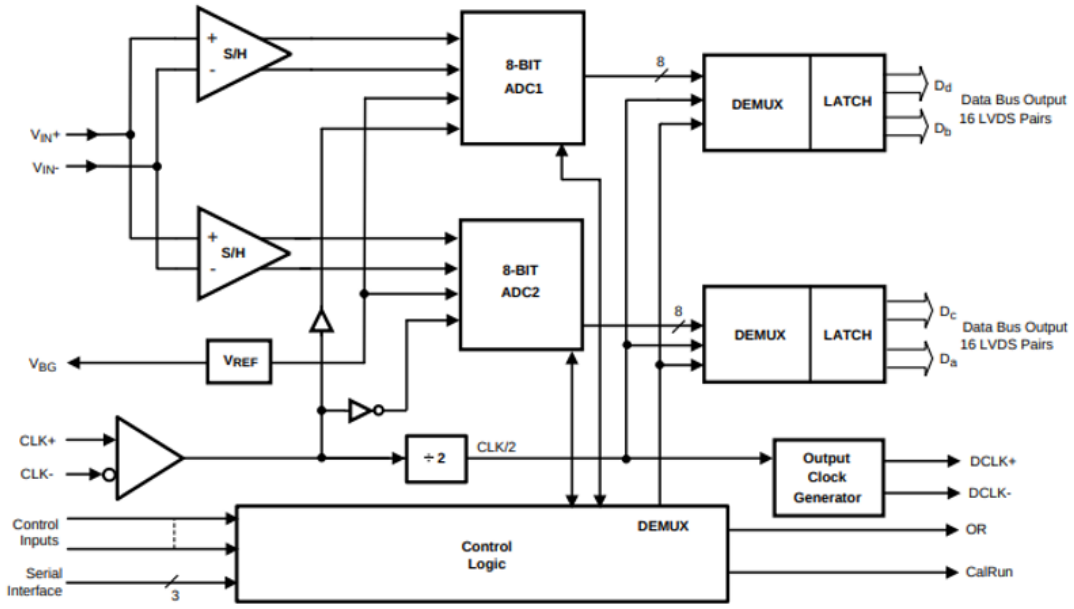

Figura 3.7: Diagrama de bloque de la tarjeta ADC1x3000-8

La alta capacidad de muestreo del ADC es obtenida gracias a que en cada clock se obtienen 2 lecturas, 1 en la subida y 1 en la bajada. De esta forma, cuando operamos a la frecuencia máxima permitida de 1.5 GHz, obtenemos las 3 gigamuestras por segundo (Texas Instrument 2009). Los 8 bits de cuantificación nos proporcionan 48 dB de rango dinámico (ver 2.1.2).

3.1.4. Valon 5009. Frecuencia de muestreo

El ancho de banda con el que va a trabajar el espectrómetro viene determinado por la frecuencia de muestreo que utilicemos, quedando establecido a la mitad de dicha frecuencia de muestreo (ver 2.1.3). Este ancho de banda, combinado con el número de canales que estemos utilizando, determinará la resolución espectral de dicho espectrómetro. Debemos recordar que un objetivo en este proyecto es obtener una alta resolución espectral. Esto lo podemos hacer de dos formas: disminuyendo nuestro ancho de banda y/o incrementando el número de canales. En nuestro caso, el número máximo de canales es un parámetro que queda fijado durante el diseño a 2^{14}. Partiendo de esta premisa, la mejor resolución espectral que podremos obtener es la que viene dada por el menor ancho de banda posible. Si tenemos en cuenta que según Texas Instrument (2009) la frecuencia mínima de muestreo es de 500 MHz, el ancho de banda mínimo es 250 MHz.

No obstante, por medio de filtros analógicos se puede disminuir el ancho de banda que entra al ADC. Esto provocaría un sobremuestreo de la señal, pero permitiría incrementar la resolución espectral. La única limitación en este caso sería la del ancho de banda de los filtros disponibles.

El Valon 5009 es el reloj que nos va a permitir fijar la frecuencia de muestreo de nuestro ADC, estableciendo de esta forma el ancho de banda del espectrómetro y determinando nuestra resolución espectral. Este generador de senáles genera una señal de referencia de alta calidad, con frecuencias que van entre los 25 MHz y los 6000 MHz. Para el control de este dispositivo hemos implementado una pequeña librería en python, que será explicada más adelante en la sección de software, y cuyo código está incluido en el anexo.

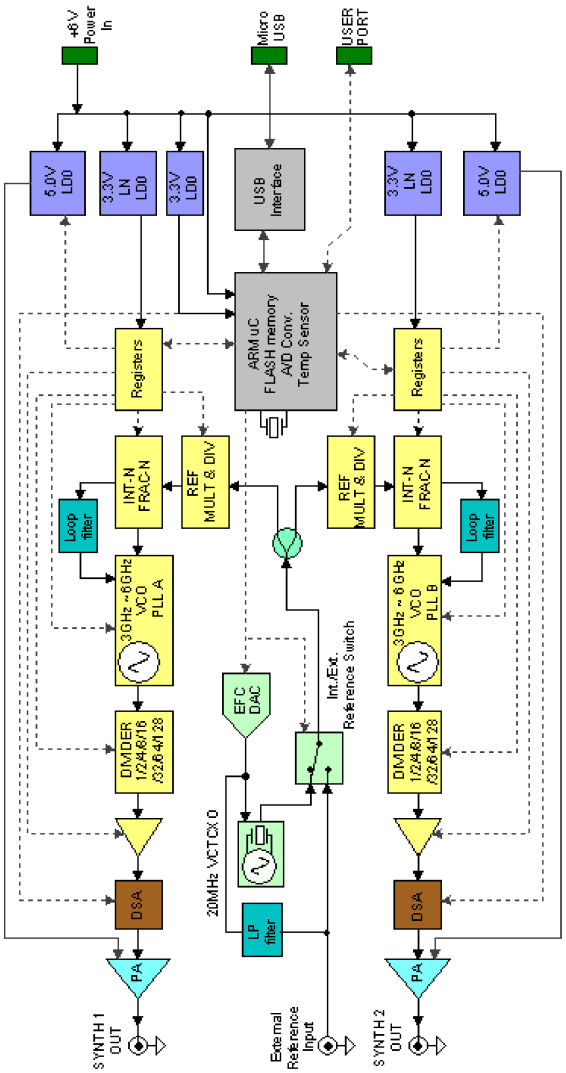

Figura 3.8: Bloque de Valon 5009

Figura 3.9: Valon 5009

3.2. Software de Diseño

A la hora de la programación debemos diferenciar entre dos grandes bloques. El primero de ellos corresponde al diseño de los bloques que se definirán en la FPGA para la programación y configuración del hardware. El segundo hace referencia a la obtención y visualización de los espectros. En esta sección presentaremos el primero de estos bloques.

3.2.1. Modelo Simulink

Para la programación de los FPGA's se utilizan lenguajes conocidos como HDL (Hardware Description Language), siendo algunos de los más conocidos **VHDL, Verilog y ABEL**. Con estos lenguajes podemos definir la estructura, diseño y operación de circuitos electrónicos digitales, haciendo posible una descripción formal de dichos circuitos y posibilitando su análisis automático y su simulación. Una vez se ha obtenido el comportamiento deseado en el simulador, se puede generar un ejecutable que, una vez instalado en el hardware, consigue el comportamiento deseado. En nuestro caso, hemos optado por la utilización de VHDL, usando como framework de programación MATLAB y Simulink,(The MathWorks, Inc. 1994-2018) junto con las herramientas proporcionadas por Xilinx y CASPER.

En la figura 3.10 se pueden apreciar 3 diagramas de bloques que señalan los distintos componentes del espectrómetro. En el cuadro etiquetado como A se aprecia la distinción

lógica de cuatro bloques, correspondientes a Control, Muestreo, Espectro y Volcado, cada uno con un color. En el cuadro etiquetado como B aparecen los distintos componentes, así como las librerías que hemos usado, cada una con el color de fondo correspondiente al bloque lógico del recuadro A. En el recuadro etiquetado como C aparece nuestro modelo de Simulink, dividido en los cuatro bloques con el color de fondo correspondiente. Este último recuadro lo podemos ver ampliado en la figura 3.11.

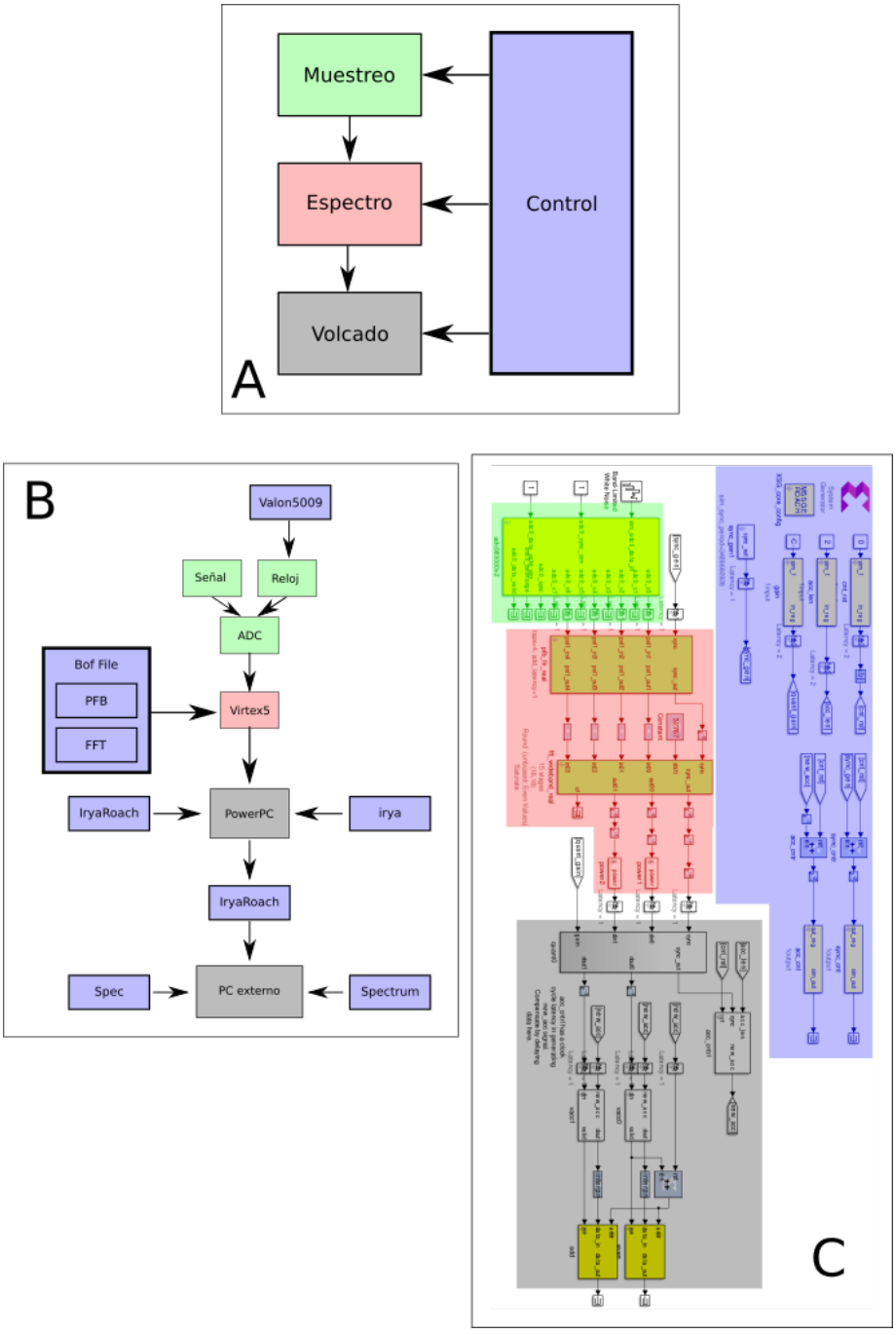

Figura 3.10: Diseño de bloque del espectrómetro

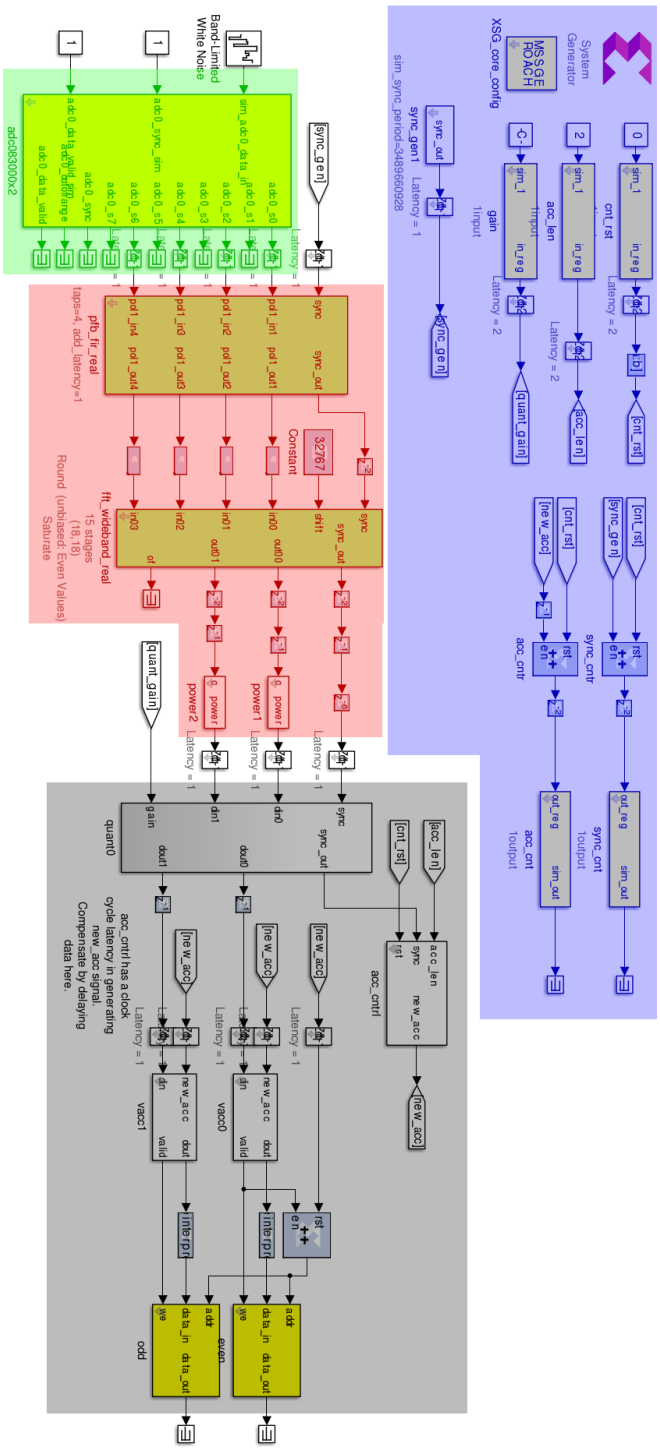

Figura 3.11: Diseño Simulink

3.2.1.1. Bloque de control

Este bloque es el utilizado para definir los parámetros básicos de nuestro modelo, así como el modelo hardware que vamos a utilizar. Permite la definición en tiempo de ejecución de los parámetros que explicamos a continuación. En el caso de este proyecto, algunos de estos parámetros son indicados en tiempo de ejecución a través de la pipeline y otros deben ser indicados en tiempo de compilación a través del GUI de Simulink.

- **XSG_core_config**. Parámetros de configuración para el hardware, tales como el tipo de ROACH, fuente del reloj para la FPGA o frecuencia de muestreo (fig. 3.14).
- **System Generator**. Indica parámetros de compilación (fig. 3.15).
- **sync_gen1**. Registra los clocks y compara con el timing de sincronización, estableciendo el período de sincronización del sistema. Para calcular nuestro período de sincronización nos hemos basado en la documentación del proyecto CASPER (Chen et al. 2008) (fig. 3.13).
- **gain**. Establece la ganancia que se va a aplicar en el momento de la cuantización.
- **cnt_rst**. Nos va a permitir resetear los contadores y reiniciar el proceso de adquisición de espectros.
- **acc_len**. Define el número de espectros a acumular en la FPGA antes de volcar los datos. Una vez se alcanza este número de espectros en la FPGA, el acumulado de estos espectros inidividuales es volcado a la salida de la FPGA. Este espectro, resultado de acumular *acc_len* espectros, es el que la pipeline va a poder leer. Este parámetro va a definir la resolución temporal del espectrómetro, y es configurable como veremos posteriormente en los scripts de la pipeline.
- **sync_cnt**. Contador para llevar un control del período de sincronización.
- **acc_cnt**. Número de espectros volcados. Cuando se alcanza el número de espectros individuales indicados por *acc_len*, el registro new_acc se pone a 1 durante un clock, activando la entrada de enable del sumador acc_cntr e incrementando en 1 el valor del registro *acc_cnt*.

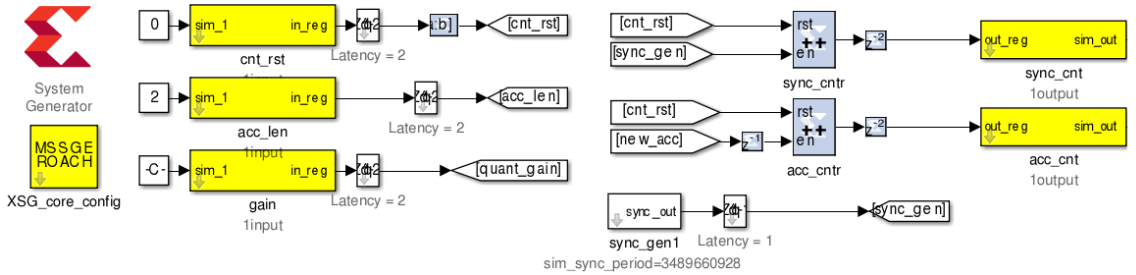

Figura 3.12: Bloque de Control

En la figura 3.12 se muestra el bloque de control del espectrómetro. Cada vez que se inicia una nueva adquisición de datos, se deben ejecutar una serie de acciones que describimos a continuación.

1. Se debe escribir en el registro *acc_len* el número de espectros individuales que la FPGA debe acumular antes de volcar el espectro acumulado en los registros de salida.

2. Se debe escribir en el registro *gain* la ganancia digital que la FPGA va a aplicar.

3. El *cnt_rst* debe ponerse a 1, para indicar que se debe poner el *acc_cnt* a 0. En el siguiente clock el valor se debe volver a poner a 0 para que no se vuelva resetear.

Una vez comienza el proceso, el bloque *sync_gen1* va registrando el número de clocks que van transcurriendo, y una vez alcanza el valor calculado para el timing del sistema, pone a 1 su salida e incrementa el valor de *sync_cnt*. El registro *new_acc* se pone a 1 cada vez que un espectro individual es calculado, acumulando *acc_len* espectros individuales.

Los bloques *sync_gen1*, System Generator y XSG_core_config de la figura 3.12 hacen referenia a parámetros que se deben definir en tiempo de compilación. Para definirlos, se utiliza el GUI de Simulink.

En la figura 3.13 se ven los parámetros que se utilizan para calcular el período de sincronización del sistema. El algoritmo utilizado para este cálculo se detalla en Chen et al. (2008). Los parámetros más relevantes para el diseño del espectrómetro los detallaremos a continuación.

- **FFT Size**. Número de puntos utilizados para el cálculo de la FFT. Este valor debe ser dos veces el número de canales que se desea obtener en el espectro (ver 2.1.8). En este caso tenemos 2^{15}, ya que el espectrómetro tendrá 2^{14} canales.

- **Simultaneous Inputs (FFT)**. Se debe indicar el número de entradas que tiene el bloque de FFT. En este caso son 4, que es el número de salidas habilitadas del ADC.

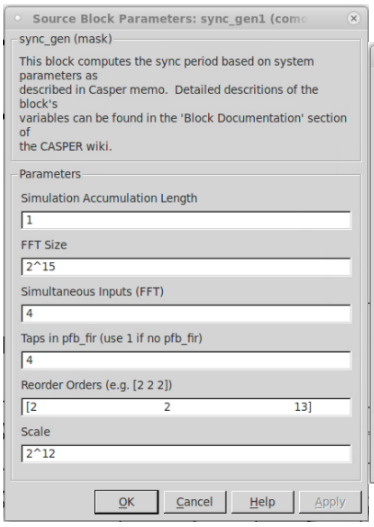

Figura 3.13: Timing

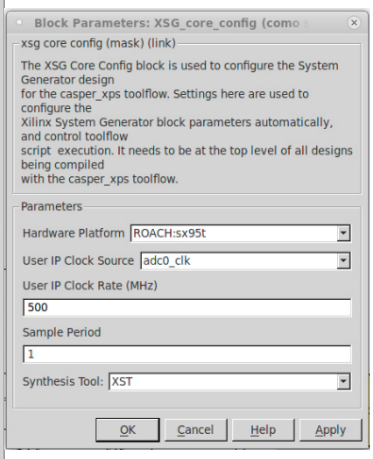

Figura 3.14: Hardware

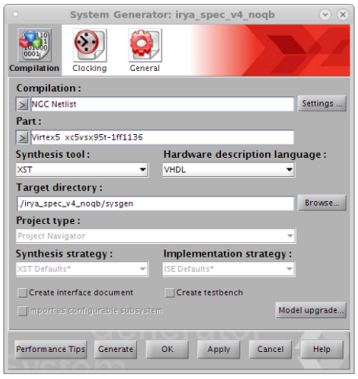

Figura 3.15: Compilación

- **Taps in pfb_fir**. Coeficientes del filtro. Incrementar el número de coeficientes empleados en el PFB incrementa el costo computacional.

En la figura 3.14 se ve la definición de los parámetros relativos al hardware que vamos a utilizar.

- **Hardware Platform**. Se indica la plataforma que se está utilizando.
- **User IP Clock Source**. Se especifica la fuente de la señal que va a servir para la sincronización del sistema.
- **Sample Period**. Este parámetro no tiene importancia porque únicamente hace afecta a las simulaciones.

En la figura 3.15 se ven los parámetros que indican el lenguaje y el compilador utilizados.

3.2.1.2. Muestreo

El bloque de muestreo es el bloque conformado por la tarjeta ADC, encargada de muestrear la señal (fig. 3.16). El bloque amarillo representa la tarjeta ADC, mientras que entre los bloques blancos podemos distinguir dos tipos: los destacados con el marco verde establecen delays que permiten una mejor sincronización del sistema, mientras que los

destacados con el marco rojo son terminales, que indican que esa línea no se va a utilizar en el resto del proceso. Cada una de las líneas adc0_sX saca una muestra de 8 bits. El motivo por el que se descartan estas líneas se verá en párrafos posteriores.

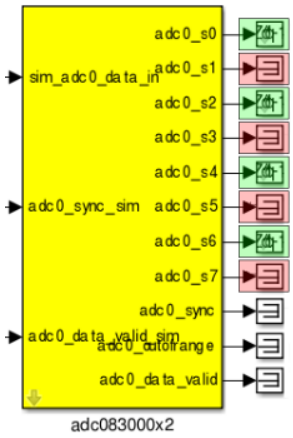

Figura 3.16: Bloque de muestreo

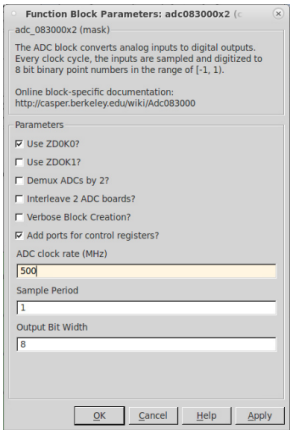

Figura 3.17: Configuración ADC

Algunos de los parámetros de configuración del ADC deben ser introducidos en tiempo de compilación, para lo que se utiliza el GUI de Simulink (figura 3.17). Entre estos parámetros de configuración cabe destacar el que hace referencia a la frecuencia del reloj, *ADC clock rate*. Este parámetro indica la tasa de muestreo a la que va a funcionar el ADC, pero es solo a efectos de simulación. Tal y como hemos explicado en la sección 3.1.3, el número de muestras leídas por nuestro ADC es el doble de la tasa de muestreo, ya que en cada clock se capturan 2 muestras.

Para este espectrómetro se ha establecido como objetivo una resolución espectral de 15 kHz por canal. La frecuencia mínima de operación de nuestro ADC es de 500 MHz, por lo que teniendo en cuenta lo explicado en el párrafo anterior vamos a obtener 1 megamuestras por segundo, lo que define un ancho de banda de 500 MHz. Sin embargo, dado que el espectrómetro tiene 2^{14} canales (los costes computacionales desaconsejan elevar el número de canales), estos 500 MHz divididos en 2^{14} da una resolución de 30 kHz. Para conseguir la resolución deseada, el ancho de banda debe ser fijado a 250 MHz, lo que establece una frecuencia de muestreo de 500 megamuestras por segundo; debido a las limitaciones de operación de nuestro reloj, para llegar a la tasa de muestreo deseada se han silenciado la mitad de las entradas, con lo que obtenemos la mitad de muestras y reducimos el ancho de banda a la mitad.

3.2.1.3. Generación del espectro

Este es el bloque en el que se concentra el procesamiento de la señal. Corresponde a los dos bloques verdes más grandes (PFB y FFT) y a dos bloques blancos (*power*) que calculan la potencia de un número complejo (fig. 3.18), y son los responsables de la obtención de los espectros.

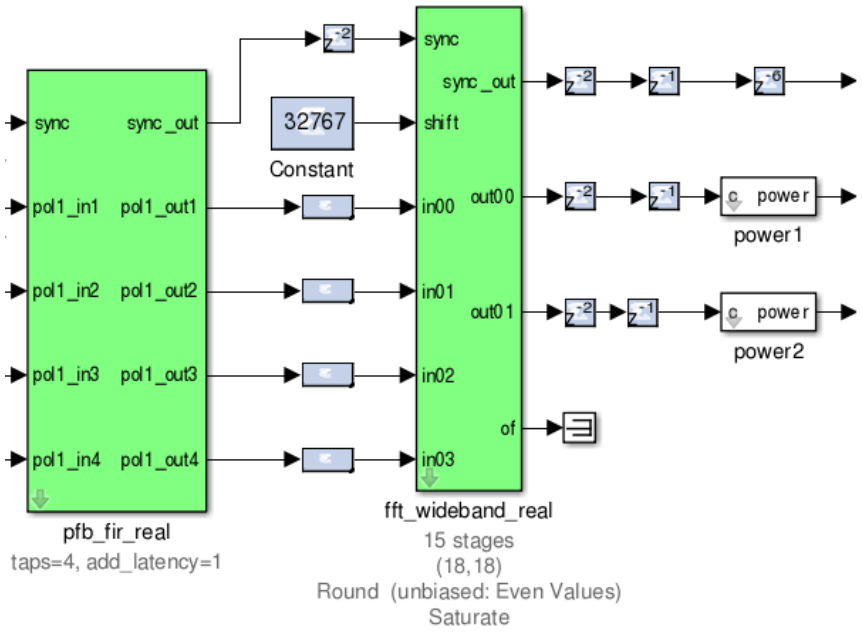

Figura 3.18: Filtros y FFT

El primero de los bloques verdes corresponde al filtro polifásico. El proceso de la transformada discreta de Fourier (DFT) provoca un fenómeno conocido como *leakage*, consistente en que parte de la potencia presente en un canal determinado se filtra hacia los canales adyacentes, generando ruido; en la figura **??** podemos ver dicho efecto. La técnica de los filtros polifásicos atenúa este fenómeno, tal y como vemos en la figura **??**.

El segundo de los bloques es el del FFT; este bloque obtiene 2 canales (las salidas etiquetadas como *out0* y *out1*) por clock de reloj, de forma que en el primer clock obtenemos los canales 0 y 1, en el segundo los canales 2 y 3 y así sucesivamente; estas salidas son recogidas por los bloques *power*, que calculan el cuadrado de la amplitud

calculada por la FFT. De esta forma se determina la potencia, ya que la potencia es proporcional al cuadrado de la amplitud de la FFT.

3.2.1.4. Volcado de datos

Este bloque (fig. 3.19) es el responsable de volcar los datos del espectro en los registros de salida que después leeremos con nuestros scripts de la pipeline.

El primer bloque gris, etiquetado como *quant0*, se encarga de cuantizar los valores obtenidos para la potencia en valores sin signo para conseguir un ahorro en los recursos de memoria.

El bloque etiquetado como *acc_cntrl* es el responsable de emitir la señal de acumulación; cuando se pone a 1, el acumulado de los espectros es volcado a los registros de salida para que nuestros scripts de control puedan leeros. El período de acumulación viene definido por la variable *acc_len*, que determina el número de espectros a acumular. El sistema tarda N ciclos de reloj en acumular un espectro, por lo que desde nuestro script de control podemos controlar el tiempo de acumulación estableciendo el valor de *acc_len* a un valor determinado, siguiendo la fórmula

$$acc\_len = \frac{t_s \cdot f_s}{N}$$

El acumulado se calcula en los módulos etiquetados como *vacc0* y *vacc1*.

Por último, los bloques amarillos etiquetados como *even* y *odd* representan los registros de salida. Tal y como se comenta en la sección 3.2.1.3, el bloque que calcula la FFT devuelve en cada clock 2 canales. Para almacenar cada uno de estos dos canales es necesario la utilización de un registro. Estos canales son procesados posteriormente, intercalándolos en la posición adecuada para obetener el espectro correctamente.

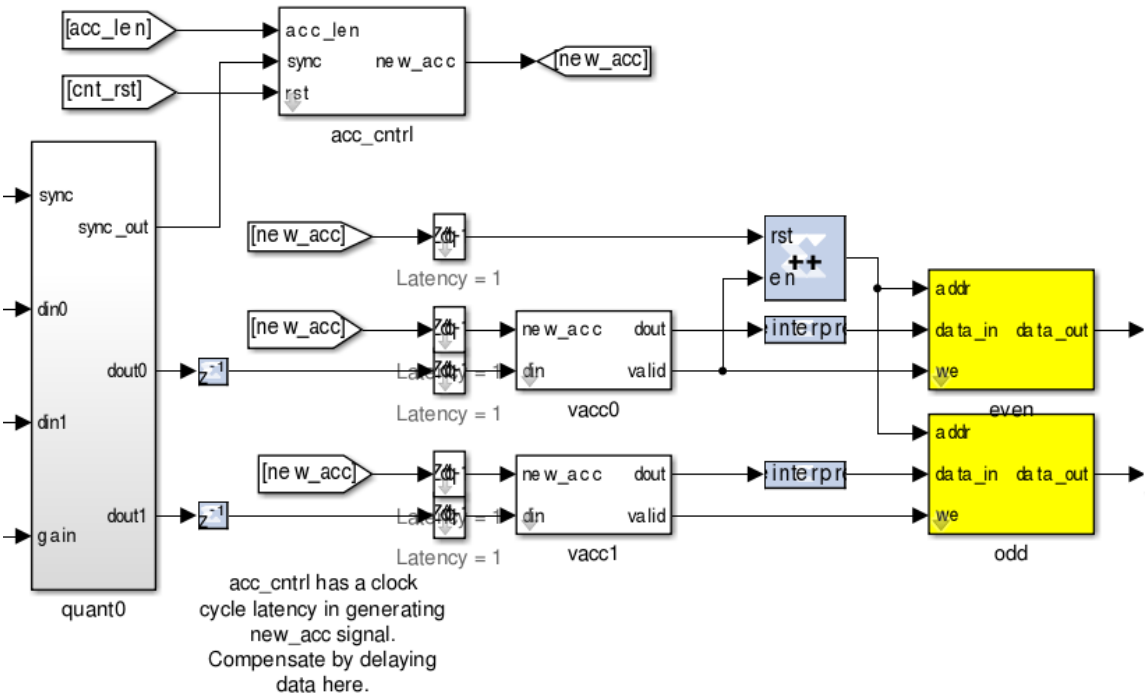

Figura 3.19: Volcado de datos

3.3. Pipeline

En esta sección se va a presentar la pipeline que procesa los espectros obtenidos por el espectrómetro. La pipeline de este espectrómetro se ha desarrollado pensando en utilizarla en línea de comandos, con el objetivo de que sea fácilmente integrable en scripts de automatización. A través de esta pipeline el usuario debe inicializar los parámetros necesarios para comenzar con la adquisición de datos (ver 3.2.1.1) y podrá obtener el espectro. Para la programación de la pipeline hemos escogido Python como lenguaje, aprovechando librerías ya escritas como **casperfpga**. Hemos desarrollado dos librerías y un script, cuyo código adjuntamos en el Anexo.

- **casperfpga.** No ha sido desarrollado por nosotros, pero dada su importancia en el funcionamiento general del espectrómetro, hemos considerado importante destacarla aquí. Tal y como reza el README del repositorio GIT, "casperfpga es una librería de python usada para interactuar con el Hardware de CASPER. Las funcionalidades incluidas son capaces de reconfigurar el firmware, así como de leer y escribir registros

utilizando para ello varios protocolos de comunicación" (SKA Africa 2014-2018).

- **valon5009.** Es una librería que hemos desarrollado para facilitar la comunicación con el generador de señales Valon 5009. Está basada en la librería pyserial, pensada para permitir la comunicación con puertos serie, e implementa diversos comandos de terminal de los disponibles en (Valon Technology 2017), siendo las funciones más destacadas *get_freq* y *set_freq*.
- **irya_libs.** Contiene las clases desarrolladas para este proyecto. Son tres: IryaRoach, para la comunicación con la ROACH; Spec, para la definición del espectrómetro; y Spectrum, para el manejo de los ficheros generados por el espectrómetro.
- **irya.** Es el script de control que arranca el espectrómetro; recibe varios parámetros: *time*, que especifica el tiempo de integración; *bandwidth*, para especificar el ancho de banda; *gain*, especificando la ganancia; *channels*, para definir el número de canales; y *bof*, que permite especificar cuál es el modelo VHDL cargado en la FPGA.

3.3.1. valon5009.py

Esta librería ha sido desarrollada para poder programar el generador de señales Valon 5009. Este generador ofrece una interfaz gráfica para Windows para su programación, así como una herramienta desde línea de comandos, también para Windows. Dado que este poyecto se ha desarrollado bajo Linux, nos hemos visto en la necesidad de impementar esta librería.

3.3.1.1. Valon5009

La librería valon5009 contiene una única clase, llamada Valon5009. Esta clase contiene 2 constantes, un constructor y 8 métodos de clase. Las variables de clase son SRC_1 y SRC_2, y hacen referencia a las dos señales que podemos generar con el Valon 5009. A continuación explicaremos el constructor y los métodos de clase.

- **__init__**. Tiene un parámetro de entrada obligatorio, *port*, en el que se debe especificar el puero al que está conectado el Valon 5009. Inicializa una instancia de la clase Valon5009, estableciendo una conexión serial.
- **send_command**. Envía un comando al Valon 5009 utilizando la conexión establecida.
- **set_freq**. Establece la frecuencia a la que se debe generar una de las señales. Tiene dos parámetros de entrada obligatorios, *source* y *freq*. El parámetro *source* indica

cuál de las dos señales va a ser modificada, y solo puede tomar dos valores, 1 y 2. El parámetro *freq* indica la frecuencia en MHz que se quiere establecer; el rango de valores permitidos está 25 y 6000, aceptando valores decimales. Este método devuelve un valor booleando: True si no ha habido problemas, False si se ha detectado alguna excepción.

- **get_freq**. Devuelve la frecuencia o frecuencias establecidas. Tiene dos parámetros opcionales, *source* y *verbose*. El parámetro *source* indica la señal cuya frecuencia se quiere obtener, pudiendo tomar los dos valores numéricos 1 y 2; el parámetro *verbose* establece si se debe mostrar por salida estándar o no los valores. Si se indica el parámetro *source*, el método devuelve un valor decimal con la frecuencia en MHz a la que se está emitiendo el tono indicado. Si no se indica el parámetro *source* el método devuelve una tupla de dos elementos, compuesta por los valores decimales en MHz de los dos tonos.

- **set_pow**. Establece el nivel de voltaje a la que se debe generar una de las señales. Tiene dos parámetros de entrada obligatorios, *source* y *pow*. El parámetro *source* indica cuál de las dos señales va a ser modificada, y solo puede tomar dos valores, 1 y 2. El parámetro *pow* indica el nivel de voltaje que se quiere establecer; los valores permitidos son 0,1,2,3. Este método devuelve el nivel que se ha establecido o -1 en caso de que haya un error.

- **get_pow**. Devuelve el nivel de voltaje establecidos. Tiene dos parámetros opcionales, *source* y *verbose*. El parámetro *source* indica la señal cuyo nivel de voltaje se quiere obtener, pudiendo tomar los dos valores numéricos 1 y 2; el parámetro *verbose* establece si se debe mostrar por salida estándar o no los valores. Si se indica el parámetro *source*, el método devuelve el nivel de voltaje que tiene la señal indicada. Si no se indica el parámetro *source* el método devuelve una tupla de dos elementos, compuesta por los niveles de voltaje de los dos tonos.

- **set_att**. Establece la atenuación que se debe aplicar a una de las señales. Tiene dos parámetros de entrada obligatorios, *source* y *att*. El parámetro *source* indica cuál de las dos señales va a ser modificada, y solo puede tomar dos valores, 1 y 2. El parámetro *att* indica la atenuación que se quiere establecer; los valores permitidos están en el rango de 0 a 32, con un paso de 0.5. Este método devuelve la atenuación que se ha establecido o -1 en caso de que haya un error.

- **get_att**. Devuelve la atenuación establecida. Tiene dos parámetros opcionales, *source* y *verbose*. El parámetro *source* indica la señal cuya atenuación se quiere obtener, pudiendo tomar los dos valores numéricos 1 y 2; el parámetro *verbose* establece si se debe mostrar por salida estándar o no los valores. Si se indica el parámetro *source*, el método devuelve la atenuación que tiene la señal indicada. Si no se indica el

parámetro *source* el método devuelve una tupla de dos elementos, compuesta por las atenuaciones de los dos tonos.

- **save**. No acepta parámetros de entrada y no devuelve ningún valor. Almacena los valores que tenga en ese momento el Valon 5009 como valores de inicializacón para la próxima vez que se reinicie.

3.3.2. irya_libs.py

En la librería **irya_libs.py** hemos implementado las clases que gestionan el espectrómetro y los espectros que se obtienen. Existen dos clases responsables de la comunicación con el espectrómetro, IryaRoach y Spec. Y existe una clase que permite la manipulación de los espectros, Spectrum.

3.3.2.1. IryaRoach

La clase IryaRoach se encarga de la comunicación con la FPGA, proporcionando los métodos necesarios para el acceso a los distintos recursos que la FPGA ofrece. Esta clase tiene un constructor y diez métodos de clase.

- **__init__**. Método constructor. Tiene 2 parámetros obligatorios, *roach* y *katcp_port*. En el parámetro *roach* se debe especificar la dirección de la ROACH; en el parámetro *katcp_port* se debe especificar el puerto que se va a utilizar para enviar los comandos a la Roach. Este método establece la comunicación y determina la versión de la ROACH que estamos usando.

- **listbof**. Método sin parámetros. Devuelve una lista con los ficheros .bof y.fpg (ficheros de definición y reconfiguración de la plataforma) existentes en la ROACH.

- **wait_connected**. Método con un parámetro opcional, *timeout*. Este método deja pasar un tiempo de espera de *timeout* segundos (por defecto es 5 segundos) y entonces verifica si la ROACH está conectada o no. Devuelve True si la ROACH está conectada y False si la ROACH no está conectada.

- **is_running**. Método sin parámetros. Devuelve True si la ROACH está en operación y False en caso contrario.

- **progdev**. Tiene un parámetro obligatorio, que es el nombre del fichero .bof o .fpg que se va a utilizar para reconfigurar la ROACH. Devuelve True si no hay problemas a la hora de la reconfiguración y False en caso de que haya problemas.

- **program**. Este método tiene un parámetro opcional, *filename*. Este método es llamado desde el método **progdev** cuando el fichero de definición es un .bof. Devuelve True si la reconfiguración se ha hecho correctamente y False en caso de que haya habido algún problema durante la reconfiguración.
- **write_int**. Método con 2 parámetros obligatorios, *device_name* y *data*, y 2 métodos opcionales, *offset* y *blindwrite*. Este método escribe el entero indicado en *data* en el registro indicado por *device_name*, en la posición indicada por *offset*. Si el parámetro *blindwrite* es False, verifica que una vez se ha producido la escritura, el contenido del registro coincide con lo esperado.
- **read_int**. Método con un parámetro obligatorio, *device_name*, y un método opcional *offset*. Lee un entero desde el registro *device_name* con el desplazamiento *offset*.
- **read_ram**. Método con un parámetro obligatorio, que es *device*; y con hasta 3 paraámetros opcionales, que son *n_words*, *data_format* y *offset*. Este método lee *n_words* elementos con el formato *data_format* del registro en memoria llamado *device*, comenzando en la posición *offset* .
- **read_full_stream**. Método con un parámetro obligatorio, que es *device_prefix*; y con hasta 3 paraámetros opcionales, que son *n_words*, *data_format* y *offset*. Este método lee *n_words* elementos con el formato *data_format* de todos los registro en memoria cuyo nombre comience por *device_prefix*, comenzando en la posición *offset* de dichos registros.

3.3.2.2. Spec

La clase Spec permite la inicialización del espectrómetro, definiendo los parámetros para su funcionamiento. Esta clase tiene un constructor.

- **__init__**. Método constructor. Tiene un parámetro obligatorio, *roach_address* y 6 parámetros opcionales. Los 6 parámetros opcionales son *katcp_port*, *bitstream*, *channels*, *acc_time*, *gain* y *bw*. Este método configura el espectrómetro, estableciendo la comunicación con el espectrómetro a través de la dirección *roach_address* y el puerto *katcp_port*, cargando el fichero de diseño *bitstream*, estableciendo el número de canales a *channels*, el tiempo de acumulación entre espectros como *acc_time* segundos, fijando la ganancia digital a *gain* y estableciendo el ancha de banda a *bw*.

3.3.2.3. Spectrum

La clase Spectrum permite manipular los ficheros de espectros obtenidos del espectrómetro. Esta clase tiene un constructor y seis métodos de clase.

- **__init__**. Este método tiene 4 parámetros opcionales. El método carga el espectro, ancho de banda, tiempo de integración y número de canales desde el fichero indicado por el parámetro *path*. Si el parámetro *path* no es definido o se define como None, entonces se deben especificar los otros 3 parámetros y lo que se inicializa es un espectro simulado. El parámetro *bw* indica el ancho de banda, el parámetor *channels* indica el número de canales y el parámetro *integ* inidica el tiempo de integración del espectro simulado.

- **show**. Este método tiene 2 parámetros opcionales, *zoom* y *units*. Si se utiliza sin parámetros, muestra el espectro completo con las unidades en MHz. Al parámetro *zoom* se le deben pasar los límites del rango en el que se quiere hacer zoom, límites que pueden estar indicados en canales o en frecuencia. El parámetro *units* cambia las unidades en las que el espectro es visualizado.

- **detections**. Este método tiene un único parámetro opcional, *sigma*. Si este parámetro no es indicado en la llamada, toma un valor por defecto de 3. El método devuelve una lista con los canales cuya potencia es superior a $n \cdot \sigma$, con $n = sigma$.

- **show_detections**. Este método tiene tres parámetros opcionales, que son *sigma*, *zoom* y *show_labels*. Muestra el espectro y marca los canales que tienen una potencia superior a $n \cdot \sigma$, con $n = sigma$. El parámetro *zoom* funciona igual que en el método **show** y el parámetro *show_label* es un booleano que indica si se deben mostar los canales en la leyenda.

- **get_channel**. Este método tiene un parámetro obligatorio, *freq*. Devuele el canal en el que está contenida la frecuencia *freq*.

- **get_harmonic**. Este método tiene un parámetro obligatorio, *freq*. El método devuelve la frecuencia del harmónico correspondiente a la frecuencia *freq*, el canal en el que está contenido dicho harmónico y la potencia de ese canal.

- **clean_noise**. Este método tiene un parámetro opcional, *sigma*; si no recibe este parámetro, le asigna un valor de 3. El método clasifica como ruido todos los canales cuya potencia sea inferior a $n \cdot \sigma$ con $n = sigma$ y los lleva a 0.

3.3.3. irya.py

El script **irya.py** permite el control del espectrómetro, haciendo uso de las clases contenidas en las librerías **irya_libs.py** y **valon5009.py** para configurarlo. El script debe recibir obligatoriamente la dirección de la ROACH, y puede recibir hasta 9 modificadores de opción.

- **-t**. Especifica en segundos el tiempo de integración por espectro.
- **-c**. Especifica el número de canales que tiene el espectrómetro. Este parámetro **NO** configura el espectrómetro, solo se ofrece como información. Debe coincidir con el número de canales que se especifica en tiempo de compilación en el diseño del espectrómetro.
- **-N**. Especifica el número de espectros a leer.
- −−**bw**. Especifica el ancho de banda del espectrómetro.
- **-g**. Especifica la ganancia digital a aplicar.
- **-s**. Si esta opción está presente, no se reconfigura ni se inicializa el espectrómetro.
- **-b**. Especifica el nombre del archivo .bof o .fpg a cargar en la FPGA.
- −−**no-save**. Si esta opción está presente, no se almacenan los datos y solo se observan los espectros en tiempo real.
- **-h**. Muestra la ayuda.

Capítulo 4

Pruebas de Banco

En este capítulo presentaremos una serie de pruebas preliminares que hemos realizado para comprobar el adecuado funcionamiento del espectrómetro, y mostraremos los resultados obtenidos en las mismas. Las pruebas definitivas se realizarán integrando el espectrómetro en un radiotelescopio y realizando observaciones de un astro.

Las pruebas preliminares que hemos hecho en nuestro laboratorio nos han permitido caracterizar el espectrómetro, determinando los parámetros de resolución espectral, rango dinámico y estabilidad.

Para la realización de las pruebas hemos utilizado una antena dipolo de 408 MGHz (figura 4.1), un generador de señales Valon5009 (ver sección 3.3.1.1) para la generación de tonos, un PC y el espectrómetro.

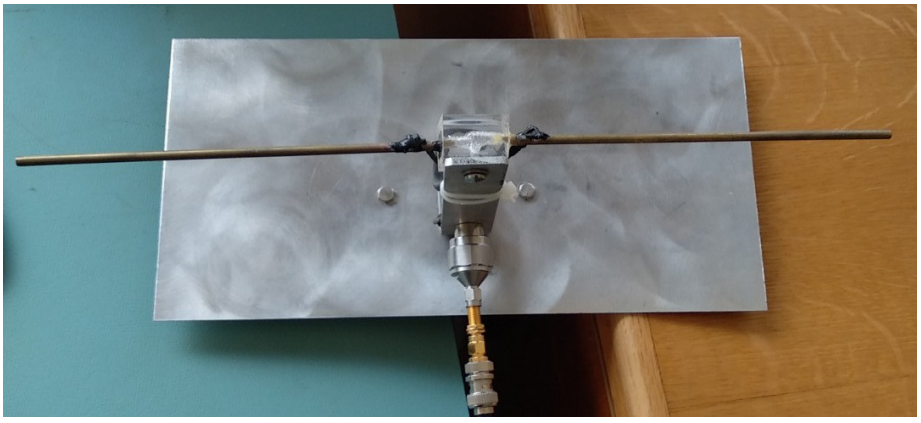

Figura 4.1: Antena Dipolo

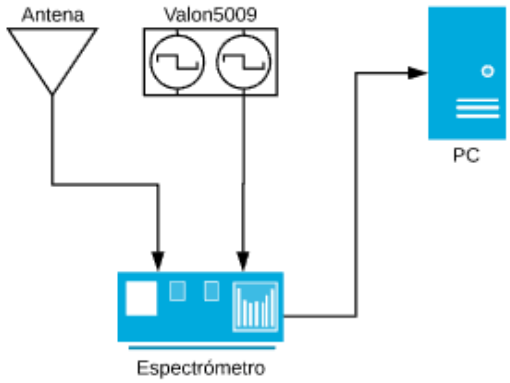

Figura 4.2: Pruebas Banco

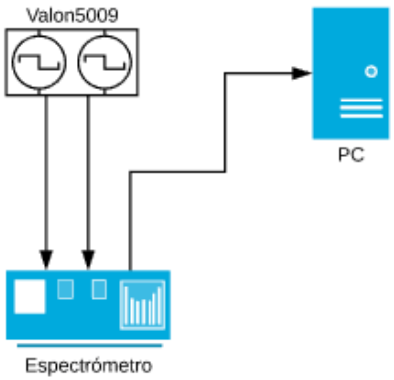

Figura 4.3: Pruebas Campo

4.1. Comportamiento del espectrómetro

Para probar el comportamiento del espectrómetro hemos utilizado el mismo modelo de sintetizador que utilizamos para establecer la frecuencia de muestreo en nuestro ADC, un Valon 5009 (ver sección 3.3.1.1). Este modelo dispone de dos salidas; una de ellas la hemos utilizado para establecer la frecuencia de muestreo, y la otra para conectarla a la entrada del ADC y utilizar el Valon 5009 como generador de tono.

Hemos realizado pruebas en varios rangos de frecuencia, para observar el comportamiento del espectrómetro. Para ello partimos de una frecuencia inicial y corrimos el script durante 10 minutos, cambiando la frecuencia cada 8 segundos y teniendo como paso de frecuencia el ancho de un canal, esto es, 15 kHz. Esto nos da un rango de frecuencias de 1.2 MHz.

A continuación presentamos una tabla con las frecuencias escogidas. Cuando los tonos generados están por encima de nuestra frecuencia de Nyquist (recordemos que es 250 MHz), lo que debemos ver en el espectro es el armónico correspondiente. Cuando nos encontramos en un armónico par, el espectro lo podemos ver reflejado; si por el contrario, el armónico es impar, el espectro es directo.

$\mathbf{F_0}$ **(MHz)**[1]	$\mathbf{F_s}$ **(MHz)**[2]	**BW**[3]	**Ganancia**[4]	**Armónico (MHz)**[5]	**Par/Impar**[6]
53	500	250	0x1f	53	Impar (1)
453	500	250	0x1f	47	Par (2)
1420	500	250	0x8f	80	Par (6)
2153	500	250	0x8f	153	Impar (9)
4903	500	250	0x8fff	97	Par (20)

Cuadro 4.1: Parametrización

(a) [1] Frecuencia del tono. [2] Frecuencia de muestreo. [3] Ancho de banda. [4] Ganancia digital. [5] Frecuencia donde debe aparecer el armónico correspondiente en el espectro. [6] Si se trata de un armónico par o impar.

El espectrómetro se ha configurado para volcar 25 espectros, integrando en cada uno de ellos durante 30 segundos. Una vez tenemos los 25 espectros, sumamos todos los espectros y obtenemos el espectro que mostramos a continuación. El procedimiento utilizado para el test provoca que se detecten picos superiores en el comienzo y en el final del rango de frecuencias que recorremos, ya que el generador de tonos pasa más tiempo en la frecuencia inicial para su inicialización y pasa mayor tiempo en la última de las frecuencias (la diferencia entre los 12 minutos y 30 segundos que está tomando muestras y la suma entre los 10 minutos que está barriendo las frecuencias y el tiempo de inicialización).

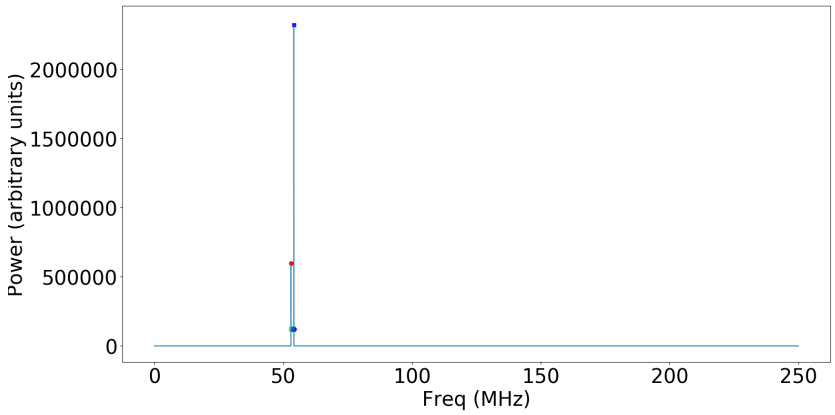

Figura 4.4: Espectro completo con tono a 53 MHz

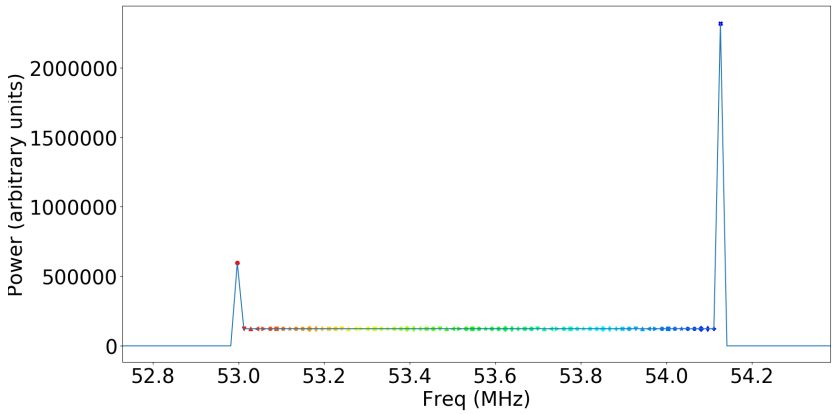

Figura 4.5: Zoom con tono a 53 MHz

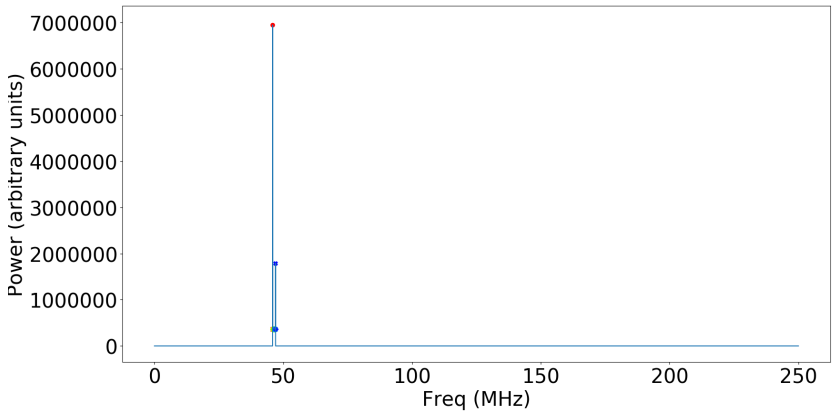

Figura 4.6: Espectro completo con tono a 453 MHz

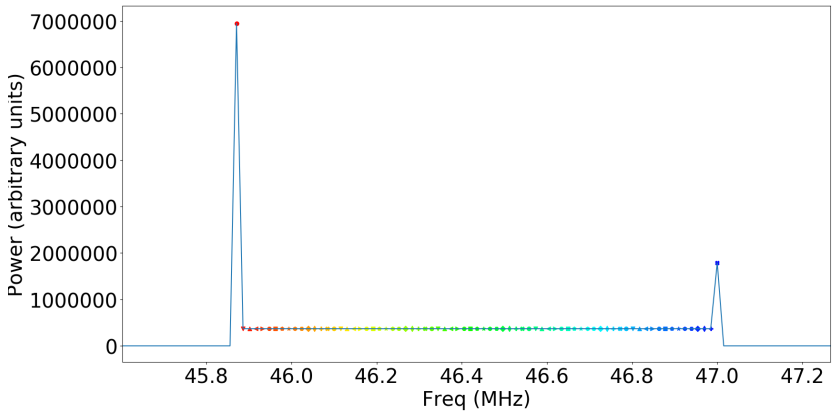

Figura 4.7: Zoom con tono a 453 MHz

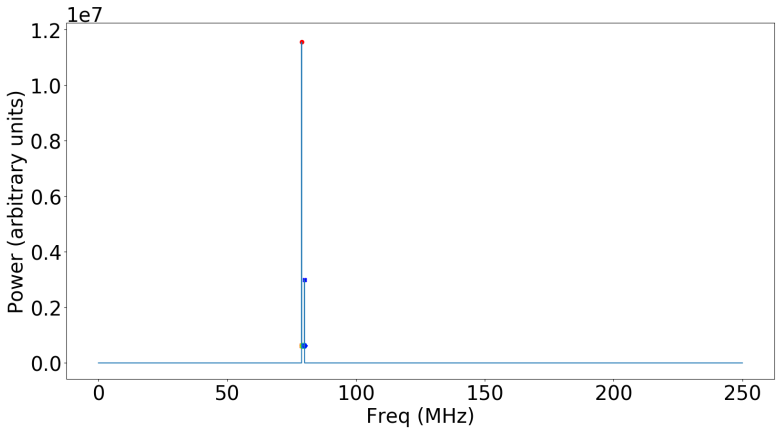

Figura 4.8: Espectro completo con tono a 1420 MHz

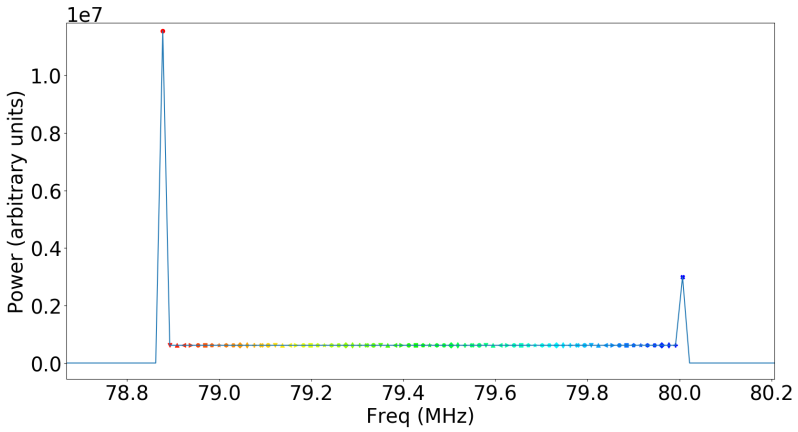

Figura 4.9: Zoom con tono a 1420 MHz

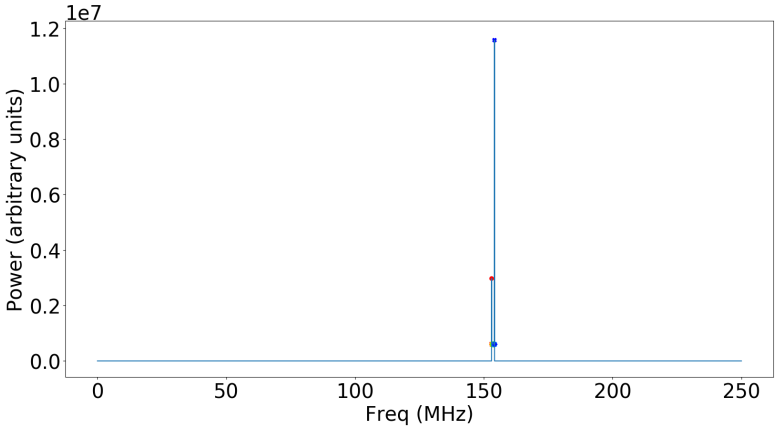

Figura 4.10: Espectro completo con tono a 2153 MHz

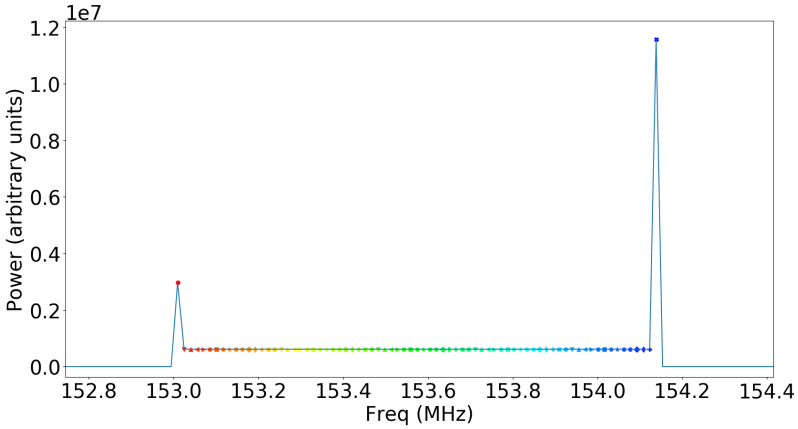

Figura 4.11: Zoom con tono a 2153 MHz

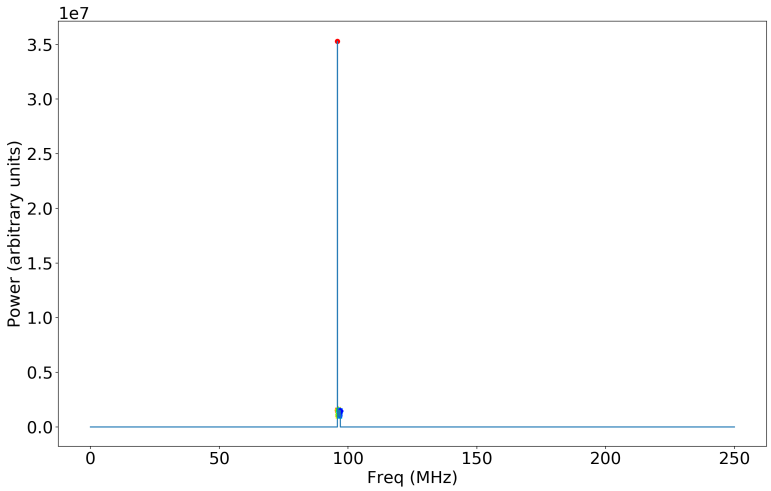

Figura 4.12: Espectro completo con tono a 4903 MHz

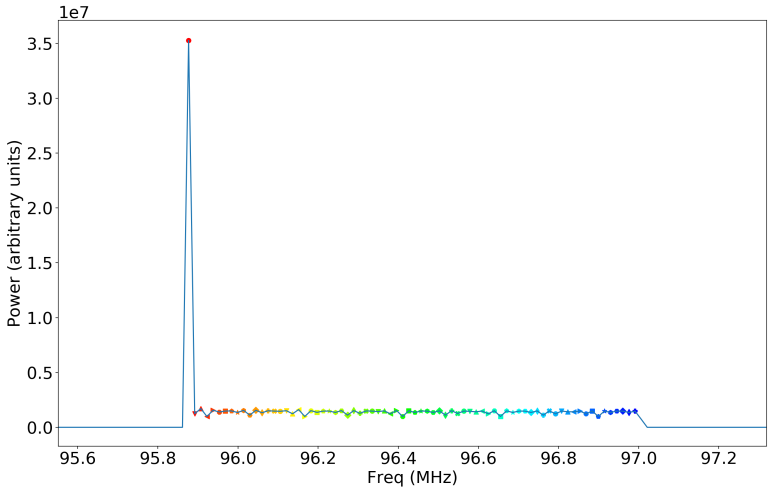

Figura 4.13: Zoom a 4903 MHz

En las figuras 4.5, 4.7, 4.9, 4.11 y 4.13 podemos apreciar como la resolución permite ver cada canal. El hecho de que el procedimiento del test registre sistemáticamente mayor potencia en el último de los canales muestreados nos permite apreciar el efecto de

espectro invertido en los armónicos pares. En las figuras 4.7, 4.9 y 4.13 podemos apreciar el efecto de los armónicos pares y el espectro invertido. En las figuras 4.5 y 4.11 se puede apreciar como los armónicos impares y el tono principal presentan un espectro no invertido.

4.2. SFDR

El rango dinámico libre de espúreos o SFDR (por sus siglas en inglés, Spurious Free Dynamic Range) es el margen que hay entre la potencia de la señal fundamental y la potencia del mayor de los armónicos detectados. En nuestro caso hemos hecho las pruebas sobre 600 espectros de 1 segundo. Durante estas pruebas se ha registrado una potencia de señal de 60 dB y una potencia del armónico de 42 dB, con lo que se obtiene un SDFR $\approx$ 18 dB. En la figura 4.14 podemos ver la potencia en unidades arbitrarias, y en la 4.15 podemos ver la potencia en dB.

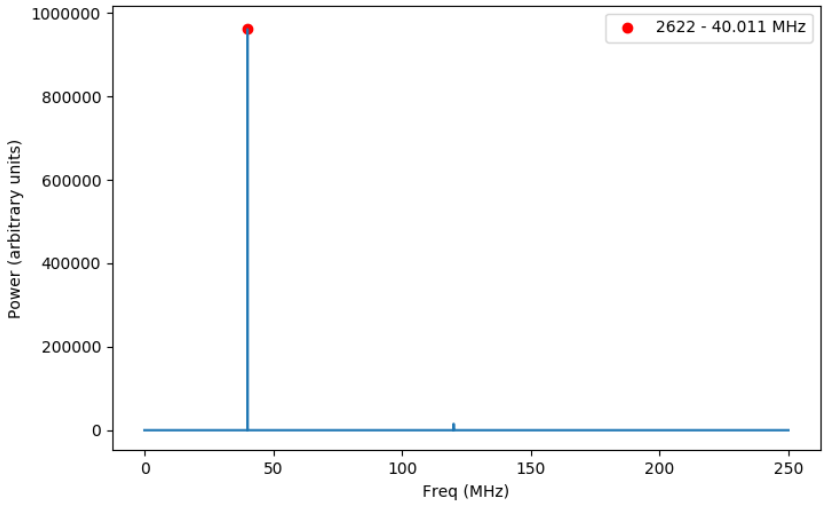

Figura 4.14: Espectro en unidades arbitrarias

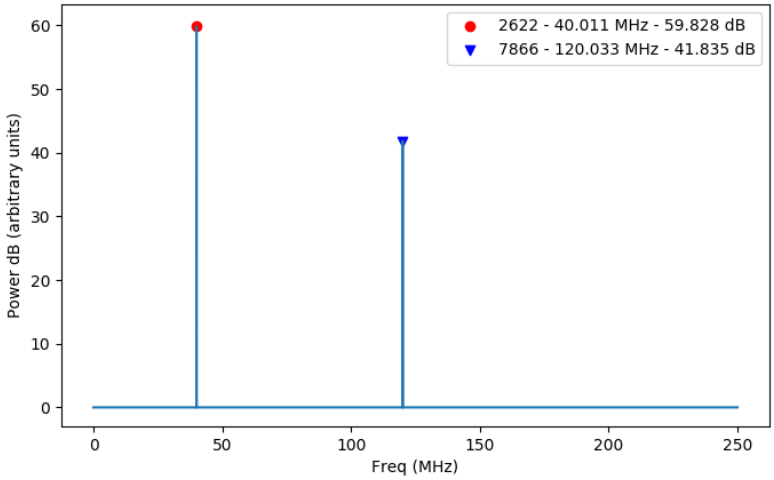

Figura 4.15: Espectro en dB

4.3. Linealidad

La linealidad es la propiedad por la que el espectrómetro registra una mayor cantidad de potencia ante una señal de mayor voltaje. En la figura 4.16 podemos ver los valores que ha registrado el espectrómetro ante un tono de 40 MHz. El voltaje de la señal es la que se presenta en la tabla 4.3.

Voltaje $[mV_{rmm}]$	**Potencia registrada**
35	15258
100	61032
200	259386
300	579804
390	961254

Cuadro 4.3: Potencia Registrada

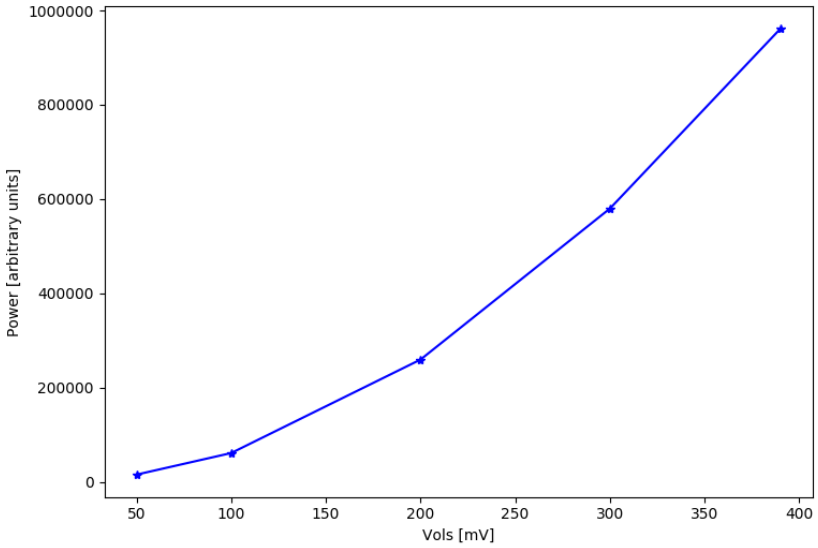

Figura 4.16: Potencia registrada

4.4. Desviación de Allan

Con el fin de obtener una mejor señal a ruido se suelen realizar integraciones durante períodos largos, consiguiendo con esta práctica disminuir la influencia del ruido gausiano inherente a la señal medida. Sin embargo, si integramos durante demasiado tiempo, las contribuciones de otros efectos empiezan a ser relevantes, haciendo que la señal a ruido vuelva a empeorar. Por tanto, la importancia de determinar cuál es el tiempo de integracíon óptimo está clara.

Para determinar este tiempo se recurre normalmente a la medición de la desviación de Allan (Schieder & Kramer 2001). Con esta prueba obtenemos una gráfica en la que podemos distinguir la importancia de los distintos tipos de ruido (Land et al. 2007). El ruido gausiano que conseguimos minimizar gracias a la integración larga viene representado en esta gráfica por una pendiente de $-0{,}5$, mientras que el resto de efectos tienen pendiente ≥ 0, por lo que podemos determinar que el tiempo óptimo de integración será aquel en el que la pendiente de nuestra gráfica deje de ser negativa.

Para el cálculo de la desviación hemos utilizado dos sets de muestras. La figura 4.17

corresponde a la toma de espectros durante 5 horas, tomando un espectro por segundo. Esto nos da un total de 20000 espectros.

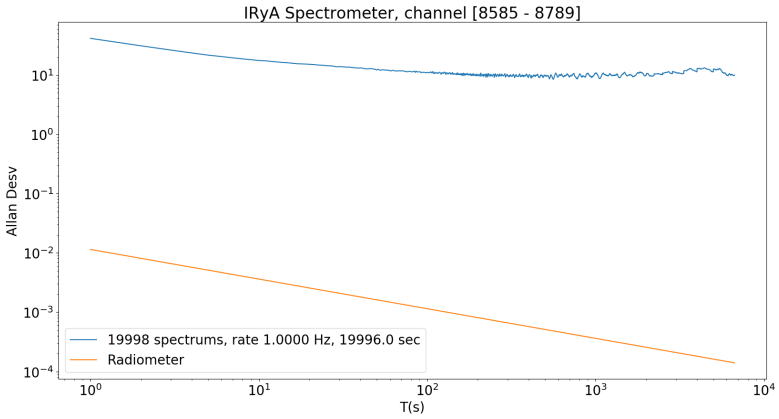

Figura 4.17: Desviación de Allan. 1 espectro por segundo.

La figura 4.18 corresponde a la toma de espectros durante el mismo tiempo que el caso anterior, 5 horas, pero ahora tomando cinco espectros por segundo. Esto nos da un total de 100000 espectros.

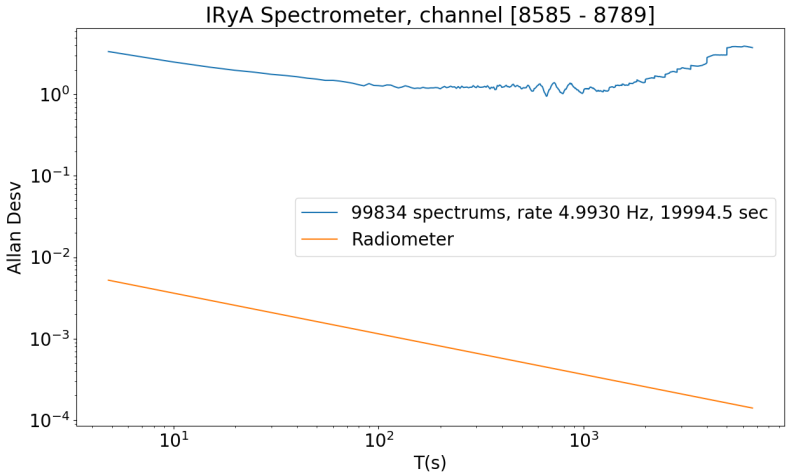

Figura 4.18: Desviación de Allan. 5 espectros por segundo.

En ambos casos podemos ver que el tiempo óptimo es del orden de 200 segundos. En ambos set de pruebas hemos utilizado la antena de 408 MHz, y hemos calculado la desviacı'on en 200 canales, del 8585 al 8785, promediando los resultados. La elección de estos canales ha sido basándonos en que en esos canales no se ha detectado ninguna señal, detectando
'unicamente ruido.

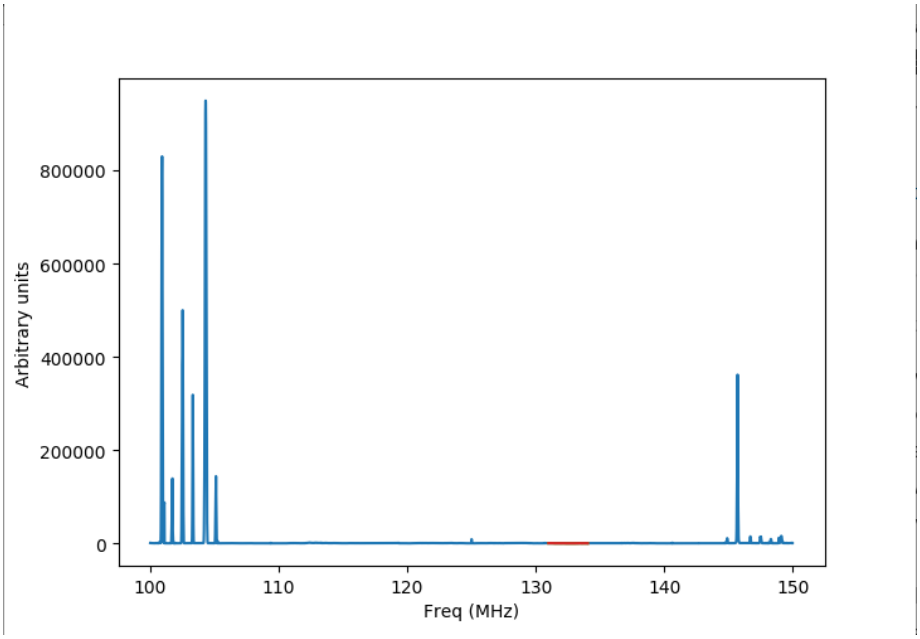

Figura 4.19: Zoom del espectro entre 100 MHz y 150 MHz. En rojo los canales escogidos.

4.5. Prueba de campo

Debido a que hasta el momento de entrega de esta tesis no hemos tenido oportunidad de integrar el espectrómetro en un radiotelescopio, hemos optado por hacer una prueba de campo con una antena diseñada para una frecuencia de 408 MHz. Los resultados han sido que hemos podido registrar con bastante fidelidad el espectro de FM de la zona próxima a nuestro laboratorio.

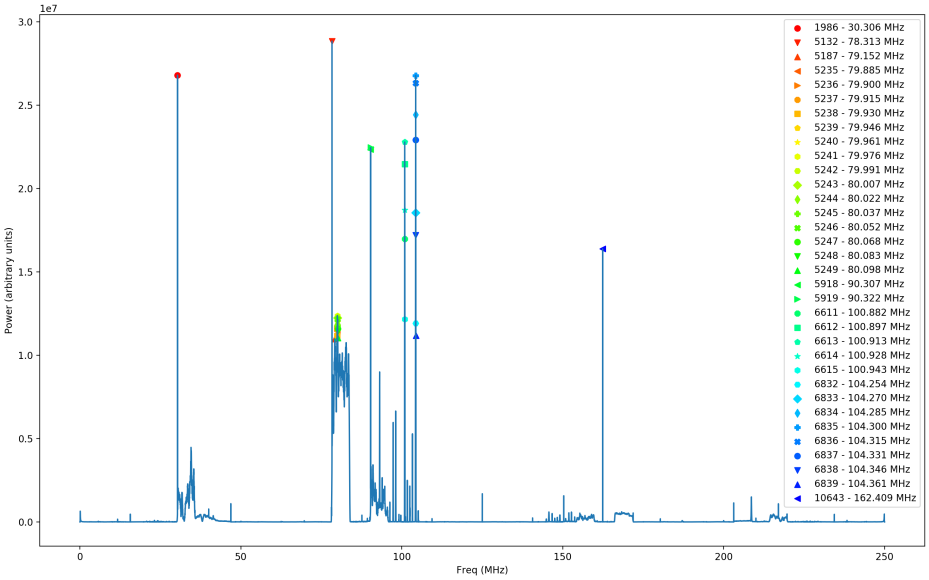

Figura 4.20: Espectro de la señal captada por una antena dipolo

En la figura 4.20 mostramos el espectro detectado, marcando los canales con SNR superior a 7 (valor de conveniencia escogido por nosotros). Los parámetros utilizados para dicha detección son los siguientes. Para ello hemos configurado el espectrómetro con un tiempo de acumulación de 30 segundos.

En las proximidades de nuestro laboratorio sabemos que hay una emisora de radio que emite su señal en la frecuencia 104.3 MHz; podemos ver esta señal reflejada en el espectro, con la potencia de la misma repartida en los canales que van del 6832 al 6839 (8 canales), ocupando un total de $8 \cdot 15{,}25879$ kHz = 122,0703 kHz (recordemos que cada emisora de FM tiene reservada una banda de 200 kHz, debiendo dejar libres los primeros y últimos 25 kHz).

Si filtramos las frecuencias que están por debajo y por encima de la banda de FM (88.1 MHz y 108.1 MHz), y marcamos los canales con SNR mayor a 3, nos queda el siguiente espectro:

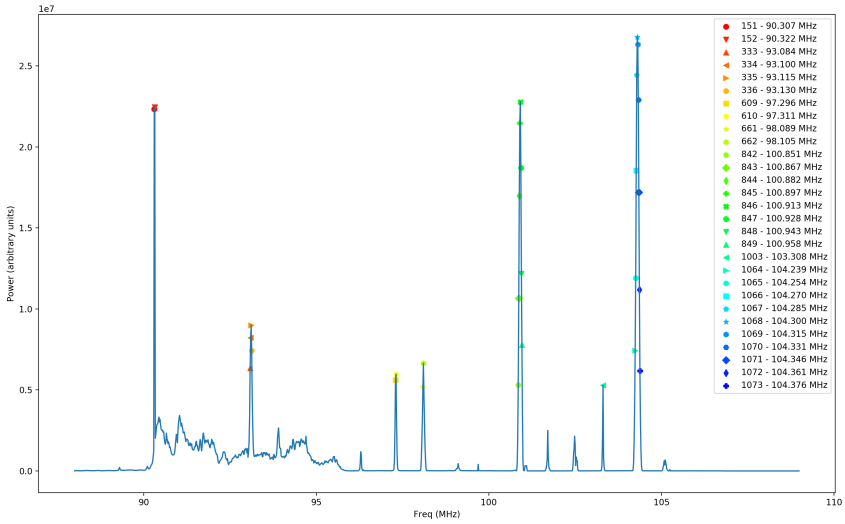

Figura 4.21: Banda FM

En este espectro podemos apreciar, al menos, las siguienes 7 emisoras:

Emisora	**Freq. Nominal (MHz)**	**Canales**	**Rango de Freq (MHz)**
		5918 a 5919 (2)	90.307026 a 90.322285
Grupera 93.1	93.1	6100 a 6103 (4)	92.084295 a 93.130074
Capital FM	97.3	6376 a 6377 (2)	97.295978 a 97.311237
UVE Radio	98.1	6428 a 6429 (2)	98.089483 a 98.104743
Máxima	100.9	6609 a 6616 (8)	100.851492 a 100.958310
Vox 103.3	103.3	6770 (1)	103.308307
Radio Nicolaita	104.3	6831 a 6840 (10)	104.239150 a 104.376488

Cuadro 4.4: Emisoras de FM

Además también se puede apreciar con bastante claridad la señal del sistema de alerta de sismos, que opera a 162.4 MHz.

4.6. Anchos de banda inferiores

En la literatura siempre se hace referencia a que nuestra ADC (ADC1x3000-8) tiene una frecuencia de operación mínima de 500 MHz; sin embargo, durante nuestras pruebas hemos establecido dicha frecuencia de operación por debajo de este mínimo, obteniendo buenos resultados. Hemos llegado a realizar prueabas operando a una frecuencia mínima de hasta 25 MHz, lo que suponde un incremento sustancial en nuestra resolución espectral, ya que pasaríamos de los 15,25879 kHz a 0,76294 kHz.

Capítulo 5

Conclusiones y trabajos futuros

Durante el desarrollo de este trabajo hemos conseguido implementar un espectrómetro portátil de alta resolución, fácilmente integrable en el backend de prácticamente cualquier radiotelescopio, y a un bajo costo económico. Las herramientas y modelos creados son fácilmente reproducibles, por lo que la implementación de nuevos espectrómetros siguiendo este trabajo se puede hacer de una forma rápida y sencilla.

Nos queda como tarea pendiente probar el rendimiento del espectrómetro en un entorno real de producción, por lo que en un futuro próximo instalaremos el espectrómetro en una antena de 5 metros, con el fin de analizar y caracterizar su comportamiento. También nos gustaría integrar el espectrómetro en otras antenas de mayor dimensiones e interferómetros, para lo cual estamos evaluando varias localizaciones.

Otra de las tareas pendientes es adaptar el diseño de nuestro instrumento para su funcionamiento como espectrómetro de ancho de banda variable, y probar de forma robusta el ADC1x3000-8 a tasas de muestreo por debajo de las indicadas por el fabricante. Tal y como indicamos en 4.6, ya hemos hecho algunas pruebas prometedoras, pero nos gustaría desarrollar un set de pruebas más exhaustivo.

También forma parte de nuestros planes seguir con el desarrollo de la pipeline, atendiendo para ello a las sugerencias que pueda hacer la comunidad; entendemos que esta es una labor importante, por lo que nos comprometemos a hacer una labor constante a lo largo del tiempo.

A más largo plazo, pretendemos seguir trabajando en varios proyectos de instrumentación, probando las nuevas tarjetas que Xilinx ha lanzado recientemente. De esta forma, pretendemos participar activamente en colaboraciones internacionales, participando en el desarrollo e implementación de distintos backends.

Como proyecto más ambicioso, pretendemos trabajar en el diseño y desarrollo de un nanosatélite o CubeSat que sirva como prototipo para una pequeña flota que sea capaz de funcionar como interferómetro espacial.

Apéndice A

Código fuente

A continuación presentamos el código fuente utilizado durante el desarrollo de esta tesis. Aparte de las librerías propias de la Pipeline, y que ya describimos en la sección 3.3, incluimos el código de dos scripts más que hemos utilizado en nuestras pruebas. Además el código puede ser descargado desde el repositorio de git **https://github.com/anyaterme/irya-spec.git**.

A.1. irya.py

```
#!/usr/bin/env python
'''
\nAuthor: Daniel Diaz, January 2018
'''

#TODO: add support for ADC histogram plotting.
#TODO: add support for determining ADC input level

import corr,time,numpy,struct,sys,logging,pylab,matplotlib, time
from irya_libs import IryaRoach
import numpy as np
import os

integration_time = 1

#bitstream = 'spec_2018_Sep_21_1213.bof'
#bitstream = 'spec_2018_Sep_26_1205.bof'
katcp_port=7147

def exit_fail():
    print 'FAILURE DETECTED. Log entries:\n',lh.printMessages()
    try:
```

```
        fpga.stop()
    except: pass
    raise
    exit()

def exit_clean():
    try:
        fpga.stop()
    except: pass
    exit()

def get_data(channels=2048):
    #get the data...
    acc_n = fpga.read_uint('acc_cnt')
    a_0=struct.unpack('>%dl' % int(channels/2),fpga.read('even',int(channels/2)*4,0))
    a_1=struct.unpack('>%dl' % int(channels/2),fpga.read('odd',int(channels/2)*4,0))

    interleave_a=[]

    for i in range(int(channels/2)):
        interleave_a.append(a_0[i])
        interleave_a.append(a_1[i])
    return acc_n, interleave_a

def save_to_file(data,timestamp,integration_time,numchannels, bandwidth):
    fname = "%s%06d.dat" %(time.strftime('%Y%m%d%H%M%S', time.localtime(timestamp)), int((timestamp*1e6)%1e6))
    path = os.path.join(os.path.dirname(os.path.realpath(__file__)), 'datas')
    path = os.path.join(path, fname)
    f = open(path, "wb")
    f.write(struct.pack(">dddi%dl" %numchannels, timestamp, integration_time, bandwidth, int(numchannels),*data))
    f.close()

def read_bw():
    global clk, bw
    oldbw = bw
    try:
        if not (clk.isOpen()):
            clk.open()
        bw = clk.get_freq()[1]/2.
        clk.close()
    except:
        bw = oldbw
    return(bw)

def plot_spectrum():
    global channel_max, last_acc, last_time,  bw, clk, integration_time, channels, specs, savefile
    acc_n, interleave_a = get_data(channels)
    interleave_a = np.asarray(interleave_a)
    interleave_a[0:5] = 0
    interleave_a[-5:] = 0
    if last_acc != acc_n:
```

```
        t = time.time()
        last_acc = acc_n


        matplotlib.pyplot.clf()
        matplotlib.pylab.plot(np.linspace(0,bw,channels), interleave_a)
        #interleave_a[np.where(interleave_a == 0)] = 1
        #matplotlib.pylab.semilogy(np.linspace(0,bw,channels), interleave_a)
        matplotlib.pylab.title('Integration number %i.'%acc_n)
        matplotlib.pylab.ylabel('Power (arbitrary units)')
        matplotlib.pylab.ylim(0)
        matplotlib.pylab.grid()
        #matplotlib.pylab.xlabel('Channel')
        #matplotlib.pylab.xlim(0,channels)
        matplotlib.pylab.xlabel('Freq (MHz)')
        fig.canvas.draw()
        if np.argmax(interleave_a) != channel_max:
            channel_max = np.argmax(interleave_a)
            #print (channel_max)
        if (acc_n > 0):
            #bw = read_bw()
            if (savefile):
                save_to_file(interleave_a, t, t-last_time, channels, bw)
            msg = "Accumulation time = %.4lf seconds." %(t - last_time)
            last_time = t
            msg = "%s Detection in channel %d.\r" %(msg,channel_max)
            print msg
    if (acc_n <= specs) or (specs == 0):
        fig.canvas.manager.window.after(100, plot_spectrum)
    else:
        print ("Adquisition finished...")
        exit_clean()


#START OF MAIN:

if __name__ == '__main__':
    from optparse import OptionParser


    p = OptionParser()
    p.set_usage('spectrometer.py <ROACH_HOSTNAME_or_IP> [options]')
    p.set_description(__doc__)
    p.add_option('-t', '--time', dest='time', type='float',default=2, help='Set the time to accumulations')
    p.add_option('-o', '--obstime', dest='obstime', type='int',default=300, help='Set the observation duration')
    p.add_option('-c', '--channels', dest='channels', type='int',default=2**14, help='Set the number channels.')
    p.add_option('-N', '--specs', dest='specs', type='int',default=0, help='Set the number of spectrums to read.')
    p.add_option('--bw', dest='bandwidth', type='float',default=250., help='Set the bandwidth [MHz] between 12.5 -
    p.add_option('-g', '--gain', dest='gain', type='int',default=0xffffffff, help='Set the digital gain (6bit qua
    p.add_option('-s', '--skip', dest='skip', action='store_true', help='Skip reprogramming the FPGA and configuri
    p.add_option('-b', '--bof', dest='boffile',type='str', default='spec16_2018_Oct_10_0905.bof', help='Specify th
```

```
    p.add_option('--no-save', dest='nosave', action='store_true', help='Not save data files.', default=False)
    opts, args = p.parse_args(sys.argv[1:])

    savefile = not (opts.nosave)
    if args==[]:
        print 'Please specify a ROACH board. Run with the -h flag to see all options.\nExiting.'
        exit()
    else:
        roach = args[0]
    if opts.bandwidth < 12.5 or opts.bandwidth > 700.:
        print "Please specify a bandwidth (MHz) between 12.5 y 700."
        exit()
    if opts.boffile != '':
        bitstream = opts.boffile

try:
    try:
        from valon5009 import Valon5009
        clk = Valon5009('/dev/ttyUSB2')
        if opts.bandwidth is not None:
            print ("Setting bandwidth at %.2lf MHz..." %opts.bandwidth)
            clk.set_freq(2,opts.bandwidth * 2.)
        bw = clk.get_freq()[1]/2.
        clk.close()
    except:
        bw = opts.bandwidth
    print ("Bandwidth set at %.2lf MHz..." %bw)

    loggers = []
    lh=corr.log_handlers.DebugLogHandler()
    logger = logging.getLogger(roach)
    logger.addHandler(lh)
    logger.setLevel(10)
    channel_max = 0
    channels = opts.channels

    print('Connecting to server %s on port %i ... ' %(roach,katcp_port)),
    #fpga = corr.katcp_wrapper.FpgaClient(roach, katcp_port, timeout=10,logger=logger)
    fpga = IryaRoach(roach, katcp_port, timeout=10,logger=logger)

    time.sleep(1)

    if fpga.is_connected():
        print 'ok\n'
    else:
        print 'ERROR connecting to server %s on port %i.\n'%(roach,katcp_port)
        exit_fail()

    print '------------------------'
    print 'Programming FPGA with %s ... ' %bitstream,
    if not opts.skip:
```

```
        fpga.progdev(bitstream)
        print 'done'
    else:
        print 'Skipped.'

    integration_time = opts.time
    specs = opts.specs
    #acc_len = int (opts.time * ((2**(np.log2(bw*1e6))) / channels))
    #acc_len = int(opts.time * 2**(int(np.ceil(np.log2(bw*1e6)))) / (channels * 2.))
    acc_len = int(opts.time * bw * 1e6 / (channels))
    print "Setting accumulation time at %.2lf sec ... " % opts.time
    print 'Configuring accumulation period to %d ...' % acc_len,
    fpga.write_int('acc_len',acc_len)
    print 'done'

    print 'Resetting counters...',
    fpga.write_int('cnt_rst',1)
    fpga.write_int('cnt_rst',0)
    print 'done'

    print 'Setting digital gain of all channels to %i ...'%opts.gain,
    if not opts.skip:
        fpga.write_int('gain',opts.gain) #write the same gain for all inputs, all channels
        print 'done'
    else:
        print 'Skipped.'

    if not savefile:
        print "Not save file"

    last_time = time.time()
    last_acc = None
    #set up the figure with a subplot to be plotted
    fig = matplotlib.pyplot.figure()
    ax = fig.add_subplot(1,1,1)

    # start the process
    print 'Plot started.'
    fig.canvas.manager.window.after(100, plot_spectrum)
    matplotlib.pyplot.show()
    print '\nBye'

except KeyboardInterrupt:
    exit_clean()
except Exception as e:
    print (e)
    exit_fail()

exit_clean()
```

A.2. irya_libs.py

```
import matplotlib.pyplot as plt
import astropy.units as u
import numpy as np
import struct
from colour import Color
import datetime

def find_nearest(array, value):
    idx = (np.abs(array - value)).argmin()
    return array[idx]

def find_nearest_index(array, value):
    idx = (np.abs(array - value)).argmin()
    return idx

try:
    import casperfpga
    import corr,time,numpy,struct,sys,logging,pylab
    class IryaRoach(casperfpga.CasperFpga):
        def __init__(self, *args, **kwargs):
            print ("You are using casperfpga.CasperFpga")
            super(IryaRoach, self).__init__(*args, **kwargs)
            reply, _ = self.transport.katcprequest( name='fpgastatus', request_timeout=self.transport._timeout, require_ok=
            if (reply.arguments[0] == 'invalid'):
                self.roach_version=1
            else:
                self.roach_version=2

        def listbof(self):
            return self.transport.listbof()

        def is_running(self):
            if self.roach_version == 2:
                reply, _ = self.transport.katcprequest( name='fpgastatus', request_timeout=self.transport._timeout, require
            else:
                reply, _ = self.transport.katcprequest( name='status', request_timeout=self.transport._timeout, require_ok=
            return reply.arguments[0] == 'ok'

        def progdev(self, filename):
            if (".fpg" in filename):
                self.upload_to_ram_and_program(filename)
                return True
            if (".bof" in filename):
                if (filename not in self.listbof()):
                    self.transport.upload_to_flash(filename)
                self.program(filename.split('/')[-1])
                return True
```

```
        return False

    def program(self, filename=None):
        """
        Program the FPGA with the specified binary file.
        :param filename: name of file to program, can vary depending on
            the formats supported by the device. e.g. fpg, bof, bin
        :return:
        """
        # raise DeprecationWarning('This does not seem to be used anymore.'
        #                          'Use upload_to_ram_and_program')
        # TODO - The logic here is for broken TCPBORPHSERVER, needs fixing
        if 'program_filename' in self.transport.system_info.keys():
            if filename is None:
                filename = self.transport.system_info['program_filename']
            elif filename != self.transport.system_info['program_filename']:
                print('%s: programming filename %s, configured ' 'programming_filename %s' %(self.transport.h
                # This doesn't seem as though it should really be an error...
        if filename is None:
            raise RuntimeError('Cannot program with no filename given. ' 'Exiting.')
        unhandled_informs = []
        # set the unhandled informs callback
        self.transport.unhandled_inform_handler = \
            lambda msg: unhandled_informs.append(msg)
        reply, _ = self.transport.katcprequest(name='progdev', request_timeout=10, request_args=(filename, ))
        self.transport.unhandled_inform_handler = None
        if reply.arguments[0] == 'ok':
            complete_okay = False
            for inf in unhandled_informs:
                if (inf.name == 'fpga') and (inf.arguments[0] == 'ready'):
                    complete_okay = True
            if not complete_okay: # Modify to do an extra check
                if self.is_running():
                    complete_okay = True
                else:
                    raise RuntimeError('%s: programming %s failed.' %(self.transport.host, filename))
            self.transport.system_info['last_programmed'] = filename
        else:
            raise RuntimeError('%s: progdev request %s failed.' %(self.transport.host, filename))
        if filename[-3:] == 'fpg':
            self.transport.get_system_information()
        else:
            print('%s: %s is not an fpg file, could not parse ' 'system information.' %(self.transport.host,
        print('%s: programmed %s okay.' %(self.transport.host, filename))


    def write_int(self, device_name, data, offset=0, blindwrite=False):
        return(super(IryaRoach, self).write_int(device_name, data, word_offset=offset, blindwrite=blindwrite))

    def read_int(self, device_name, offset=0):
        return(super(IryaRoach, self).read_int(device_name, word_offset=offset))
```

```
        def est_brd_clk(self):
            return(self.estimate_fpga_clock())

        def read_ram(self,device, n_words=1024, data_format='B', offset=0):
            size_cat = {'b':1., 'B':1., 'l':4., 'L':4., 'f':4., 'd':8., 'Q':8., 'q':8.}
            if device in self.listdev():
                snapshot = self.read(device,n_words * size_cat[data_format],offset=offset)
                string_data = struct.unpack('>%d%s' %(n_words, data_format), snapshot)
                array_data = np.array(string_data, dtype='float')
                return array_data
            else:
                print "ERROR. %s is not present in bof file." %device
                return np.zeros(n_words)

        def wait_connected(self, timeout=5):
            init_time = time.time()
            while init_time + timeout > time.time() and not self.is_connected():
                pass
            return self.is_connected()

        def read_full_stream(self, device_prefix, n_words=1024, data_format='B', offset=0):
            bram_names = []
            list_dev = self.listdev()
            for dev in list_dev:
                if device_prefix in dev:
                    bram_names.append(dev)
            bram_names.sort()
            num_brams = len (bram_names)
            print "Reading %d devices: %s" %(num_brams, bram_names)
            y = np.zeros(n_words)
            for k in range(num_brams):
                data = self.read_ram(bram_names[k], int(n_words/num_brams), data_format, offset)
                y[k::num_brams] = data
            return y

        def stop(self):
            pass
except Exception as e:
    print ("WARNING!!!!!", e)
    pass


class Spectrum():
    def __init__ (self, path=None, bw=1., channels=1, integ=1.):
        if path is not None:
            f = open (path,"rb")
            data_raw = f.read()
            f.close()
            try:
                (self.timestamp, self.integration, self.bw, self.channels) = struct.unpack(">dddi", data_raw[:28])
```

```
                self.data = np.asarray(struct.unpack(">%dl" % self.channels, data_raw[28:]))
            except:
                (self.timestamp, self.integration, self.bw, self.channels) = struct.unpack(">fffi", data_raw[:28])
                self.data = np.asarray(struct.unpack(">%dl" % self.channels, data_raw[16:]))
        else:
            self.bw = bw
            self.integration = integ
            self.channels=channels
            self.data = np.random.random(channels)

        self.data_db = self.data

        self.data_db[np.where(self.data_db == 0)] = 1
        self.data_db = 10 * np.log10(self.data_db)

        self.bandwidth = np.linspace(0,self.bw,self.channels)

    def show(self, zoom=None, units=u.MHz):

        if zoom is None:
            spec_x = np.linspace(0, (self.bw*u.MHz).to(units), self.channels)
            spec_y = self.data
        else:
            if hasattr(zoom,'unit'):
                spec_x = np.linspace(0, (self.bw*u.MHz).to(units), self.channels)
                idx0 = find_nearest_index(spec_x, zoom[0])
                idx1 = find_nearest_index(spec_x, zoom[1])
                spec_x = spec_x[idx0:idx1]
                spec_y = self.data[idx0:idx1]
            else:
                spec_x = np.linspace(0, (self.bw*u.MHz).to(units), self.channels)
                spec_x = spec_x[zoom[0]:zoom[1]]
                spec_y = self.data[zoom[0]:zoom[1]]
        plt.xlabel('Freq (%s)' % spec_x.unit)
        plt.ylabel('Arbitrary units')
        plt.plot(spec_x, spec_y)
        #plt.show()

    def detections(self, sigma=3):
        channels = np.where(self.data > np.mean(self.data) + sigma*np.std(self.data))
        return channels, self.bandwidth[channels],self.data[channels]

    def show_detections(self, sigma=3, zoom=None, show_labels=True, db = False):
        if db :
            data = self.data_db
        else:
            data = self.data
        if zoom is None:
            spec_x = np.linspace(0, (self.bw*u.MHz), self.channels)
            spec_y = data
        else:
```

```
            if hasattr(zoom,'unit'):
                spec_x = np.linspace(0, (self.bw*u.MHz).to(zoom.unit), self.channels)
                idx0 = find_nearest_index(spec_x, zoom[0])
                idx1 = find_nearest_index(spec_x, zoom[1])
                spec_x = spec_x[idx0:idx1]
                spec_y = data[idx0:idx1]
            else:
                spec_x = np.linspace(0, (self.bw*u.MHz).to(units), self.channels)
                spec_x = spec_x[zoom[0]:zoom[1]]
                spec_y = data[zoom[0]:zoom[1]]




        plt.plot(spec_x, spec_y)
        channels = np.where(spec_y > np.mean(data) + sigma*np.std(data))
        freq = spec_x[channels]
        data = spec_y[channels]
        colors = list(Color('red').range_to(Color("Blue"),len(channels[0])))
        labels=('o','v','^','<','>','8','s','p','*','h','H','D','d','P','X')
        plots = []
        legend_labels = []
        for i in range(len(channels[0])):
            plots.append(plt.scatter(freq[i],data[i],color=colors[i].get_rgb(),marker=labels[i %len(labels)]))
            legend_labels.append("%d - %s" %(channels[0][i], "{0:0.03f}".format(freq[i])))
        if show_labels:
            plt.legend(list(plots), list(legend_labels))

        if db:
            plt.ylabel('Poser dB (arbitrary units)')
        else:
            plt.ylabel('Power (arbitrary units)')
        plt.xlabel('Freq (MHz)')
        #plt.title(str(datetime.datetime.fromtimestamp(self.timestamp)))
        #plt.show()
        return channels, freq, data

    def get_channel(self, freq):
        if not hasattr(freq,'unit'):
            freq = freq * u.MHz

        return find_nearest_index(self.bandwidth * u.MHz, freq)

    def clean_noise(self, sigma=3):
        nonoise = np.copy(self.data)
        nonoise = nonoise - (np.mean(self.data) + sigma * np.std(self.data))
        nonoise[np.where(nonoise<0)]=0
        return nonoise

    def get_snr(self):
        detections = self.detections()
```

```
        if(len (detections[0]) > 0):
            return max(detections[2] * 1. / self.get_noise())
        else:
            return 0.

    def get_harmonic(self, freq):
        if not hasattr(freq, 'unit'):
            freq = freq * u.MHz
        bw = (self.bw*u.MHz).to(freq.unit)
        fact =  ((-1)**int(int(freq.value/bw.value)%2) * (freq.value%bw.value))
        armonic = (bw.value + ((-1)**int(int(freq.value/bw.value)%2) * (freq.value%bw.value))) %(bw.value)
        armonic = armonic * freq.unit
        idx = find_nearest_index(self.bandwidth*u.MHz, armonic)
        return idx,armonic,self.data[idx]

    def get_noise(self):
        return np.mean(self.data)

class Spec():
    def __init__(self, roach_address, katcp_port=7147, bitstream=None, channels=2**13, acc_time=1, gain=0xffffffff
        self.bw = bw
        loggers = []
        lh=corr.log_handlers.DebugLogHandler()
        logger = logging.getLogger(roach_address)
        logger.addHandler(lh)
        logger.setLevel(10)
        channel_max = 0
        self.channels = channels
        print('Connecting to server %s on port %i... '%(roach_address,katcp_port)),
        self.fpga = IryaRoach(roach_address, katcp_port, timeout=10,logger=logger)
        fpga = self.fpga
        time.sleep(1)
        if self.fpga.is_connected():
            print 'ok\n'
        else:
            print 'ERROR connecting to server %s on port %i.\n'%(roach_address,katcp_port)
            exit()

        if bitstream is not None:
            print '------------------------'
            print 'Programming FPGA with %s...' %bitstream,
            fpga.progdev(bitstream)
            print 'done'
            print 'Setting digital gain of all channels to %i...'%opts.gain,
            fpga.write_int('gain',opts.gain) #write the same gain for all inputs, all channels
            print 'done'
        else:
            print 'Using current program in FPGA.'

        acc_len = int (acc_time * (2**(np.log2(bw*1e6))) / channels)
        print 'Configuring accumulation period to %d...' % acc_len,
```

```
        fpga.write_int('acc_len',acc_len)
        print 'done'

        print 'Resetting counters...',
        fpga.write_int('cnt_rst',1)
        fpga.write_int('cnt_rst',0)
        print 'done'
```

A.3. valon5009.py

```
import sys
import serial
import logging
import numpy as np

class Valon5009(serial.Serial):
    SRC_1 = 1
    SRC_2 = 2
    def __init__(self, port):
        super(Valon5009, self).__init__(port=port, baudrate=9600, timeout=1)
        try:
            if (not self.isOpen()):
                self.open()
        except serial.SerialException as e:
            logging.error("Could not open serial port {}: {}".format(self.name, e))
            sys.exit(1)

    def send_command(self, cmd):
        if self.isOpen():
            self.write(cmd.encode('ascii')+'\r\n')
        else:
            print "You have to open the port"


    def set_freq(self, source, freq):
        if source not in [1,2]:
            print("You must specify the source [1,2].")
            return False
        if freq < 25:
            print("Minimum frequency is 25 [MHz]")
            return False
        if freq > 6000:
            print("Maximum frequency is 6000 [MHz]")
            return False
        cmd = "s%d; f%df" %(source, freq)
        self.send_command(cmd)
        self.readlines()
        return True
```

```
def get_freq(self, source=None, verbose=False):
    if source not in [1,2]:
        cmd = "s1; f"
        self.send_command(cmd)
        out1 = self.readlines()
        self.reset_output_buffer()
        cmd = "s2; f"
        self.send_command(cmd)
        out2 = self.readlines()
        self.reset_output_buffer()
        if (verbose):
            print "Source 1: ", out1[1]
            print "Source 2: ", out2[1]
        return float((out1[1].split(';')[0].split(' ')[1])), float((out2[1].split(';')[0].split(' ')[1]))
    cmd = "s%d; f" %(source)
    self.send_command(cmd)
    out = self.readlines()
    self.reset_output_buffer()
    if (verbose):
        print "Source %d: " %source, out[1]
    return float((out[1].split(';')[0].split(' ')[1]))

def set_pow(self, source=1, level=0):
    if source not in [1,2]:
        print("You must specify the source [1,2].")
        return False
    if level not in [0,1,2,3]:
        print("You must specify a valid power level [0,1,2,3].")
        return False
    cmd = "s%d; plev %d" %(source, level)
    self.send_command(cmd)
    out = self.readlines()
    self.reset_output_buffer()
    print out[1]
    return float((out[1].split(';')[0].split(' ')[1]))

def get_pow(self, source=None):
    if source is None:
        return (self.get_pow(1), self.get_pow(2))
    else:
        if source not in [1,2]:
            print("You must specify the source [1,2].")
            return False
        cmd = "s%d; plev?" %(source)
        self.send_command(cmd)
        out = self.readlines()
        self.reset_output_buffer()
        return float((out[1].split(';')[0].split(' ')[1]))

def set_att(self, source=1, att=0):
    if source not in [1,2]:
```

```
            print("You must specify the source [1,2].")
            return False

        valid_range = np.arange(0, 32, 0.5)
        if att not in valid_range:
            att = valid_range[np.argmin(abs(att-valid_range))]
            print "Your attenuation is not valid. Set attenuation to %.1lf" % att
        cmd = "s%d;att %d" %(source, att)
        self.send_command(cmd)
        out = self.readlines()
        self.reset_output_buffer()
        print out[1]
        return float((out[1].split(';')[0].split(' ')[1]))

    def get_att(self, source=None):
        if source is None:
            return(self.get_att(1), self.get_att(2))
        else:
            if source not in [1,2]:
                print("You must specify the source [1,2].")
                return False
            cmd = "s%d;att?" %(source)
            self.send_command(cmd)
            out = self.readlines()
            self.reset_output_buffer()
            return float((out[1].split(';')[0].split(' ')[1]))

    def save(self):
        cmd = "sav"
        self.send_command(cmd)
        out = self.readlines()
        self.reset_output_buffer()
        print out[1]




if __name__ == '__main__':
    import argparse

    parser = argparse.ArgumentParser( description="Valon Synthetizer 5009 ctrl", epilog="""\ """)
    parser.add_argument('SERIALPORT')
    parser.add_argument( '-v', '--verbose', dest='verbosity', action='count', help='print more diagnostic messages (option
    args = parser.parse_args()


    if args.verbosity > 3:
        args.verbosity = 3
    level = (logging.WARNING, logging.INFO, logging.DEBUG, logging.NOTSET)[args.verbosity]
    logging.basicConfig(level=logging.INFO)
    #logging.getLogger('root').setLevel(logging.INFO)
```

```
    logging.getLogger('valon5009').setLevel(level)
    valon = Valon5009(args.SERIALPORT)
    valon.get_freq(1)
    valon.get_freq(2)
    valon.write('s1;_f')
```

A.4. allan.py

```
import allantools # https://github.com/aewallin/allantools/
import glob
from irya_libs import *
import matplotlib.pyplot as plt

import argparse
import os

parser = argparse.ArgumentParser(description='Join_spectrums')
parser.add_argument('path', metavar='<path>', type=str, help='path', default=".")
parser.add_argument("--norm", help="Normalize_spectrum", action="store_true")
parser.add_argument("--num", type=int, default=-1, help="Number_of_espectrum")
args = parser.parse_args()


files=glob.glob(os.path.join(args.path,'*.dat'))
files.sort()
files = files[1:args.num]
spec_total=Spectrum(files[0])
spec_total.data *= 0
taus_spec = []
power = []
timestamps = []
chann_range = np.arange(8585, 8785)
#chann_range = 8585
print "Reading_%d_spectrums..." %len(files)
for i in files:
    spec = Spectrum(i)
    spec_total.data += spec.data
    taus_spec.append(spec.integration)
    timestamps.append(spec.timestamp)
    power.append(spec.data[chann_range])



power = np.asarray(power)

#Normalize power (at mean)
if args.norm:
    max_by_chan=power.max(axis=0)
    mean = power.mean()
```

```
    power = power / mean
    #power = power * mean


rate = 1./np.mean(taus_spec)
taus = np.arange(0,int(rate*len(files)/2),rate)
print taus[-1]
# fractional frequency data
print "Calculating allan variance from %d spectrums, with rate %.4lf Hz and %.1lf sec of duration..." %(len(files), rate,
adev_avg = []
errors_avg =[]
for i in range(power.shape[1]):
    (taus_used, adev, adeverror, adev_n) = allantools.adev(power[:,i], data_type='freq', rate=rate, taus=taus)
    adev_avg.append(adev)
    errors_avg.append(adeverror)

adev_avg = np.asarray(adev_avg)
errors_avg = np.asarray(errors_avg)
plt.rcParams.update({'font.size': 20})
plt.loglog(taus_used, adev_avg.mean(axis=0), label = r'%d spectrums, rate %.4lf Hz, %.1lf sec' %(len(files), rate, timestan
#plt.errorbar(taus_used,adev_avg.mean(axis=0), yerr=errors_avg.mean(axis=0))


ideal_dev = np.sqrt(2. / (1.*250e6 / 2**14 * taus_used))
plt.loglog(taus_used, ideal_dev, label='Radiometer')

plt.ylabel('Allan Desv')
plt.xlabel(r'T(s)')
if not type(chann_range) is int:
    plt.title('IRyA Spectrometer, channel [%d - %d]' %(chann_range[0], chann_range[-1]))
else:
    plt.title('IRyA Spectrometer, channel %d' %(chann_range))

plt.legend()
plt.show()


#aux = np.ediff1d(adev)
#aux = aux / np.abs(aux)
#idx = np.where(aux > 0)
#if (len(idx[0]) > 0):
#    print ("The optimum value is %lf sec." %(taus_used[idx[0][0]]))
#else:
#    print ("The optimum value is %lf sec." %(taus_used[-1]))
```

A.5. shift_freq.py

```
from valon5009 import Valon5009
import time
```

```
if __name__ == '__main__':
    import argparse

    parser = argparse.ArgumentParser(description='Process some integers.')
    parser.add_argument('f0', metavar='freq_ini', type=int,
                        help='initial frequency')
    parser.add_argument('total_time', type=int,
                        help='total time')
    parser.add_argument('delta_time', type=int,
                        help='time for each step in frequency')
    parser.add_argument('--inc', dest='inc_delta', type=float, default=1.0,
                        help='factor to increment delta time after each step')
    parser.add_argument('--ch', dest='n_chan', type=int, default=2**14,
                        help='number of channels')
    parser.add_argument('--bw', dest='bw', type=float, default=250.0,
                        help='bandwidth')
    parser.add_argument('--dev', dest='dev', type=str, default='/dev/ttyUSB1',
                        help='dev for clock')

    args = parser.parse_args()

    clk = Valon5009(args.dev)
    f0 = args.f0
    bw = args.bw
    n_chan = args.n_chan
    step = bw / n_chan
    total_time = args.total_time
    delta_time = args.delta_time
    inc_delta = args.inc_delta
    t0 = time.time()
    while (time.time() - t0 < total_time):
        print ("[%lf] Setting freq %.6lf" %(time.time(), f0))
        clk.set_freq(1, f0)
        t1 = time.time()
        while (time.time() - t1 < delta_time):
            pass
        delta_time *= inc_delta
        f0 += step
```

Bibliografía

Batrla, W., Matthews, H., Menten, K., & Walmsley, C. 1987, Nature, 326, 49

Berkeley University of California. 2006, Collaboration for Astronomy Signal Processing and Electronics Research (CASPER), https://casper.berkeley.edu

Bianchi, G., Perini, F., Bortolotti, C., et al. 2008, 13th Workshop on ADC Modeling and Testing

Chen, H., McMahon, P., & Parsons, A. 2008, Sync Pulse Usage in CASPER DSP Blocks, https://casper.berkeley.edu/memos/sync_memo_v1.pdf

Chen, X., Ellingsen, S. P., Shen, Z.-Q., Titmarsh, A., & Gan, C.-G. 2011, APJS, 196, 9

Collaboration for Astronomy Signal Processing and Electronics Research (CASPER). 2016, The Polyphase Filter Bank Technique, https://casper.berkeley.edu/wiki/The_Polyphase_Filter_Bank_Technique

Collaboration for Astronomy Signal Processing and Electronics Research (CASPER). 2018, CASPER Processing Platform Comparison Matrix, https://casper.berkeley.edu/wiki/Hardware

Estalella, R. & Anglada, G. 1997, Introducción a la Física del Medio Interestelar (Edicions de la Universitat de Barcelona, Spain)

Genzel, R., Downes, D., Schneps, M., et al. 1981, APJ, 247, 1039

Group, T. M. I.-S. E. 2006, Analog-Digital Converter Bit Number and Input Power Evaluation, Tech. Rep. 390, IRA-INAF

Harris, C. & Haines, K. 2011, Publications of the Astronomical Society of Australia, 28, 317

Herrera-Martínez, G., Luna, A., Carrasco, L., et al. 2009, Revista Mexicana de Astronomía y Astrofísica, Vol. 37

Keto, E., Zhang, Q., & Kurtz, S. 2008, The Astrophysical Journal, 672, 423

Kurtz, S., Hofner, P., & Álvarez, C. V. 2004, APJS, 155, 149

Land, D., Levick, A., & Hand, J. 2007, Measurement Science and Technology, 18, 1917

Lo, K., Walker, R., Burke, B., et al. 1975, ApJ202

Lumen Learning. 2016, Radiation and Spectra. The Electromagnetic Spectrum., https://courses.lumenlearning.com/astronomy/chapter/the-electromagnetic-spectrum/

Masson, J., Chabrier, G., Hennebelle, P., Vaytet, N., & Commerçon, B. 2016, APJ, 587, A32

Menten, K. M. 1991, The Astrophysical Journal, 380, L75

Momjian, E. & Sarma, A. P. 2017, APJ, 834, 168

National Science Foundations. 2018, Green Bank Observatory, https://greenbankobservatory.org/

Neufeld, D. & Melnick, G. 1990, ApJ352

Nixon, C. J. & Pringle, J. E. 2019, New Astronomy, 67, 89

Patoka, O. M., Shulga, V. M., Antyufeyev, O. V., et al. 2018, Kinematics and Physics of Celestial Bodies, 34, 217

Reid, M. J., Menten, K. M., Zheng, X. W., et al. 2009, APJ, 700, 137

Rodriguez-Andina, J. J., Moure, M. J., & Valdes, M. D. 2007, IEEE Transactions on Industrial Electronics, 54, 1810

Schieder, R. & Kramer, C. 2001, Astronomy & Astrophysics

Shannon, C. E. 1949, Proceedings of the Insitute of Radio Engineer, Vol. 37

Siva, T., Latha, S., & Supraja, Y. N. 2012, International Journal of Emerging Technology and Advanced Engineering, 2

SKA Africa. 2014-2018, Software control for CASPER FPGAs, https://github.com/ska-sa/casperfpga/blob/master/README.md

Slysh, V. I., Kalenskii, S. V., Val'tts, I. E., & Golubev, V. V. 1997, , 478, L37

Sun, Y., Xu, Y., Chen, X., et al. 2018, APJ, 869, 148

Surcis, G., Vlemmings, W., Dodson, R., & Van Langevelde, H. 2009, Astronomy & Astrophysics, 506, 757

Swinburne University. 2018, Emission Line - COSMOS, http://astronomy.swin.edu.au/cosmos/E/Emission+Line

Texas Instrument. 2009, ADC083000 8-Bit, 3 GSPS, High Performance, Low Power A/D Converter, http://www.ti.com/lit/ds/symlink/adc083000.pdf

The MathWorks, Inc. 1994-2018, Matlab Simulink

Valon Technology. 2017

Walden, R. H. 1999, IEEE Journal on selected areas in communications, 17, 539

Wikipedia contributors. 2018, Leck-Effekt

Wikipedia.org. 2018a, Electromagnetic spectrum, https://en.wikipedia.org/wiki/Electromagnetic_spectrum

Wikipedia.org. 2018b, Receptor Superheterodino, https://es.wikipedia.org/wiki/Receptor_superheterodino

Xilinx. 2015, Virtex-5 Family Overview, Xilinx, rev. 5.1

Yang, H., Zhang, J., Sun, J., & Yu, L. 2014, Journal of Electronics (China), 31, 371