UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

DISEÑO DE UN NODO ACÚSTICO HARDWARE

Alberto Monleón Rojas

Tutores: María de Diego Antón, Miguel Ferrer Contreras.

Cotutor: Germán Ramos Peinado.

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2017-18

Resumen

El estudio a realizar se trata del diseño de un nodo acústico Hardware, en colaboración con el iTeam de la Universidad Politécnica de Valencia.

La propuesta se realiza con la finalidad diseñar un nodo acústico Hardware que sea capaz de realizar una cancelación activa de ruido, para ello se me ha proporcionado una tarjeta DSP de la familia 5000 de Texas Instruments. En concreto la USBSTK5505, contiene el DSP TMS320C5505 el cual trabaja en punto fijo. Ya que se han realizado unos estudios previos a la realización de este trabajo en el cual se ha determinado que esta tarjeta tiene la suficiente capacidad computacional para el desarrollo del trabajo.

El iTeam me ha proporcionado todo el material necesario para el desarrollo del mismo, así como cables RCA-miniJack, adaptadores, tarjeta de sonido…

Los primeros pasos realizados se basaban en el estudio de la tarjeta proporcionada. En la página web de Spectrum Digital podemos encontrar una serie de Demos con las que podemos trabajar. Entre ellas se probó una que generaba dos tonos diferentes, un bucle stereo y el encendido del Led que hay en la tarjeta.

A continuación se probó un filtro FIR sencillo que trabaja muestra a muestra y analizando su respuesta, así comprobaríamos su latencia y capacidad. En la carrera he dado la asignatura DSP impartida por el profesor Germán Ramos Peinado en la que realizamos proyectos similares. Gracias a ello pude implementar con facilidad ese filtro FIR, su respuesta fue positiva ya que se observó que no se producía prácticamente latencia. Se trataba de un filtro FIR paso-banda de 10 coeficientes. Primero se genera en Matlab utilizando la herramienta *fdatool,* en esta herramienta puedes elegir el tipo de filtro que quieres implementar así como el orden del filtro, la frecuencia de corte, la frecuencia de muestreo… La herramienta te muestra una gráfica de la respuesta en frecuencia del filtro que es la que utilizamos para comparar con la simulación de nuestro filtrado que implementamos en la tarjeta y analizamos en ARTA.

Una vez se llegó a este punto vimos que la tarjeta USBSTK5505 nos proporcionaba las condiciones necesarias para el desarrollo de este proyecto. Las estructuras a seguir me fueron proporcionadas por el equipo técnico del iTeam, en la que se me explicó que tendría que desarrollar filtros adaptativos y un filtrado-x basado en el algoritmo LMS para conseguir esa cancelación activa de ruido.

Como ante cualquier trabajo, hay que realizar un estudio teórico y diferentes pruebas para llegar a nuestro objetivo. Estas pruebas serán: la programación de un bypass, un filtrado FIR, filtrado adaptativo y un filtrado-x finalmente.

Tras las pruebas realizadas se puede determinar que diseño del nodo acústico hardware ha sido positivo y la tarjeta proporcionada da un buen rendimiento. Por lo que puede ser factible realizar la reducción de ruido que buscamos con la tarjeta.

Abstract

The study to be carried out is about the design of a hardware acoustic node, in collaboration with the iTeam of the Polytechnic University of Valencia.

The proposal is made with the purpose of designing a hardware acoustic node that is capable of performing an active noise cancellation, for this I have been provided with a DSP card of the family 5000 of Texas Instruments. In particular the USBSTK5505, contains the DSP TMS320C5505 which works in fixed point. Since there have been some studies prior to the completion of this work in which it has been determined that this card has sufficient computational capacity for the development of work.

The iTeam has provided me with all the necessary material for the development of it, as well as RCA-miniJack cables, adapters, sound card ...

The first steps were based on the study of the card provided. On the Spectrum Digital website we can find a series of Demos with which we can work. Among them, one was tested that generated two different tones, a stereo loop and the lighting of the LED on the card.

Then we tried a simple FIR filter that works sample to sample and analyzing its response, so we would check its latency and capacity. In the race I have given the DSP course taught by Professor Germán Ramos Peinado in which we carry out similar projects. Thanks to this I was able to easily implement this FIR filter, its response was positive since it was observed that there was practically no latency. It was a FIR pass-band filter with 10 coefficients. First generated in Matlab using the fdatool tool, in this tool you can choose the type of filter you want to implement, as well as the order of the filter, the cutoff frequency, the sampling frequency ... The tool shows you a graph of the frequency response of the filter that is what we use to compare with the simulation of our filtering that we implemented on the card and analyzed in ARTA.

Once this point was reached, we saw that the USBSTK5505 card provided the necessary conditions for the development of this project. The structures to follow were provided to me by the iTeam technical team, in which it was explained to me that I would have to develop adaptive filters and an x-filtering based on the LMS algorithm to achieve that active noise cancellation.

As with any job, you have to perform a theoretical study and different tests to reach our goal. These tests will be: the programming of a bypass, a FIR filtering, adaptive filtering and an x-filtering finally.

After the tests carried out, it can be determined which design of the acoustic hardware node has been positive and the card provided gives a good performance. So it may be feasible to make the noise reduction we are looking for with the card.

ÍNDICE

Capítulo 1. Estudio teórico

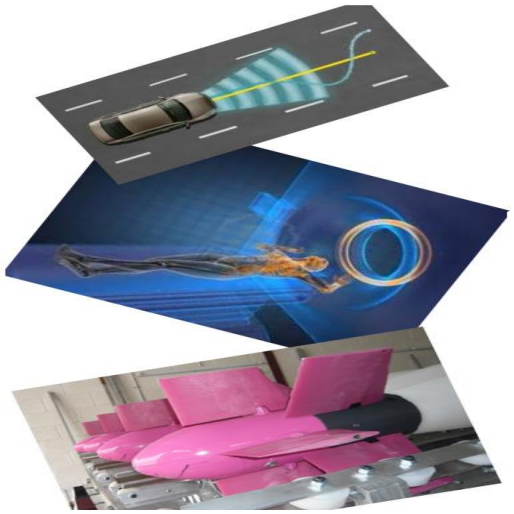
1.1 DSP

Un DSP no es más que un microprocesador muy rápido con una arquitectura diseñada especialmente para ejecutar algoritmos de procesado de señal con un elevado contenido aritmético a gran velocidad.

El tratamiento digital de una señal nos proporciona una serie de ventajas:

- Desaparece la tolerancia de los componentes.
- Aparece precisión numérica lo que lo hace todo más predecible y controlable.
- Estabilidad.
- Causalidad: ante una misma entrada tenemos una misma salida.
- Re-programabilidad. Son actualizables.
- Mayor inmunidad ante el ruido, la corrupción…
- Funciones no disponibles si trabajamos en analógico: filtro lineales, buffers, procesado adaptativo, sistemas expertos, procesado estático, compresión/descompresión…
- Menor tiempo de diseño, consumo y tamaño final.

El uso de los DSP nos proporcionan una serie de aplicaciones ya que proporciona: una gran potencia de cálculo, gran manejo de datos, arquitecturas internas paralelas, interacción con el mundo externo y periféricos, diseño para trabajar en tiempo real… Por ello se utilizan en un amplio número de trabajos: audio (ecualizadores, compresores, reductores de ruido…), telecomunicación (módems, codificador de tonos, encriptado, conmutación digital…), automoción, medicina, militar…



El DSP con el que vamos a trabajar funciona en punto fijo, ha sido elegido entre otras cosas porque nos proporciona arquitecturas muy rápidas, un bajo consumo y un menor coste. Aunque la programación trabajando en punto fijo es mucho más compleja que si lo hiciéramos en punto flotante.

1.2 Filtro FIR

Finite Impulse Response, es un sistema discreto, lineal e invariante en el tiempo (*LTI*), cuya respuesta al impulso es de longitud finita. Un filtrado *FIR* no es más que una convolución en el dominio temporal entre dos señales, la primera la señal de entrada al filtro y la segunda la respuesta al impulso del filtro.

En frecuencia, el efecto es que el contenido frecuencial de la señal de entrada se multiplica por la respuesta en frecuencia del filtro. La ecuación que implementa el filtro *FIR* es por tanto la convolución de las señales en el dominio temporal, siguiendo la siguiente ecuación:

$$y[n] = x[n] * h[n]$$

- ***x[n]* = señal de entrada**
- ***y[n]* = señal de salida**
- ***h[n]* = respuesta del impulso**

Esta ecuación también se puede expresar como:

$$y[n] = \sum_{k=0}^{N-1} h[k]\, x[n-k]$$

- **N-1 es el orden del filtro**

La salida del filtro solo depende de la señal de entrada y la respuesta del filtro, en el caso de los filtros IIR la salida dependerá de muestras de salidas en instantes anteriores. Los filtros *FIR* tienen la particularidad de ser no recursivos, es por ello que la estabilidad del filtro se garantiza siempre que haya un numero de coeficientes *h[n]* finitos. Se define como un sistema estable aquel que para una señal de entrada finita la señal de salida también lo es.

Para que un filtro *FIR* pueda trabajar necesita conocer la muestra de la señal de entrada en ese instante y las *N-1* muestras anteriores a ella. Como la memoria de un filtro *FIR* no depende de las muestras de entrada posteriores se dice que es un filtro de tipo causal. La salida del filtro solo podrá estar retrasada con respecto a la entrada.

Este tipo de filtros permite diseñar filtros de respuesta en fase lineal si la respuesta al impulso es simétrica.

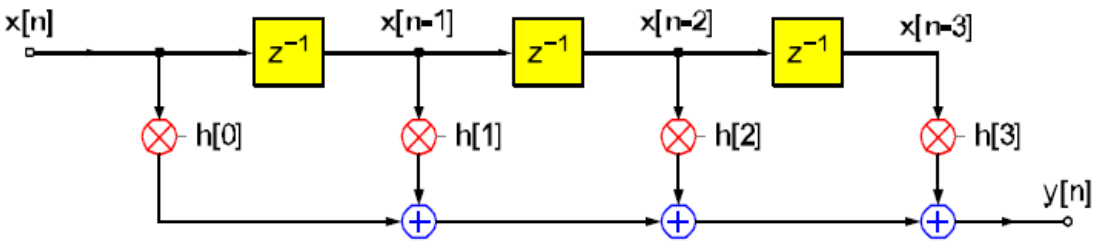


Fig 1: Estructura transversal utilizada para el filtro FIR.

Se puede diseñar un filtro *FIR* siguiendo la estructura que se ve en la figura anterior (*Fig 1*). Si analizamos los productos implicados en el cálculo de una muestra de la salida, vemos que tenemos que calcula $\boldsymbol{h[0] * x[n]}, \boldsymbol{h[1] * x[n-1]}$, etc. Y tenemos que acumular el resultado de todos esos productos. Si pensamos que tenemos los datos almacenados en dos arrays, habría que multiplicar cada pareja de elementos y sumar todos los resultados. Todo ese cálculo nos ofrece una única muestra nueva a la salida del filtro.

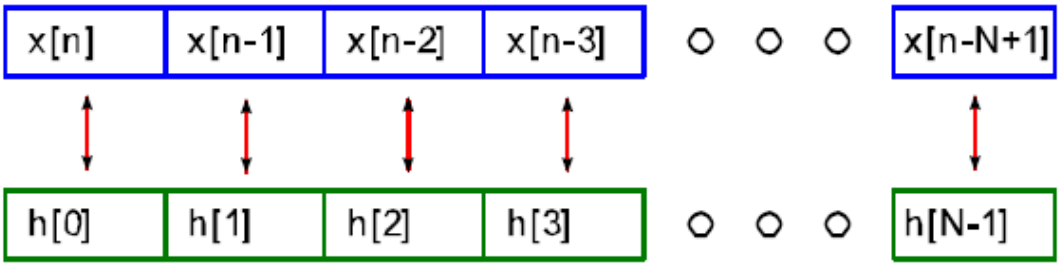


Fig 2: Estructura operaciones FIR.

Si tenemos las muestras dispuestas siguiendo el esquema anterior, es sencillo realizar los N productos con un ciclo "*for*", e ir acumulando los resultados. Pero mantener ordenado el vector de las últimas N muestras de entrada puede tener un coste computacional elevado.

GESTION DE LA MEMORIA

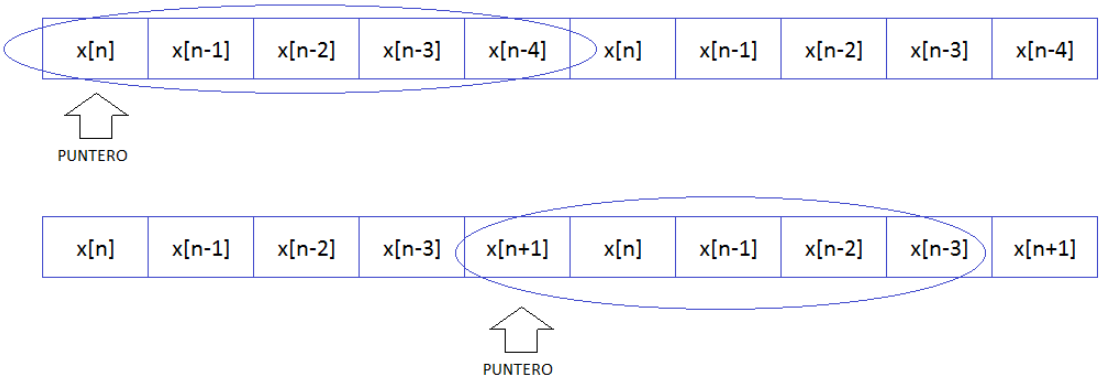


Fig 3: Gestión del buffer para filtro FIR de cuarto orden.

Para ser más eficientes creamos un buffer del doble del orden del filtro. En el esquema (*Fig 3*) vemos como el puntero pasa de apuntar la posición 0 a apuntar la posición 4, de ahí la posición del puntero irá decrementando hasta volver a la posición 0 añadiendo las nuevas muestras cada vez (x[n+1], x[n+2], x[n+3]...). En este caso las posiciones válidas serán entre las que apunta el puntero y las posiciones consecutivas hasta llegar a N posiciones después de la posición que apunta el puntero. De esta forma siempre tendremos la muestra actual en una posición inicial y seguidas las muestras anteriores ordenadas y la gestión de memoria se hace mucho más sencilla.

1.3 Filtro adaptativo

Los filtros adaptativos se utilizan mejor en los casos en que las condiciones de la señal o los parámetros del sistema están cambiando lentamente y el filtro se ajusta para compensar este cambio. También cuando no se conoce a priori el filtro a diseñar. Un filtro muy sencillo pero potente se llama *linear adaptive combiner*, que no es nada más que un filtro *FIR* ajustable. El criterio *LMS* es un algoritmo de búsqueda que puede utilizarse para ajustar los coeficientes del filtro.

1.3.1 Introducción

En los filtros digitales *FIR* e *IIR* convencionales, se supone que se conocen los parámetros del proceso para determinar las características del filtro. Pueden variar con el tiempo, pero se supone que la naturaleza de la variación es conocida. En muchos problemas prácticos, puede haber una gran incertidumbre en algunos parámetros debido a la insuficiencia de datos previos sobre el proceso. Se espera que algunos parámetros cambien con el tiempo, pero la naturaleza exacta del cambio no es predecible. En tales casos, es muy deseable diseñar el filtro para que sea autoadaptable.

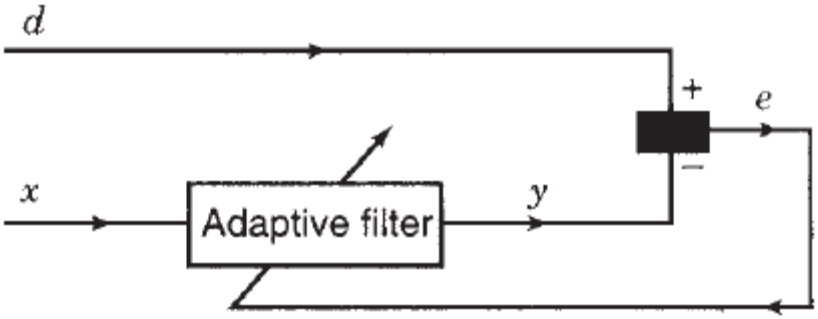


Fig 4: Estructura básica filtro adaptativo.

Los coeficientes de un filtro adaptativo se ajustan para compensar los cambios en la señal de entrada, la señal de salida o los parámetros del sistema. En lugar de ser fijo, un sistema adaptativo puede aprender de las características de la señal. Un filtro adaptativo puede ser muy útil cuando hay incertidumbre sobre las características de una señal o cuando cambian estas características.

Conceptualmente, el esquema adaptativo es bastante simple. La mayoría de los esquemas adaptativos pueden ser descritos por la estructura mostrada en la *Figura 4*. Ésta es una estructura básica de filtro adaptativo, en la que la salida del filtro adaptativo se compara con una señal deseada "*d"* para producir una señal de error "*e"* que se devuelve al filtro adaptativo. La señal de error se introduce en el algoritmo adaptativo, que ajusta el filtro variable para satisfacer algunos criterios o reglas predeterminadas. Una de las primeras preguntas que probablemente viene a la mente es: ¿Por qué estamos tratando de generar la señal deseada en "*y"* si ya tenemos esa señal deseada? Sorprendentemente, en muchas aplicaciones la señal deseada existe en alguna parte del sistema o se conoce a priori.

Los coeficientes del filtro adaptativo se pueden ajustar utilizando un algoritmo *LMS* basado en la señal de error. Aquí se utiliza solamente el algoritmo de búsqueda *LMS* con un combinador lineal (filtro *FIR*), aunque existen varias estrategias para realizar el filtrado adaptativo. La salida del filtro adaptativo en la *figura 4* es

$$\boldsymbol{y(n) = \sum_{k=0}^{N-1} w_k(n) x(n-k)}$$

donde $w_k(n)$ representa el peso k-ésimo de un filtro de N coeficientes para un tiempo específico n. Se necesita una medida de rendimiento para determinar cuán bueno es el filtro. Esta medida se basa en la señal de error,

$$\boldsymbol{e(n) = d(n) - y(n)}$$

que es la diferencia entre la señal deseada *d(n)* y la salida *y(n)* del filtro adaptativo. Los pesos o coeficientes $w_k(n)$ se ajustan de manera que se minimiza una función dependiente del valor cuadrático medio de la señal de error. Esta función dependiente del valor de error cuadrático medio es $E([e^2(n)]$, donde *E* representa el valor esperado. Como hay pesos o coeficientes, se requiere un gradiente de la función de error cuadrático medio. Se puede encontrar una estimación usando el gradiente de $e^2(n)$, produciendo:

$$\boldsymbol{w_k(n+1) = w_k(n) + 2\beta e(n) x(n-k) \qquad k = 0, 1, \ldots\ldots, N-1}$$

que representa el algoritmo LMS . La ecuación anterior proporciona un medio simple pero potente y eficiente de actualizar los pesos o coeficientes, sin necesidad de promediar o diferenciar, y se utilizará para implementar filtros adaptativos. La entrada al filtro adaptativo es *x(n)*, y la velocidad de convergencia y precisión del proceso de adaptación (tamaño de paso adaptativo) es $\boldsymbol{\beta}$.

Para cada instante *n*, cada coeficiente o peso $w_k(n)$ es actualizado o reemplazado por un nuevo coeficiente basado en la ecuación anterior, a menos que la señal de error e (n) sea cero. Después de la salida *y(n)* del filtro, se actualiza la señal de error *e(n)* y cada uno de los coeficientes $w_k(n)$ durante ese instante *n*. Se adquiere una nueva muestra (de un ADC) y se repite el proceso de adaptación en un instante diferente.

El *linear adaptive combiner* es una de las estructuras de filtro adaptativo más útiles y no es más que un filtro FIR ajustable. Mientras que los coeficientes del filtro FIR fijo son invariantes, los coeficientes o pesos del filtro FIR adaptativo pueden ser ajustados basados en un entorno cambiante siguiendo la señal de entrada. También se pueden usar filtros IIR adaptativos (no discutidos aquí). Un problema importante con un filtro IIR adaptativo es que sus polos pueden ser actualizados durante el proceso de adaptación a valores fuera del círculo unitario, haciendo que el filtro sea inestable.

1.3.2 Estructuras adaptativas

Se pueden utilizar varias estructuras adaptativas para diferentes aplicaciones en el filtrado adaptativo.

1. **Para la cancelación del ruido**. La *Figura 5* muestra la estructura adaptativo modificada para una aplicación de cancelación de ruido. La señal deseada **d** está corrompida por un ruido añadido no correlacionado **n**. La entrada al filtro adaptativo es un ruido **n'** que está correlacionado con el ruido **n**. El ruido **n'** podría provenir de la misma fuente que **n** pero modificado. La salida **y** del filtro adaptativo está adaptada al ruido **n**. Cuando esto sucede, la señal de error se aproxima a la señal deseada **d**. La salida global es esta señal de error y no la salida **y** del filtro adaptativo. Si **d** no está correlacionada con n, la estrategia es minimizar E (e^2),

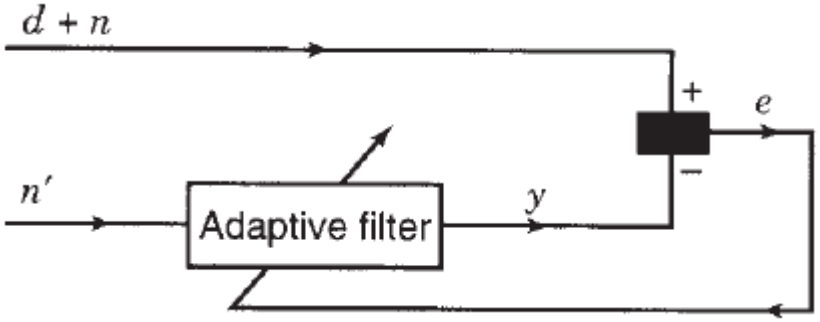


Fig 5: Estructura filtro adaptativo para la cancelación de ruido.

donde E () es el valor esperado. El valor esperado es generalmente desconocido; por lo tanto, se suele aproximar con un promedio en curso o con la función instantánea en sí. Su componente de señal, $E(d^2)$, no será afectado y sólo su componente de ruido $E[(n-y)^2]$ será minimizado.

2. **Para la identificación del sistema**. La *figura 6* muestra una estructura de filtro adaptativo que puede utilizarse para la identificación o modelado del sistema. La misma entrada se utilizan para un sistema desconocido en paralelo con un filtro adaptativo. La señal de error **e** es la diferencia entre la respuesta del sistema desconocido **d** y la respuesta del filtro adaptativo **y**. Esta señal de error es devuelta al filtro adaptativo y se utiliza para actualizar los coeficientes del filtro adaptativo hasta obtener la salida global **y = d**. Cuando esto sucede, el proceso de adaptación está terminado, y **e** se aproxima a cero. Si el sistema desconocido es lineal y no varía en el tiempo, después de que la adaptación esté completa, las características del filtro ya no cambian. En este esquema, el filtro adaptativo modela el sistema desconocido.

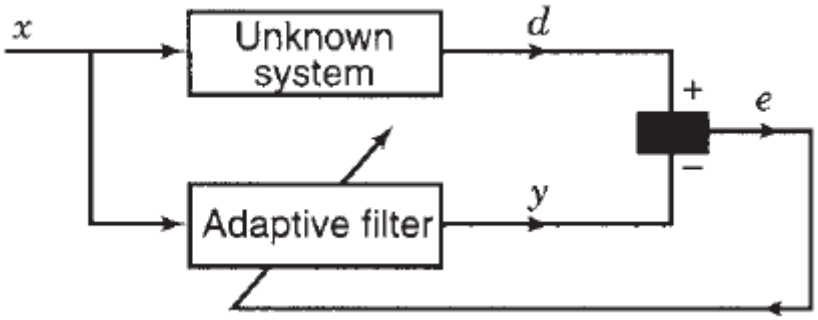


Fig 6: Estructura filtro adaptativo para sistema de identificación.

3. El predictor adaptativo. La Figura 7 muestra una estructura predictiva adaptativa que puede proporcionar una estimación de una entrada**.**

4. Estructuras adicionales, tales como:

   a) Cancelación con dos pesos, que se puede utilizar para cancelar o reducir una señal de ruido sinusoidal. Esta estructura tiene sólo dos pesos o coeficientes.

   b) Ecualización adaptativa de canales, utilizada en un módem para reducir la distorsión de canal resultante de la alta velocidad de transmisión de datos a través de los canales telefónicos**.**

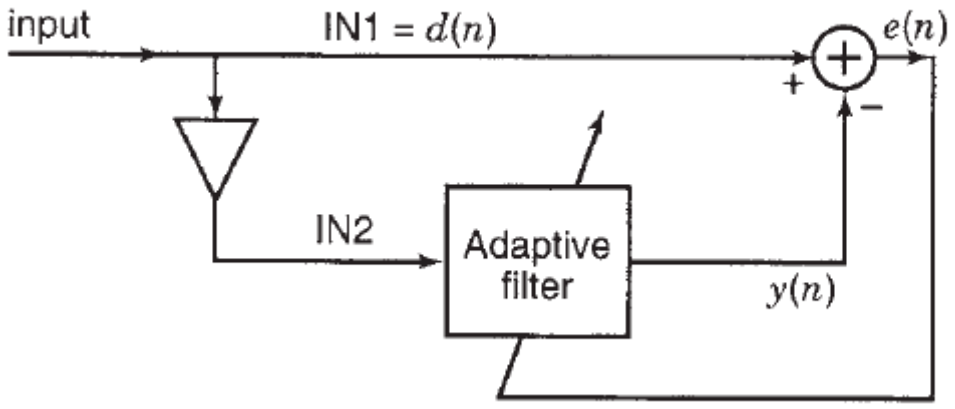


Fig 7: Estructura adaptativa para predicción.

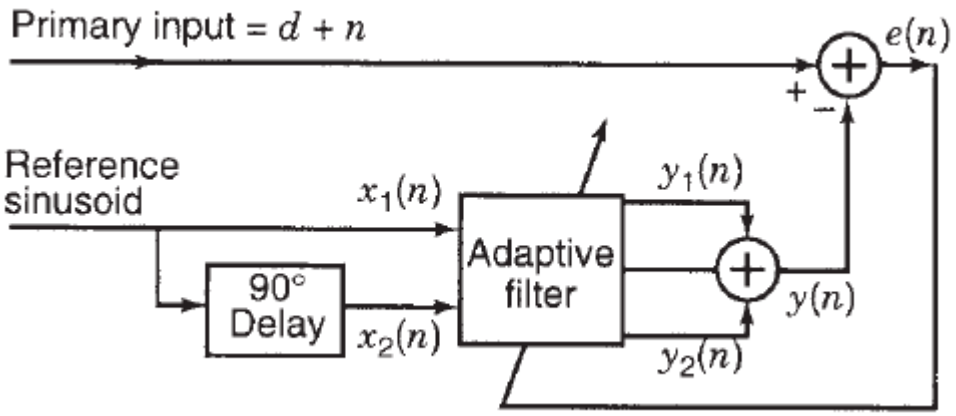


Fig 8: Estructura adaptativa con dos pesos.

1.3.3 Algoritmo LMS

El LMS es adecuado para una serie de aplicaciones, incluyendo la cancelación adaptable de eco y ruido, ecualización y predicción. Se han empleado otras variantes del algoritmo LMS, tales como el error de signo LMS, el signo-data LMS y el signo-signo LMS.

1. Para el algoritmo LMS de error de signo, se convierte en:

$$w_k(n+1) = w_k(n) + \beta \mathrm{sgn}[e(n)]x(n-k)$$

Donde *sgn* es la función *signum:*

$$\mathrm{sgn}(u) = \begin{cases} 1 \text{ si } u \geq 0 \\ -1 \text{ si } u < 0 \end{cases}$$

2. Para el algoritmo LMS de datos de signos, se convierte en:

$$w_k(n+1) = w_k(n) + \beta e(n)\mathrm{sgn}[x(n-k)]$$

3. Para el algoritmo LMS de signo-signo, se convierte en:

$$w_k(n+1) = w_k(n) + \beta \mathrm{sgn}[e(n)]\mathrm{sgn}\,[x(n-k)]$$

El cual se reduce a:

$$w_k(n+1) = \begin{cases} w_k(n) + \beta & \text{si } \mathrm{sgn}[e(n)] = \mathrm{sgn}[x(n-k)] \\ w_k(n) - \beta & \text{si no} \end{cases}$$

Que es más concisa desde un punto de vista matemático porque no se requiere operación de multiplicación para este algoritmo.

El algoritmo LMS ha sido muy útil en ecualizadores adaptativos, celulares telefónicos, etc. Otros métodos, como el algoritmo recursivo de mínimos cuadrados (RLS), pueden ofrecer una convergencia más rápida que el LMS básico, pero a expensas de más cálculos.

El RLS se basa en comenzar con la solución óptima y luego usar cada muestra de entrada para actualizar la respuesta de impulso con el fin de mantener esa solución óptima, se usan filtro adaptativos para hallar los coeficientes del filtro para obtener, de forma recursiva, el mínimo cuadrado de la señal de error. El tamaño del paso y la dirección correcta se definen en cada instante.

Para calcular el valor del parámetro β seguimos los valores que determinó Widrow, habitualmente se utiliza el 10% de ese valor para que se compense precisión y velocidad.

$$\beta = \frac{0.1}{P(x[n]) * N}$$

- **N número de coeficientes del filtro**
- **P(x[n]) potencia de la señal del entrada**

1.4 Cancelación activa de ruido

1.4.1 Estudios teóricos realizados

El control activo o cancelación activa de ruido es un sistema de anulación de un ruido que no deseamos.

El estudio se realiza a través de las ondas sonoras ya que es la forma en la que se transmite el ruido. Dos ondas sonoras pueden sumarse entre ellas y el resultado daría una sola onda que sería un sonido que el receptor escucharía, en el que caso de que esas ondas trabajasen en la misma frecuencia y espacio hablaríamos de una interferencia.

- **Interferencia constructiva**: es cuando hay dos ondas de frecuencia idéntica o similar y se superpone la cresta de una onda y la cresta de otra onda, los efectos se suman, y hacen una onda de mayor amplitud, las ondas han de estar en la misma fase.

- **Interferencia destructiva**: en este caso la cresta de una onda se superpone al valle de otra onda y se anulan. Dos ondas de igual frecuencia y amplitud en contrafase (desfase de 180º) que se interfieren, se anulan totalmente por un instante.

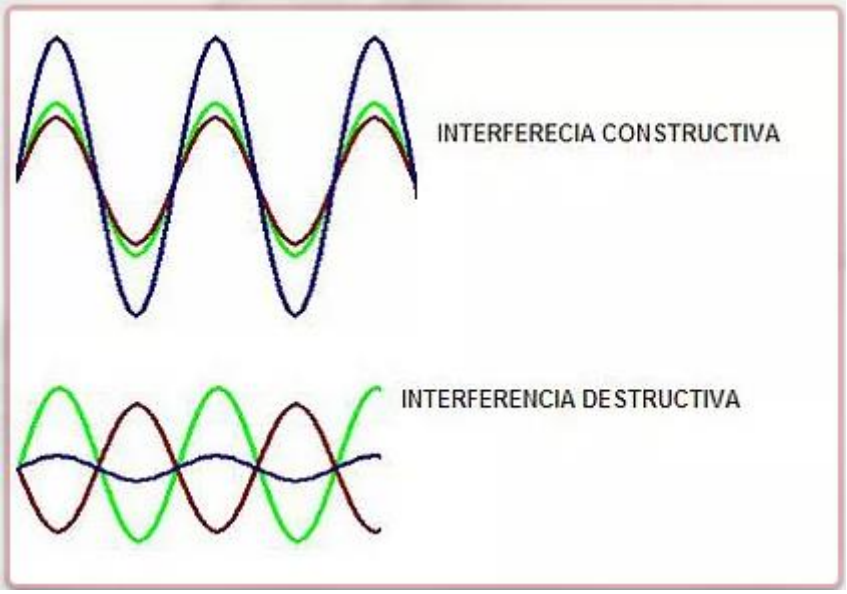


Fig 9: ejemplos de ondas constructivas y destructivas.

Entre los años 30 y 50 se realizan los primeros estudios sobre la cancelación activa de ruido pero no fue hasta los años 70 en los que se producen los grandes avances en el tema con los trabajos sobre los filtrados adaptativos. A finales de esos años la evolución tecnológica permite los primero descubrimientos sobre los procesadores digitales de la señal (DSP).

1.4.1.1 Otras técnicas utilizadas para reducir el ruido

- **Aislamiento acústico**

Se refiere al conjunto de materiales, técnicas y tecnologías desarrolladas para aislar o atenuar el nivel sonoro en un espacio determinado. Los métodos que se usan para conseguirlo son con la actuación sobre las paredes y ventanas.

Al incidir la onda acústica sobre un elemento constructivo (una pared por ejemplo) una parte de la energía se refleja, otra se absorbe y otra se transmite al otro lado. El aislamiento que ofrece el elemento es la diferencia entre la energía incidente y la energía transmitida, es decir, la suma de la parte reflejada y la parte absorbida. Se consigue mediante la desadaptación de la impedancia característica del medio entre el exterior y el interior del recinto, lo cual dificulta la propagación de la onda acústica.

El aislamiento acústico sirve cuando trabajamos con ruidos de alta y media frecuencia. Si quisiéramos conseguir reducción de ruido para bajas frecuencias, necesitaríamos barreras con mucha masa y no podríamos usarlo en muchas aplicaciones.

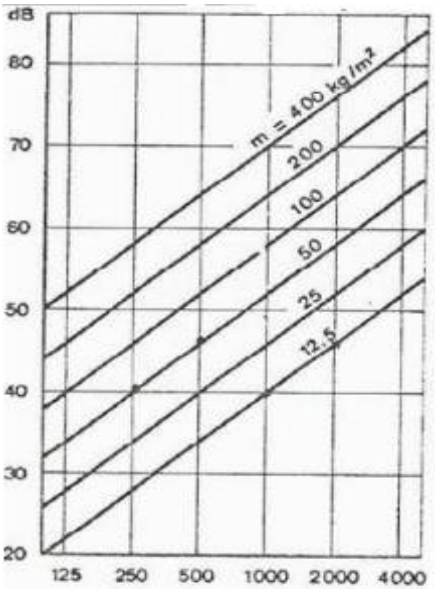


Fig 10: aislamiento en función de la frecuencia para barreras de distinta masa.

- **Acondicionamiento acústico**

Se trata de conseguir que un sonido que proviene de una fuente sonora sea irradiado por igual en todas las direcciones logrando un campo sonoro difuso ideal. Esta uniformidad no siempre se consigue, la acústica arquitectónica intenta aproximarse al máximo a este ideal a través de ciertas técnicas que aprovechan las cualidades de absorción , reflexión y difusión de los materiales constructivos de techos, paredes y suelos y de los objetos u otros elementos presentes en el recinto. De hecho, cosas tan aparentemente triviales como la colocación o eliminación de una moqueta, una cortina o un panel, son cruciales y pueden cambiar las condiciones acústicas de un recinto.

1.4.1.2 Principio de Huygens

"Todo punto de un frente de onda inicial puede considerarse como una fuente de ondas esféricas secundarias que se extienden en todas las direcciones con la misma velocidad, frecuencia y longitud de onda que el frente de onda del que proceden"

Huygens propuso que en cada punto alcanzado por una perturbación luminosa se convierta en una fuente de onda esférica. La suma de estas ondas secundarias determina la forma de la onda en cualquier momento posterior. Huygens supuso que las ondas viajaban únicamente "hacia delante" sin explicar en su teoría por qué este es el caso. Fue capaz de dar una explicación cualitativa de la propagación de la onda lineal y esférica, y de derivar las leyes de reflexión y la refracción con este principio, pero no pudo explicar las desviaciones de la propagación rectilínea que se producen cuando la luz se encuentra en bornes, aberturas y pantallas, comúnmente conocidos como efectos de difracción.

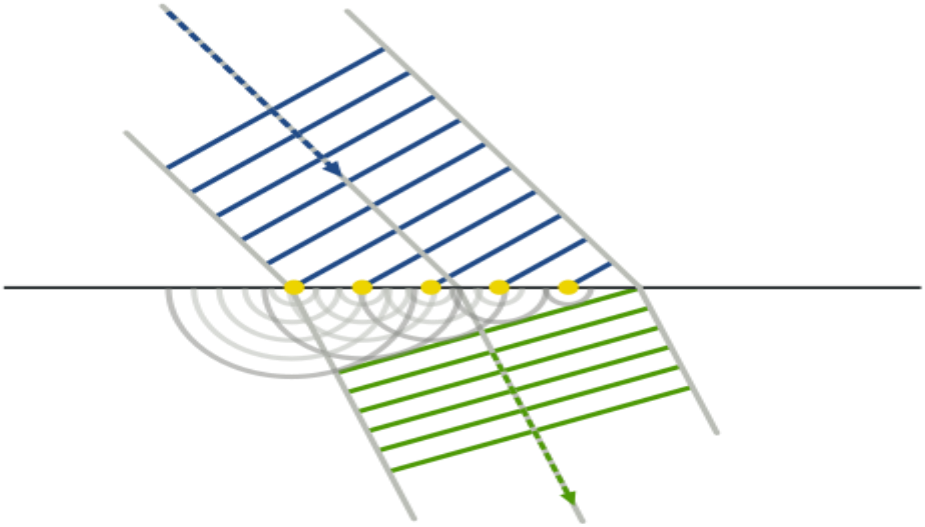


Fig11: Refracción en el principio de Huygens

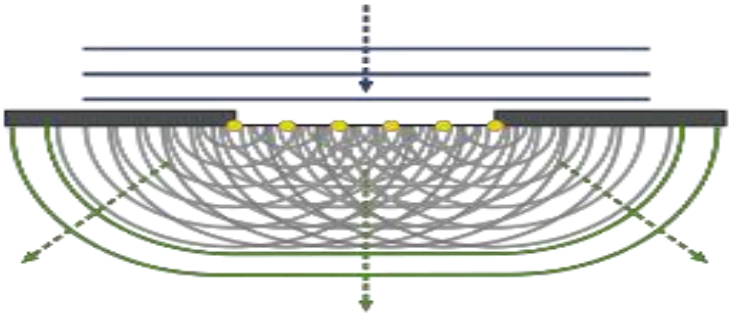


Fig 12: difracción en el principio de Huygens.

El campo acústico presente en un volumen sería posible imitarlo con infinitas fuentes puntuales situadas en la superficie que lo envuelve. De este modo, con la inversión de fase, se podría conseguir la cancelación completa del ruido dentro de ese entorno acústico. Si la fuente sonora estuviera dentro de ese entorno acústico la cancelación de ruido se produciría en el exterior.

Si usásemos un número finito de altavoces que estuviesen separados, esa separación determinaría la frecuencia máxima en la que la reducción efectiva de ruido se apreciaría.

1.4.1.3 Principio de Young

Young comprobó un patrón de interferencias en la luz procedentes de una fuente lejana al difractarse en el paso por dos rejillas, el resultado contribuyó a la teoría de la naturaleza ondulatoria de la luz. Posteriormente, la experiencia ha sido considerada fundamental a la hora de demostrar la dualidad onda corpúsculo, una característica de la mecánica

cuántica. El experimento también puede realizarse con electrones, protones o neutrones, produciendo patrones de interferencia similares a los obtenidos cuando se realiza con luz.

Aunque este experimento se presenta habitualmente en el contexto de la mecánica cuántica, fue diseñado mucho antes de la llegada de esta teoría para responder a la pregunta de si la luz tenía una naturaleza corpuscular o si, más bien, consistía en ondas viajando por el éter, análogamente a las ondas sonoras viajando en el aire. La naturaleza corpuscular de la luz es basada principalmente en los trabajos de Newton. La naturaleza ondulatoria, en los trabajos clásicos de Hooke y Huygens.

Los patrones de interferencia observados restaban crédito a la teoría corpuscular. La teoría ondulatoria se mostró muy robusta hasta los comienzos del siglo XX, cuando nuevos experimentos empezaron a mostrar un comportamiento que sólo podía ser explicado por una naturaleza corpuscular de la luz. De este modo el experimento de la doble rendija y sus múltiples variantes se convirtieron en un experimento clásico por su claridad a la hora de presentar una de las principales características de la mecánica cuántica.

La formulación original de Young es muy diferente de la moderna formulación del experimento y utiliza una doble rendija. En el experimento original un estrecho haz de luz, procedente de un pequeño agujero en la entrada de la cámara, es dividido en dos por una tarjeta de una anchura de unos 0.2 mm. La tarjeta se mantiene paralela al haz que penetra horizontalmente, es orientado por un simple espejo. El haz de luz tenía una anchura ligeramente superior al ancho de la tarjeta divisoria por lo que cuando ésta se posicionaba correctamente el haz era dividido en dos, cada uno pasando por un lado distinto de la pared divisoria. El resultado puede verse proyectado sobre una pared en una habitación oscurecida. Young realizó el experimento en la misma reunión de la Royal Society mostrando el patrón de interferencias producido demostrando la naturaleza ondulatoria de la luz.

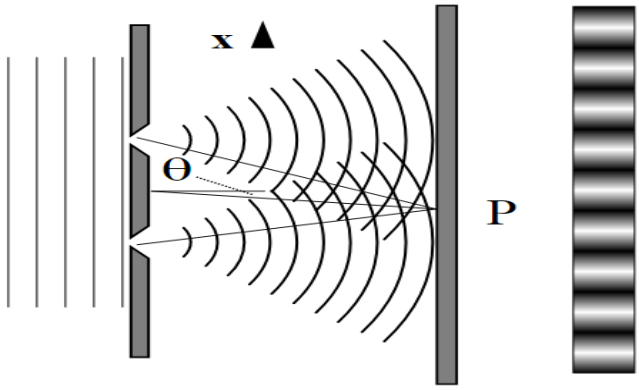


Fig 13: Ejemplo del experimento de Young.

1.4.2 Diseño trabajado para la reducción de ruido

En la figura se muestra el esquema general que se ha seguido en este proyecto para el diseño del nodo acústico que permita la cancelación activa de ruido. Se ha seguido el esquema básico del filtrado adaptativo pero realizando ajustes para que cumplan nuestras necesidades.

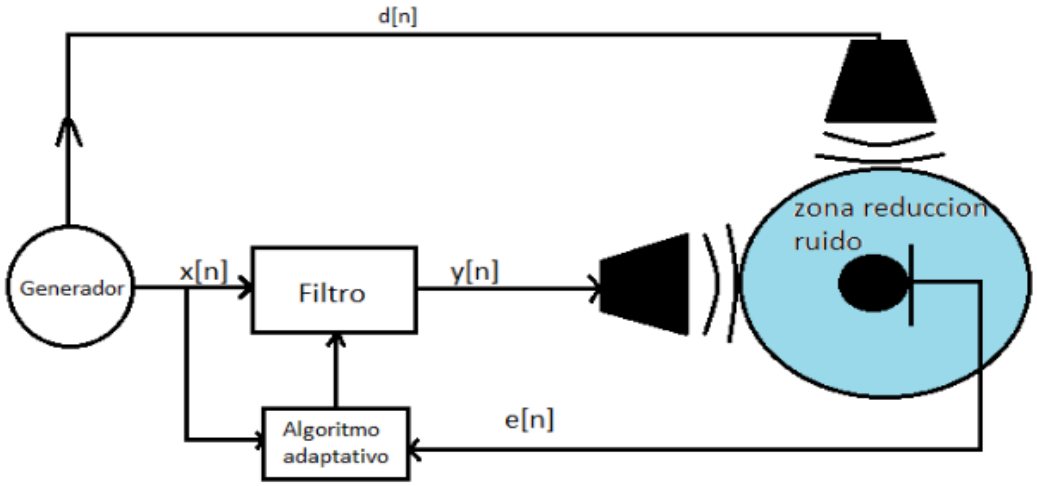


Fig 14: Estructura seguida para cancelación activa de ruido

- **x[n]: muestras de la señal de entrada al filtro adaptativo proveniente de la fuente de ruido.**
- **y[n]: muestras de salida del filtro FIR a partir de la entrada x[n].**
- **d[n]: señal deseada.**
- **e[n]: señal error, referencia del campo acústico de la zona de silencio. Como hemos visto en el punto anterior e[n]=d[n]-y[n]**

Hay dos requisitos importantes que se han de seguir a la hora de trabajar en la cancelación de ruido. El primero es que debe existir correlación entre la señal de entrada al filtro y la señal que deseamos cancelar, en este caso se cumple ya que ambas señales provienen de la misma fuente de sonido. La segunda es que se debe garantizar que la señal x[n] esté adelantada siempre con respecto a la señal d[n] para la causalidad del filtro.

Es por esto que el micrófono de referencia de ruido, se ha de colocar lo más cerca posible a la fuente de ruido que se desea eliminar. Esto hace cumplir la causalidad del filtro FIR y reduce la probabilidades de que señales que no queremos que se introduzcan en nuestro sistema de filtrado aparezcan.

Existen una serie de condiciones acústicas que hay tener en cuenta a la hora de diseñar nuestro nodo acústico. Un desfase temporal entre dos señales acústicas hace que la diferencia de fase (radianes) entre ellas incremente linealmente con la frecuencia. Si trabajamos con frecuencias bajas esa diferencia no es muy significativa pero si por el contrario hablamos de frecuencias elevadas sí que puede afectar a nuestro resultado final.

Condiciones acústicas:

Frecuencia (Hz)	**Esfera de silencio Diámetro (cm)**
100	34
500	7
> 500	**Prácticamente inapreciable**

En la tabla anterior observamos que cuanto más aumenta la frecuencia a la que trabaja la fuente de ruido el diámetro de la esfera en la que se produciría la cancelación de ruido disminuye. Es por ello que es conveniente realizar un filtro paso bajo con una frecuencia de corte de 500 Hz en la señales x[n], e[n] y y[n].

Si el altavoz está lo suficientemente cerca de la fuente de ruido y trabajamos con bajas frecuencias la esfera de silencio tendrá un diámetro mucho mayor.

Como hemos visto anteriormente en algoritmo LMS la señal de error e[n] viene de la resta de las señales y[n] y d[n], por lo tanto es como si hubiera un sistema H(z). Por la presencia de este sistema H(z) es necesario realizar modificaciones en el algoritmo LMS y que se garantice que se pueda trabajar con estabilidad. Es por ello que utilizamos el algoritmo filtrado-x LMS, el cual compensa la presencia de este sistema.

1.4.3 Filtro adaptativo con filtrado-x LMS.

Vemos en la *Figura 15* que se ha añadido un bloque H(z) a la entrada del algoritmo LMS, ese bloque tiene la misma respuesta en frecuencia que la respuesta en frecuencia del camino acústico entre el altavoz y el sensor de error del sistema cancelador. Así el sistema es mucho más estable y funciona mejor. Esto de define como filtro-x LMS.

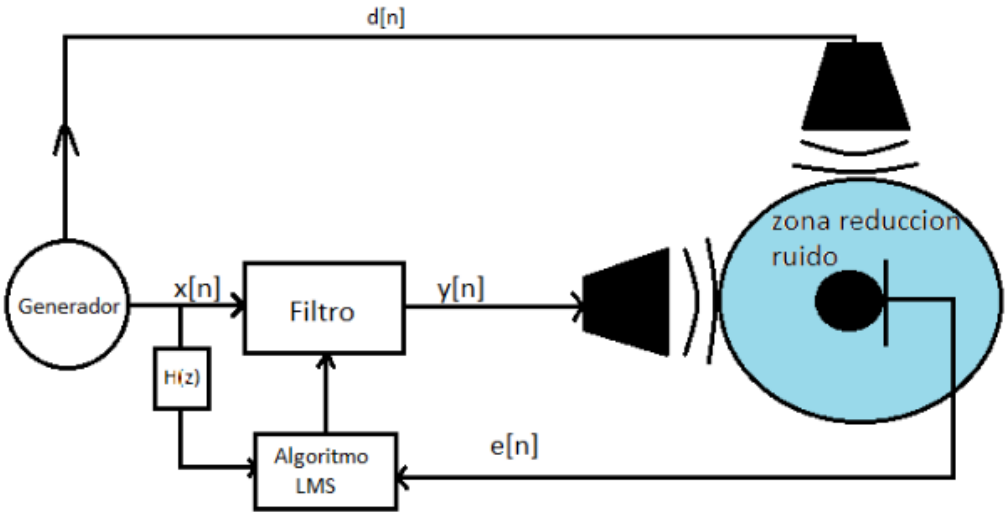


Fig 15: Estructura para control activo de ruido usando el algoritmo fxlms

Utilizando el esquema mostrado en la figura, el sistema H(z) está compensando con el filtro a la entrada, la ecuación del algoritmo LMS se tiene que modificar para incluir la señal x filtrada quedando de la siguiente forma:

$$h[k]_{n+1} = h[k]_n + 2\,\mu\,e[n]\,x_h\,[n-k]$$

Siendo x_h la señal x filtrada.

Estimación del sistema H(z):

Podríamos realizar dos métodos diferentes para el cálculo de los valores de H(z), uno de ellos en el que calcularíamos primero los valores de H(z) siguiendo una estructura de identificación del sistema y luego realizaríamos la cancelación del ruido. El problema de este método sería que cualquier cambio en el entorno acústico supondría un cambio en resultado final que llevaría a una mala cancelación del ruido.

Por eso también se implementa un método en el cual se calculan los valores para H(z) en el mismo momento en el que se realiza la reducción de ruido. Es por esto que al esquema general para la cancelación activa de ruido le añadimos una serie de ramas con filtros que sus usos se describen a continuación.

- **Filtro adaptativo C**: Es el filtro principal que se usa para la reducción de ruido.
- **Filtro adaptativo H**: Sirve para calcular los coeficientes que se utilizarán en el sistema H(z), es decir identifica el sistema, lo que nos crearía un sesgo con

respecto a H(z) ya que este sistema del filtro H es estimado. Este filtro toma como entrada la salida del filtro. Ya que la señal d[n] atraviesa el sistema a identificar, esta varía, así que añadimos una rama a la salida de la reducción de ruido que contendría el valor de d[n] que sumaríamos a la que generada como ruido cancelador.

- **Filtro adaptativo D**: Sirve para eliminar el sesgo entre el sistema H(z) y el filtro H que estima sus coeficientes.

Como hemos comentado antes podríamos operar de dos formas diferentes: Identificando previamente el sistema o hacerlo durante la cancelación. Ya que estamos utilizando el esquema que hemos descrito y podemos ver en la figura, tendríamos que operar de dos formas distintas. La primera identificando previamente el sistema, en la segunda reducimos el valor de β del filtro C para que su tiempo de adaptación sea mayor y que no se desestabilice el sistema.

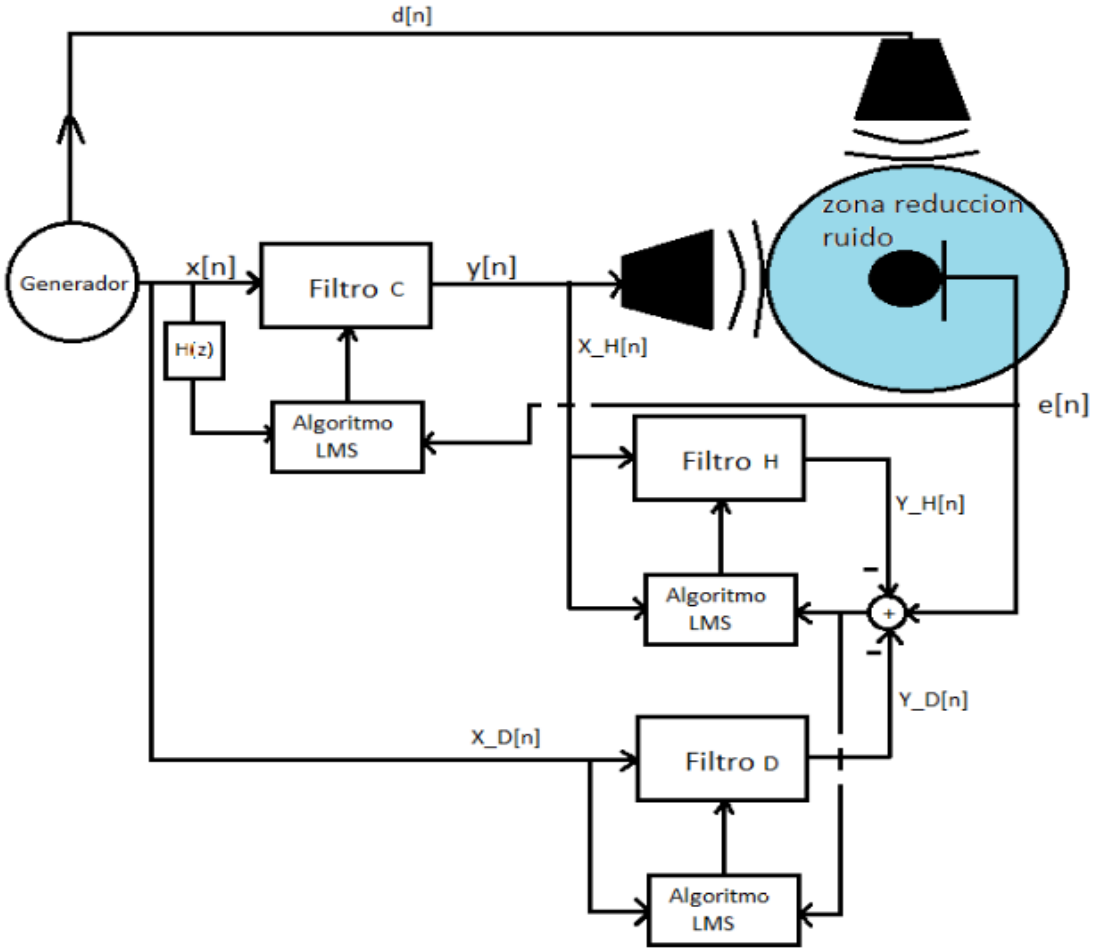


Fig 16: Estructura para la estimación de H(z).

Capítulo 2. Objetivo del proyecto

El objetivo de este proyecto es el diseño de un nodo acústico para el control activo de ruido, para ello disponemos de una tarjeta DSP TMS320C5505 de Spectrum Digital la cual describiremos y evaluaremos más adelante.

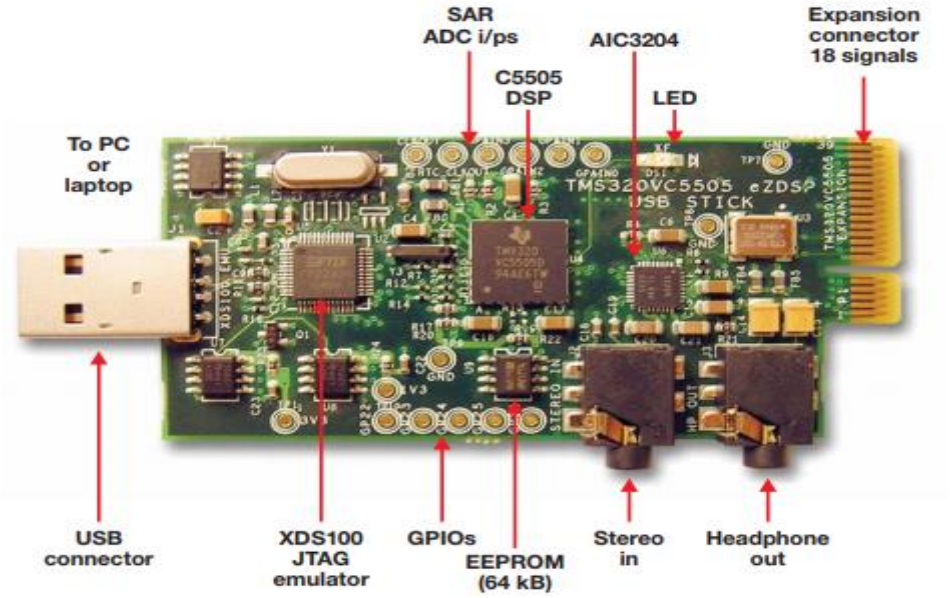


Fig 16:USBSTK5505

Las características del nodo acústico son:

- Tener suficiente capacidad computacional para ejecutar en tiempo real algoritmos complejos de procesado de audio
- Ser capaz de manejar señales de audio en tiempo real, tanto en captación como en reproducción, con la menor latencia posible. Idealmente en tiempo de muestra.

Metodología de trabajo:

- Documentarse acerca de la DSP. Ver las posibilidades que ofrece incluyendo el manejo de las demos
- Primeras pruebas:
  - Software que capte y genere muestras continuamente. Comprobar la latencia que tiene.
- Añadir procesado. Implementar un filtrado sencillo: Retardo, filtro con coeficientes aleatorios, etc… Comprobar la generación de señales.
- Diseño de un sistema de identificación. Implementar un filtro adaptativo que permita identificar el canal entro dos puntos de la sala. Para ello debe ser capaz de generar un ruido aleatorio mediante el altavoz en uno de los extremos y en el otro captar señal mediante el micrófono.
- Implementar un algoritmo con filtrado-x. Esta estructura particular permite desarrollar los algoritmos adaptativos para aplicaciones de control de sonido.

Capítulo 3. Descripción DSP

El dispositivo es miembro de la familia de productos TMS320C5000 de punto fijo de procesador de señal digital (DSP) y está diseñado para aplicaciones de baja potencia.

El DSP de punto fijo se basa en el núcleo del procesador de CPU de generación DSP TMS320C55x . La arquitectura DSP de C55x consigue un alto rendimiento y baja potencia gracias a un mayor paralelismo y a un enfoque total en el ahorro de energía. La CPU admite una estructura de bus interna que está compuesta por un bus de programa, un bus de lectura de datos de 32 bits y dos buses de lectura de datos de 16 bits, dos buses de escritura de datos de 16 bits y buses adicionales dedicados a la actividad periférica y DMA. Estos buses proporcionan la capacidad de realizar hasta cuatro lecturas de datos de 16 bits y dos escrituras de datos de 16 bits en un solo ciclo. El dispositivo también incluye cuatro controladores DMA, cada uno con 4 canales, proporcionando movimiento de datos para 16 contextos de canal independientes sin intervención de la CPU. Cada controlador DMA puede realizar una transferencia de datos de 32 bits por ciclo, en paralelo e independiente de la actividad de CPU.

La CPU C55x proporciona dos unidades de multiplicación acumulada (MAC), cada una capaz de multiplicar 17 bits x 17 bits y agregar 32 bits en un solo ciclo. Una unidad aritmética / lógica (ALU) central de 40 bits está soportada por una ALU adicional de 16 bits. El uso de las ALUs está bajo control de conjunto de instrucciones, proporcionando la capacidad de optimizar la actividad paralela y el consumo de energía. Estos recursos se gestionan en la unidad de direcciones (AU) y en la unidad de datos (DU) de la CPU C55x.

La CPU C55x admite un conjunto de instrucciones de anchura de byte variable para mejorar la densidad del código. La unidad de instrucción (UI) realiza búsquedas de programa de 32 bits desde la memoria interna o externa y las instrucciones de colas para la Unidad de Programa (PU). La Unidad de Programa decodifica las instrucciones, dirige las tareas a los recursos de la Unidad de Direcciones (AU) y de la Unidad de Datos (DU), y gestiona el canal totalmente protegido.

Las funciones de entrada y salida de propósito general, junto con el ADC de SAR de 10 bits, proporcionan suficientes clavijas para el estado, las interrupciones y la E / S de bits para pantallas LCD, teclados e interfaces de medios. Soporte de medios serie a través de dos periféricos MultiMedia Card / Secure Digital (MMC / SD), cuatro módulos Inter-IC Sound (I2S Bus ™), una Interfaz de Puerto Serial (SPI) con hasta 4 chips selectos, una interfaz I2C multimaestro , y una interfaz de receptor / transmisor universal asíncrono (UART).

El conjunto periférico de dispositivo incluye una interfaz de memoria externa (EMIF) que proporciona acceso sin cola a memorias asíncronas como EPROM, NOR, NAND y SRAM, así como a memorias de alta densidad y alta velocidad, tales como DRAM síncrona (SDRAM) y móvil SDRAM (mSDRAM). Los periféricos adicionales incluyen: un modo de dispositivo Universal Serial Bus (USB2.0) de alta velocidad solamente y un reloj en tiempo real (RTC). Este dispositivo también incluye tres temporizadores de uso general con uno configurable como temporizador de vigilancia y un generador de reloj analógico de bloqueo de fase (APLL).

Las características del TMSC5505eZdsp incluyen entre otras:

- Procesador Digital de Señal de **punto fijo**
- Frecuencias de reloj : **60, 75, 100, 120, 150 MHz**
- Una o dos instrucciones ejecutadas por ciclos.
- 320K Bytes RAM compuestas por :
    - 64K Bytes de Dual-Access RAM (DARAM)
    - 256K Bytes de Single-Access RAM (SARAM)

- 128K Bytes ROM
- Receptor y transmisor asíncronos universal
- Interfaz de Puerto serie (SPI)
- Dispositivo USB 2.0 de alta velocidad

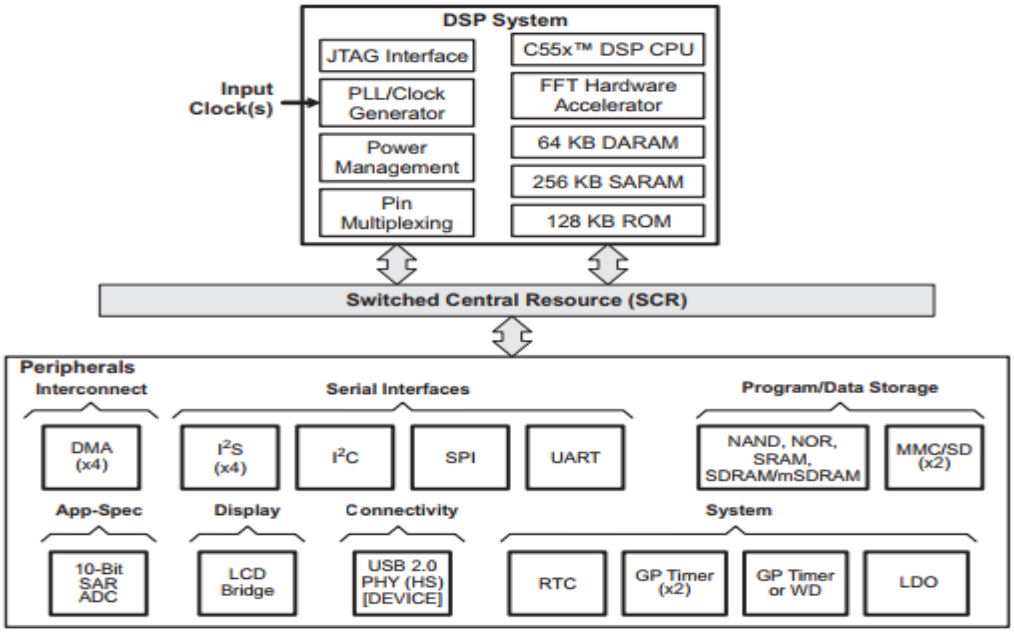


Fig 17: Diagrama de flujo de USBSTK5505

Aplicaciones

• Dispositivos de audio inalámbricos (auriculares, micrófonos, altavoces)
• Auriculares de cancelación de eco
• Dispositivos Médicos Portátiles
• Aplicaciones de voz
• Controles Industriales
• Biometría de Huellas Digitales
• Radio Definida por Software

Capítulo 4. Instrumentación

4.1 Software utilizado

Tras la compra de nuestra DSP USBSTK5505 la primera tarea que se ha de realizar es la comprobación de la misma. Para ello el proveedor de la misma (Spectrum Digital) nos proporciona en la compra de la tarjeta un CD que incluye el programa Code Composer Studio v.4.

➢ Code Composer Studio v.4

Code Composer Studio es un entorno de desarrollo integrado (IDE) que admite la cartera de microcontroladores y procesadores embebidos de Texas Instruments. Code Composer Studio incluye un conjunto de herramientas utilizadas para desarrollar y depurar aplicaciones embebidas. Incluye un compilador C / C ++ optimizador, editor de código fuente, entorno de creación de proyecto, depurador, generador de perfiles y muchas otras herramientas.

Code Composer Studio combina las ventajas del framework de software Eclipse con capacidades avanzadas de depuración integradas de Texas Instruments, resultando un entorno de desarrollo atractivo para desarrolladores incrustados.

➢ ARTA

Programa para medida de respuesta a impulsos y análisis en tiempo real de espectros y respuesta en frecuencia. Es un programa gratuito que ya había utilizado antes en clase por lo que conocía su funcionamiento y me resultaba más fácil trabajar con él.

4.2 Hardware utilizado

En esta parte del trabajo vamos a describir el material que he utilizado en este proyecto, a parte de la tarjeta USBSTK5505 que contiene la DSP que hemos detallado con anterioridad en capítulo 3. Con este material hemos realizado las simulaciones que se comentan más adelante en el siguiente capítulo del trabajo:

- Altavoces Logitech Z200 Multimedia con sonido estéreo 2.0 y 10 vatios de potencia. Tiene salida mini-Jack que es la que necesitamos para utilizar la salida de nuestra tarjeta, además de conexión para auriculares.

Fig 18: Altavoces utilizados

- Micrófono Vivanco DM10, salida Mono. La salida de nuestro micrófono es mini-Jack, por lo que podemos usarlo directamente a la entrada de nuestra tarjeta. En la compra también venía un adaptador mini-Jack→Jack que ha sido utilizado para introducir la señal que capta el micrófono en la tarjeta de sonido y realizar medidas. Por ejemplo cuando queremos captar la señal de error resultante del filtro-x.

Fig 19: Micrófono utilizado

- Cables de conexión RCA-miniJack, estos cables los utilizo para la conexión entre la tajeta de sónido y la USBSTK5505. Las salidas que utilizamos en la tarjeta de sonido son del tipo RCA, sin embargo las entradas de la tarjeta de sonido que utilizamos son Jack, es por eso que utilizamos los adaptadores RCA→Jack.

Fig 20: Cableado utilizado

- Tarjeta de sonido Fast Track Pro de M-AUDIO. El cable de alimentación es USB y sirve tanto para alimentar la tarjeta de sonido como para la transmisión de datos con el PC. Solo hay que descargarse los drivers en el PC para configurar el ARTA y poder leer los datos de la tarjeta de sonido.

Fig 21: Tarjeta de sonido

Capítulo 5. Implementación del código

La implementación de código para nuestro nodo acústico se ha realizado guardando los las distintas funciones en archivos .c, que se explican más adelante, y las estructuras utilizadas en los archivos de cabecera .h.

Se ha utilizado archivos que te facilitan la página de Spectrum Digital, los tipos de datos se han definido en el archivo de cabecera *usbstk5505.h* que son un tipo de datos de número enteros en como fija, así como el archivo de comandos de enlazado *lnkx.cmd*. También el archivo para habilitar el códec aic3204 en el que hemos realizado algunos cambios para que se ajuste a nuestras especificaciones. La habilitación del puerto serie I2C se implementa en el archivo *usbstk5505_i2c.h*.

En las imágenes aparecen algunas partes del código que muestran las estructuras y sistemas utilizados.

5.1 Códec aic3204

Esta función habilita el códec de audio de nuestra tarjeta, para ello se han descrito una serie de funciones para la lectura y escritura del códec. Se podrán elegir diferentes frecuencias de muestreo, para las pruebas realizadas se ha utilizado la frecuencia de muestreo de 8 kHz que es la menor frecuencia de muestreo que permite la tarjeta. Podemos elegir como queremos que sea la entrada de audio (estéreo o diferencial) o elegir la ganancia o resistencia de entrada.

En el archivo se realizan las siguientes funciones:

- **Init_aic3204**: Iniciamos el códec con los parámetros establecidos.
- **Desactivar_aic3204**: Desactivamos el códec.
- **Lectura_CH0**: Lee las muestras que entran por el canal izquierdo del Jack.
- **Lectura_CH1**: Lee las muestras que entran por el canal derecho del Jack.
- **Lectura**: lee las muestras de ambos canales.
- **Escritura_CH0**: Envía el valor de la muestra que entra por el canal izquierdo.
- **Escritura_CH1**: Envía el valor de la muestra que entra por el canal derecho.
- **Escritura**: Envía los valores de las muestras que entran por ambos canales.

```
#include "usbstk5505_i2c.h"
#include "usbstk5505.h"

//-----------------Frecuencias de muestreo--------------------//
#define FS_8KHZ     8000
#define FS_16KHZ    16000
#define FS_24KHZ    24000
#define FS_32KHZ    32000
#define FS_44_1KHZ  44100
#define FS_48KHZ    48000

//-----------------ganancia de los amplificadores----------------//
#define Ganacia_IN_max      47
#define Ganancia_IN_min     0
#define Ganancia_OUT_max    29
#define Ganancia_OUT_min    0
//---------------------Resistencia de entrada-----------------//
#define resistencia_40K 3

//Configuración de la entrada jack
#define IN2_diferencial 0x2D    //Diferencial
#define IN2_estereo     0x22    //Estéreo

//Dirección del módulo I2C
#define AIC3204_I2C_ADDR    0x18

//Banderas de ADC/DAC listo
#define RcvR 0x08
#define RcvL 0x04
#define XmitR 0x20
#define XmitL 0x04

void init_AIC3204 (Uint16 fs, Uint8 resistencia_entrada,
                   Uint8 *ganancia_entrada, Uint8 config_entrada,
                   Int8 *ganancia_salida);
void desactivar_AIC3204 (void);

void lectura_CH0 (Int16 *sample);
void lectura_CH1 (Int16 *sample);
void lectura (Int16 *sample0, Int16 *sample1);

void escritura_CH0 (Int16 sample);
void escritura_CH1 (Int16 sample);
void escritura (Int16 sample0, Int16 sample1);
```

Fig 22: codec_aic3204.h

```
#include "codec_aic3204.h"

void lectura_CH0 (Int16 *sample){
while((RcvL & I2S0_IR) == 0);   // Esperamos iterrupcion
*sample = I2S0_W0_MSW_R;    // leemos los 16 primeros bits (izq)
}

void lectura_CH1 (Int16 *sample){
while((RcvR & I2S0_IR) == 0);   // esperamos interrupción
*sample = I2S0_W1_MSW_R;    // leemos los 16 segundos bits (der)
}
```

```
void escritura (Int16 sample0, Int16 sample1){
while((XmitR & I2S0_IR) == 0);  //Esperando interrupción
I2S0_W0_MSW_W = sample0;    // escribimos muestra en canal izquierdo
I2S0_W1_MSW_W = sample1;    // escribimo muestra en canal derecho
}
```

Fig 23: codec_aic3204.c

5.2 Herramientas

Este archivo se creó para usar una serie de herramienta que necesitamos para las simulaciones entre las que están:

- **Generador_tono:** Inicializa el generador de tonos. Generador de tonos implementados con un filtro FIR resonante de segundo orden.
- **Obtener_muestra_tono:** para obtener una muestra del generador de tonos.
- **Generador_muestra**: Inicializad de generador de tonos aleatorios. Generador de un ruido aleatorio implementado mediante un algoritmo iterativo.
- **Obtener_muestra_ruido**: para obtener una muestra del generador de tonos aleatorios
- **Medida_potencia**:inicializar medidor de potencia.
- **Obtener_potencia**: obtener la potencia de una señal.
- **Potencia_fijo_a_flotante**: potencia de la señal de como fija a coma flotante.
- **Potencia_flotante_a_fijo**: potencia de una señal de como flotante a coma fija.

```
#include "filtro_FIR.h"

    //Generador de ruido aleatorio
    typedef struct{
        Uint16 aleatorio_sample; //16 bit de la muestra actual
        Uint16 amp; //En formato Q1.15 amplitud del señal
    }generador_ruido_aleatorio;

generador_tono generador_tono_1 (float dBFS, Uint16 fc, Uint16 fs);
Int16 obtener_muestra_tono (generador_tono *gen);
generador_ruido_aleatorio generador_ruido_aleatorio_1(float dBFS, Uint16 seed);
Int16 obtener_muestra_ruido (generator_ruido *gen);

medida_potencia medida_potencia_1(float a);
Uint32 obtener_potencia (Int16 sample, Uint32 previous_power, medida_potencia medida);
float potencia_fijo_a_flotante(Uint32 potencia);
Uint32 potencia_flotante_a_fijo(float potencia );
```

Fig 24: herramientas.h

```c
#include "herramientas.h"
generador_tono generador_tono_1 (float dBFS, Uint16 fc, Uint16 fs){
    generador_tono gen;
    float w = fc*2*PI/fs;

    float float_h[2];
    float float_mem[4];

    float a1 = pow(10,dBFS/20);
    float a2 = 1.0*MAX_16BITS/abs(MIN_16BITS);
    float a = a1*a2;

    float_h[0] = 2*cos(w);
    float_h[1] = -1;

    float_mem[0] = a;
    float_mem[1] = a*cos(w);
    float_mem[2] = a;
    float_mem[3] = a*cos(w);


    gen.FIR = iniciamos_FIR(float_h,2,14,&gen.h,&gen.buff);

    fijo(float_mem, 4, 15, gen.buff.data);

    return gen;
}
float potencia_fijo_a_flotante(Uint32 potencia){
    return potencia*pow(2,-30);
}
Uint32 potencia_flotante_a_fijo(float potencia){
    return round(potencia*pow(2,30));
}
generador_ruido_aleatorio generador_ruido_aleatorio_1 (float dBFS,
                                                       Uint16 seed){
    generador_ruido_aleatorio gen;
    gen.aleatorio_sample = seed;
    gen.amp = round(pow(10,dBFS/20)*pow(2,15));
    return gen;
}
```

Fig 25: herramientas.c

5.3 Gestión de la memoria

Esta función gestiona el almacenamiento de los buffers que posteriormente utilizaremos para los filtros. Necesitaremos definir una variable utilizaremos como puntero de escritura, el tamaño de la ventana, el número de desplazamiento que recorreremos desde el puntero señalado y el modo de almacenamiento del buffer.

Modos de almacenamiento del buffer:

- *Buffer simple*: el buffer será igual a la cantidad de datos almacenados en el array.
- *Buffer doble*: la ventana es igual a la mitad de los datos almacenados en el array.
- *Buffer vacio*.

Funciones implementadas:

- **Nuevo_buffer:** crea un nuevo buffer en el que definiremos su tamaño y modo de funcionamiento
- **Fin_buffer:** vacia y deshabilita el buffer.
- **reset_buffer:** vacia los datos del buffer.
- **Introducir:** posiciona un nuevo buffer en la posición en la que estamos.
- **Coger:** proporciona el valor del dato en la posición que elijamos.
- **Pos_ventana:** posición en la que está un dato.
- **Siguiente_muestra:** resta 1 al valor del desplazamiento (offset) y nos dice la posición del siguiente dato.
- **Introducir_dato:** resta 1 al desplazamiento y añade un dato.
- **Obtener_dato:** resta 1 al desplazamiento y proporciona un dato
- **Float_a_16:** saturación de los datos float a 16 bits.
- **32_a_16:** saturación de datos de 32 a 16 bits.
- **Fijo:** convierte los datos de coma flotante a fija.

```
#include "usbstk5505.h"
#include "stdlib.h"
#include "math.h"
//--------------------VARIABLES---------------///
#define N_BUFFER        0
#define SIMPLE_BUFFER   1
#define DOBLE_BUFFER    2

#define MIN_16BITS  -32768
#define MAX_16BITS   32767
//--------------------ESTRUCTURA---------------///
typedef struct{
    Uint8 modo;     //Modo de almacenamiento de datos
    Uint16 ventana; //Tamaño de la ventana de los datos almacenados
    Int16 *datos;   //Puntero al array de datos
    Uint16 offset;  //Posición del dato actual dentro del array
}buffer;
```

Fig 26: gestion_memoria.h

```
#include "gestion_memoria.h"
buffer nuevo_buffer (Uint16 buffer_ventana, Uint8 modo){
    buffer buff;
    switch (modo){
        case SIMPLE_BUFFER:
            buff.modo   = SIMPLE_BUFFER;
            buff.ventana = buffer_ventana;
            buff.datos  = calloc(buffer_ventana,sizeof(Int16));
            buff.offset = 0;
            break;
        case DOBLE_BUFFER:
            buff.ventana    = DOBLE_BUFFER;
            buff.ventana    = buffer_ventana;
            buff.data   = calloc(2*buffer_ventana,sizeof(Int16));
            buff.offset = 0;
            break;
        default:
            buff.modo    =  N_BUFFER;
            buff.ventana =  N_BUFFER;
            buff.datos   =  N_BUFFER;
            buff.offset  =  N_BUFFER;
    }

    return buff;
}


Int16* Introducir_dato (Int16 nuevo_valor, buffer buff){
    Int16 *mem;

    switch (buff.modo){
        case SIMPLE_BUFFER:
            mem = &buff.data[buff.offset];
            mem[0] = nuevo_valor;
            break;
        case DOBLE_BUFFER:
            mem = &buff.datos[buff.offset];
            mem[0] = nuevo_valor;
            mem[buff.ventana] = nuevo_valor;
            break;
        default:
            mem = N_BUFFER;
    }

    return mem;
}
Int16 32_a_16 (Int32 x){
    if (x < MIN_16BITS)
        return MIN_16BITS;
    if (x > MAX_16BITS)
        return MAX_16BITS;
    return x;
}
```

Fig 27:gestion_memoria.c

5.4 Filtro FIR

Archivo que realizará la función del filtro FIR, se precisará los parámetros que se van a utilizar

Funciones:

- **Iniciamos_FIR:** gestiona las variables y las posiciones necesarias para los buffers.
- **Iniciamos_FIR_parametros:** gestiona los parámetros del filtro y su estructura.
- **Fin_FIR:** vacía la memoria del filtro.
- **FIR:** realiza el filtro FIR.
- **Get_lowpass:** contiene tanto los coeficientes de un low pass filter de segundo orden a diferentes frecuencias de corte. Coeficientes que generamos con la herramienta ya nombrada *fdatool* de Matlab.

```
#include "gestion_memoria.h"


//-----------------------VARIABLES---------------------------//

#define PI 3.14159265358979323846
#define coef_filtro 33

//----------------------ESTRUCTURAS-------------------------//

//Parámetros de un filtro FIR
typedef struct
{
Uint16 coeficientes;    //Número de coeficientes
Uint8 f;    //f del formato Qm.f de los coeficientes
}parametros_fir;
```

Fig 28: filtro_FIR.h

```
#include "filtro_FIR.h"

FIR_parametros iniciamos_FIR (float *h, Uint16 coeficiente, Uint8 F,
                    Int16 *coefs_fijo[], buffer *buff){
    FIR_params params = init_FIR_params(coeficientes,F);
    *coefs_fijo = calloc(coeficientes,sizeof(Int16));
    fijo (h,coeficientes,F,*coefs_fijo);
    *buff = nuevo_buffer(coeficientes,DOUBLE_BUFFER);
    return parametros;
}
 FIR_parametros iniciamos_FIR_parametros (Uint16 coeicientes, Uint8 F){
    FIR_parametros parametros;
    parametros.coeicientes  = coeficientes;
    if (F < 32)
        parametros.f = F;
    else
        parametros.f = 32;
    return parametros;
}

Int16 filtro_FIR (Int16 *h, Int16 *mem, FIR_parametros parametros){
    Int32 acumulador = 0;
    Int32 round = (Int32)1<<(parametros.f-1);
    Uint16 j;
    for (j=0;j<parametros.taps;j++)
        acumulador += (Int32)h[i]*mem[i];
    acumulador = (acumulador + round)>>parametros.f;
    return 32_a_16(acumulador);}
```

Fig 29: filtro_FIR.c

5.5 Filtro adaptativo

Archivo que realizará la función del filtro adaptativo, tendremos variables de tipo FIR en el que tendremos los parámetros del filtro adaptativo y el signo que se emplea en el algoritmo LMS.

Realizamos un desplazamiento_2beta que aplicaremos en el algoritmo adaptativo en lugar de la ecuación explicada en el capítulo del filtro adaptativo.

$$h[k]_{n+1} = h[k] + 2\,\beta\,e[n]\,x[n-k]$$

la sustituimos por:

$$desplazamiento\_2beta = redondear(-\log_2(2\,\mu)) + 30 - desplazamiento$$

Desplazamiento_2beta: número de desplazamientos que equivalen a la multiplicación de 2 β.

Funciones:

- **Iniciamos_filtro_adaptativo:** gestiona las variables y las posiciones necesarias para los buffers.
- **Iniciamos_parametros_filtro_adaptativo:** gestiona los parámetros del filtro y su estructura.

- **Fin_fadapt:** vacía la memoria del filtro.
- **Filtro_adaptativo:** realiza el filtro adaptativo.
- **LMS:** actualiza los coeficientes del filtro. Algoritmo LMS con signo en función de la estructura de los parámetros.
- **LMS_Positivo:** actualiza los coeficientes del filtro. Algoritmo LMS con signo positivo.
- **LMS_Negativo:** actualiza los coeficientes del filtro. Algoritmo LMS con signo negativo.

```
#include "filtro_FIR.h"

///----------------------------VARIABLES---------------------------------//
#define LMS_Postivo         0x00 //Algoritmo LMS con signo positivo
#define LMS_Negativo        0x01 //Algoritmo LMS con negativo
#define LMS_Mask            0x01 //Máscara del bit de signo
//----------------------------ESTRUCTURAS---------------------------------//
typedef struct
{
    parametros_fir paramsFIR;       //Parámetros del filtro FIR
    Uint8   desplazamiento_2beta;   //desplazamientos y multiplicamos por 2beta
    Uint8   signo_LMS;              //Signo del algoritmo LMS
    }parametros_filtro_adaptativo;
```

Fig 30: filtro_adaptativo.h

```
#include "filtro_adaptativo.h"

adaptativo_filtro_parametros iniciamos_parametros_adaptativo (Uint16 coeficientes,
                          Uint8 F, float beta, Uint8 signo_LMS) {
    adaptativo_filtro_parametros parametros;
    parametros.paramsFIR = iniciamos_FIR_parametros(coeficientes,F);
    parametros.desplazamiento_2beta = round(-log2(2*beta)) + 30 - parametros.paramsFIR.f;
    if (parametros.desplazamiento_2beta>32)
        parametros.desplazamiento_2beta = 32;
    params.signo_LMS = signo_LMS & LMS_Mask;
    return parametros;
}

Int16 filtro_adaptativo (Int16 *h, Int16 *mem, Int16 d,
                         parametros_filtro_adaptativo parametros, Int16 *y){
        Int16 e;
        *y = filtro_FIR (h, mem, parametros.paramsFIR);
        e = d - *y;
        LMS (h, mem, e, parametros);
        return e;
}

void LMS_Positivo (Int16 *h, Int16 *mem, Int16 e, parametros_filtro_adaptativo parametros){
    Uint16 j;
    Int32 valor;
    Int32 round = (Int32)1<<(parametros.desplazamiento_2beta-1);
    for (j=0;j<params.paramsFIR.coeficientes;j++){
    value = h[i] + (((Int32) e*mem[i] + round)>>parametros.desplazamiento_2beta);
    h[i] = 32_a_16(valor);
    }
}
```

Fig 31: filtro_adaptativo.c

5.6 Filtrado-x LMS

Finalmente para el control activo de ruido implementamos la parte del filtrado-x.

Se describen las siguientes funciones:

- **Iniciamos_filtro_x:** gestiona las variables y las posiciones necesarias para los buffers.
- **Iniciamos_parametros_fx:** gestiona los parámetros del filtro y su estructura.
- **Iniciamos_filtros_fx:** genera la estructura de los filtros y reserva memoria que se utilizará para guardar los coeficientes de los filtros.
- **Iniciamos_buffer_fx:** gestiona los buffers del filtro y su estructura.
- **Fin_fx:** vacía la memoria del filtro.
- **Filtrado_x:** implementa el algoritmo para el filtrado-x, nos devolverá la señal y[n] a partir de las señales de entrada x[n] y e[n].
- **Actualiza_fx:** actualización de los coeficientes de los filtros.

```
#include "filtro_adaptativo.h"

//--------------------------------ESTRUCTURAS---------------------------------------//

//-----------------------------Paramentros para los filtros en fxLMS----------------------///
    typedef struct
    {
        fadapt_params C;    //Filtro  C
        fadapt_params H;    //Filtro  H
        fadapt_params D;    //Filtro D
        Uint16 path_config; //Estructura palabra filtro-x
    }parametros_filtros_x;

//-----------------------------Coeicientes filtros en fxLMS----------------------///
    typedef struct
    {
        Int16 *C;
        Int16 *H;
        Int16 *D;
    };filtros_x

//-----------------------------Buffers filtros en fxLMS----------------------///
    typedef struct
    {
        buffer x;
        buffer xf; //Buffer a la salida de del sistema H(z)
        buffer y;
    }buffers_filtros_x;

//-----------------------------Señales filtros en fxLMS----------------------///
    typedef struct
    {
        Int16 xf;
        Int16 y;
        Int16 y_H;
        Int16 y_D;
        Int16 e_C;
        Int16 e_H;
        Int16 e_D;
    }señales_filto_x;
```

Fig 32: f_x.h

```
#include "filtro_x.h"

filtros_x iniciamos_filtro_x (parametros_filtros_x parametros){
    float *h;
    Uint16 coeficientes_maximos;
    filtros_x filtros;
    coeficientes_maximos = max(parametros.C.paramsFIR.coeficientes,
                        parametros.H.paramsFIR.coeficientes,
                        parametros.D.paramsFIR.coeficientes);
    h = calloc(coeficientes_maximos,sizeof(double));
    h[0] = 1;

    filters.C = calloc(parametros.C.paramsFIR.coeficientes,sizeof(Int16));
    filters.H = calloc(parametros.H.paramsFIR.coeficientes,sizeof(Int16));
    filters.D = calloc(parametros.D.paramsFIR.coeficientes,sizeof(Int16));
    fijo(h,parametros.C.paramsFIR.coeficientes,parametros.C.paramsFIR.f,filtro_x.C);
    fijo(h,parametros.H.paramsFIR.coeficientes,parametros.H.paramsFIR.f,filros_x.H);
    fijo(h,parametros.D.paramsFIR.coeficientes,parametros.D.paramsFIR.f,filtros_x.D);
    free(h);

    return filtros;
}
fxlms_buffers iniciamos_buffer_fx (parametros_f_x parametros){
    buffers_filtros_x buffers;
    Uint16 coeficientes_maximos;
    coeficientes_maximos = max(parametros.C.paramsFIR.coeficientes,
                               parametros.H.paramsFIR.coeficientes,
                               parametros.D.paramsFIR.coeficientes);
    buffers.x  = nuevo_buffer(coeficientes_maximos, DOBLE_BUFFER);
    buffers.xf = nuevo_buffer(parametros.C.paramsFIR.coeficientes, DOBLE_BUFFER);
    buffers.y  =  nuevo_buffer(parametros.H.paramsFIR.coeficientes, DOBLE_BUFFER);

    return buffers;
}

Int16 filtrado_x(Int16 x, filtros_x filtros, buffers_filtros_x *buffers,
                                            parametros_filtros_x parametros){
    Int16 *mem_x;
    Int16 y;
    mem_x  = Introducir_dato(x,&buffers->x);
    y      = filtro_FIR (filtros.C, mem_x, parametros.C.paramsFIR);
    Introducir_dato(y,&buffers->y);

    return y;
}
```

Fig 33 f_x.c

Capítulo 6. Simulaciones realizadas

Para comprobar el funcionamiento de nuestro programa disponemos de los siguientes elementos:

- Nuestra tarjeta usbstk5505 detallada previamente.
- El entorno de desarrollo integrado (IDE) de Texas Instruments, Code Composer Studio (CCS v4.1.3)
- Tarjeta de sonido Fast Track Pro.
- El cableado y adaptadores nombrados en la parte de Hardware utilizado
- Altavoces y micrófono.

Fig 34: Fast Track Pro M-audio

- ARTA programa para medida de respuesta a impulsos y análisis en tiempo real de espectros y respuesta en frecuencia.

Fig 35: ARTA.

6.1 Simulación Bypass

Lo primero que tenemos que simular para comprobar el funcionamiento de nuestro programa es un bypass, introducir mediante Arta una señal y analizar las salidas de la tarjeta. Así comprobaremos si hemos activado bien el aic3204 y los datos se transmiten correctamente por nuestra tarjeta.

La estructura que seguimos para la simulación de nuestro bypass es la siguiente:

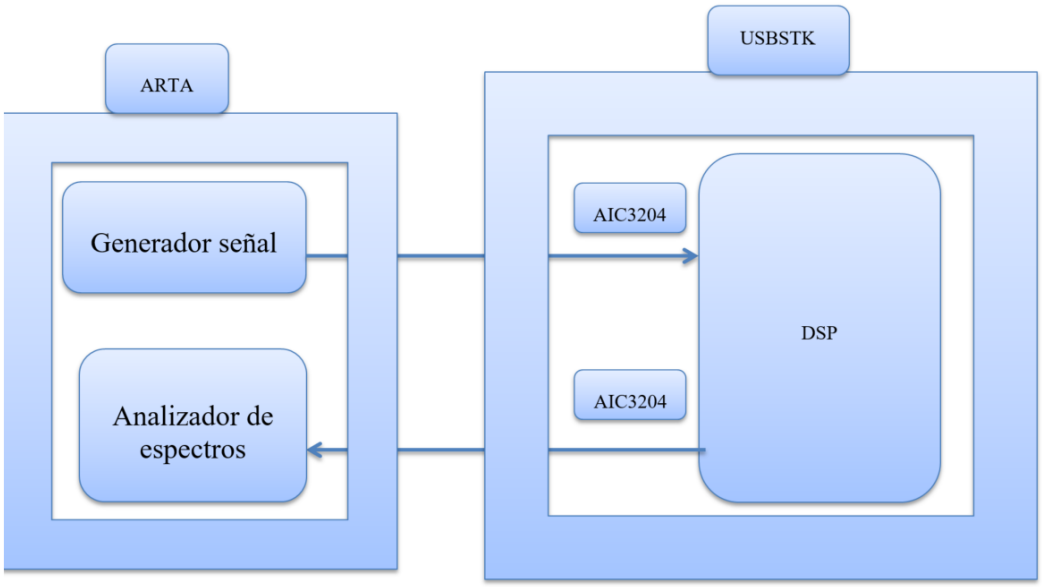


Fig 36: Estructura bypass.

Tanto el generador de señal como el analizador de espectros los utilizamos gracias a nuestras estación digital de trabajo ARTA. Utilizamos el códec *aic3204* que está presente en nuestra tarjeta para transportar la información de la señal.

Primero vamos a generar un tono a 1000 Hz, para ellos configuramos nuestro programa a una frecuencia de muestreo de 44100 Hz. Primero configuramos nuestro código para que se lea el canal izquierdo de la señal que introducimos en nuestra tarjeta y que se reproduzca la misma por el canal izquierdo de salida.

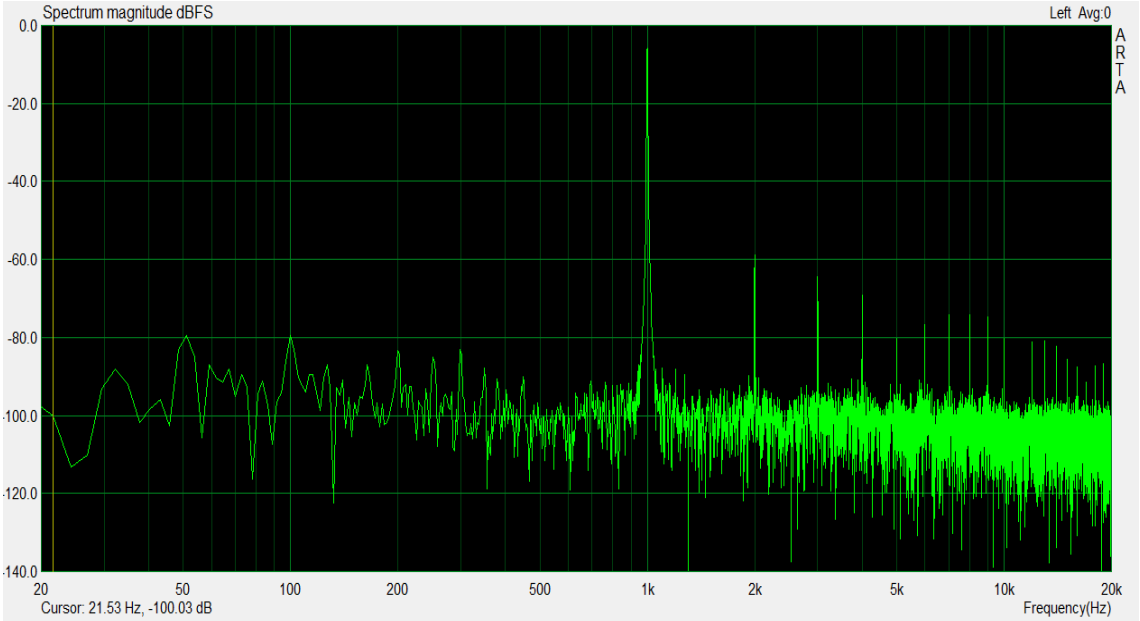


Fig 37: Lectura tono 1k canal derecho.

Luego cambiamos el código para que se escriba la salida en el canal izquierdo de la tarjea

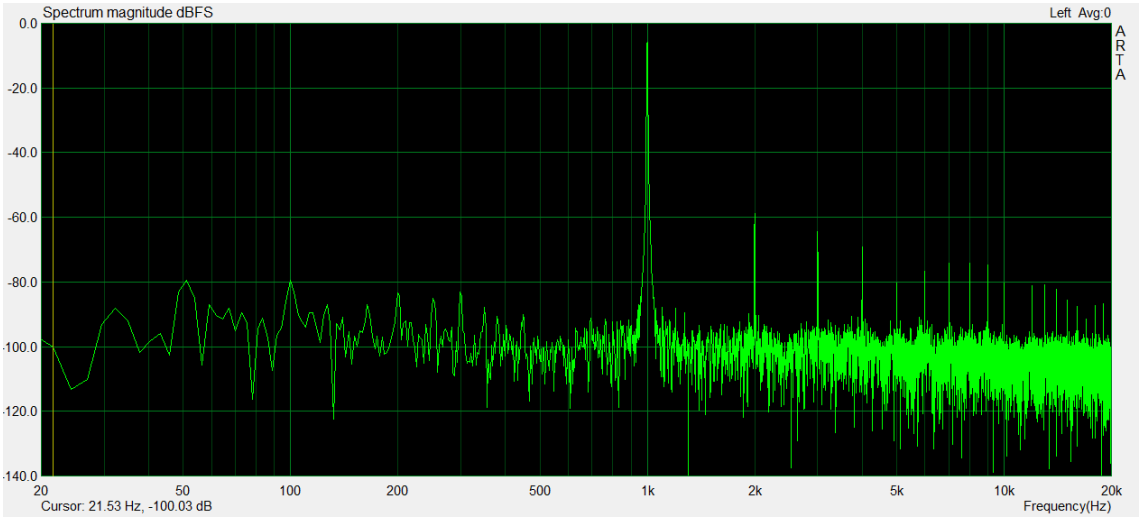


Fig 38: Lectura tono 1k canal izquierdo

En el analizador de espectros vemos que se representa un tono a la frecuencia de 1kHz que es la frecuencia a la que estamos generando el tono desde ARTA. Con ello comprobamos que estamos activando y utilizando correctamente el *aic3204*.

6.2 Simulación para los generadores de señal

Dentro de nuestra DSP generamos la señal, la configuramos para que genere dos tipos de señal. Una de ella será un tono que fijamos a **150 Hz** y la otra señal será un ruido aleatorio a **-40dBFs**. La señal tonal la sacaremos por el canal izquierdo, mientras que el ruido lo haremos por el derecho. El aic3204 transporta datos de 32bits, para leer o escribir en un canal o el otro de la tarjeta tendremos que hacerlos en los 16 primeros bits para un canal y los siguiente para el otro.

La estructura que se sigue para esta simulación de generador señal es la siguiente:

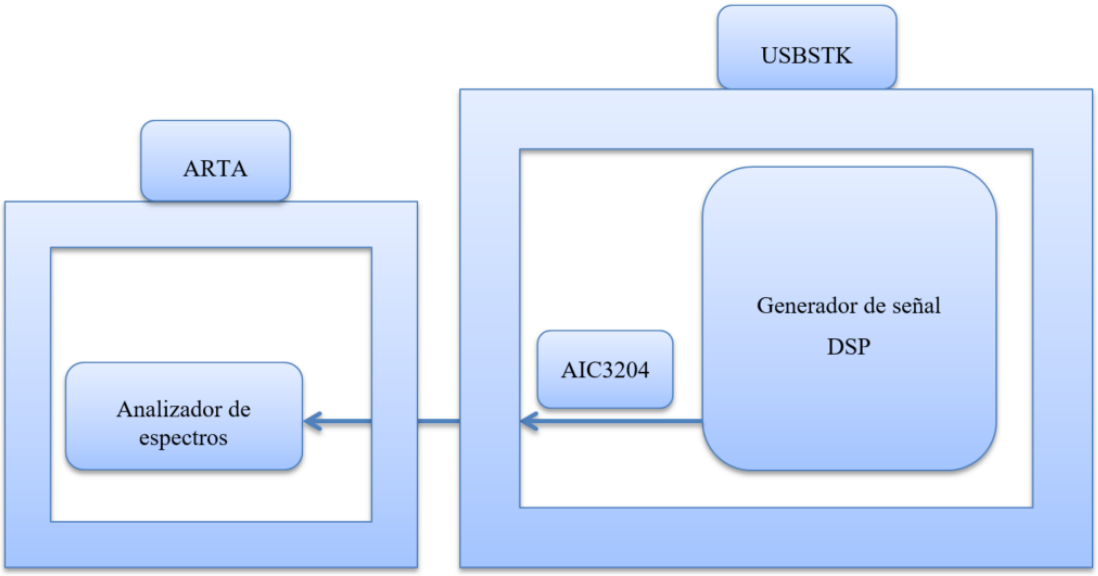


Fig 39: Estructura generador de señal

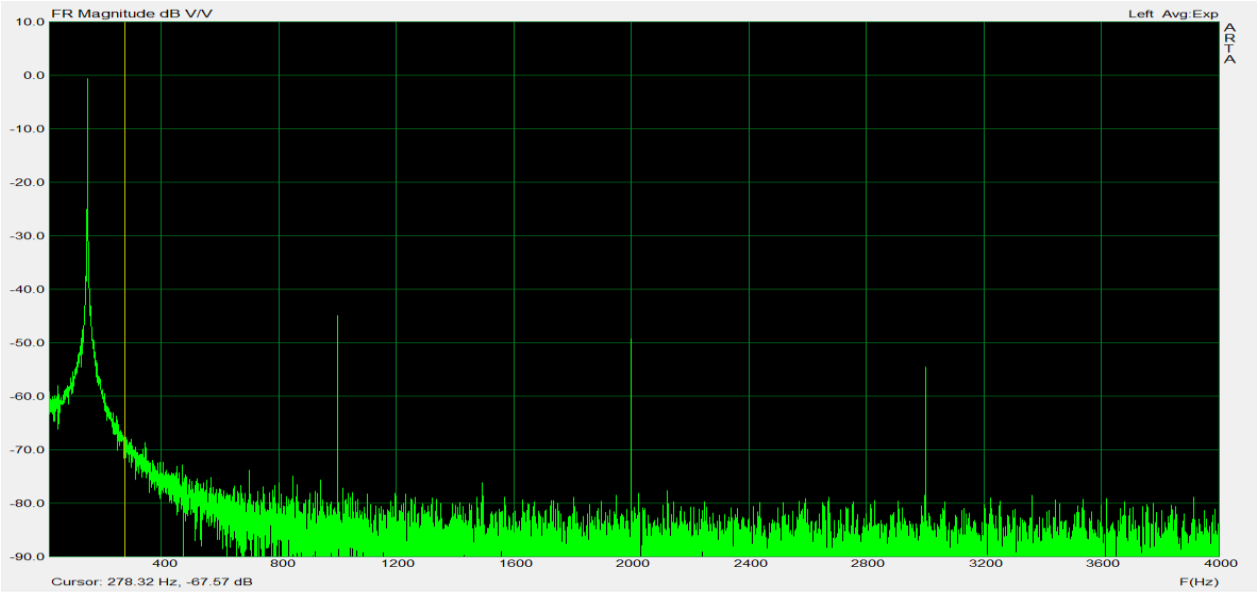


Fig 40: Generador de un tono 150Hz

Aquí tenemos la respuesta en frecuencia del tono que generamos con nuestra tarjeta a **150 Hz** y con eso verificamos su funcionamiento.

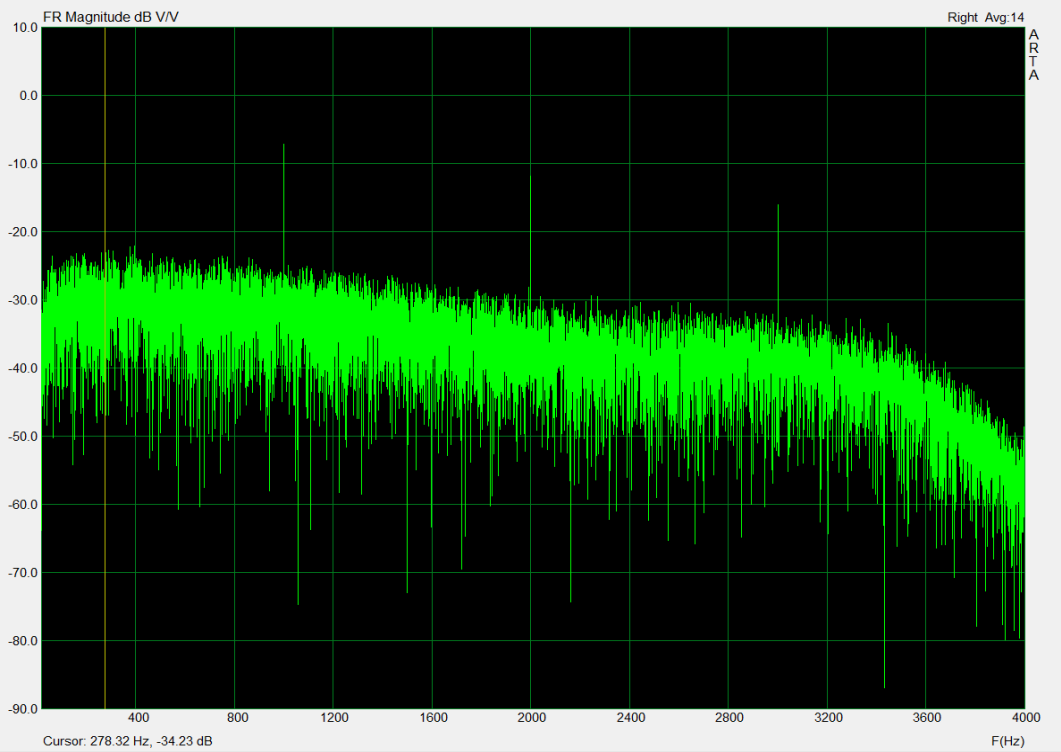


Fig 41: Generador de ruido -40 dBFs

Esta imagen es la respuesta en frecuencia de ruido aleatorio que introducimos de -**40dBFs**.

Vemos que las gráficas con respecto a la simulación del bypass han cambiado, antes las dimensiones de las gráficas eran de 22kHz en el eje de la frecuencia y ahora hasta 4kHz. Es así porque se ha cambiado la frecuencia de muestreo entre una simulación y otra para comprobar el funcionamiento a distintas frecuencias de muestreo.

6.3 Simulación FIR

Hemos generado un código ***fir_lowpass_filter.c*** que contiene los coeficientes que utilizaremos en los filtros FIR, generados con MATLAB utilizando la herramienta *fdatool* hemos generado un total de 33 coeficientes que variaran según la frecuencia de corte que queramos.

Frecuencias de corte***: 0, pi, pi/2, pi/3, pi/4, pi/5, pi/6, pi/7, pi/8, pi/9, pi/10, pi/11, pi/12, pi/13, pi/14, pi/15, pi/16, pi/17, pi/18, pi/19 y pi/20.***

Desde ARTA vamos a generar un ruido blanco que será filtrado con un filtro paso bajo, utilizaremos distintas frecuencias de corte para ver cómo responde nuestra tarjeta. Como estamos empleando una frecuencia de muestreo de 8000 Hz si queremos que la frecuencia de corte sea de 500 Hz se utilizarán los coeficientes que generamos con la frecuencia de muestreo de ***pi/8***.

Esta función nos devuelve la posición a la que queremos coger los coeficientes que corresponderían a la frecuencia de corte deseada:

```
float* get_lowpass(Uint16 fc, Uint16 fs){
    Uint8 value = round(1.0*fs/(2*fc));
    if (value >20)
        value = 20;
    return &h_lowpass_coef[value][0];
}
```

Fig 42: Función que nos devuelve la posición de los coeficientes para el filtro

La estructura que seguimos para la comprobación de nuestro filtro es la siguiente:

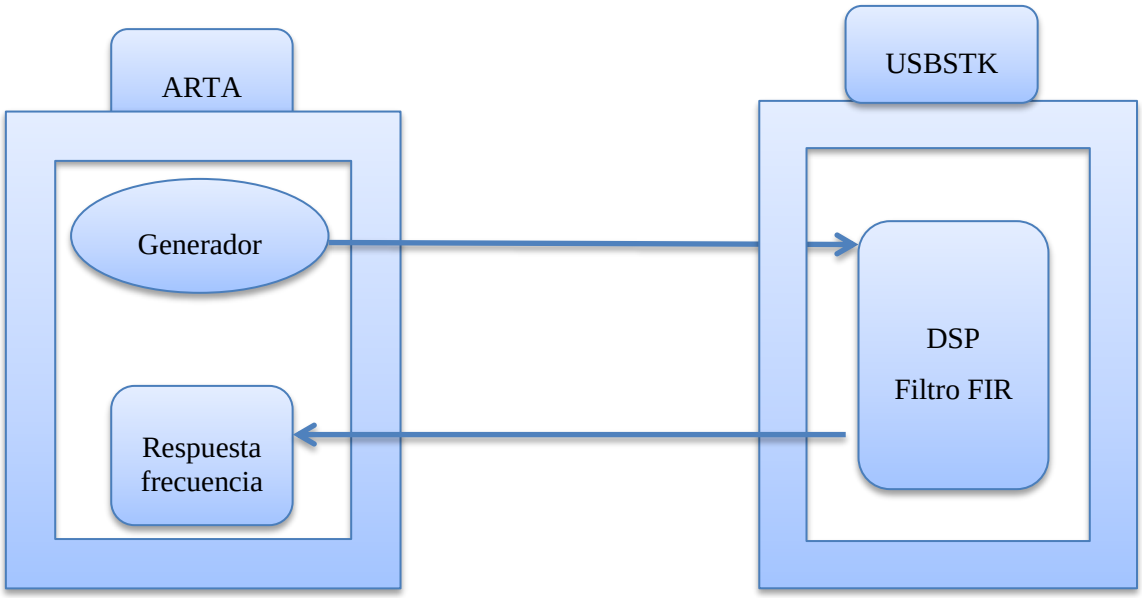


Fig 43: Estructura filtro FIR

Tras las simulaciones podemos decir que nuestro filtro FIR funciona correctamente para las distintas pruebas que hemos realizado: ***fc=500 Hz fc = 1000 Hz fc = 2000 Hz***

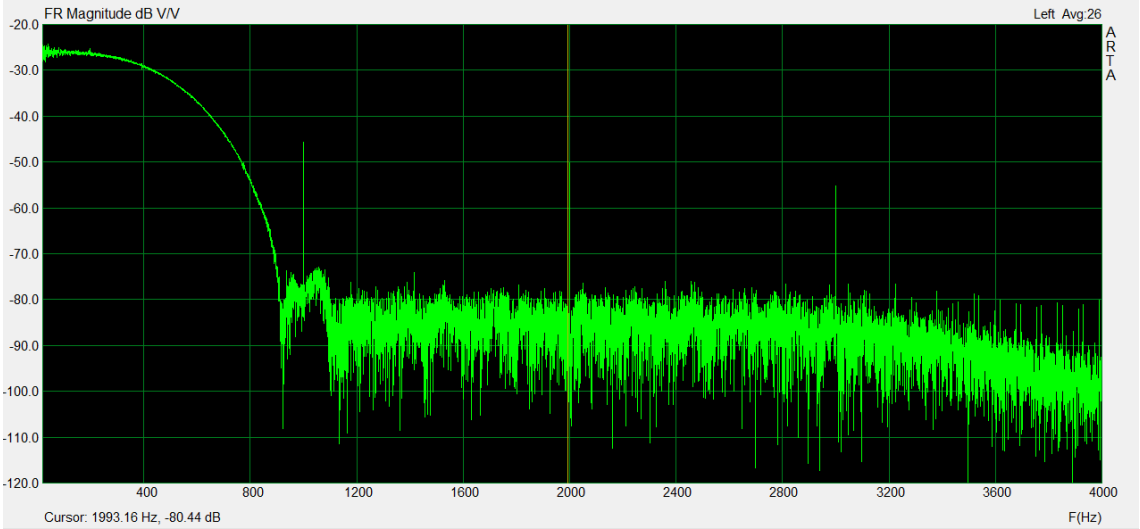


Fig 43: Salida y[n] tras el filtrado FIR con fc:500Hz

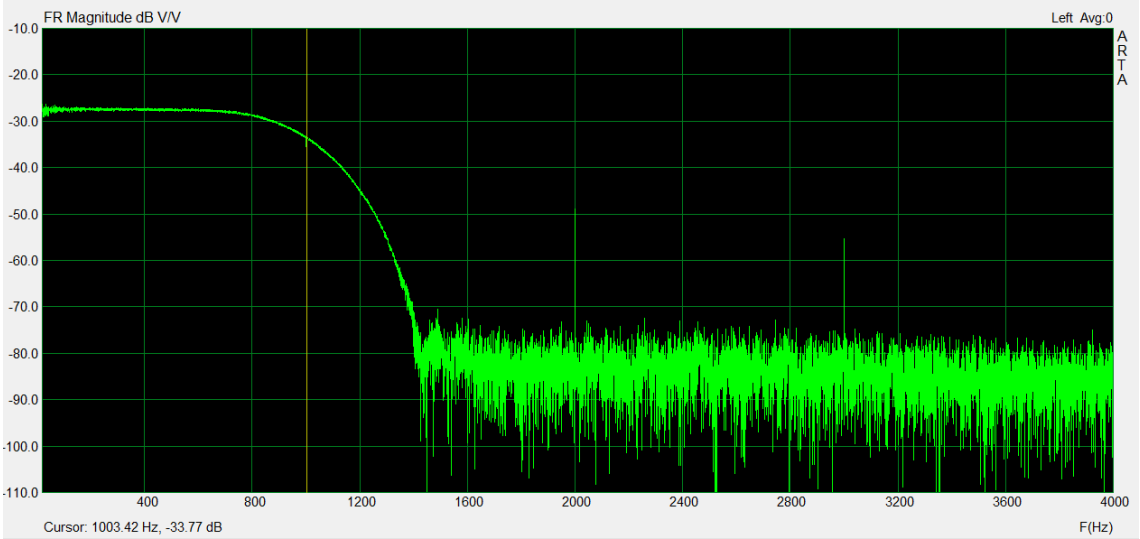


Fig 44: Salida y[n] tras el filtrado FIR con fc:1000Hz

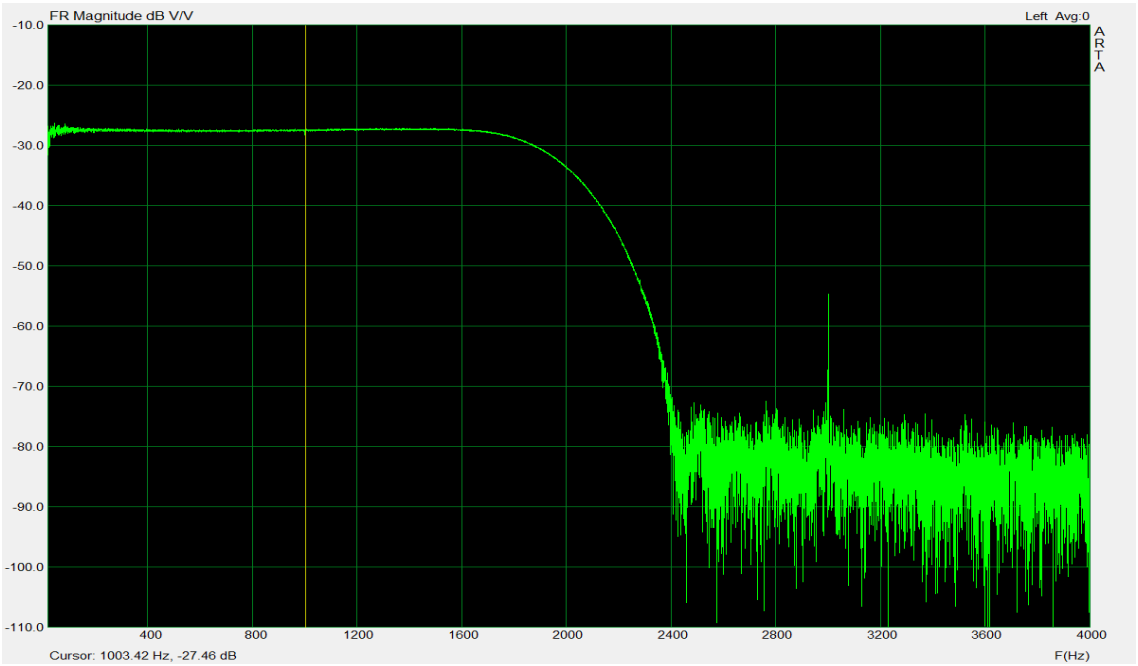


Fig 45: Salida y[n] tras el filtrado FIR con fc:2000Hz

Las pruebas del filtro FIR paso bajo han sido muy positivas para nuestro nodo acústico. Podemos variar las frecuencias de corte y los resultados son buenos, observamos como en todas ellas la señal cae unos 50 dB una vez llega a la frecuencia de corte que le hemos introducido. En todos ellos la señal empieza a caer instantes antes de llegar a la frecuencia de corte que hemos configurado, en el instante en la que la frecuencia llega a la frecuencia de corte que le hemos puesto la señal ha decaído unos 6dB.

Por lo tanto la parte del diseño de un filtro FIR en nuestro nodo acústico podemos concluirla por su buen funcionamiento.

6.4 Simulación Filtro adaptativo

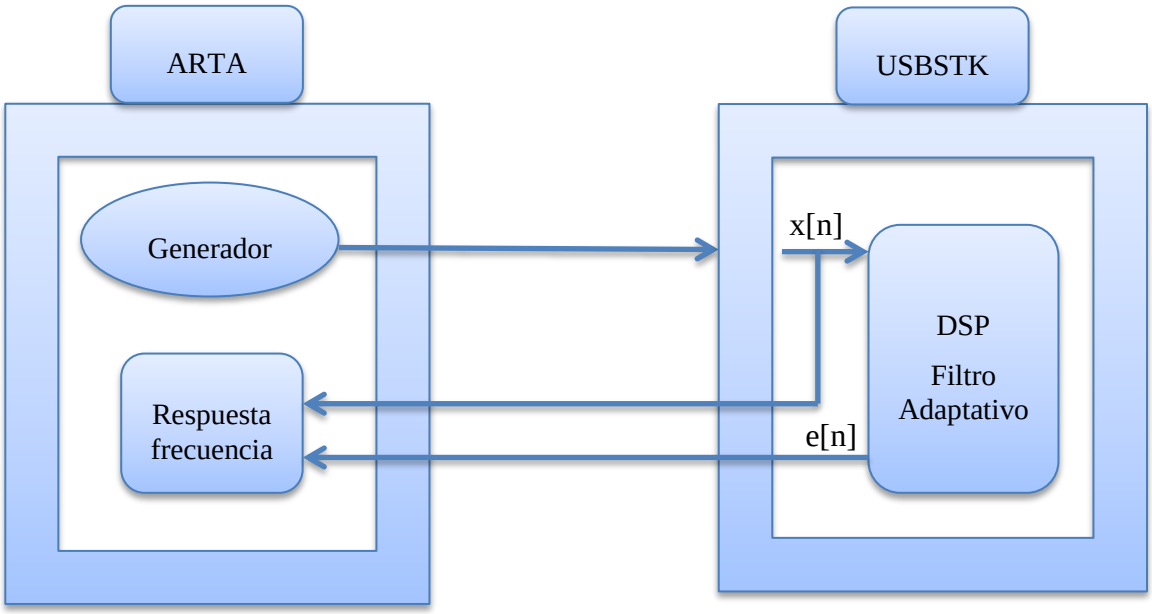
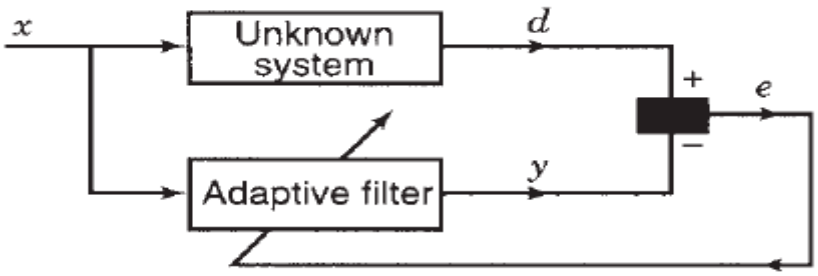


Fig 46: Estructura filtro adaptativo

En nuestra estación de trabajo vamos a leer dos señales diferentes, una de ellas será la que generamos directamente desde ARTA y la otra será la señal de error resultante. La estructura que seguimos es para la identificación del sistema, las limitaciones que tenemos en ARTA para generar diferentes señales para cada uno de las salidas de la tarjeta de sonido nos complica a la hora de demostrar el funcionamiento de nuestro filtro adaptativo.

Otra forma de demostrar el funcionamiento del filtro es generar una señal que será x[n] y misma señal pero retrasada d[n]. Si por ejemplo el retraso de la señal d[n] con respecto de la original fuese de 5ms tras esperar un rato, en los coeficientes del filtro adaptativo, deberíamos observar en la posición 40 un valor cercano a 32768 lo que en el formato que estamos siguiendo Q0.15 (dividiéndolo entre 2^{15}) sería el valor 1. El retraso de 5ms representa un retraso en la señal de 40 muestras y por lo tanto demostraríamos su funcionamiento.



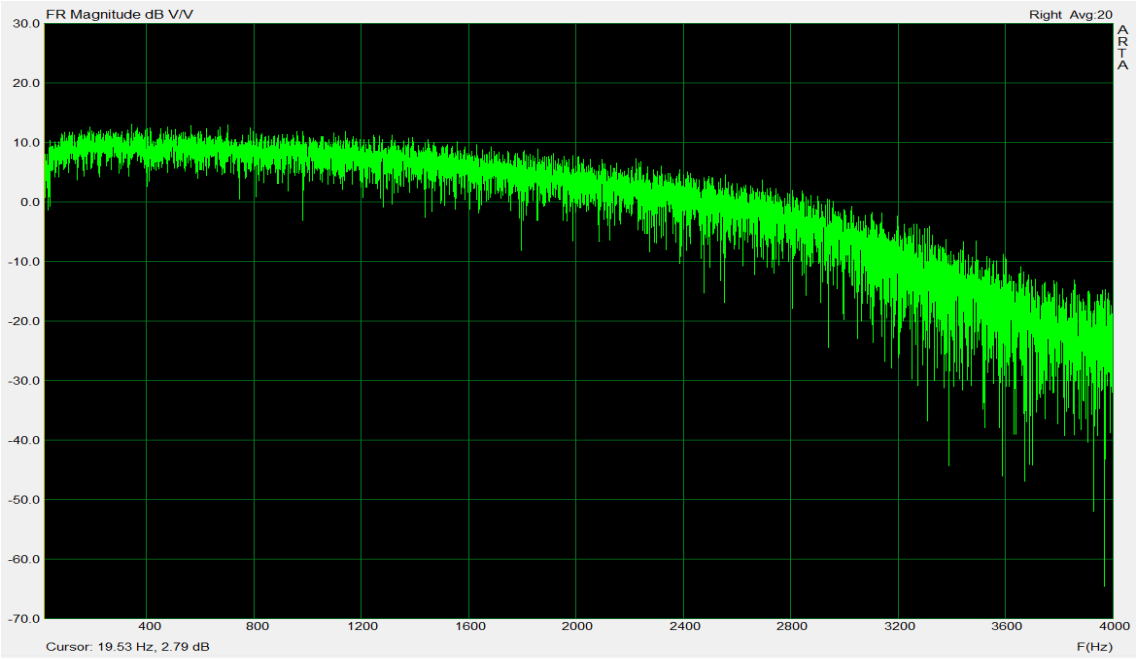


Fig 47: Señal original x[n] filtro adaptativo

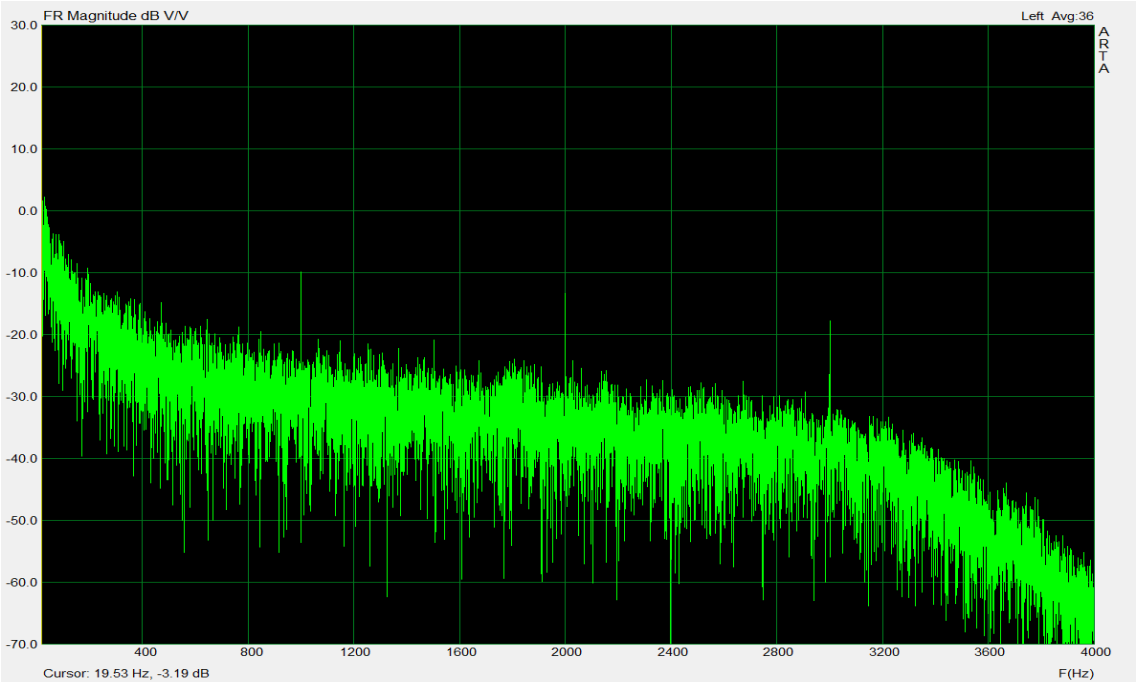


Fig 48: Señal error e[n] filtro adaptativo

La primera gráfica, *Fig 47*, muestra la señal que introducimos en el filtro adaptativo x[n] que es la señal que generemos desde ARTA a -10dB. Mientras que la segunda, *Fig48*, es el resultado de la señal de error tras unos instantes. Tras eso instantes vemos que el valor de la señal de error e[n] está en -40 dB mientras que la señal original estaba en 10 dB. Ya que el valor de e[n] se da con la ecuación $\boldsymbol{e(n) = d(n) - y(n)}$, significa que hemos conseguido que la señal y[n] sea similar a la señal d[n] y que el filtro adaptativo funciona correctamente.

Como desde ARTA hemos generado la misma señal para x[n] y d[n]. El canal acústico será el sistema a identificar, en este caso un cable. No se tarda mucho en reducir la señal de error ya que amabas señales son iguales.

Como hemos introducido la misma señal para la entrada del filtro y la señal deseada, cuando miremos los coeficientes del filtro adaptativo, que se almacenan dentro de una posición de memoria, el valor que hemos comentado antes "32768" aparece enseguida ya que no existe prácticamente retraso. Code Composer nos ofrece la posibilidad de pausar el programa y ver cómo está la posición de memoria que queramos en ese instante.

0x0018E0	6164	-5969	67	11744
0x0018E4	32767	0	0	0
0x0018E8	0	0	0	0
0x0018EC	0	0	0	0
0x0018F0	0	0	0	0
0x0018F4	0	0	0	0
0x0018F8	0	0	0	0
0x0018FC	0	0	0	0
0x001900	0	0	0	0
0x001904	0	0	0	0
0x001908	0	0	0	0

Fig 49: Coeficientes filtro adaptativo

6.5 Simulación de filtrado-x LMS.

Para comprobar el correcto funcionamiento de nuestro filtrado-x basado en el algoritmo LMS realizamos dos tipos de simulación.

6.5.1 Simulando entorno acústico

En la primera de ellas simularemos en camino acústico, en la salida de nuestra tarjeta tendremos por un canal la señal original y por la otra la señal de error resultante de juntar x[n] y y[n] (salida del filtro). Seguiremos la estructura que tenemos a continuación:

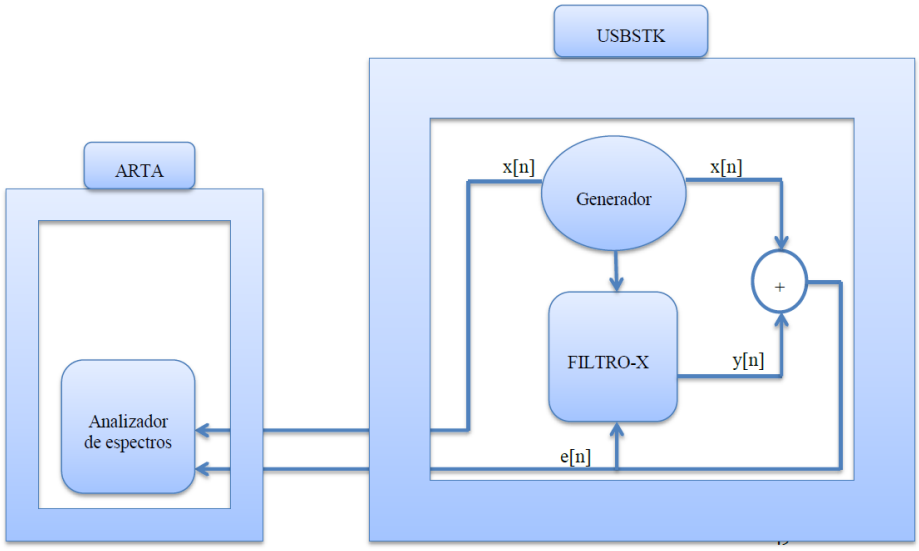


Fig 50: Estructura simulación filtrado-x simulando el entorno acústico.

Introducimos una señal de 30 Hz, ya que es a la frecuencia a la que mejor trabaja el filtrado. Se han realizado diferentes pruebas a diferentes frecuencias y es la que mejor respuesta ha dado.

Como ya hemos comentado antes podemos trabajar de dos formas diferentes siguiendo el esquema de la *Fig 16.* Dentro de esta estructura podemos simular de dos formas, en la primera de ellas previamente se hace una identificación del sistema.

- Gráficas simulación con previa identificación del sistema

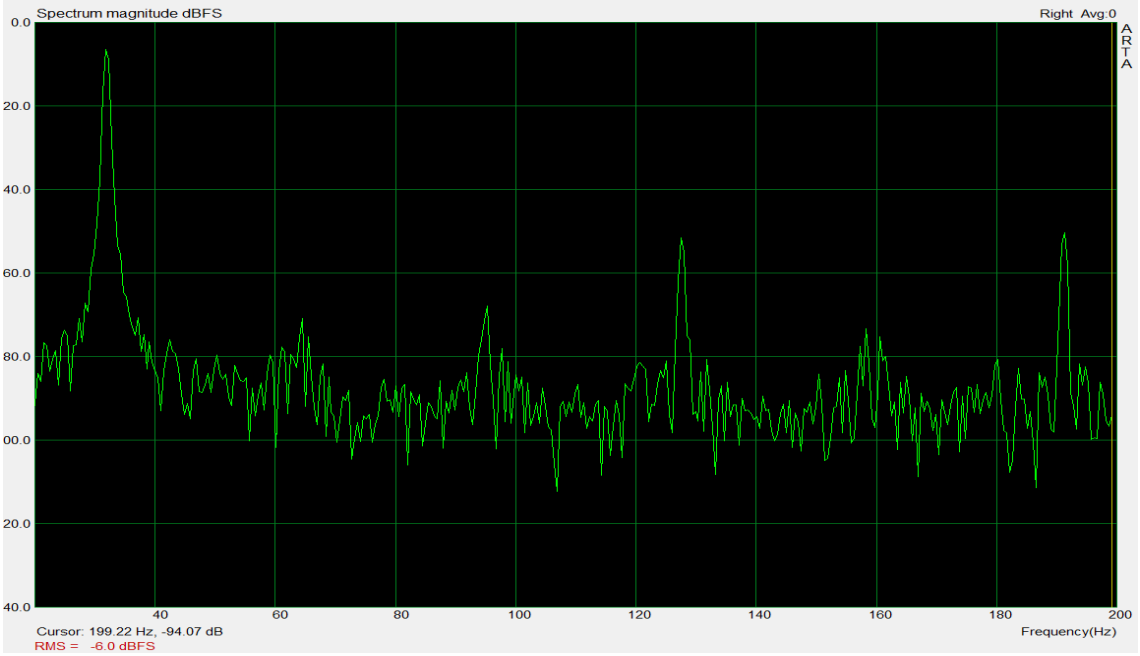


Fig 51: Señal original x[n] que introducimos en el filtro-x con previa identificación del sistema

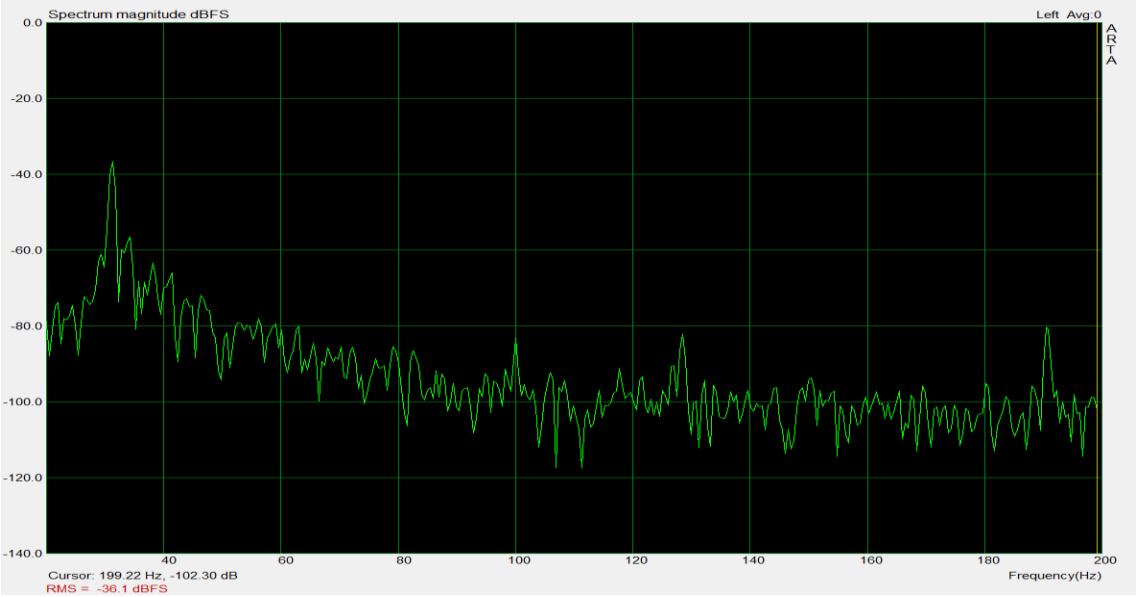


Fig 52: Señal error e[n] tras el filtrado-x con previa identificación del sistema

La previa identificación del sistema genera los coeficientes que utilizaremos en H(z) directamente, vemos que la señal de error e[n] representada en la *Fig52* se ha visto reducida en unos 30 dB con respecto a la señal original que introducimos en el sistema, representada en la *Fig51*, en el punto de frecuencia en la que está fijada la señal tonal por lo que en este caso la reducción de ruido se ha producido de forma satisfactoria.

En la segunda forma de esta parte de estas simulaciones la simulación la haremos directamente, sin identificación previa del sistema.

- Gráficas simulación sin previa identificación del sistema.

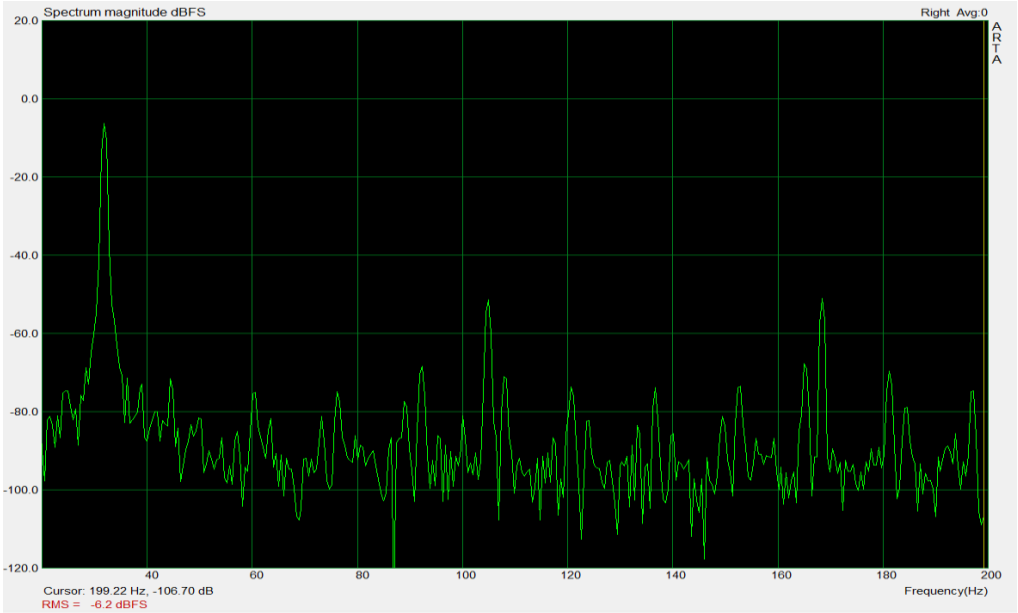


Fig 53: Señal original x[n] que introducimos en el filtro-x sin previa identificación del sistema

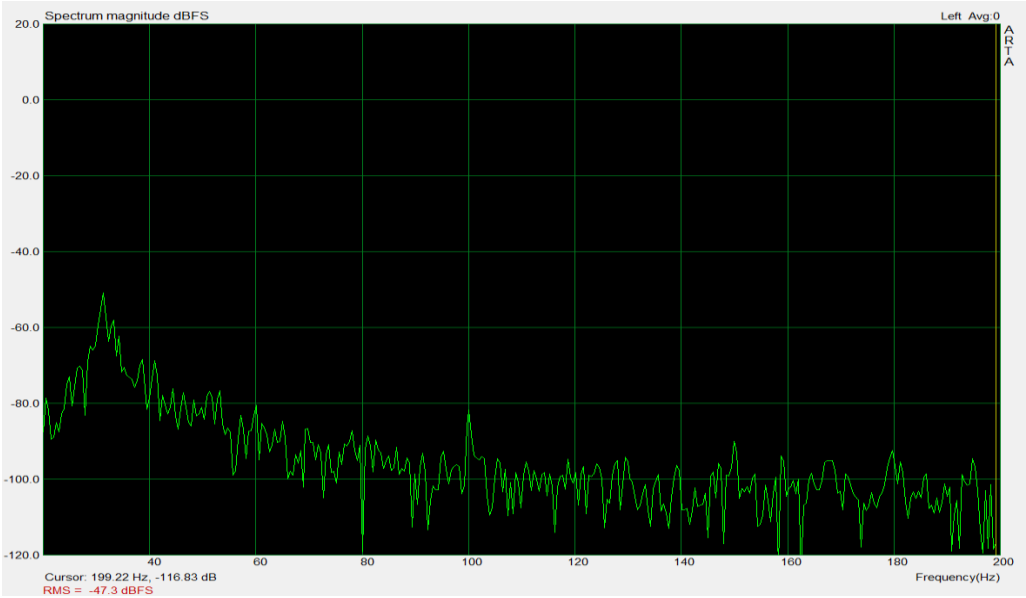


Fig 54: Señal error e[n] tras el filtrado-x sin previa identificación del sistema

En este caso vemos que reducción que se ha producido con respecto de la original es de 40 dB, así que la reducción de ruido es mayor que identificando previamente el sistema.

6.5.2 Simulación real

En la simulación de antes, la suma de la señal original x[n] y de la señal a la salida del filtro y[n], que dan como resultado la señal de error e[n], se produce dentro de nuestra DSP, así que a la salida de la tarjeta teníamos directamente la señal de error. Ahora vamos a utilizar los altavoces y el micrófono, en uno de los altavoces sacaremos la señal original y en el otro el ruido cancelador y[n]. Con el micrófono captaremos la señal de error que volveremos a introducir en la tarjeta para que actúe en el filtro-x.

La estructura que se sigue en esta parte de la simulación es la siguiente:

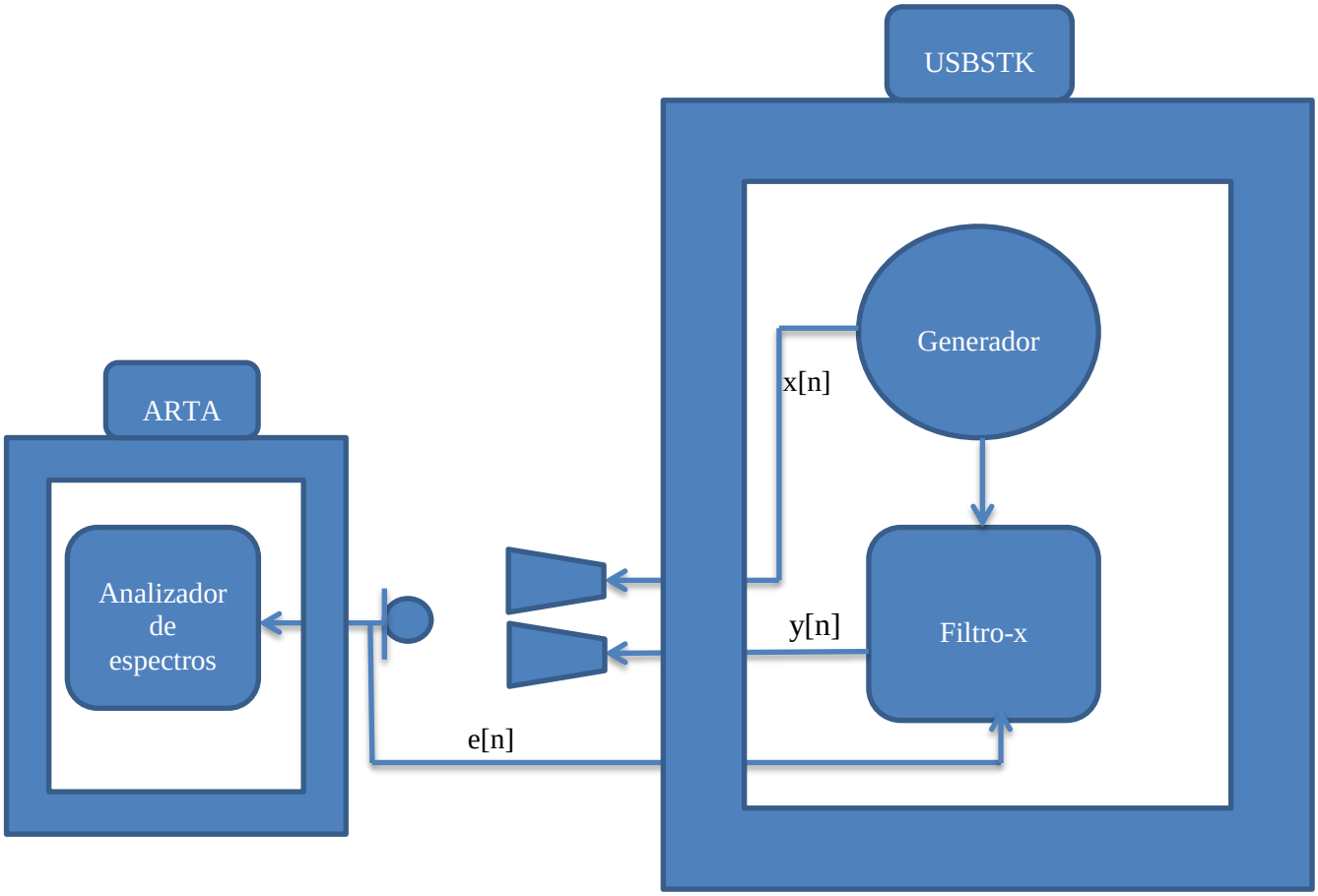


Fig 55: Estructura simulación real Filtrado_X

La separación que habrá entre los altavoces y el micrófono es de unos 30 cm. La reducción que se produce es en todo el espacio tridimensional ya que los altavoces que se utilizan tienen un patrón de radiación omnidireccional.

En este caso utilizaremos una frecuencia de 60 Hz y no hacemos ninguna identificación previa del sistema ya que la señal que generamos por el altavoz para hacer la identificación del sistema no es bien captada por el micrófono y no da un buen resultado.

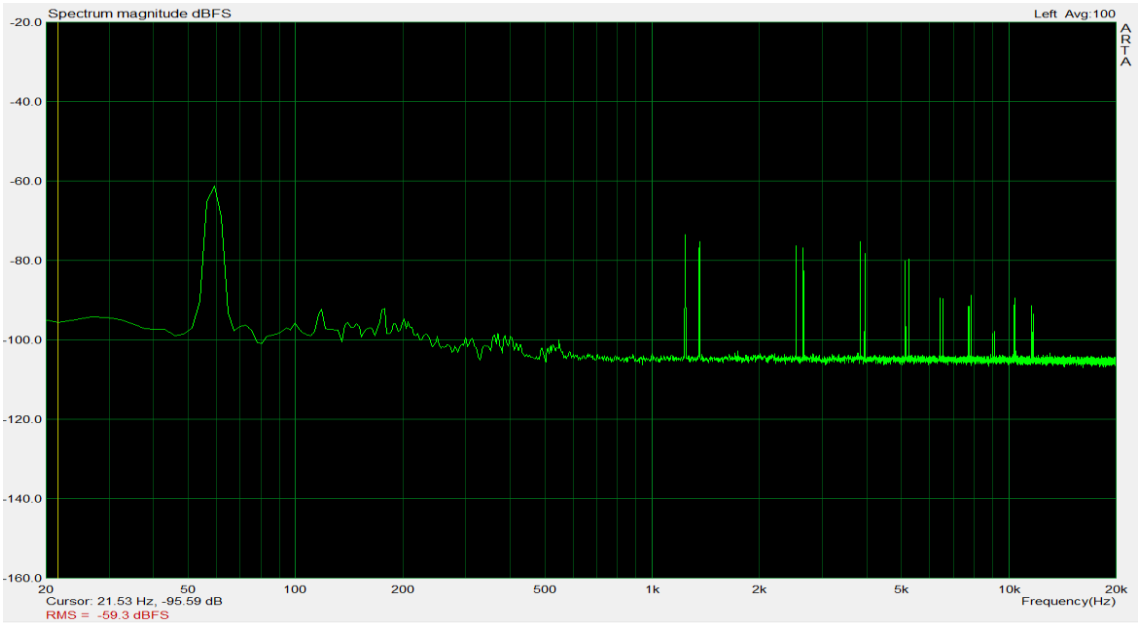


Fig 56: Señal original x[n] capatada por el micrófono, señal que introducimos en el filtrado-x

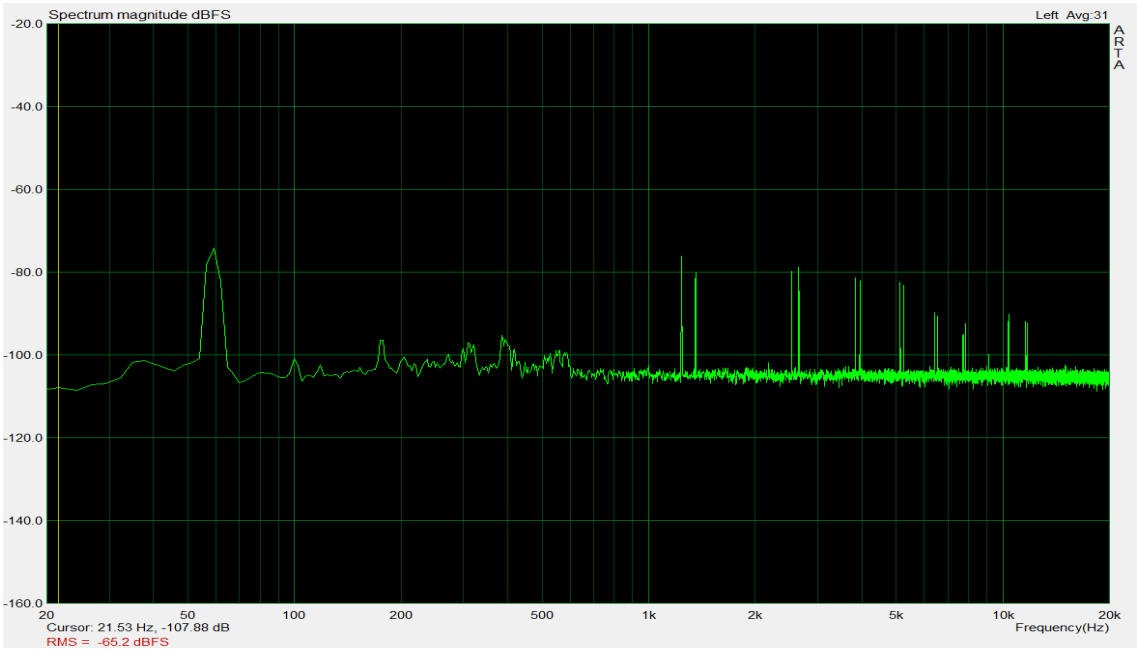


Fig 57: Señal error e[n] captada por el micrófono como resultado de juntar las señales que generan ambos altavoces.

En la *Fig 56* está representada la señal original x[n] que introducimos en el filtrado-x, observamos que los niveles de potencia de esta señal tonal son muy bajos, de *-60dB*. Como en las simulaciones del filtrado-x simulando el entorno acústico y en las simulaciones de generar la señal vemos que la señal generada era correcta, se deduce que la señal que se genera tras el proceso del filtrado-x sufre un cambio que hace que la señal pierda potencia. Observamos que la reducción de ruido que se produce es casi muy reducida, de unos 10 dB, esto es debido a que la señal que se está introduciendo al sistema ha perdido potencia por lo que la reducción de ruido es mucho menor. Aun así hemos visto una pequeña reducción del ruido por lo que la cancelación activa de ruido ha funcionado aunque no todo lo satisfactorio que debería.

Capítulo 7. Conclusiones.

El objetivo de este proyecto es el diseño de un nodo acústico hardware, tras su diseño podemos concluir que existe una amplia gama de posibilidades a desarrollar usando esta tecnología.

Usando un presupuesto muy reducido se pueden obtener grandes resultados, he tenido que adecuarme a ese presupuesto. Está claro que si utilizásemos mejores materiales los resultados serían mucho mejores.

En todas las gráficas se visualizan una serie de picos que he estado intentando eliminar durante todo el proyecto, con el paso del tiempo me resultaba más complicado ya que el material se va desgastando con el uso y empeora la calidad de las señales. El micrófono al no ser de muy buena calidad es difícil trabajar con él. Aunque no lo haya mostrado en las simulaciones, he utilizado las funciones de generar señal usando como salida los altavoces y captando el sonido de los altavoces con el micrófono, ya que esas señales son bastante limpias el micrófono las capta bien. Pero como vemos en las simulaciones reales del uso del filtrado-x la señal es muy reducida, es el coste que tiene las operaciones computacionales que realizamos durante todo el proceso.

He observado que con cosas muy simples la tarjeta funciona a la perfección pero a medida que le vamos introduciendo cálculos más complejos y que requieren mucha velocidad la tarjeta pierde eficiencia y no muestra resultados tan buenos.

En conclusión el diseño del nodo ha sido correcto, para continuar este proyecto hay una serie de pasos que se podrían realizar:

- Estudio del programa para intentar solucionar la perdida de potencia en la simulación real.
- Estudio de otras tarjetas que puedan cumplir con los cálculos con mayor rendimiento, quizás una tarjeta de la misma familia pero algún modelo superior sería suficiente.
- Incrementar la estabilidad del sistema, queremos trabajar con filtros de mucho mayor tamaño y poder trabajar con sus coeficientes.
- Realizar pruebas de reducción de ruido de carácter aleatorio.
- Posibilidad de comunicarse con otros nodos acústicos, estudiar la posibilidad de utilizar la forma inalámbrica.
- Implementación en una plataforma RASPBERRY PI.

CAPITULO 8. BIBLIOGRAFIA

1. Procesadores Digitales de Señal, Grado en Ingeniería y Servicios de Telecomunicación. E.T.S.I.T. Germán Ramos Peinado.
2. TMS320VC5505 eZDSP USB STICK Documentation: support.spectrumdigital: FAQ, Board schematics, quickstart guide, technical reference, Layout info, Demo, CCSv4…
3. Procesamiento digital de señales, FCEFyN, Universidad Nacional de Cordoba.
4. ARTA User Manual.
5. FAST TRACK PRO, user manual.
6. Design of Active Noise Control System With the TMS320 Family.
7. «Aislamiento acústico - Aislacustic Ingeniería Acústica». Aislacustic Ingeniería Acústica.
8. Texas Instruments, «TMS320C5505 eZdsp™ USB Stick»:
9. Texas Instruments, «TMS320C5505 Fixed-Point Digital Signal Processor»
10. Texas Instruments, «TMS320C55XX Assably language tool»
11. Texas Instruments, «TMS320C55XX DSP Library»
12. Texas Instruments, «TMS320C55XX DSP Programmer´s Guide»
13. Texas Instruments, «TMS320C55XX DSP Reference Guide»
14. Texas Instruments, «TMS320C55XX Optimazing C_C++ Compiler v4.4»
15. D. Guicking, «Active Control of Sound and Vibration, History - Fundamentals - State of the Art,» Universitätsverlag Göttingen, Göttingen, 2007.
16. H. F. Olson, «Electronic Control of Noise, Vibration, and Reverberation,» The Journal of the Acoustical Society of America
17. Digital Signal Processing and Applications with the C6713 and C6416 DSK
18. Chr. Huygens, Traitė de la Lumiere (completed in 1678, published in Leyden in 1690)
19. Volver arriba↑ OS Heavens and RW Ditchburn, Insight into Optics, 1987, Wiley & Sons, Chichester ISBN 0-471-92769-4
20. Richard P. Feynman, Robert B. Leighton y Matthew Sands, *The Feynman Lectures on Physics*, Vol III, Capítulo 1. Addison Wesley (español).
21. Thomas Young, *Experimental Demonstration of the General Law of the Interference of Light*, "Philosophical Transactions of the Royal Society of London", vol 94, 2 (1804). El artículo puede encontrarse en Morris Shamos, ed., "Great Experiments in Physics" p96-101, Holt Reinhart and Winston, New York, 1959.
22. Code Composer Studio, User guide.
23. 1965 "A critical comparison of two kinds of adaptive classification networks", K. Steinbuch and B. Widrow, *IEEE Transactions on Electronic Computers*, pp. 737–740.
24. 1985 B. Widrow and S. D. Stearns. *Adaptive Signal Processing.* New Jersey: Prentice-Hall, Inc., 1985.
25. 1994 B. Widrow and E. Walach. *Adaptive Inverse Control.* New Jersey: Prentice-Hall, Inc., 1994.
26. 2008 B. Widrow and I. Kollar. *Quantization Noise: Roundoff Error in Digital Computation, Signal Processing, Control, and Communications.* Cambridge University Press, 2008.