Proyecto Fin de Carrera

ESTETOSCOPIO CON CANCELACIÓN DE RUIDO

Realizado por:
NATANAEL GIL VIDAL

Dirigido por:
Dr. FRANCISCO J. MARTÍNEZ GÓMEZ
Director del Grupo de Vibroacústica
Departamento de Ingeniería Mecánica

Ingeniería Técnica Industrial
Marzo 2011

Este proyecto fue realizado en la Universidad de Glasgow (*University of Glasgow*) durante el curso 2009/2010 bajo la supervisión del Dr. Fernando Rodríguez y la Prof. A. Catrina Bryce

Índice general

Capítulo 1

Objeto y alcance del proyecto

A lo largo de la historia de la medicina se han ido desarrollando tecnologías que permiten profundizar cada vez más en el funcionamiento del cuerpo humano. Desde los simples alicates hasta las complejísimas tomografías por emisión de positrones pasando por las prótesis la tecnología sanitaria ha ido siguiendo un desarrollo parejo a lo que las distintas ciencias han ido descubriendo.

Un instrumento que se asocia inmediatamente a los profesionales de la salud es el estetoscopio (también llamado fonendoscopio), es quizás una de las herramientas más versátiles cuya idea es bastante simple: ayudar a escuchar los sonidos internos del cuerpo humano (o animal). A pesar de este sencillo concepto, este producto también ha ido evolucionando y adaptando los conocimientos de otras ramas del conocimiento como es la electrónica.

Los primeros estetoscopios electrónicos hacían uso de electrónica analógica (bobinas, válvulas...) para amplificar el sonido y facilitar la escucha al facultativo.

La aparición del mundo digital supuso toda una revolución tecnología, procesos anteriormente muy costosos se han ido abaratando y agilizando a pasos agigantados, ideas antes atrapadas en las mentes de sus creadores, han podido materializarse. Miniaturización, bajo consumos energético, capacidad de manejo y procesamiento de grandes cantidades de datos... En esta nueva rama han ido apareciendo muchas nuevas disciplinas de las que, por supuesto, la medicina en general y los estetoscopios en particular se han podido ir nutriendo para llegar a lugares que antes eran impensables.

El objetivo de este proyecto es utilizar esta rama de la ciencia para diseñar un estetoscopio que sea capaz de ofrecer un sonido más limpio de lo que puede ofrecer un fonendoscopio normal, y hacerlo sin perder las propiedades prácticas que caracterizan a esta herramienta (facilidad de uso, portabilidad, ligereza, robustez...).

A la hora de realizar una auscultación el principal problema que se encuentra el profesional es el ruido ambiente, para un examen óptimo se requiere total silencio, algo que en la práctica es muy difícil por no decir imposible de conseguir. Al hablar de ruido ambiente en lo primero que se suele pensar es en el que interfiere directamente en el estetoscopio: se introduce por las "olivas", la

"campana", etc. y es el que tiene una solución más fácil: mejorar el aislamiento acústico. Pero este ruido también se introduce en el cuerpo que esta siendo auscultado y cuando el personal sanitario procede a escuchar los sonidos interiores, además del corazón, pulmones, circulación... se encuentra con los ruidos que ocurren en el ambiente. Lo peculiar de este ruido es que está modificado, el cuerpo actúa como un filtro que hace que el proveniente del exterior que es capturado por el estetoscopio a través del paciente sea distinto al que hay directamente en el exterior.

En este proyecto se estudiarán las técnicas de procesamiento de la señal necesarias para poder crear, mediante software, un filtro que se adapte a las distintas situaciones que se puedan presentar y que ofrezca robustez suficiente para, en caso de ser llevado a hardware, poder funcionar en unas condiciones en las que el uso de un estetoscopio común sería imposible.

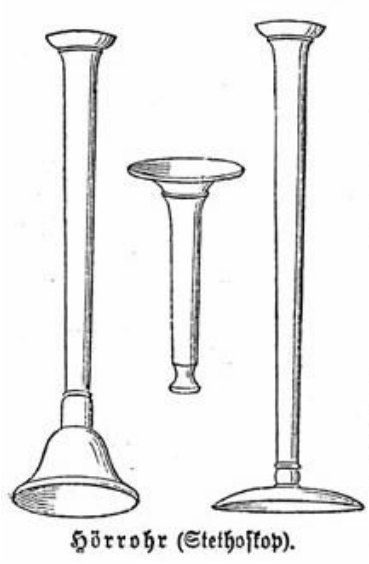
Capítulo 2

Antecedentes

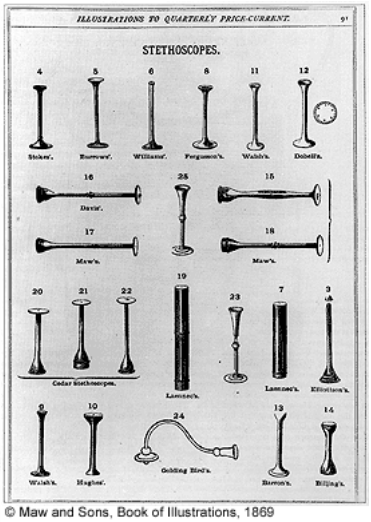
Antes de comenzar a trabajar directamente en el proyecto necesitamos plantearnos qué es lo que queremos (es decir, qué esperaría del producto un profesional de la medicina), cómo lo queremos y cómo podemos obtenerlo (la base tecnológica que es necesaria para alcanzar nuestros objetivos).

2.1. Médicos

"El estetoscopio es un dispositivo médico utilizado para la auscultación, escuchar los sonidos internos de un cuerpo animal"[1] Normalmente es utilizado para escuchar el sonido del corazón y de los pulmones, pero a veces también pueden interesar los intestinos y la circulación de la sangre.



(a) Primeros estetoscopios [2]



(b) Distintos modelos de estetoscopios antiguos [3]

Figura 2.1: Estetoscopios

Antes de la invención del estetoscopio, los doctores escuchaban el sonido del cuerpo directamente poniendo la oreja en el paciente, pero en 1816, Rene Theophile-Hyacinthe Laennec, que tenía aprensión a tocar directamente los cuerpos, cogió una "trompeta" de madera y descubrió que era mucho mas fácil escuchar los sonido mediante este método.

Desde entonces este instrumento se ha ido refinando a lo largo del tiempo. No hubo ningún desarrollo significativo en la siguiente década hasta que el Dr. Charles Williams dividió el instrumento en dos partes con una articulación que permitía que girara en diferentes ángulos algo que permitía al practicante mantener una postura más cómoda, aplicar una presión menor sobre el paciente y la posibilidad de observar las pulsaciones en el cuello. Durante los siguientes 40 años no hubo muchos avances aparte de la creación de un estetoscopio con dos campanas que iban cada una a una oreja. En 1926 Howard Sprague diseño la primera combinación de campana y diafragma que se ha mantenido (de una forma bastante refinada) hasta nuestros días y no fue hasta 1940 cuando este mismo doctor (en conjunto con Maurice Rappaport) investigó los principios físicos del estetoscopio. La primera aparición de un estetoscopio electrónico fue en 1961 desarrollado por Amplivox, debido a que usaba válvulas de vacío tenía un tamaño y peso considerables por lo que su uso fue abandonado enseguida. La revolución más grande vino con el Dr. David Littmann, que en ese mismo año diseñó un estetoscopio ligero con un sólo tubo binaural. Este modelo se extendió rápidamente y se convirtió en el más popular debido a su ligereza, flexibilidad y excelentes propiedades acústicas. Desde entonces esta herramienta ha ido mejorando y creciendo tanto en calidad como en popularidad. [5]

Figura 2.2: Partes de un estetoscopio [6]

Hoy en día, los estetoscopios acústicos siguen siendo los más utilizados, el sonido es transmitido a través de unos tubos llenos de aire, los estetoscopios

electrónicos (también llamados estetófonos) son un poco más complicados pero tienen más posibilidades. La forma más fácil de capturar el sonido es con un micrófono directamente sobre el paciente, pero de esta forma, ruidos externos pueden interferir fácilmente, por eso los fabricantes diseñan sus propios métodos para escuchar el sonido del cuerpo (piezoeléctricos, diafragmas electromagnéticos...). Además de procesar el sonido (amplificarlo, filtrarlo) como ya tenemos una señal electrónica, puede ser transmitido sin cables a dispositivos externos (procesamiento de señal más complejo, representación gráfica, grabación del sonido, tele-medicina, etc.) [1]

Los estetoscopios actuales tiene la posibilidad de filtrar las frecuencias del cuerpo utilizando directamente la *campana* o el *diafragma*, un lado actúa como un filtro paso alto que permite al usuario escuchar más claramente las frecuencias altas que podrían ser superpuestas por sonidos graves mas fuertes. Se me sugirió la posibilidad de implementar esta opción también en el código (que podría ser activada en el hardware mediante algún interruptor), pero considerando que en nuestro sistema la velocidad es un parámetro crítico, creo que es mucho mejor dejar este filtrado en una forma analógica para evitar un retardo extra.

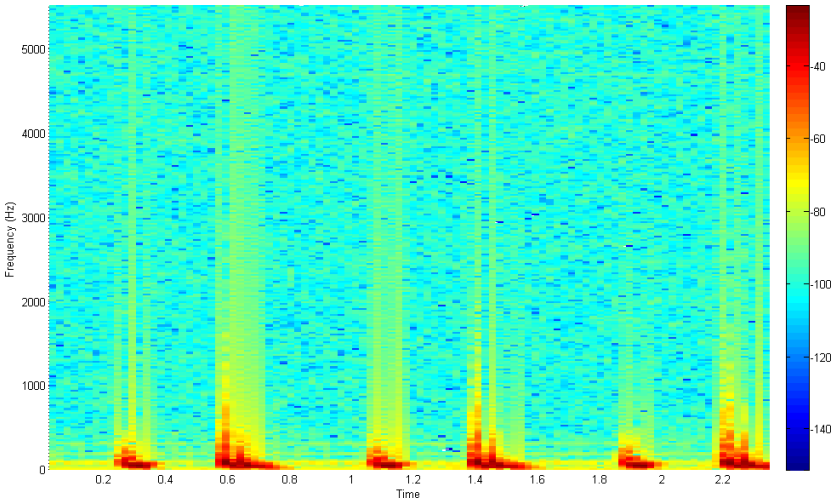
Como el objetivo de este producto es, principalmente, la aplicación médica en un ambiente más o menos controlado (podemos saber la mayoría de las situaciones en las que va a ser usado), podemos optimizarlo para que supla las necesidades lo mejor posible.

Un estetoscopio es un instrumento ligero y portátil, por lo que nuestro diseño tiene que cumplir los mismos requerimientos: el "corazón" del producto será un microprocesador, DSP... algo pequeño que pueda ser usado con comodidad. Al ser portátil también necesitamos que funcione con baterías (de pequeño tamaño), por lo que el consumo eléctrico tiene que mantenerse al mínimo, esto lo podemos conseguir eligiendo un circuito integrado adecuado; haciendo un buen diseño; manteniendo el algoritmo lo más simple posible, si se necesita procesar muchas operaciones, habrá que elevar la velocidad (el sistema también tiene que ser rápido, lo mas cercano a tiempo real que sea posible) y la energía necesaria también subirá, esta es la razón por la cual se ha escogido el algoritmo LMS para el filtrado (esto será explicado más en profundidad en el siguiente apartado); y la cantidad de datos tiene que ser la mínima para funcionar correctamente, demasiado baja y el sistema no será efectivo, demasiado alta y el sistema no sera eficiente.

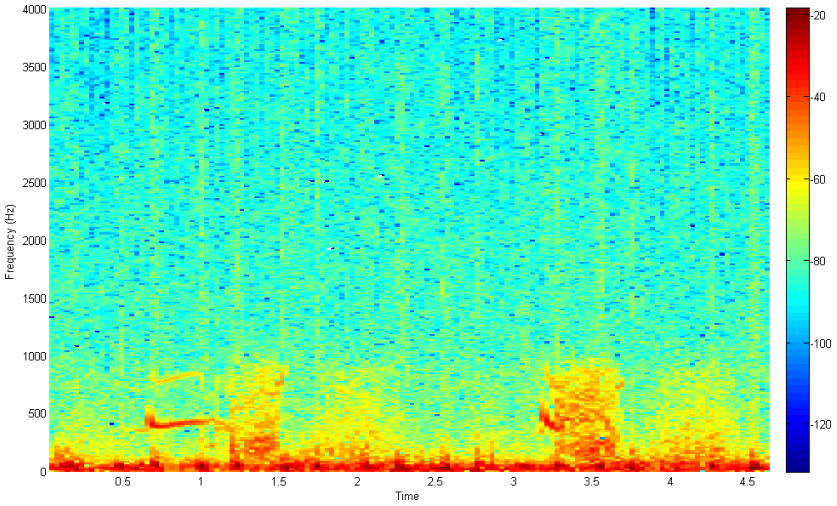
En este caso, la cantidad de datos está determinada por la *resolución* y la *frecuencia de muestreo*. La resolución determina cuántos valores podemos representar dentro de nuestros límites de trabajo: con una resolución alta podemos tener más valores (a costa de más tiempo de procesamiento), no he podido desarrollar esta parte en las especificaciones, pero el objetivo sería encontrar el valor mínimo que es suficiente para representar y procesar la señal sin mucha distorsión.

La frecuencia de muestreo es el número de muestras que tomamos en un segundo, el oído humano puede discernir hasta 20kHz, por eso la frecuencia mínima a la que deberíamos muestrear sería 40kHz y por lo tanto nuestro reloj debería ir mucho más rápido para permitir que nuestro sistema pueda hacer todas las operaciones necesarias entre capturas. Pero como he dicho anteriormente, este producto va a ser utilizado en un ambiente controlado, será utilizado para escuchar los sonidos en el interior del cuerpo, por lo que podemos inves-

tigar cuáles son estos sonidos y sus frecuencias ya que sabemos que las altas frecuencias no nos van a ofrecer ninguna información "interesante" sobre lo que ocurre dentro del cuerpo.



(a) Desdoblamiento del segundo ruido



(b) Sibiliaciones

Figura 2.3: Algunas de las muestras de corazón y respiración analizadas. Se puede encontrar el resto del ejemplos con una breve explicación en el apéndice A

Haciendo algo de investigación he encontrado que la frecuencia máxima generada por un cuerpo humano (para ser escuchada por un estetoscopio, por ejemplo la circulación de la sangre también genera sonidos, pero de alta frecuencia y no es algo que queramos escuchar en esta situación, al menos no en esta versión) está sobre 1kHz. Para estar más seguros sobre estos valores, lo mejor que se puede hacer es analizar diferentes sonidos que queremos escuchar, como no ha sido posible adquirir estos sonido en "vivo", después de algo de

investigación en Internet he encontrado un sitio web [7] diseñado para ayudar a los estudiantes de medicina y enfermería a conocer qué deberían buscar cuando están auscultando pacientes, la calidad de estas muestras no es muy buena pero es suficiente para nuestros propósitos (para tener unos sonidos más limpios y exactos se pueden comprar paquetes de sonidos de alta calidad).

Como se ve en la figura 2.3 casi toda la energía de las muestras está sobre 1kHz, algunas de ellas sólo un poco por encima, podemos coger un margen de seguridad hasta 1.5kHz, en tal caso la frecuencia de muestreos sería *3kHz* (muchísimo más baja que los 20kHz iniciales). Probablemente este estetoscopio será usado con bebés y niños, en ese caso los sonidos tendrán una frecuencias más elevadas, se podría hacer un poco investigación extra y encontrar cuáles son las frecuencias límites en este caso (he intentado encontrar información sobre estos casos pero sin resultados, al menos que puedan ser obtenidos de forma gratuita). Con esta información, en vez de elevar la frecuencia de muestreo todo el tiempo, se podria añadir un switch que permitiera al usuario utilizar esta opción si se va a auscultar a un niño; exceptuando el caso de que el facultativo esté trabajando con bebés y niños (pediatras, departamento de neonatos, etc.) la mayor parte del tiempo tratará con adolescentes y adultos por lo que sería una forma fácil de ahorrar bateria a la vez que se ofrece esta mejora.

2.2. Técnicos

2.2.1. Señales digitales

La principal diferencia entre una señal digital y una analógica es que cuando trabajamos con el primer tipo de señales, tenemos información durante toda la linea de tiempo, puntos infinitos; pero cuando queremos trabajar con una señal digital la cantidad de información que podemos manejar y procesar está limitada, lo que hacemos es coger una muestra cada cierto periodo de tiempo, de esta forma en vez de hablar de tiempo pasamos a hablar de "números de muestra".

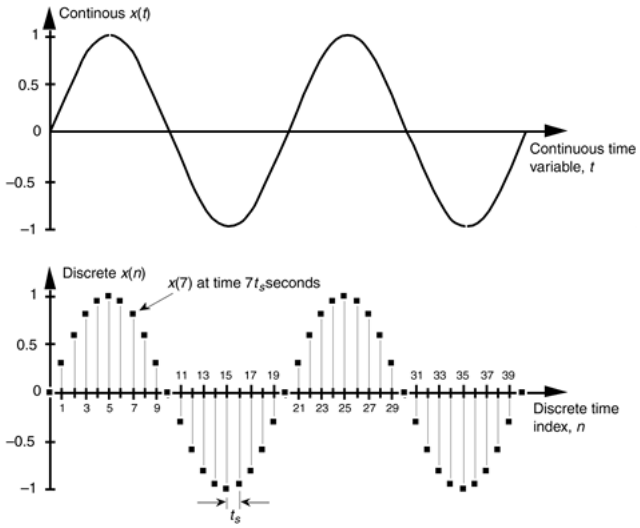


Figura 2.4: Diferencia entre una señal contínua de una discreta [8]

Otra forma muy útil de representar la señal es, en vez de utilizar el *dominio temporal*, usar el *dominio frecuencia*, analizar la señal y mostrar qué frecuencias la componen. Con esta representación es más fácil trabajar sobre la señal y modificarla para obtener lo que deseamos. Para saltar de un dominio a otro se utiliza una herramienta muy potente: *Transformada Discreta de Fourier* (y su inversa)

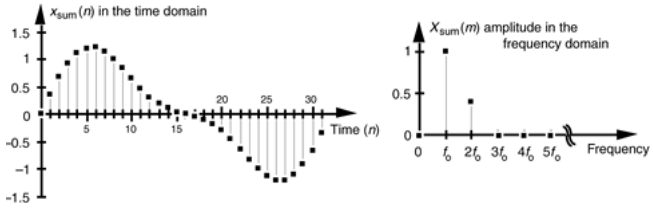


Figura 2.5: Ejemplo de representación en el dominio frecuencial [8]

2.2.2. DFT

La Transformada Discreta de Fourier (Discrete Fourier Transform) está basada en la Transormada de Fourier (2.1) que se usa en el mundo analógico, pero como este proyecto va a estar desarrollado completamente en el dominio digital, solo voy a hablar sobre la transformada discreta (2.2).

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-2j\pi ft}dt \tag{2.1}$$

$$X(m) = \sum_{n=0}^{N-1} x(n)e^{-2j\pi nm/N} \tag{2.2}$$

$$X(m) = \sum_{n=0}^{N-1} x(n)[\cos 2\pi nm/N - j \sin 2\pi nm/N] \tag{2.3}$$

Gracias a la *relación de Euler* la forma exponencial (2.2) puede ser escrita de forma rectangular (2.3) lo cual permitirá un punto de vista diferente. Antes de empezar a entender cómo trabajar con esta ecuación, necesitamos saber qué es cada componente:

- N es el número de muestras de entrada y de salida.
- $X(m)$ es la salida siendo m su índice (0 a $N-1$).
- $x(n)$ es la muestra de entrada.

Sólo necesitamos rellenar los valores y obtendremos un valor complejo por cada índice de salida. Para conocer la frecuencia de estos términos, utilizaremos la siguiente ecuación:

$$f_{analysis}(m) = mf_s/N \tag{2.4}$$

Como se ha dicho anteriormente, la salida de cada frecuencia es un valor complejo, por lo que tendrá dos componentes y seremos capaces de conocer tanto

la magnitud en determinada frecuencia como su fase y también su potencia $X^2_{mag}(m)$

Un propiedad importante de la DFT es su linealidad

$$X_{sum}(m) = X_1(m) + X_2(m) \tag{2.5}$$

Gracias a esta propiedad podemos analizar señales "reales" y no sólo senoides puras.

Otra propiedad muy útil es el *teorema de desplazamiento*

$$X_{shifted}(m) = e^{j2\pi km/N} X(m) \tag{2.6}$$

Si sabemos que los datos que estamos observando tienen un desfase de k muestras, todavia podemos obtener un resultado correcto.

IDFT

La DFT se utiliza para cambiar del dominio temporal al dominio frecuencial, si queremos hacerlo en la forma inversa, se utiliza una ecuación muy similar, la *Inversa de la Transformada Discreta de Fourier* (Inverse Discrete Fourier Transform).

$$x(n) = 1/N \sum_{m=0}^{N-1} x(n) e^{-2j\pi mn/N} \tag{2.7}$$

$$x(n) = 1/N \sum_{m=0}^{N-1} x(n)[\cos 2\pi mn/N - j \sin 2\pi mn/N] \tag{2.8}$$

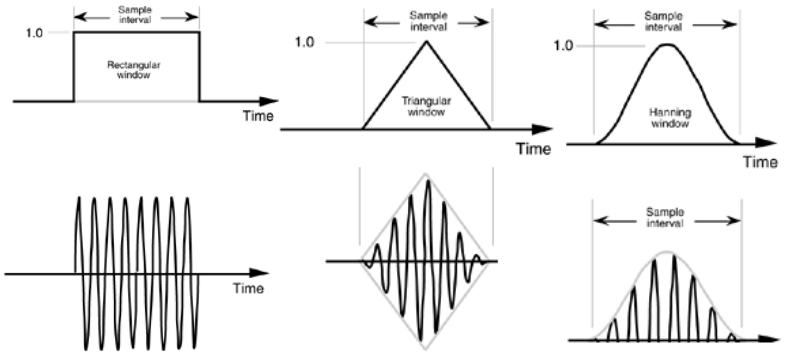
Usando los componentes frecuenciales en la ecuación exponencial (2.7) o en la rectangular (2.8), se obtendrá la magnitud (y fase) de cada muestra.

DFT leakage y ventanas

Saliendo del dominio de la teoría, las cosas se vuelven un poco mas complicadas de lo que se ha explicado anteriormente, las condiciones de trabajo no pueden ser controladas para tener un análisis perfecto. Por ejemplo, estamos analizando cierta señal a una frecuencia de muestreo f_s, cuando aplicamos la DFT, los resultados obtenidos son sólo para nf_s/N (siendo N enteros positivos) pero en el mundo real, tenemos cosas mucho mas complicadas que eso, la señal va tener frecuencias intermedias que en vez de ser invisibles para nosotros, se van a filtrar y van a tener una componente en todas las salidas, es lo que llamamos *leakage*.

Para mitigar este problema, lo primero que se nos podria venir a la cabeza es incrementar N, el número de salidas a leer, pero esto es algo que no es siempre posible debido a límites en el procesamiento, manejo de datos... y para estar seguros de que todas las frecuencias son tenidas en cuenta N deberia ser infinito.

El método seguido para reducir estas "filtraciones" se hace a través de *ventanas*, que consiste en cambiar la forma de cada intervalo de muestreo, esto se puede hacer fácilmente a través de la aplicación de expresiones matemáticas a las muestras antes de realizar la DFT.



(a) Ventana rectangular (b) Ventana triangular (c) Ventana Hanning

Figura 2.6: Algunos ejemplos de ventanas [8]

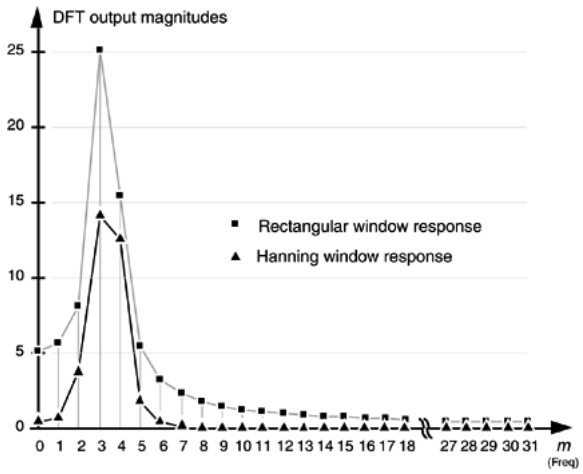


Figura 2.7: Efecto de una ventana Hanning [8]

En la figura 2.6 tenemos algunas de las expresiones (ventanas) mas utilizadas, cada una tiene diferentes propiedades y, dependiendo de en qué necesitemos fijarnos (resultados más limitados, magnitud más exacta...) escogeremos la más adecuada para cada sistema.

FFT

Como podemos imaginar, el DFT es un proceso *muy ineficiente* para analizar un señal, cuando los puntos del DFT se incrementan, los datos que debemos manejar se vuelven enormes. Por eso algunos métodos más rápidos han sido desarrollados (todavía se sigue trabajando en ello, es una ciencia), son lo que se llama *Transformada Rápida de Fourier* (Fast Fourier Transform, FFT), estos algoritmos obtienen resultados similares a la DFT estándar pero con un coste computacional mucho mas bajo. Hoy en día el más utilizado es el *radix-2 FFT*.

2.2.3. Filtros

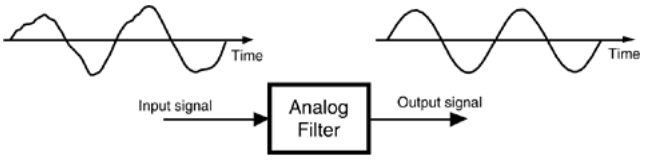


Figura 2.8: Filtro [8]

El procesamiento de señales está basado principalmente en lo que llamamos *filtros*, cogemos una señal y la modificamos con un filtro. Dependiendo del resultado, algunos tipos comunes de filtros son *paso bajo* (la salida sólo tendrá las frecuencias bajas de la señal de entrada), *paso alto* (sólo dejará pasar frecuencias altas), *paso banda* (sólo contendrá ciertas frecuencias entre un límite superior y otro inferior)...

Otros tipos de filtros, clasificados por *cómo* trabajan, son *Finite Impulse Response* (FIR, Respuesta finita al impulso) y *Infinite Impulse Response* filters (IIR, Respuesta infinita al impulso). Los filtros FIR son simples y estables mientras que los filtros IIR pueden ser más exactos pero también más inestables y pueden necesitar una gran carga de computación. Aquí sólo pasaré a explicar el primero que es el que se necesita en este proyecto.

Filtros FIR

Una de las principales propiedades de este tipo de filtros (también llamados no-recursivos) es que usa una cantidad limitada de datos, sólo entradas *pasadas y presentes*.

La operación de este filtro es básicamente, multiplicar el valor de k muestras por determinados coeficientes h y después sumar los resultados.

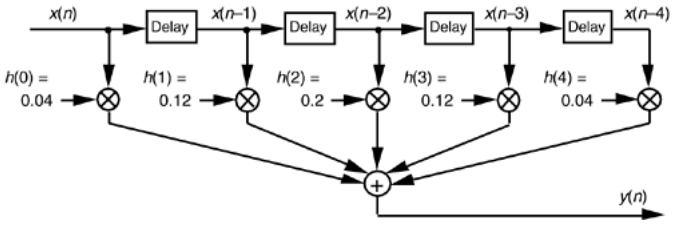


Figura 2.9: Cómo funciona un filtro FIR [8]

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k) \tag{2.9}$$

- M es el tamaño del filtro, el número de *taps*, cuántos valores son usados.
- n es el índice de salida.
- $h(k)$ son los coeficientes del filtro, k es el índice.
- x son los valores de entrada.

Esta operación recibe el nombre de *convolución*, y puede ser muy simplificada mediante DFT como se muestra en la ecuación (2.10)

$$y(n) = h(k) * x(n) \overset{DFT}{\underset{IDFT}{\rightleftharpoons}} H(m)X(m) = Y(m) \tag{2.10}$$

Para diseñar un filtro de este tipo "sólo" necesitamos calcular los coeficientes, a grandes rasgos esto puede hacerse siguiendo los siguientes pasos:

- Definir cuál va a ser la respuesta en el dominio frecuencial (para obtener un mejor resultado valoraremos el uso de ventanas).
- Convertir esa expresión al dominio temporal (IDFT) para obtener los coeficientes del filtro.
- Aplicar el filtro (convolución).

También hay rutinas de software que permiten hacer todo este proceso de forma que la posibilidad de cometer errores desciende considerablemente.

Toda la información explicada hasta este punto (Señales digitales, DFT, filtros...) ha sido estudiada y aprendida de [8]

2.2.4. Filtros adaptativos

Si las condiciones en las que fuera a trabajar nuestro producto fueran siempre las mismas, podríamos utilizar el filtro anteriormente descrito, pero obviamente esto no es lo que va a ocurrir, por eso necesitamos algo que nos permita trabajar en condiciones cambiantes y no predecibles.

Esto se hace a través de *filtros adaptativos*, filtros que en vez de tener los parámetros fijos, los ajustan ellos mismos.

El filtro escogido para hacer este proyecto es el llamado *Least Mean Squares* (LMS) y se ha elegido por ser estable y simple por lo que no necesitamos mucho tiempo de computación para obtener los resultados.

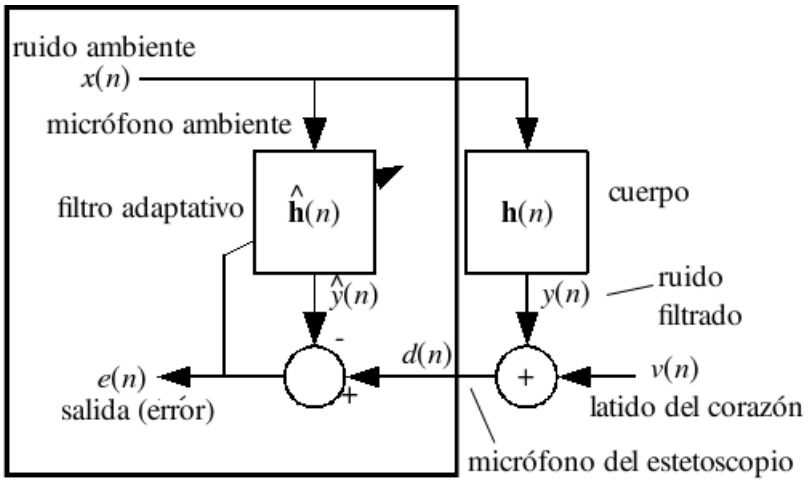


Figura 2.10: Filtro LMS [9]

Este filtro es básicamente un filtro FIR con coeficientes cambiantes, cada vez que lo aplicamos tenemos que actualizarlos mediante un algoritmo.

El algoritmo LMS es un algoritmo de gradiente estocástico (se adapta en base al error en el instante actual) que funciona como se explica a continuación.

$$\hat{h}(n+1) = \hat{h}(n) + \mu e*(n)x(n) \tag{2.11}$$

- h son los coeficientes, ahora llamados *weights*.
- e es el "error"; como se ve en la imagen, el sonido del estetoscopio menos nuestro sonido modificado.
- x es el sonido capturado por el micrófono ambiente.
- μ es el tamaño de paso, la "velocidad" a la que el filtro encuentra los "weights" óptimos. Un valor mas alto ofrece más velocidad pero más imprecisión, un valor más bajo va más lento pero obtiene mejores resultados.

El filtro LMS trata de minimizar las diferencias entre las dos entradas modificando una de ellas. Sin el sonido del corazón (interferencia), y con un filtro perfecto, la salida sería cero.

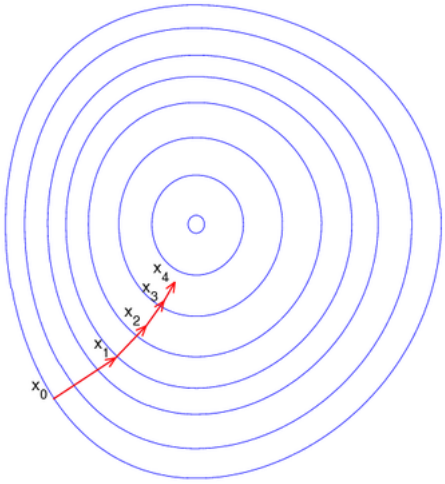


Figura 2.11: Con altos valores de μ el filtro estará más lejos del resultado deseado [10]

Si μ es muy elevado el sistema será inestable, para reducir este problema se puede utilizar el algoritmo *LMSnormalizado*.

$$h(n+1) = h(n) + \mu e*(n)x(n)/p \tag{2.12}$$

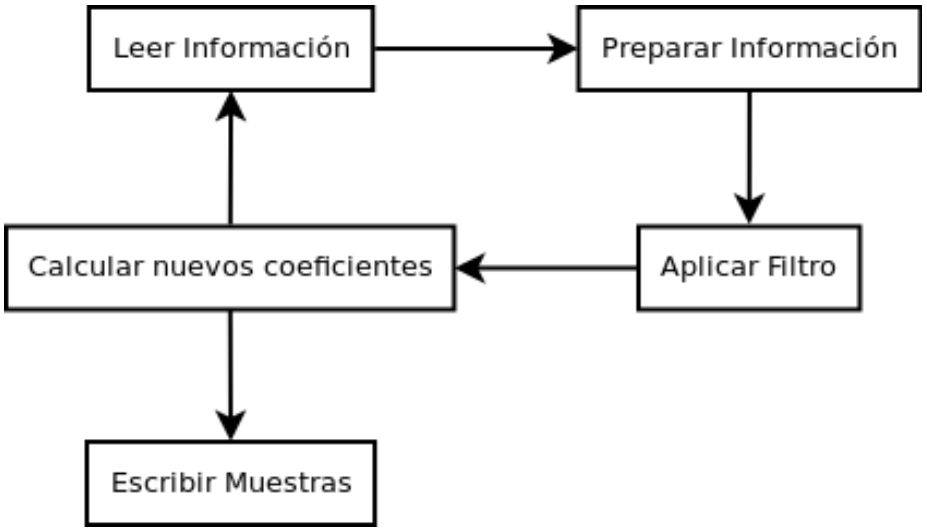
La única diferencia con LMS es dividir la potencia de las muestras que están siendo utilizadas, siguiendo la notación en la imagen 2.10 x^2.

Capítulo 3

Diseño e implementación

En este apartado sólo se adjuntarán los extractos de las partes del código más significativo, para consultar el código completo diríjase al apéndice D.

3.1. Diseño e implementación de un prototipo



Esta es una idea general de lo que hay que hacer.

Para la primera aproximación práctica a un prototipo, se ha elegido GNU/Octave porque es un software libre que tiene una forma sencilla de trabajar con señales digitales: matrices (en nuestro caso de orden nx1), procesamiento...

```
[dirty,fs] = wavread('file1.wav');
[noise,fs] = wavread('file2.wav');
```

Lo primero de todo, necesitamos leer los archivos que queremos procesar, esto puede hacerse con el comando *wavread*, en mi caso estoy leyendo el audio

y también la frecuencia de muestreo, esto es porque por defecto GNU/Octave considera que esta frecuencia es 22kHz.

```
u = noise(n:-1:n-M+1);
```

Como el filtro está basado en un filtro FIR, nuestro *vector de "weights"* debe tener los valores en un orden inverso por lo que nuestro siguiente paso será rellenar el vector de esta forma. Octave permite hacer esto con un pequeño ajuste en la función utilizada para extraer parte de un vector: podemos indicar la dirección (y periodo) en la que los datos serán leídos (por ejemplo, con 1 se leerá de forma "estándar", con 2 se leerá cada dos valores, con -1 en orden inverso, etc.) Esto no es completamente necesario pero ayudara a simplificar las operaciones que hay que hacer más adelante.

```
yn = w' * u; %Applying filter
e(n) = d(n) - yn; %Output

p = (u'*u);
w = w + (mu * u * e(n))/(p); %Normalized LMS
```

Con *GNU/Octave* podemos aplicar filtros fácilmente con la función *filter* o con la función *convolution*, pero en este caso he considerado apropiado utilizar algo que pueda ser portado más fácilmente a otro lenguaje de programación. Como ya tenemos preparados los valores, sólo tenemos que multiplicar la traspuesta de la matriz de los *weights* por la que contiene las *muestras de entrada* y el resultado será un número escalar (estamos trabajando con matrices de orden 1xn).

El siguiente paso será calcular el *error* del filtro, y esto se hace simplemente encontrando la diferencia de la señal que es captada directamente del cuerpo con la señal procesada.

Y por último, antes de leer nuevos datos, necesitamos actualizar los coeficientes haciendo uso del algoritmo en cuestión, como Octave permite trabajar fácilmente con matrices, simplemente tendríamos que escribir la ecuación (2.12), pero como en este proyecto sólo vamos a trabajar con señales reales, puede ser simplificado un poco más como vemos en (3.1).

$$h(n+1) = h(n) + \mu e(n)x(n)/p \tag{3.1}$$

Como ya sabemos que valores elevados de μ nos dan una rápida aproximación al resultado y valores más pequeños una aproximación más exacta, la única forma de tener un poco da cada es utilizar un valor de μ variable, al inicio, durante unas cuantas muestras, utilizaremos un valor bastante elevado (como estamos usando el algoritmo normalizado el filtro seguirá siendo estable aunque μ sea un número grande) y después lo cambiaremos por otro mas pequeño.

```
soundsc(e,fs);
```

Una vez que el archivo ha sido procesado, GNU/Octave nos permite escuchar el vector directamente, esto se hace a través de la función *sound*, tanto la lectura y escritura de archivos .wav se hace por defecto a 22kHz pero podemos indicarle nuestra propia frecuencia de muestreo. Otro comando interesante es la función

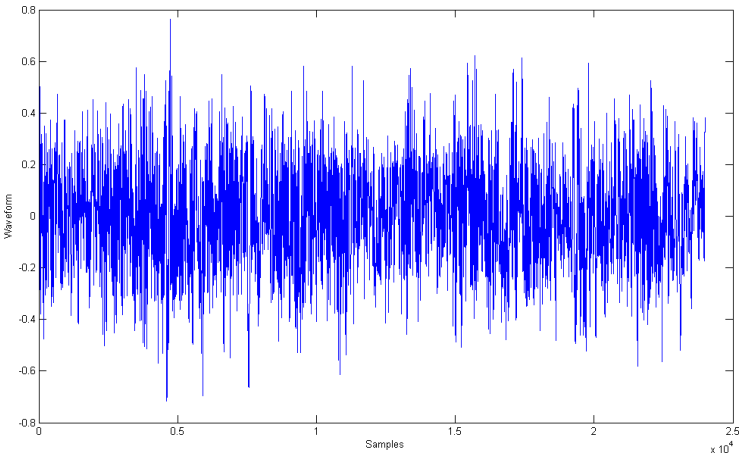
soundcs que hace los mismo que la anterior pero reproduciendo el sonido al máximo volumen posible sin que llegue a distorsionar.

Una vez que el prototipo ha sido diseñado y está funcionando correctamente con parámetros aleatorios tenemos que elegirlos de forma que cumplan nuestros requisitos:

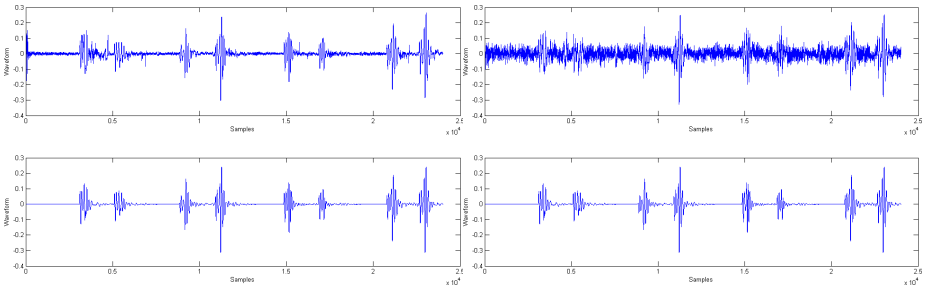
μ puede ser un número elevado porque estamos trabajando con el algoritmo normalizado *NLMS* por lo que voy a elegir primero un valor bastante alto, 0,5 que será reducido después de unos cuantos ciclos hasta alcanzar 0,01.

El otro parámetro clave en este tipo de filtro es el orden del filtro, necesitamos encontrar el equilibrio entre un buen desempeño y no demasiado coste de procesamiento (es el caso de un orden alto), por ello he comenzado las pruebas del filtro con valores elevados y luego los he ido bajando hasta que he encontrado un punto que me ha parecido adecuado, equilibrado.

Para hacer las pruebas de este filtro primero he buscado algunas muestras de sonido de corazón [11]. Para el ruido he cogido diversos sonidos (ruido, música, voces...) y los he mezclado juntos, esto será el *ruido ambiente*. Por último he filtrado este sonido con un filtro paso-bajo a 100Hz (después de algo de investigación he encontrado que esta es, más o menos, la frecuencia a la que el cuerpo humano adulto filtra los sonidos del exterior) y le he añadido el sonido del corazón, este sera el sonido capturado del cuerpo por el *estetoscopio*.



(a) El sonido antes de ser procesado



(b) 32 taps

(c) 16 taps

Figura 3.1: Respuesta del prototipo a filtros de diferente orden. Se pueden encontrar el resto del ejemplos apéndice B

Como podemos ver en la figura 3.1, 16 taps es muy bajo y 32 taps quizás podría ser reducido. He encontrado un buen equilibro utilizando 24 tpas; en la figura 3.3 podemos ver que la adaptación es conseguida bastante rápida, en menos de 250 muestras; los archivos de sonido están a una frecuencia de muestreo de 8kHZ por lo que esta adaptación toma menos de 50ms.

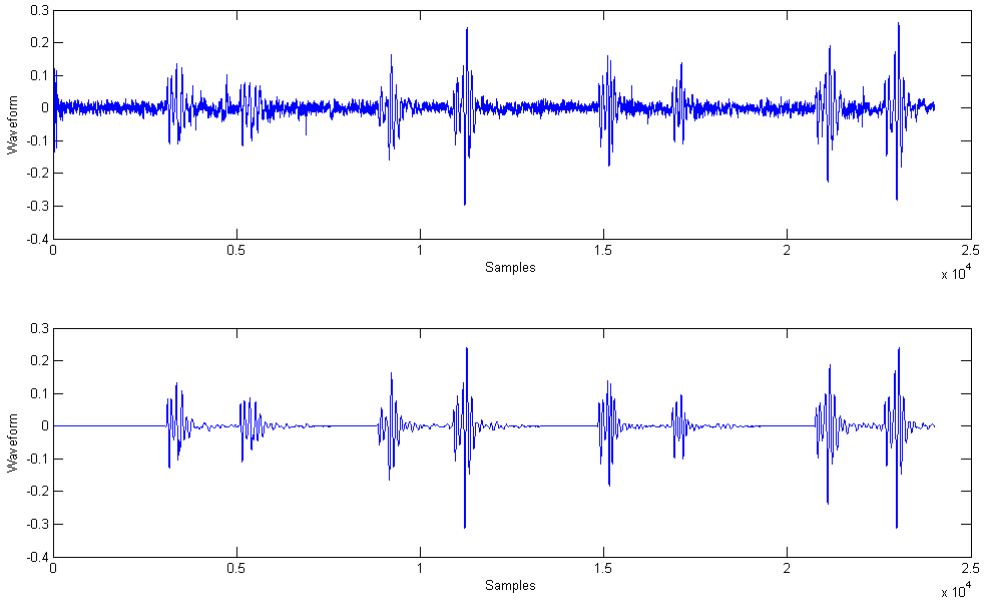


Figura 3.2: Respuesta del filtro con 24 taps

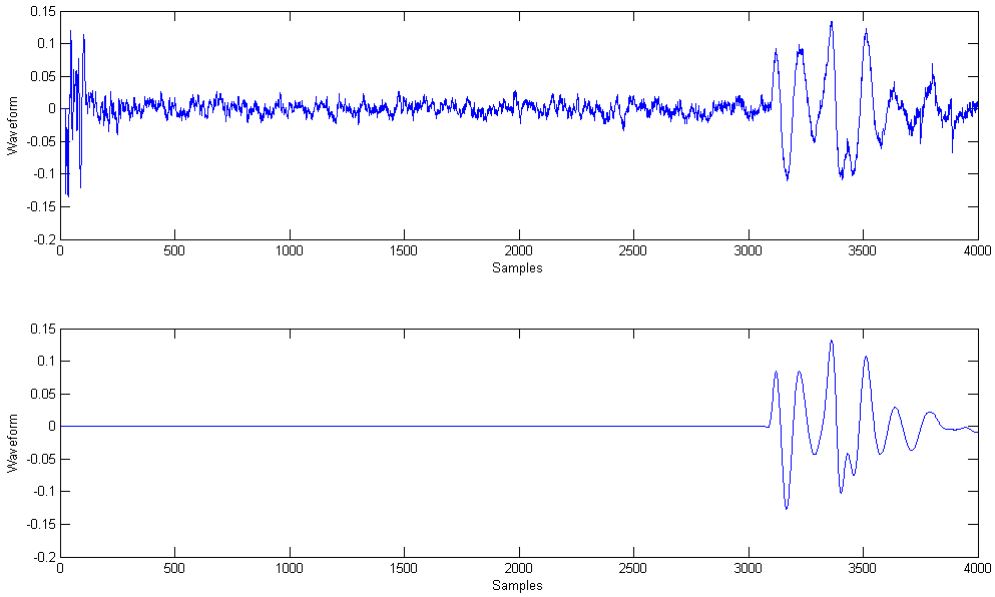


Figura 3.3: Mirada más cercana a la respuesta del filtro a 24 taps

3.2. Implementación de software

El siguiente paso en el proyecto es implementar el prototipo como un software funcional que nos permita probar el algoritmo, el lenguaje escogido ha sido C, que puede ser utilizado en microprocesadores, por lo que si, en el futuro, queremos implementar este código directamente en hardware, no serán necesarios grandes cambios. Para tener una mejor comprensión y probar si el programa funciona correctamente, primero he utilizado coma flotante en las operaciones de mi aplicación.

3.2.1. Coma flotante

Una de las principales diferencias con el código de GNU/Octave es la forma de manejar los archivos .wav. En GNU/Octave cargamos el archivo completo en un vector, leemos los valores almacenados en esa estructura, después de procesarlos los guardamos en un diferente vector y finalmente escribimos eso datos en un archivo (o lo reproducimos directamente). Si quisiéramos hacer esto en C, necesitaríamos disponer de una gran cantidad de memoria disponible y otro problema es que la perspectiva para este código es que sea implementado en hardware para trabajar en el mundo real y en esta situación sólo tenemos acceso a muestras pasadas (si las vamos almacenando) y presentes, por ello tenemos que hacer algo que se asimile a cómo funcionará el producto final: vamos a leer las muestras directamente de nuestro archivo y escribir el resultado directamente en otra.

```
noiseFile = fopen("file1.wav","rb");
dirtyFile = fopen("file2.wav","rb");
outputFile = fopen("output1.wav","wb");

char buffer [44];
fread(buffer, 1, 44, noiseFile);
fwrite(buffer, 1, 44, outputFile);

fseek(dirtyFile, 34, SEEK_SET);
fread(&bs, 2, 1, dirtyFile);
Bs = bs / 8;
fseek(dirtyFile,44,SEEK_SET);

while (!feof(dirtyFile)) {
    fread(&readInputBody, Bs, 1, dirtyFile);
    fread(&readInputAmbience, Bs, 1, noiseFile);
    ...
```

C permite acceder a los archivos en modo binario mediante el proceso *fopen*, necesitamos indicar en los parámetros si queremos acceder a este archivo en modo de lectura, escritura o ambos.

Como estamos leyendo los archivos en modo binario necesitamos saber cómo es el formato. Mirando las especificaciones vemos que los archivos .wav tienen un encabezado de 44 bytes. Este programa va a ser ejecutado en un ambiente controlado: nuestros dos archivos serán de la misma longitud, y por lo tanto,

la salida también tendrá la misma longitud por lo que podemos utilizar el encabezado de cualquiera de los ficheros de entrada para crear el de salida. Dos valores importantes para nosotros en este código son la *frecuencia de muestreo* y el número de *bits por muestra*, en este caso sólo necesitamos leer el número de bits por muestra (tenemos que saber dónde empieza y acaba cada muestra). Podemos hacer fácilmente en C con el comando *fseek* que mueve el puntero del fichero a determinado byte. Para leer bits se utiliza el comando *fread*, aparte del fichero que queremos leer y donde salvar la información leída, necesitamos indicar el número de bytes por bloque, y el número de bloque para leer.

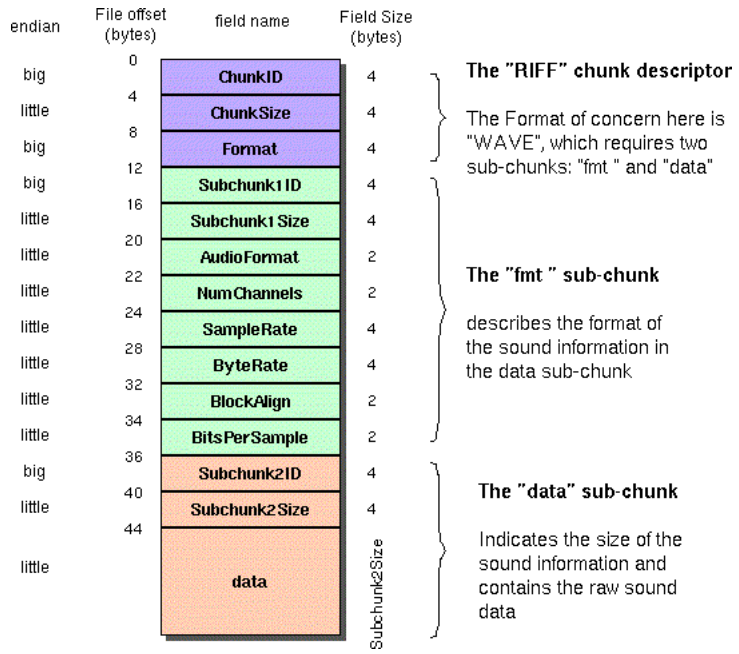


Figura 3.4: Especificaciones del formato .wav

El filtro no puede trabajar hasta que tiene suficientes valores, por eso las primeras muestras son simplemente almacenadas y copiadas en el fichero de salida, cuando el vector tiene suficientes datos, el filtro comienza a trabajar. Igual que en el código de GNU/Octave, tengo tres valores diferentes asignados a μ, para cambiarlos utilizo un contador que deja de trabajar cuando llega al tercer y último valor de μ.

Para almacenar nuestros valores muestreados necesitamos hacer uso de una estructura FIFO (First In First Out), esto es, que el primer valor que se introduce será el primero en ser borrado. La primera idea fue crear y utilizar una cola, pero quizás eso complicaría significativamente el código y no seria la solución mas sencilla (sólo es necesitada para una variable) por lo que he decidido utilizar simplemente un vector e ir moviendo todos los valores cada vez que se introduce nueva información. Este almacenaje se hace de forma inversa ya que es el vector que utilizaremos para los "weights".

```
yn = 0;
for (i = 0; i<ORDER;i++){
```

```
    yn += w[i]*u[i];
}
```

En C no podemos manejar los vectores como matrices, por eso para aplicar el filtro (hacer la convolución), necesitamos multiplicar cada uno de los valores de los vectores, como hemos preparado una de las estructuras anteriores de forma que se rellena inversamente, no hay ningún problema en realizar esta operación directamente sin pasos intermedios.

```
e = calcBody - yn;

if (e > 1) {e = 1;};
if (e < -1) {e = -1;};
```

Después de obtener el error (nuestra salida) comprobamos que el resultado está dentro de unos límites de trabajo, en caso de salirse, simplemente saturamos el valor, trabajando de esta forma siempre tenemos control sobre lo que está ocurriendo en el programa.

```
power = power*(ORDER-1)/ORDER + u[0] * u [0] /ORDER;
```

Como estamos utilizando el algoritmo *LMS Normalizado* necesitamos calcular la potencia de la parte de la señal que estamos manejando, esto requiere bastante procesamiento, en cada iteración se necesita multiplicar y sumar N veces (siendo N el orden del filtro), una forma de relajar este requerimiento sería almacenar todos los valores que están siendo utilizados para trabajar con el filtro y en la siguiente iteración sumar el nuevo y restar el mas viejo, pero esto requiere bastante memoria; la mejor elección sería lo que recibe el nombre de *media móvil*.

He estado tratando de implementar esto en mi código pero sólo he obtenido resultados erróneos, creo que la razón es que en este caso en concreto los valores son muy diferentes unos de otros por lo que la aproximación no es válida. Como cuando se realiza el cambio a coma fija más adelante, esto no se necesita, he decidido dejar las operaciones en la forma "teórica" (multiplicando todos los valores uno por uno).

```
for (i = 0; i<ORDER;i++){
    w[i] = w[i] + e*mu*u[i]/power;
}
```

El siguiente paso es actualizar los "weights", necesitamos volver a iterar para sumar los términos de los vectores.

La última operación antes de leer los siguientes valores de los archivos es escribir las salidas en el fichero de salida, se usa el proceso *fwrite* con los mismos parámetros que se han utilizado anteriormente con *fread*.

3.2.2. Coma fija

Como un paso más hacia la implementación en hardware, en vez de utilizar coma flotante que es la forma más fácil para hacer operaciones matemáticas pero necesita una *unidad de procesamiento de punto flotante* para que sea eficiente (y

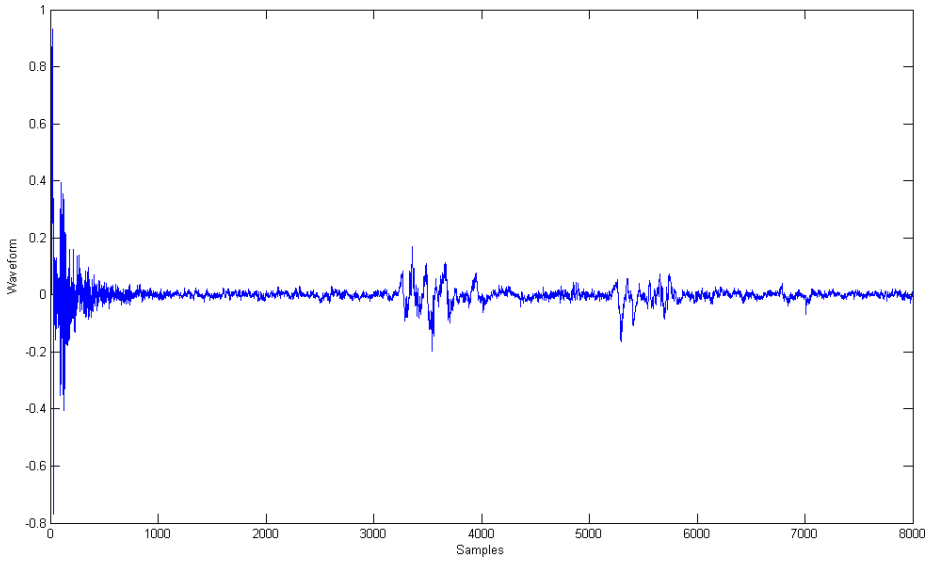


Figura 3.5: Salida del código en C usando *coma flotante*

probablemente el hardware utilizado para el producto final no dispondrá de ella), implementaré el programa haciendo uso de la *coma fija*. Como explicación rápida se puede decir que cuando estamos utilizando número decimales en nuestro sistema, dividimos nuestra variables en dos partes y consideramos algunos bits como el entero y el resto de bits como la parte decimal, esto limitará el número de valores que podemos representar y perderemos resolución en nuestros decimales, pero seremos capaces de trabajar con número reales. Para hacer uso de este sistema sólo necesitamos crear algunos procesos para reemplazar las operaciones estándar de punto flotante. En mi caso he decidido considerar todos los bits como parte decimal, quiero decir, sin parte entera.

Primeramente, para tener un código reutilizable, me defino un nuevo tipo *QRESOLUTION*, que me permite cambiar la resolución a la que estoy trabajando simplemente cambiando esa definición.

```
QRESOLUTION AddFix(QRESOLUTION a, QRESOLUTION b) {
    long int c;
    c= (int)a+(int)b;
    if (c>=MAX_POS) c=MAX_POS;
    if (c<=MAX_NEG) c=MAX_NEG;
    return ((QRESOLUTION)c);
}
```

Comenzando con la suma, el primer problema a considerar es la posibilidad de tener desbordamiento, probablemente trabajemos con un microprocesador que disponga de un flag que nos indique si esto ha ocurrido después de hacer una operación, pero en nuestra situación (trabajando exclusivamente con software) la forma más fácil de tener en cuenta este problema es almacenando el resultado de la operación en una variable más grande, comprobar si el resultado esta fuera de nuestros límites, y, en caso, saturarlo forzando el valor máximo (o mínimo).

```
QRESOLUTION MultFix(QRESOLUTION a, QRESOLUTION b) {
    long long int c;
    c=(long long)a*(long long)b;
    c=c>>RESOLUTION;
    if (c>=MAX_POS) c=MAX_POS;
    if (c<=MAX_NEG) c=MAX_NEG;
    return (QRESOLUTION)c;
}
```

Para la multiplicación utilizamos el mismo concepto, pero necesitamos tener en cuenta una cuestión extra, como estamos trabajando con decimales también tenemos que desplazar los bits hacia la derecha para obtener un resultado correcto ($(3 * 5 = 15$ pero $0{,}3 * 0{,}5 = 0{,}15)$)

La división en coma fija requiere unos procesos muy complicados, y teniendo en cuenta que este es un sistema donde la velocidad es muy importante, he decidido no usarla: sólo es necesaria para normalizar el algoritmo LMS, pero esta no es una especificación crucial porque podemos evitar que el sistema se inestabilice escogiendo valores bajos de μ ya que no necesitamos un tiempo de adaptación muy bajo, además el sistema ya es bastante rápido con el algoritmo LMS estándar, teniendo esto en cuenta he fijado el valor de μ a 0,01.

3.2.3. Pruebas

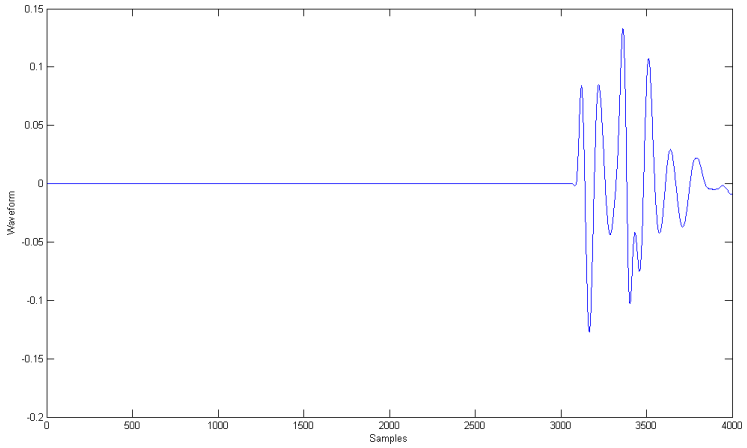
Una vez que el filtro está funcionando correctamente le he realizado algunas pruebas para conocer sus limitaciones. Para ello he preparado diferentes muestras con distintas relaciones entre el sonido del corazón y el ruido "infiltrado", entonces he aplicado el filtro y analizado los resultados. En la figura 3.6 podemos ver el efecto de hacer el parámetro μ más pequeño, ahora el filtro es más lento pero aún así sólo necesita menos de 500 ciclos para llegar a un resultado aceptable. Conforme el ruido se hace mas fuerte que el sonido del corazón, el filtro comienza a tener dificultades para limpiar la muestra y finalmente no puede distinguir el corazón.

La figura 3.7 muestra que en algunos sonidos el latido del corazón no puede ser reconocido viendo la forma de onda (figure 3.6), a pesar de ello está todavía ahí. Esto se puede confirmar observando los resultados de una encuesta que he realizado a diferentes personas, algunas de ellas podían distinguir el sonido del corazón incluso cuando este era totalmente indistinguible mirando la forma de onda.

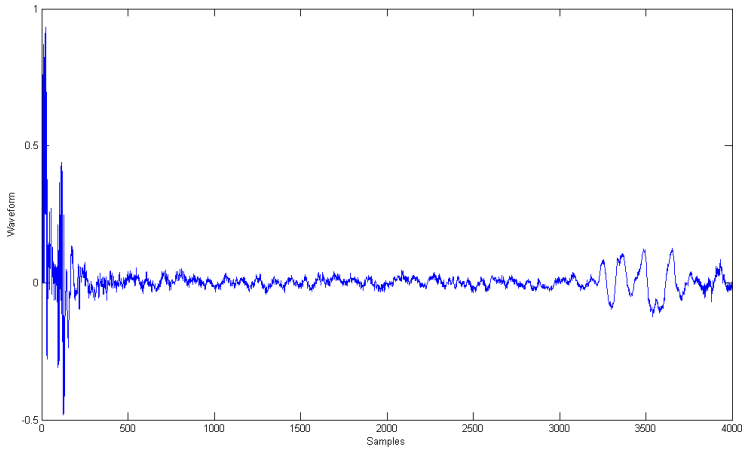
He reproducido las muestras a 21 personas de diferentes edades entre 16 y 40 años (algunas de ellas estudiando o trabajando en medicina, enfermería...) y les he preguntado en qué ejemplos podían escuchar el corazón obteniendo los resultados mostrados en la figura 3.8.

Género	**Muestras elegidas en dB**											
Mujeres	-27	-27	-22	-32	-32	-27	-27	-17	-22	-27		
Hombres	-27	-22	-12	-22	-27	-22	-27	-22	-17	-27	-22	-17

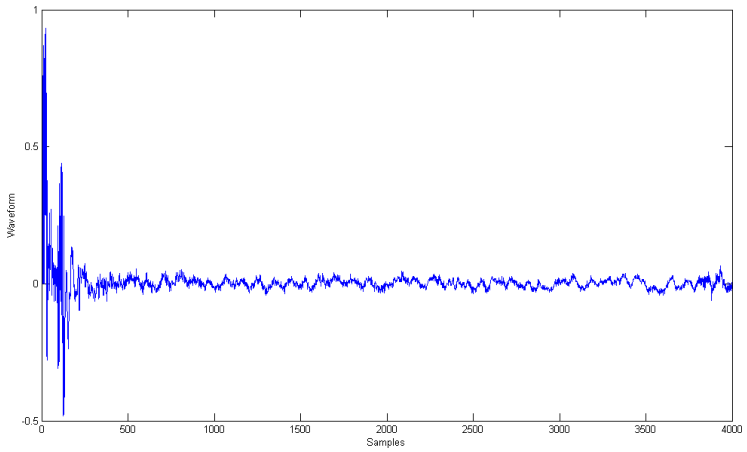
Figura 3.8: Resultados de la encuesta



(a) Sólo corazón

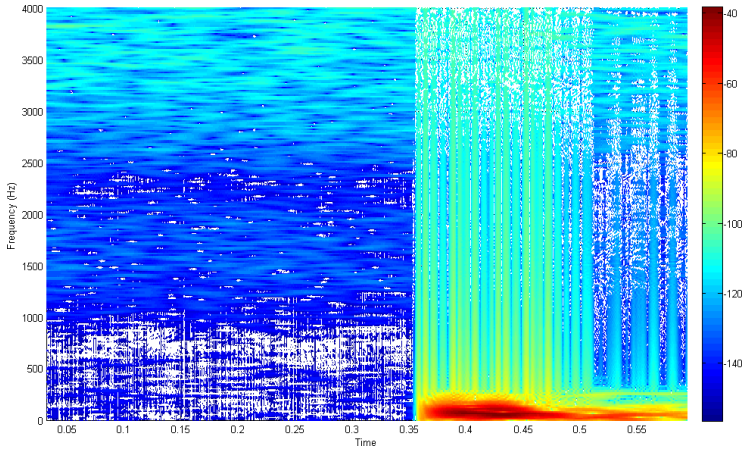


(b) -12dB

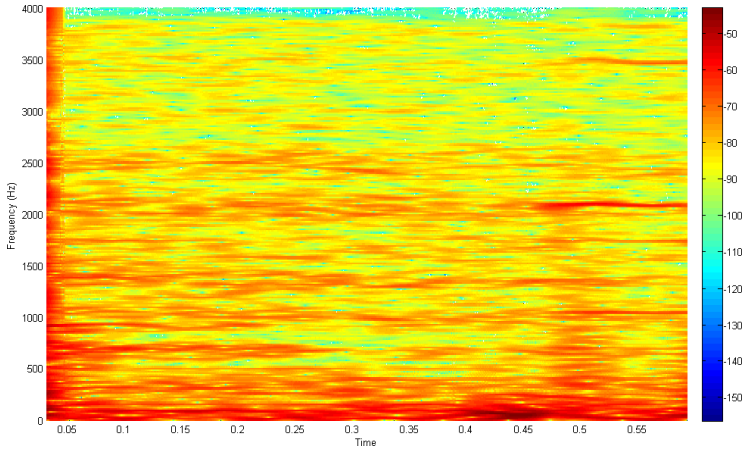


(c) -32dB

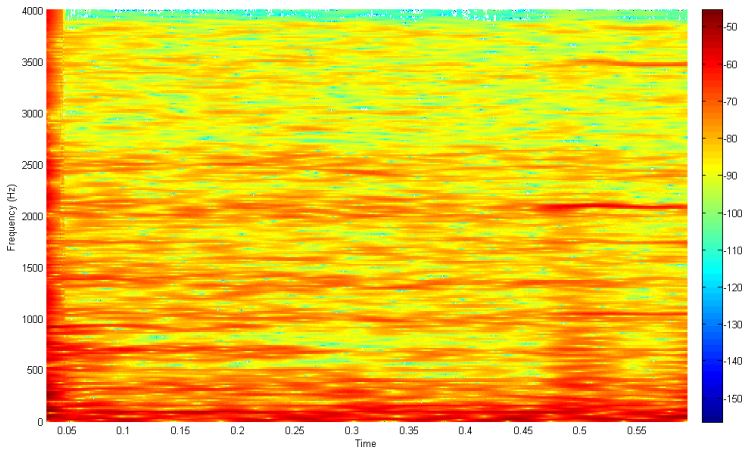
Figura 3.6: Respuesta del filtro a diferentes relaciones entre el ruido y el sonido del corazón, el resto de ejemplos se pueden encontrar en el apéndice C.1 (Sólo las 4000 primeras muestras).



(a) Sólo corazón



(b) -17dB



(c) -32dB

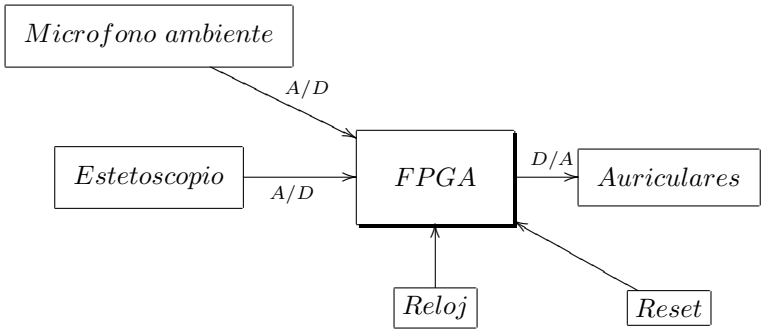
Figura 3.7: Espectrograma de la respuesta a diferentes relaciones entre el ruido y el sonido del corazón, el resto de ejemplos se pueden encontrar en el apéndice C.2 (Sólo las 4000 primeras muestras).

Esta encuesta ha sido realizada en una situación controlada, usando el mismo equipamiento en todo momento.

Como podemos ver, para la mayoría de la gente, una diferencia de 22dB - 27dB es suficiente para escuchar claramente el corazón, desde mi punto de vista es difícil tener estas relaciones de ruido dentro del cuerpo por lo que la adaptación es suficiente.

3.3. Planteamiento de implementación en Hardware

Una vez que la implementación en software es completamente funcional, el siguiente paso es enfocarnos en el hardware, esto se puede realizar en dos formas: elegir un microcontrolador o DSP y modificar el codigo en C ahora escrito para que funcione en este nuevo ambiente o diseñar una *FPGA* (Field Programmable Gate Array) que nos dará más velocidad. La implementación en un microcontrolador sería bastante directa, por eso he decidido hacer una pequeña aproximación a lo que sería la implementación en una FPGA.



Este hardware necesita ser "programado" en un modo completamente diferente, está enfocado a hardware electrónico (como su nombre indica es una matriz de puertas lógicas), uno de los lenguajes para hacerlo es *VHDL*; como era la primera vez trabajando con él (y nunca he trabajado con diseño digital, con lo que se requeriría un tiempo elevadísimo de aprendizaje) y el objetivo final es simplemente simular el código, no construir un hardware, en vez de trabajar en todo el programa con *señales* y *operaciones lógicas*, lo he realizado utilizando la librería "IEEE.NUMERIC_STD" que permite trabajar numéricamente con *variables*.

```
entity Filter is
 Port (
  Micro : in  std_logic_vector(15 downto 0);
  Estetoscopio : in  std_logic_vector(15 downto 0);
  Salida : out  std_logic_vector (15 downto 0);
  clk: in std_logic;
  rst: in std_logic
 );
end Filter;
```

Un código VHDL estándar está dividido en dos partes, primero necesitamos definir lo que se llama *entidad*, esto es donde definimos nuestro sistema como una "caja negra", sólo necesitamos saber cuáles son las entradas y las salidas. En este caso necesitamos dos entradas de datos (micrófono ambiente y estetoscopio) y una salida (los auriculares, sólo una línea en mono, no merece la pena el trabajar en stereo); he fijado los tamaños a un array de 16 bits cada una porque en los archivos que estoy utilizando en la simulación, el número de bits por muestra son también 16.

También se necesita un reloj y he añadido un botón de reset en caso de que el usuario quiera comenzar el filtrado de nuevo.

```
architecture Behavioral of Filter is
.
.
.
end Behavioral;
```

La otra parte es la *arquitectura* que es donde escribimos qué es lo que nuestro programa hace.

Como he comentado, al ser una primera aproximación a VHDL, sólo utilizo variables y asignaciones directas en el código, por lo que va a ser muy similar al que tenemos en C.

Los puntos críticos serán una vez más, la suma y la multiplicación.

```
function AddFix (b,a:sample) return sample is
variable c: signed (2*RESOLUTION-1 downto 0)
    := (others => '0');
variable d: sample;
begin
        c := to_signed(to_integer(a),2*RESOLUTION)
             +to_signed(to_integer(b),2*RESOLUTION);
        if (c>=MAX_POS) then
                d:=MAX_POS;
        elsif (c<=MAX_NEG) then
                d := MAX_NEG;
        else
           d:= to_signed(to_integer(c),RESOLUTION);
        end if;
return d;
end function AddFix;
```

Primero de todo he definido un nuevo tipo de *muestra* que permitirá cambiar la resolución sin cambiar mucho código, en realidad es simplemente un *signed* con el tamaño de la resolución.

El método utilizado es, nuevamente, hacer la operación, almacenar el resultado en una variable más grande y a continuación comprobar si es un resultado razonable. Pero como VHDL es un lenguaje *fuertemente tipado*, necesito hacer algunos arreglos.

Hay que almacenar el resultado en una variable más grande que los operandos, para hacer esto todas las variables tienen que ser del mismo tipo y tamaño, por lo que necesito cambiar los operando de ser *signed* de un tamaño

determinado a ser *signed* pero de distinto tamaño, VHDL no permite hacer esto directamente, la forma de es convertir nuestro *signed* en un *integer* para finalmente convertir este último en nuestro nuevo tipo *signed*.

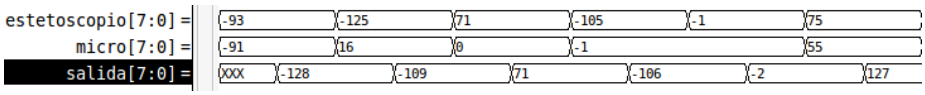


Figura 3.9: Funciones para la suma: *estetoscopio* + *ruido* = *salida*

El único problema con esta conversión es que estamos limitados a un máximo de 16 bit que es la máxima longitud de un entero.

```
function MultFix (b,a:sample) return sample is
variable c: signed (4*RESOLUTION-1 downto 0)
    := (others => '0');
variable d: sample;
begin
        c:= to_signed(to_integer(a),2*RESOLUTION)
            *to_signed(to_integer(b),2*RESOLUTION);
    c := shift_right (c,RESOLUTION);
        if (c>=MAX_POS) then
                d:=MAX_POS;
        elsif (c<=MAX_NEG) then
                d:=MAX_NEG;
        else
           d:= to_signed(to_integer(c),RESOLUTION);
        end if;
return d;
end function MultFix;
```

Para la multiplicación necesitamos hacer lo mismo otra vez, además de desplazar el array para tener un resultado correcto en la parte decimal (en este caso sería el array completo).

Banco de pruebas

Como VHDL está preparado para diseño de hardware, para realizar una simulación necesitamos preparar un *banco de trabajo*, otra pieza de código en la que escribiremos la entradas y salidas del sistema.

```
variable linea:line;
file micFile:text open read_mode is "mic.dat";
readline(micFile,linea);
read(linea, micData);
```

Para probar este sistema se necesita una gran cantidad de datos, no podemos ver si el sistema está funcionando correctamente simplemente leyendo unos cuantos valores, por eso la forma más fácil de trabajar sería leer directamente el archivo .wav VHDL no permite esto pero permite leer arrays de caracteres de un fichero de texto. He creado dos pequeñas aplicaciones en C (se pueden

ver en el anexo D.4) que transforman un archivo .wav en un fichero de texto (y viceversa). El formato de este archivo es una muestra por línea, y cada muestra es en binario (unos y ceros). Ahora ya podemos realizar la simulación, sólo necesitamos transformar el archivo .wav utilizando *wav2txt.c*, después ejecutar el programa VHDL que nos dará una salida en un fichero de texto que puede ser transformado a .wav de nuevo utilizando *txt2wav.c* para comprobar que el fichero está funcionando.

Capítulo 4

Conclusiones

Los resultados conseguidos en este proyecto han sido bastante satisfactorios.

Esta ha sido mi primera aproximación al "mundo de la señal digital" y estoy agradablemente sorprendido del potencial de este área de la ingeniería.

Creo que el filtro está trabajando muy bien, con un algoritmo muy sencillo consigue resultados espectaculares en muy poco tiempo, suficiente para cumplir con los requisitos de diseño, no creo que la relación entre los sonidos del cuerpo y las interferencias externas que se filtran sean tan exageradas como la que he estado utilizando en las muestras por lo que podría ser utilizado en situaciones reales.

No he obtenido muchos resultados visibles y palpables pero definitivamente el realizar este proyecto ha merecido la pena. He necesitado aprender desde lo más básico sobre señales digitales, procesamiento digital, filtros, lenguajes de programación... obviamente no he podido profundizar mucho en cada una de las áreas por la cantidad de tiempo que sería necesaria e incluso en algunos de ellas no tengo base suficiente para comprenderlas completamente.

Futuro

Basado en el trabajo realizado hasta ahora, se deja abierto el proyecto a una futura implementación en hardware. Podría usarse un microcontrolador o un DSP para lo que se debería adaptar el código en C, y probablemente añadir alguna optimización para algún tipo particular de hardware. Otro camino a seguir sería implementar la FPGA, esto sí que requeriría más recursos y trabajo en el código ya que debería ser reescrito al estar preparado actualmente exclusivamente para simulación.

También se podría desarrollar un software de procesamiento y gestión de los datos una vez capturados tales como visualización de los sonidos, posibilidad de enviar las muestras por medios digitales (sin tener que hacer que el paciente se tenga que desplazar), almacenaje de las muestras para un posterior análisis o comparación, desarrollo de aplicaciones de detección automática, esto es, que basándose en una análisis inteligente de las muestras, sin intervención del facultativo (excepto para posicionar el estetoscopio en las zonas correctas) se pudiera indicar qué afecciones podría estar padeciendo el paciente.

Apéndice A

Afecciones del corazón y pulmones

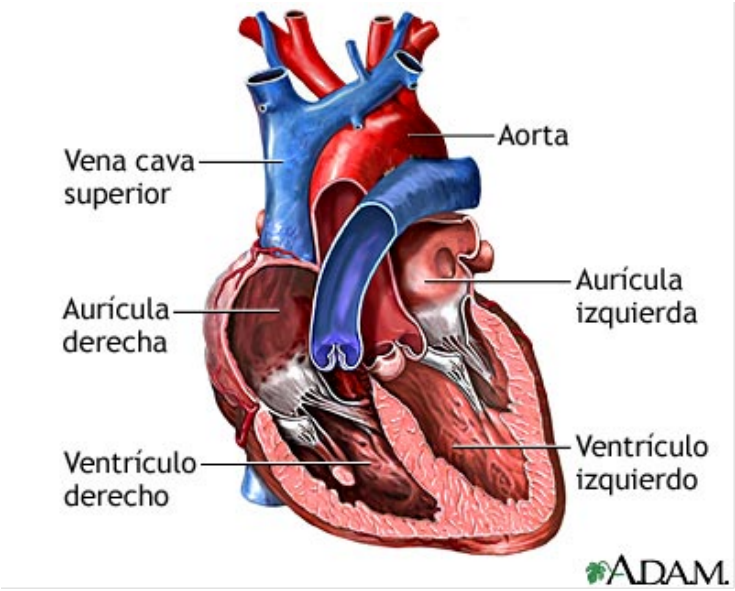


Figura A.1: Partes del corazón [13]

El corazón es el órgano principal del aparato circulatorio. Anatómicamente está subdividido en cuatro cavidades, dos derechas y dos izquierdas, separadas por un tabique medial. De manera que funcionalmente hablamos de dos bombas, que no comunican entre sí: el corazón derecho (que recibe sangre de los órganos periféricos y la bombea a los pulmones) y el izquierdo (que recibe sangre oxigenada de los pulmones y la bombea de nuevo a los órganos periféricos). Las dos cavidades superiores son llamadas aurículas, y las cavidades inferiores se denominan ventrículos. Cada aurícula comunica con el ventrículo que se encuentra por debajo mediante un orificio (orificio auriculoventricular), que puede estar cerrado por una válvula denominada tricúspide en el lado derecho, y mitral en

el izquierdo. De los ventrículos salen las arterias (Pulmonares en el derecho y Aorta en el izquierdo), y a las aurículas afluyen las venas (cavas en la derecha, y pulmonares en la izquierda).El órgano está contenido por dos hojas, una de ellas íntimamente adherida (epicardio) y otra que se continúa con la primera y rodea completamente al corazón (pericardio propiamente dicho); entre las dos hojas, existe una cavidad virtual que permite los libres movimientos de la contracción cardíaca.

Recibe el nombre de ciclo cardíaco el conjunto de acontecimientos que tienen lugar desde el inicio del latido cardíaco, y que duran hasta el inicio del latido siguiente. Cada latido comienza con un potencial de acción espontáneo iniciado en una estructura denominada Nódulo Sinusal, situado en la aurícula derecha. Dicho potencial de acción viaja a través de las dos aurículas y del Nódulo Aurículo Ventricular y el Haz de Hiss hacia los ventrículos. En el Nódulo y el Haz se produce un retraso de una décima de segundo, fundamental para que las aurículas se contraigan antes que los ventrículos, llenándolos de esa forma. En la diástole, los ventrículos se llenan de sangre, y se contraen en la sístole, expulsándola hacia los vasos sanguíneos de todo el organismo. [12]

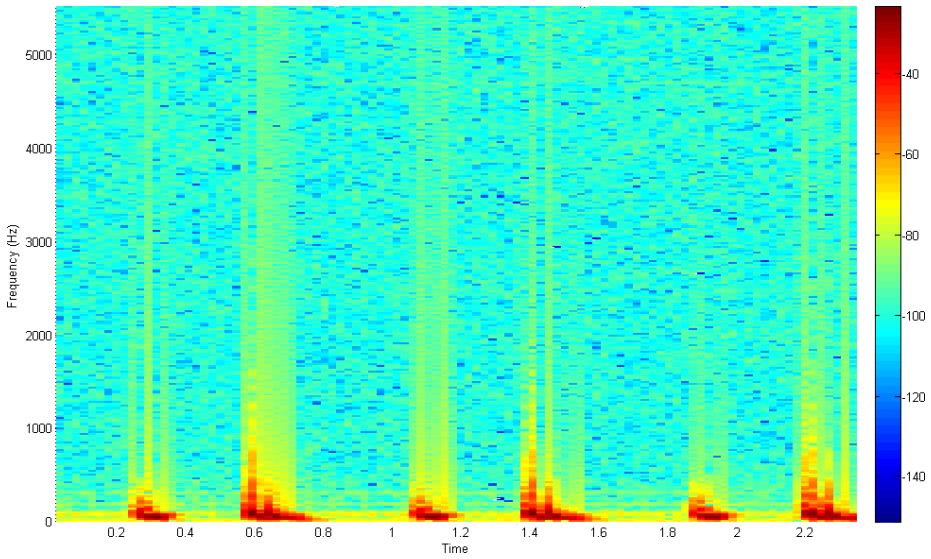


Figura A.2: Espectrograma de una muestra de sonido de un corazón sano

A.1. Fisiológicas

A.1.1. Desdoblamiento del segundo ruido

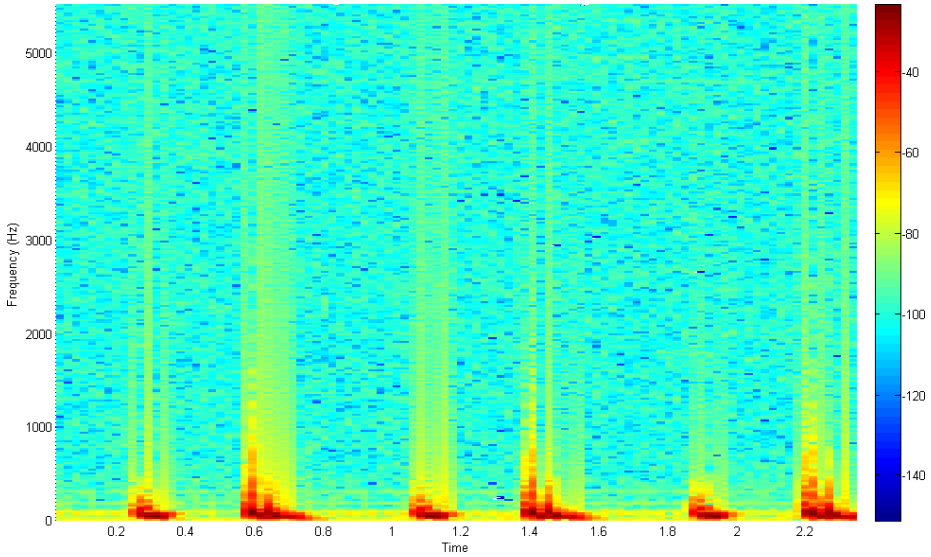


Figura A.3: Espectrograma de una muestra de sonido de un desdoblamiento del segundo ruido

El desdoblamiento del segundo ruido puede ser debido a una causa fisiológica: la respiración. En la mayoría de los adultos sanos, se puede escuchar un desdoblamiento del segundo ruido durante una inspiración profunda. La razón de esto es que el segundo ruido del corazón es realmente la mezcla del cierre de dos válvulas distintas. Normalmente la válvula aórtica se cierra antes que la pulmonar, pero están tan juntas que el sonido suena uniforme e instantáneo. Al inspirar profundamente disminuye la presión intratorácica lo que causa un incremento del retorno venoso, esto causa que la aurícula y ventrículos derechos se llenen un poco más de lo normal y que al ventrículo le cueste un poco más de tiempo expulsar la sangre. Este retraso fuerza a la válvula pulmonar a mantenerse abierta un poco más de lo normal haciendo que la pequeña diferencia anteriormente comentada se haga perceptible. En un paciente sano el sonido irá cambiando entre normal y desdoblado, si se escucha que hay un patrón podría ser un desdoblamiento fijo lo cual no es un estado normal en el corazón.

A.1.2. Soplo funcional

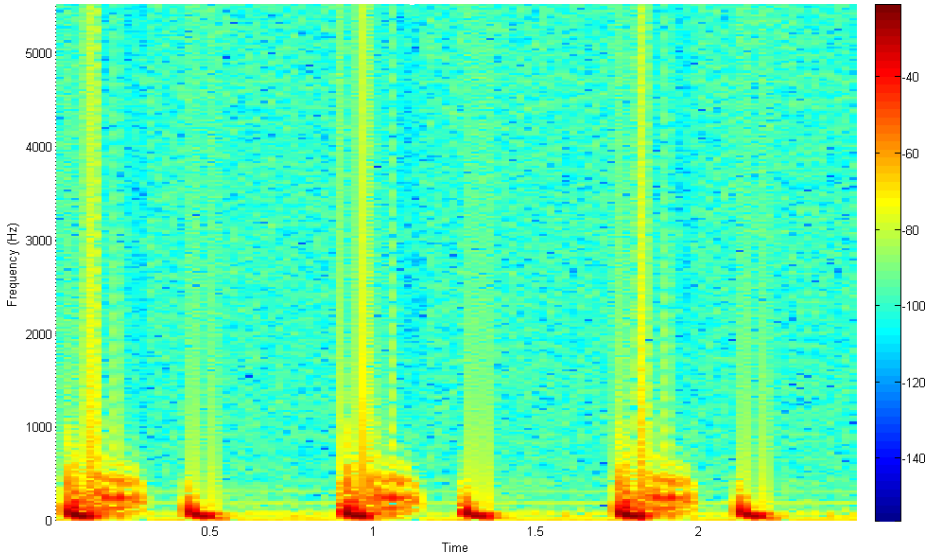


Figura A.4: Espectrograma de una muestra de sonido de un soplo funcional

Un soplo cardíaco se produce cuando hay un flujo de sangre anormalmente turbulento, esto puede ser causado por alguna patología como alguna alteración en una válvula cardíaca por una anomalía intercardiaca o extracardiaca. Se habla de soplo funcional cuando es causado principalmente por condiciones fisiológicas fuera del corazón en vez de en el corazón mismo.

A.1.3. Zumbido venoso

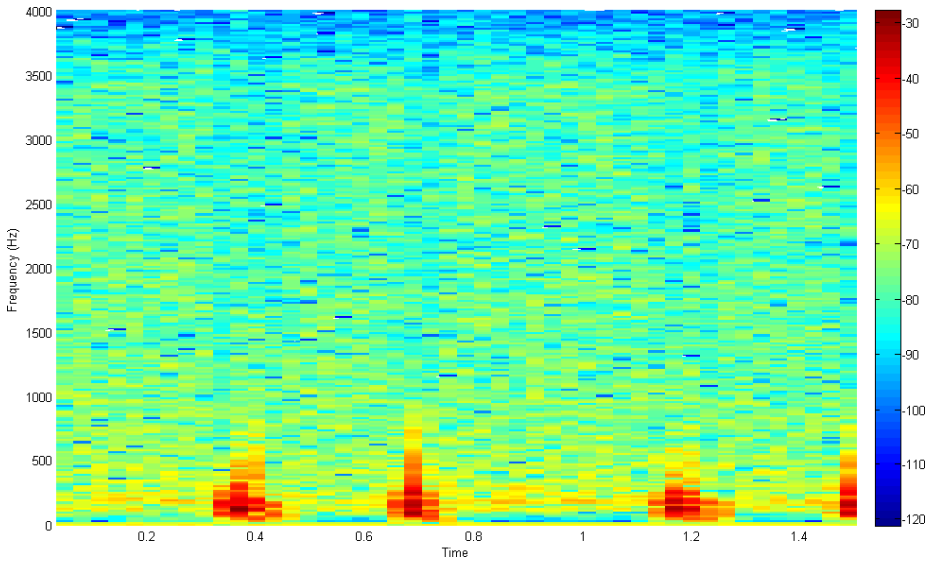
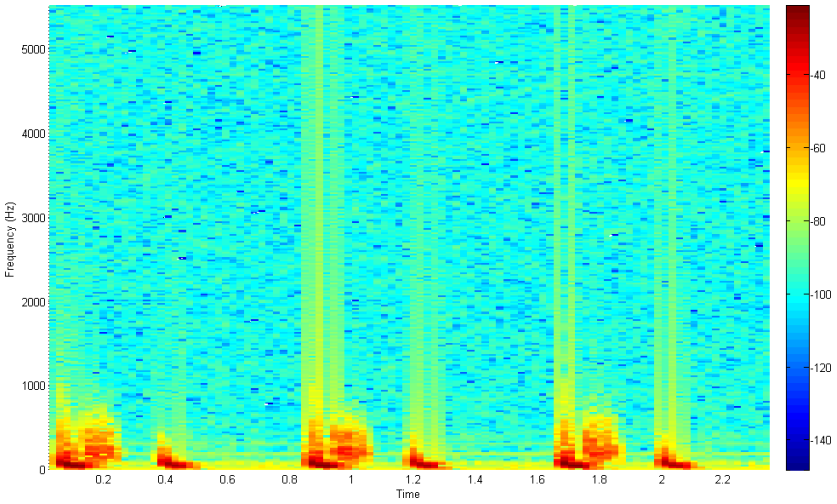


Figura A.5: Espectrograma de una muestra de sonido de un zumbido venoso

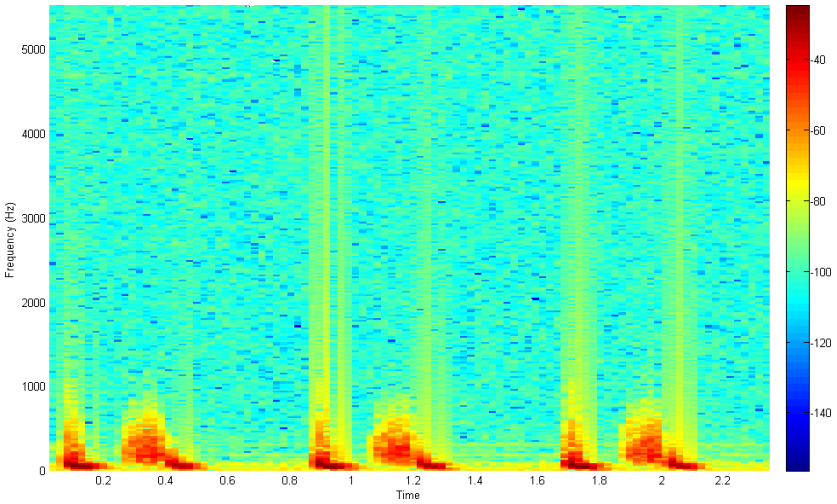
Un zumbido venoso es un soplo continuo benigno. La sangre fluye hacia el cerebro a través de las arterias carótida y vertebral, el retorno lo hace por las venas yugulares. Este flujo puede hacer que las paredes de las venas vibren creando un zumbido.

A.2. Sistólicas

A.2.1. Estenosis aórtica



(a) Estenosis aórtica temprana



(b) Estenosis aórtica tardía

Figura A.6: Espectrogramas de muestras de sonido de estenosis

Uno de los soplos sistólicos patológicos más frecuente es debido a una estenosis aórtica. En esta patología la válvula aórtica no se abre completamente disminuyendo, por lo tanto, el flujo desde el corazón. Conforme la válvula se va estrechando, la presión en el ventrículo izquierdo va aumentando haciendo que éste se vuelva más grueso y que a su vez disminuya el flujo sanguíneo. Conforme la presión sigue incrementándose la sangre puede quedarse en los pulmones habiendo dificultad para respirar.

A.2.2. Prolapso mitral

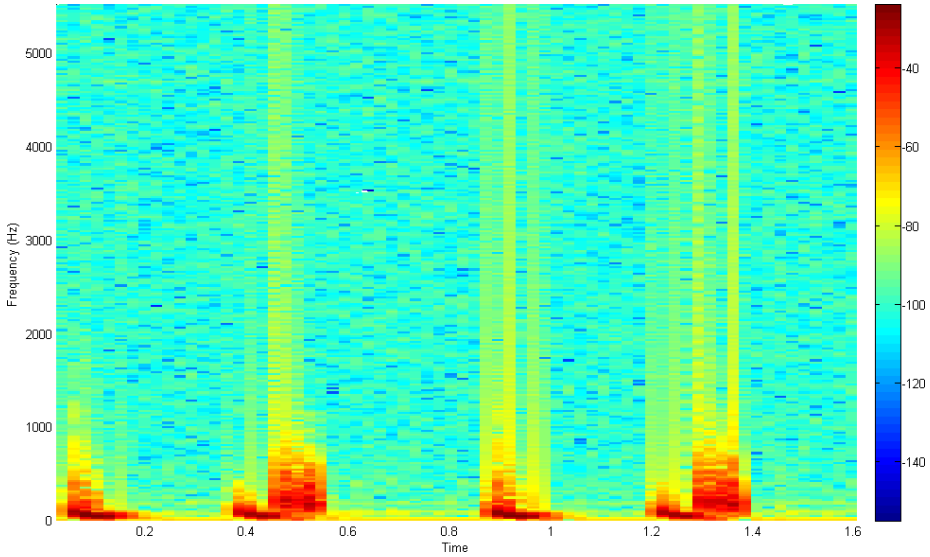


Figura A.7: Espectrograma de una muestra de sonido de un prolapso mitral

El prolapso de la válvula mitral ocurre cuando una de las válvulas del corazón no funciona correctamente. Cuando el ventrículo izquierdo disminuye de tamaño y no se puede mantener la tensión de la válvula mitral puede aparecer un breve tiempo de regurgitación en la aurícula derecha. Estas válvulas defectuosas tienen mayor riesgo de infección bacteriana. En la mayoría de los casos es inofensivo y los pacientes ni siquiera saben que tienen este problema.

A.2.3. Estenosis pulmonar

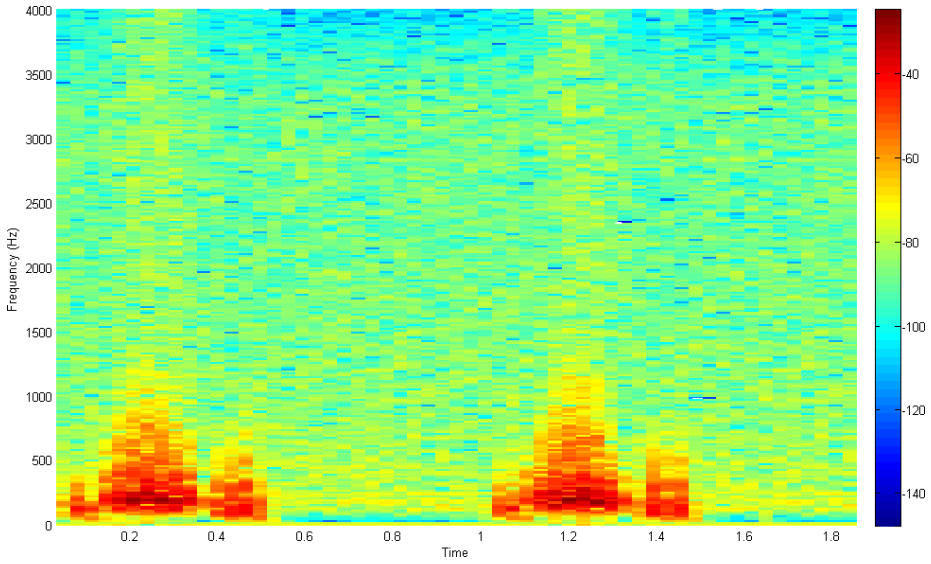


Figura A.8: Espectrograma de una muestra de sonido de una estenosis pulmonar

La estenosis de la válvula pulmonar es el estrechamiento de la válvula que separa el ventrículo derecho de la arteria pulmonar con lo que hay una reducción del flujo de sangre hacia los pulmones y sobrecarga del ventrículo que deja de bombear eficientemente lo que hace que aumente la presión en la aurícula derecha y en las venas que traen la sangre de regreso al lado derecho.

A.2.4. Comunicación interventricular

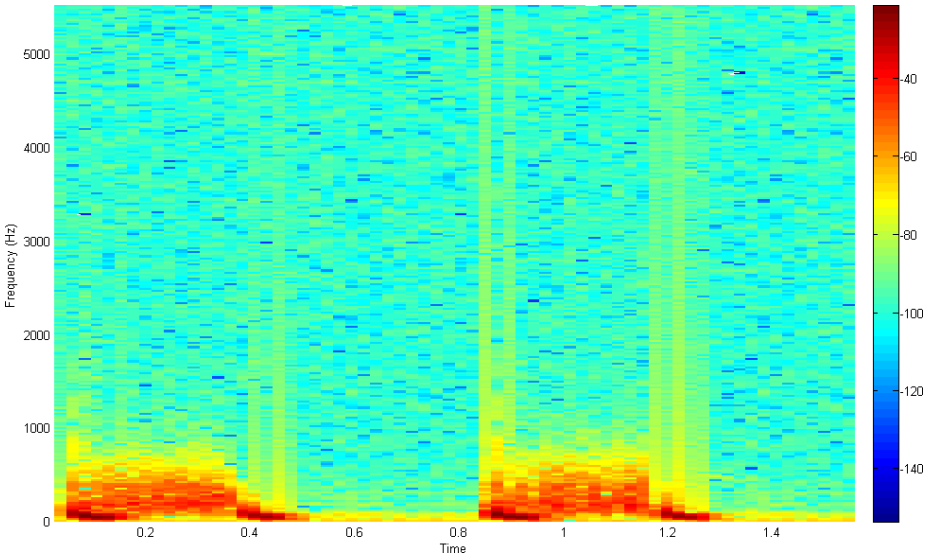


Figura A.9: Espectrograma de una muestra de sonido de una comunicación interventricular

Es uno de los defectos cardíacos congénitos más comunes, hay un cierre incompleto de la pared que separa los dos ventrículos del corazón (tabique interventricular) lo que hace que haya libre comunicación entre ellos. Parte de la sangre es impulsada del ventrículo izquierdo al derecho en vez de circular por el resto del organismo, del ventrículo derecho fluye a los pulmones pudiendo dañar los vasos sanguíneos debido a al volumen extra de sangre que reciben. Al verse obligado a suministrar suficiente sangre el ventrículo izquierdo puede verse obligado a bombear más fuerte y rápido de lo normal.

A.2.5. Comunicación interauricular

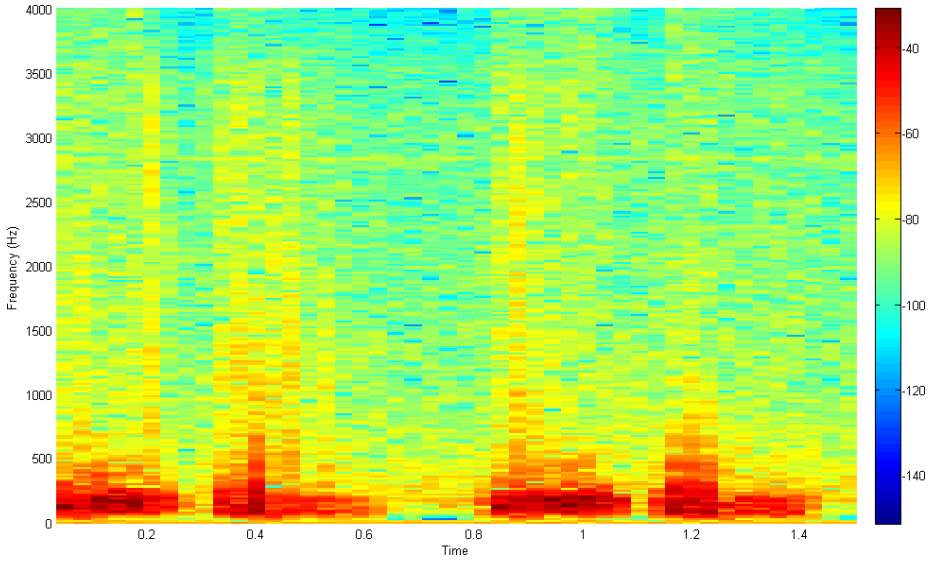


Figura A.10: Espectrograma de una muestra de sonido de una comunicación interauricular

La comunicación interauricular es una deficiencia en la que hay libre comunicación entre la aurícula izquierda y derecha causando una sobrecarga de esta última que repercute en los pulmones y el corazón llegando a ocasionar inversión del flujo sanguíneo.

A.3. Diastólicas

A.3.1. Insuficiencia aórtica

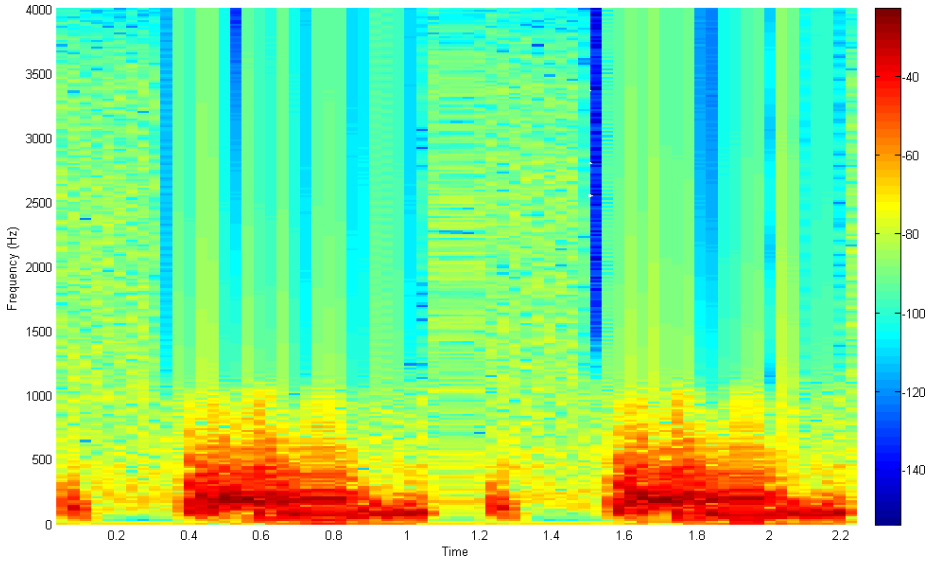


Figura A.11: Espectrograma de una muestra de sonido de una insuficiencia aórtica

En la insuficiencia aórtica la válvula aórtica se debilita impidiendo que esta cierre completamente causando un flujo retrógrado de sangre desde la aorta al ventrículo izquierdo, esto causa que la cámara inferior izquierda del corazón se vaya dilatando dando como resultado una menor capacidad de bombear sangre a la aorta con lo que el corazón trata de compensar este problema enviando mayores cantidades de sangre con cada contracción dando un pulso fuerte y forzado.

A.3.2. Estenosis mitral

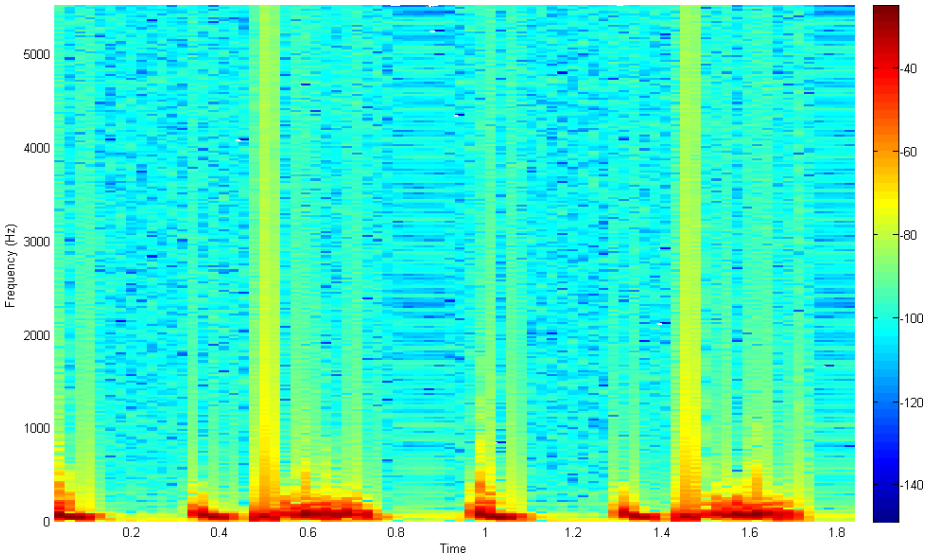


Figura A.12: Espectrograma de una muestra de sonido de una estenosis mitral

La válvula mitral separa las cámaras inferiores y superiores del lado izquierdo del corazón, en una estenosis mitral esta válvula no se abre completamente y restringe el flujo de sangre dando como resultado un menor flujo de sangre hacia el cuerpo y un aumento de presión en la cámara superior llegando a darse un reflujo de fluido hacia los pulmones que puede dar lugar a un edema pulmonar que dificulte la respiración.

A.4. Roces, galopes y soplos continuos

A.4.1. Roces

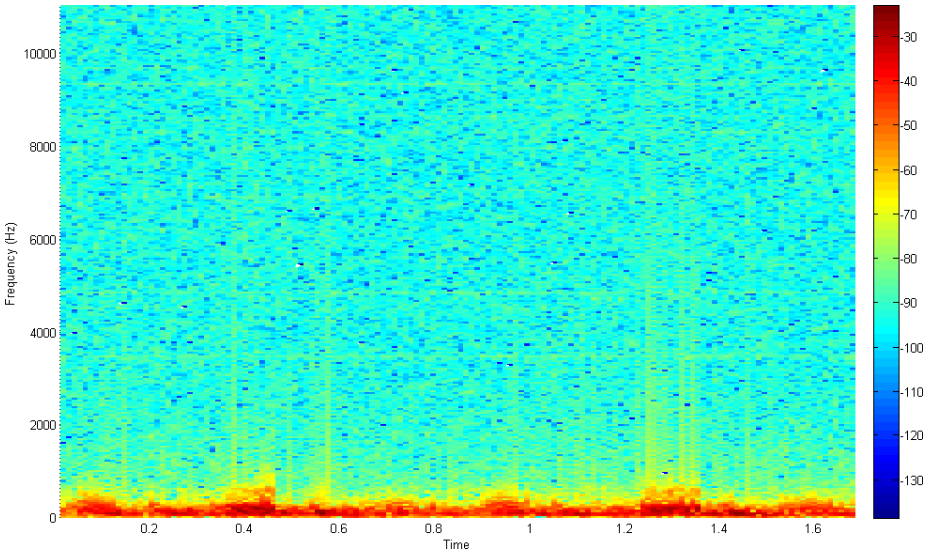


Figura A.13: Espectrograma de una muestra de sonido de un roce pericárdico

Normalmente las paredes interna y externa que componen el pericardio están lubricadas, pero la inflamación del mismo hace que estas paredes rocen una con otra. Es un indicio de presencia de pericarditis (fibrinosa, compresiva...) pudiendo morir por taponamiento cardíaco.

A.4.2. Tercer ruido

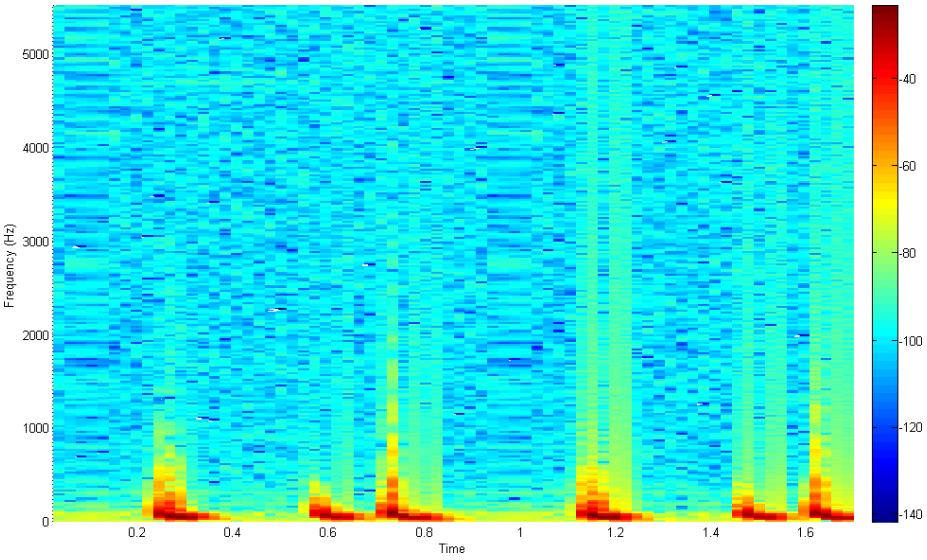


Figura A.14: Espectrograma de una muestra de sonido de un tercer ruido

Como su nombre indica es un sonido que se produce después del primer y segundo sonidos normales en el corazón, ocurre al principio de la diástole. Se cree que está causado por la oscilación de la sangre entre las paredes de los ventrículos iniciada por la entrada de fluido de la aurícula.

A.4.3. Cuarto ruido

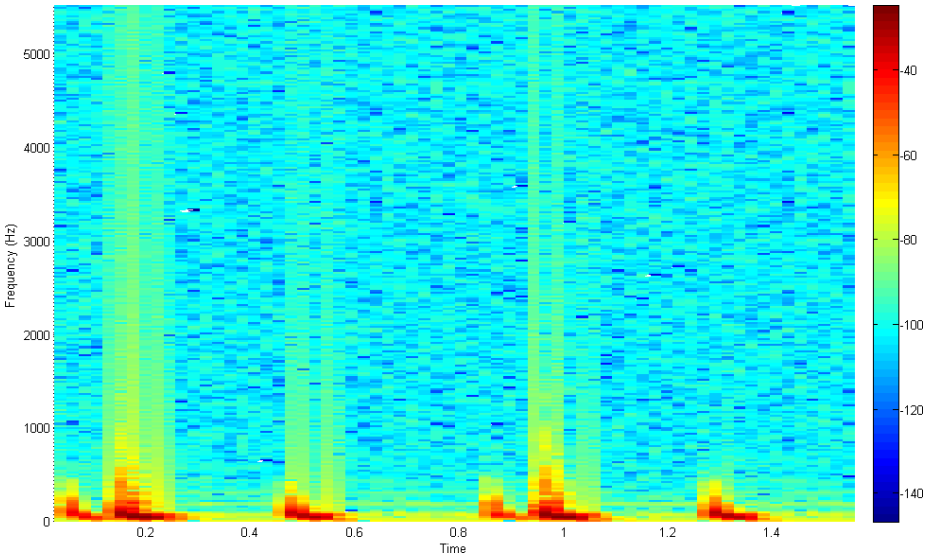


Figura A.15: Espectrograma de una muestra de sonido de un cuarto ruido

Es otro sonido extra que se produce después del primer y segundo sonidos normales en el corazón, por definición ocurre inmediatamente antes del primero mientras la aurícula se está contrayendo. Se piensa que está causado por rigidez en paredes de los ventrículos que causan un flujo anormalmente turbulento cuando la aurícula se contrae forzando a la sangre entrar.

A.4.4. Ductus arterioso persistente

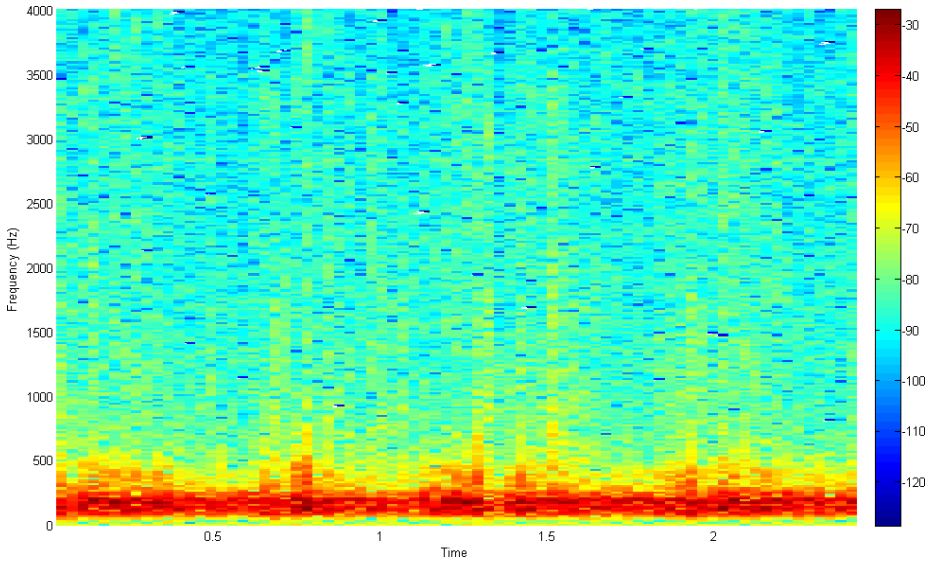


Figura A.16: Espectrograma de una muestra de sonido de un ductus arterioso persistente

Es una enfermedad congénita del corazón donde el Ductus Arteriosus (el conducto situado en la parte inferior de la arteria aorta comunicando con la arteria pulmonar presente durante la etapa fetal y un breve periodo después del nacimiento que previene excesos de presión en los pulmones) no se cierra después del nacimiento, esto permite que una parte de la sangre oxigenada de la parte izquierda del corazón vuelva a los pulmones causando inflamación de los pulmones y dificultades para respirar.

A.5. Pulmones

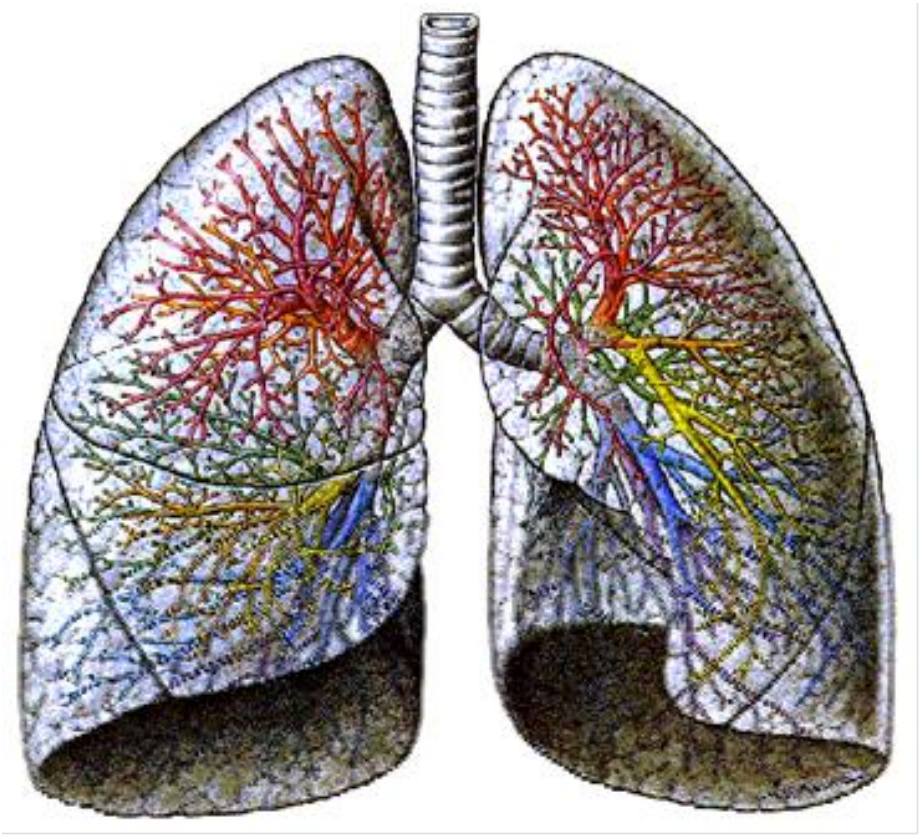


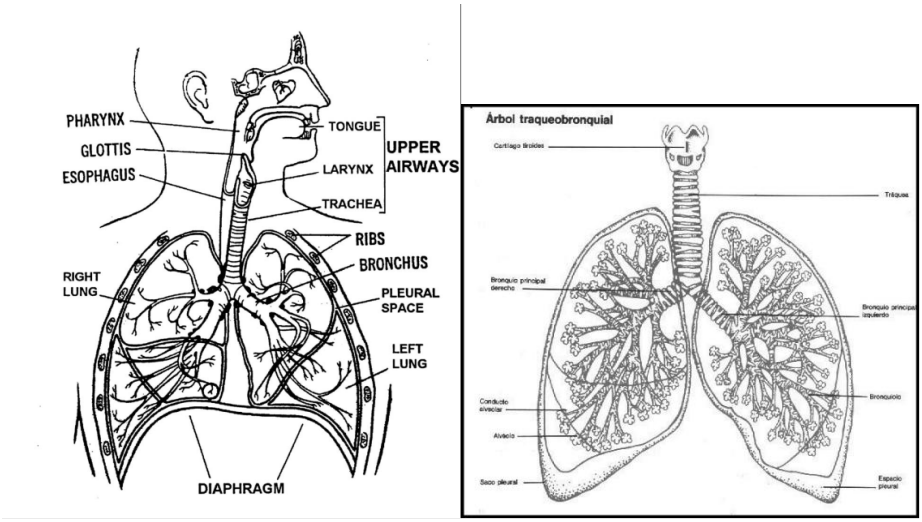
Figura A.17: Pulmones [16]

Cada célula del cuerpo necesita un suministro continuo de oxígeno para producir energía y crecer, repararse y mantener sus funciones vitales. El aparato respiratorio es el encargado de aportar ese oxígeno a los tejidos, y de eliminar el dióxido de carbono.

El sistema respiratorio incluye el diafragma y los músculos torácicos, la nariz y la boca, la faringe y la tráquea, el árbol bronquial y los pulmones, que constituyen el órgano esencial del aparato respiratorio.

El oxígeno contenido en el aire que respiramos, entra al cuerpo a través de la nariz (y/o la boca), atraviesa la faringe, llega a la tráquea que se divide en dos bronquios principales, los cuales llegan cada uno a un pulmón. Los bronquios se ramifican en varias ocasiones formando bronquios más pequeños, que a su vez se vuelven a ramificar formando bronquiolos.. Después de alrededor de 23 divisiones, los bronquiolos terminan en los conductos alveolares, al final de cada cual se encuentran cúmulos de alvéolos (sacos alveolares). En ellos se lleva a cabo el intercambio gaseoso, que consiste en la difusión del O_2 y CO_2 entre la sangre y los alvéolos.

Anatómicamente los pulmones presentan una hendidura profunda dirigida oblicuamente de arriba abajo y de atrás adelante, llamada cisura oblicua, que es



(a) Partes del aparato respiratorio [14] (b) Partes de los pulmones [15]

Figura A.18: Pulmones y aparato respiratorio

única en el pulmón izquierdo, pero que se bifurca en el derecho, formando una segunda cisura llamada cisura horizontal. Estas cisuras dividen los pulmones en lóbulos , de manera que el pulmón izquierdo comprende dos lóbulos (superior e inferior) mientras que el pulmón derecho tiene tres (superior, medio e inferior. Ambos pulmones están recubiertos por dos hojas de pleura: la visceral, íntimamente ligada al parénquima pulmonar, y la parietal, con un espacio virtual entre ellas.

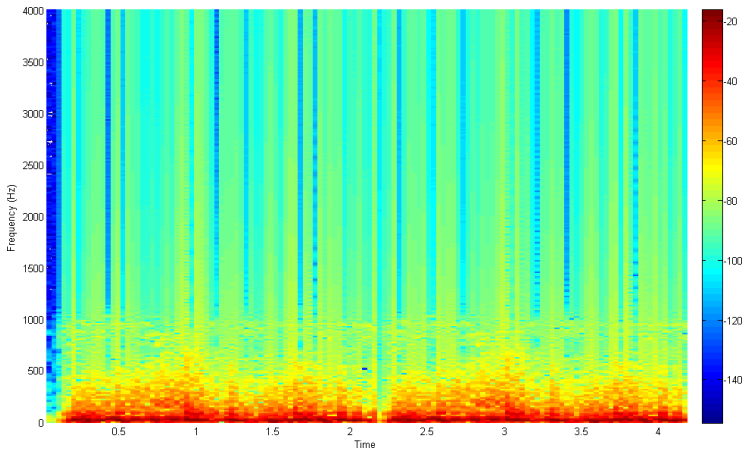


Figura A.19: Espectrograma de una muestra de sonido de respiración normal

A.5.1. Crepitantes

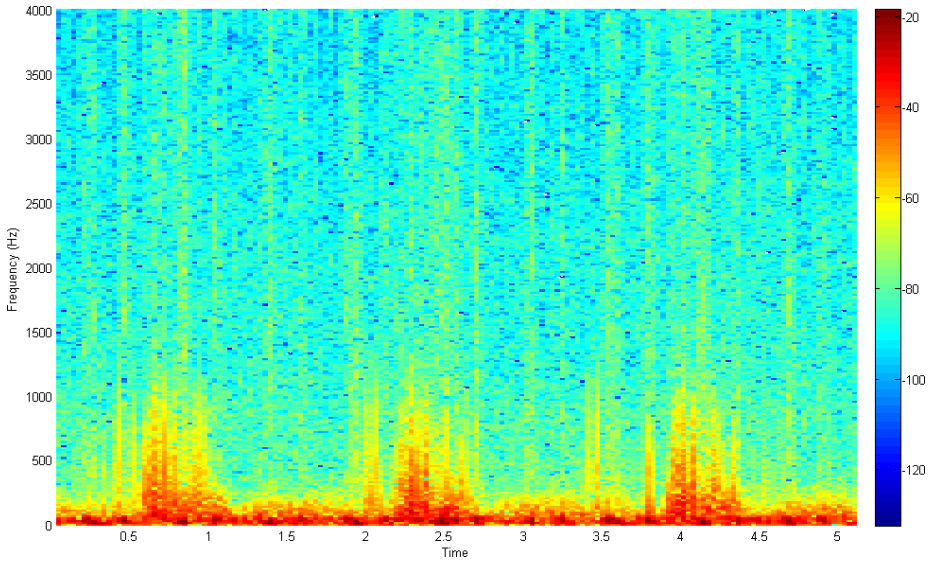


Figura A.20: Espectrograma de una muestra de sonido de crepitaciones

Es crepitante es un sonido que suele deberse a la aparición de secreciones en los bronquiolos o alvéolos. Es un bastante característico ya que se asemejan al sonido de nieve siendo comprimida (al pisarla por ejemplo) o al roce del pelo (frotar un mechón de pelo al lado de la oreja). Una de las causas es un fallo súbito en el lado izquierdo del corazón provoca la acumulación de líquido en los alvéolos produciendo este sonido.

A.5.2. Sibilancias

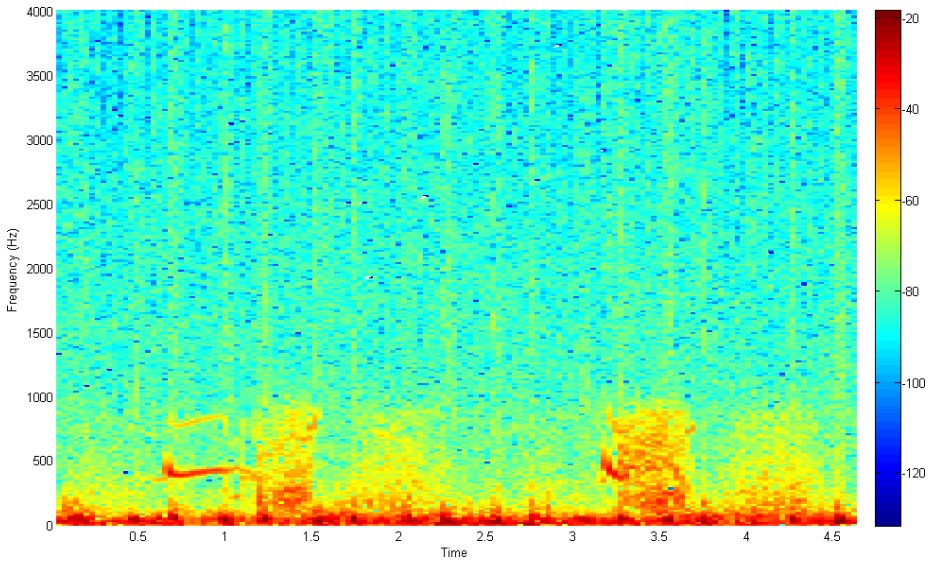


Figura A.21: Espectrograma de una muestra de sonido de sibilancias

Es el sonido que hace el aire al pasar por unas vías respiratorias congestionadas, como un silbido. Son frecuentemente debidas a obstrucciones en los conductos bronquiales torácicos más pequeños aunque también pueden ser de vías mayores o por problemas en las cuerdas vocales.

Apéndice B

Formas de onda con filtros de distinta longitud

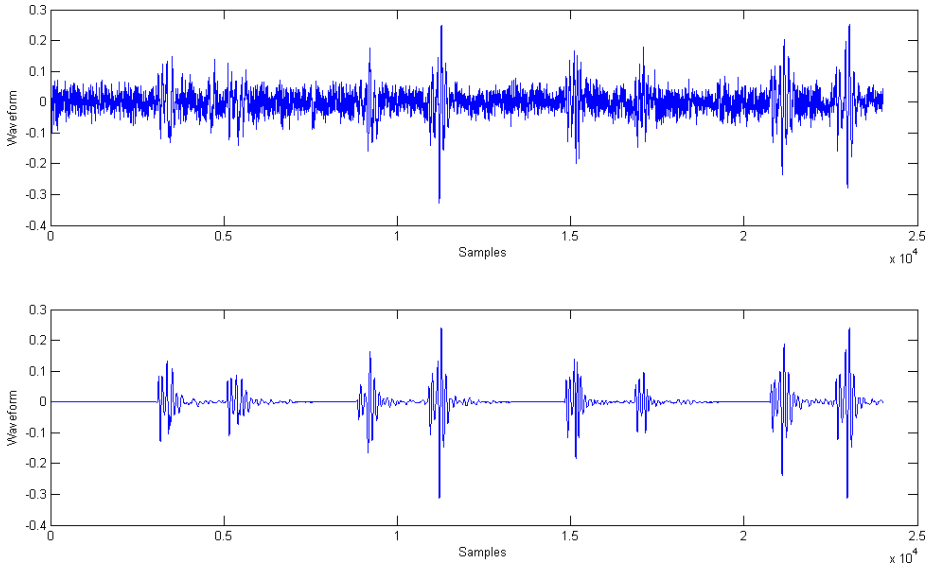


Figura B.1: Filtro de 16 taps

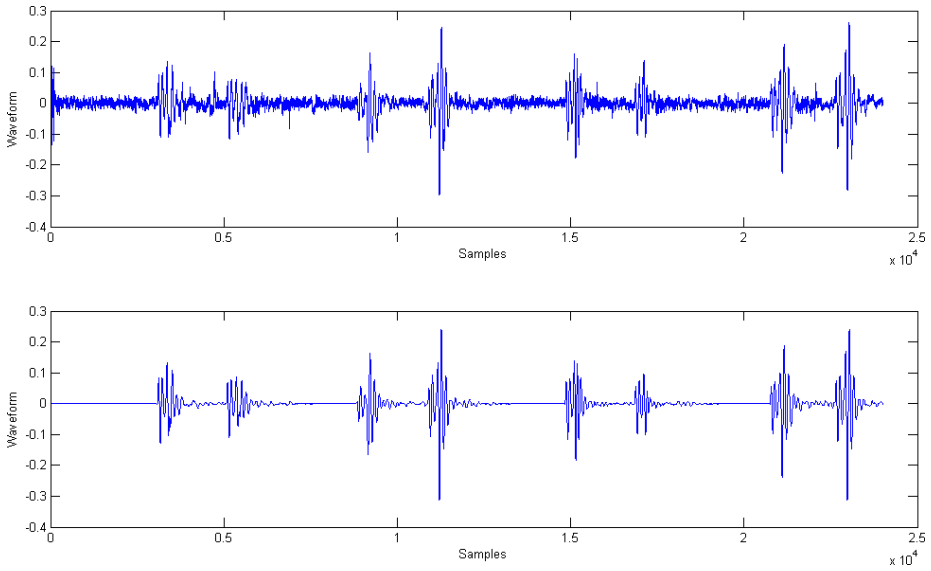


Figura B.2: Filtro de 24 taps

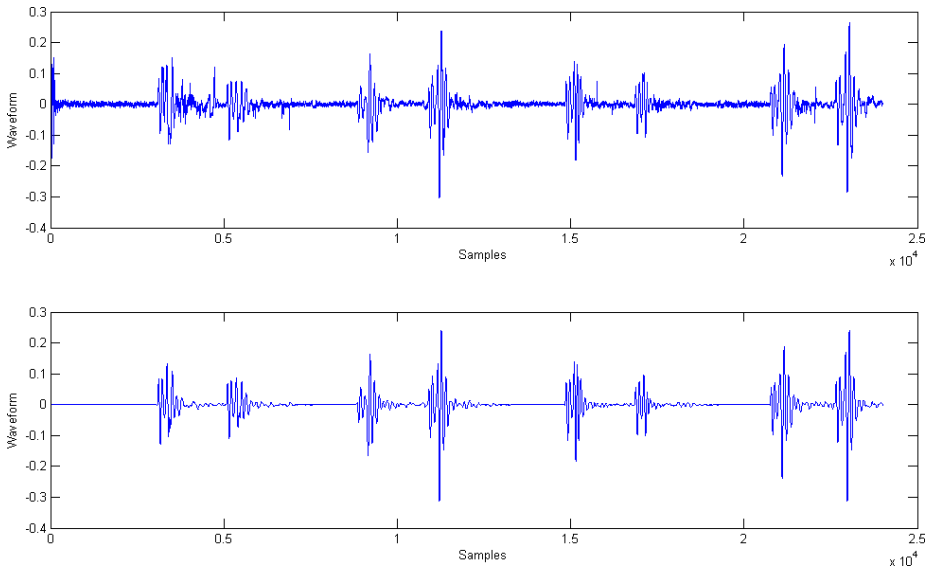


Figura B.3: Filtro de 32 taps

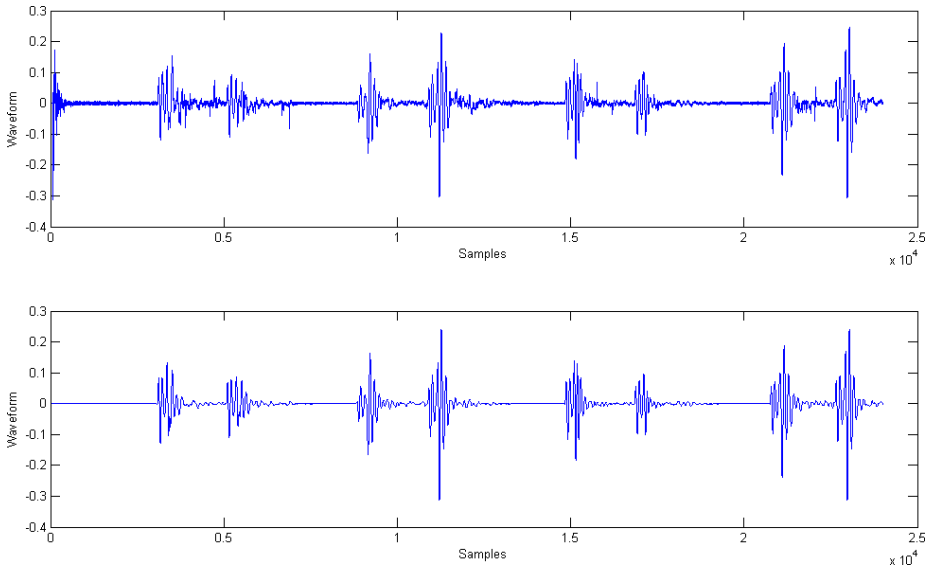


Figura B.4: Filtro de 64 taps

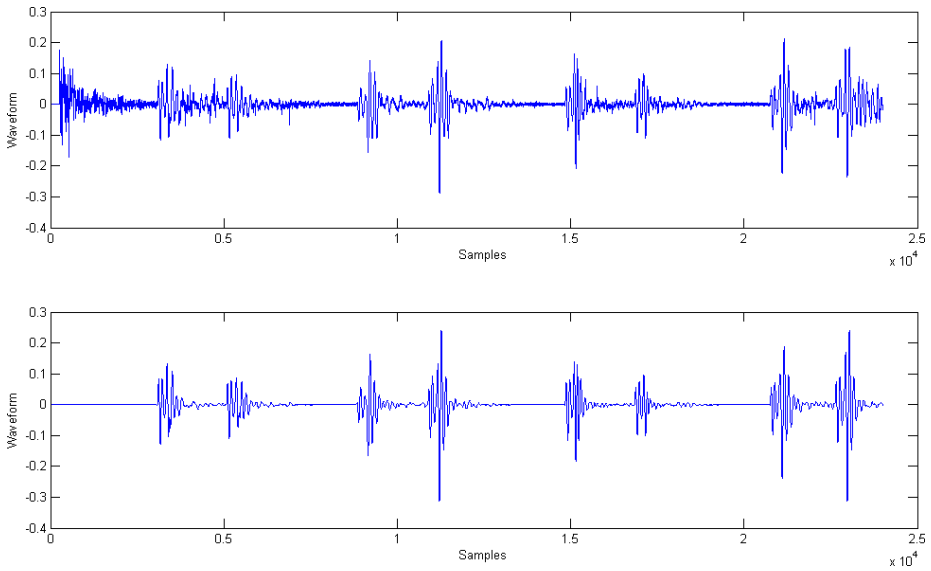


Figura B.5: Filtro de 256 taps

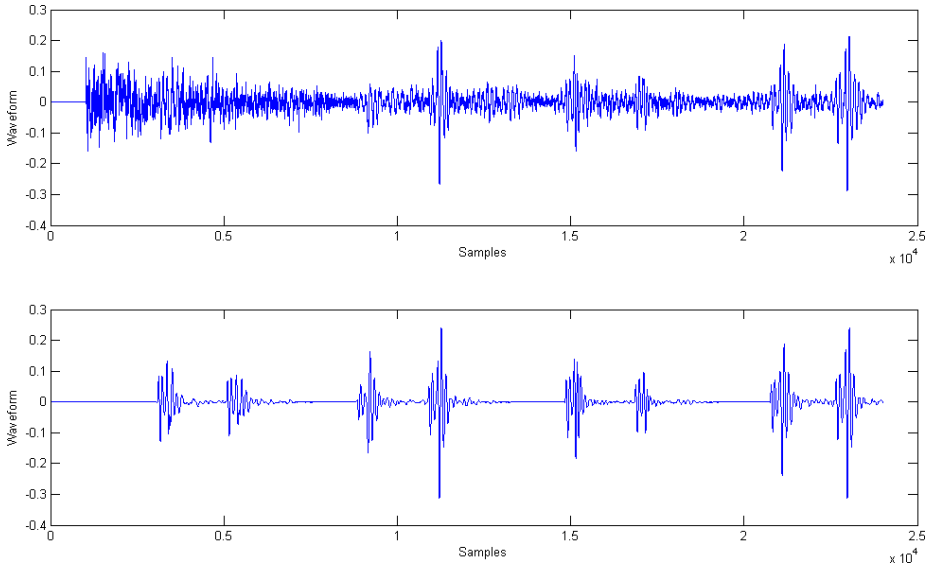


Figura B.6: Filtro de 1024 taps

Apéndice C

Respuestas del filtro con diferentes SNR

C.1. Formas de onda

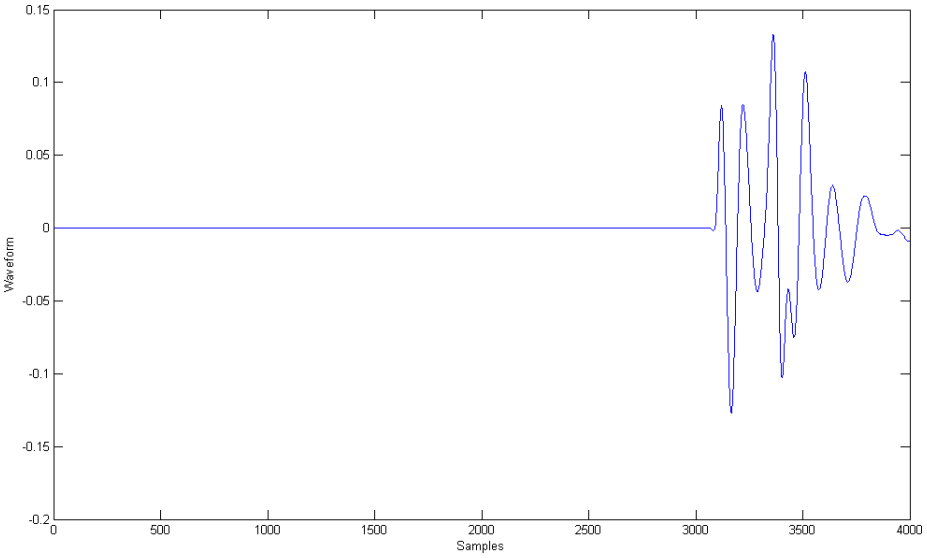


Figura C.1: Sonido del corazón

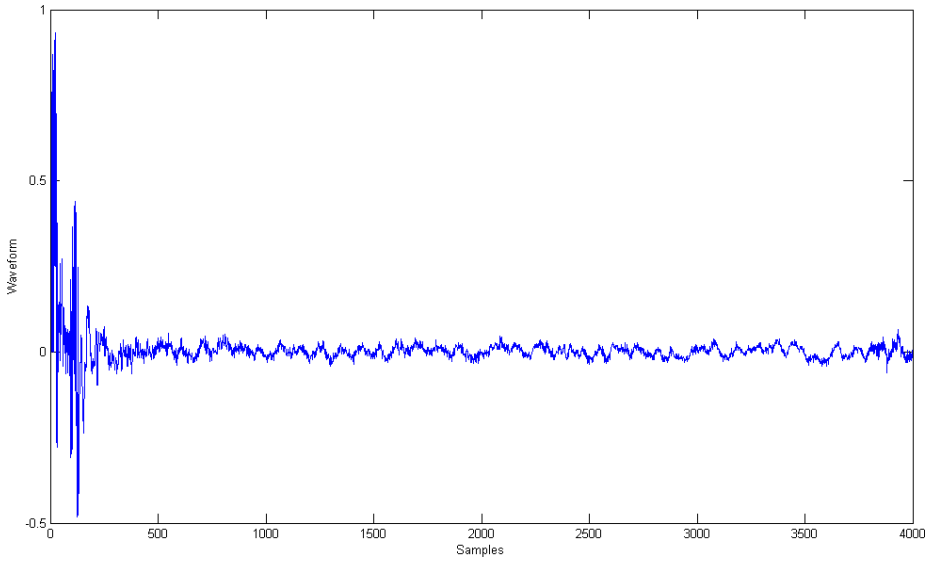


Figura C.2: SNR de -32dB

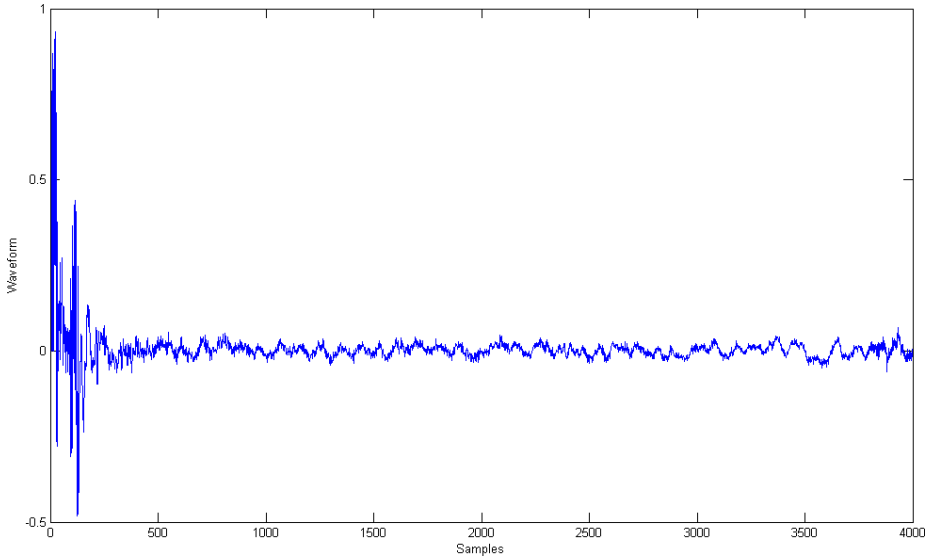


Figura C.3: SNR de -27dB

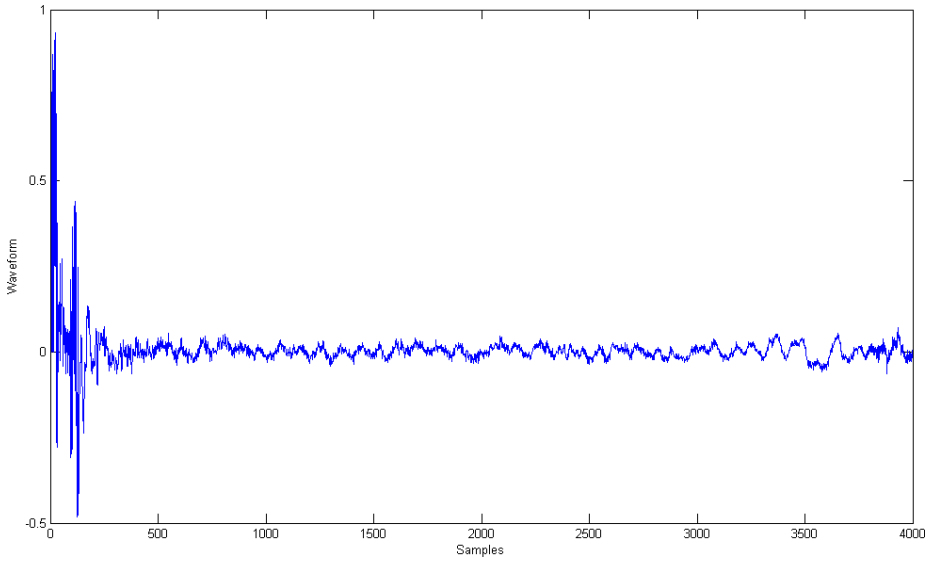


Figura C.4: SNR de -22dB

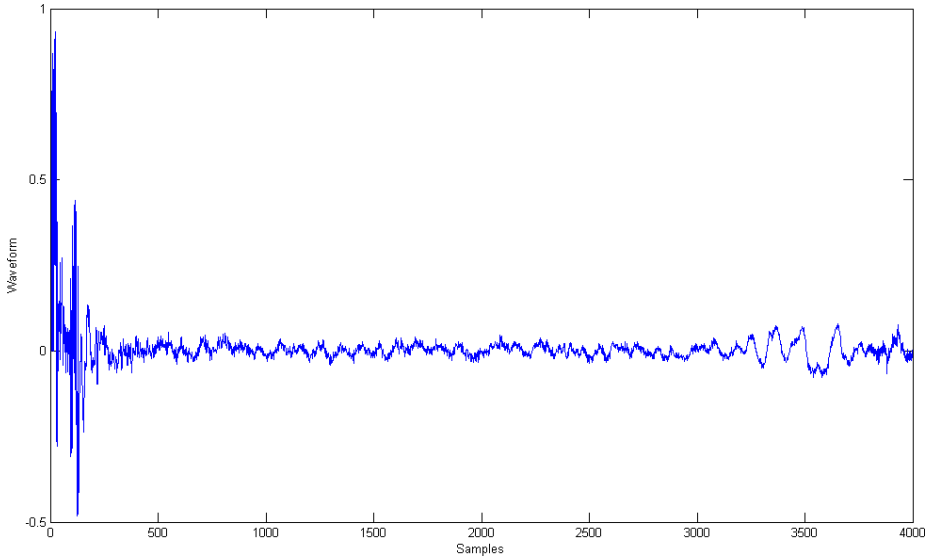


Figura C.5: SNR de -17dB

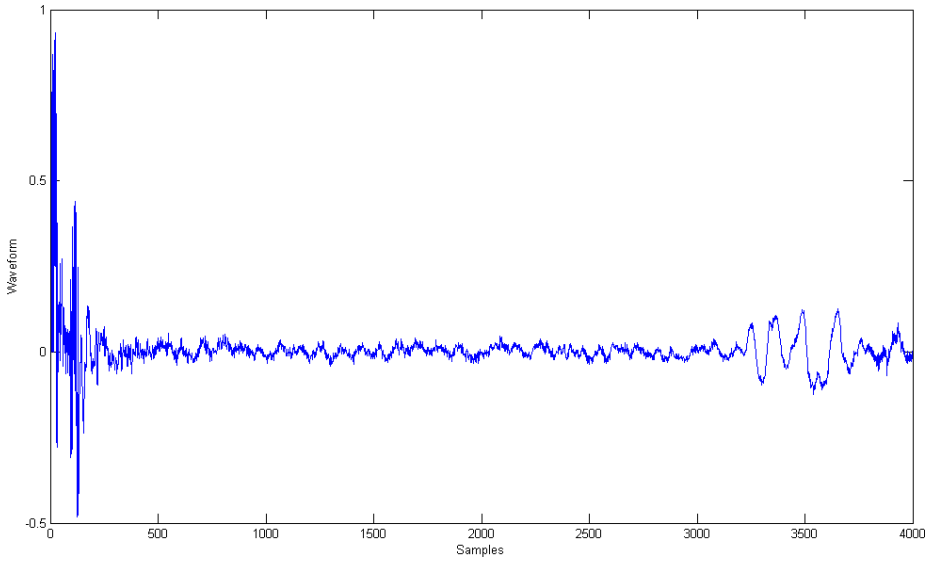


Figura C.6: SNR de -12dB

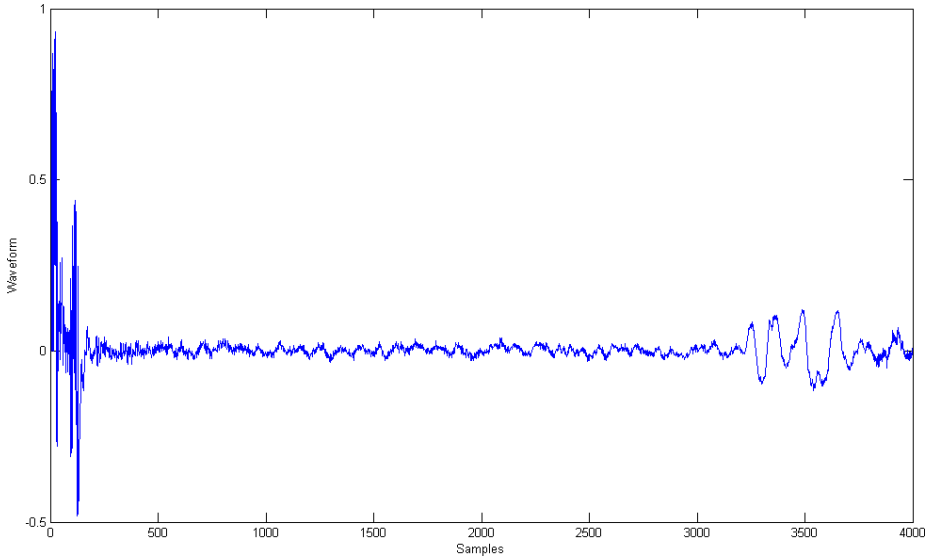


Figura C.7: SNR de -7dB

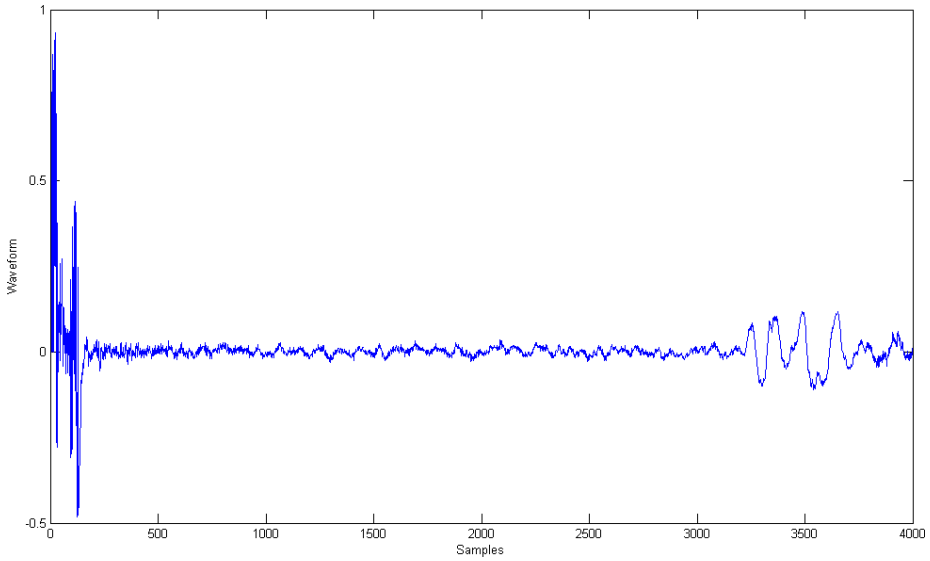


Figura C.8: SNR de -2dB

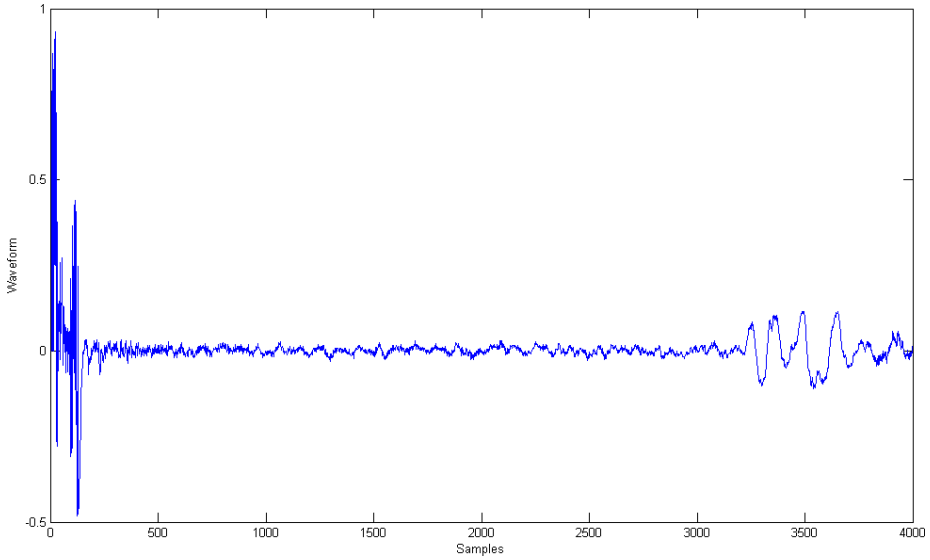


Figura C.9: SNR de 3dB

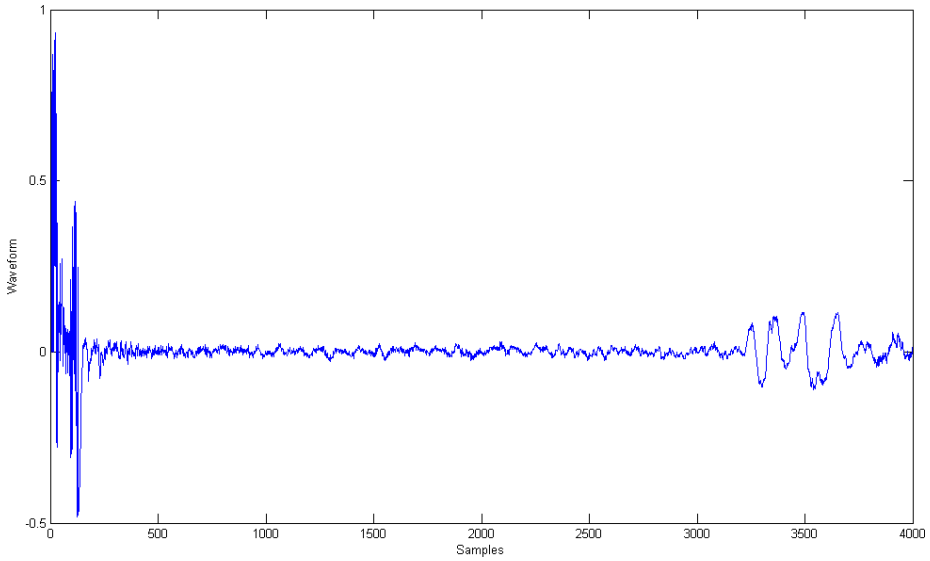


Figura C.10: SNR de 8dB

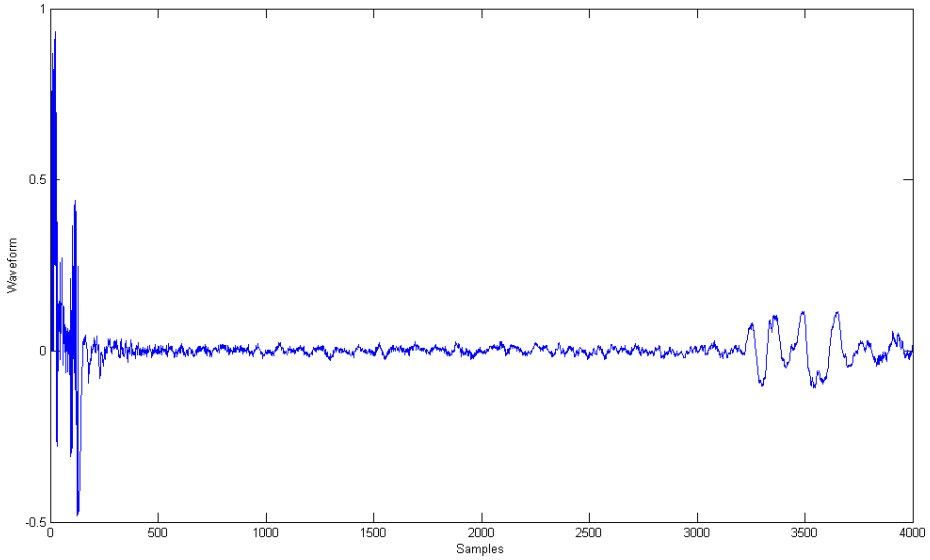


Figura C.11: SNR de 13dB

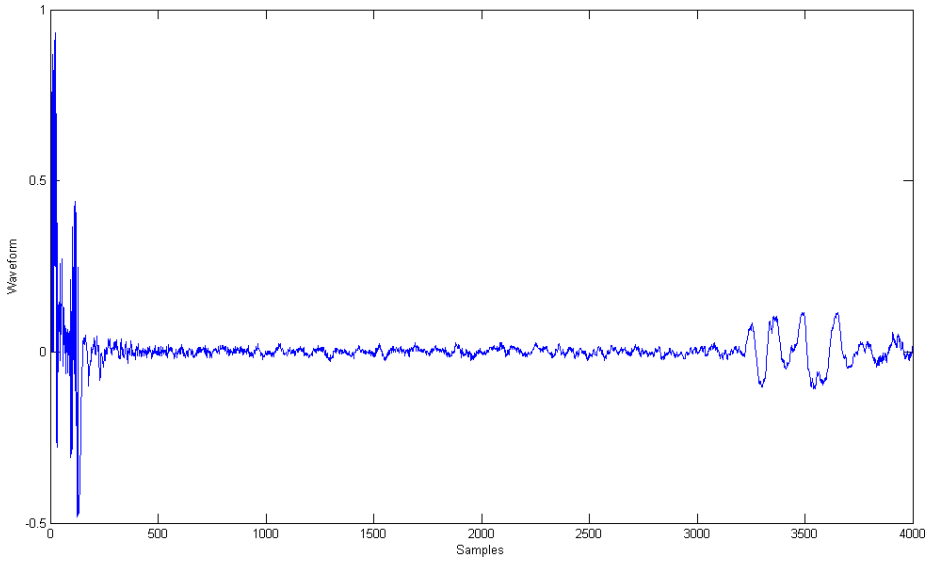


Figura C.12: SNR de 18dB

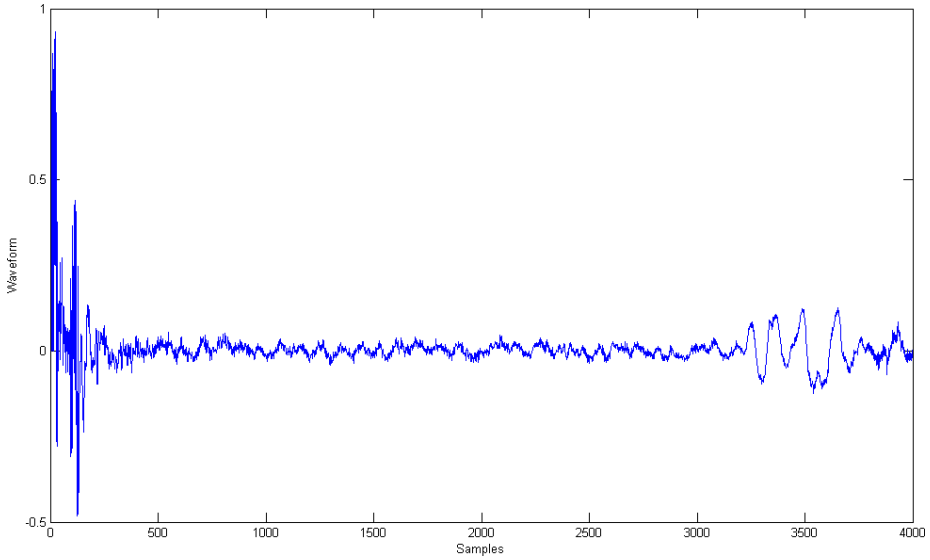


Figura C.13: SNR de 23dB

C.2. Espectrogramas

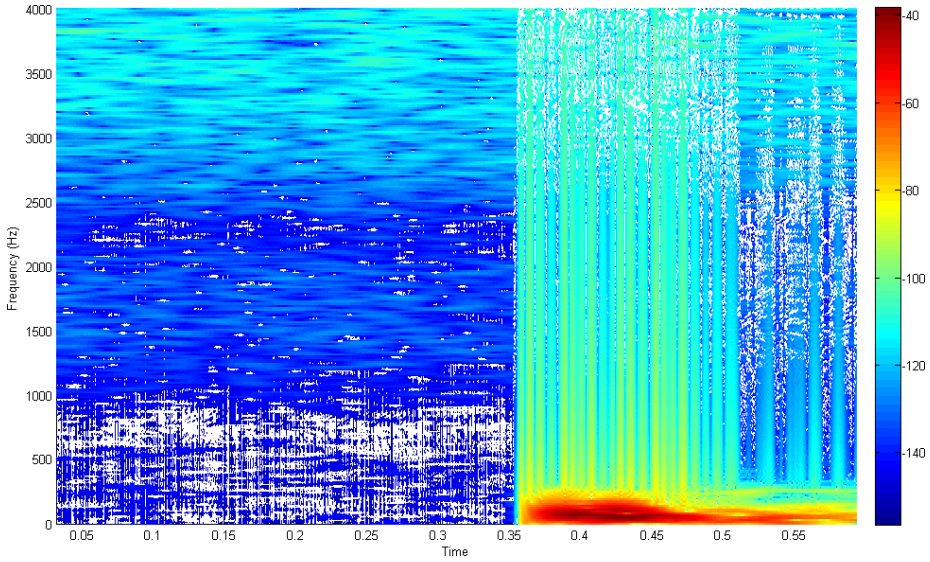


Figura C.14: Sólo corazón

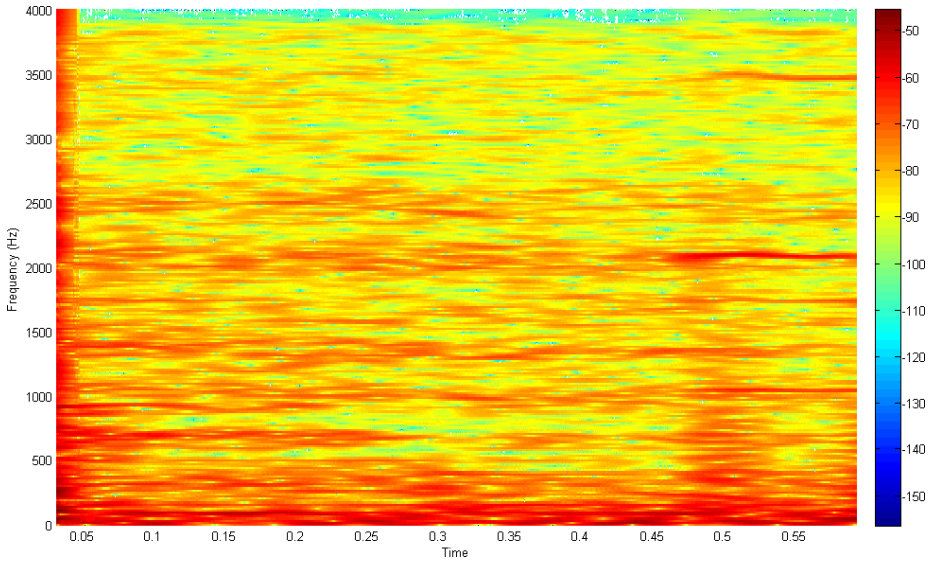


Figura C.15: SNR de -32dB

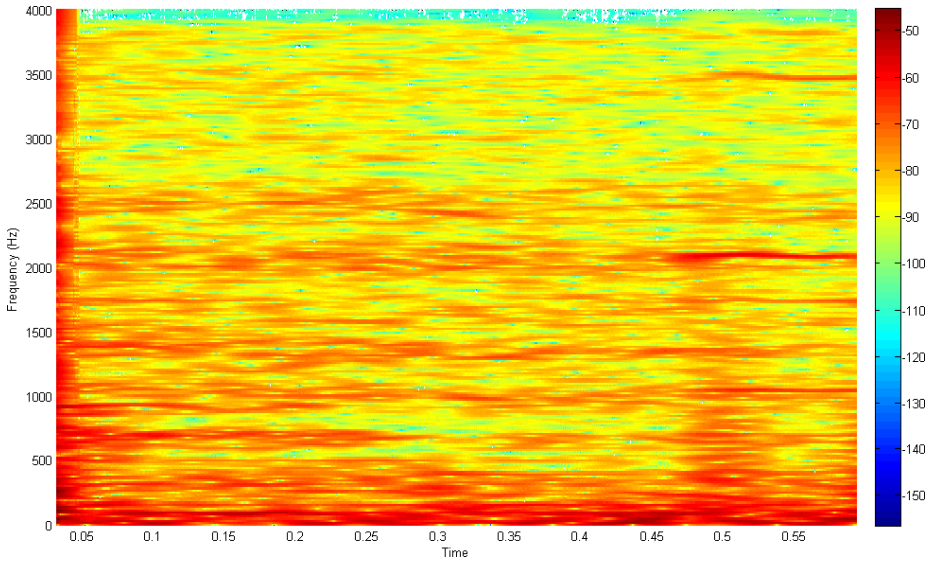


Figura C.16: SNR de -27dB

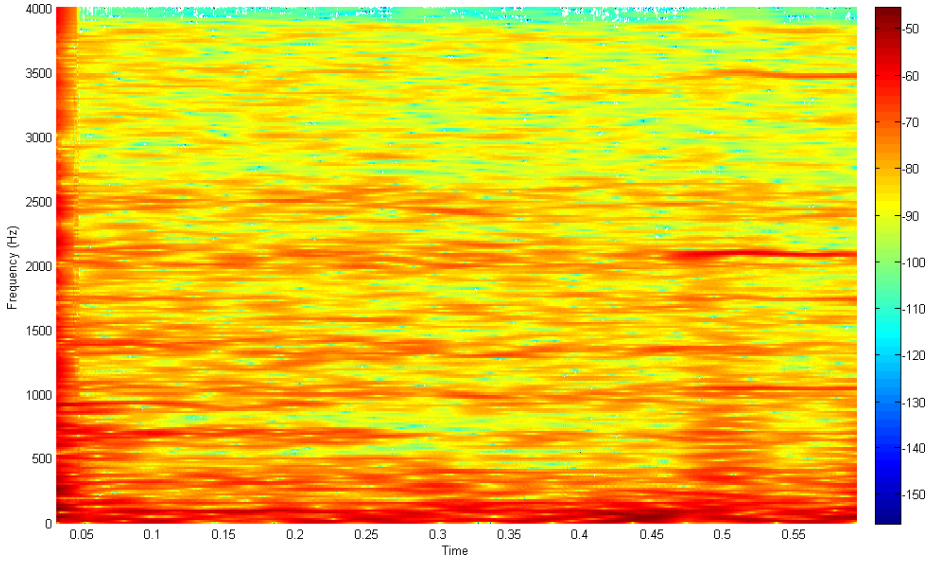


Figura C.17: SNR de -22dB

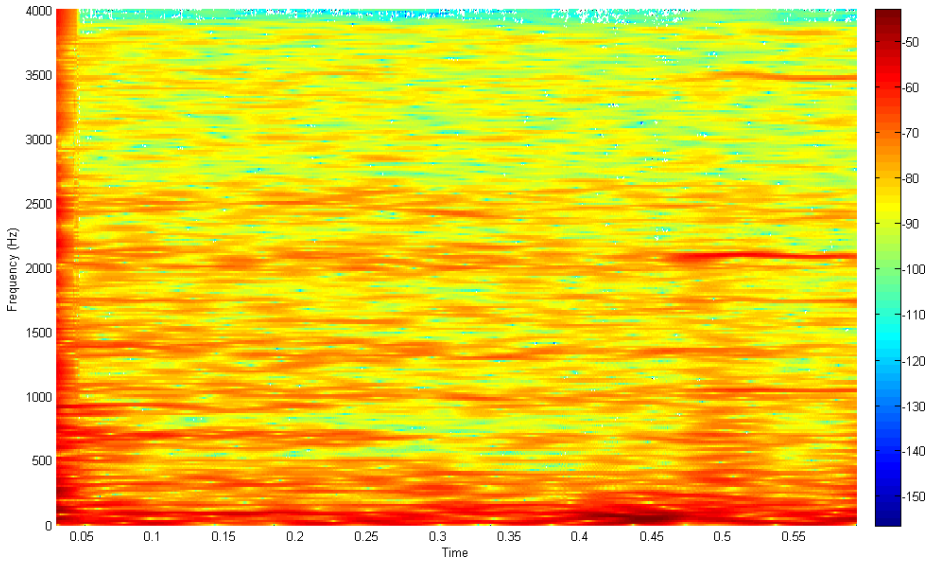


Figura C.18: SNR de -17dB

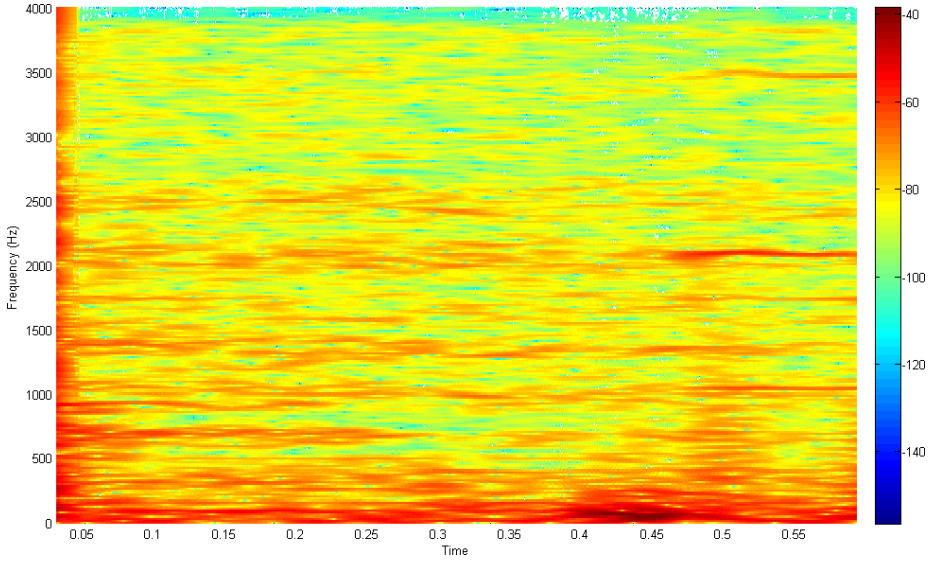


Figura C.19: SNR de -12dB

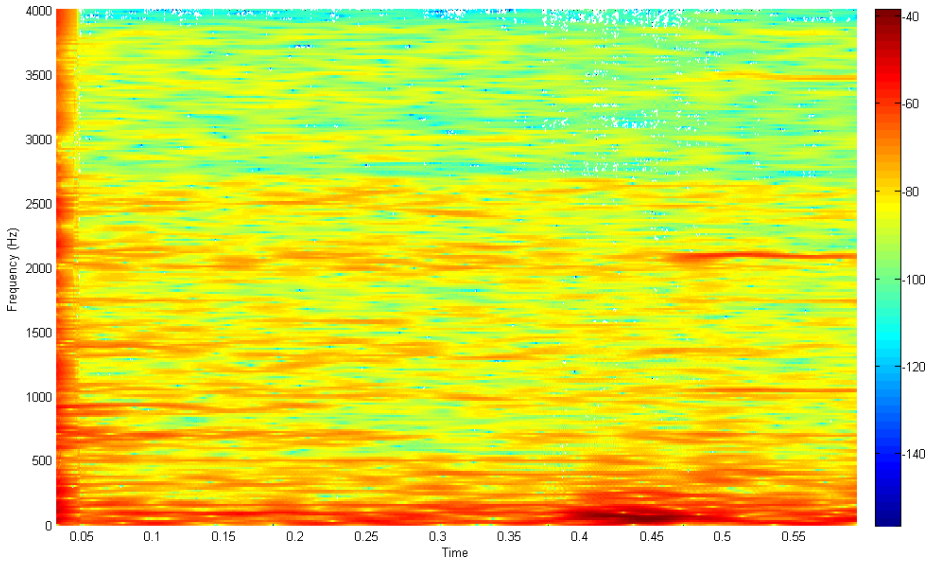


Figura C.20: SNR de -7dB

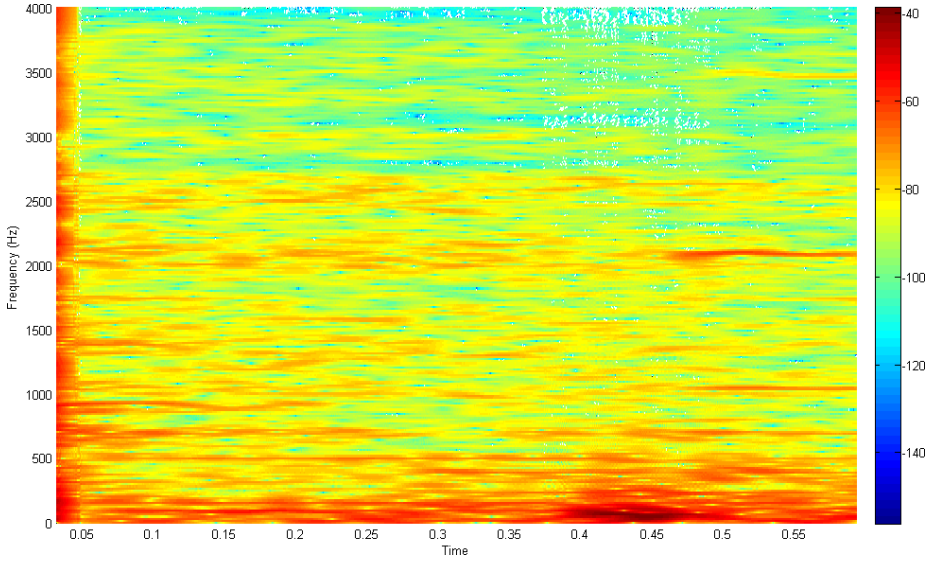


Figura C.21: SNR de -2dB

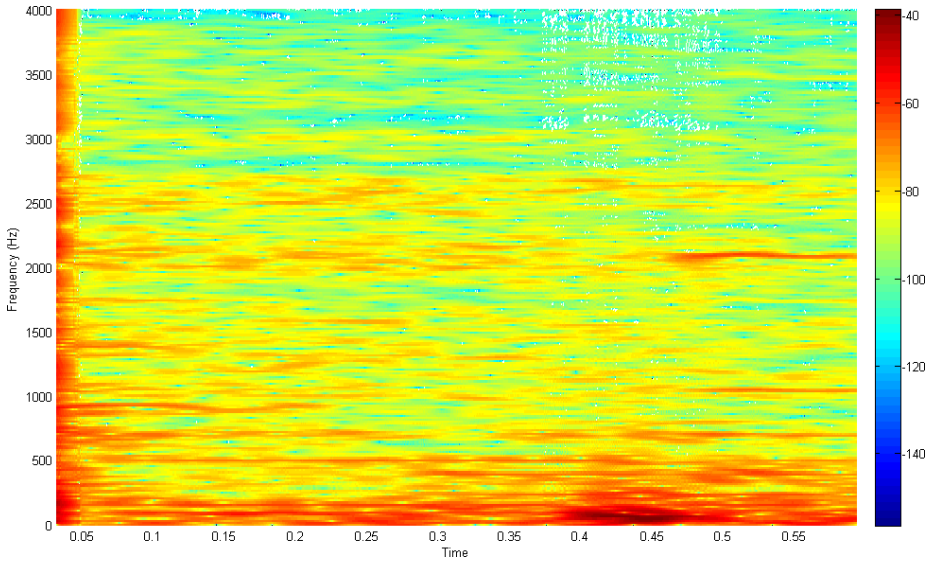


Figura C.22: SNR de 3dB

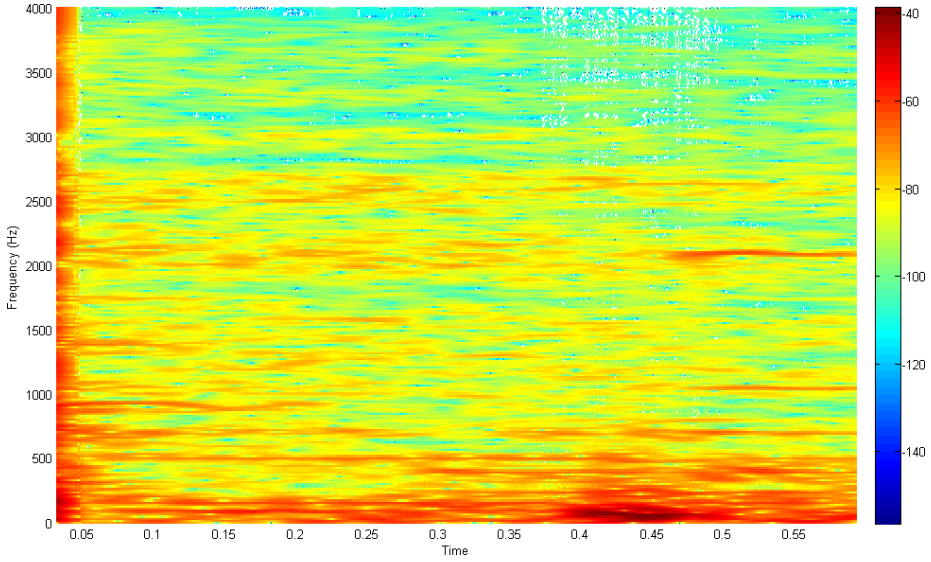


Figura C.23: SNR de 8dB

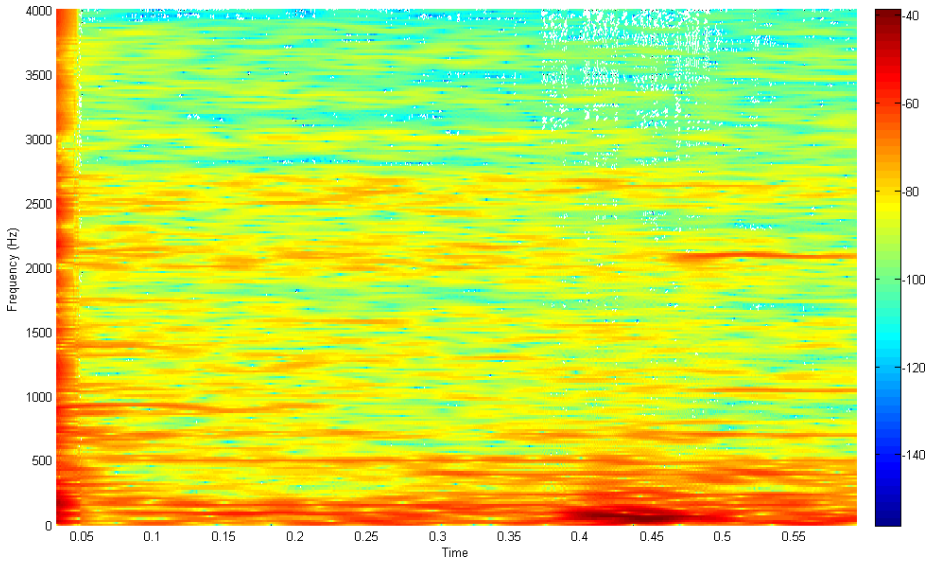


Figura C.24: SNR de 13dB

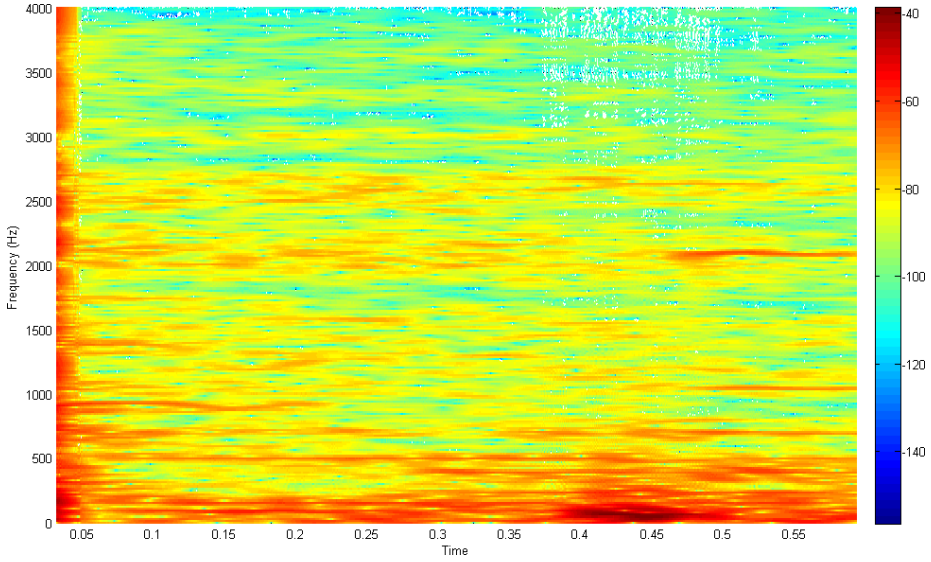


Figura C.25: SNR de 18dB

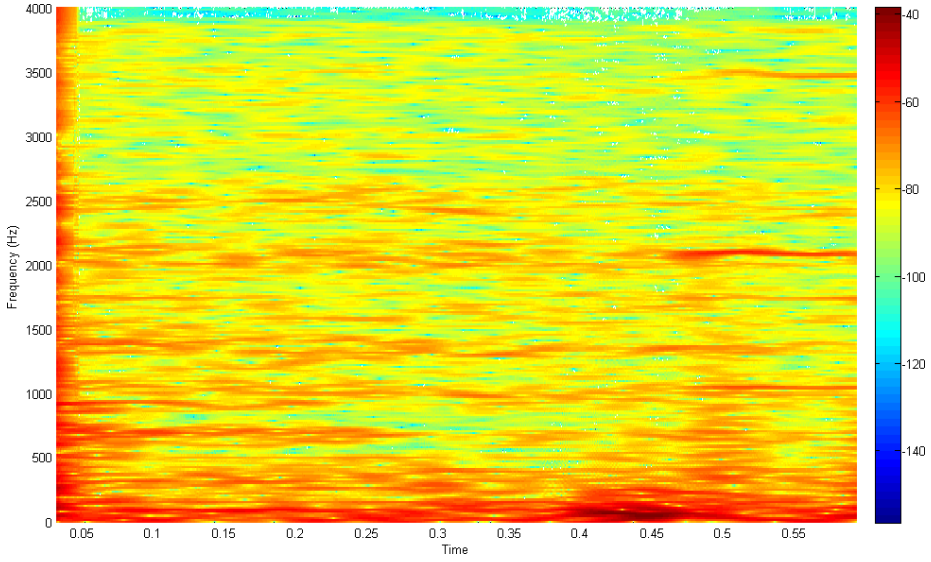


Figura C.26: SNR de 23dB

Apéndice D

Código

D.1. Prototipo Matlab

```
01 clear all
02 close all
03
04 [dirty,fs] = wavread('sound_stethoscope.wav');
05 [noise,fs] = wavread('sound_ambient.wav');
06
07 d = dirty;
08
09 M = 1024; %Filter Order
10 mu1 = 0.5; %Step sizes
11 mu2 = 0.1;
12 mu3 = 0.01;
13
14 N = length(dirty);
15
16 w = zeros ( M  , 1 ) ;
17
18 for n=M:N, % Go through the whole vector
19
20  if n < 2*M,      %Changes of mu
21         mu=mu1;
22     end;
23     if n > 10*M,
24  mu=mu2;
25     end;
26     if n > 100*M,
27  mu=mu3;
28
29  end;
30     u = noise(n:-1:n-M+1);  %Read in opposite way, -1 is direction.
31
```

```
32      yn = w' * u; %Apply filter
33      e(n) = d(n) - yn;  %Obtain error = output
34
35      p = (u'*u); %Calculate power
36      w = w + (mu * u * e(n))/(p); %Normalized LMS, more stable
37  end
38
39  soundsc(e,fs);  %Play sound
```

D.2. Software en C

D.2.1. Coma flotante

```
01     #include "stdio.h"
02     #include "math.h"
03     #include "string.h"
04     #include "stdlib.h"
05
06     #define ORDER 24
07     #define STEP1 0.5
08     #define STEP2 0.1
09     #define STEP3 0.01
10
11     int main()
12     {
13          short int readInputAmbience, readInputBody, writeOutput;
14          double calcAmbience, calcBody;
15          double mu = STEP1;
16          double w [ORDER];
17          double u [ORDER];
18          double power;
19          double e;
20          double yn;
21          int n1,n2,n3;
22          int i;
23          short int bs, Bs;
24          FILE *noiseFile, *dirtyFile, *outputFile;
25
26   noiseFile = fopen("sound_ambient.wav","rb");
27   dirtyFile = fopen("sound_body.wav","rb");
28   outputFile = fopen("output_sound.wav","wb");
29
30 //The two files need to have the same header
31
32   char buffer [44];
33   fread(buffer, 1, 44, noiseFile); //Duplicating header
34   fwrite(buffer, 1, 44, outputFile);
35
36   fseek(dirtyFile, 34, SEEK_SET);
37   fread(&bs, 2, 1, dirtyFile);  //Bits per sample
38   Bs = bs / 8;        //Bytes per sample
39
40          fseek(noiseFile,44,SEEK_SET);
41
42   for (i = 0; i<ORDER;i++){w[i]=0;}; //Inicialize weights
43
44          n1=0;
45          n2=0;
```

```
46
47          while (!feof(dirtyFile)) {
48
49      starting:
50      fread(&readInputBody, Bs, 1, dirtyFile);   //Read one sample
of the file
51      fread(&readInputAmbience, Bs, 1, noiseFile);
52
53      calcBody = readInputBody;
54      calcBody /= (1<<bs);
55
56      calcAmbience=readInputAmbience;
57      calcAmbience /= (1<<bs);
58
59      while (n1<ORDER-1) {
60       fwrite(&readInputBody, Bs , 1, outputFile);
61       u[ORDER-n1-2] = calcAmbience;
62       n1++;
63       goto starting;   //Wait until we have enough data to start the
filter
64      }
65
66              if (n2 < 2*ORDER) {    //Changing mu
67       n2++;
68      }
69
70              if (n2 < 10*ORDER && n2>2*ORDER)  {
71                 mu = STEP2;
72                  n2++;
73              }
74
75              if (n2 == 10*ORDER) {
76                  mu = STEP3;
77                  n2++;
78              }
79
80              for (i=ORDER-1; i>0;i--){
81       u[i] = u[i-1];
82      }
83
84              u[0]=calcAmbience;
85
86              yn = 0;                 // Apply filter
87             for (i = 0; i<ORDER;i++){
88        yn += w[i]*u[i];
89      }
90
91              e = calcBody - yn;      // Error, output
92
93              if (e > 1) {e = 1;};  //If over my working range, saturing.
```

```
94    if (e < -1) {e = -1;};
95
96              power = 0;      //Calculate power for NLMS
97              for (i = 0; i<ORDER;i++){
98     power += u[i]*u[i];
99    }
100
101              for (i = 0; i<ORDER;i++){   // Update of the weights
102     w[i] = w[i] + e*mu*u[i]/power;
103    }
104
105    e *= (1<<16);
106    writeOutput = e;
107
108    fwrite(&writeOutput, Bs, 1, outputFile); // Write the output
in the file
109
110         }
111
112   fcloseall();
113
114         return 0;
115     }
```

D.2.2. Coma fija

```
01     #include "stdio.h"
02
03     #define RESOLUTION 15
04   #define MAX_POS ((1<<RESOLUTION)-1)
05   #define MAX_NEG (-(1<<RESOLUTION))
06
07     #define ORDER 24
08     #define MU 0.01*MAX_POS
09
10   typedef short int QRESOLUTION;  //Define a new type to work in
fixed point
11
12     QRESOLUTION AddFix(QRESOLUTION a, QRESOLUTION b) {
13    long int c;
14    c = (int)a+(int)b;          //Store the result of the operation
in a bigger variable
15    if (c>=MAX_POS) c=MAX_POS;  //If out of limits, saturate
16    if (c<=MAX_NEG) c=MAX_NEG;
17    return ((QRESOLUTION)c);
18   }
19
20   QRESOLUTION MultFix(QRESOLUTION a, QRESOLUTION b) {
21    long long int c;
22    c=(long long)a*(long long)b;  //Store the result of the operation
in a bigger variable
23    c=c>>RESOLUTION;   //Shift to adjust decimals.
24    if (c>=MAX_POS) c=MAX_POS;  //If out of limits, saturate
25    if (c<=MAX_NEG) c=MAX_NEG;
26    return (QRESOLUTION)c;
27   }
28
29     int main()    {
30         short int readInputAmbience, readInputBody, writeOutput;
//We need to read at 16bit
31         QRESOLUTION calcAmbience, calcBody;
32         QRESOLUTION w [ORDER];
33         QRESOLUTION u [ORDER];
34         QRESOLUTION e;
35         QRESOLUTION yn;
36         int n1,n2;
37         int i;
38         short int Bs;
39         FILE *noiseFile, *dirtyFile, *outputFile, *aa;
40
41         aa = fopen("ambiente1.wav","rb");
42    noiseFile = fopen("amb.wav","rb");
43    dirtyFile = fopen("25.wav","rb");
```

```
44    outputFile = fopen("salidafixed.wav","wb");
45
46 //The two files need to have the same header
47
48    char buffer [44];
49    fread(buffer, 1, 44, aa); //Duplicating header
50    fwrite(buffer, 1, 44, outputFile);
51
52    fseek(aa, 34, SEEK_SET);
53    fread(&Bs, 2, 1, aa);  //Bits per sample
54    Bs /= 8;       //Bytes per sample
55
56    fseek(dirtyFile,44,SEEK_SET);
57          fseek(noiseFile,44,SEEK_SET);
58
59          w[ORDER-1]=0;  //Inicialize weights
60
61          n1=0;
62          n2=0;
63
64          while (!feof(dirtyFile)) {
65
66     starting:
67     fread(&readInputBody, Bs, 1, dirtyFile);  //Read one sample of
the file
68     fread(&readInputAmbience, Bs, 1, noiseFile);
69
70     calcBody = readInputBody;
71
72     calcAmbience=readInputAmbience;
73
74     while (n1<ORDER-1) {
75      fwrite(&readInputBody, Bs , 1, outputFile);
76      u[ORDER-n1-2] = calcAmbience;
77                  w[n1]=0;
78      n1++;
79      goto starting;   //Wait until we have enough data to start the
filter
80     }
81
82     for (i=ORDER-1; i>0;i--){
83      u[i] = u[i-1];
84     }
85     u[0]=calcAmbience;
86
87             yn = 0;                // Apply filter
88              for (i = 0; i<ORDER;i++){
89       short int temp;
90       temp= MultFix(w[i],u[i]);
91       yn = AddFix(yn,temp);
```

```
92    }
93
94              e = AddFix(calcBody,-yn);      // Error, output
95
96              for (i = 0; i<ORDER;i++){    // Update of the weights
97     short int temp;
98     temp = MultFix(e,MU);
99     temp = MultFix(temp,u[i]);
100     w[i] = AddFix(w[i],temp);
101    }
102
103    writeOutput = e;
104
105    fwrite(&writeOutput, Bs, 1, outputFile);    // Write the output
in the file
106
107         }
108
109   fcloseall();
110
111         return 0;
112     }
```

D.3. VHDL

D.3.1. Entidad y arquitectura

```
01 library IEEE;
02 use IEEE.STD_LOGIC_1164.ALL;
03 --use IEEE.STD_LOGIC_ARITH.ALL;
04 --use IEEE.STD_LOGIC_SIGNED.ALL;
05 use IEEE.NUMERIC_STD.ALL;
06
07 entity Filter is
08     Port (
09         Micro : in  std_logic_vector(15 downto 0);
10         Estetoscopio : in  std_logic_vector(15 downto 0);
11         Salida : out  std_logic_vector (15 downto 0);
12   clk: in std_logic;
13         rst: in std_logic
14   );
15 end Filter;
16
17 architecture Behavioral of Filter is
18
19 constant ORDER: integer := 32;
20 constant MU: integer := 32;
21 constant RESOLUTION: integer := 16;
22 constant MAX_POS: signed := "0111111111111111";
23 constant MAX_NEG: signed := "1000000000000000";
24
25 subtype sample is signed (RESOLUTION-1 downto 0);
26 type bit16 is array (0 to ORDER-1) of sample;
27
28 signal Mic : std_logic_vector(RESOLUTION-1 downto 0);
29 signal Cuerpo : std_logic_vector(RESOLUTION-1 downto 0);
30 signal u_sig:  bit16;
31 signal w_sig: bit16;
32 signal n1_sig:integer;
33 signal empezando_sig:boolean;
34
35 begin
36
37 Mic <= Micro;
38 Cuerpo <= Estetoscopio;
39
40 accion: process (rst, clk)
41
42 variable empezando: boolean;
43 variable calcAmbience: sample := (others => '0');
44 variable calcBody: sample := (others => '0');
45 variable output: sample := (others => '0');
```

```
46 variable u:  bit16;
47 variable w: bit16;
48 variable e: sample := (others => '0');
49 variable yn: sample := (others => '0');
50 variable temp: sample := (others => '0');
51 variable i: integer;
52 variable n1: integer;
53 variable n2: integer;
54
55 function AddFix (b,a:sample) return sample is  --Add two fixed point
values
56 variable c: signed (2*RESOLUTION-1 downto 0) := (others => '0');
57 variable d: sample;
58 begin
59   c := to_signed(to_integer(a),2*RESOLUTION)+to_signed(to_integer(b),2*RESOLUTION);
--I need to adapt the length to check overflow
60   if (c>=MAX_POS) then
61    d:=MAX_POS;
62   elsif (c<=MAX_NEG) then
63    d := MAX_NEG;
64   else
65    d:= to_signed(to_integer(c),RESOLUTION);
66   end if;
67 return d;
68 end function AddFix;
69
70 function MultFix (b,a:sample) return sample is  --Multiply to fixed
point values
71 variable c: signed (4*RESOLUTION-1 downto 0):= (others => '0');
72 variable d: sample;
73
74 begin
75   c:= to_signed(to_integer(a),2*RESOLUTION)*to_signed(to_integer(b),2*RESOLUTION);
--Need to adapt the length to check overflow
76         c := shift_right (c,RESOLUTION);
77   if (c>=MAX_POS) then
78    d:=MAX_POS;
79   elsif (c<=MAX_NEG) then
80    d:=MAX_NEG;
81   else
82    d:= to_signed(to_integer(c),RESOLUTION);
83   end if;
84 return d;
85 end function MultFix;
86
87 begin
88  if (rst='0') then
89   w(ORDER-1):= (others => '0');       --Inicialize weights
90   n1_sig<=0;
91   empezando_sig <= true;
```

```
92    Salida <= (others => '0');
93
94   elsif (rising_edge(clk)) then
95
96      calcAmbience:=signed(mic); --Take te values of the signal to
work with them as variables
97   calcBody := signed(cuerpo);
98   u := u_sig;
99   w := w_sig;
100   empezando := empezando_sig;
101
102    if (empezando) then
103     n1:=n1_sig;
104     w(n1):=(others => '0'); --Keep inizialiting w
105     u(ORDER-n1-2) := calcAmbience;
106     n1 := n1+1;
107     if (n1=ORDER-1)then
108      empezando := false;
109      w(n1):=(others => '0');
110              end if;
111     empezando_sig<= empezando;
112     n1_sig <= n1;
113              Salida <= (others => '0');
114
115    else
116
117     for i in ORDER-1 downto 1 loop
118      u(i) := u(i-1);      --Move the array to add new data
119     end loop;
120     u(0):= calcAmbience;
121
122     yn:= (others => '0');     --Apply the filter
123              for i in 0 to ORDER-1 loop
124      temp:=MultFix(w(i),u(i));
125      yn:= AddFix(yn,temp);
126     end loop;
127
128     e := AddFix(calcBody,-yn);      -- Error
129
130     for i in 0 to ORDER-1 loop    -- Update weights
131      temp := MultFix(e,to_signed(MU,RESOLUTION));
132      temp := MultFix(temp,u(i));
133      w(i) := AddFix(w(i),temp);
134     end loop;
135
136     Salida <= std_logic_vector(e);    --Write the new value in the
signal.
137
138    end if;
139
```

```
140    w_sig <= w;
141    u_sig <= u;
142
143   end if;
144
145 end process accion;
146
147 end Behavioral;
```

D.3.2. Banco de pruebas manual

Para hacer las pruebas, en un primer momento, se introducen los datos manualmente, en una versión posterior el programa los capturará automáticamente de un fichero de texto.

```
01 LIBRARY ieee;
02 USE ieee.std_logic_1164.ALL;
03
04
05 ENTITY banco_manual IS
06 END banco_manual;
07
08 ARCHITECTURE behavior OF banco_manual IS
09
10
11     COMPONENT Filter
12     PORT(
13          Micro : IN  std_logic_vector(7 downto 0);
14          Estetoscopio : IN  std_logic_vector(7 downto 0);
15          Salida : OUT  std_logic_vector(7 downto 0);
16          clk : IN  std_logic;
17          rst : IN  std_logic
18         );
19     END COMPONENT;
20
21
22    --Inputs
23    signal Micro : std_logic_vector(7 downto 0) := (others => '0');
24    signal Estetoscopio : std_logic_vector(7 downto 0) := (others
=> '0');
25    signal clk : std_logic := '0';
26    signal rst : std_logic := '0';
27
28   --Outputs
29    signal Salida : std_logic_vector(7 downto 0);
30
31    -- Clock period definitions
32    constant clk_period : time := 1 us;
33
34 BEGIN
35
36  -- Instantiate the Unit Under Test (UUT)
37    uut: Filter PORT MAP (
38           Micro => Micro,
39           Estetoscopio => Estetoscopio,
40           Salida => Salida,
41           clk => clk,
42           rst => rst
43         );
```

```
44
45    -- Clock process definitions
46    clk_process :process
47    begin
48   clk <= '0';
49   wait for clk_period/2;
50   clk <= '1';
51   wait for clk_period/2;
52    end process;
53
54
55    -- Stimulus process
56    stim_proc: process
57    begin
58       -- hold reset  .
59   rst <= '0';
60       wait for 1 us;
61       rst <= '1';
62
63
64   Estetoscopio <= "10010110";
65   Micro <= "10101001";
66     wait for 1 us;
67
68   Estetoscopio <= "10100110";
69   Micro <= "10100101";
70     wait for 1 us;
71
72   Estetoscopio <= "01011010";
73   Micro <= "11100101";
74     wait for  1 us;
75
76   Estetoscopio <= "00010111";
77   Micro <= "11001010";
78     wait for 1 us;
79
80   Estetoscopio <= "00001100";
81   Micro <= "11100010";
82     wait for 1 us;
83   Estetoscopio <= "10010110";
84   Micro <= "10101001";
85     wait for 1 us;
86
87   Estetoscopio <= "10100110";
88   Micro <= "10100101";
89     wait for 1 us;
90
91   Estetoscopio <= "01011010";
92   Micro <= "11100101";
93     wait for  1 us;
```

```
94
95   Estetoscopio <= "00010111";
96   Micro <= "11001010";
97     wait for 1 us;
98
99   Estetoscopio <= "00001100";
100   Micro <= "11100010";
101     wait for 1 us;
102   Estetoscopio <= "10010110";
103   Micro <= "10101001";
104     wait for 1 us;
105
106   Estetoscopio <= "10100110";
107   Micro <= "10100101";
108     wait for 1 us;
109
110   Estetoscopio <= "01011010";
111   Micro <= "11100101";
112     wait for  1 us;
113
114   Estetoscopio <= "00010111";
115   Micro <= "11001010";
116     wait for 1 us;
117
118   Estetoscopio <= "00001100";
119   Micro <= "11100010";
120     wait for 1 us;
121   Estetoscopio <= "10010110";
122   Micro <= "10101001";
123     wait for 1 us;
124
125   Estetoscopio <= "10100110";
126   Micro <= "10100101";
127     wait for 1 us;
128
129   Estetoscopio <= "01011010";
130   Micro <= "11100101";
131     wait for  1 us;
132
133   Estetoscopio <= "00010111";
134   Micro <= "11001010";
135     wait for 1 us;
136
137   Estetoscopio <= "00001100";
138   Micro <= "11100010";
139     wait for 1 us;
140   Estetoscopio <= "10010110";
141   Micro <= "10101001";
142     wait for 1 us;
143
```

```
144   Estetoscopio <= "10100110";
145   Micro <= "10100101";
146     wait for 1 us;
147
148   Estetoscopio <= "01011010";
149   Micro <= "11100101";
150     wait for  1 us;
151
152   Estetoscopio <= "00010111";
153   Micro <= "11001010";
154     wait for 1 us;
155
156   Estetoscopio <= "00001100";
157   Micro <= "11100010";
158     wait for 1 us;
159   Estetoscopio <= "10010110";
160   Micro <= "10101001";
161     wait for 1 us;
162
163   Estetoscopio <= "10100110";
164   Micro <= "10100101";
165     wait for 1 us;
166
167   Estetoscopio <= "01011010";
168   Micro <= "11100101";
169     wait for  1 us;
170
171   Estetoscopio <= "00010111";
172   Micro <= "11001010";
173     wait for 1 us;
174
175   Estetoscopio <= "00001100";
176   Micro <= "11100010";
177     wait for 1 us;
178   Estetoscopio <= "10010110";
179   Micro <= "10101001";
180     wait for 1 us;
181
182   Estetoscopio <= "10100110";
183   Micro <= "10100101";
184     wait for 1 us;
185
186   Estetoscopio <= "01011010";
187   Micro <= "11100101";
188     wait for  1 us;
189
190   Estetoscopio <= "00010111";
191   Micro <= "11001010";
192     wait for 1 us;
193
```

```
194   Estetoscopio <= "00001100";
195   Micro <= "11100010";
196     wait for 1 us;
197   Estetoscopio <= "10010110";
198   Micro <= "10101001";
199     wait for 1 us;
200
201   Estetoscopio <= "10100110";
202   Micro <= "10100101";
203     wait for 1 us;
204
205   Estetoscopio <= "01011010";
206   Micro <= "11100101";
207     wait for  1 us;
208
209   Estetoscopio <= "00010111";
210   Micro <= "11001010";
211     wait for 1 us;
212
213   Estetoscopio <= "00001100";
214   Micro <= "11100010";
215     wait for 1 us;
216   Estetoscopio <= "10010110";
217   Micro <= "10101001";
218     wait for 1 us;
219
220   Estetoscopio <= "10100110";
221   Micro <= "10100101";
222     wait for 1 us;
223
224   Estetoscopio <= "01011010";
225   Micro <= "11100101";
226     wait for  1 us;
227
228   Estetoscopio <= "00010111";
229   Micro <= "11001010";
230     wait for 1 us;
231
232   Estetoscopio <= "00001100";
233   Micro <= "11100010";
234     wait for 1 us;
235   Estetoscopio <= "10010110";
236   Micro <= "10101001";
237     wait for 1 us;
238
239   Estetoscopio <= "10100110";
240   Micro <= "10100101";
241     wait for 1 us;
242
243   Estetoscopio <= "01011010";
```

```
244   Micro <= "11100101";
245     wait for  1 us;
246
247   Estetoscopio <= "00010111";
248   Micro <= "11001010";
249     wait for 1 us;
250
251   Estetoscopio <= "00001100";
252   Micro <= "11100010";
253     wait for 1 us;
254   Estetoscopio <= "10010110";
255   Micro <= "10101001";
256     wait for 1 us;
257
258   Estetoscopio <= "10100110";
259   Micro <= "10100101";
260     wait for 1 us;
261
262   Estetoscopio <= "01011010";
263   Micro <= "11100101";
264     wait for  1 us;
265
266   Estetoscopio <= "00010111";
267   Micro <= "11001010";
268     wait for 1 us;
269
270   Estetoscopio <= "00001100";
271   Micro <= "11100010";
272     wait for 1 us;
273   Estetoscopio <= "10010110";
274   Micro <= "10101001";
275     wait for 1 us;
276
277   Estetoscopio <= "10100110";
278   Micro <= "10100101";
279     wait for 1 us;
280
281   Estetoscopio <= "01011010";
282   Micro <= "11100101";
283     wait for  1 us;
284
285   Estetoscopio <= "00010111";
286   Micro <= "11001010";
287     wait for 1 us;
288
289   Estetoscopio <= "00001100";
290   Micro <= "11100010";
291     wait for 1 us;
292   Estetoscopio <= "10010110";
293   Micro <= "10101001";
```

```
294     wait for 1 us;
295
296   Estetoscopio <= "10100110";
297   Micro <= "10100101";
298     wait for 1 us;
299
300   Estetoscopio <= "01011010";
301   Micro <= "11100101";
302     wait for  1 us;
303
304   Estetoscopio <= "00010111";
305   Micro <= "11001010";
306     wait for 1 us;
307
308   Estetoscopio <= "00001100";
309   Micro <= "11100010";
310     wait for 1 us;
311   Estetoscopio <= "10010110";
312   Micro <= "10101001";
313     wait for 1 us;
314
315   Estetoscopio <= "10100110";
316   Micro <= "10100101";
317     wait for 1 us;
318
319   Estetoscopio <= "01011010";
320   Micro <= "11100101";
321     wait for  1 us;
322
323   Estetoscopio <= "00010111";
324   Micro <= "11001010";
325     wait for 1 us;
326
327   Estetoscopio <= "00001100";
328   Micro <= "11100010";
329     wait for 1 us;
330   Estetoscopio <= "10010110";
331   Micro <= "10101001";
332     wait for 1 us;
333
334   Estetoscopio <= "10100110";
335   Micro <= "10100101";
336     wait for 1 us;
337
338   Estetoscopio <= "01011010";
339   Micro <= "11100101";
340     wait for  1 us;
341
342   Estetoscopio <= "00010111";
343   Micro <= "11001010";
```

```
344     wait for 1 us;
345
346   Estetoscopio <= "00001100";
347   Micro <= "11100010";
348     wait for 1 us;
349   Estetoscopio <= "10010110";
350   Micro <= "10101001";
351     wait for 1 us;
352
353   Estetoscopio <= "10100110";
354   Micro <= "10100101";
355     wait for 1 us;
356
357   Estetoscopio <= "01011010";
358   Micro <= "11100101";
359     wait for  1 us;
360
361   Estetoscopio <= "00010111";
362   Micro <= "11001010";
363     wait for 1 us;
364
365   Estetoscopio <= "00001100";
366   Micro <= "11100010";
367     wait for 1 us;
368   Estetoscopio <= "10010110";
369   Micro <= "10101001";
370     wait for 1 us;
371
372   Estetoscopio <= "10100110";
373   Micro <= "10100101";
374     wait for 1 us;
375
376   Estetoscopio <= "01011010";
377   Micro <= "11100101";
378     wait for  1 us;
379
380   Estetoscopio <= "00010111";
381   Micro <= "11001010";
382     wait for 1 us;
383
384   Estetoscopio <= "00001100";
385   Micro <= "11100010";
386     wait for 1 us;
387
388       wait;
389    end process;
390
391 END;
```

D.3.3. Banco de pruebas automático

```
01 LIBRARY ieee;
02 USE ieee.std_logic_1164.ALL;
03 use IEEE.STD_LOGIC_TEXTIO.ALL;
04
05 LIBRARY std;
06 use std.textio.all;
07
08 ENTITY banco IS
09 END banco;
10
11 ARCHITECTURE behavior OF banco IS
12
13     COMPONENT Filter
14     PORT(
15          Micro : IN   std_logic_vector(15 downto 0);
16          Estetoscopio : IN   std_logic_vector(15 downto 0);
17          Salida : OUT   std_logic_vector(15 downto 0);
18          clk : IN   std_logic;
19          rst : IN   std_logic
20         );
21     END COMPONENT;
22
23    --Inputs
24    signal Micro : std_logic_vector(15 downto 0) := (others => '0');
25    signal Estetoscopio : std_logic_vector(15 downto 0) := (others
=> '0');
26    signal clk : std_logic := '0';
27    signal rst : std_logic := '0';
28
29   --Outputs
30    signal Salida : std_logic_vector(15 downto 0);
31
32    -- Clock period definitions
33    constant clk_period : time := 1 us;
34
35 BEGIN
36
37  -- Instantiate the Unit Under Test (UUT)
38    uut: Filter PORT MAP (
39           Micro => Micro,
40           Estetoscopio => Estetoscopio,
41           Salida => Salida,
42           clk => clk,
43           rst => rst
44         );
45
46    -- Clock process definitions
```

```
47    clk_process :process
48    begin
49   clk <= '0';
50   wait for clk_period/2;
51   clk <= '1';
52   wait for clk_period/2;
53    end process;
54
55    -- Stimulus process
56    stim_proc: process
57
58     variable micData : std_logic_vector (15 downto 0);
59     variable stetData : std_logic_vector (15 downto 0);
60     variable outpute : std_logic_vector (15 downto 0);
61
62  variable linea:line;
63
64     file micFile:text open read_mode is "mic.dat";  --Open the files
in read or write mode
65     file stetFile:text open read_mode is "stet.dat";
66     file salidaFile:text open write_mode is "salida.dat";
67  begin
68
69          --hold reset
70   rst <= '0';
71          wait for 1 us;
72          rst <= '1';
73
74   while not endfile(micFile) loop
75    outpute:= Salida;
76              WRITE(linea, outpute);      --Write the new value in
the text ouput file.
77              WRITELINE(salidaFile, linea);
78
79              readline(micFile,linea);  --Read new data from the files.
80              read(linea, micData);
81
82              readline(stetFile,linea);
83              read(linea, stetData);
84
85              Micro <= micData;
86              Estetoscopio <= stetData;
87
88              wait for 1 us;
89         end loop;
90
91         outpute:= Salida;
92         WRITE(linea, outpute);
93         WRITELINE(salidaFile, linea);
94
```

```
95   wait;
96     end process;
97
98 END;
```

D.4. Conversión Binario ⟷ Texto

D.4.1. Binario ⟶ Texto

```
01     #include "stdio.h"
02     #include "math.h"
03     #include "string.h"
04     #include "stdlib.h"
05
06     void dec2bin(unsigned short int decimal, char *salida) {
07         unsigned short int remain;
08         char temp[16];
09         int i;
10
11         for (i = 0; i<16; i++){
12             remain = decimal % 2;
13             decimal = decimal / 2;
14             temp[i] = remain + '0';
15         }
16
17         for(i = 0; i<16;i++){
18             salida[i]=temp[16-1-i];
19         }
20
21 }
22
23     int main()
24     {
25         short int buffer;
26         char binary[16];
27         short int bs, Bs;
28         FILE *wavFile, *txtFile;
29
30   wavFile = fopen("ambiente2.wav","rb");
31   txtFile = fopen("a.dat","wb");
32
33   fseek(wavFile, 34, SEEK_SET);
34   fread(&bs, 2, 1, wavFile);  //Bits per sample
35   Bs = bs / 8;       //Bytes per sample
36
37   fseek(wavFile,44,SEEK_SET);
38
39         while (!feof(wavFile)) {
40
41    fread(&buffer, Bs, 1, wavFile);   //Read sample from wav file
42
43             dec2bin(buffer,binary);        //Convert to ascii char
array
44
```

```
45              fprintf(txtFile,"%s\n",binary);  //write line in text
file
46
47          }
48
49   fcloseall();
50
51          return 0;
52      }
```

D.4.2. Texto ⟶ Binario

```
01    #include "stdio.h"
02    #include "math.h"
03    #include "string.h"
04    #include "stdlib.h"
05
06    int main()
07    {
08
09        unsigned char buffer[16];
10        unsigned short int binario[16];
11        unsigned short  int entero;
12
13        int i;
14        short int bs, Bs;
15        FILE *txtFile, *jokerFile, *wavFile;
16
17  txtFile = fopen("salida.dat","rb");
18  jokerFile = fopen("cuerpo1.wav","rb");
19  wavFile = fopen("procesado.wav","wb");
20
21  unsigned char bufferdup [44];
22  fread(bufferdup, 1, 44, jokerFile); //Duplicate headers
23  fwrite(bufferdup, 1, 44, wavFile);
24
25  fseek(jokerFile, 34, SEEK_SET);
26  fread(&bs, 2, 1, jokerFile);  //Bits per sample
27  Bs = bs / 8;       //Bytes per sample
28
29        while (!feof(txtFile)) {
30
31            fscanf(txtFile,"%s",buffer); //Read line
32
33            entero=0;       //convert to integer
34            for (i=0; i<16;i++){
35                entero+= ((int)buffer[i]-(int)'0')*(1<<(16-i));
36            }
37
38            fwrite(&entero, Bs, 1, wavFile);  //write value in wav
file
39
40        }
41
42  fcloseall();
43
44        return 0;
45    }
```

Bibliografía

[1] Wikipedia, "Stethoscope"http://en.wikipedia.org/wiki/Stethoscope

[2] Meyers Konversations-Lexikon (1885-90). Public Domain.

[3] Maw and Sons, Book of Illustrations, (1869). Public Domain.

[4] Howard Hughes Medical Institute, History of Stethoscopes and Sphygmomanometers, http://www.hhmi.org/biointeractive/museum/exhibit98/content/b6_17info.html

[5] History of Stethoscopes, https://www.standris.com/education_history.cfm

[6] Atlas de Ruidos respiratorios, http://escuela.med.puc.cl/Publ/AtlasRuidos/introduccion.html

[7] "Chris", The Auscultation Assistant, http://www.med.ucla.edu/wilkes/intro.html

[8] Richard G. Lyons, Understanding Digital Signal Processing, Second Edition, Prentice Hall,2004

[9] Modificado de: Wikipedia, http://en.wikipedia.org/wiki/File:Lms_filter.png

[10] Wikipedia, http://en.wikipedia.org/wiki/File:Gradient_descent.png

[11] The free sound project, "greyseraphim"http://www.freesound.org/samplesViewSingle.php?id=21409

[12] John E. may. Guyton y Hall, Compendio de Fisiología médica, 11ª ed. Madrid, Elsevier, 2007

[13] Fauci et al.Harrison, Principios de Medicina Interna, Vol.2 17ªed. Madrid, MaGrawHill, 2009

[14] Una introducción a las enfermedades pulmonares ocupacionales, American Lung Association, New York, Macmillan, 1979

[15] R.Putz, R.Pabst, Atlas de anatomía humana Sobotta: volumen 2, 21ªed, Madrid, Panamericana, 2000

[16] J.M Prieto Valtueña, Exploración clínica práctica, 26ªed.Barcelona, Elsevier, 2005

Anexo I

Informe original presentado en la Universidad de Glasgow (*University of Glasgow*)