Escuela Politécnica Superior de Linares

Grado en I.T. de las Telecomunicaciones

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Linares

Trabajo Fin de Grado

AFINADOR POLIFONICO PARA IPHONE

Alumno: José María Torres Ortega.

Tutor: Prof. D Pedro Vera Candeas.
Depto.: Ingeniería de Telecomunicación.

Septiembre, 2014

Gracias a toda mi familia y amigos
por ayudarme y apoyarme siempre.

INDICE DE CONTENIDO

INDICE DE FIGURAS

INDICE DE TABLAS

Capítulo 1

Resumen

En este trabajo fin de grado tendremos implementar un afinador capaz de soportar tanto señales monofónicas como señales polifónicas. El primer objetivo del mismo será implementar técnicas de detección de pitch en frecuencia para señales monofónicas. A continuación se realizará la extensión del algoritmo a señales polifónicas.

La metodología a desarrollar será la siguiente:

1) Desarrollo de una librería en MATLAB para detección de pitch con técnicas frecuenciales.
2) Pruebas de la librería con señales monofónicas y polifónicas.
3) Programación de la aplicación para soportar señales monofónicas.
4) Extensión de la aplicación para señales polifónicas.
5) Pruebas de la aplicación.

Capítulo 2

Introducción.

2.1. Introducción de la memoria.

La afinación de los instrumentos es de gran importancia para poder sacar el mayor rendimiento a nuestro instrumento en este caso a una guitarra y a nuestra creatividad.

Para afinar los instrumentos tiene gran interés el "la 440" que es la frecuencia de sonido referencia para la afinación de todos los instrumentos musicales. Por regla general es necesario algún instrumento o componente electrónico para poder determinar si una nota esta afinada o no.

Los instrumentos encargados de esto son los afinadores los cuales en un principio eran analógicos posteriormente se desarrollaron los digitales. También existen varios tipos de afinadores: los clásicos o monocromáticos y los polifónicos o cromáticos.

En los afinadores clásicos es necesario afinar (tocar) una a una cada cuerda de la guitarra puesto que se basan en medir la diferencia entre el sonido recogido y la referencia (la, 440 Hz).

Por otro lado, en los afinadores polifónicos no es necesario afinar (tocar) una a una cada cuerda de la guitarra, solo bastara con tocar todas las cuerdas de una vez y este nos dirá como de afinada o desafinada cada una de las seis cuerdas de la guitarra. El principio de estos afinadores polifónicos o cromáticos es que pueden reconocer las tonalidades de una o mas cuerdas. Hace unos años lo normal era adquirir estos equipos en tiendas de música.

En la actualidad el auge de las telecomunicaciones ha provocado un aumento considerable del número de personas que son poseedores de Smartphones o teléfonos inteligentes.

Estos teléfonos permiten a los usuarios realizar diversas funciones y también les ofrece la posibilidad de tener múltiples aplicaciones. Las cuales pueden adquirirse en las tiendas de

aplicaciones de cada plataforma (Android, IOS, Windows) como puede ser: la AppStore, PlayStore, etc. Por estas razones los teléfonos móviles han sustituido a todo tipo de dispositivos además de esto poseen una gran capacidad de procesamiento que hace unos años eran inimaginable y que puede ser comparables a un ordenador de mesa o portátil de bajas prestaciones técnicas.

Entre el extenso catalogo de aplicaciones disponibles para móviles podemos encontrar los afinadores. Estas aplicaciones de afinación se encargan de simular el funcionamiento de un afinador tradicional ya sea monocromático o cromático, las cuales obtienen unos resultados igual de buenos que los afinadores como tal. Para ello harán uso de sus interfaces de captación de audio y su gran capacidad de procesamiento, mostrando los resultados casi de forma instantánea.

Por otro lado una ventaja reseñable de las aplicaciones de afinación es que llega a un público mayor y también normalmente suelen ser igual o más sencillos que los afinadores tradicionales.

En este trabajo, por tanto describiremos la elaboración de un afinador polifónico para una plataforma móvil, concretamente para IOS.

Los conceptos matemáticos que harán posible su funcionamiento serán la transformada rápida de Fourier (FFT) , los métodos de estimación multi-pitch y otros muchos mas conceptos que trataremos con detenimiento en los próximos capítulos.

2.2. Contenido de la memoria.

Este apartado está destinado a ofrecer una visión global de cada uno de las partes o apartados en los que se encuentra dividida la memoria. Este documento se compone de ocho capítulos en total.

El capitulo 1 recibe el nombre de “Resumen” y tiene como objetivo principal el de dar una pequeña explicación sobre lo que va consistir el trabajo que es objeto de la realización de esta memoria.

El capítulo 2 se titula “Introducción” y pretende dar una visión global del proyecto, así como de proporcionar un pequeño resumen de lo que vamos a poder ver en el resto de capítulos.

El capítulo 3 se denomina “Objetivos” y aquí describiremos cuales son los objetivos o

premisas que se desean alcanzar con el desarrollo y elaboración de este trabajo. Este capítulo se dividirá en dos apartados para distinguir entre objetivos de carácter general y especifico.

El capítulo 4, “Materiales y Métodos”, servirá para explicar con detenimiento la elaboración y funcionamiento de la aplicación por lo tanto incluirá conceptos de diversa índole relacionados con la programación orientada a objetos y el procesamiento de audio en tiempo real. Debido a todo lo comentado anteriormente, este capítulo estará dividido en una amplia serie de apartados por lo cual supondrá el capítulo más importante de toda la memoria.

El siguiente capítulo, el 5, lleva por título, “Resultados y Discusión”. En este capítulo se realizaran una serie de pruebas en la aplicación para distintos casos de evaluación y se discutirá el resultado de los mismos, considerando si esos resultados son los suficientemente satisfactorios o por el contrario no lo son. Si el resultado no es satisfactorio se expondrán una serie de soluciones para que el resultado pase a ser positivo.

El capítulo 6, “Conclusión”, pretende dar una valoración global del trabajo realizado, además, se exponen algunas ideas para la futura mejora de la aplicación. En este capitulo también se incluye una visión de futuro sobre las posibles aplicaciones que puede tener la app en un futuro.

El capitulo 7 se denomina “Anexos y Planos” y aquí se incluye todas las líneas de código de la aplicación y todas las vistas que nuestra aplicación puede mostrar. Además en este capítulo también se incluirá un manual de usuario para que se comprenda el funcionamiento de la misma y que así se pueda hacer un uso correcto de la misma.

El último capítulo se titula “Referencias Bibliográficas”, en él se listan todas las fuentes en las que se ha basado la realización de trabajo.

Capítulo 3

Objetivos

3.1 Objetivo General.

El objetivos son diseñar un afinador polifónico para guitarras que pueda utilizarse en dispositivos con sistema operativo IOS (Iphone, Ipad y Ipod) y que soporte tanto señales monofónicas como polifónicas.

Por lo cual deberá crearse usando el lenguaje Objective-C, para que puede funcionar sobre el sistema operativo IOS y hacer uso de diferentes frameworks para poder captar el audio y realizar el procesamiento de dicho audio. No obstante primero se utilizara Matlab para su desarrollo.

3.2 Objetivos Específicos.

Los objetivos específicos son los siguientes:

- Conseguir que la aplicación sea apta para poder ejecutarse en equipos con capacidades de hardware mas modestas como IPHONEs, IPADs y IPODs de generaciones anteriores a la actual.
- Implementar un algoritmo que permita que la aplicación se pueda ejecutar en tiempo real y con la mayor rapidez posible.
- Diseñar una interfaz sencilla de manera que pueda ser utilizada por una gran numero de usuarios sin importar sus conocimientos musicales ni tecnológicos.
- Que los resultados mostrados por la aplicación móvil se correspondan con la realidad, sin que indique la presencia de notas erróneamente.

- Implementar técnicas de detección de pitch en frecuencia para señales monofónicas y extensión del mismo para señales polifónicas.

Capítulo 4

Materiales y métodos

En este capítulo vamos a ver los distintos métodos que hemos utilizado para desarrollar la aplicación. En un primer lugar vamos a ver los distintos métodos de detección y vamos a explicar cuál es el que hemos elegido, mas tarde pasaremos a la parte de programación en IOS, en la que veremos las bibliotecas que hemos utilizado y como hemos adaptado el código que como anterioridad hemos desarrollado para detectar las notas. El código para la detección se desarrollara en el software Matlab.

4.1. Estimación multi-pitch

El problema de encontrar la frecuencia fundamental, o el tono, de una forma de onda periódica aparece en muchas aplicaciones de procesamiento de señales, por ejemplo, en aplicaciones que implican señales de voz y de audio. Por ejemplo, el procesamiento de la frecuencia fundamental de audio juega un papel clave en la transcripción automática y clasificación de la música. Debido a la importancia del problema, existen una amplia variedad de métodos fundamentales de estimación de frecuencia que se han desarrollado en la literatura específica. En la mayoría casos, estos métodos se basan en un modelo en el que sólo un único conjunto de sinusoides relacionadas armónicamente están presentes al mismo tiempo.

De hecho, el problema de la estimación de varias frecuencias o pitch, es decir, el problema de la estimación de las frecuencias fundamentales de múltiples formas de onda periódicas, es difícil, y este ha recibido mucha menos atención que el caso de un solo tono, aunque se pueden encontrar notables excepciones. El escenario de varios tonos o

frecuencias ocurre regularmente en las señales de música, tal vez incluso con más frecuencia que el caso de un solo tono, ya que menudo también se produce para el procesamiento del habla. Por lo general, la situación se produce cuando múltiples instrumentos o altavoces están presentes al mismo tiempo o cuando múltiples tonos se están tocando en un instrumento musical.

El problema de la estimación de varios pitchs o tonos se puede definir de la siguiente manera:

Se considere una señal que consta de varios, K, series de armónicos (que se pueden nombrar como fuentes) con frecuencias fundamentalesω_k, para k = 1,. . . , K, que está dañada debido a la presencia de ruido blanco aditivo complejo circularmente simétrico gaussiano, w (n), que tiene varianza σ^2, para n = 0,. . . , N - 1, es decir,

$$x(n) = \sum_{k=1}^{K} \sum_{l=1}^{L} a_x l e^{j\,\omega_k ln} + \omega(n) \qquad (1)$$

donde $a_{k,l} = A_{k,l} e^{j\,\emptyset_{k,l}}$, l, con $A_{k,l} < 0$ y$\emptyset_{k,l}$ la amplitud y la fase de la $l'th$armónico de la fuente k-ésima, respectivamente. El problema es entonces estimar la frecuencia fundamentalω_k, o los pitchs, de un conjunto de N muestras medidas, x (n). En el presente trabajo, se supone que el numero de fuentes, K, es conocida y que el número de armónicos, L, de cada fuente también se conoce y es la misma para todas las fuentes.

Para el caso de un solo tono, a menudo también se asumió un orden L conocido en la literatura de que se trate con estimación paramétrica de la frecuencia fundamental Aun así, puede parecer como un supuesto restrictivo que K y L se consideran conocido, pero para muchas aplicaciones prácticas, no se requiere que el orden se conozca con precisión. Siempre que el orden no varié demasiado, es suficiente simplemente asumir una media de pedidos.

El papel del orden estimación es principalmente para evitar ambigüedades en las funciones de coste que pueden causar estimaciones espurias de la verdadera frecuencia fundamental (con q, g $\in$ N), tales como los conocidos problemas de halvings y duplicaciones.

Señalamos, de paso, que aquí consideramos las amplitudes y las fases $\{ A_{k,l} e^{j\,\emptyset_{k,l}} \}$ como parámetros de perturbación que no son de interés. Sin embargo, vemos a partir de (1) que

las amplitudes complejas son parámetros lineales que son, en principio, mucho más fácil de encontrar que las frecuencias fundamentales no lineales $\{\omega_k\}$. Dado que las frecuencias fundamentales $\{\omega_k\}$, las amplitudes y fases puede ser fácilmente encontrado usando uno de los estimadores existentes.

Además, remarcamos que las señales de valores reales también pueden escribirse utilizando el modelo complejo en (1) a través de la utilización de la señal analítica en tiempo discreto (downsampled), siempre que existan armónicos en la señal real cerca de 0 y π relativa a N. En este caso, hemos utilizado la formulación compleja debido a su simplicidad de notación y porque conduce a algoritmos computacionalmente simples.

De modo que proponemos y evaluamos varios estimadores para encontrar las frecuencias fundamentales $\{\omega_k\}$ basado en principios bien fundados de procesamiento estadístico de señales.

Estos métodos tiene la siguiente forma simple:

$$\langle\widehat{\omega}_k\rangle = \arg max_{\langle\omega_k\rangle} \sum_{k=1}^{K} J(\omega_K) \tag{2}$$

donde la función J (·) sólo depende de la fuente k.

Este significa que una estimación del conjunto de frecuencias fundamentales se puede obtener mediante la evaluación de una función de coste J (ω_k). Específicamente, el coste de la función J (·) tendría que ser calculado para un número diferente de armónicos con el fin de determinar la frecuencia fundamental, pero las frecuencias fundamentales todavía se pueden determinar de forma independiente para las fuentes individuales.

Por otra parte para estimar la frecuencia fundamental o frecuencias fundamentales si se trata de un caso en el que tenemos más de una fuente, podemos utilizar distintos métodos que se pueden dividir según en los modelos que están basados.

De esta manera los métodos de estimación pueden ser:

- Métodos estadísticos que están basados en modelos estadísticos.
- Métodos de filtrado que se basan en el principio de filtrado optimo.
- Métodos de sub-espacios que se basan en los principios de ortogonalidad del sub-espacio.

De todos estos métodos, vamos a tratar con detenimiento los métodos estadísticos pues son los que nos interesan para el desarrollo de nuestra aplicación, de los demás métodos se podrán unos cuantos ejemplos pero si se desea mas información se podrá encontrar en literatura técnica especifica de estimación simple y multi-pitch de la frecuencia fundamental.

4.2. Conceptos preliminares.

4.2.1. Modelo de señal.

Comenzamos por introducir algunas notaciones útiles y definiciones. En primer lugar, llevaremos a cabo la construcción de un vector formado a partir de M muestras consecutivas de la señal observada como:

$$x(n) = [x(n) \cdots x(n + M - 1)]^T \tag{3}$$

con M ≤ N. Recordamos que estas señales se define para $n = 0, \dots, N - 1$ y hay que tener en cuenta que la constante M se puede elegir de manera diferente dependiendo del contexto. Se define la matriz$Z_k \in \mathbb{C}^{M \times L}$que tiene una estructura de Vandermonde, está construida a partir de L vectores complejos sinusoidales como$Z_k = [\, z(\omega_k) \cdots z(\omega_k L)]$, con $z(\omega) = \left[1\ e^{j\omega} \cdots e^{j\omega(M-1)}\right]^T$, y un vector que contiene las amplitudes complejas $a_k = \left[a_{k,1} \cdots a_{k,L}\right]^T$. Introduciendo $z_k = e^{jw_k}$, la matriz de Vandermonde puede ser vista con la siguiente estructura:

$$Z_k = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ e^{jw_k} & e^{jw_k 2} & \dots & e^{jw_k L_k} \\ \vdots & \vdots & \ddots & \vdots \\ e^{jw_k(M-1)} & e^{jw_k(M-1)L_k} & \dots & e^{jw_k L_k(M-1)} \end{bmatrix} \tag{4}$$

$$= \begin{bmatrix} 1 & 1 & \cdots & 1 \\ z_k^1 z_k^2 & \cdots & z_k^{L_k} \\ \vdots & \vdots & \ddots & \vdots \\ z_k^{(M-1)} z_k^{(M-1)2} & \cdots & z_k^{(M-1)L_k} \end{bmatrix} \quad (5)$$

Usando estas definiciones, el modelo señal en (1) puede ser escrito como

$$x(n) = \sum_{k=1}^{K} \begin{bmatrix} e^{j\omega_k 1n} & & 0 \\ & \ddots & \\ 0 & & e^{j\omega_k Ln} \end{bmatrix} a_k + e(n) \quad (6)$$

$$\triangleq \sum_{k=1}^{K} Z_k(n) a_k + e(n) \quad (7)$$

Como puede verse, ya sea la amplitud compleja o la matriz Vandermonde pueden ser vistos como variables dependientes de n, es decir, $a_k(n) = D_n a_k$ y $Z_k(n) = Z_k D_n$ con

$$D_n = \begin{bmatrix} e^{j\omega_k 1n} & & 0 \\ & \ddots & \\ 0 & & e^{j\omega_k L_k n} \end{bmatrix} \quad (8)$$

Tomamos nota de que la constante M como hemos comentado anteriormente se elige de manera diferente en función del método que estemos utilizando.

4.2.2. Modelo de la matriz de covarianza.

A continuación, se define el matriz de covarianza como

$$R = E\langle x(n)x^H(n)\rangle \tag{9}$$

Aquí, E $\{\cdot\}$ y $(\cdot)^H$ denota la esperanza matemática y la transpuesta conjugada, respectivamente. En la práctica, la matriz de covarianza es desconocida y se sustituye por la matriz de covarianza de la muestra que se define como

$$\hat{R} = \frac{1}{N-M+1}\sum_{n=0}^{N-M} x(n)x^H(n) \tag{10}$$

Claramente, para que$\hat{R}$ pueda tener inversa, es necesario que M $\leq \frac{N}{2}$. En lo que sigue, vamos a suponer que M se elige en función de si se utiliza la inversa de la matriz de covarianza.

En general las dimensiones de la matriz de covarianza va a ser proporcional al número de muestras ya que el rendimiento final dependerá de ambos parámetros.

Como bien sabemos, la estimación de matriz de covarianza en (10) es la estimación de probabilidad máxima para las señales de distribución gaussiana, siempre que las sub-vectores son independientes. Además, es generalmente consistente para los procesos ergódicos independientemente de sus distribuciones, lo que significa que la estimación tenderá hacia la verdadera matriz de covarianza cuando el número de observaciones tiende a infinito.

Al derivar los estimadores, vamos a hacer un uso extensivo de las propiedades de la matriz de covarianza ideal en (9), pero en la práctica será reemplazada por la matriz de covarianza de la muestra como se define en (10). Debe quedar claro a partir del contexto a cuál de las dos nos estamos refiriendo en cada ocasión. Suponiendo que las fuentes son estadísticamente independientes, la matriz de covarianza de la señal observada puede ser escrita como

$$R = \sum_{k=1}^{K} R_k = \sum_{k=1}^{K} E\{x_k(n)x_k^H(n)\} \tag{11}$$

Insertando el modelo de señal para fuentes individuales en esta expresión, podemos escribir la matriz de covarianza como

$$R = \sum_{k=1}^{K} E\left\langle \left(Z_k a_k(n) + e_k(n)\right)\left(Z_k a_k(n) + e_k(n)\right)^H \right\rangle \tag{12}$$

$$= \sum_{k=1}^{K} Z_k E\langle a_k(n) a_k^H(n)\rangle Z_k^H + E\{e_k(n) e_k^H(n)\} \tag{13}$$

$$= \sum_{k=1}^{K} Z_k P_k Z_k^H + Q \tag{14}$$

donde Z_k asumimos que es determinista. La matriz Q es la matriz de la covarianza de $e(n)$, es decir,

$$Q = E\{e(n) e^H(n)\} \tag{15}$$

$$= \sum_{k=1}^{K} E\{e(n) e^H(n)\} = \sum_{k=1}^{K} Q_k \tag{16}$$

también conocida como la matriz de covarianza de ruido con Q_k que denota la matriz de covarianza de ruido de la fuente de observación de ruido k-ésima, $e_k(n)$. La matriz P_k es la matriz de covarianza de las amplitudes, es decir,

$$P_k = E\{a_k(n) a_k^H(n)\} \tag{17}$$

Si las fases de los armónicos son estadísticamente independientes y están distribuidas uniformemente en el intervalo $(-\pi, \pi]$, esta matriz se puede simplificar y quedara como

$$P_k = diag\left(\left[A_{k,1}^2 \cdots A_{k,L}^2\right]\right) \tag{18}$$

4.2.3. Parámetros de estimación limite.

El Cramer-Rao limite inferior (CRLB) expresa una cota inferior en la varianza de los estimadores de un parámetro determinista.

Para una sola fuente y un número alto de muestras, es decir, para el caso de que N »1, el límite Cramer-Rao inferior asintótico(CRLB) para la fuente k-ésima puede demostrarse que es

$$CRLB_K = \frac{6\sigma^2}{N^3 \sum_{l=1}^{L} A_{k,l}^2 l^2} \tag{19}$$

El CRLB se puede observar que va a depender de la pseudo relación señal-ruido (PSNR), definida como

$$PSNR_k = 10 \log_{10} \frac{\sum_{l=1}^{L} A_{k,l}^2 l^2}{\sigma^2} \; [dB] \tag{20}$$

Bajo el supuesto de que las fuentes son independientes y que las frecuencias armónicas son distintas, también se puede esperar que (19) se mantenga aproximadamente igual para el problema de la estimación de las frecuencias fundamentales en (1). Sin embargo, para un número bajo de muestras, la CRLB exacta para una frecuencia fundamental dependerá también de los parámetros de otras fuentes.

4.3. Métodos de estimación estadísticos.

La idea básica de los métodos de estimación estadística consiste en plantear un modelo de la señal observada en términos de una función de densidad de probabilidad (PDF), que en nuestro caso está parametrizada por los parámetros del conjunto de sinusoides armónicamente relacionadas. Dentro de este marco, los parámetros de estas sinusoides puede ser vistos como deterministas, pero desconocidos de forma que ninguno de los componentes estocásticos de la señal, se considera aleatorio. Estos parámetros deterministas pero desconocidos, pueden estimarse entonces mediante la maximización de la denominada probabilidad de la señal observada, es decir, los parámetros que son más probables la señal observada y estas son las estimaciones de máxima verosimilitud. La función objetivo se conoce como función de probabilidad, ya que se considera como una función de los parámetros desconocidos y no de la señal observada. El principio de la estimación de máxima verosimilitud es probablemente el más comúnmente utilizado y los estimadores basados en este principio son bien conocidos por tener un excelente rendimiento para un gran número de muestras. Para ciertos casos, como en el caso gaussiano, el estimador de máxima verosimilitud también tiene una forma simple, de hecho, es el minimizador del error cuadrático medio y del ruido blanco gaussiano de observación, es decir, el estimador de mínimos cuadrados. Es, sin embargo, complicado derivar métodos computacionalmente eficientes para la búsqueda de parámetros no lineales como las frecuencias de sinusoides. Por lo cual a continuación presentaremos los fundamentos de la estimación de máxima verosimilitud y máxima a posteriori (MAP), y luego mostraremos cómo estos principios pueden ser utilizados de diversas formas para poder encontrar la frecuencia fundamental de las distintas señales periódicas y resolver algunos de los problemas que estos llevan asociados, tales como la selección del modelo y el orden de estimación.

4.3.1. Estimación de máxima verosimilitud.

Vamos a considerar primero sólo el caso de un solo tono, es decir, $K = 1$. El método se puede generalizar para el escenario de varios largos o tonos, pero en su forma exacta, no es muy útil, ya que da lugar a un problema de optimización multidimensional no lineal algo complicado, no obstante más tarde vamos a considerar una solución aproximada del mismo. Ahora vamos a presentar el método de máxima verosimilitud para encontrar los

parámetros de una señal periódica que se encuentra sola. El algoritmo funciona en un sub-vector de la señal en el tiempo n, definido como

$$x_k(n) = [x(n) \cdots x(n + M - 1)]^T \tag{21}$$

que se construye a partir de la señal observada de un solo tono $x_k(n)$. Para muchas señales, un sub-vector pueden modelarse como la suma de L_k sinusoides complejasarmónicamente relacionadas con ruido gaussiano coloreado e_k que tiene unan matriz de covarianza Q_k, es decir,

$$x_k = s_k(n) + e_k(n) \tag{22}$$

$$= Z_k(n)a_k + e_k(n) \tag{23}$$

con $a_k = \left[A_{k,1}e^{j\phi_{k,1}} \cdots A_{k,L_k}e^{j\phi_{k,L_k}}\right]^T$ siendo el vector que contiene las amplitudes complejas. Ademas $Z_k(n)$ es una matriz de estructura Vandermonde en tiempo n, definida como

$$Z_k(n) = \left[z_1(n) \cdots Z_{L_k}(n)\right] \tag{24}$$

donde la entrada m-esima del vector columna $z_l(n) \in \mathbb{C}^M$ esta definida como $[z_l(n)]_m = e^{j\omega_0 l(n+m-1)}$.Ya que $x_k(n)$tiene longitud M y tenemos N observaciones de $x_k(n)$, podemos así construir un conjunto de $G = N - M + 1$ diferentes sub-vectores $\{x_k(n)\}_{n=0}^{G-1}$.

A continuación, se introduce la señal y el parámetro de ruido θ_k que contiene la frecuencia fundamentalω_k, las amplitudes complejas y por lo tanto implícitamente el orden L_k y la matriz Q de covarianza de ruido del modelo (23). Suponiendo que Q_k tiene inversa,

la función de probabilidad de la señal observada para un sub-vector $x(n)$, entonces puede escribirse como

$$p(x_k(n);\theta_k) = \frac{1}{\pi^M \det(Q_k)} e^{-e_k^H(n)Q_k^{-1}e_k(n)} \tag{25}$$

Con $det\ (\cdot)$ que denota el determinante de la matriz. Ahora, suponiendo que la parte determinista de $s_k(n)$ es estacionaria y que $e_k(n)$ es independiente e idénticamente distribuida sobre n, la probabilidad del conjunto de vectores observados $\{x_k(n)\}_{n=0}^{G-1}$ se puede escribir como

$$\begin{aligned} p(\{x_k(n)\};\theta_k) &= \prod_{n=0}^{G-1} p(x_k(n);\theta_k) \\ &= \frac{1}{\pi^{MG} \det(Q_k)^G} e^{-\sum_{n=0}^{G-1} e_k^H(n)Q_k^{-1}e_k(n)} \end{aligned} \tag{26}$$

Aunque el enfoque de dividir la señal en sub-vectores $x(n)$ es inherentemente subóptima, ya que ignora las dependencias intervector, se requiere su utilización con el fin de estimar los parámetros de la señal y la matriz de covarianza de ruido. Tomando el logaritmo de (26), obtenemos la llamada función de log-verosimilitud

$$\begin{aligned} \mathcal{L}(\theta_k) &= \sum_{n=0}^{G-1} \ln p(x_k(n);\theta_k) \\ &= -GM\ ln\ \pi - G \ln \det(Q_k) - \sum_{n=0}^{G-1} e_k^H(n)Q_k^{-1}e_k(n) \end{aligned} \tag{27}$$

las estimaciones de máxima verosimilitud de los parámetros θ_k son entonces

$$\hat{\theta}_k = \arg\max \mathcal{L}(\theta_k) \tag{28}$$

que se encuentran mediante la estimación del vector de ruido como

$$\hat{e}_k(n) = x_k(n) - \hat{s}_k(n) \tag{29}$$

donde la parte determinista del modelo de señal $s_k(n)$ se construye a partir de los parámetros estimados.

La principal dificultad en la evaluación de esta función de coste es que la frecuencia fundamental es un parámetro no lineal y que la matriz de covarianza de ruido Q_k es generalmente desconocida. Las amplitudes y fases, por otro lado, se puede ver en (23) y son parámetros lineales complejos que se pueden encontrar fácilmente dada la frecuencia fundamental y la matriz de covarianza de ruido. Dada la matriz de covarianza de ruido Q_k que va a ser invertible, las amplitudes complejas a_k se pueden encontrar, para una determinada frecuencia fundamental, utilizando el principio de mínimos cuadrados ponderados (WLS) como

$$\hat{a}_k = \left(\sum_{n=0}^{G-1} Z_k^H(n) Q_k^{-1} Z_k(n) \right)^{-1} \sum_{n=0}^{G-1} Z_k^H(n) Q_k^{-1} x(n) \tag{30}$$

donde existe la inverso para todo Z_k, es decir, para $L_k \leq M$.

4.3.2. Estimación de la matriz de covarianza de ruido.

A continuación, abordaremos cómo estimar la matriz de covarianza de ruido de una manera eficiente. Definiendo$Z_k = Z_k(0)$ y suponiendo que las fases de los armónicos

son independientes y están distribuidas de manera uniforme en el intervalo $(-\pi, \pi]$, la matriz de covarianza de la señal de $R_k \in \mathbb{C}^{MxM}$ en (23) se puede escribir

$$R_k = E\{x_k(n)x_k^H(n)\} = Z_k P_k Z_k^H + Q_k \tag{31}$$

donde

$$P_k = E\{a_k a_k^H\} = diag\left(\left[A_{k,1}^2 \cdots A_{k,L_k}^2\right]\right) \tag{32}$$

La matriz de covarianza de ruido $Q_k$va a ser normalmente desconocida y por lo que tendremos que obtener una estimación de la misma. Aquí, nosotros vamos a hacer esto basándonos en el modelo de covarianza de señal en (31). En la práctica, la matriz de covarianza de la señal también será desconocida y se sustituirá por una estimación, que denomina matriz de covarianza de la muestra y está definida como

$$\hat{R}_k = \frac{1}{G}\sum_{n=0}^{G-1} x_k(n)x_k^H(n) \tag{33}$$

Hay algunas ventajas y desventajas inherentes en la elección del valor de los parámetros$N$ y M y por lo tanto también de G. El número de observaciones N debe elegir adecuadamente, de tal manera que la señal de $x_k\ (n)$ se pueda suponer que es estacionaria, mientras que M debe ser elegido de tal manera que todas las correlaciones significativas que sean distintas de cero en la matriz de covarianza se modelaran. Por otro lado, M no debe ser superior a lo estrictamente necesario, ya que la calidad de la estimación de matriz de covarianza de la señal en (33) depende de que el valor de G sea tan alto como sea posible. Adicionalmente, puesto que la evaluación de la función de verosimilitud requiere la existencia y el cálculo de la inversa de la matriz de covarianza de ruido, se requiere que

el rango de $(Q_k) = M$. Esto a su vez implica que $G \geq M$ y por lo tanto que $M \leq \frac{N}{2}$. Para una determinada frecuencia fundamental individual, podemos construir la matriz de Vandermonde Z_k en (31). Sin embargo, también necesitaremos una estimación de las amplitudes sinusoidales en P_k para obtener una estimación de covarianza de ruido. Un enfoque acertado será el uso de la matriz de covarianza de señal estimada en lugar de la matriz de covarianza de ruido que es el resultado de un estimador de amplitud Capon. Una estimación simple y asintóticamente eficiente de las amplitudes complejas para la aproximación dela probabilidad de diferentes frecuencias fundamentales y órdenes es

$$\hat{a}_k = \left(\sum_{n=0}^{G-1} Z_k^H(n) Z_k(n) \right)^{-1} \sum_{n=0}^{G-1} Z_k^H(n) x_k(n) \tag{34}$$

que se puede aproximar para un N grande como

$$\hat{a}_k \approx \frac{1}{MG} \sum_{n=0}^{G-1} Z_k^H(n) x_k(n) \tag{35}$$

Este es un conjunto de transformadas rápidas de Fourier desfasadas (FFTs), ya que la matriz $Z_k(n)$se puede escribir como

$$Z_k(n) = Z_k \begin{bmatrix} e^{j\omega_k n} & & 0 \\ & \ddots & \\ 0 & & e^{j\omega_k L_{kn}} \end{bmatrix} \tag{36}$$

Después de haber encontrado las amplitudes complejas asociadas a una determinada frecuencia fundamental candidata ahora podemos estimar la matriz de covarianza de ruido de orden $L_k$del modelo sinusoidal como

$$\hat{Q}_k = \hat{R}_k - Z_K \hat{P}_k Z_k^H = \hat{R}_k \sum_{l=1}^{L_k} \hat{A}_{k,l}^2 z_l z_l^H \tag{37}$$

con $P_k = diag\langle[\hat{A}_{k,1}^2 \ \cdots \ \hat{A}_{k,l}^2]\rangle$. Puede que sea necesaria la matriz de covarianza de ruido inversa y la inversión directa para diferentes órdenes de modelos y frecuencias fundamentales que posen una significativa carga computacional. Por lo tanto, es ventajoso y acertado calcularlo utilizando el principio de la inversión de la matriz. Esto conduce a la siguiente expresión iterativa para la matriz de covarianza inversa para $l = 1, 2, \ldots, L_k$:

$$\left(\hat{Q}_k^{(l)}\right)^{-1} = \left(\hat{Q}_k^{(l-1)}\right)^{-1} + \left(\hat{Q}_k^{(l-1)}\right)^{-1} \frac{\hat{A}_{k,l}^2 z_l z_l^H}{1 - \hat{A}_{k,l}^2 z_l^H \left(\hat{Q}_k^{(l-1)}\right)^{-1} z_l} \left(\hat{Q}_k^{(l)}\right)^{-1} \tag{38}$$

con $\left(\hat{Q}_k^{(0)}\right)^{-1} = \hat{R}_k^{-1}$. Del mismo modo, el principio del determinante de la matriz se puede utilizar para calcular el determinante de $Q_k^{(l)}$ en l iteraciones de forma recursiva como

$$det\left(\hat{Q}_k^{(l)}\right) = det\left(\hat{Q}_k^{(l-1)} - \hat{A}_{k,l}^2 z_l z_l^H\right) \tag{39}$$

$$= \left(1 - \hat{A}_{k,l}^2 z_l^H \left(\hat{Q}_k^{(l-1)}\right)^{-1} z_l\right) det\left(\hat{Q}_k^{(l-1)}\right) \tag{40}$$

Hay que subrayar que las estimaciones de amplitud utilizadas en la actualización del determinante y las estimaciones de la matriz de covarianza de ruido inversas deben ser elegidas con cuidado, ya que las estimaciones particulares pueden conducir a la deficiencia de rango en algunos casos. Se puede observar que la función de probabilidad contiene el

término $\sum_{n=0}^{G-1} e_k^H(n) Q_k^{-1} e_k(n)$. Si sustituimos los vectores $e_k(n)$ por las estimaciones obtenidas como

$$\hat{e}_k = x_k(n) - \hat{s}_k(n) \tag{41}$$

y la matriz de covarianza de ruido por su $\hat{Q}_k$ estimación, podemos escribir la expresión como

$$\sum_{n=0}^{G-1} \hat{e}_k^H(n) \hat{Q}_k^{-1} \hat{e}_k(n) = Tr\left\{\sum_{n=0}^{G-1} \hat{e}_k^H(n) \hat{Q}_k^{-1} \hat{e}_k(n)\right\} \tag{42}$$

$$= Tr\left\{\hat{Q}_k^{-1} \sum_{n=0}^{G-1} \hat{e}_k^H(n) \hat{e}_k(n)\right\} \tag{43}$$

darse cuenta de que$\sum_{n=0}^{G-1} \hat{e}_k^H(n) \hat{e}_k(n)$ es la matriz de covarianza de ruido reducida de la muestra utilizando los componentes deterministas estimados de la señal$\hat{s}_k(n)$, obtenemos que

$$\sum_{n=0}^{G-1} \hat{e}_k^H(n) \hat{Q}_k^{-1} \hat{e}_k(n) = Tr\{\hat{Q}_k^{-1} \hat{Q}_k G\} = MG \tag{44}$$

lo que significa que este término es constante. La función de verosimilitud, por tanto, sólo depende de la determinante de $\hat{Q}_k$, pero podemos hacer uso de la fórmula de actualización de covarianza de ruido inversa para que sea computación más eficiente de obtener. Cabe señalar que la matriz obtenida a partir $\sum_{n=0}^{G-1} \hat{e}_k^H(n) \hat{e}_k(n)$puede que no sea el inverso exacto de la matriz obtenida utilizando (38), ya que este último se basa en $P_k$debe ser diagonal.

4.3.3. Caso de ruido blanco.

En algunos casos, la parte estocástica del modelo de señal en (1) es de color blanco o aproximadamente blanco. En este caso no sólo puede reducirse la complejidad computacional del estimador de máxima verosimilitud significativamente, sino que además el inherente sub-optimo rendimiento de los métodos basados en matriz de covarianza puede ser evitado. Para el caso de ruido blanco, la estructura de la matriz de covarianza de ruido es ahora conocida, es decir, que se reduce a una matriz escalar diagonal $Q_k = \sigma_k^2 I$ donde σ_K es la varianza del ruido$e_k(n)$. Esto tiene como consecuencia directa que ya no es necesario estimar una matriz de covarianza completa pues ahora sólo nos hace falta la varianza, y, por lo tanto, no hay necesidad de dividir la señal observada en sub-vectores, es decir, podemos simplemente establecer $M = N$ y por lo tanto $G = 1$. Para simplificar la notación, definimos $\hat{e}_k(0) = \hat{e}_k$ y$x_k(0) = x_k$. La función de log-verosimilitud ahora se puede escribir como

$$\mathcal{L}(\theta_k) = -N \ln \pi - N \ln \sigma_k^2 - \frac{1}{\sigma_k^2} \|\hat{e}_k\|_2^2 \tag{45}$$

se puede observar que el estimador de máxima verosimilitud (MLE) es simplemente el minimizador de la 2-norma del error modeladoe_k. Puesto que la frecuencia fundamental es un parámetro no lineal, el método basado en la minimización de la 2-norma se denomina el método no lineal de mínimos cuadrados (NLS, non-linear least-squares). La varianza del ruido también es generalmente desconocida, y por tanto tenemos que tomar una estimación para evaluar la probabilidad logarítmica, y al igual que la estimación de la matriz de covarianza de ruido, esta estimación dependerá de la orden L_k. Para una frecuencia fundamental particular candidata, asumiendo L_k, armónicos la estimación de la varianza del ruido de máxima verosimilitud es

$$\hat{\sigma}_k^2 = \frac{1}{N} \|x_k - \Pi_Z x_k\|_2^2 \tag{46}$$

donde Π_Z es la matriz de proyección definida como

$$\Pi_{Z_k} = Z_k (Z_k^H Z_k)^{-1} Z_k^H \tag{47}$$

La estimación en (46) se obtiene básicamente mediante la sustitución de las amplitudes por las estimaciones de mínimos cuadrados. Observamos que, dadas las frecuencias de los armónicos, las estimaciones de mínimos cuadrados de amplitud se pueden obtener de una manera eficiente por una implementación recursiva. Las sinusoides complejas son por tanto asintóticamente ortogonales para cualquier conjunto de frecuencias distintas, lo que significa que la matriz de proyección Π_Z se puede aproximar como

$$\lim_{n\to\infty} M\Pi_{Z_k} = \lim_{n\to\infty} M\Pi_{Z_k} (Z_k^H Z_k)^{-1} Z_k^H = Z_k Z_k^H \tag{48}$$

Cabe destacar, que para un determinado N, esta aproximación va empeorar progresivamente cuando ω_k se hace más pequeño y puede que sea bastante pobre para una ω_k baja. Usando la aproximación en (48), la estimación de la varianza del ruido se puede simplificar quedando de la siguiente forma

$$\hat{\sigma}_k^2 = \frac{1}{N} \left\| x_k - \frac{1}{N} Z_k Z_k^H x_k \right\|_2^2 \tag{49}$$

A continuación, obtenemos la siguiente función de log-verosimilitud, que sólo depende de ω_k insertando esta estimación de la varianza en (45),

$$\mathcal{L}(\omega_k) \approx -N \ln \pi - N \ln \hat{\sigma}_k^2 - N \tag{50}$$

Vemos que la amplitud se puede encontrar para varias frecuencias fundamentales y órdenes utilizando una FFT, esta vez, sin embargo, estamos ignorando el ruido de color. A partir de (50), se puede observar un problema en relación con el orden del modelo. A medida que aumenta el orden L_k, la log-verosimilitud se incrementa y, por tanto, el estimador de máxima verosimilitud dará lugar a la elección del orden más alto posible. Este fenómeno es ilustrado en la Figura 1, donde (50) se muestra como una función del orden del modelo para una señal sintética con $L_k = 5$.

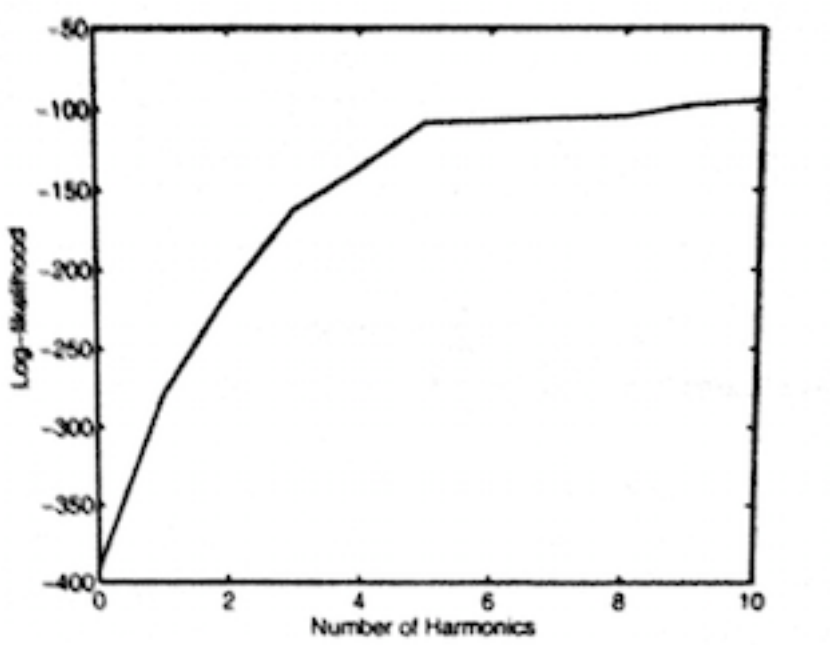

Figura1. Ejemplo del incremento de la log-verosimilitud para una señal periódica.

Escribiendo la función de log-verosimilitud, vemos que para un determinado orden de modelo L_k el estimador se reduce a

$$\omega_k = \arg\max_{\omega_k} x_k^H \Pi_{Z_k} x_k \tag{51}$$

$$\approx \arg\max_{\omega_k} x_k^H Z_k Z_k^H x_k \tag{52}$$

$$= \arg max_{\omega_k} \|Z_k^H x_k\|_2^2 \tag{53}$$

donde las dos últimas líneas siguen la aproximación asintótica, mientras que la primera línea es exacta. En la práctica, el estimador se puede implementar para primera evaluación de la función de coste en una malla basta de la que se puede obtener una estimación inicial. Esta estimación inicial puede ser refinada mediante diversas técnicas de optimización numérica como el algoritmo de máxima pendiente y el método de Newton.

El estimador en (53) se puede implementar eficientemente mediante la formación de una estimación inicial tosca usando una FFT. Para ver esto, se introduce la transformada de Fourier de la señal observada $x_k(n)$como

$$X_k(\omega) = \sum_{n=0}^{N-1} x_k(n) e^{-j\omega n} \tag{54}$$

de manera que la función de coste se puede expresar como

$$\|Z_k^H x_k\|_2^2 = \sum_{l=1}^{L_k} \left| \sum_{n=0}^{N-1} x_k(n) e^{-j\,\omega_k ln} \right|^2 \tag{55}$$

$$= \sum_{l=1}^{L_k} |X_k(\omega_k l)|^2 \tag{56}$$

La cantidad $|X_k(\omega_k l)|^2$ se conoce en términos de estimación teóricos como el periodograma (excepto un factor de escala) cuando se obtienen registros de datos de longitud finita. Se puede observar que el estimador se reduce a una suma sobre la estimación del periodograma de la densidad espectral de potencia evaluado en las frecuencias armónicas. Por otra parte, en la figura 2, se representa la función de coste (56) de una señal real periódica, muestreada a 8 kHz, y cuya frecuencia fundamental es cercana a 205 Hz y que además tiene ruido blanco Gaussiano para dos diferentes órdenes de modelo.

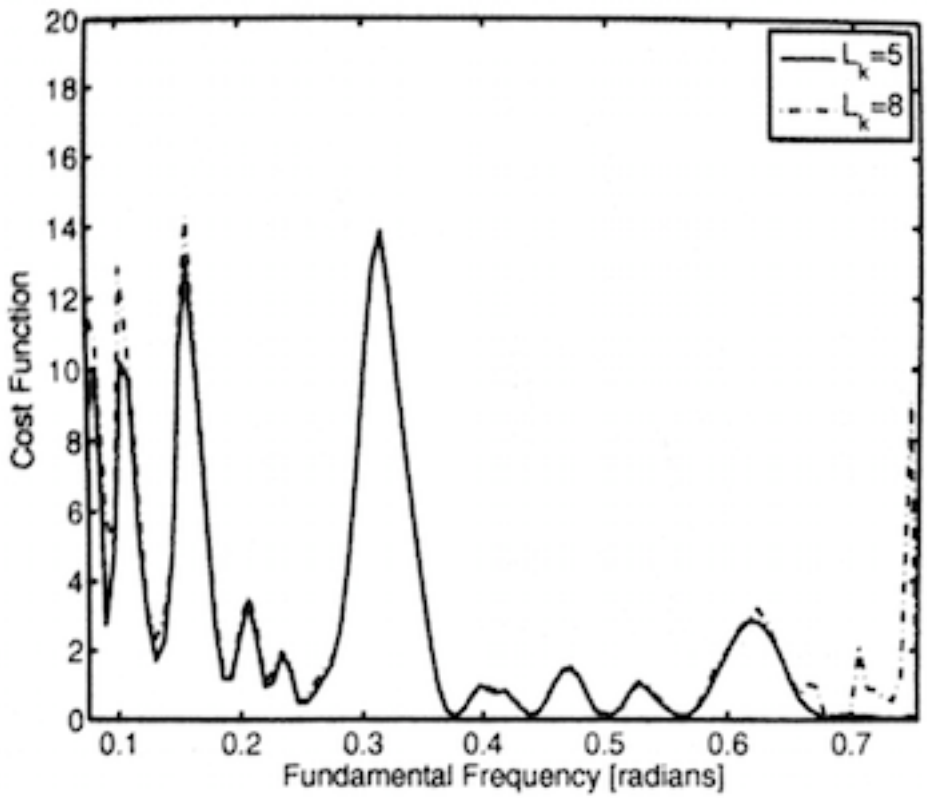

Figura 2. Función de coste de máxima verosimilitud evaluado con distintos ordenes de modelo.

En dicha figura 2 se puede observar distintas de cosas. En primer lugar, para el verdadero orden del modelo, $L_k = 5$, la función de costes tiene un máximo global para el verdadero fundamental en $\omega_k = 0{,}3142$, pero para $L_k = 8$ el máximo global está en la mitad del verdadero fundamental. Esto muestra la importancia de utilizar el orden correcto. En segundo lugar, también puede verse que las funciones de coste son multimodales, lo que significa que tienen múltiples extremos. En la Figura 3, se muestran las funciones de coste para una señal real periódica como la de la figura anterior pero en este caso sin ruido y para una señal periódica real, muestreada a 8 kHz, y cuya frecuencia fundamental es cercana a 165 Hz .

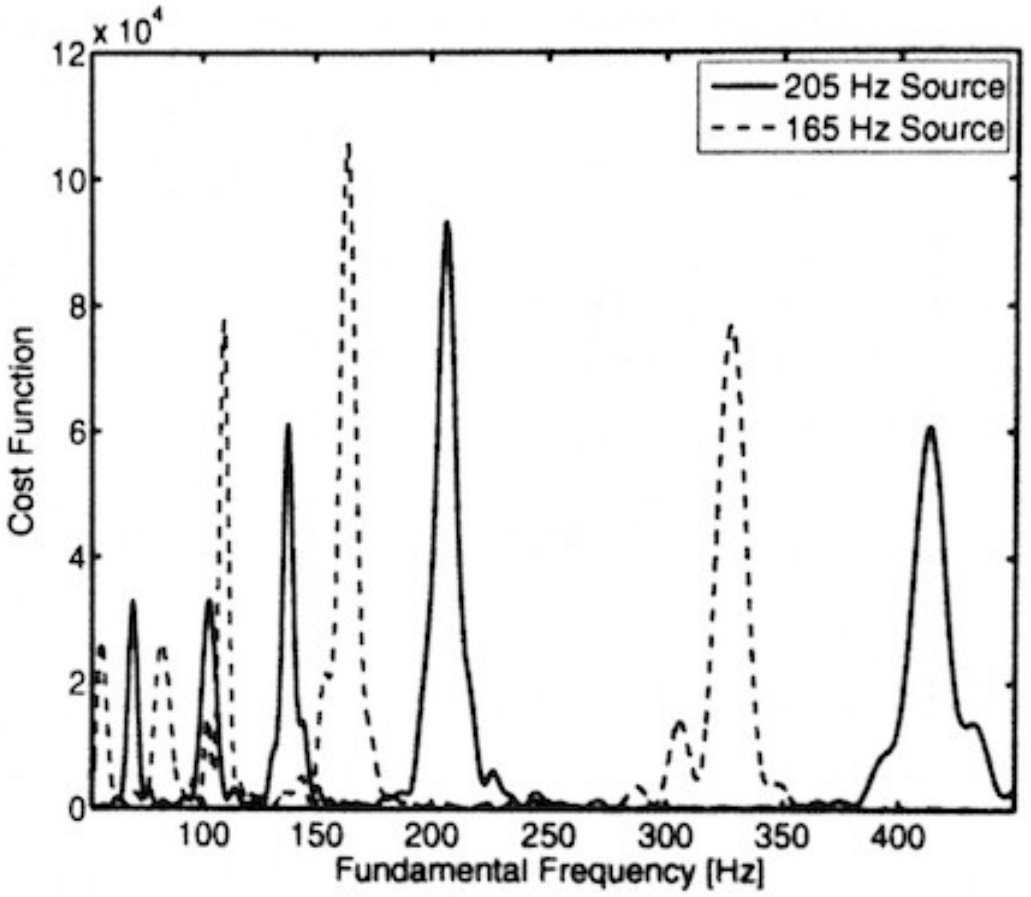

Figura 3. Función de coste de máxima verosimilitud para 2 señales.

Sobre del uso de las aproximaciones asintóticas, cabe señalar que estos pueden que no serán muy precisas para señales de voz, mientras que esto es a menudo un problema menor para señales de música, ya que las señales de música con frecuencia se pueden suponer estacionarias en intervalos de tiempo largos. Hemos de comentar que la aproximación se hace cada vez peor cuanto menos separados están los armónicos en frecuencia. También cabe señalar que un cambio en la frecuencia de muestreo no suele aliviar o mitigar este problema. Sin embargo, la CRLB sugiere que, con respecto a la influencia del ruido, que es, de hecho, beneficioso utilizar una frecuencia de muestreo alta.

Si nosotros queremos aplicar un modelo paramétrico inarmónico al estimador (56), nosotros solo tenemos que sustituir $\omega_k l$ por $\psi_{k,l} = \omega_k l\sqrt{1 + B_k l^2}$ de manera que el estimador queda de la siguiente manera

$$(\widehat{\omega}_k, \hat{B}_k) = \arg\max_{\omega_k, B_k} \sum_{l=1}^{L_k} \left| X_k(\psi_{k,l}) \right|^2 \tag{57}$$

$$= \arg\max_{\omega_k, B_k} \sum_{l=1}^{L_k} \left| \omega_k l \sqrt{1 + B_k l^2} \right|^2 \tag{58}$$

lo que significa que, en principio, tenemos que barrer sobre combinaciones de dos parámetros no lineales para obtener las estimaciones.

Vale la pena señalar que, debido a que el caso de ruido blanco es mucho más simple que el color, el estimador que se presenta aquí puede ser preferido sobre el estimador de ruido coloreado en aplicaciones en tiempo real, incluso si el ruido es conocido y no es completamente blanco, ya que todavía puede proporcionar estimaciones precisas, especialmente si el orden del modelo es conocido, ya que el método de mínimos cuadrados no lineal es asintóticamente eficiente incluso para ruido coloreado.

4.3.4. Estimación rápida multi-pitchs.

Ahora se procederá a presentar un método sencillo y computacionalmente eficiente para la estimación de varios largos en base a principios considerados en los apartados anteriores. Como hemos visto, el estimador de máxima verosimilitud se reduce a minimizar la norma 2 del error modelado para ruido gaussiano blanco. La obtención de la frecuencia fundamental basándonos en este principio es lo que se denomina como método de los mínimos cuadrados no lineal (NLS). Resulta que aun cuando el ruido es de color, el método NLS sigue siendo asintóticamente eficiente lo que significa que alcanza el CRLB para un N grande. Esto significa que para encontrar la frecuencia fundamental, no tenemos por qué preocuparnos demasiado porque el ruido sea coloreado y esto simplifica el problema de manera significativa.

Para mayor comodidad, se define un vector de la señal que contiene todas las N muestras de la señal observada como $x = x(0)$ con $M = N$. Las estimacionese NLS se obtienen como el conjunto de frecuencias y amplitudes fundamentales que minimizan la norma al cuadrado de la diferencia entre vector de señal xy el modelo de señal, es decir,

$$\{\widehat{\omega}_k\} = arg_{\{a_k\},\{\omega_k\}} \min \left\| x - \sum_{k=1}^{K} Z_K a_k \right\|_2^2 \quad (59)$$

donde $\|\cdot\|_2$ denota la norma 2. Suponiendo que todas las frecuencias en $\{Z_k\}$ son distintas y están bien separados y que la N>>1, (59) puede ser bien aproximada mediante la búsqueda de la frecuencia fundamental de las fuentes individuales, es decir,

$$\widehat{\omega}_k = arg_{a_k,\omega_k} \min \| x - Z_K a_k \|_2^2 \quad (60)$$

Las minimizaciones de (59) y (60) son equivalentes sólo cuando las matrices $\{Z_k\}$ son ortogonales. Por lo tanto, con las dos funciones de coste se puede obtener resultados similares para un N grande.

Minimizando (60) con respecto a las amplitudes complejas a_k se obtienen las estimaciones $\hat{a}_k = (Z_K^H Z_k)^{-1} Z_k^H x$, que, cuando insertamos en (60), resulta

$$\widehat{\omega}_k = arg \max_{\omega_k} x^H Z_K (Z_K^H Z_k)^{-1} Z_k^H x \quad (61)$$

$$\approx arg \max_{\omega_k} x^H Z_K Z_k^H x \quad (62)$$

donde la última línea se deriva de la suposición de que el N »1. En la figura 4, se muestra la función de coste del estimador en (62) para la mezcla de dos señales (ambas señales reales periódicas muestreadas a 8 kHz, la primera con frecuencia fundamental en torno a 165 Hz y la segunda en torno a 205 Hz) a las se ha añadido ruido gaussiano blanco aditivo. Las frecuencias fundamentales pueden ser identificadas como los dos picos más grandes, y se observa que las frecuencias fundamentales de las dos fuentes, son, de hecho, encontradas usando este método. Sin embargo, también puede verse que la función de coste es extremadamente complicada si contiene múltiples picos.

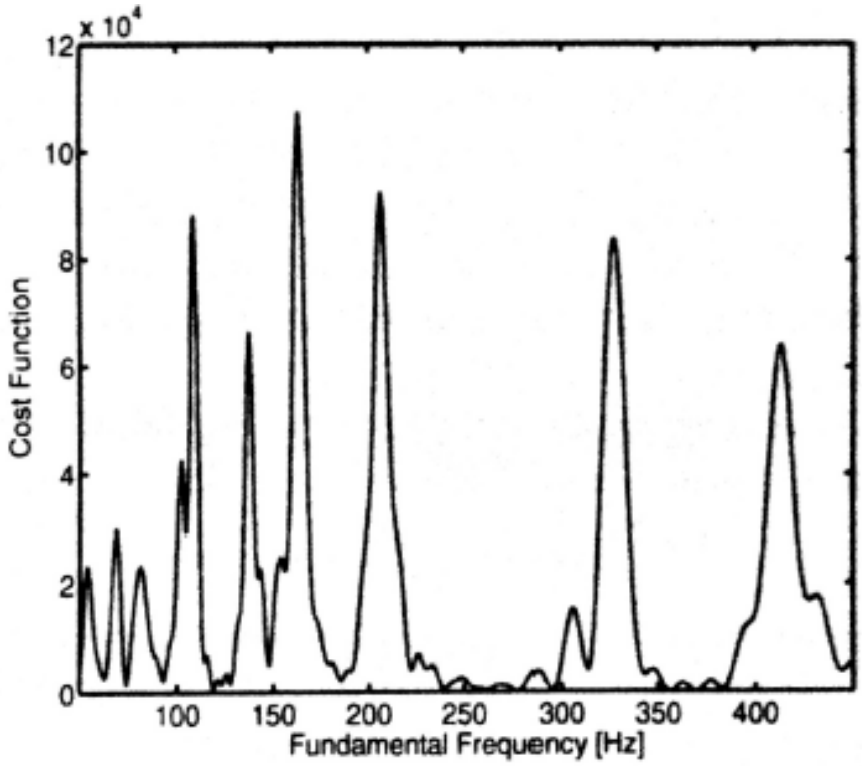

Figura 4. Función de coste de máxima verosimilitud para la mezcla de 2 señales.

El primer estimador de (61) es exacto para el caso de un solo tono, donde x es reemplazado por x_k.

La función de coste resultante puede ser escrita como

$$J(\omega_k) = \|Z_k^H x\|_2^2 \tag{12}$$

donde el producto de la matriz $Z_k^H x$puede ser implementado de manera eficiente para la búsqueda de ω_k utilizando una transformada rápida de Fourier (FFT). El método NLS para el caso de un solo tono puede ampliarse para hacer frente a un orden desconocido y al ruido gaussiano coloreado de una forma computacionalmente eficiente. Una interpretación alternativa del estimador NLS aproximada es la siguiente: la función de coste se puede escribir como $J(\omega_k) = \sum_{l=1}^{L} \|z(\omega_k l)^H x\|_2^2$que es la estimación del periodograma de la densidad espectral de potencia de x evaluada y sumada sobre todas las frecuencias armónicas$\omega_k l$.

Además, observamos que la función de costo NLS en (62) se puede escribir como

$$\|Z_k^H x\|_2^2 = Tr\,[Z_k^H x x^H Z_K] \tag{63}$$

Como una alternativa al uso de la función de coste determinista en (63), se puede tomar en lugar del valor esperado después de sustituir x por el sub-vector x (n), con M <N, en (63), es decir,

$$E\{\|Z_k^H x\|_2^2\} = Tr\,[Z_k^H R Z_K] \tag{64}$$

dando lugar al siguiente estimador de frecuencia fundamental

$$\widehat{\omega}_k = \arg max_{\omega_k} Tr\left[Z_k^H \hat{R} Z_K\right] \tag{65}$$

en lugar de hacer coincidir el modelo de señal para un solo punto de x como en (63) partimos de la matriz de covarianza.

Teniendo de momento en cuenta sólo una fuente, el gradiente de la función de coste en (61) puede ser demostrado como

$$\begin{aligned} \nabla J(\omega_k) &\triangleq \frac{\partial J(\omega_k)}{\partial \omega_k} \\ &= x^H\,[Y_K\,(Z_k^H Z_k)^{-1} Z_k^H + Z_K (Z_k^H Z_k)^{-1} Y_k^H \\ &\quad - Z_k (Z_k^H Z_k)^{-1} W_k (Z_k^H Z_k)^{-1} Z_k^H] x \end{aligned} \tag{66}$$

con $Y_k \in \mathbb{C}^{NxL}$ siendo la derivada de la matriz de Vandermonde con respecto a la frecuencia fundamental cuyos elementos se definen como

$$[Y_K]_{nl} \triangleq \left[\frac{\partial}{\partial \omega} Z_k\right]_{nl} = j(n-1)le^{j\,\omega_k l(n-1)} \tag{67}$$

con $[Y_K]_{lm}$ denotando el (n, l)-ésimo elemento de la matrizY_K. Además, $W_k \in \mathbb{C}^{LxL}$es la derivada de la matriz $Z_k^H Z_k$, es decir,

$$[W_K]_{lm} \triangleq \left[\frac{\partial}{\partial \omega_k} Z_k Z_K\right]_{lm} = \sum_{n=0}^{N-1} (j(l-m)n)e^{j\,\omega_k (l-m)n} \tag{68}$$

El gradiente en (66) se puede utilizar para encontrar las estimaciones refinadas.

Aquí, nos encontramos de forma iterativa tales estimaciones refinadas de la frecuencia fundamental como

$$\widehat{\omega}_k^{(i+1)} = \widehat{\omega}_k^{(i)} + \delta \nabla J\left(\widehat{\omega}_k^{(i)}\right) \tag{69}$$

siendo i el índice de iteración y δ una pequeña y positiva constante que se encuentra de forma adaptativa mediante la búsqueda de línea aproximada. Para la solución aproximada en (61), el gradiente correspondiente es una expresión mucho más simple

$$\nabla \mathrm{J}(\omega_k) \approx 2\, Re(x^H Y_k Z_k^H x) \tag{70}$$

con $Re(\cdot)$ detonando el valor real.

4.4. Otros métodos de estimación.

4.4.1. Método MUSIC.

Se procede a examinar un enfoque del subespacio basado en el principio de ortogonalidad MUSIC. Se ha demostrado que estimaciones de alta resolución en frecuencia y el orden fundamental se pueden obtener con este principio, el enfoque se generaliza para el problema de la estimación de varios largos o pitchs. Revisaremos aquí brevemente estas ideas en el contexto de este trabajo, es decir, para el caso de orden y el número de fuentes conocidas. Suponiendo que las fases de los armónicos son independientes y distribuidos de manera uniforme en el intervalo $[-\pi, \pi]$, la matriz de covarianza y su descomposición de valor propio (EVD) se puede escribir como

$$R = U \wedge U^H = \sum_{k=1}^{K} Z_k P_k Z_k^H + \sigma^2 I \tag{71}$$

donde U está formado por los vectores propios ortonormales M de R, es decir, $U = [u_1 \cdots u_M]$, ∧es una matriz diagonal que contiene los valores propiosλ_k y

$$P_k = diag\left([A_{k,1}^2 \cdots A_{k,L}^2]\right) \tag{72}$$

Sea G el subespacio de ruido formado a partir de los vectores propios correspondientes a los $M - KL$ valores propios de menor peso y observe que

$$rank\left(\sum_{k=1}^{K} Z_k P_k Z_k^H\right) = KL \tag{73}$$

Entonces, se puede demostrar que el subespacio abarcado por el ruido G es ortogonal a todas las matrices de Vandermonde $\{Z_k\}$ que abarcan el subespacio de señal formada por los vectores propios que corresponden a los valores propios más significativos KL. Por lo tanto, el conjunto de frecuencias fundamentales puede ser encontrado como

$$\widehat{\{\omega_k\}} = \arg min_{\{\omega_k\}} \sum_{k=1}^{K} \|Z_k^H G\|_F^2 \tag{74}$$

donde $\|\cdot\|_F$ denota la norma de Frobenius y G se encuentra desde el EVD de la muestra matriz de covarianza $\hat{R}$. Finalmente, definimos la función de costes maximizada para cada fuente individual como

$$J(\omega_k) = -\|Z_k^H G\|_F^2 \tag{75}$$

que puede ser evaluada de manera eficiente mediante el cálculo de la FFT de los vectores propios del subespacio de ruido para cada segmento. El gradiente de la función de coste (75) puede ser demostrado como

$$\nabla \mathrm{J}(\omega_k) = -2\, Re(Tr\{Z_k^H G G^H Y_k\}) \tag{76}$$

con $Y_k \in \mathbb{C}^{MxL}$ que tiene elementos definidos como en (68). Debido a la simplicidad de la función de coste MUSIC, el Hessiano se deriva fácilmente como

$$\nabla^2 J(\omega_k) \triangleq \frac{\partial^2 J(\omega_k)}{\partial \omega_k^2}$$

$$= -2Re(Tr\{Z_k^H G G^H V_k + Y_k^H G G^H Y_k\}) \quad (77)$$

con $V_k \in \mathbb{C}^{MxL}$ siendo la derivada de segundo orden de Z_k, es decir,

$$[V_k]_{nl} \triangleq \left[\frac{\partial^2}{\partial \omega_k^2} Z_k\right]_{nl} = -(n-1)^2 l^2 e^{j\omega_k l(n-1)} \quad (78)$$

El gradiente y el Hessiano se pueden utilizar para la búsqueda de estimaciones refinadas usando el método de Newton, es decir,

$$\widehat{\omega}_k^{(i+1)} = \widehat{\omega}_k^{(i)} - \delta \frac{\nabla J\left(\widehat{\omega}_k^{(i)}\right)}{\nabla^2 J\left(\widehat{\omega}_k^{(i)}\right)} \quad (79)$$

El método se inicializa para i = 0 usando la estimación gruesa de frecuencia fundamental obtenida a partir de (76).

Tenga en cuenta que mientras que el método NLS se basa en una suposición asintótica que facilita la búsqueda de frecuencias fundamentales individuales de forma independiente, no hay tal aproximación en el enfoque MUSIC. La matriz de covarianza de descomposición en el enfoque MUSIC, sin embargo, depende de la distribución de las fases y la blancura, pero no la función de densidad de probabilidad de ruido. El enfoque NLS, por otra parte, depende del ruido es gaussiano, pero todavía es asintóticamente eficiente para el ruido de color. También hay que señalar que, a diferencia de los enfoques Capon y NLS, el enfoque MUSIC requiere un conocimiento a priori sobre el número de fuentes para la evaluación de la función de coste.

4.4.2. Método Capon.

Se procede a introducir un estimador basado en el enfoque Capon, que se basa en el diseño de un conjunto de filtros que pasan potencia no distorsionada a frecuencias específicas, aquí las frecuencias armónicas, minimizando al mismo tiempo la potencia en todas las demás frecuencias. Definiendo la matriz de banco de filtrosH_k^H, que consta de L filtros de longitud M, el problema de diseño de filtros se puede afirmar como el problema de optimización:

$$min_{H_k} Tr\left[H_k^H \hat{R} H_k\right] \qquad sujeto\ a\ \ H_k^H Z_k = I \tag{80}$$

donde I es la matriz identidad L × L. La resolución de la matriz de banco de filtros H_k (80) viene dada por

$$H_k = \hat{R}^{-1} Z_k \left(Z_k^H \hat{R}^{-1} Z_k\right)^{-1} \tag{81}$$

Este banco de filtros depende de los datos y la frecuencia por lo cual se puede para estimar las frecuencias fundamentales mediante la maximización de la potencia de la salida del filtro, es decir, $Tr\left[H_k^H \hat{R} H_k\right]$. Insertando (81) en esta expresión el resultado es

$$\hat{\omega}_k = \arg max_{\omega_k} Tr\left[\left(Z_k^H \hat{R}^{-1} Z_k\right)^{-1}\right] \tag{82}$$

La función de coste puede ser evaluado para diferentesω_k como

$$J(\omega_k) = Tr\left[\left(Z_k^H \hat{R}^{-1} Z_k\right)^{-1}\right] \tag{83}$$

De manera similar al método MUSIC, la complejidad computacional del método Capon puede reducirse algo mediante el cálculo $Z_k^H \hat{R}^{-1} Z_k$ utilizando FFT. El gradiente de la función de coste en Capon (83) se puede obtener como

$$\nabla J(\omega_k) = -2Re\left(Tr\left\{\left(Z_k^H \hat{R}^{-1} Z_k\right)^{-1} \times \left(Z_k^H \hat{R}^{-1} Y_k\right)\left(Z_k^H \hat{R}^{-1} Z_k\right)^{-1}\right\}\right) \tag{84}$$

La matriz $Y_k \in \mathbb{C}^{MxL}$ esta construida como en el método NLS, es decir, tiene elementos definidos como en (67). Como en los casos previos, nos encontramos con una forma iterativa de estimaciones refinadas de la frecuencia fundamental como

$$\hat{\omega}_k^{(i+1)} = \hat{\omega}_k^{(i)} + \delta \nabla J\left(\hat{\omega}_k^{(i)}\right) \tag{85}$$

Alternativamente, el diseño de banco de filtros en (30) se puede formular como el diseño de un único filtro que está sujeto a las limitaciones de L, uno para cada armónico.

4.4.3. Método EM.

Finalmente, se propone un estimador basado en la expectativa de Maximización (EM) algoritmo. El algoritmo EM es un método iterativo para la estimación de máxima verosimilitud por lo cuanto podría ir también incluido en el apartado 4.3. El método que aquí se presenta es un caso especial, que trata de la estimación de los parámetros de las señales superpuestas. En nuestro caso, las señales superpuestas son las fuentes de armónicos. Primero, escribimos el modelo de señal observada en (6) como una suma de K fuentes con ruido blanco aditivo gaussiano, es decir,

$$x = \sum_{k=1}^{K} y_k \tag{86}$$

donde las fuentes individuales se dan como

$$y_k = Z_k a_k + \beta_k w \tag{87}$$

con la fuente de ruido que se descompone de forma arbitraria en K fuentes como$\beta_k w$ donde $\beta_k \geq 0$ se elige de manera que $\sum_{k=1}^{K} \beta_k = 1$.

En el algoritmo EM, el conjunto de vectores y = $\{y_k\}$ se conoce como los datos completos, mientras que los datos observados x se refiere como los datos incompletos. Los datos completos e incompletos se relacionan a través de un mapeo muchos-a-uno. El algoritmo EM consta de dos pasos. La primera, denominada la expectativa o E-step, es el cálculo de la esperanza condicional de la probabilidad logarítmica de los datos completos, es decir,

$$U\left(\theta, \theta^{(I)}\right) = \int \left(In\, p_y(y;\theta)\right) p\left(y|x;\theta^{(i)}\right) dy \tag{88}$$

dónde $\theta^{(i)}$ es un vector que contiene las estimaciones de iteración i-ésimo de los parámetros en (6) y θ es el vector de parámetros desconocidos que parametriza la función de verosimilitud. En el siguiente superíndice (i) denota el número de iteración. Entonces, los parámetros actualizados se encuentran en la denominada maximización o m-paso mediante la maximización de la función anterior, es decir,

$$\theta^{(i+1)} = \arg max_{\theta}\ U\left(\theta, \theta^{(I)}\right) \tag{89}$$

Para el problema que nos ocupa, los dos pasos del algoritmo EM se vuelven particularmente sencillos debido al término de ruido que es Gaussiano y blanco.

Esencialmente, el E-paso se reduce a lo siguiente donde se obtiene una estimación de la k-ésima fuente de ruido en base a los parámetros de la iteración anterior:

$$\hat{y}_k^{(i)} = \hat{Z}_k^{(i)} \hat{a}_k^{(i)} + \beta_k \left(x - \sum_{k=1}^{K} \hat{Z}_k^{(i)} \hat{a}_k^{(i)} \right) \tag{90}$$

donde $\hat{Z}_k^{(i+1)}$ es la matriz Vandermonde construido a partir de la estimación de la frecuencia fundamental $\widehat{\omega}_k^{(i)}$. El problema de la estimación de las frecuencias fundamentales a continuación, se convierte

$$\widehat{\omega}_k^{(i+1)} = \arg max_{\omega_k{}^{(i+1)}} \hat{y}_k^{(i)H} Z_k (Z_k^H Z_k)^{-1} Z_k^H \hat{y}_k^{(i)} \tag{91}$$

$$\approx \arg max_{\omega_k{}^{(i+1)}} \hat{y}_k^{(i)H} Z_k Z_k^H \hat{y}_k^{(i)} \tag{92}$$

y las amplitudes que se necesitan para formar las estimaciones de fuentes en (90) se puede encontrar como

$$\hat{\hat{a}}_k^{(i+1)} = \left(\hat{Z}_k^{(i+1)H} \hat{Z}_k^{(i+1)} \right)^{-1} \hat{Z}_k^{(i+1)H} \hat{y}_k^{(i)} \tag{93}$$

El paso M en (92) se puede ver que va a ser idéntico al método NLS, con la excepción de que (92) opera en la fuente estimada $\hat{y}_k^{(i)}$ en lugar de la señal observada x.En consecuencia, las estimaciones refinadas se pueden obtener en este marco, utilizando un gradiente que recuerda a la del método NLS. El E-paso (90) y el paso M (92) se repiten hasta que se cumple algún criterio de convergencia. Como se puede ver, el algoritmo EM divide el problema de estimación de varios pitchs que en principio es complejo en un

número de problemas de estimación mucho más simples mediante la estimación de las fuentes individuales. En cada iteración del algoritmo, el logaritmo de la verosimilitud de los datos observados se incrementa y el algoritmo garantiza la convergencia, al menos para un máximo local, en condiciones suaves. La principal dificultad en el uso del algoritmo EM es la obtención de las estimaciones iníciales de los parámetros necesarios para estimar las fuentes individuales en (90). En la Figura 5 se pueden observar los resultados de aplicar los 3 métodos.

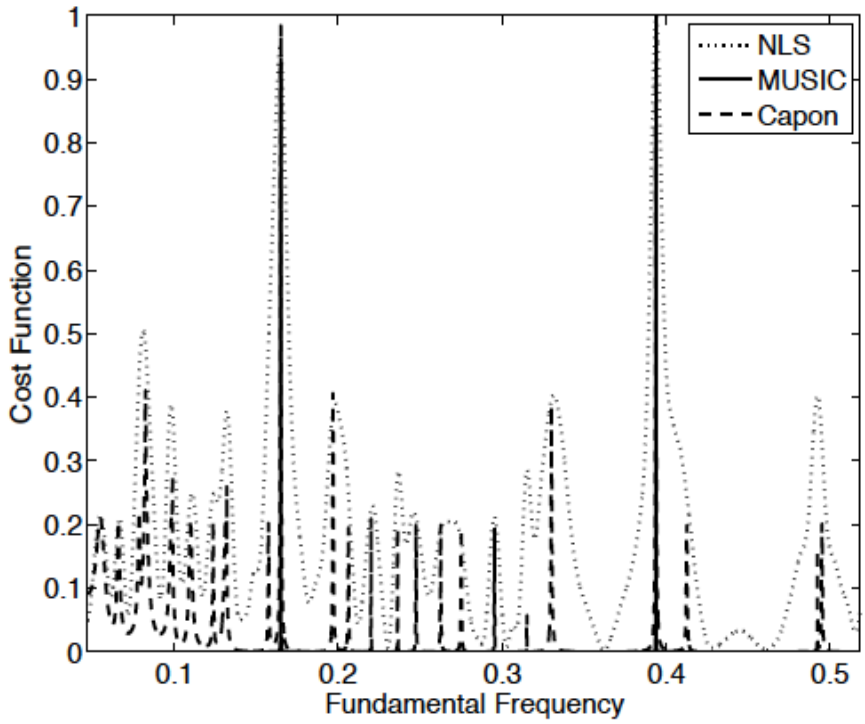

Figura 5. Ejemplo de funciones de costes (escalados por conveniencia) para dos fuentes sintéticas que tienen cinco armónicos cada uno y verdaderas frecuencias fundamentales de $\omega_1 = 0{,}1650$ *y* $\omega_2 = 0{,}3937$ *para* $N = 160$ *y PSNR = 40 dB.*

4.5. Sistema operativo IOS.

4.5.1. Historia.

iOS (*iPhone Operating System*) es un sistema operativo móvil de la empresa Apple Inc.

Originalmente desarrollado para el iPhone (iPhone OS), siendo después usado en dispositivos como el iPod Touch, iPad y el Apple TV. Apple, Inc. no permite la instalación de iOS en hardware de terceros. El sistema operativo gestiona el hardware del dispositivo y proporciona las tecnologías necesarias para implementar aplicaciones nativas. El sistema operativo también viene con varias aplicaciones del sistema, tales como teléfono, correo, y Safari, que proporcionan servicios estándar del sistema para el usuario.

La interfaz de usuario de iOS está basada en el concepto de manipulación directa, usando gestos multitactiles. Los elementos de control consisten de deslizadores, interruptores y botones. La respuesta a las órdenes del usuario es inmediata y provee de una interfaz fluida. La interacción con el sistema operativo incluye gestos como deslices, toques, pellizcos, los cuales tienen definiciones diferentes dependiendo del contexto de la interfaz. Se utilizan acelerómetros internos para hacer que algunas aplicaciones respondan a sacudir el dispositivo (por ejemplo, para el comando deshacer) o rotarlo en tres dimensiones (un resultado común es cambiar de modo vertical al apaisado u horizontal).

iOS se deriva de Mac OS X, que a su vez está basado en Darwin BSD, y por lo tanto es un sistema operativo Tipo Unix.

Figura 6. Logo iOS 8

4.5.2. Arquitectura de iOS.

La arquitectura o diseño de iOS se podría describir con la siguiente frase: iOS son capas. Al más alto nivel, iOS actúa como intermediario entre el hardware subyacente y las aplicaciones que se crean para el mismo.

Las aplicaciones no se van a comunicar directamente con el hardware subyacente. En su lugar, se comunican con el hardware a través de un conjunto de interfaces de sistema bien definidas. Estas interfaces facilitan la tarea de escribir aplicaciones que pueden trabajan en los dispositivos los cuales tienen diferentes capacidades de hardware.

La aplicación de las distintas tecnologías de iOS puede ser visto como un conjunto de capas, que se muestran en la Figura 7. Las capas inferiores contienen servicios y tecnologías fundamentales mientras que las capas de nivel superior se van a basar en las capas más bajas y proporcionan servicios y tecnologías más sofisticadas.

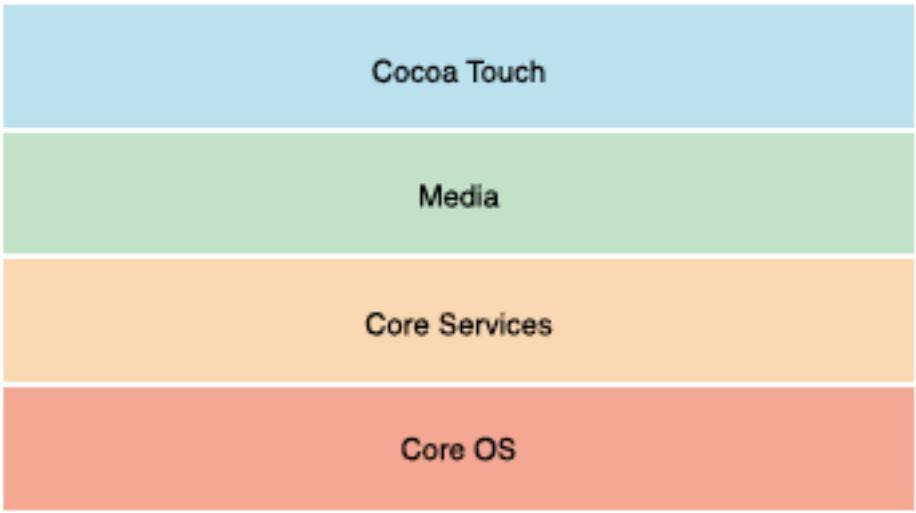

Figura 7. Capas de iOS.

4.5.2.1. Capa Cocoa Touch

La capa de Cocoa Touch contiene marcos o frameworks fundamentales para la construcción de aplicaciones de iOS. Estos marcos van a definir el aspecto de nuestra aplicación. También proporcionan la infraestructura básica de una aplicación y el apoyo a tecnologías clave como la multitarea, la agregación de un contacto, las notificaciones push, y muchos servicios de alto nivel del sistema. Al diseñar una aplicación, debemos investigar las tecnologías de esta capa para ver si se ajustan a sus necesidades.

Algunas de las principales tecnologías disponibles en la capa de Cocoa Touch son las siguientes:

- Airdrop.
- Text Kit.
- UIKit Dynamics.
- Multitarea.
- Auto Layout.
- Storyboards.
- UI State Preservation.
- Apple Push Notification Service.
- Local Notifications.
- Gesture Recognizers.
- Standard System View Controllers.

4.5.2.2. Capa Media.

La capa Media o de los medios de comunicación contiene todo lo referente a gráficos, audio y tecnologías de vídeo que se puede utilizar en nuestras aplicaciones. Las tecnologías de esta capa hacen que sea más fácil para nosotros construir y desarrollar aplicaciones que se vean bien y que tengan una buena calidad de audio.

A continuación vamos a ver los tipos de tecnologías que podemos encontrar en esta capa diferenciando si se trata de audio, video o gráficos.

- Gráficos. Los gráficos de alta calidad son una parte importante de todas las apps y iOS proporciona numerosas tecnologías para ayudar a mostrar gráficos de alta calidad en la pantalla de nuestro dispositivo. Las tecnologías de gráficos que iOS ofrece trabaja a la perfección con la arquitectura de vista UIKit para poder hacer más fácil la entrega de contenido. Podemos utilizar las vistas estándar para ofrecer rápidamente una interfaz de alta calidad, o podemos crear nuestras propias vistas personalizadas y utilizar cualquiera de las tecnologías que figuran en la Tabla 1 para ofrecer una experiencia gráfica aún más rica.

Tecnología	Descripción
UIKit	UIKit define una herramienta de alto nivel para la elaboración de

Graphics	imágenes y animaciones que podremos incluir en nuestras vistas. Además también ofrece clases para implementar el soporte de dibujo. Las vistas UIKit proporcionan una manera rápida y eficaz para hacer uso de imágenes y contenido basado en texto. Las vistas también se pueden animar, como el uso de la dinamics UIKit, para proporcionar información y promover la interactividad del usuario.
Core Graphics framework	Core Graphics (también conocidos como Quartz) es el motor de dibujo nativo para aplicaciones de iOS y proporciona soporte para convertir imágenes en 2D y vectores. Aunque no es tan rápido como OpenGL ES, este marco es adecuado para situaciones en las que desee hacer formas personalizadas en 2D y imágenes de forma dinámica.
Core Animation	Core Animation (parte del marco de cuarzo Core) es una tecnología fundamental que optimiza la experiencia de animación de sus aplicaciones. UIKit vistas utilizan Core Animation para proporcionar animación a nuestras vistas. Puede utilizar Core Animation directamente cuando desee un mayor control sobre el comportamiento de sus animaciones.
Core Image	Core Image proporciona soporte avanzado para la manipulación de vídeo e imágenes fijas.
OpenGL ES and GLKit	OpenGL ES maneja 2D avanzado y 3D utilizando interfaces de acelerado hardware. Este marco es utilizado tradicionalmente por los desarrolladores de juegos, o por cualquiera que desee poner en práctica una experiencia gráfica envolvente. Este marco le da un control total sobre el proceso de renderizado y ofrece las velocidades necesarias de fotogramas para crear animaciones fluidas. GLKit es un conjunto de clases de Objective-C que proporcionan el poder de OpenGL ES utilizando una interfaz orientada a objetos.
Text Kit and Core Text	Text Kit es una familia de clases de UIKit utilizadas para realizar la gestión de texto. Si su aplicación realiza manipulaciones de texto avanzadas, Text Kit ofrece una perfecta integración con el resto de de vistas de nuestra aplicación. Core texto es un marco de nivel inferior basado en C para el manejo

	de la tipografía y el diseño avanzado.
Image I/O	Imagen de E / S proporciona interfaces para leer y escribir en la mayoría de formatos de imagen.
Assets Librería	La librería assets le permite acceder a fotos, videos y audio de un usuario. Se utiliza este marco en los lugares donde se desea integrar contenidos propios del usuario con la aplicación.

Tabla 1. Tecnologías Graficas.

- Audio. Las tecnologías de audio de iOS funcionan con el hardware subyacente para proporcionar una experiencia de audio rica para los usuarios. Esta experiencia incluye la capacidad de reproducir y grabar audio de alta calidad, manejar contenidos MIDI, y poder trabajar con sonidos incorporados en un dispositivo. Si nuestra aplicación utiliza audio, hay varias tecnologías disponibles que nosotros podremos utilizar. iOS es compatible con muchos estándares de la industria y los formatos de audio de Apple-específicas, incluyendo las siguientes: AAC, Apple Lossless (ALAC), A-law, IMA/ADPCM, Linear PCM, etc.La Tabla 2 enumera estos marcos y describe las situaciones en las que usted puede ser que utilice cada uno.

Tecnología	Descripción
Media Player framework	Este marco de alto nivel proporciona un fácil acceso a la biblioteca de iTunes del usuario y soporte para la reproducción de pistas y listas de reproducción. Utilicemos este marco cuando se quiere integrar audio en nuestra aplicación de forma rápida y cuando no es necesario controlar el comportamiento de reproducción.
AV Foundation	Fundación AV es una interfaz Objective-C para la gestión de la grabación y reproducción de audio y vídeo. Utilizamos este marco para la grabación de audio y cuando se necesita un control detallado sobre el proceso de reproducción de audio.
OpenAL	OpenAL es una tecnología estándar de la industria para la entrega de audio posicional. Los desarrolladores de juegos utilizan con

	frecuencia esta tecnología para ofrecer audio de alta calidad, utilizando un conjunto de interfaces multiplataforma.
Core Audio	Core Audio es un conjunto de marcos que proporcionan interfaces simples y sofisticados para la grabación y reproducción de contenido de audio y MIDI. Este marco es para los desarrolladores avanzados que necesitan un control preciso sobre su audio.

Tabla 2. Tecnologías Audio.

- Video. Las tecnologías de vídeo iOS proporcionan apoyo para la gestión de contenido de vídeo en nuestra aplicación o para la reproducción de streaming de contenido desde Internet. Para los dispositivos con el hardware de grabación adecuado, también puede grabar vídeo e incorporarlo en su aplicación .iOS es compatible con muchos formatos de vídeo estándar de la industria y estándares de compresión. La Tabla 3 enumera las tecnologías que soportan la reproducción de vídeo y la grabación.

Tecnología	Descripción
UIImagePicker Controller	La clase UIImagePickerController es un controlador de vista UIKit para elegir los archivos multimedia del usuario. Usted puede usar este controlador de vista para solicitar al usuario una imagen o un vídeo existente o para permitir al usuario capturar nuevos contenidos.
Media Player	El marco Media Player proporciona un conjunto de interfaces fáciles de usar para la presentación de vídeo. Este marco apoya tanto a pantalla completa y la reproducción de vídeo a pantalla parcial y soporta controles de reproducción opcionales para el usuario.
AV Foundation	Fundación AV ofrece capacidades de reproducción de vídeo y grabación avanzados. Utilice este marco en situaciones donde se necesita un mayor control sobre la presentación o la grabación de vídeo. Por ejemplo, aplicaciones de realidad aumentada podrían utilizar este marco para la capa de contenido de vídeo en directo

	con otros contenidos app-proporcionado.
Core Media	El marco Core Media define los tipos de datos de bajo nivel e interfaces para la manipulación de los medios de comunicación. La mayoría de las aplicaciones no necesitan utilizar este marco directamente, pero está disponible cuando se necesita un control sin precedentes sobre el contenido de vídeo de su aplicación.

Tabla 3. Tecnologías Video.

4.5.2.3. Capa Core Services.

La capa Core Services contiene los servicios fundamentales del sistema para las aplicaciones. Clave entre estos servicios son los marcos o frameworks Core Foundation y Foundation, que definen los tipos básicos que todas las aplicaciones utilizan. Esta capa también contiene las tecnologías individuales para admitir características como la ubicación, iCloud, medios de comunicación social, y la creación de redes.

4.5.2.4. Capa Core OS

La capa Core OS contiene las características de bajo nivel en que la mayoría de las otras tecnologías se basan. Incluso si no utilizamos estas tecnologías directamente en nuestras aplicaciones, lo más probable es que estén siendo utilizados por otros marcos o frameworks. Y en situaciones en las que usted necesita para hacer frente de forma explícita con la seguridad o la comunicación con un accesorio de hardware externo, lo hace utilizando el marco que en esta capa.

4.5.3. Diseño de una aplicación IOS.

No importa qué tipo de aplicación estemos creando, hay algunos patrones de diseño fundamentales y técnicas que nosotros debemos saber antes de empezar a escribir código. En iOS, los frameworks o marcos del sistema proporcionan la infraestructura crítica para

nuestra aplicación y en la mayoría de los casos son la única manera de acceder al hardware subyacente. A su vez, los marcos utilizan muchos patrones de diseño específico y se supone que debemos estar familiarizados con ellos. Por lo tanto, la comprensión de estos patrones de diseño es un importante primer paso para entender cómo el sistema puede ayudarnos a desarrollar nuestra aplicación.

Los patrones de diseño más importantes que se deben saber son:

- "Modelo Vista Controlador" Este patrón de diseño rige la estructura general de nuestra aplicación.
- Delegación. Este patrón de diseño facilita la transferencia de información y datos de un objeto a otro.
- Target acción. Este patrón de diseño se traduce en interacciones del usuario con los botones y controles en el código que su aplicación se puede ejecutar.
- Bloque de objetos. Podemos utilizar bloques para implementar las devoluciones de llamada y código asíncrono.
- Sandboxing. Todas las aplicaciones iOS se colocan en cajas que aíslan el proceso para proteger el sistema y otras aplicaciones. La estructura de la caja afecta a la colocación de los archivos de su aplicación y tiene implicaciones para las copias de seguridad de datos y algunas características relacionadas con la aplicación.

Seguir una gestión de memoria precisa y eficiente es importante cuando estamos desarrollando aplicaciones para iOS. Debido a que las aplicaciones de iOS suelen tener menos memoria utilizable que un ordenador de sobremesa, las aplicaciones tienen que ser agresivas en cuanto a eliminación de objetos que no sean necesarios y ser reticente a la creación de objetos. En primer lugar, las aplicaciones utilizan el compilador Automatic Reference Counting (ARC)para gestionar la memoria eficientemente. Aunque el uso de ARC no es necesario, es muy recomendable. La alternativa es gestionar la memoria por sí mismo mediante la retención de forma explícita y la liberación de objetos.

4.5.3.1. MVC

"Modelo Vista Controlador, o MVC, es un patrón de diseño utilizado en el desarrollo de aplicaciones en iOS. En MVC, cada objeto es un objeto de modelo, un objeto de la vista, o un objeto de controlador.

- Los objetos de vista objetos son visibles para el usuario. Ejemplos de objetos de vista son los botones, campos de texto y controles deslizantes. Los vista objetos conforman la interfaz de usuario de una aplicación..
- Objetos de modelo contienen datos y no saben nada de la interfaz de usuario. Por lo general, los objetos de modelo están modelando cosas reales del mundo de usuario. Por ejemplo, cuando se escribe una aplicación para una compañía de seguros, es casi seguro que terminará con un modelo de clase personalizada llamada PolizaSeguro.
- Objetos del controlador son los responsables de una aplicación. Los controladores se encargan de configurar las vistas que el usuario ve y se aseguran de que los objetos de vista y modelo se mantienen sincronizados. En general, los controladores son los encargados de manejar toda la aplicación. Por ejemplo, cuando el usuario selecciona un elemento de una lista, el controlador determina lo que el usuario ve al lado ".

"La figura 8 muestra el flujo de control en una aplicación en respuesta a la entrada del usuario, tal como el usuario pulsando un botón."

"Tenga en cuenta que los modelos y puntos de vista no se comunican entre sí directamente; sino que los controladores son los encargados de dicha comunicación, llevando a cabo la recepción de mensajes de algunos objetos y el envío de instrucciones a los demás ".

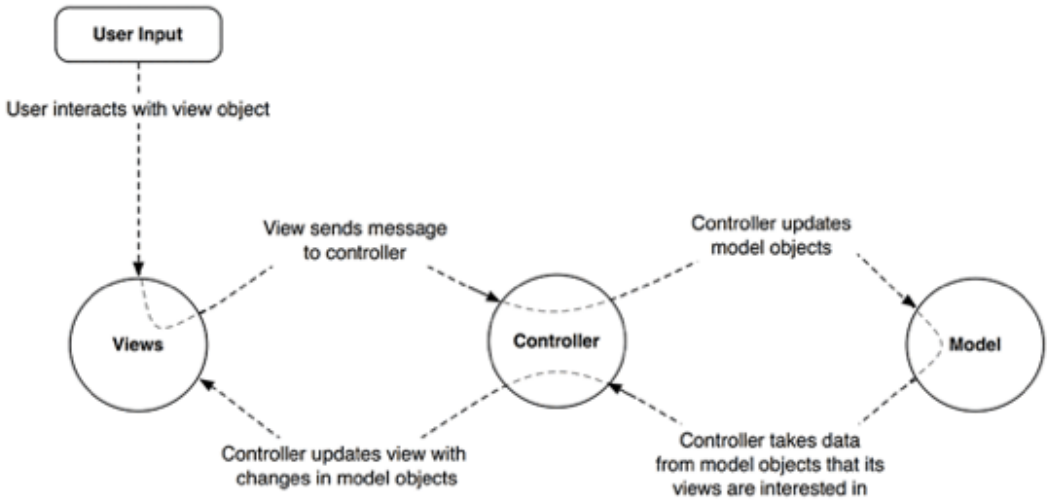

Figura 8. Diseño MVC.

4.5.3.2. Enfoque.

iOS da por sentado que todas las aplicaciones se construyen utilizando el patrón de diseño Modelo-Vista-Controlador. Por lo tanto, el primer paso que puede tomar para lograr este objetivo es elegir un enfoque para las porciones de datos y para la interfaz de su aplicación.

Elección un enfoque básico para su modelo de datos:

- Código de modelo de datos existentes. Si usted ya tiene código de modelo de datos escrito en un lenguaje basado en C, puede integrar ese código directamente en sus aplicaciones de iOS. Debido a que las aplicaciones de iOS están escritos en Objective-C, que funcionan bien con el código escrito en otros idiomas basados en C. Por supuesto, también se benefician de escribir una envoltura de Objective-C para cualquier código que no sea Objective-C.
- Objetos personalizados del modelo de datos. Un objeto personalizado típicamente combina algunos datos simples (cadenas, números, fechas, direcciones URL, etc) con la lógica de negocio necesaria para gestionar los datos y asegurar su consistencia. Los objetos personalizados pueden almacenar una combinación de valores escalares y punteros a otros objetos. Por ejemplo, el marco de la Fundación define clases para muchos tipos de datos simples y para almacenar colecciones de

otros objetos. Estas clases hacen que sea mucho más fácil definir sus propios objetos personalizados.

- Modelo de datos estructurado. Si sus datos son altamente estructurado, es decir, que se presta para el almacenamiento en una base de datos use Core Data (o SQLite) para almacenar los datos. Core Data proporciona un modelo orientado a objetos simple para la gestión de sus datos estructurados. También proporciona soporte integrado para algunas características avanzadas como deshacer y iCloud.

La elección de un enfoque para la interfaz de usuario:

- Enfoque modular. La forma más fácil de crear su interfaz de usuario es el montaje empleando objetos de vista existentes. Las vistas representan elementos visuales tales como tablas, botones, campos de texto, etc. Utiliza muchos puntos de vista tal cual, pero también puede personalizar la apariencia y el comportamiento de las vistas estándar, según sea necesario para satisfacer sus necesidades. También puede implementar nuevos elementos visuales usando vistas personalizadas y mezclar su opinión libremente con las vistas estándar en su interfaz. Las ventajas de las vistas es que proporcionan una experiencia de usuario consistente y que permiten definir interfaces complejas de forma rápida y con relativamente poco código. En la figura 9 se muestra un ejemplo de una interfaz creada con objetos de vista.

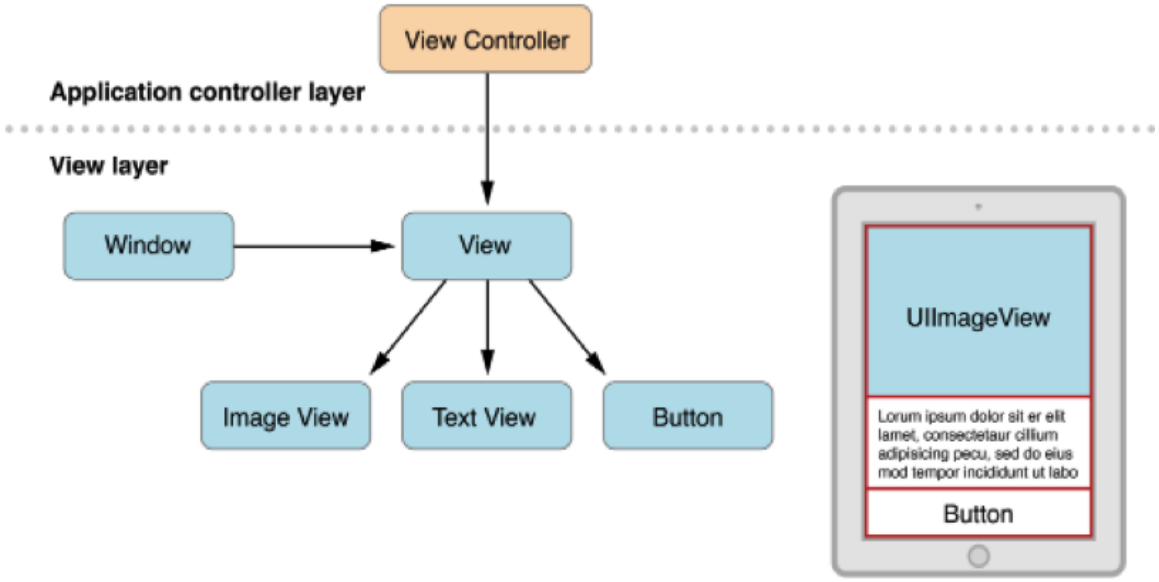

Figura 9. Ejemplo de interfaz con objetos de vista

- Enfoque basado en OpenGL ES. Si su aplicación requiere cambios frecuentes de pantalla o prestación sofisticada, es probable que necesite para sacar ese contenido directamente usando OpenGL ES. El principal uso de OpenGL ES es para los juegos y aplicaciones que dependen en gran medida de gráficos sofisticados, y por lo tanto necesitan el mejor rendimiento posible.

4.6. Frameworks.

En los apartados anteriores hemos hablando de marcos o frameworks sin explicar lo que son, por tanto ahora vamos a describir lo que son.

Un framework o marco es un directorio jerárquico que encapsula los recursos compartidos, como una biblioteca dinámica compartida, archivos nib, archivos de imágenes, cadenas localizadas, archivos de cabecera y documentación de referencia en un solo paquete. Múltiples aplicaciones pueden utilizar todos estos recursos al mismo tiempo. El sistema los carga en la memoria según sea necesario y comparte una copia del recurso entre todas las aplicaciones siempre que sea posible.

Un marco es también un paquete y a su contenido se puede acceder usando Core Foundation Bundle Servicios o la clase Cocoa NSBundle. Sin embargo, a diferencia de la mayoría de los paquetes, un paquete marco no aparece en el Finder como un archivo opaco. Un paquete marco es un directorio estándar por cual el usuario puede navegar. Esto hace que sea más fácil para los desarrolladores examinar el contenido del marco y ver todos los archivos de cabecera y documentación que incluye.

Los marcos o framework tienen el mismo propósito que las bibliotecas compartidas estáticas y dinámicas, es decir, que proporcionan una biblioteca de rutinas que se pueden ser llamadas por una aplicación para realizar una tarea específica. Por ejemplo, el kit de aplicación y los marcos de la Foundation proveen las interfaces de programación para las clases y métodos de Cocoa. Los marcos ofrecen las siguientes ventajas sobre las bibliotecas estáticas vinculadas y otros tipos de bibliotecas compartidas dinámicas:

- Grupo de marcos relacionados, pero separados, tienen los recursos juntos. Esta agrupación hace que sea más fácil de instalar, desinstalar y localizar esos recursos.

- Los marcos o frameworks pueden incluir una variedad más amplia de tipos de recursos que las bibliotecas. Por ejemplo, un marco puede incluir todos los archivos y documentación de cabecera relevantes.
- Varias versiones de un marco se pueden incluir en el mismo paquete. Esto hace posible que sea compatible hacia atrás con los programas más antiguos.
- Sólo una copia de los recursos de sólo lectura de un marco residen físicamente en memoria en un momento dado, independientemente del número de procesos que están usando esos recursos. Esta puesta en común de los recursos reduce el consumo de memoria del sistema y ayuda a mejorar el rendimiento.

La capa de Darwin contiene muchas librerías estáticas y dinámicas pero por lo demás, la mayoría de las interfaces de OS X se empaquetan como los marcos. Algunos marcos de referencia, incluyendo claves de carbono, Cocoa, Servicios de Aplicación, y Core Services proporcionan agrupaciones convenientes de varios marcos más pequeños pero relacionados. Estos grupos de marcos se denominan marcos paraguas y actúan como una capa de abstracción entre una tecnología y los submarcos que implementan esta tecnología. Además de utilizar los marcos del sistema, puede crear sus propios marcos y utilizarlas de manera privada para sus propias aplicaciones o ponerlos a disposición del público para que pueden hacer uso otros desarrolladores. Los marcos privados son apropiados para módulos de código que se van a utilizar en sus propias aplicaciones, pero no desea que otros desarrolladores usen. Los marcos públicos están destinados al uso de otros desarrolladores y por lo general incluyen cabeceras y documentación que definen la interfaz pública del marco.

En OS X, los recursos compartidos se empaquetan utilizando marcos estándar y marcos paraguas. Ambos tipos de marco disponen de la misma estructura básica y pueden contener recursos tales como una biblioteca compartida, archivos nib, archivos de imágenes, archivos de secuencias, listas de propiedades de información, documentación, archivos de cabecera, y así sucesivamente. Un marco paraguas añade pequeñas mejoras a la estructura del marco estándar, tales como la capacidad para abarcar otros marcos.

Los marcos están empaquetados en una estructura de paquete. El directorio de paquete marco termina con la extensión .framework, y a diferencia de la mayoría de otros tipos de lotes, un paquete marco se presenta al usuario como un directorio y no como un archivo. Esta apertura hace que sea fácil para los desarrolladores para navegar por los archivos de cabecera y documentación que se incluyen con el marco.

Después de describir lo que son y para que sirven los frameworks o marcos vamos a estudiar con detenimiento los marcos que son de suma importancia en el funcionamiento de nuestra aplicación, estos son los siguientes: AudioToolbox, AudioUnit, Accelerate, Foundation, CoreGraphics, UIKit..

4.6.1. Accelerate.

Contiene las API de C para manejar vectores y matrices matemáticas, procesamiento de señal digital, la manipulación números de gran tamaño, y el procesamiento de imágenes.

Este framework contiene muchas librerías entre las que destacan la vDSP. La cabecera vDSP proporciona una serie de funciones relacionadas con el procesamiento de señal digital, incluyendo:

- vectorial y matricial aritmética
- transformadas de Fourier
- convolución, correlación, y la generación de ventana

Esta cabecera será la que utilicemos para realizar las fft para el cálculo de las frecuencias fundamentales utilizando el método nlsfast.

4.6.2. AudioUnit.

El marco Audio Unit proporciona interfaces para el uso de procesamiento integrado y audio personalizado plug-ins, conocidas como unidades de audio. En iOS, una aplicación puede utilizar las unidades de audio incorporadas. En OS X, una aplicación puede utilizar cualquier unidad integrada o de audio personalizado, incluyendo las de terceros.

iOS ofrece procesamiento de plug-ins de audio que admite la combinación, ecualización, conversión de formato y para la grabación de entrada/salida de audio en tiempo real, la reproducción, la representación sin conexión y conversación en vivo, como para VoIP (Voz sobre Protocolo de Internet). Usted puede cargar dinámicamente y hacer uso las conocidas como unidades de audio, desde la aplicación de iOS.

Las unidades de audio suelen hacer su trabajo en el contexto de un objeto que encierra un llamado un gráfico de procesamiento de audio, como se muestra en la figura 10.En este ejemplo, la aplicación envía audio a las primeras unidades de audio en el gráfico por medio

de una o más funciones de devolución de llamada y ejerce un control individual sobre cada unidad de audio. La salida de la unidad I / O de la última unidad de audio en este o en cualquier proceso de audio gráfico se conecta directamente al hardware de salida.

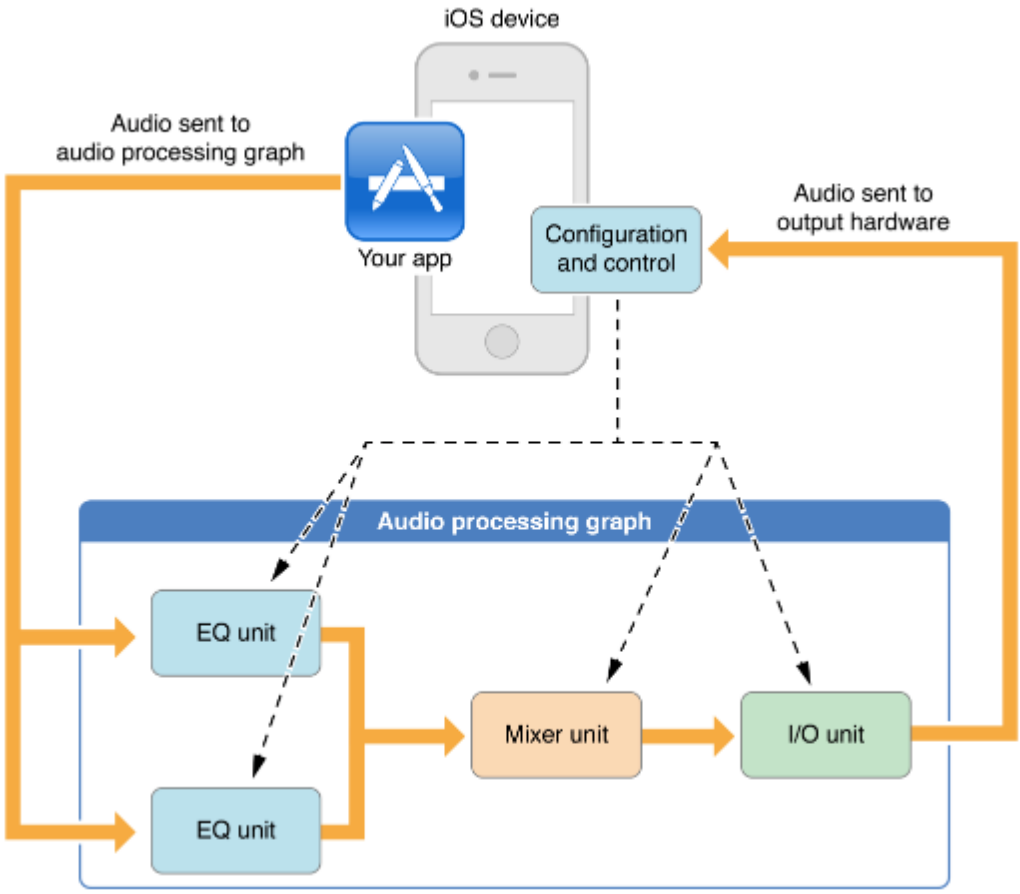

Figura 10. Ejemplo de utilización de Audio Unit.

Las dos mayores ventajas de utilizar directamente las unidades de audio son:

- Excelente capacidad de respuesta. Debido a que tiene acceso a un subproceso de prioridad en tiempo real en un sistema de audio para hacer que la función de devolución de llamada, el código de audio es lo más cerca posible del metal. Instrumentos musicales sintéticos y en tiempo real de voz simultáneos de E / S se benefician más de usar directamente las unidades de audio.
- La reconfiguración dinámica. La API gráfica de procesamiento de audio, construido alrededor del tipo opaco AUGraph, le permite ensamblar

dinámicamente, reconfigurar y reorganizar las cadenas de procesamiento de audio complejas de una manera segura para subprocesos, todo al procesar audio. Esta es la única API de audio en iOS que ofrece esta capacidad.

Ciclo de vida de una unidad de audio procede de la siguiente manera:

1. En tiempo de ejecución, obtener una referencia a la biblioteca de forma dinámica susceptible de enlace que define una unidad de audio que desea utilizar.
2. Crear instancias de la unidad de audio.
3. Configure la unidad de audio con arreglo a su tipo y para dar cabida a la intención de su aplicación.
4. Inicialice la unidad de audio para prepararlo para manejar audio.
5. Iniciar el flujo de audio.
6. Controlar la unidad de audio.
7. Cuando haya terminado, desasignar la unidad de audio.

Las unidades de audio proporcionan características individuales muy útiles como paneo estéreo, mezcla, control de volumen, y la medición del nivel de audio. El mantenimiento de unidades de audio le permite agregar estas características a su aplicación. Para obtener estos beneficios, sin embargo, debe tener facilidad con un conjunto de conceptos fundamentales que incluyen formatos de flujo de datos de audio, hacer llamadas de retorno, y la arquitectura de la unidad de audio.

4.6.3. AudioToolbox.

El marco Audio Toolbox proporciona interfaces para la grabación, reproducción y análisis de flujo. En iOS, el marco ofrece interfaces adicionales para la gestión de sesiones de audio.

4.6.4. CoreGraphics.

El marco Core Graphics es una API basada en C que se basa en el motor avanzado de dibujo Quartz. Provee a bajo nivel, la representación 2D ligera con la fidelidad de salida sin igual. Se utiliza este marco para manejar dibujo basado en transformaciones, gestión

del color, fuera de la pantalla de representación, patrones, degradados y matices, la gestión de datos de imágenes, creación de imágenes, máscaras y creación de documentos PDF, visualización y análisis.

4.6.5. Foundation.

El marco de Foundation define una capa de clases base de Objective-C. Además de proporcionar un conjunto de clases de objetos primitivos útiles, que introduce varios paradigmas que definen la funcionalidad no cubierta por el lenguaje Objective-C. El marco de la Fundación se ha diseñado con estos objetivos en mente:

- Proveer un pequeño conjunto de clases básicas de servicios públicos.
- Hacer más fácil el desarrollo de software mediante la introducción de convenciones consistentes para cosas tales como cancelación de asignación.
- Strings de soporte Unicode, la persistencia de objetos, y la distribución de objetos.
- Proveer un nivel de independencia del sistema operativo, para mejorar la portabilidad.

El marco de Foundation incluye la clase de objeto de raíz, las clases que representan los tipos de datos básicos, tales como cadenas y matrices de bytes, clases de colección para almacenar otros objetos, las clases que representan la información del sistema, como las fechas, y las clases que representan a los puertos de comunicación. En Figura 11 se puede observar lista de las clases que componen el marco de Foundation.

El marco de Foundacion introduce varios paradigmas para evitar confusiones en situaciones comunes, y para introducir un nivel de consistencia a través de las jerarquías de clase. Esta coherencia se realiza con algunas políticas estándar, como para que la propiedad de objetos (es decir, que es responsable de la eliminación de los objetos), y con las clases abstractas como NSEnumerator. Estos nuevos paradigmas reducen el número de casos particulares y excepcionales en un API y le permiten codificar de manera más eficiente mediante la reutilización de los mismos mecanismos con diversos tipos de objetos.

Figura 11. Clases del framework Foundation.

4.6.6. UIKit.

El marco UIKit proporciona las clases necesarias para construir y administrar la interfaz de usuario de una aplicación para iOS. Proporciona un objeto de aplicación, manejo de

eventos, elaboración de modelos, ventanas, vistas y controles diseñados específicamente para una interfaz de pantalla táctil. Figura 12 ilustra las clases en este marco. Este marco es imprescindible en cualquier aplicación iOS que desarrollemos.

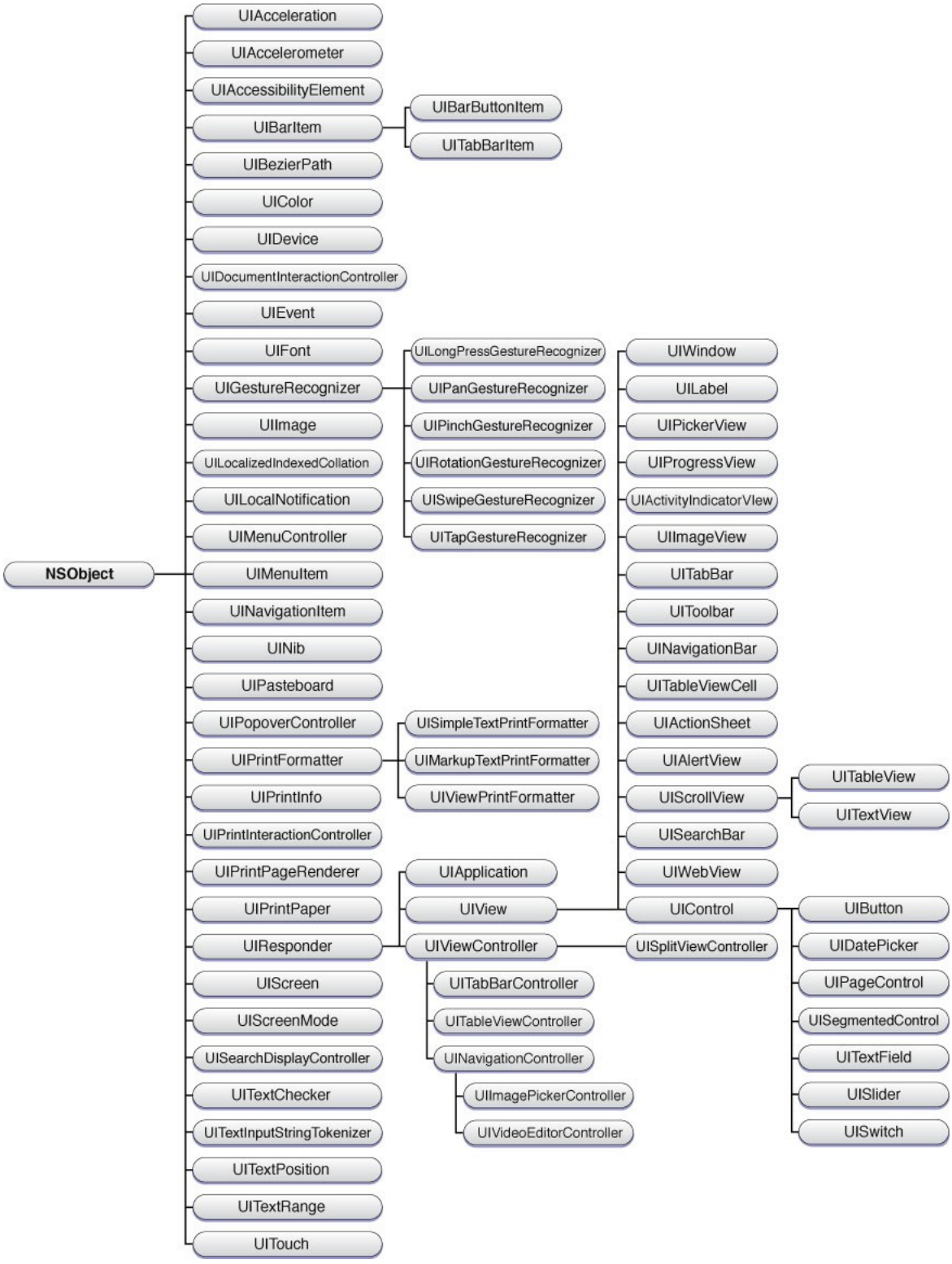

Figura 12. Clases del framework UIKit.

4.7. Elaboración de nuestro afinador.

Una vez que hemos visto y explicado todos los conceptos de procesado de la señal y de programación de objetos ya tenemos todos los conocimientos necesarios para poder llevar a cabo el desarrollo de nuestro afinador. Por cual en esta sección vamos a explicar los puntos que debemos seguir para crear nuestra aplicación.

4.7.1. Diseño y estructura de la aplicación.

La estructura básica de la aplicación consistirá en la siguiente:

- Creación de un buffer que vaya recogiendo datos en tiempo real.
- Elaboración de un método de estimación.
- Aplicación del método de estimación a los datos recogidos en el buffer.
- Obtención de los resultados de aplicar el algoritmo.
- Considerar con dichos resultados si está sonando una nota en un instante concreto de tiempo, para ello se aplicaran unos umbrales de decisión.
- Si se superan dichos umbrales, se mostraran en pantalla las notas que se están tocando.

A pesar que en la estructura de nuestra aplicación hemos puesto que la creación del buffer se desarrollara antes que el método de estimación eso no es así ya que el método es la base de la aplicación por lo cual será lo primero que hemos de desarrollar y aprobar. Además hemos de añadir que el método de estimación que aplicaremos al buffer de datos será previamente diseñado y probado en Matlab y posteriormente lo traduciremos a C para integrarlo en nuestra aplicación, la cual desarrollaremos con software de desarrollo de Apple denominado Xcode.

4.7.2. Creación del buffer de datos.

Para la creación del buffer de datos haremos usos de las frameworks que visto que el apartado 4.6, y lo que habrá que hacer será crear una Audio Queue o cola de audio.

4.7.2.1. Audio Queue.

Una cola de audio es un objeto de software que se utiliza para la grabación o la reproducción de audio en iOS o Mac OS X. Está representado por el tipo de datos opaco AudioQueueRef, declarada en el fichero de cabecera AudioQueue.h.

Una cola de audio hace el trabajo de:

- Conexión a hardware de audio
- Memoria de Gestión
- El empleo de codecs, según sea necesario, para los formatos de audio comprimido
- Grabación o reproducción de audio

Puede usar colas de audio con otras interfaces de Core Audio, y una cantidad relativamente pequeña de código personalizado, para crear una solución completa de audio digital de grabación o reproducción de su aplicación.

4.7.2.2. Audio Queue para grabación.

Una cola de grabación de audio, creada con la función AudioQueueNewInput, tiene la estructura mostrada en la figura 13.

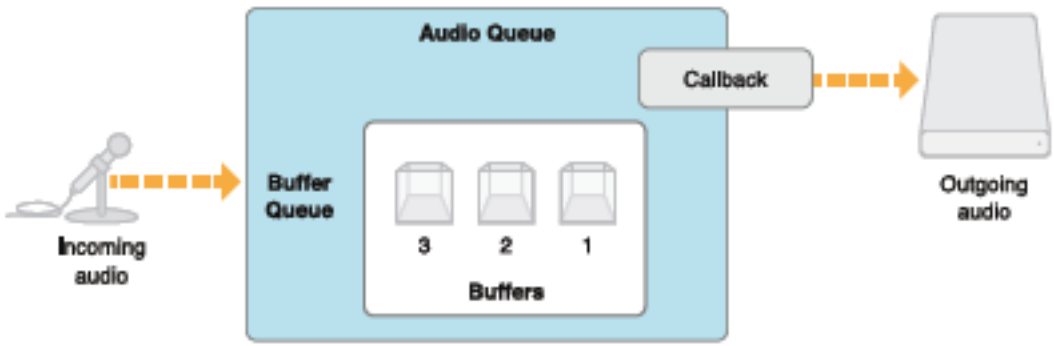

Figura 13. Estructura de una cola de grabación de audio.

En el lado de entrada de una cola de grabación de audio normalmente se conecta a la tarjeta de sonido externa, como un micrófono.

En iOS, el audio del dispositivo proviene del micrófono o de los auriculares con micrófono-usuario incorporado conectados en salida jack de 3.5mm. En el caso por defecto de Mac OS X, el audio proviene de dispositivo de entrada de audio predeterminado del sistema según lo establecido por un usuario en Preferencias del Sistema.

Por el lado de la salida de una cola de grabación de audio hace uso de una función de devolución de llamada que usted escribe. Cuando se graba en el disco, la devolución de llamada, escribe buffers de nuevos datos de audio, que recibe de su cola de audio, un archivo de audio.

Sin embargo, la grabación de colas de audio se puede utilizar de otras maneras. También puede usar una, por ejemplo, en un analizador de audio en tiempo real. En tal caso, la devolución de llamada proporcionaría datos de audio directamente a su aplicación en lugar de escribir en el disco.

4.7.2.3. El proceso de grabación.

Durante la grabación, una memoria intermedia de cola de audio se llena con datos de audio adquirido desde un dispositivo de entrada, tales como un micrófono. Los tampones quedan en la cola de búfer se alinean detrás del búfer en uso, a la espera de ser llenado con los datos de audio a su vez.

La cola de audio devuelve los datos de audio mediante la función de devolución en el orden en que fueron adquiridos.

La figura 14 ilustra cómo funciona la grabación cuando se utiliza una cola de audio.

En el paso 1 de la Figura 14, se inicia la grabación. La cola de audio llena un buffer con los datos adquiridos.

En el paso 2, la primera memoria intermedia se ha llenado. La cola de audio invoca la devolución de llamada, entregando el búfer completo (buffer 1).

La devolución de llamada (paso 3) escribe el contenido de la memoria intermedia en un archivo de audio. Al mismo tiempo, la cola de audio llena otro buffer (buffer 2) con datos recién adquiridos.

En el paso 4, la devolución de llamada encola la memoria intermedia (búfer 1) que acaba de ser escrita en el disco, poniéndola en línea para ser llenada de nuevo.

La cola de audio de nuevo invoca la devolución de llamada (paso 5), entregando la próxima memoria llena (buffer 2).

La devolución de llamada (paso 6) escribe el contenido de este buffer en el archivo de audio.

Este bucle continúa hasta que el usuario detiene la grabación.

No obstante el proceso de grabación de nuestra aplicación no será exactamente así, puesto que los datos de los buffers no se guardaran en archivos de audio sino que se procesaran en tiempo real.

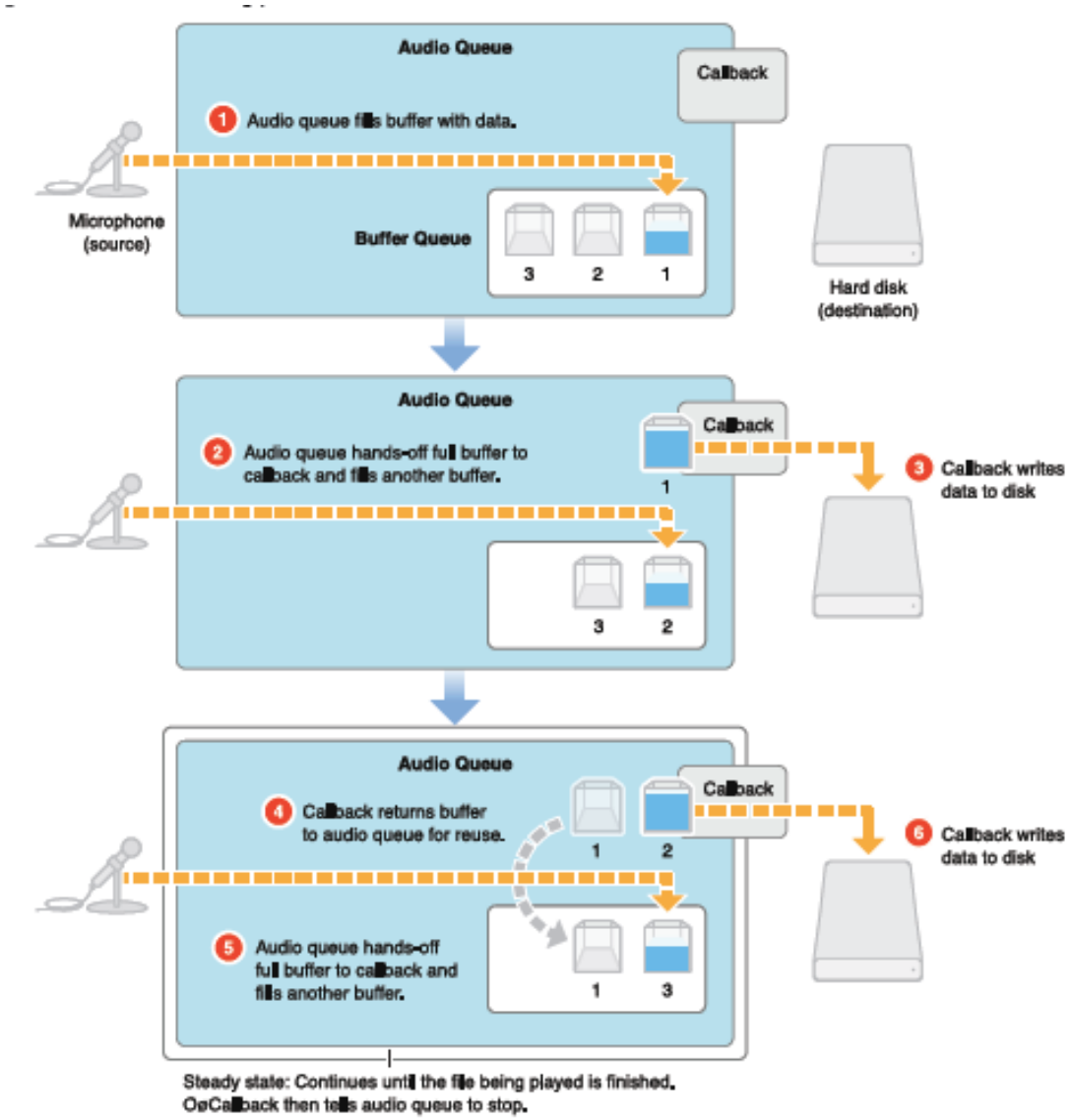

Figura 14. El proceso de grabación.

4.7.2.4. La grabación del audio.

Para agregar la funcionalidad de grabación a su aplicación, por lo general hemos de realizar los siguientes pasos:

1. Definir una estructura personalizada para administrar el estado, el formato y la información de ruta.
2. Escribir una función de devolución de llamada de cola de audio para realizar la grabación real.
3. Escribir un código para determinar un buen tamaño para los buffers de las colas de audio. Este paso es opcional.
4. Rellenar los campos de la estructura. Esto incluye la especificación de los datos de flujo de la cola de audio que envía al archivo de grabación en está, así como la ruta de acceso a ese archivo.
5. Crear una cola de grabación de audio o Audio Queue y pedirle que cree un conjunto de buffers de colas de audio.
6. Llamar a la cola de audio para iniciar la grabación.
7. Cuando haya terminado, decirle a la cola de audio que se detenga pudiendo disponer luego de ella.

Cada uno de estos pasos se pondrá ver como se han desarrollado en el código de la aplicación que está disponible en el capítulo 7 de esta memoria. No obstante en la figura 15 se puede observar un trozo de código de la Audio Queue o cola de audio concretamente de la definición de la estructura personalizada, en la cual se puede apreciar que la cola de audio va a estar formada por tres buffers.

```
static const int kNumberBuffers = 3;

typedef struct AudioStreamData
{
    AudioStreamBasicDescription dataFormat;
    AudioQueueRef queue;
    SInt64 currentPacket;
    AudioQueueBufferRef buffers[3];
    UInt32 bufferByteSize;
    bool mIsRunning;
}AudioStreamData;
```

Figura 15. Ejemplo del código de la Audio Queue.

4.7.3. Estimación de frecuencias.

Para ello habrá que seguir una serie de pasos, los cuales se llevaran a cabo en primer lugar con el software matemático Matlab para posteriormente adaptarlos al lenguaje de programación C y Objective C, las tareas que habrá que llevar a cabo serán las que enumeramos a continuación:

- Implementación del método de estimación
- Adaptación de la señal.
- Ejecución del método de estimación.

4.7.3.1. Implementación del método de estimación.

De todos los métodos que hemos visto en apartado 4.3 y 4.4, en nuestra aplicación vamos a utilizar el método NLS que vimos en el apartado 4.3.4 puesto que con el se podrán obtener unos resultados bastantes aceptables y además de que es computacionalmente eficiente ya que para la estimación de las frecuencias fundamentales hace uso de FFTs.

La función de coste que hay que implementar es la siguiente:

$$\widehat{\omega}_k = \arg max_{\omega_k} Tr\left[Z_k^H \hat{R} Z_K\right] \quad (15)$$

la cual habíamos visto con anterioridad.

Aunque a simple vista parezca algo complicada debido a las matrices de estructura Vandermonde que aparecen en ella, estas matrices lo único que harán serán seleccionar los armónicos de la frecuencia fundamental, por lo cual el estimador únicamente consistirá en el sumatorio al cuadrado de los múltiplos o armónicos de la frecuencia fundamental. En la tabla 4 aparecen los notas de las 6 cuerdas de la guitarra que serán las frecuencias fundamentales. Además hemos de añadir que por facilidad vamos a trabajar en MIDI.

Cuerda	Nota Musical	Nota MIDI	Frecuencia
1ª	E	64	329,63 Hz
2ª	B	59	246,94 Hz
3ª	G	55	196,00 Hz
4ª	D	50	146,83 Hz
5ª	A	45	110,00 Hz
6ª	E	40	82,41 Hz

Tabla 4. Notas de la guitarra (frecuencias fundamentales).

Ya tenemos las frecuencias fundamentales, ahora lo que necesitamos para poder aplicar el método de estimación es calcular los armónicos o múltiplos de las frecuencias fundamentales. Los 9 primeros armónicos de las frecuencias fundamentales o notas de la guitarra en MIDI se pueden ver en la tabla 5.

Frec.Fund.	1 arm.	2 arm.	3 arm.	4 arm.	5 arm.	6 arm.	7 arm.	8 arm.	9 arm.
40	52	59	64	68	71	74	76	78	80
45	57	64	69	73	76	79	81	83	85
50	62	69	74	78	81	84	86	88	90
55	67	74	79	83	86	89	91	93	95
59	71	78	83	87	90	93	95	97	99
64	76	83	88	92	95	98	100	102	104

Tabla 5. Armónicos de las notas en MIDI.

Como se quieren estimar más de una fuente no se podrán coger los armónicos que sean coincidentes por lo cual se cogerán los múltiplos que son distintos de 0 de la Tabla 6. Por lo cual los sumatorios solo se harán con esos múltiplos, pudiendo verse que se utilizar 4 o 5 múltiplos para cada estimación.

Frec.Fund.	1 arm.	2 arm.	3 arm.	4 arm.	5 arm.	6 arm.	7 arm.	8 arm.	9 arm.
40	52	0	0	68	0	0	0	0	80
45	57	0	0	73	0	0	0	0	85
50	62	0	0	0	0	84	0	0	0
55	67	0	0	0	0	89	91	0	0
0	0	0	0	87	0	0	0	97	99
0	0	0	0	92	0	98	100	102	104

Tabla 6. Armónicos validos para la estimación.

Ya que tenemos lo necesario el método se podrá implementar fácilmente en Matlab utilizando un cuantos bucles for y sentencias if-else. Para la implementación del método NLS en Matlab hemos creado una función llamada *nlsfast31* cuya implementación se puede observar en la Figura 16.

```
function [y,xs]=nlsfast31(x)
long=size(x);
xa=zeros(long(1),long(2));
s1=long(1);
s2=long(2);
for i=1:s1,

for j=1:s2;
   if j<=74
       if j==40
        xa(i,j)=power(x(i,j),2)+power(x(i,j+12),2)+power(x(i,j+28),2)+power(x(i,j+40),2);
       elseif j==45
        xa(i,j)=power(x(i,j),2)+power(x(i,j+12),2)+power(x(i,j+28),2)+power(x(i,j+40),2);
       elseif j==50
        xa(i,j)=power(x(i,j),2)+power(x(i,j+12),2)+power(x(i,j+34),2);
       elseif j==55
        xa(i,j)=power(x(i,j),2)+power(x(i,j+12),2)+power(x(i,j+34),2)+power(x(i,j+36),2);
       elseif j==59
        xa(i,j)=power(x(i,j+28),2)+power(x(i,j+38),2)+power(x(i,j+40),2);
       elseif j==64
        xa(i,j)=power(x(i,j+28),2)+power(x(i,j+34),2)+power(x(i,j+36),2)+power(x(i,j+38),2)+power(x(i,j+40),2);
       else
      end
   end
end
end
xs=xa;
y=sum(xa);

end
```

Figura 16. Implementación del método en Matlab.

Una vez que tenemos el método implementado en Matlab lo traduciremos a C para que pueda ejecutarse en nuestra aplicación de afinación, el código del método se podrá ver con detenimiento en el capítulo 7.

4.7.3.2. Adaptación de la señal.

Como de momento estamos trabajando en Matlab no habrá que hacer uso de una Audio Queue o Audio Cola ya que en Matlab trabajaremos con una serie de notas ya grabadas. Por lo tanto para poder trabajar con ellas solo habrá que ejecutar la función wavread para que nos devuelva dicha señal como un vector de muestras.

Pero estando la señal como un vector de muestras no es posible aplicarle el método de estimación que hemos implementado anteriormente por lo que hay que hacer una serie de cambios a la señal, para llevar a cabo esto vamos a utilizar dos funciones en Matlab que

tienen como nombres: *sg1* y *método_rapido_suma*. La implementación de dichas funciones se puede ver a continuación.

Función sg1.

```
function [yo,ys,fo,to] = sg(x,nfft,Fs,win,noverlap)
% S = sg(B,NFFT,Fs,win,NOVERLAP)
% win must be created like win = hanning(Nsamples)
% All parameters like SPECGRAM

nx = length(x);
nwind = length(win);
if nx < nwind    % zero-pad x if it has length less than the win length
    x(nwind)=0;  nx=nwind;
end
x = x(:); % make a column vector for ease later
win = win(:); % be consistent with data set

ncol = fix((nx-noverlap)/(nwind-noverlap));
colindex = 1 + (0:(ncol-1))*(nwind-noverlap);
rowindex = (1:nwind)';
if length(x)<(nwind+colindex(ncol)-1)
    x(nwind+colindex(ncol)-1) = 0;   % zero-pad x
end
y = zeros(nwind,ncol);
y(:) = x(rowindex(:,ones(1,ncol))+colindex(ones(nwind,1),:)-1); %Va
poniendo los datos de x de longitud 1024 (1-1024),(9-1035) hasta formar
1528 columnas
y = win(:,ones(1,ncol)).*y;
y2=y;
y = fft(y,nfft);
ys=y;
if ~any(any(imag(x)))    % x purely real
if rem(nfft,2),    % nfft odd
        select = [1:(nfft+1)/2];
else
        select = [1:nfft/2+1];
end
    y = y(select,:);
else
    select = 1:nfft;
end
f = (select - 1)'*Fs/nfft;
t = (colindex-1)'/Fs;
if nargout == 1,
    yo = y;
elseif nargout == 2,
    yo = y;
    fo = f;
elseif nargout == 3,
    yo = y;
    fo = f;
    to = t;
end
```

Función método_rapido_suma.

```
function
[cfreq_amplitudes,miditobins]=modelo_rapido_suma(X,fs,longitud_frec,n_tra
mas,midi_inc)
```

```
% Inicializaciones

midi_min=24;
midi_max=ceil(12*log2((fs/2)/440)+69);

miditobinskmin = zeros(1,(midi_max-midi_min+1)*midi_inc);
miditobinskmax = zeros(1,(midi_max-midi_min+1)*midi_inc);
for nota_midi=midi_min:midi_max,

for midi_interval = 0:midi_inc-1,

        step_midi = 1/midi_inc;

        fmin=((2^(((nota_midi+midi_interval*step_midi-step_midi/2)-
69)/12))*440);
        kmin=ceil(fmin/fs*(2*longitud_frec)+1);
        kmin=min(kmin,longitud_frec+1);

        fmax=((2^(((nota_midi+midi_interval*step_midi+step_midi/2)-
69)/12))*440);
        kmax=fix(fmax/fs*(2*longitud_frec)+1);
        kmax=min(kmax,longitud_frec);

        miditobinskmin((nota_midi-midi_min)*midi_inc+midi_interval+1) =
kmin;
        miditobinskmax((nota_midi-midi_min)*midi_inc+midi_interval+1) =
kmax;

end;

end;

if midi_inc==1,

miditobinskmax(miditobinskmax<miditobinskmin)=miditobinskmin(miditobinskm
ax<miditobinskmin);
else,
    indval = (miditobinskmax>=miditobinskmin);
    miditobinskmin = miditobinskmin(indval);
    miditobinskmax = miditobinskmax(indval);
    minkmin = min(miditobinskmin); % Lineal hasta el primer midi
    miditobinskmin = [1:minkmin-1 miditobinskmin];
    miditobinskmax = [1:minkmin-1 miditobinskmax];
end;
miditobinskmax(end) = longitud_frec+1;
muestrasmidi = length(miditobinskmin);
miditobins=[miditobinskmin;miditobinskmax];
% ventana=window(@hanning,windowsize)'; %ventana de hanning

% n_tramas=fix((length(x)-windowsize)/salto)+1;

% Inicializacion de variables de salida
cfreq_amplitudes=zeros(n_tramas,muestrasmidi);
cfreq_amplitudes = cfreq_amplitudes';

% Espectrograma
% X=sg(x,2*longitud_frec,fs,ventana,windowsize-salto);

%nota_midi=round(12*log2(frecuencias_Hz/440)+69);

% Calculo del espectrograma en frecuencia midi
for midi_index=1:muestrasmidi,

    kmin=miditobinskmin(midi_index);

    kmax=miditobinskmax(midi_index);
```

```
if (kmax-kmin)==0,
        cfreq_amplitudes(midi_index,:) = abs(X(kmin,:));
elseif (kmax-kmin)>0,
        cfreq_amplitudes(midi_index,:) =
sqrt(sum(abs(X(kmin:kmax,:)).^2,1)); % / (kmax-kmin+1));
end;

end;

% cfreq_amplitudes = cfreq_amplitudes';

return;
```

El cometido de estas funciones es realizar el espectrograma del conjunto de muestras de entrada y sustituir en dicho espectrograma las frecuencias por notas MIDI. Para crear dicho espectrograma se va llevar a cabo los siguientes pasos:

1. Crear una matriz de tramas para la cual hay que realizar los siguientes pasos:
   - Coger el vector de muestras el cual tendrá una gran longitud y dividirlo en sub-vectores de menor longitud con salto determinado.
   - Y cada uno de esos sub-vectores va a ser una la columna de una matriz. En la Figura 17 se puede observar un ejemplo de cómo se elaborara la matriz.

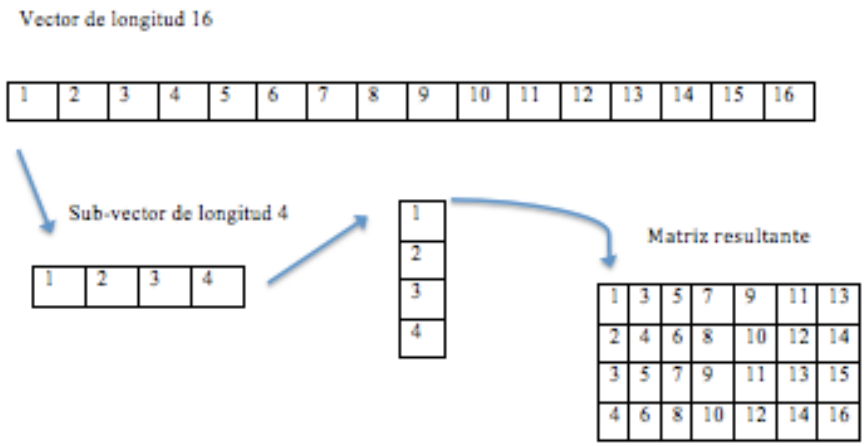

Figura 17. Ejemplo de elaboración de una matriz con salto 2.

   - Las dimensiones de este vector bidimensional o matriz va a ser las siguientes:

$$n\ filas = long\ subvector \tag{94}$$

$$n\ columnas = \left(\frac{long\ vector - long\ subvector}{salto}\right) + 1 \tag{95}$$

El salto dependerá de la frecuencia de muestreo y del tiempo de salto y será igual a

$$salto = f.muestreo * t.salto \tag{96}$$

Hemos de comentar en el En la Figura 18 y 19 se puede ver la forma de la señal a la que queremos aplicarle el método de estimación

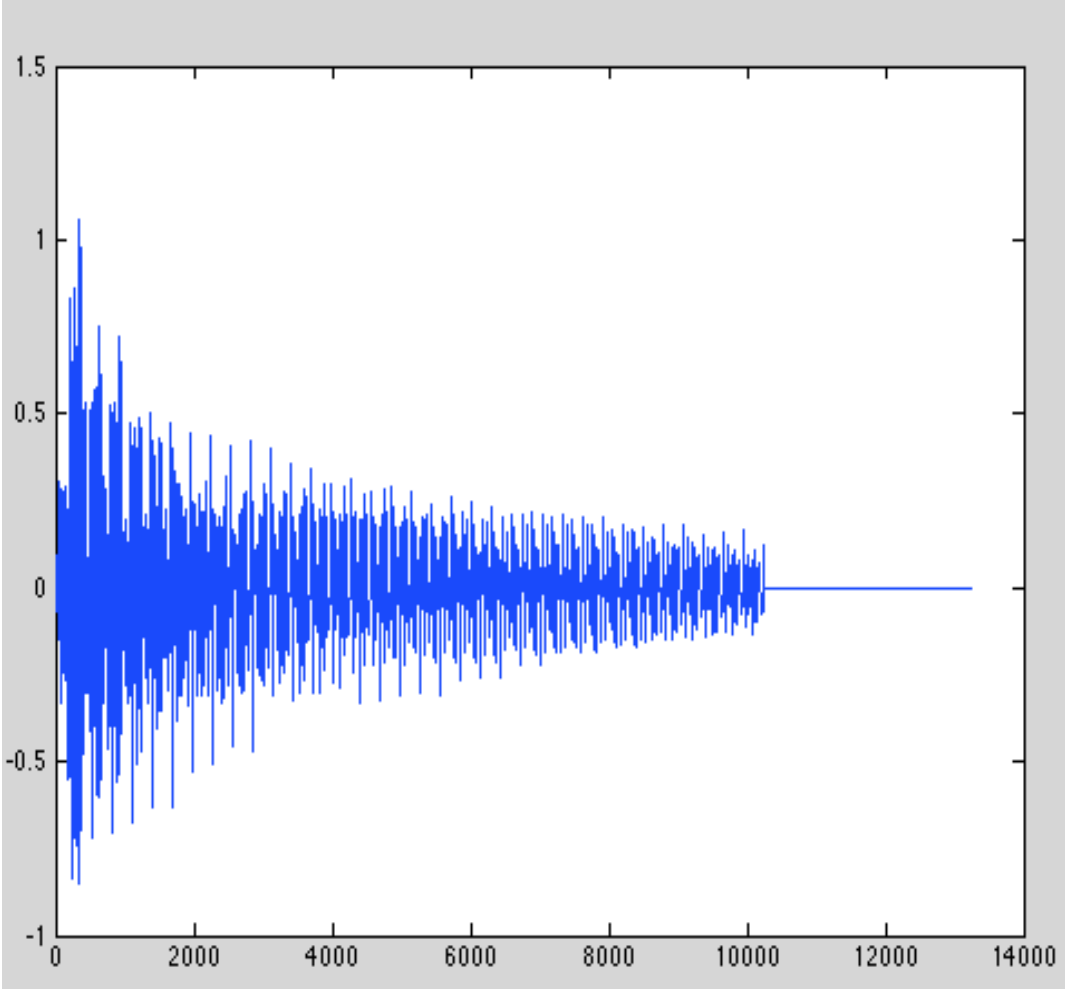

Figura 18. Señal que es suma de las notas E y A, cual vamos a aplicar el método.

2. Realizar la FFT de una longitud determinada al vector bidimensional. En la figura 19 vemos el resultado de aplicar el paso 1 y 2 a la señal de la Figura 18. Una vez aquí ya tenemos el espectrograma pero si representamos como se puede ver en la figura 19 este mostrara el numero de trama en el eje x y frecuencia en el eje y pero lo que queremos es que se muestren notas MIDI por lo cual llevaremos a cabo el paso 3.

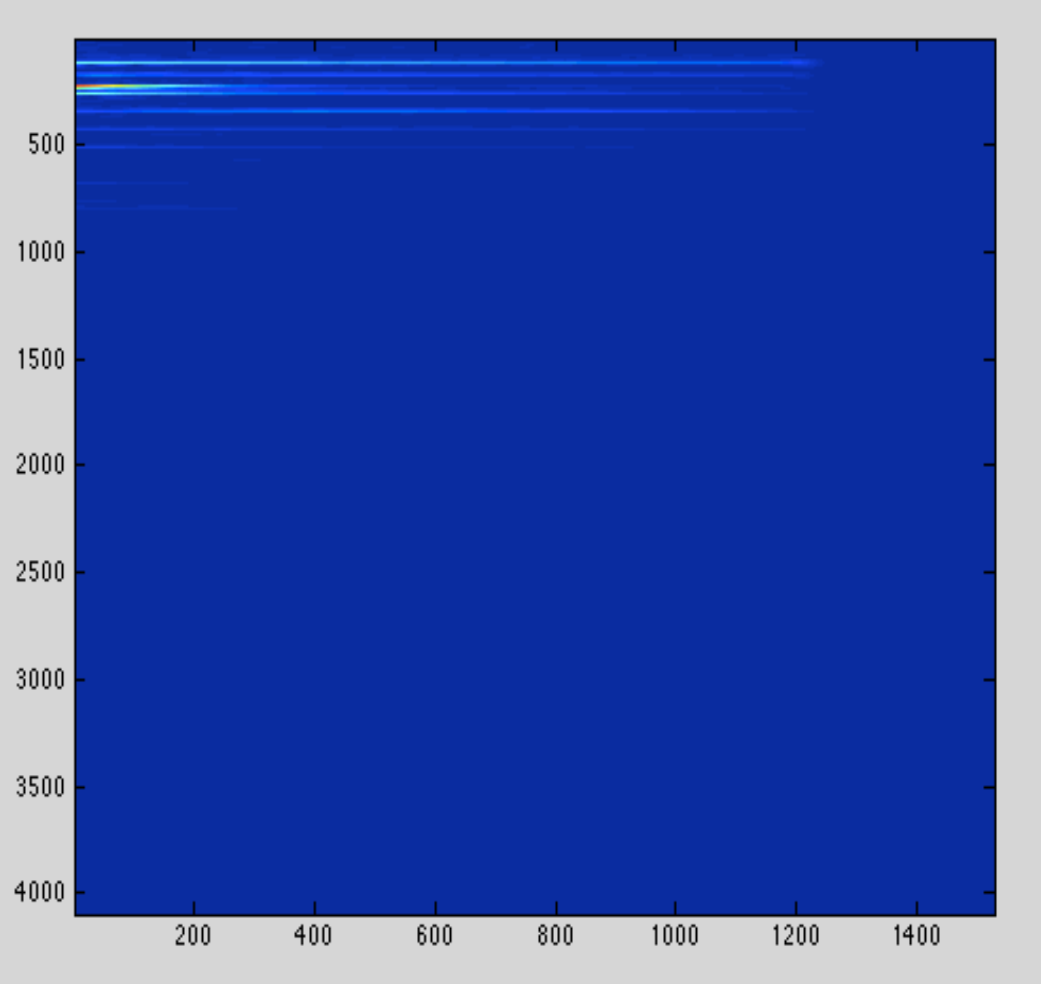

Figura 19. Representación con imagesc de la señal al aplicarle los pasos 1 y 2.

3. El vector bidimensional que obtenemos mostrara los datos en frecuencia lineal por lo que habrá que transformar dicha frecuencia a MIDI.

Después de llevar a cabo estos pasos ya tendremos los datos en el formato apropiado para poder trabajar con ellos con facilidad, es decir, habremos realizado el espectrograma de las muestras y ya poder aplicar el método de estimación que el apartado 4.7.3. hemos implementado en Matlab. Por último en la figura 20 se puede ver espectrograma del conjunto de muestras, listas para poder aplicarle el método de estimación. Al igual que el apartado anterior todo este código elaborado para adaptar la señal se transformara a C para poder utilizar en nuestra app y se podrá ver con detenimiento en el código de la aplicación que se encuentra en el capítulo 7 de esta memoria.

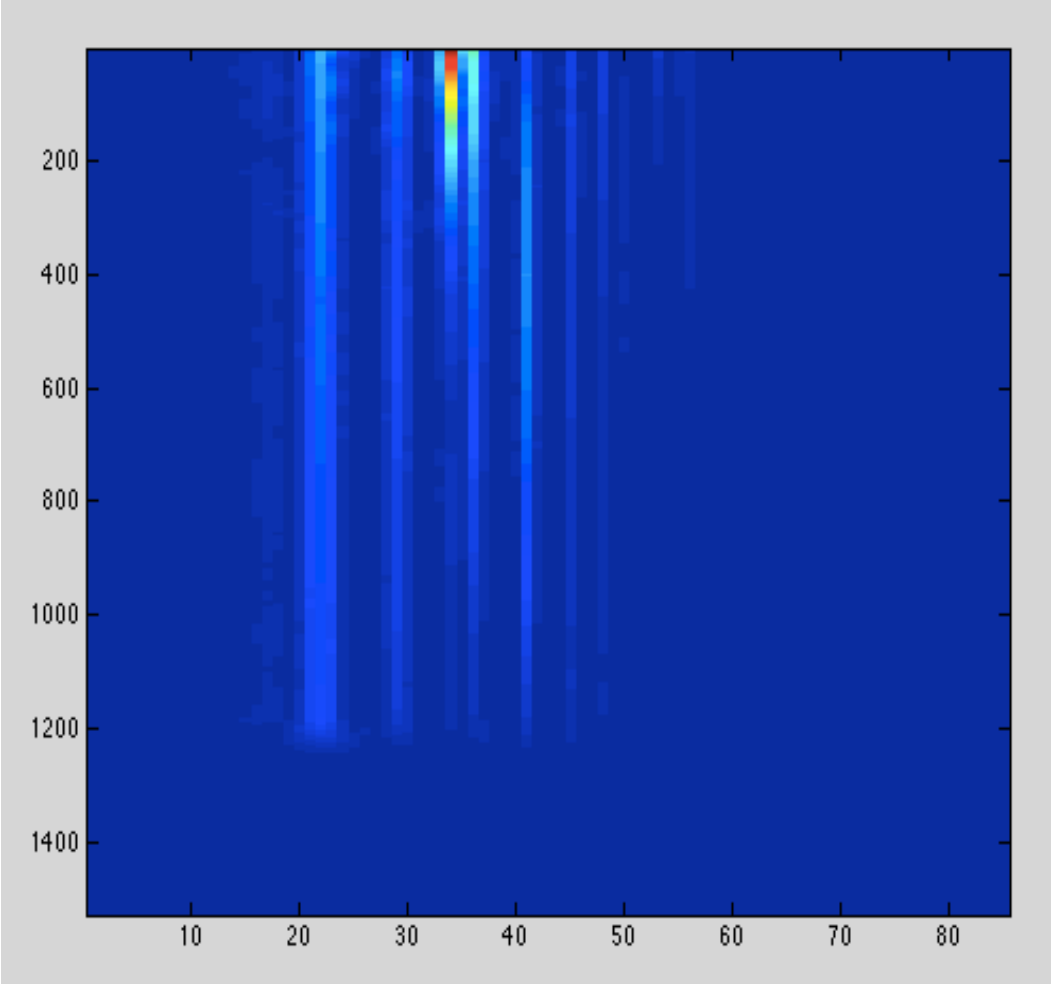

Figura 20. Representación con imagesc de los datos (señal) ya adaptados.

4.7.3.3. Ejecución del método.

Bastara con introducir en el método NLS desarrollado la señal de entrada que es el vector dimensional con el cual hemos estado trabajando en el punto anterior. Una vez introducidos dichos datos en el método de estimación obtendremos una serie de resultados, al representarlos se obtienen las imágenes que se pueden apreciar en la Figura 21 y 22, las cuales van a mostrar cual es la energía que se van tener en cada nota MIDI.

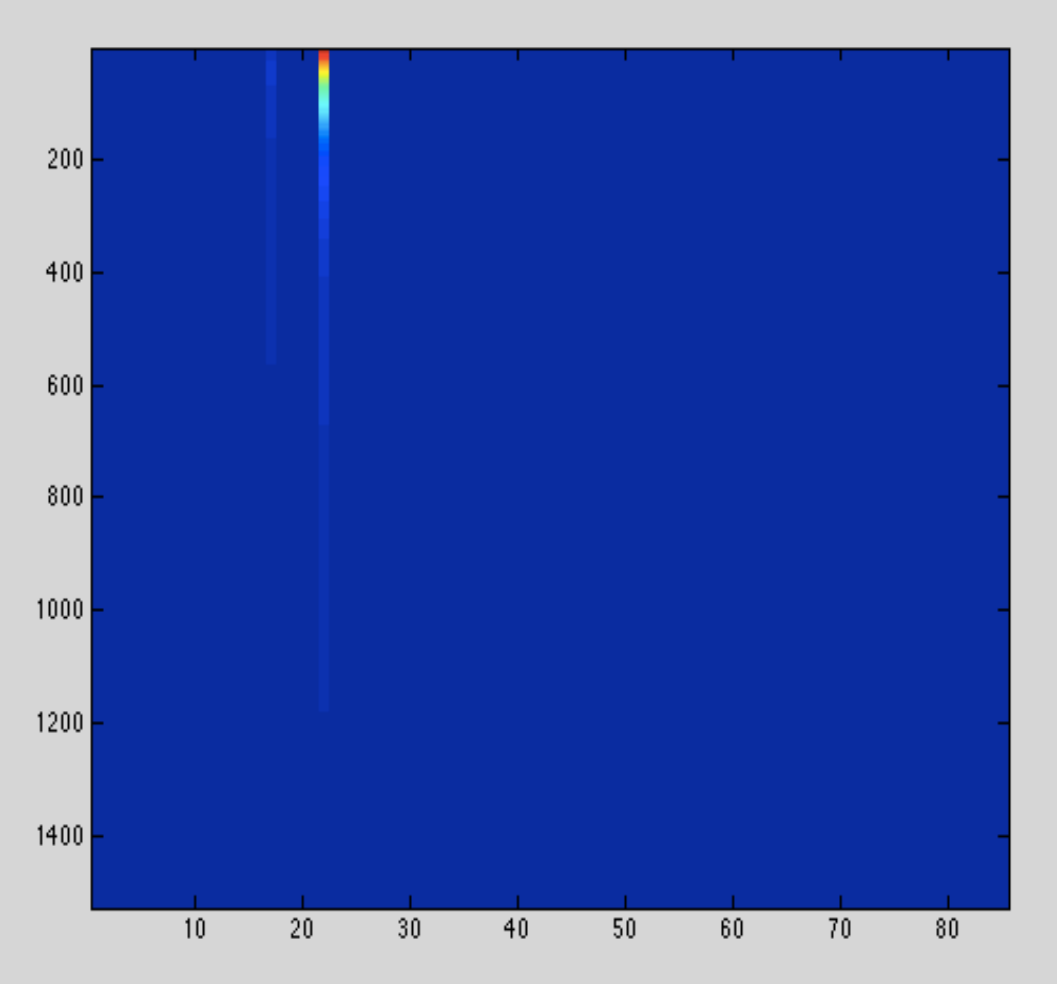

Figura 21. Representación con imagesc del vector bidimensional devuelto por el método de estimación.

La representación del espectrograma solo se hará en Matlab para poder analizarlo con detenimiento. En cambio cuando estemos hablando de la aplicación de afinación en Objetive-C directamente se cogerán dichos resultados y se trabajara con ellos puesto que los análisis y pruebas ya se habrán realizado en Matlab.

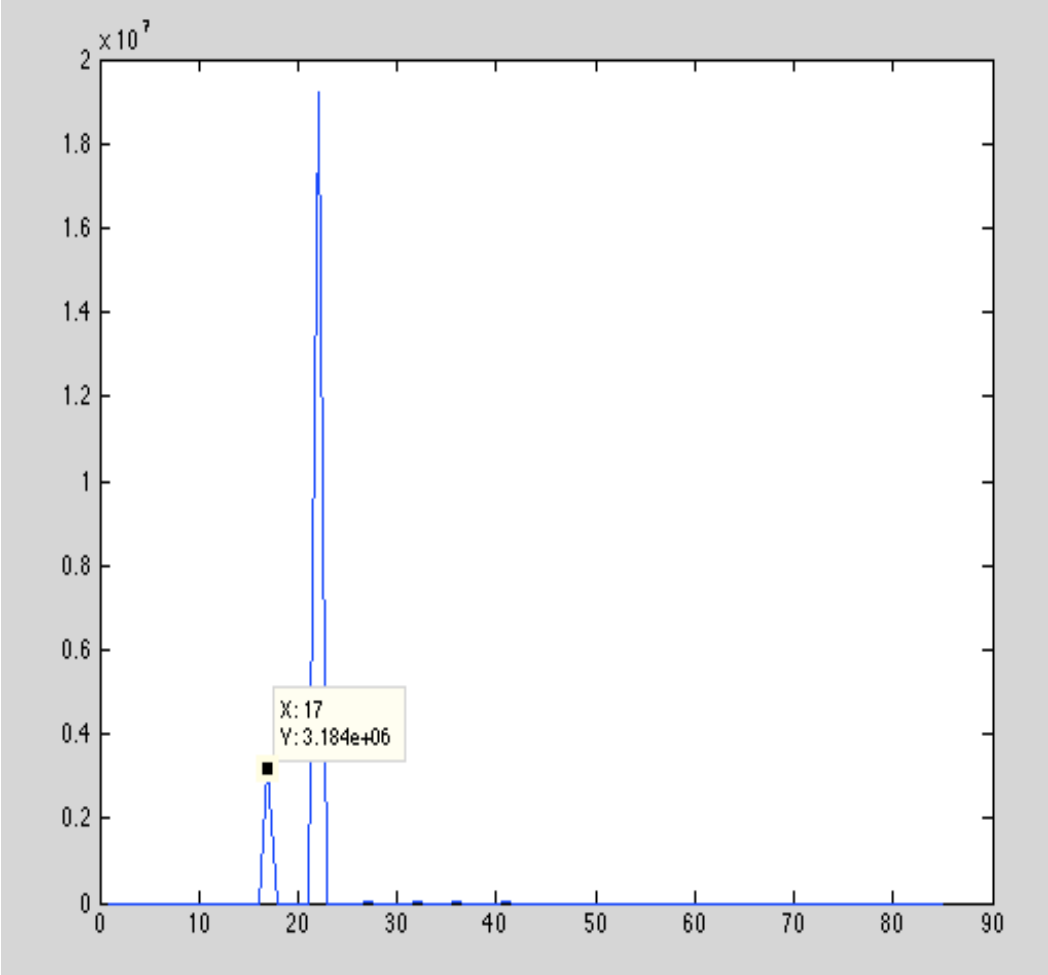

Figura 22. Representación con plot del vector unidimensional devuelto por el método.

4.7.4. Devolución de frecuencias.

Una vez hemos aplicado el método de estimación tendremos que realizar un par de cosas con los datos que este nos ha devuelto, dichos pasos se enumeran a continuación:

- Dictaminar si dichas notas o frecuencias están sonando realmente.
- Si están sonando determinar cuál es la desviación de las mismas.
- Mostrar que cuerdas están sonando y como de afinadas están.

4.7.4.1. Dictamen de resultados.

Como hemos comentado con anterioridad una vez que ejecutamos el método de estimación habrá que decidir, con los datos devueltos por este, si realmente están sonando

unas cuerdas u otras o por el contrario no está sonando ninguna. Para ello lo que habrá que hacer es observar cual es la energía en las notas MIDI que se corresponden con cada cuerda y definir una serie de umbrales o limites. Estos umbrales serán los encargados de tomar la decisión, puesto que dichos umbrales nos dirán si la energía que hay en cierta nota es suficiente como para que esa nota este sonando realmente. Por tanto necesitaremos establecer 6 umbrales uno para cada cuerda de la guitarra y estos umbrales se fijaran a través de la realización de múltiples estimaciones, es decir, se hará un estudio estadístico. Con los resultados de las estimaciones se pondrá ver cual es la energía media que tendrá una cuerda cuando está sonando y a partir de esa energía media se definirá el umbral. En tabla 7 se puede ver un ejemplo de distintas estimaciones que se han realizado para definir un umbral obstante para definir un umbral habrá que realizar muchas mas estimaciones que las mostradas aquí.

	Energía					
Estimación	Cuerda 1 MIDI 64	Cuerda 2 MIDI 59	Cuerda 3 MIDI 55	Cuerda 4 MIDI 50	Cuerda 5 MIDI 45	Cuerda 6 MIDI 40
Cuerda 6 y 5	5,80e+03	5.39e+03	5,52e+04	3,19e+04	1,19e+07	3,18e+06
Cuerda 6 y 4	5,17e+03	3,51e+03	3,63e+04	1,75e+07	2,91e+04	3,04e+06
Cuerda 6 y 3	2,51e+04	1.84e+04	1,25e+07	1,83e+04	2,04e+04	2,95+06
Cuerda 6 y 2	3,04e+04	9,57e+04	4,20e+04	4,24e+04	5,01e+05	3,02e+06
Cuerda 6 y 1	6,62e+04	2,75+e03	2,71e+04	1,69e+04	1,39+e04	2,97e+06
Cuerda 6, 5 y 4	8,66e+03	7,22e+03	5,91e+04	1,75e+07	1,93e+07	3,24e+06
Cuerda 5, 4 y 3	2,92+e04	2,17e+04	1,23e+07	1,75e+07	1,93e+07	1,26e+05
Cuerda 4, 3 y 2	5,53e+04	1,01e+05	1,29e+07	1,74e+07	5,26e+05	1,08e+05
Cuerda 3, 2 y 1	7,50e+04	1,08e+05	1,27e+07	2,63e+04	4,93e+05	1,84e+04

Tabla 7. Ejemplo de valores de energía para distintas estimaciones.

Para determinar tanto los umbrales en Matlab como en la aplicación lo que hemos hecho ha sido crear una base de datos con todas las posibles combinación de las 6 notas o cuerdas.

Por otra parte los umbrales que hemos utilizado en la aplicación se podrán ver el Anexo I y los que hemos utilizado en Matlab para hacer las pruebas en la Tabla 8.

Umbrales	Valor
Cuerda 1	4*10^4
Cuerda 2	1*10^5
Cuerda 3	9*10^6
Cuerda 4	1*10^7
Cuerda 5	1*10^7
Cuerda 6	2*10^6

Tabla 8. Umbrales de decisión utilizados en Matlab.

4.7.4.2. Calculo de la desviación.

Una vez que se ha dictaminado que está sonando tal cuerda habrá que ver como de afinada o desafinada esta, para ello crearemos un método o función el cual va a hacer lo siguiente:

- Realizar la FFT de x longitud del buffer de datos, la longitud de la FFT dependerá de la Δf que deseemos que vendrá determinada por la siguiente fórmula:

$$\Delta f = \frac{fs}{N} \tag{97}$$

donde fs es la frecuencia de muestreo y N el numero de muestras. Como queríamos una $\Delta f = 0{,}5\ Hz$ hemos optado porque $N = 16384$ y $fs = 8000\ Hz$.

- Cuando ya tenemos la Transformada de Fourier, lo que haremos es irnos a la frecuencia fundamental y a los armónicos no solapados. Para ello haremos uso de la siguiente ecuación:

$$f_k = \omega_k \frac{fs}{2\pi} = \frac{k}{N} fs \tag{98}$$

puesto que la transformada nos devuelve muestras k por amplitud no frecuencia por amplitud.

- Una vez allí vamos a analizar la transformada en los intervalo $[f_0 * 2^{-\frac{1}{24}}, f_0 * 2^{\frac{1}{24}}]$ para cada frecuencia fundamental y armónico no solapado, intentado encontrar a que frecuencia la amplitud es la mayor para ello haremos uso de (99).
- A continuación cogeremos las frecuencias que hemos obtenido en el paso anterior y calcularemos la diferencia con las frecuencias teóricas $dif = freal - fteorica$.
- Por ultimo, calcularemos la desviación para cada frecuencia con la siguiente ecuación:

$$desviacion = \frac{dif}{2^{\frac{1}{24}}} \tag{99}$$

Y la desviación total de cada cuerda será igual a la media de la desviación para cada frecuencia.

Una vez realizados estos pasos ya sabremos como esta de desafinada la cuerda que se está tocando. Unas líneas más abajo se muestra la implementación de estos pasos en

Matlab mediante la función *Obtfrec* y en la figura 23 se muestra un ejemplo de lo que devuelve esta función.

Función Obtfrec.

```
function [frecdefinitivas,desviacion]=Obtfrec(frecs,y1posa,fs,nfft)

fx=zeros(length(frecs),3);
for s=1:length(frecs);
    fx(s,1)=frecs(s)*2^(-1/24);
    fx(s,2)=frecs(s);
    fx(s,3)=frecs(s)*2^(1/24);
end


[s1,s2]=size(fx);
mult=zeros(s1,s2);

for c=1:s1


for j=1:s2
    mult(c,j)=round((fx(c,j)*16384)/fs)+1;

end


end

maxs=zeros(s1,1);
maxsk=zeros(s1,1);
for d=1:s1;

    maxs(d)=max(y1posa(mult(d,1):mult(d,3)));
    maxsk(d)=find(y1posa==maxs(d))-1;
end

frecdefinitivas=zeros(1,length(maxsk));

for o=1:length(maxsk)

    frecdefinitivas(o)=maxsk(o)*fs/nfft;

end

desviacion=zeros(1,length(frecdefinitivas));
resto=zeros(1,length(desviacion));
for t=1:length(desviacion)
    resto(t)=abs(fx(t,1)-fx(t,2));
    desviacion(t)=(((frecdefinitivas(t)-frecs(t))/resto(t))/2)*100;
end
end
```

```
La nota es: E
La nota es: A

a =

E
A


d =

   82.5195  165.0391  412.5977         0         0
  109.8633  219.2383  551.2695         0         0


c =

    2.4197    2.4197    2.4197       NaN       NaN
   -2.1830   -6.0811    4.0541       NaN       NaN
```

Figura 23. Resultados que devuelve la función Obtfrec.

4.7.4.3. Visualización de resultados.

Ya que sabemos que cuerdas se están tocando y si están afinadas o no lo único queda por hacer es mostrar los resultados.

En Matlab lo que hemos hecho es implementar una función que tiene por nombre *lnznota1y2*. Dicha función lo que hará es ejecutar todos los métodos que hemos visto en los apartados anteriores y mostrara los resultados en Command Window de Matlab como se puede ver en la Figura 23. El código de la función se muestra a continuación:

Función lnznota1y2.

```
function [sol,z,a,x,y]=lnznota1y2(i,v)
[x,y,X,f0,tam]=getTFs1(i,v,0);
```

```
sol=lanzar1(x);
z=Obtnotas3(sol)
a=nota2midi2(z)
[d,c]=defrecmult(y,z)

end
```

En el caso de la aplicación lo que haremos es crear una vista. Para ello tendremos que añadir a nuestro proyecto de la aplicación llamado PruebaAfinador un storyboard. En este storyboard a su vez añadiremos un View Controller como se muestra en la Figura 24, del funcionamiento de este ViewController se encargara la clase ViewController.m, la cual se puede ver en el anexo I.

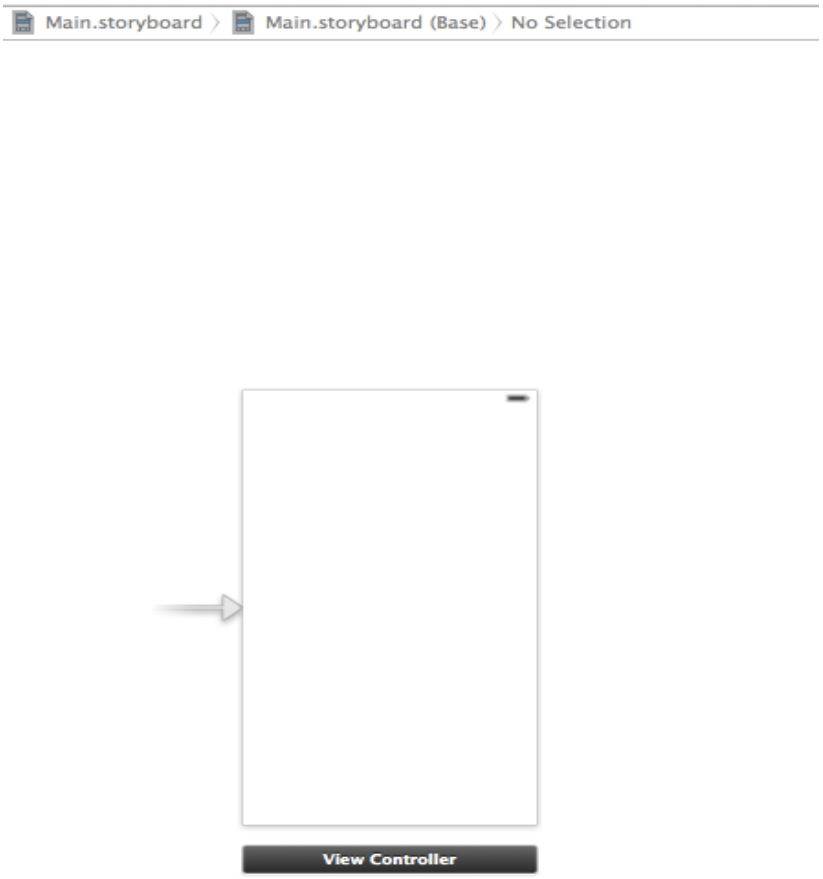

Figura 24. Visualización de un vista View Controller en Xcode.

Para indicar si una cuerda está sonando utilizaremos un puntero o indicador. Por tanto necesitaremos 6 indicadores uno para cada cuerda, cada uno de estos indicadores se

iluminara en el caso de que la cuerda a la que corresponde se este tocando en ese momento. Además dichos indicadores se moverán en torno a un punto central, a la derecha o la izquierda según si la cuerda esta desafinada por encima o por debajo de la frecuencia fundamental. Para implementar estos indicadores en nuestra aplicación hemos utilizado una serie de objetos de la framework UIKit llamados UIImageView, los cuales se añadirán a la vista que se puede observar en la Figura 24. El código que se encargara del funcionamiento de estos punteros o indicadores se puede ver en la Figura 25 que ejecutara cada vez que haya un nuevo conjunto de datos en el buffer o Audio Queue.

```
if (tam>0) {
    //[ledE1 setAlpha:1];
    for (int i=0; i<tam; i++) {

        if (notas[i]==16) {
            [ledE2 setBackgroundColor:[UIColor blueColor]];

              [ledE2  setFrame:CGRectMake(151+(2*des[i]),313,16 , 41)]

        }else if (notas[i]==21){
             [ledA setBackgroundColor:[UIColor redColor]];
             [ledA setFrame:CGRectMake(151+(2*des[i]), 264,16 , 41)];
        }else if (notas[i]==26){
             [ledD setBackgroundColor:[UIColor yellowColor]];
            [ledD  setFrame:CGRectMake(152+(2*des[i]), 215,16 , 41)];
       }else if (notas[i]==31){
             [ledG setBackgroundColor:[UIColor orangeColor]];
            [ledG setFrame:CGRectMake(152+(2*des[i]), 163,16 , 41)];
       }else if (notas[i]==35){
              [ledB setBackgroundColor:[UIColor greenColor]];
            [ledB  setFrame:CGRectMake(152+(2*des[i]), 114,16 , 41)];
       }else if (notas[i]==40){
             [ledE1 setBackgroundColor:[UIColor blackColor]];
            [ledE1  setFrame:CGRectMake(151+(2*des[i]), 65,16 , 41)];
        }
    }

    tam=0;


}else{

    ledE1.backgroundColor=[UIColor whiteColor];
ledB.backgroundColor=[UIColor whiteColor];
    ledA.backgroundColor=[UIColor whiteColor];
    ledG.backgroundColor=[UIColor whiteColor];

    ledD.backgroundColor=[UIColor whiteColor];
    ledE2.backgroundColor=[UIColor whiteColor];
```

Figura 25. Código encargado del funcionamiento de los punteros o indicadores.

Como se puede ver el método setBackgoundColor será el encargado de encender o apagar el puntero y el método setFrame el de mover los punteros a un lado u otro según el valor de la desviación que vendrá dado por la variable *des* que es un vector.

Una vez llegado a este punto lo único que nos quedara por hacer es intentar buscar formas de mejorar el aspecto y funcionamiento de nuestra aplicación en general, por ejemplo, cambiando el fondo de la vista que mostrara nuestra aplicación, mejorando el diseño de los indicadores o intentando optimizar los métodos y clases encargadas de estimar y tomar la decisión de si una cuerda está sonando o no. En la figura 26 se puede observar el aspecto final de la aplicación.

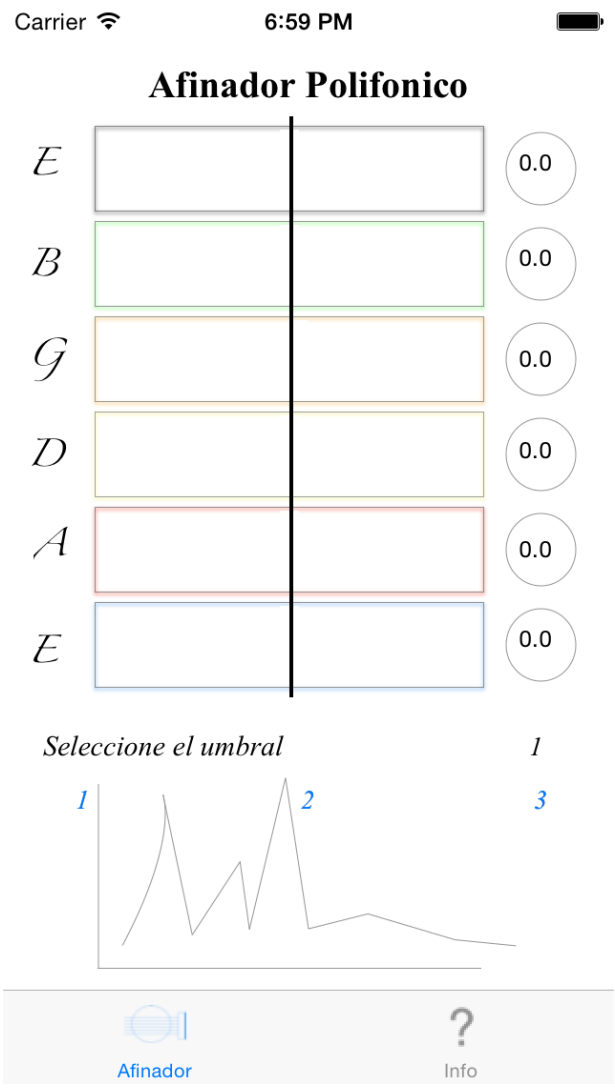

Figura 26. Aspecto final de la aplicación.

4.7.4.4. Otras consideraciones.

Aquí vamos a comentar un par de problemas que se han aparecido en una nuestra aplicación una vez realizados todos los pasos anteriores.

Problema con los umbrales

En uno de los sub-apartados anteriores hemos visto que se han establecido una serie de umbrales pero estos solo serán validos para un nivel de señal de entrada determinado por lo tanto esto será un problema puesto que si el nivel de señal de entrada es muy bajo no detectara ninguna nota aunque se esté tocando puesto que nunca se superaran ninguno de los umbrales y para el caso del nivel alto será al contrario. Por tanto a continuación proponemos una serie de soluciones a este problema:

- Establecer unos umbrales fijos para unos determinados niveles de señal los cuales el usuario podrá elegir según le convenga.
- Crear un control automático de la ganancia que fije los umbrales automáticamente según el nivel de entrada que se tenga.

Nosotros hemos obstado por la primera opción pues es la solución mas sencilla de implementar no obstante no es la solución más optima y eficiente. Para implementar esta opción solo hará falta establecer 3 o 4 serie de umbrales y colocar un botón o botones en nuestra de vista de la aplicación para que se pueda elegir una u otra serie de umbrales. En la figura 26 se puede ver el código en Xcode encargado de los controlar los botones para poder elegir uno u otro umbral.

```
- (IBAction)cambiarImagen:(id)sender {
    //ViewController2 *controller=[[ViewController2 alloc]init];
    //int uvalor=0;
    //controller.umbral=1;
    UIButton *boton = (UIButton *) sender;
    int tag = boton.tag;


    switch (tag) {
        case 1:

            umbnotas=1;
            break;
        case 2:

            umbnotas=2;
            break;
        case 3:


            umbnotas=3;
            break;
        default:
            break;

    }
    //_umbral=nota;

    NSLog(@"%d",umbnotas);
    self.valueLabel.text=[NSString stringWithFormat:@"%2d",umbnotas];


}
```

Figura 27. Código encargado del control de los botones de elección de umbral.

Problema de detección con señales de voz.

Por otro lado también se nos presenta el problema de que no queremos que nuestra aplicación muestre que está sonando una cuerda cuando la señal de entrada son señales de voz. Para solucionar esto lo que haremos será coger la FFT que hemos calculado y a partir de esta determinar de si se trata de una señal de voz o musical. Para determinar qué tipo de señal es, vamos a hacer lo siguiente:

- Coger la transformada de Fourier de la señal de entrada.
- Determinar el máximo de la transformada y a partir de este determinar el valor medio de la misma.

- Comprobar cuantas muestras de la transformada tienen un valor superior al valor medio de la misma.
- Si el numero de muestras es elevado se tratara de una señal de voz y no se aplicara el método de estimación en caso contrario sí.

En la figura 28 se muestra el código encargado de determinar el tipo de señal de entrada.

```
int GetTFs::detector(Float32 **cfreqrV){
    Float32 *sumbdet;
    sumbdet = new Float32 [longmidi];
    Float32 suma,a;
    Float32 sumatorio=0;
    mayor=0;
    supera=0;

    for (int i=0; i<longmidi; i++) {
        suma=0;
        for (int j=0; j<colindex; j++) {

            a=cfreqrV[j][i];

            suma=suma+a;

        }
        sumbdet[i]=suma;
        sumatorio+=sumbdet[i];
        if (sumbdet[i]>mayor) {
            mayor=sumbdet[i];
        }

    }
    valormedio=sumatorio/(float)longmidi;
     for (int i=0; i<longmidi; i++) {

         if (sumbdet[i]>valormedio) {
             supera++;
         }


     }
```

Figura 28. Código para determinar si debemos aplicar el método de estimación.

Capítulo 5

Resultados y discusión

Los resultados que obtendremos serán de dos tipos: los resultados devueltos por Matlab y los resultados devueltos por la aplicación en nuestro dispositivo con iOS. En caso de Matlab para obtener resultados veraces con los que discutir el buen funcionamiento del método de estimación hemos creado una base de datos que a partir de 6 señales básicas que se corresponde con las notas de cada cuerda se pueden obtener todas las combinaciones posibles, concretamente 1950 combinaciones. A continuación se muestra el código del algoritmo encargado de elaborar la base de datos.

Código de elaboración de la base de datos.

```
pathname = uigetdir;
D = dir([pathname '/*.mat']);
nfiles=length(D);
recorte=0;

for k=1:nfiles,

     nombre=D(k).name(1:end-4);
    path=[pathname '/' nombre];
    ruta_mat=[path,'.mat'];
    load(ruta_mat)

end

c='nota';

y=[y1;y2;y3;y4;y5;y6];

for i=1:6
for j=1:6
if i~=j
```

```
        nota=[y(i,:)+y(j,:)];
        x=[c num2str(i) num2str(j)];
        eval(sprintf('save %s  nota', ['basesum/',x,'.mat']));
for r=1:6
if i~=j &&  ( r~=i && r~=j)
             nota=[y(i,:)+y(j,:)+y(r,:)];
        x=[c num2str(i) num2str(j) num2str(r)];
        eval(sprintf('save %s  nota', ['basesum/',x,'.mat']));
end
for f=1:6
if (i~=j &&  ( r~=i && r~=j)) && (f~=i &&  ( f~=j && f~=r))
             nota=[y(i,:)+y(j,:)+y(r,:)+y(f,:)];
        x=[c num2str(i) num2str(j) num2str(r) num2str(f)];
        eval(sprintf('save %s  nota', ['basesum/',x,'.mat']));
end
for t=1:6
if ((i~=j &&  ( r~=i && r~=j)) && (f~=i &&  ( f~=j && f~=r))) && ((t~=i
&& t~=j)&&( t~=r && t~=f))
             nota=[y(i,:)+y(j,:)+y(r,:)+y(f,:)+y(t,:)];
        x=[c num2str(i) num2str(j) num2str(r) num2str(f) num2str(t)];
        eval(sprintf('save %s  nota', ['basesum/',x,'.mat']));
end
for g=1:6
if (((i~=j &&  ( r~=i && r~=j)) && (f~=i &&  ( f~=j && f~=r))) && ((t~=i
&& t~=j)&&( t~=r && t~=f)) && (((g~=i && g~=j)&&( g~=r && g~=f)) &&
(g~=t)))
             nota=[y(i,:)+y(j,:)+y(r,:)+y(f,:)+y(t,:)+y(g,:)];
        x=[c num2str(i) num2str(j) num2str(r) num2str(f) num2str(t)
num2str(g)];
        eval(sprintf('save %s  nota', ['basesum/',x,'.mat']));
end
end
end
end
end
end
end
end
```

Para el caso del afinador en nuestro IPhone lo que hemos hecho ha sido tocar un numero elevados de combinaciones de notas con la guitarra no obstante también hemos utilizado las señales de la base de datos creada anteriormente, las cuales hemos tenido que transformar a archivos .wav con la función wavwrite de Matlab, en la figura 28 se puede observar cómo se aplicara.

```
pathname = uigetdir;
D = dir([pathname '/*.mat']);
nfiles=length(D);
recorte=0;

for k=1:nfiles,

     nombre=D(k).name(1:end-4);
    path=[pathname '/' nombre];
    ruta_mat=[path,'.mat'];
    load(ruta_mat)
    wavwrite(nota,44100,nombre);
end
```

Figura 28. Código encargado de transformar las señales de la base de datos en archivos .wav

5.1. Resultados en Matlab.

Como vimos en apartados anteriores los resultados en Matlab se mostraran en el Command Window y en distintas figuras. Con dichos resultados hemos elaborado la Tabla 9para unas 30 o 40 señales de la base de datos y así hacer una estimación sobre los resultados. Estas 30 señales se han elegido aleatoriamente y se muestran en la Tabla 8 (en la nomenclatura de la señal: 1 corresponderá con la cuerda 6 y la nota midi 40, 2 con la cuerda 5 y nota midi 45, 3 con la cuerda 4 y la nota midi 50, 4 con la cuerda 3 y la nota midi 55, 5 con la cuerda 2 y la nota midi 59 y por ultimo 6 con la cuerda y la nota midi 64)

.

Nº de señal	Nombre de la señal
1	Nota12
2	Nota26
3	Nota53
4	Nota135
5	Nota245
6	Nota612
7	Nota1235
8	Nota1345

9	Nota2361
10	Nota3245
11	Nota5421
12	Nota12463
13	Nota14625
14	Nota31465
15	Nota32651
16	Nota35462
17	Nota43652
18	Nota45316
19	Nota52164
20	Nota63154
21	Nota123645
22	Nota124356
23	Nota231456
24	Nota425613
25	Nota521346

Tabla 9. Señales elegidas aleatoriamente para realizar la estimación.

Nº de señal	Notas estimadas	Desviación estimada
1	E 40 (cuerda 6) A 45 (cuerda 5)	Para E: 2,4197 % Para A: -1,4033 %
2	A 45 (cuerda 5) E 64 (cuerda 1)	Para A: 16,5626 % Para E: 8,72038 %
3	B 59 (cuerda 2) D 50 (cuerda 4)	Para B: 6,7144 % Para D: 12,399 %
4	E 40 (cuerda 6) D 50 (cuerda 4) B 59 (cuerda 2)	Para E: 19,202 % Para D: 12,0399 % Para B: 6.4062 %
5	A 45 (cuerda 5)	Para A: 1,1614 %

	G 55 (cuerda 3) B 59 (cuerda 2)	Para G: -0,27048 % Para B: 6.4002 %
6	E 64 (cuerda 1) E 40 (cuerda 6) A 45 (cuerda 5)	Para E: 4,9015 % Para E: 15,8317 % Para A: 1,6633 %
7	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) B 59 (cuerda 2)	Para E: -4,3897 % Para A: 1,1633 % Para D: 12,0399 % Para B: 6.4062 %
8	E 40 (cuerda 6) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2)	Para E: 2,3597 % Para D: 12,3049 % Para G: 1,4569% Para B: 6.5262 %
9	A 45 (cuerda 5) D 50 (cuerda 4) E 64 (cuerda 1) E 40 (cuerda 6)	Para A: 1,16133 % Para D: -8,23157 % Para E: 23,3211% Para E: -1,0197 %
10	A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2)	Para A: 1,6141 % Para D: 12,3043 % Para G: -5,1872% Para B: 6,4002 %
11	B 59 (cuerda 2) G 55 (cuerda 3) A 45 (cuerda 5) E 40 (cuerda 6)	Para B: 6.9362 % Para G: -5,1872 % Para A: 2,4238 % Para E: 2,3508 %
12	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) E 64 (cuerda 1)	Para E: 2,4197 % Para A: -1,4033 % Para D: -10,4196 % Para G: 0,2669% Para E: 7,8492%
13	E 40 (cuerda 6) A 45 (cuerda 5) G 55 (cuerda 3)	Para E: 2,4197 % Para A: -1,4033 % Para G: 0,4036 %

	B 59 (cuerda 2) E 64 (cuerda 1)	Para B: 6,9062 % Para E: 20.3518 %
14	E 40 (cuerda 6) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 2,4197 % Para D: -10,6977 % Para G: 0,4036 % Para B: 6,9062 % Para E: 24.1801 %
15	E 40 (cuerda 6) A 45 (cuerda 5) D 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 2,4197 % Para A: -1,4033 % Para D: 10,6977 % Para B: 6,9062 % Para E: 14.8315 %
16	A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para A: -1,4033 % Para D: -10,4196 % Para G: 0,2669 % Para B: 7,744 % Para E: 9.7406 %
17	A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para A: -1,4033 % Para D: -10,4196 % Para G: 0,2669 % Para B: 7,744 % Para E: 9.7406 %
18	E 40 (cuerda 6) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 2,4197 % Para D: -10,6977 % Para G: 0,4036 % Para B: 6,9062 % Para E: 24.1801 %
19	E 40 (cuerda 6) A 45 (cuerda 5) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 2,4197 % Para A: -1,4033 % Para G: 0,4036 % Para B: 6,9062 % Para E: 20,3518 %
20	E 40 (cuerda 6)	Para E: 2,4197 %

	D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para D: -10,6977 % Para G: 0,4036 % Para B: 6,9062 % Para E: 24.1801 %
21	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 2,4197 % Para A: -1,4033 % Para D: -10,6977 % Para G: 0,4036 % Para B: 6,9062 % Para E: 20.3518 %
22	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 2,4197 % Para A: -1,4033 % Para D: -10,6977 % Para G: 0,4036 % Para B: 6,9062 % Para E: 20.3518 %
23	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 2,4197 % Para A: -1,4033 % Para D: -10,6977 % Para G: 0,4036 % Para B: 6,9062 % Para E: 20.3518 %
24	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 2,4197 % Para A: -1,4033 % Para D: -10,6977 % Para G: 0,4036 % Para B: 6,9062 % Para E: 20.3518 %
25	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2)	Para E: 2,4197 % Para A: -1,4033 % Para D: -10,6977 % Para G: 0,4036 % Para B: 6,9062 %

	E 64 (cuerda 1)	Para E: 20.3518 %

Tabla 10. Resultados de la estimación de las señales de la tabla 8 en Matlab.

De estos resultados se pueden sacar una serie de conclusiones:

- Que nuestra aplicación está realizando una buena estimación puesto que para todas esas 25 señales que hemos probado se han detectado las notas que realmente estaban presentes.
- También que el algoritmo de determinación funciona correctamente puesto que las desviaciones que nos han dado son pequeñas y se mantiene para una misma nota.
- Que aunque estos resultados son buenos no nos van a servir para sacar una valoración final de nuestra aplicación puesto que estas señales no se verán afectadas por ningún agente no obstante nos dice que el método de estimación esta funcionando satisfactoriamente.

5.2. Resultados en la aplicación.

En la aplicación los resultados se mostraran gráficamente pero también hemos añadido a nuestro código una serie de NSLog para que se muestren numéricamente en la consola de XCode, ambos resultados se pueden observar en las Figuras 29 y 30. Al igual que para Matlab, elaboramos la tabla 10 que recoge los resultados para distintas señales de entrada.

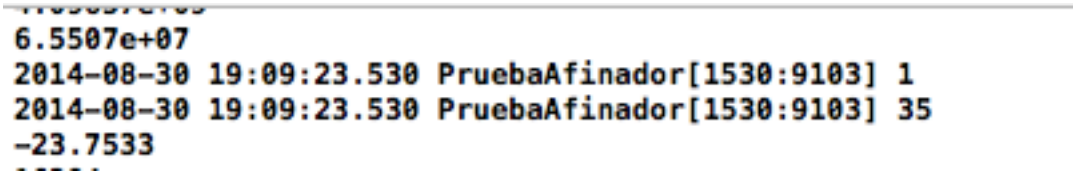

Figura 29. Resultados mostrados en XCode.

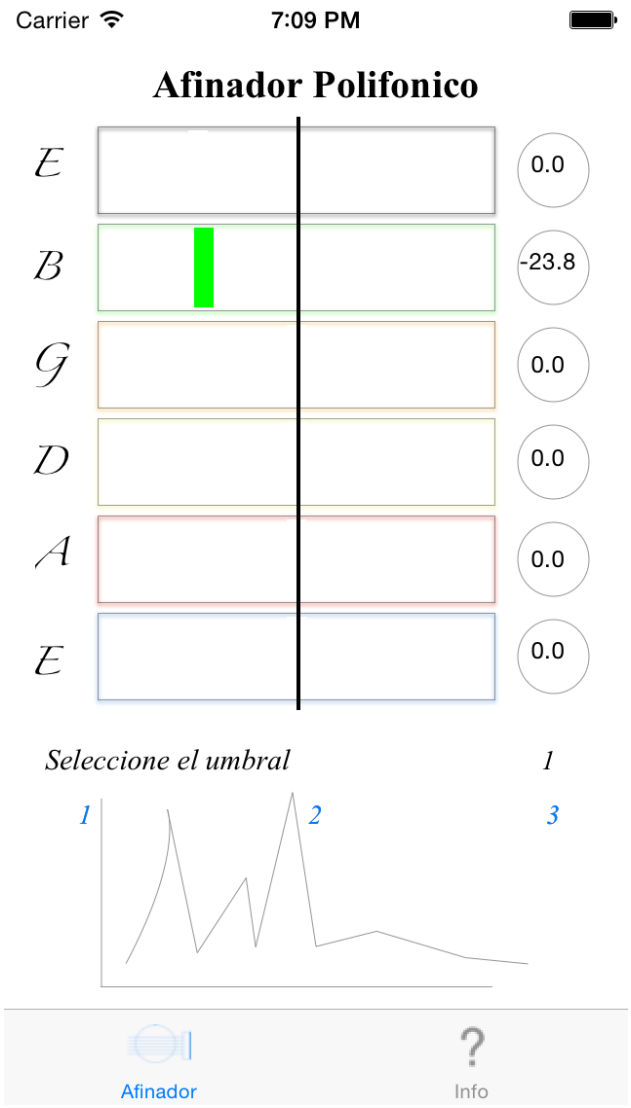

Figura 30. Aplicación mostrando resultados de la estimación.

Nº de señal	Notas estimadas	Desviación estimada
1	E 40 (cuerda 6) A 45 (cuerda 5)	Para E: 5,7221 % Para A: 1,1614 %
2	A 45 (cuerda 5) E 64 (cuerda 1)	Para A: -1,4033 % Para E: 20,6884 %
3	B 59 (cuerda 2) D 50 (cuerda 4)	Para B: 7,7744 % Para D: 10,4399 %
4	E 40 (cuerda 6)	Para E: 2,4197 %

	D 50 (cuerda 4) B 59 (cuerda 2)	Para D: 10,4399 % Para B: 6.9062 %
5	A 45 (cuerda 5) G 55 (cuerda 3) B 59 (cuerda 2)	Para A: -1,4033 % Para G: 0,2669 % Para B: 7.7744 %
6	E 64 (cuerda 1) E 40 (cuerda 6) A 45 (cuerda 5)	Para E: 18,1445 % Para E: 2,4197 % Para A: -1,4033 %
7	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) B 59 (cuerda 2)	Para E: 2,4197 % Para A: -1,4033 % Para D: 10,4399 % Para B: 6.9062 %
8	E 40 (cuerda 6) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2)	Para E: 2,4197 % Para D: 10,4399 % Para G: 0,2669% Para B: 6.9062 %
9	A 45 (cuerda 5) D 50 (cuerda 4) E 64 (cuerda 1) E 40 (cuerda 6)	Para A: -1,4033 % Para D: -10,697 % Para E: 28,4811% Para E: 2,4197 %
10	A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2)	Para A: -1,4033 % Para D: 10,4399 % Para G: 0,1302% Para B: 7,7744 %
11	B 59 (cuerda 2) G 55 (cuerda 3) A 45 (cuerda 5) E 40 (cuerda 6)	Para B: 6.9062 % Para G: 11,0299 % Para A: -1,4033 % Para E: 2,4197 %
12	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) E 64 (cuerda 1)	Para E: 5,7211 % Para A: -1,3633 % Para D: -7,9613 % Para G: 0,2403% Para E: 21,5645%

13	E 40 (cuerda 6) A 45 (cuerda 5) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: -2,7044 % Para A: -7,6753 % Para G: 2,1193 % Para B: -9,8199 % Para E: 2.1602 %
14	E 40 (cuerda 6) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 2,3500 % Para D: -7,6911 % Para G: 1,4549 % Para B: 6,9372 % Para E: 25.4911 %
15	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: -7,7598 % Para A: -1,3633 % Para D: -8,2315 % Para B: 6,4002 % Para E: 19.8815 %
16	A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para A: -1,3633 % Para D: -8,2319 % Para G: -0,4239 % Para B: 6,4002 % Para E: 24.9988 %
17	A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para A: -2,9287 % Para D: -7,9613 % Para G: -5,1827 % Para B: 6,5252 % Para E: 25.3647 %
18	E 40 (cuerda 6) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: -1,0197 % Para D: -7,9617 % Para G: 1,4549 % Para B: 6,4062 % Para E: 25.2155 %
19	E 40 (cuerda 6) A 45 (cuerda 5) G 55 (cuerda 3)	Para E: 19,202 % Para A: 1,1614 % Para G: 1,4549 %

	B 59 (cuerda 2) E 64 (cuerda 1)	Para B: 6,4002 % Para E: 22,239 %
20	E 40 (cuerda 6) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: -11,13 % Para D: -7,9613 % Para G: 1,4545 % Para B: 6,4002 % Para E: 25.2801 %
21	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 17,5164 % Para A: 6,7159 % Para D: -5,7997 % Para G: 2,0056 % Para B: 7,2459 % Para E: 21.84 %
22	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 5,7211 % Para A: 1,1614 % Para D: -8,2315 % Para G: 0,3921 % Para B: 6,4062 % Para E: 25.3518 %
23	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 19,202 % Para A: -1,3633 % Para D: -7,9613 % Para G: 1,4549 % Para B: 6,4062 % Para E: 25.3618 %
24	E 40 (cuerda 6) A 45 (cuerda 5) D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para E: 5,7211 % Para A: -1,3633 % Para D: -8,2315 % Para G: 0,3921 % Para B: 6,4002 % Para E: 25.2857 %
25	E 40 (cuerda 6) A 45 (cuerda 5)	Para E: 12,4616 % Para A: 16,8151 %

	D 50 (cuerda 4) G 55 (cuerda 3) B 59 (cuerda 2) E 64 (cuerda 1)	Para D: -8,3666 % Para G: -4,3213 % Para B: 7,9497 % Para E: 13,8647 %

Tabla 11. Resultados de la estimación de las señales de la tabla 8 en la aplicación.

Como vemos hemos obtenido unos resultados muy parecidos a los que obteníamos en Matlab, exceptuando en el cálculo de las desviaciones que para algunos casos la dispersión o variación con los valores obtenidos en Matlab es considerable. Pero esta dispersión no va a ser muy importante ya que en este caso la señal está expuesta a distintos agentes que son las causantes de dicha variación de resultados.

Por tanto podremos decir en base a estos resultados que nuestra aplicación va a proporcionar una muy buena estimación por lo cual vamos a poder afinador con éxito nuestra guitarra.

Capítulo 6

Conclusiones

Como se ha visto en esta memoria se ha desarrollado una aplicación de afinación basada en métodos estadísticos de estimación multi-pitch, concretamente, utilizando el método NLSFast o método de los mínimos cuadrados.

Con dicha aplicación también se ha visto que se pueden obtener resultados muy satisfactorios pero también hay que añadir que en el desarrollo de la misma se han encontrado unas serie de dificultades y también se han visto algunos aspectos de la aplicación que se pueden mejorar. Todas estas dificultades y posibles mejoras se verán en los siguientes sub-apartados.

6.1. Dificultades encontradas.

Uno de los principales problemas es que el manejo de audio en tiempo real en iOS se hace con ayuda de las frameworks Core Audio y Audio Unit las cuales vimos en el apartado 4.6, y en el cual se puedo comprobar que estas frameworks trabajaban a bajo nivel por lo cual tiene una curva de aprendizaje bastante pronunciada. Otro dificultad que hemos encontrado es que hay que conocer bien las frameworks disponibles ya que estas suelen ofrecer las mejores soluciones para implementar un determinado algoritmo pero muchas veces las desconocemos. Otro problema es que la señal de entrada del buffer pueden verse afectada por distintos agentes por lo cual a la hora de desarrollar nuestra aplicación habrá que tenerlo en cuenta.

6.2. Posibles mejoras.

Como hemos vistos la aplicación que hemos desarrollado obtiene unos buenos resultados pero también vimos que para obtener dichos resultados debíamos ajustar manualmente los umbrales entre uno de los disponibles, por tanto puede que para algún caso ninguno de los umbrales se ajuste bien. Por lo cual una posible mejora para nuestra aplicación podría ser como comentamos en apartados anteriores la implementación de una especie de control automático de la ganancia que ajuste automáticamente los umbrales según el nivel de la señal de entrada.

6.3. Líneas de futuro.

En este apartado se van a señalar algunas de las posibles líneas de futuro para la aplicación que hemos desarrollado, algunas líneas interesantes son las siguientes:

- Una línea de trabajo puede ser la extensión de esta aplicación como un entrenador de acordes. El entrenador de acordes tendrá gran utilidad en instrumentos de cuerda sin traste en los que hay que memorizar la posición de los dedos en el mástil o diapasón. Como no hay traste se tendrá una libertad musical puesto que se podrá variar con facilidad el pitch o frecuencia de cada nota. Pero cuando se pretenda ejecutar o tocar un acorde (tocar varias notas simultáneamente) los dedos se deberán colocar en la posición adecuada y de manera perfecta. Lo cual puede resultar algo complicado sino se es un músico profesional por lo cual con la ayuda de un entrenador de acordes se facilitara mucho esta tarea.
- Otra línea puede ser la de portar esta aplicación de afinación a otros sistemas operativos móviles como pueden Windows Phone, Android, Blackberry 7 o Firefox OS y comercializarla igual que se ha hecho con iOS.
- El método de estimación de que hemos utilizado a lo largo de este trabajo puede ser utilizado en otras aplicaciones de procesamiento de audio. Un posible campo de desarrollo podría ser una aplicación de clasificación del tipo de audio que esta sonando indicando si se trata de una señal de voz o musical. Por otra parte, otra posible línea de desarrollo podría ser la de crear un algoritmo basado en una señal musical con unas características determinadas no obstante esta aplicación seria más

curiosa que eficiente puesto que una señal de audio podría ser replicada con cierta facilidad.

Como se puede observar esta aplicación ofrece unas líneas de futuro que pueden ser muy fructíferas.

Capítulo 7

Planos y anexos

Anexo I: Código de la aplicación.

Archivos de implementación.

AppDelegate.m

```
#import "AppDelegate.h"

@implementation AppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
   return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
}

- (void)applicationDidEnterBackground:(UIApplication *)application
{
}

- (void)applicationWillEnterForeground:(UIApplication *)application
{
}

- (void)applicationDidBecomeActive:(UIApplication *)application
{
```

```
}

- (void)applicationWillTerminate:(UIApplication *)application
{
}

@end
```

main.m

```
#import <UIKit/UIKit.h>
#import "AppDelegate.h"

int main(int argc, char * argv[])
{
   @autoreleasepool {
      return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
   }
}
```

AudioInput.mm

```
#define AUDIO_BUF_SIZE 16384
#include "AudioInput.h"

#include <iostream>
#include <AudioToolbox/AudioToolbox.h>
#include <AudioUnit/AudioUnit.h>

/* Check for initalization errors and notify user */

#define checkStatus( err ) \
if(err) { cout << "CoreAudio Error " << __func__ << " " << err << " line "<< __LINE__ ;
return (-1);}\

using namespace std;

/*
*      - Creates an audio buffer at start up to write audio data to
 *                analyze.  Also inits pthread locks and variables
 */

CTAudioBuffer::CTAudioBuffer()
{
pthread_mutex_init(&flag_lock,NULL);
```

```
	pthread_mutex_init(&spin_lock,NULL);
	pthread_mutex_lock(&spin_lock);

	buffer = (int16_t  *) calloc(AUDIO_BUF_SIZE, sizeof(int16_t ));

	write_flag = TRUE;

	ReInit();
}

/*
*	- Zeros out the Audio Buffer
 */

void CTAudioBuffer::ReInit()
{
	memset(buffer, 0, sizeof(int16_t )*AUDIO_BUF_SIZE);
	buf_index = 0;
}

/*
 * Details		This sets the write flags causing the AudioInputProc() method to
 * bypass writing data to the shared buffer
 */

void CTAudioBuffer::SetWriteOkay()
{
	/* Take control of buffer write flag */

	pthread_mutex_lock(&flag_lock);

	write_flag = TRUE;

	/* Lock the FFT thread spin lock */

	pthread_mutex_lock(&spin_lock);

	pthread_mutex_unlock(&flag_lock);
}

/*
 * Details		- This causes the FFT analyzation thread to spin lock and wait
 * for the shared audio buffer to be ready to have a full second of audio data
 * points to be read
 */

void CTAudioBuffer::Synchronize()
{
	pthread_mutex_lock(&spin_lock);
	pthread_mutex_unlock(&spin_lock);
```

```
}


/*
* Details                   - Unlocks the spin lock syncronization lock (See above) and
 * clears the FFT thread to analyze the data. Also causes the AudioInputProc()
 * method to bypass writing data to the shared buffer
 */

void CTAudioBuffer::SetReadOkay()
{
        /* Take control of buffer write flag */

        pthread_mutex_lock(&flag_lock);

        write_flag = FALSE;

        /* unlock the FFT thread spin lock */

        pthread_mutex_unlock(&spin_lock);

        pthread_mutex_unlock(&flag_lock);
}

static void HandleInputBuffer (void *userData,
                    // my data
                                                AudioQueueRef inAQ,
                                // the qudio queue that owns this callback
                                                AudioQueueBufferRef inBuffer,
                    // defined buffer with the incoming data
                                                const AudioTimeStamp *inStartTime,
                    // time of the first sample in the audio queue
                                                UInt32 inNumPackets,
                                // Number of packets in the buffer
                                                const AudioStreamPacketDescription
*inPacketDesc)// The compressed packet data description if applicable
{
        CTAudioBuffer * ctAudioBuffer = (CTAudioBuffer *) userData;
        int x;

    if (inNumPackets == 0 && ctAudioBuffer-
>audioStreamData.dataFormat.mBytesPerPacket != 0)
        {
       inNumPackets = inBuffer->mAudioDataByteSize / ctAudioBuffer-
>audioStreamData.dataFormat.mBytesPerPacket;

          ctAudioBuffer->audioStreamData.currentPacket += inNumPackets;
        }

        if(!ctAudioBuffer->WriteStatus())
```

```
        {
          AudioQueueEnqueueBuffer(ctAudioBuffer->audioStreamData.queue, inBuffer,0,
NULL);
          return;
        }

        for(x = 0;(ctAudioBuffer->buf_index < AUDIO_BUF_SIZE) && (x <
inNumPackets); ctAudioBuffer->buf_index += 1, x++)
        {
          ctAudioBuffer->buffer[ctAudioBuffer->buf_index] = ((int16_t *)inBuffer-
>mAudioData)[x];
        }

        if(ctAudioBuffer->buf_index == AUDIO_BUF_SIZE)
        {
          // Buffer was filled so set a FFT read to be okay

          ctAudioBuffer->SetReadOkay();
        }

        AudioQueueEnqueueBuffer(ctAudioBuffer->audioStreamData.queue, inBuffer,0,
NULL);

}

/*
 * Details                  - Sets up an AudioUnit and creates a thread that reads from the
 * default input device
 */

CTAudioBuffer * InitAudio(void)
{
        CTAudioBuffer * myBuffer = new CTAudioBuffer();

        myBuffer->audioStreamData.dataFormat.mSampleRate = 8000.0;
    myBuffer->audioStreamData.dataFormat.mFormatID = kAudioFormatLinearPCM;

    myBuffer->audioStreamData.dataFormat.mChannelsPerFrame = 1; // mono
    myBuffer->audioStreamData.dataFormat.mBitsPerChannel = 16;
    myBuffer->audioStreamData.dataFormat.mFramesPerPacket = 1;
    myBuffer->audioStreamData.dataFormat.mBytesPerPacket = 2;
    myBuffer->audioStreamData.dataFormat.mBytesPerFrame = 2;
    myBuffer->audioStreamData.dataFormat.mReserved = 0;

        myBuffer->audioStreamData.dataFormat.mFormatFlags       =
    kLinearPCMFormatFlagIsSignedInteger |
    kLinearPCMFormatFlagIsPacked;

        DeriveBufferSize(myBuffer->audioStreamData.queue,
                                 &myBuffer->audioStreamData.dataFormat,
```

```
                                0.5,
                                &myBuffer->audioStreamData.bufferByteSize);

    OSStatus status;

  status = AudioQueueNewInput(&myBuffer->audioStreamData.dataFormat,
                                                    HandleInputBuffer,
                                                    (void *) myBuffer,
                                                    NULL,
                                                    kCFRunLoopCommonModes,
                                                    0,
                                                    &myBuffer->audioStreamData.queue);

    if (status) { printf("Could not establish new queue\n"); return NULL;
  }

    for (int i = 0; i < kNumberBuffers; ++i) {          // 1

      AudioQueueAllocateBuffer (myBuffer->audioStreamData.queue,
                                                     myBuffer-
>audioStreamData.bufferByteSize,
                                                    &myBuffer-
>audioStreamData.buffers[i]);

      AudioQueueEnqueueBuffer (myBuffer->audioStreamData.queue,
                                                    myBuffer-
>audioStreamData.buffers[i],
                                                    0,
                                                    NULL);

      myBuffer->audioStreamData.currentPacket = 0;
      myBuffer->audioStreamData.mIsRunning = true;

      AudioQueueStart(myBuffer->audioStreamData.queue, NULL);

    }

    return myBuffer;
}

void DeriveBufferSize (AudioQueueRef              audioQueue,              // 1
                                AudioStreamBasicDescription  * ASBDescription,
// 2
                                Float64                 seconds,                 // 3
                                UInt32                  *outBufferSize)
    //4
{

  static const int maxBufferSize = 0x50000;                                  //327k
```

```
    int maxPacketSize = ASBDescription->mBytesPerPacket;       // 6

    if (maxPacketSize == 0) {                                  // 7

        UInt32 maxVBRPacketSize = sizeof(maxPacketSize);

        AudioQueueGetProperty (audioQueue,
                               kAudioQueueProperty_MaximumOutputPacketSize,
                                                    // in Mac OS X v10.5, instead use
                                                    //
kAudioConverterPropertyMaximumOutputPacketSize
                                                &maxPacketSize,
                                                &maxVBRPacketSize);
    }



    Float64 numBytesForTime =

        ASBDescription->mSampleRate * maxPacketSize * seconds; // 327k * .5

    *outBufferSize =

    UInt32 (numBytesForTime < maxBufferSize ? numBytesForTime : maxBufferSize);
// 9

}
```

Implementacion.cpp

```
#include "Implementacion.h"
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <Accelerate/Accelerate.h>
#include <math.h>

using namespace std;

void GetTFs::InitBuffers(void * buffer, int number_of_samples)
{
        unsigned int x;

        sample_count = number_of_samples;
    printf("%i\n",sample_count);

        for(x = 0; x < sample_count; x++)
```

```
        {
          /* Init all buffers */

          sample_cp[x] = sample_ar[x] = (static_cast<int16_t  *> (buffer))[x];
        sample_cp[x]=sample_cp[x]/AUDIO_BUF_SIZE;
        sample_ar[x]=sample_ar[x]/AUDIO_BUF_SIZE;

          /* Imaginary starts out as all zeros */

          sample_ai[x] = 0;
        }

        /* Any leftover space zero out */

        for(; x < AUDIO_BUF_SIZE; x++) sample_cp[x] = sample_ai[x] = sample_ar[x] =
0;
}

void GetTFs::GetParametros(int Obtfrecmuestreo){

    fs = Obtfrecmuestreo;
    long_frec=4096;
    t_tramas=0.128;
    tsalto=0.064;
    //tsalto=0.001;
    windowsize=(t_tramas*fs/2)*2;
    salto=(tsalto*fs/2)*2;

    unsigned int i;
    for(i=0; i< windowsize; i++){
       ventana[i]=0.5*(1 - cos((2.0*M_PI*i)/(windowsize-1)));
    }
    overlap=windowsize-salto;
}

Float32** GetTFs::Sg(){
    int nx=sizeof(sample_cp)/sizeof(Float32);
    int nwin=sizeof(ventana)/sizeof(double);
    int ncol= (nx-overlap)/(nwin-overlap);
 samplefftr=new Float32*[AUDIO_BUF_SIZE/2];
    for (int i=0; i<(AUDIO_BUF_SIZE/2);i++) {
       samplefftr[i]=new Float32[colindex];
    }

    unsigned int j;
    unsigned int s;
    int p=0;
    j=0;
    s=0;
```

```
    for(j=0; j<ncol; j++){
        for(s=0; s<8192; s++){
            if (s<windowsize) {
                y[s][j]=sample_cp[s+p];
                samplefftr[s][j]=y[s][j]*ventana[s];
            } else {
                samplefftr[s][j]=0;
}
        }
       p=salto*j;
    }

    return samplefftr;
    for (int i = 0; i < AUDIO_BUF_SIZE/2; i++)
        delete samplefftr[i];
    delete [] samplefftr;

}


Float32** GetTFs::FFTransform(Float32 **samplefftr1){
       int numSamples = 8192;
    int col=colindex;
     yfftr=new Float32*[long2];
    for (int i=0; i<long2;i++) {
        yfftr[i]=new Float32[colindex];
    }
    for (int k=0; k<col; k++) {

 float *time = (float *)malloc(sizeof(float)*numSamples);
     for (int i=0; i<numSamples; i++) {
         time[i]=samplefftr1[i][k];
     }
         vDSP_Length log2n = log2f(numSamples);
         fftSetup=vDSP_create_fftsetup(log2n, FFT_RADIX2);
         int nOver2 = (numSamples/2);
         A.realp = (float *) malloc(nOver2*sizeof(float));
         A.imagp = (float *) malloc(nOver2*sizeof(float));
    int i;

    //Convert float array of reals samples to COMPLEX_SPLIT array A
         vDSP_ctoz((COMPLEX*)time,2,&A,1,numSamples/2);

    //Perform FFT using fftSetup and A
    //Results are returned in A
         vDSP_fft_zrip(fftSetup, &A, 1, log2n, FFT_FORWARD);

    //Convert COMPLEX_SPLIT A result to float array to be returned

         for(i=1;i<nOver2;i++){
```

```
            yfftr[i][k]=A.realp[i];
}
        free(A.realp);
        free(A.imagp);
        free(time);
        vDSP_destroy_fftsetup(fftSetup);
    }
    return yfftr;
  for (int i = 0; i <long2; i++)
        delete yfftr[i];

    delete [] yfftr;
}

Float32** GetTFs::FFTransform1(Float32 **samplefftr1){
    int numSamples = 8192;
    int col=colindex;
yffti=new Float32*[long2];
    for (int i=0; i<long2;i++) {
        yffti[i]=new Float32[colindex];
    }
    for (int k=0; k<col; k++) {

        float *time = (float *)malloc(sizeof(float)*numSamples);
        for (int i=0; i<numSamples; i++) {
            time[i]=samplefftr1[i][k];
        }

        vDSP_Length log2n = log2f(numSamples);
        fftSetup=vDSP_create_fftsetup(log2n, FFT_RADIX2);
        int nOver2 = (numSamples/2);
        A.realp = (float *) malloc(nOver2*sizeof(float));
        A.imagp = (float *) malloc(nOver2*sizeof(float));
        int i;

        //Convert float array of reals samples to COMPLEX_SPLIT array A
        vDSP_ctoz((COMPLEX*)time,2,&A,1,numSamples/2);

        //Perform FFT using fftSetup and A
        //Results are returned in A
        vDSP_fft_zrip(fftSetup, &A, 1, log2n, FFT_FORWARD);

        //Convert COMPLEX_SPLIT A result to float array to be returned

        for(i=1;i<nOver2;i++){
            yffti[i][k]=A.imagp[i];
        }
        free(A.realp);
        free(A.imagp);
        free(time);
```

```
        vDSP_destroy_fftsetup(fftSetup);
    }
    return yffti;
    for (int i = 0; i < long2; i++)
        delete yffti[i];

    delete [] yffti;

}
Float32** GetTFs::MetodoSuma(int midi,Float32 **yfftr1,Float32 **yffti1){

     cfreqr=new Float32*[colindex];
     for (int i=0; i<colindex;i++) {
     cfreqr[i]=new Float32[longmidi];
     }

    int step,kmin,kmax;
    float fmin,fmax;
    int long_frec=4096;
    int midi_inc=midi;
    int midi_min=24;
    int midi_max=ceil(12*log2((fs/2)/440)+69);
    int dif=midi_max-midi_min+1;
    int i=0;

    for (int n=midi_min; n<midi_max; n++) {
            step=1/midi_inc;
  float sum;
            sum=float((n+i*step-step/(float)2)-69)/12;
            fmin=pow(2,sum)*440;
            kmin=ceil(fmin/fs*(2*long_frec)+1);
            kmin=min(kmin,long_frec+1);
            sum=0;
            sum=float((n+i*step+step/(float)2)-69)/12;
            fmax=pow(2, sum)*440;
            kmax=round(fmax/fs*(2*long_frec)+1);
            kmax=min(kmax,long_frec);
            miditobinskmin[n-midi_min] = kmin;
            miditobinskmax[n-midi_min] = kmax;
}

    miditobinskmin[84]=long_frec;
   miditobinskmax[84]=long_frec+1;
    int muestras=dif;

    for (int x=0; x<muestras; x++) {
        //i<1921 se cambia por i<31
        for (int i=0; i<31; i++) {
            float a,b;
            float c,suma=0;
```

```
      kmin=miditobinskmin[x];
      kmax=miditobinskmax[x];
      if ((kmax-kmin)==0) {
         //int a,b;
         a=yfftr1[kmin][i];
         b=yffti1[kmin][i];
         cfreqr[i][x]=pow(a,2)+pow(b,2);

      } else {
         //int a,b;
         int j=0;
         for (j=kmin; j<kmax; j++) {

            a=yfftr1[j][i];

            b=yffti1[j][i];
            c=(pow(a, 2)+pow(b, 2));
            suma=suma+(pow(a, 2)+pow(b, 2));
}
         cfreqr[i][x]=suma;

      }
   }
   }
   return cfreqr;
   for (int i = 0; i < colindex; i++)
      delete cfreqr[i];

   delete [] cfreqr;
}


Float32* GetTFs::nlsfast(Float32 **cfreqr1){

   sumb = new Float32 [longmidi];
   int n1=31;
   int n2=85;
   float s1,s2,s3,s4,s5,s6=0;
   for (int i=0; i<n1; i++) {
      for (int j=0; j<n2; j++) {
         if (j<41) {
            if (j==16) {

            b[i][j]=pow(cfreqr1[i][j], 2)+pow(cfreqr1[i][j+12], 2)+pow(cfreqr1[i][j]+28, 2);
            s1=s1+b[i][j];
            sumb[j]=s1;
         }else{
            if (j==21) {
               b[i][j]=pow(cfreqr1[i][j], 2)+pow(cfreqr1[i][j+12], 2)+pow(cfreqr1[i][j]+28,
2);
```

```
                s2=s2+b[i][j];
                sumb[j]=s2;
            } else {
                if (j==26) {
                    b[i][j]=pow(cfreqr1[i][j], 2)+pow(cfreqr1[i][j+12],
2)+pow(cfreqr1[i][j]+34, 2);
                    s3=s3+b[i][j];
                    sumb[j]=s3;

                } else {
                    if (j==31) {
                        b[i][j]=pow(cfreqr1[i][j], 2)+pow(cfreqr1[i][j+12],
2)+pow(cfreqr1[i][j]+34, 2)+pow(cfreqr1[i][j]+36, 2);
                        s4=s4+b[i][j];
                        sumb[j]=s4;
                    } else {
                        if (j==35) {
                            b[i][j]=pow(cfreqr1[i][j]+28, 2)+pow(cfreqr1[i][j+12]+38, 2);
                            s5=s5+b[i][j];
                            sumb[j]=s5;

                        } else {
                            if (j==40) {
                                b[i][j]=pow(cfreqr1[i][j]+28, 2)+pow(cfreqr1[i][j+12]+34,
2)+pow(cfreqr1[i][j]+36, 2)+pow(cfreqr1[i][j]+38, 2);
                                s6=s6+b[i][j];
                                sumb[j]=s6;

                            }
                        }
                    }
                }
            }
        }
    }
   }
   return sumb;
   delete [] sumb;
}

int* GetTFs::Obtnotas(float *sumb1,int umb){
      int pos[6]={16,21,26,31,35,40};
   int cont=0;
  for (int i=0; i<6; i++) {
     x[i]=sumb1[pos[i]];

  }
   for (int i=0; i<6; i++) {
      s[i]=0;
```

```
    }

      NSLog(@"%d",umb);

    for (int j=0; j<6; j++) {
       cout<<x[j]<<" ";
       cout<<endl;
    }

    float
umbrales0,umbrales1,umbrales2,umbrales3,umbrales4,umbrales5,umbrales6,umbrales7;
    if (umb==1) {

       umbrales0= 2000000;
       umbrales1=100000000;
       umbrales2=500000000;
       umbrales3=1000000000;
       umbrales4=100000000;
       umbrales5=6000000000;
       umbrales6=10000000;
       umbrales7=10000000000;

}else if (umb==2){

       umbrales0= 100000000;
       umbrales1= 100000000;
       umbrales2= 100000000;
       umbrales3= 1000000000;
       umbrales4= 10000000000;
       umbrales5= 80000000000;
       umbrales6= 1000000000;
       umbrales7= 150000000000;

    }else if (umb==3){

       umbrales0= 100000000000;
       umbrales1= 700000000000;
       umbrales2= 70000000000000;
       umbrales3= 700000000000;
       umbrales4= 1000000000000;
       umbrales5= 100000000000000;
       umbrales6= 1000000000000;
       umbrales7= 100000000000000;

    }


    for (int j=0; j<6; j++) {
       if (j==0) {
```

```
        if ((x[j] > umbrales0)){
            s[j]=pos[j];
            cont++;
        }
    }else if (j==1) {
        if (x[j] >umbrales1) {
            s[j]=pos[j];
            cont++;
                    }
    }
    else if (j==2) {
        if (x[j] > umbrales2) {
            s[j]=pos[j];
            cont++;
                 }
    }
    else if (j==3) {
        if (x[j] > umbrales3) {
            s[j]=pos[j];
            cont++;
        }
    }
        else if (j==4) {
            if (cont==0) {
                if ((x[j]>umbrales4)) {

                s[j]=pos[j];
                cont++;}
            }
            else if (cont!=0) {
                if ((x[j]>umbrales5)){
                    s[j]=pos[j];
                cont++;
                }
            }

        }

        else{

            if ((cont==0)&&(x[j]>umbrales6)){
                  s[j]=pos[j];
                                }

            else if ((cont!=0)&&(x[j]>umbrales7)){

                   s[j]=pos[j];
            }
        }
}
```

```
    elems=0;
    for (int i =0; i<6; i++) {
        if(s[i]==0){
            elems=elems+1;
        }
    }
    elems=6-elems;

    sarray = new int [elems];
    int j=0;
    for (int i=0; i<6; i++) {
        if (s[i]==16) {
            sarray[j]=s[i];
            j=j+1;
        }else if (s[i]==21){
            sarray[j]=s[i];
            j=j+1;
        }else if (s[i]==26){
            sarray[j]=s[i];
            j=j+1;
        }else if (s[i]==31){
            sarray[j]=s[i];
            j=j+1;
        }else if (s[i]==35){
            sarray[j]=s[i];
            j=j+1;
        }else if (s[i]==40){
            sarray[j]=s[i];
            j=j+1;
        }
    }

    return sarray;
    delete [] sarray;
}


int GetTFs::longitud(){

    int tam1=0;

    for (int i =0; i<6; i++) {
        if(s[i]==0){
            tam1=tam1+1;
        }
    }
    tam1=6-tam1;
    return tam1;
```

```
}

float *GetTFs::defrec(int tamano){


   double ventana[AUDIO_BUF_SIZE];
   for(int i=0; i< AUDIO_BUF_SIZE; i++){
      ventana[i]=0.5*(1 - cos((2.0*M_PI*i)/(AUDIO_BUF_SIZE-1)));
   }
   for(int j=0; j<AUDIO_BUF_SIZE; j++){


      samplecfreq[j]=sample_cp[j]*ventana[j];
   }

   int numSamples = AUDIO_BUF_SIZE;

      float *time = (float *)malloc(sizeof(float)*numSamples);
      for (int i=0; i<numSamples; i++) {
         time[i]=samplecfreq[i];
      }

      vDSP_Length log2n = log2f(numSamples);
      fftSetup=vDSP_create_fftsetup(log2n, FFT_RADIX2);
      int nOver2 = (numSamples/2);
      A.realp = (float *) malloc(nOver2*sizeof(float));
      A.imagp = (float *) malloc(nOver2*sizeof(float));
      int i;

      //Convert float array of reals samples to COMPLEX_SPLIT array A
      vDSP_ctoz((COMPLEX*)time,2,&A,1,numSamples/2);

      //Perform FFT using fftSetup and A
      //Results are returned in A
      vDSP_fft_zrip(fftSetup, &A, 1, log2n, FFT_FORWARD);

      //Convert COMPLEX_SPLIT A result to float array to be returned

      for(i=1;i<nOver2;i++){
         yfftcreqr[i]=A.realp[i];
         yfftcreqi[i]=A.imagp[i];
      }
      free(A.realp);
      free(A.imagp);
   free(time);
      vDSP_destroy_fftsetup(fftSetup);

for(int i=0; i<(numSamples/2); i++){
float a,b=0;
   a=yfftcreqr[i];
```

```
    b=yfftcreqi[i];
    cdefrec[i]=sqrt(pow(a, 2)+pow(b, 2));
}
    int notas[]={16,21,26,31,35,40};
    int mult[]={0,12,19,24,28,31,34,36,38};
    int q[6][9];
    for (int j=0; j<6; j++) {
for (int i=0; i<9; i++) {
            q[j][i]=mult[i];
        }

    }
    for (int i=0; i<9; i++) {

        for (int j=0; j<6; j++) {
            q[j][i]=q[j][i]+notas[j];
        }
    }

 int f[6][9]={{16,28,0,0,44,0,0,0,0},{21,33,0,0,49,0,0,0,0},{26,38,0,0,0,0,60,0,0},
{31,43,0,0,0,0,65,67,0},{0,0,0,0,63,66,0,0,73},{0,0,0,0,68,0,74,76,78}};
    int s1[elems][9];
    for (int i=0; i<elems; i++) {
        if (s[i]==16) {
            for (int j=0; j<9; j++) {
                s1[i][j]=f[i][j];
            }
        }else if (s[i]==21){
            for (int j=0; j<9; j++) {
                s1[i][j]=f[i][j];
            }
        }else if (s[i]==26)
        {
            for (int j=0; j<9; j++) {
                s1[i][j]=f[i][j];
            }

        }else if (s[i]==31){
            for (int j=0; j<9; j++) {
                s1[i][j]=f[i][j];
            }
        }else if (s[i]==35){
            for (int j=0; j<9; j++) {
                s1[i][j]=f[i][j];
            }
        }else if (s[i]==40){
            for (int j=0; j<9; j++) {
                s1[i][j]=f[i][j];
            }
        }
```

```
    }
    float f0s[6][5]={{82.406,164.812,412.03,0,0},{110,220,
550,0,0},{146.8323,293.6646,1027.8261,0,0},{195.9977,391.9954,1371.9839,1567.9816,
0},{0,1234.708,1481.6496,2222.4744,0},{0,1648.1375,2307.392,2637.02,2966.6475}};
    int tam1;
    tam1=tamano;
    float frec[tam1][5];
for (int i=0; i<tam1; i++) {
        if (sarray[i]==16) {
            for (int j=0; j<5; j++) {
                frec[i][j]=f0s[0][j];
            }
        }else if (sarray[i]==21){
            for (int j=0; j<5; j++) {
                frec[i][j]=f0s[1][j];
            }
        }else if (sarray[i]==26){
            for (int j=0; j<5; j++) {
                frec[i][j]=f0s[2][j];
            }

        }else if (sarray[i]==31){
            for (int j=0; j<5; j++) {
                frec[i][j]=f0s[3][j];
            }
        }else if (sarray[i]==35){
            for (int j=0; j<5; j++) {
                frec[i][j]=f0s[4][j];
            }
        }else if (sarray[i]==40){
            for (int j=0; j<5; j++) {
                frec[i][j]=f0s[5][j];
            }
        }
    }
float fx[5][3];
    float mult1[5][3];
    float maxs[5]={0,0,0,0,0};
    float maxsk[5]={0,0,0,0,0};
    int lim1,lim2;
 float frecdef[tam1][5],desviacion[tam1][5],resto[tam1][5];

    for ( int p=0; p<tam1; p++) {
    for (int s=0; s<5; s++) {
        fx[s][0]=frec[p][s]*pow(2,-1/(float)24);
        fx[s][1]=frec[p][s];
        fx[s][2]=frec[p][s]*pow(2,1/(float)24);

    }
```

```
    for (int c=0; c<5; c++) {
       for (int j=0; j<3; j++) {
          mult1[c][j]=round(fx[c][j]*AUDIO_BUF_SIZE/(float)fs)+1;
       }
    }


    for (int d=0; d<5; d++) {
       float max=0;
       lim1=mult1[d][0];
       lim2=mult1[d][2];
       for (int i= lim1; i < lim2; i++) {
          max=cdefrec[i];
          if(max>maxs[d]){
             maxs[d]=max;
             maxsk[d]=i ;
          }
}
    }

    for (int o=0; o<5; o++) {
       if(maxsk[o]!=0){
       frecdef[p][o]=maxsk[o]*fs/(float)AUDIO_BUF_SIZE;
       }else
          frecdef[p][o]=0;
    }

    for (int t=0; t<5; t++) {
       resto[p][t]=abs(fx[t][1]-fx[t][2]);
       desviacion[p][t]=(((frecdef[p][t]-frec[p][t])/(float)resto[p][t])/(float)2)*100;
    }
       for (int i=0; i<5; i++) {
          maxs[i]=0;
          maxsk[i]=0;

       }
}

    float destotal[tam1];
  float suma=0;
    float cont=0;
    float a=0;
    for (int i=0; i<tam1; i++) {
       suma=0;
       cont=0;
       for (int j=0; j<5; j++) {

       if (isnan(desviacion[i][j])!=1) {
             if (isinf(desviacion[i][j])!=1) {
```

```
            a=desviacion[i][j];
            suma=suma+a;
            cont++;
            }
         }

      }
      destotal[i]=suma/cont;;
   }

   for (int j=0; j<tam1; j++) {
      cout<<destotal[j]<<" ";
      cout<<endl;
   }
return destotal;
}

int GetTFs::detector(Float32 **cfreqrV){
   Float32 *sumbdet;
   sumbdet = new Float32 [longmidi];
   Float32 suma,a;
   Float32 sumatorio=0;
   mayor=0;
   supera=0;

   for (int i=0; i<longmidi; i++) {
      suma=0;
      for (int j=0; j<colindex; j++) {

         a=cfreqrV[j][i];
suma=suma+a;

      }
      sumbdet[i]=suma;
      sumatorio+=sumbdet[i];
      if (sumbdet[i]>mayor) {
         mayor=sumbdet[i];
      }
   }
   valormedio=sumatorio/(float)longmidi;
    for (int i=0; i<longmidi; i++) {

       if (sumbdet[i]>valormedio) {
          supera++;
       }
    }

   return supera;
   delete [] sumbdet;
```

```
}
```

ViewController.mm

```
#import "ViewController.h"
#import "AudioInput.h"

@interface ViewController ()
@end

@implementation ViewController

GetTFs getTFs;
int tam = 0;
int *notas;
float *des;
int umbnotas=1;

- (void)viewDidLoad
{

    [super viewDidLoad];

        // Do any additional setup after loading the view, typically from a nib.
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
        pthread_t buf_update_thread;

        // Setup audio input

        CTAudioBuffer *audioInput = InitAudio();

    pthread_create(&buf_update_thread, NULL, &analyzeData, audioInput);

    [NSTimer scheduledTimerWithTimeInterval:1.0f target:self
selector:@selector(updateCTDisplay:) userInfo: nil repeats:YES];

    [super viewDidLoad];
  [pool release];
}

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

void * analyzeData(void * thread_data)
{
        CTAudioBuffer * audiobuffer_data = static_cast<CTAudioBuffer *> (thread_data);
```

```
        while(1) // Do forever
        {
          // Protect critical section until data is availible

audiobuffer_data->Synchronize();
       int frecuencia =audiobuffer_data->Obtfrmuestreo();

          // Syncronize FFT buffers

          getTFs.InitBuffers(audiobuffer_data->AudioBuffer(),audiobuffer_data-
>NumberofSamples());

        getTFs.GetParametros(frecuencia);
       Float32 **sampleff;
       Float32 **yfft1,**yfft2;

       sampleff=getTFs.Sg();
       yfft1=getTFs.FFTransform(sampleff);
       yfft2=getTFs.FFTransform1(sampleff);

       for (int i = 0; i < AUDIO_BUF_SIZE/2; i++)
          delete sampleff[i];

       delete [] sampleff;

       Float32** cfreq1;

       cfreq1=getTFs.MetodoSuma(1,yfft1,yfft2);

       for (int i = 0; i < long2; i++)
          delete yfft1[i];
delete [] yfft1;

       for (int i = 0; i < long2; i++)
          delete yfft2[i];

       delete [] yfft2;
       Float32 *sumb1;
       int umb;
       umb=getTFs.detector(cfreq1);

       if (umb<20) {

       sumb1=getTFs.nlsfast(cfreq1);

          //i<1921 por i<31
          for (int i = 0; i < 31; i++)
          delete cfreq1[i];
       delete [] cfreq1;
```

```
notas=getTFs.Obtnotas(sumb1,umbnotas);
        tam=getTFs.longitud();

         NSLog(@"%d",tam);
            for (int i=0; i<tam; i++) {
            NSLog(@"%d",notas[i]);
            }
         delete [] sumb1;
        des=getTFs.defrec(tam);
        }
        else{
        for (int i = 0; i < 31; i++)
delete cfreq1[i];
        delete [] cfreq1;

        }
            audiobuffer_data->ReInit();
            audiobuffer_data->SetWriteOkay();

         }
    return NULL;
}

- (void)dealloc {

    [ledB release];
    [ledE1 release];
    [ledG release];
    [ledD release];
    [ledA release];
    [ledE2 release];
    [super dealloc];
}


- (void) catchAudioError
{
        exit(-1);
}

- (void) updateCTDisplay: (id) none{

        if (tam>0) {
            //[ledE1 setAlpha:1];
            for (int i=0; i<tam; i++) {
                if (notas[i]==16) {
                   [ledE2 setBackgroundColor:[UIColor blueColor]];
                     [ledE2  setFrame:CGRectMake(151+(2*des[i]),313,16 , 41)];
                }else if (notas[i]==21){
                    [ledA setBackgroundColor:[UIColor redColor]];
```

```
                [ledA setFrame:CGRectMake(151+(2*des[i]), 264,16 , 41)];
            }else if (notas[i]==26){
                [ledD setBackgroundColor:[UIColor yellowColor]];
               [ledD  setFrame:CGRectMake(152+(2*des[i]), 215,16 , 41)];
            }else if (notas[i]==31){
                [ledG setBackgroundColor:[UIColor orangeColor]];
               [ledG setFrame:CGRectMake(152+(2*des[i]), 163,16 , 41)];
            }else if (notas[i]==35){
                 [ledB setBackgroundColor:[UIColor greenColor]];
               [ledB  setFrame:CGRectMake(152+(2*des[i]), 114,16 , 41)];
            }else if (notas[i]==40){
                [ledE1 setBackgroundColor:[UIColor blackColor]];
               [ledE1  setFrame:CGRectMake(151+(2*des[i]), 65,16 , 41)];
            }
        }
        tam=0;
    }else{

    ledE1.backgroundColor=[UIColor whiteColor];
  ledB.backgroundColor=[UIColor whiteColor];
        ledA.backgroundColor=[UIColor whiteColor];
        ledG.backgroundColor=[UIColor whiteColor];

        ledD.backgroundColor=[UIColor whiteColor];
        ledE2.backgroundColor=[UIColor whiteColor];
}
}
- (IBAction)cambiarImagen:(id)sender {
  //ViewController2 *controller=[[ViewController2 alloc]init];
  //int uvalor=0;
  //controller.umbral=1;
  UIButton *boton = (UIButton *) sender;
  int tag = boton.tag;
  switch (tag) {
    case 1:
      umbnotas=1;
      break;
    case 2:
      umbnotas=2;
      break;
    case 3:
umbnotas=3;
      break;
    default:
      break;
  }
  NSLog(@"%d",umbnotas);
  self.valueLabel.text=[NSString stringWithFormat:@"%2d",umbnotas];
}
@end
```

Librerías.

AppDelegate.h.

```
#import <UIKit/UIKit.h>

@interface AppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@end
```

AudioInput.h

```
#ifndef AUDIOINPUT_H
#define AUDIOINPUT_H
#include <pthread.h>
#include <string.h>
#include <unistd.h>
#include <semaphore.h>
#include "AudioToolbox/AudioToolbox.h"

static const int kNumberBuffers = 3;

typedef struct AudioStreamData
{
   AudioStreamBasicDescription dataFormat;
   AudioQueueRef queue;
   SInt64 currentPacket;
   AudioQueueBufferRef buffers[3];
   UInt32 bufferByteSize;
   bool mIsRunning;
}AudioStreamData;


class CTAudioBuffer
{
private:
       bool write_flag;                // A read/write buffer flag
       pthread_mutex_t flag_lock;      // var lock for read/write
       pthread_mutex_t spin_lock;      // var lock for syncronization


protected:
public:
```

```
    int16_t  * buffer;                          // Shared buffer
  int buf_index;                          // Current shared buffer write index;

  AudioStreamData audioStreamData;

    void ReInit();
  public :      CTAudioBuffer();
    void SetWriteOkay();
    void SetReadOkay();
    void Synchronize();

    inline int16_t  * AudioBuffer(){ return buffer; };
    inline int AudioBufferSize(){ return (buf_index*sizeof(int16_t )); };
    inline int NumberofSamples(){ return (buf_index); };
   inline int Obtfrmuestreo(){return (audioStreamData.dataFormat.mSampleRate);};
    inline bool WriteStatus(){ return write_flag; };

};

void DeriveBufferSize (AudioQueueRef             audioQueue,             // 1
                       AudioStreamBasicDescription   * ASBDescription,
// 2
                       Float64                 seconds,                // 3
                       UInt32                  *outBufferSize);        // 4

static void HandleInputBuffer (void *userData,
                               AudioQueueRef inAQ,
                               AudioQueueBufferRef inBuffer,
                               const AudioTimeStamp *inStartTime,
                               UInt32 inNumPackets,
                               AudioStreamPacketDescription
*inPacketDesc);

CTAudioBuffer * InitAudio(void);

#endif
```

Implementacion.h

```
#ifndef IMPLEMENTACION_H
#define IMPLEMENTACION_H
#import <AudioUnit/AudioUnit.h>
#import <Foundation/Foundation.h>
#import <Accelerate/Accelerate.h>
#import <math.h>
#define AUDIO_BUF_SIZE 16384
#define long2 4097
```

```
class GetTFs{

private:

        int fs;
        int long_frec;
        int n_tramas;
        float t_tramas;
        int windowsize;
        int salto;
        double ventana[1024];
        float tsalto;
     int overlap;
    int tam;
    int colindex=31;
    int rowindex=1024;
    int longmidi=85;
         int sample_count;
         int sampled_freqency;
    COMPLEX_SPLIT A;
    FFTSetup fftSetup;
    Float32 y[1024][31];
    int miditobinskmin[85];
    int miditobinskmax[85];

public:

// FFT buffers

    Float32  sample_ar[AUDIO_BUF_SIZE];
Float32  sample_ai[AUDIO_BUF_SIZE];
    Float32 **samplefftr=NULL;
    Float32 **yfftr=NULL;
    Float32 **yffti=NULL;
    Float32 samplecfreq[AUDIO_BUF_SIZE];
    Float32 **cfreqr;
    Float32 b[31][85];
 Float32 sumbdet[85];
    Float32 *sumb;
    Float32 valormedio;
    Float32 mayor;
    int supera;
    int s[6];
    int *sarray;
    int elems;
    Float32 x[6];
    Float32 yfftcreqr[AUDIO_BUF_SIZE/2];
    Float32 yfftcreqi[AUDIO_BUF_SIZE/2];
    float cdefrec[AUDIO_BUF_SIZE/2];
```

```
// Local Buffer Copy

Float32  sample_cp[AUDIO_BUF_SIZE];
void InitBuffers(void * buffer, int buffer_size);
void GetParametros (int Obtfrecmuestreo);
   Float32** Sg();
   void zeros();
   Float32** FFTransform(Float32 **samplefftr1);
   Float32** FFTransform1(Float32 **samplefftr1);
   Float32** MetodoSuma(int midi,Float32 **yfftr1,Float32 **yffti1);
   Float32* nlsfast(Float32 **cfreqr1);
   int detector(Float32 **cfreqrV);
   int* Obtnotas(Float32 *sumb1,int umb);
   int longitud();
   float* defrec(int tamano);
};

#endif /* defined(__PruebaAfinador__Implementacion__) */
```

ViewController.h

```
#import <UIKit/UIKit.h>
#import "Implementacion.h"
@interface ViewController : UIViewController{

IBOutlet UIImageView *ledE1;

   IBOutlet UIImageView *ledB;

   IBOutlet UIImageView *ledG;

   IBOutlet UIImageView *ledD;

   IBOutlet UIImageView *ledA;

   IBOutlet UIImageView *ledE2;
}
- (IBAction)cambiarImagen:(id)sender;
@property (nonatomic) IBOutlet UILabel *valueLabel;
@property (nonatomic,assign) int myvalue;
void * analyzeData(void * thread_data);
void init();

@end
```

Manual de usuario.

Observando la Figura 31 se puede apreciar que el funcionamiento del afinador va a ser extremadamente sencillo. A pesar de esto vamos a desarrollar en unas cuantas líneas un pequeño manual de la misma.

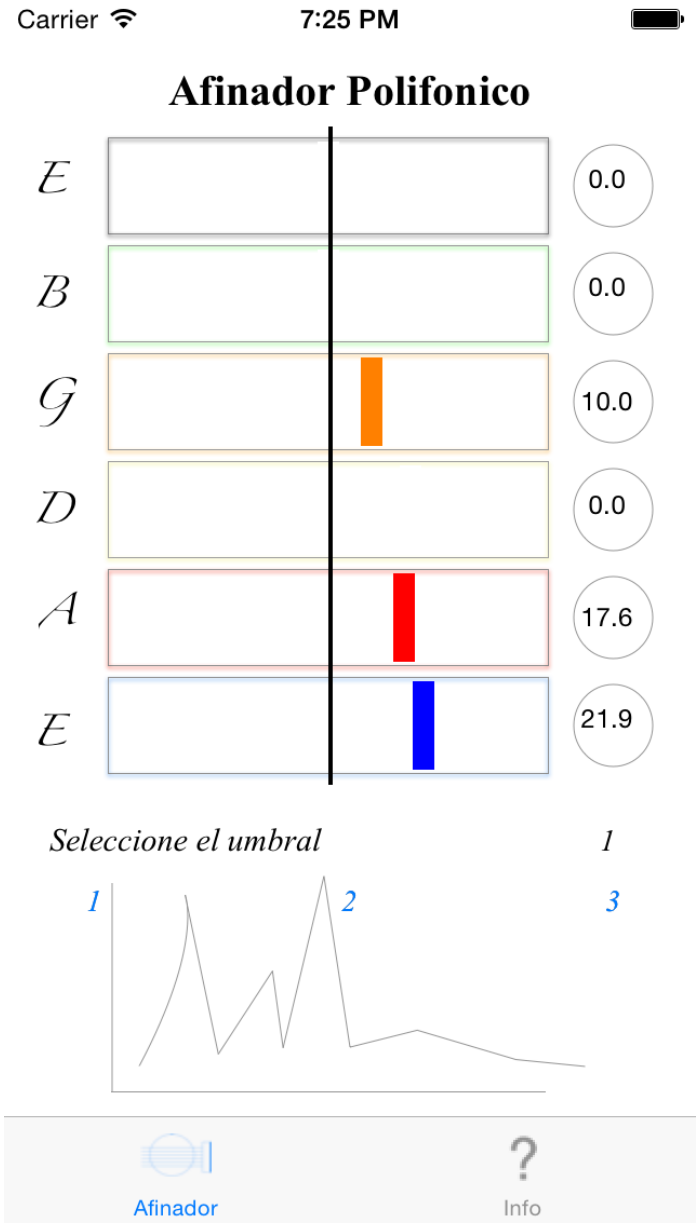

Figura 31. Primera vista (Afinador) de nuestra aplicación.

La aplicación va a constar de 2 vistas (afinador e información), en la primera vista que es la que se observa en la Figura 31 se mostraran los resultados de la afinación y la segunda vista solamente será un manual de la misma dentro de la app y se puede ver su aspecto en la Figura 32.

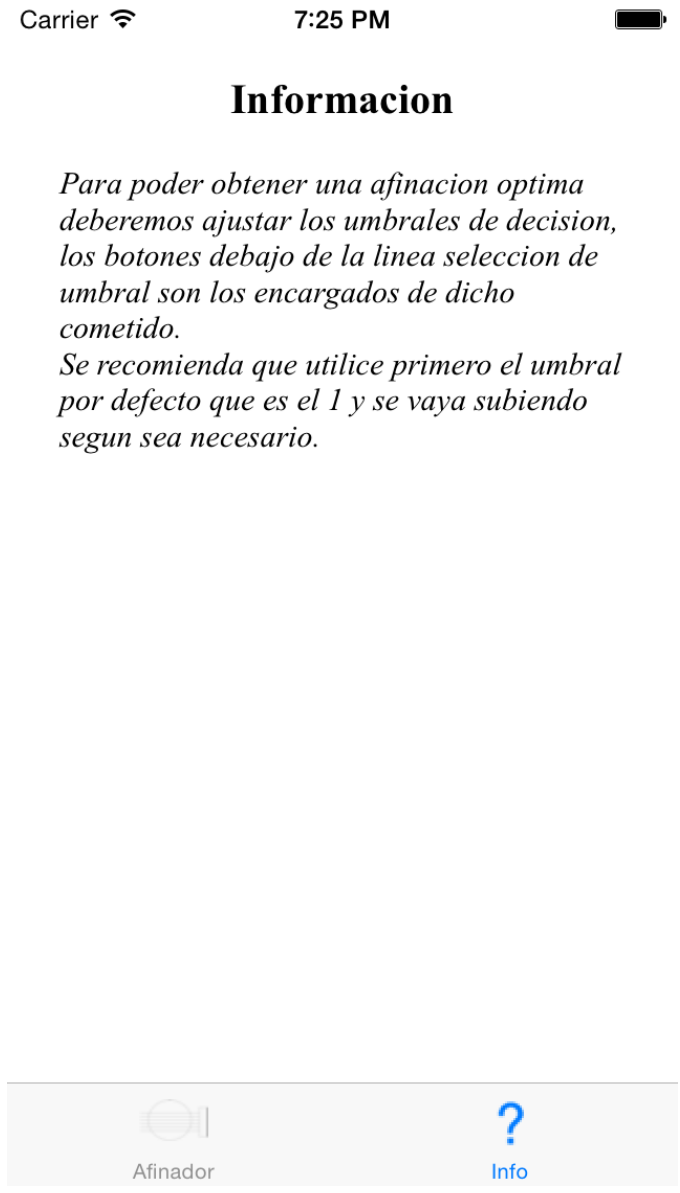

Figura 32. Segunda vista (info) de nuestra aplicación.

Para que la aplicación funcione no habrá que hacer nada más que el dispositivo reciba una señal sonora y automáticamente se encargara de analizarla. No obstante en la aplicación se podrá ajustar un parámetro que será el nivel de umbral si los resultados que se obtienen no son satisfactorios.

Para ajustar el umbral se recomienda que se toque una sola cuerda y se vaya probando del umbral más bajo al más alto en caso de que sea necesario ajustarlo. Hay que recodar que esta información también se incluye en la concretamente en la segunda vista la cual se ha comentado anteriormente.

Capítulo 8

Referencias bibliográficas

Sobre estimación Multi-Pitch

MADS, C. & ANDREAS, J. Multi-Pitch Estimation. United States: Morgan Claypool Publishers, 2009.

Sobre programación en C y Objective C.

CONWAY, J. & HILLEGASS, A. Programación iOS. España: Anaya Multimedia, 2011.

NAHAVANDIPOOR, V. iOS 6 Programming Cookbook. United States: O'Reilly Media, 2012.

SCHILDT, H. C: Manual de referencia. Mexico: S.A. MCGRAW-HILL / INTERAMERICANA DE ESPAÑA, 2001.

Apple: Concepts in Objective-C Programming[en linea]. Disponible en <https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/Introduction/Introduction.html#//apple_ref/doc/uid/TP40010810>

Sobre iOS y patrones de diseño.

ADAMSON, C. & AVILA, K. Learning Core Audio: A Hands-On Guide to Audio Programming for Mac and iOS. Indiana: Addison-Wesley, 2012.

Apple: Audio Queue Services Programming Guide[en linea]. Disponible en <https://developer.apple.com/library/ios/documentation/MusicAudio/Conceptual/AudioQueueProgrammingGuide/AudioQueueProgrammingGuide.pdf>

Apple: Audio SessionProgramming Guide [en linea]. Disponible en <https://developer.apple.com/library/ios/documentation/Audio/Conceptual/AudioSessionProgrammingGuide/AudioSessionProgrammingGuide.pdf>

Apple: iOS Technology Overview[en linea]. Disponible en <https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iOSTechOverview.pdf>

Apple: iOS App Programming[en linea]. Disponible en <https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007072>

Apple: Start Developing iOS Apps Today[en linea]. Disponible en <https://developer.apple.com/library/ios/referencelibrary/GettingStarted/RoadMapiOS/index.html#//apple_ref/doc/uid/TP40011343>

Wikipedia: iOS [en línea]. Disponible en <http://es.wikipedia.org/wiki/IOS>

Sobre frameworks de iOS.

Apple: Framework Programming Guide [en linea]. Disponible en <https://developer.apple.com/library/ios/documentation/MacOSX/Conceptual/BPFrameworks/Frameworks.html#//apple_ref/doc/uid/10000183i>

Apple: Accelerate Framework Reference [en linea]. Disponible en

<https://developer.apple.com/library/ios/documentation/Accelerate/Reference/AccelerateFWRef/_index.html#//apple_ref/doc/uid/TP40009465>

Apple: Core Audio Framework Reference [en linea]. Disponible en <https://developer.apple.com/library/ios/documentation/MusicAudio/Reference/CACoreAudioReference/CACoreAudioReference.pdf>

Apple: Foundation Framework Reference [en linea]. Disponible en <https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/ObjC_classic/_index.html#//apple_ref/doc/uid/20001091>

Otros

Universidad Carlos III de Madrid: Cómo citar bibliografía [en línea]. Disponible en <http://www.uc3m.es/portal/page/portal/biblioteca/aprende_usar/como_citar_bibliografia>