Terrassa, Junio de 2007 **EUETIT**

Mejora de un conversor de audio a MIDI e implementación en tiempo real

Proyecto Final de Carrera

Ingeniería Técnica de Telecomunicaciones
especialidad en Imagen y Sonido

Escola Universitària d'Enginyeria Tècnica Industrial de Terrassa
Universitat Politècnica de Catalunya

Autor: Aitor Pérez Pellitero
Tutor: Ignasi Esquerra Llucià

Departamento de Teoría de la Señal y Comunicaciones

Índice

1. Introducción

1.1 Introducción al PFC

El presente proyecto de final de carrera pretende desarrollar la mejora de un conversor de audio a MIDI e implementarlo para que su funcionamiento sea en tiempo real. Para ello partiremos de una entrada de audio continua, que será digitalizada y procesada a tiempo real. La orientación del proyecto abarca el mundo de la tecnología musical, por lo tanto, como es lógico, la fuente de sonido será producida por un instrumento, en nuestro caso una guitarra eléctrica. El resultado obtenido será que mientras nosotros realizamos y escuchamos nuestra interpretación con el instrumento, iremos obteniendo a la salida de la aplicación la interpretación pero en formato estándar MIDI. Es equivalente a decir que implementaremos un sistema de trascripción musical en tiempo real.

Esta conversión puede resultar bastante complicada, ya que los 2 formatos usados son completamente distintos.

Cuando hablamos de convertir formatos en informática nos referimos a adaptar un archivo diseñado para una aplicación, de manera que pueda leerse en otra correctamente. Para que esto pueda hacerse, deben darse unas similitudes fundamentales entre los formatos que usan esas dos aplicaciones. Por ejemplo, los formatos de imágenes digitales como BPM, JPG o GIF comparten algo: son imágenes digitales. Unos formatos son comprimidos, otros usan más o menos colores, pero en realidad todos están leyendo información digital de un gráfico.

En cambio, "convertir" audio real a MIDI no es una tarea tan simple.

El MIDI es un protocolo de comunicación entre aparatos musicales. MIDI en sí mismo no produce sonido alguno. Consiste en mensajes que se dirigen de unos dispositivos MIDI a otros, y que contienen información descriptiva que indica qué nota debe sonar, a qué volumen,…etc. Por lo tanto, los mensajes MIDI no contienen información sobre la forma de onda de una señal digitalizada, sino que solo contienen eventos musicales, como una partitura.

Mucha gente cree que un fichero MIDI suena porque sí solo, que lleva los sonidos dentro como los puede llevar cualquier otro fichero de audio real, dado que ellos no tienen un sintetizador en casa, sólo el ordenador. Esto no es verdad. Todos los ordenadores actuales llevan sintetizadores incorporados, ya sea en su tarjeta de sonido o virtuales. Cuando reproduces un fichero MIDI, éste le indica al sintetizador de tu ordenador cómo debe sonar. Por eso, si cambias de tarjeta de sonido o usas otro sintetizador virtual diferente, los ficheros MIDI sonarán diferentes.

A continuación podemos ver en la figura 1.1 representados distintos eventos MIDI.

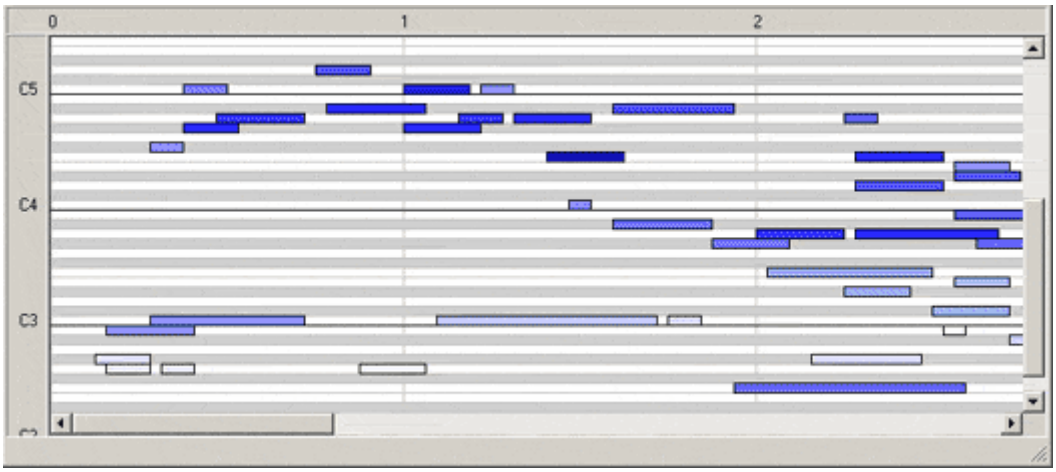


Fig.1.1 *- Ejemplo eventos MIDI*

En cambio, un fichero de audio digitalizado como puede ser un WAV**,** no tiene nada que ver con uno en formato MIDI. El WAV también es información digital, es decir, ceros y unos, pero el tipo de información que representan estos es completamente distinta. Un fichero de este tipo contiene la forma de onda digitalizada de una señal real de audio.

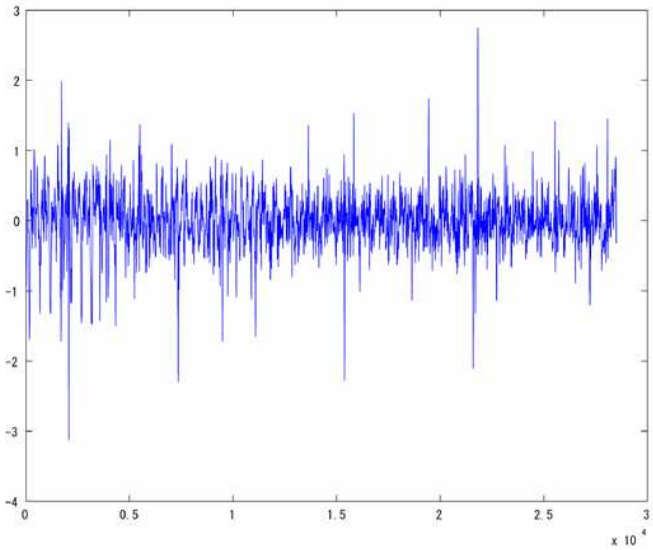


Fig. 1.2 *- Ejemplo representación señal de audio*

Aquí no hay ningún protocolo o lenguaje que le diga a ningún dispositivo cómo debe sonar, el WAV en sí mismo es el propio sonido capturado, no es una cadena de mensajes que se envían a un sintetizador, como el MIDI.

Visto lo anterior, podemos comprobar que realizar la tarea de convertir un fichero de audio real a formato MIDI puede resultar complicado, ya que aparentemente no hay ninguna relación entre los dos formatos.

Gran parte de ésta dificultad reside en que un fichero de audio contiene información muy compleja sobre frecuencias, volúmenes, resolución..., y toda ella aparece mezclada. El audio no entiende de notas ni de escalas, sólo de frecuencias y de sonido puro, y ya sabemos que el MIDI consiste precisamente en mensajes que indican notas y eventos.

Hay que resaltar que el conversor que desarrollaremos será monofónico, por lo tanto solo aceptará una nota a la vez. Esto es debido a que al tocar varias notas o acordes la detección de pitch mediante análisis espectral se complica mucho. Hoy en día en el mercado existen pastillas de guitarra que separan el sonido que produce cada una de las cuerdas de forma independientemente. Así si se podría implementar con nuestro método un conversor polifónico.

Por lo tanto para realizar la implementación deberemos hacer empleo de distintas herramientas y conocimientos, todos relacionados con el mundo de la ingeniería y de la tecnología musical.

1.2 Motivación

La motivación principal de desarrollar este proyecto viene dada por mi interés dentro del campo de la música y su relación con la tecnología. Desde pequeño siempre he estado vinculado a la música, ya que toco distintos instrumentos, y en mis últimos años, al ingresar en ésta carrera de telecomunicaciones en la cual se tratan aspectos relacionados con la imagen y el sonido, ha suscitado en mí un cierto interés por fusionar estas 2 disciplinas con la finalidad de obtener resultados interesantes.

1.3 Punto de partida

El proyecto que desarrollaré pretende mejorar uno anterior realizado por Alberto Molina, un ex-alumno de la titulación. Su proyecto cargaba un fichero completo de audio WAV en memoria y realizaba un análisis por tramas detectando para cada una el pitch y las notas, asignando a estas siempre una duración constante independientemente de su duración. Por otra parte también se producían algunos errores en la detección de notas debido a la configuración del sistema. Finalmente los parámetros extraídos eran convertidos a información MIDI y, una vez acabado todo el procesado del fichero WAV, se generaba el fichero MIDI de salida.

Para implementar el proyecto utilizaré algunos de esos algoritmos, los cuales podré modificar y variar con la finalidad de crear un sistema que cumpla los objetivos fijados.

1.4 Objetivos

A continuación se exponen los objetivos principales del proyecto.

- ✓ Procesado de audio en tiempo real
- ✓ Procesado de información MIDI en tiempo real
- ✓ Detección de notas y su duración para implementación en MIDI
- ✓ Detección de silencios
- ✓ Interfaz gráfica Matlab
- ✓ Mejora a nivel de aplicación

1.5 Estructura del proyecto

Durante el desarrollo del proyecto dividiremos éste en distintas partes. A grandes rasgos podemos observar que tenemos diferentes temáticas, en las que trataremos la programación en tiempo real mediante Visual C++ y librerías, el análisis y procesado de la señal de audio para extraer sus parámetros, convertir estos a protocolo MIDI…etc.

A continuación se adjunta un pequeño esquema en el cual se pueden apreciar los distintos bloques generales que conforman el diseño final del sistema.

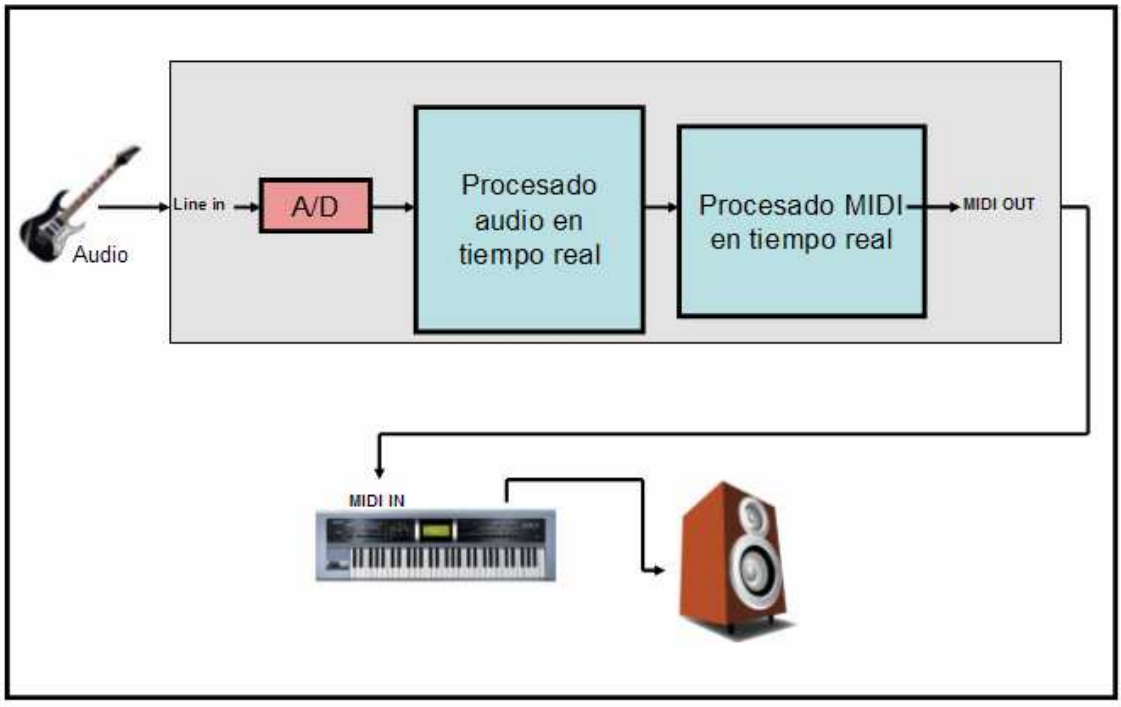


Fig. 1.3 *– Esquema bloques generales*

El primer bloque que realizaremos será el de adquirir muestras de audio de la línea de entrada de la tarjeta de sonido en tiempo real. Para ello deberemos programar en entorno Visual C++ una aplicación que vaya tomando bloques de muestras de entrada consecutivamente para que estos puedan ser procesados. Esta parte la desarrollaremos mediante el uso de la librería PortAudio, la cual tendremos que estudiar para aprender su funcionamiento.

La siguiente tarea consistirá en realizar un estudio y análisis del tipo de señal de audio que trataremos en el procesado. En nuestro caso realizaremos un estudio de la señal de guitarra eléctrica con la finalidad de conocer sus características y adaptar el sistema a estas, consiguiendo un funcionamiento lo más óptimo posible.

Continuaremos con el procesado de audio. Mediante un análisis frecuencial de cada uno de los bloques de muestras entrantes extraeremos las notas que han sido tocadas. Para ello haremos empleo de funciones de análisis frecuencial y detección de notas.

El siguiente paso consistirá en realizar la conversión a MIDI. Una vez realizado el procesado de cada bloque y extraídos sus parámetros, deberemos convertir estos a protocolo MIDI y enviarlos por el puerto de salida MIDI OUT en tiempo real. Para ello haremos empleo de la librería PortMidi, que estudiaremos más adelante.

Una vez realizado todo lo anterior ya tendremos a la salida del sistema la información MIDI resultante de la conversión de audio de la guitarra eléctrica en tiempo real. Solo quedará conectar la salida (MIDI OUT) a una entrada (MIDI IN) de un módulo sintetizador si se quiere escuchar la información MIDI convertida con algún tipo de sonido. También se puede considerar la opción de sintetizar esa información mediante software en la misma máquina.

2. Estado del arte

2.1 La tecnología musical

La tecnología musical, en nuestras últimas décadas, ha crecido de forma espectacular y ha dado grandes cambios respeto a sus inicios. En los comienzos de ésta teníamos grandes estudios de grabación, repletos de hardware con un coste bastante elevado y solo al alcance de muy pocos. No obstante, se han producido grandes cambios en esta evolución, substituyendo ése hardware por programas informáticos (software) de coste más moderado y accesibles a un mayor público. Con estos programas se pueden igualar o incluso superar los resultados que se obtenían y se obtienen mediante los sistemas hardware.

Algunos de estos nuevos sistemas de software musical se valen pese a todo de algún tipo de controlador hardware con el cual podemos introducir notas, melodías o acordes que luego podremos tratar y reproducir en el sistema informático. Gracias al estándar MIDI se pudieron crear este tipo de controladores, bajo la forma de teclados, baterías e incluso guitarras especiales de forma que el músico pudiera introducir notas al sistema musical informático.

Todo y esto, con la reciente evolución del software musical incluso es posible utilizar un instrumento convencional como controlador MIDI, como veremos en éste proyecto. El sistema parte de conectar el instrumento vía entrada de línea de una tarjeta de sonido, procesar su sonido para reconocer las notas pulsadas y demás parámetros necesarios y pasar estos datos a información MIDI. Podemos afirmar que se trataría de un sistema de trascripción musical.

2.2 La trascripción musical

Analizando el campo y la historia de la trascripción musical, podemos decir que los primeros intentos de trascripción polifónica se remontan a los setenta, cuando Moorer construyó un sistema para transcribir dúos, es decir, composiciones a dos voces. Su sistema sufría fuertes limitaciones en cuanto a la relación de frecuencias permitidas entre dos notas simultáneas y en cuanto al rango de notas utilizadas.

Uno de los avances significativos en la historia de la trascripción automática se hizo cuando un grupo de investigadores incluyó nuevas técnicas, utilizando de manera precisa reglas de separación auditiva, es decir, indicaciones auditivas que intentan tanto la fusión como la segregación de componentes de frecuencia simultánea en una señal.

Posteriormente introdujeron el procesamiento basado en modelos tonales (utilizando información sobre sonidos de los instrumentos en el procesamiento) y propusieron un algoritmo para modelar tonos automáticamente. Este algoritmo consiste en la extracción automática de modelos tonales a partir de la señal analizada.

Actualmente la trascripción musical abarca distintas técnicas de análisis: métodos en el dominio del tiempo, métodos en el dominio de la frecuencia, modelos tonales, modelos auditivos…etc. De las técnicas comentadas anteriormente, sólo la de modelos tonales ha tenido verdadero impacto en el campo de la trascripción polifónica. En los últimos años se han producido grandes avances en este campo, llegando incluso a intentos de trascripción de melodías que provienen de fuentes politímbricas, con ruido, con sonidos inarmónicos o con desviaciones de altura leves.

2.3 Tecnologías Actuales

En el mercado actual han aparecido diversos programas capaces de realizar una buena trascripción con monofonías (una sola nota a la vez), pero aún no ha salido ningún sistema de trascripción capaz de resolver completamente el problema de la trascripción musical polifónica, todo y que en los últimos años se han realizado importantes esfuerzos a nivel comercial.

La mayor parte de estos sistemas proporcionan la conversión entre formatos WAV (audio digital) y MIDI, e incluso algunos también realizan la trascripción en tiempo real (gran parte de éste proyecto tratará sobre realizar la implementación en tiempo real).

Analizando el mercado actual de éste tipo de software, podemos encontrar diversos programas:

WIDI

Widi es un programa que goza de muchas funcionalidades y opciones de configuración a la hora de realizar una trascripción musical, obteniendo unos resultados muy satisfactorios. Desde cargar un fichero de audio real para transcribirlo a MIDI, grabar una composición para transcribirla automáticamente, reconocimiento en tiempo real, y múltiples parámetros de configuración a la hora de realizar la conversión.

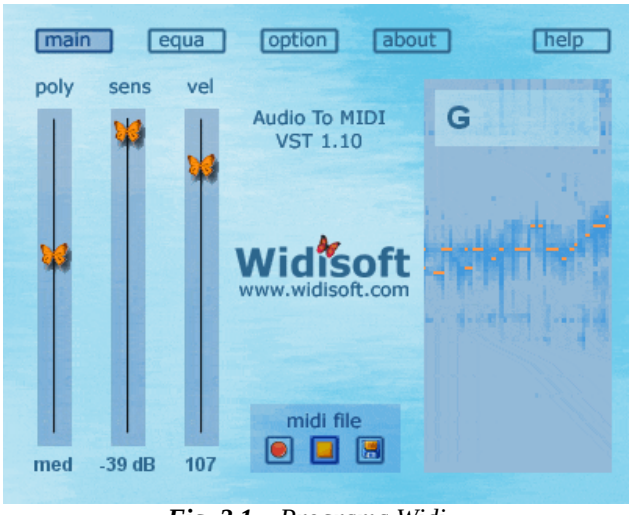


Fig. 2.1 *– Programa Widi*

Realizando pruebas para monofonías en tiempo real los resultados son muy buenos, apenas hay errores en la conversión.

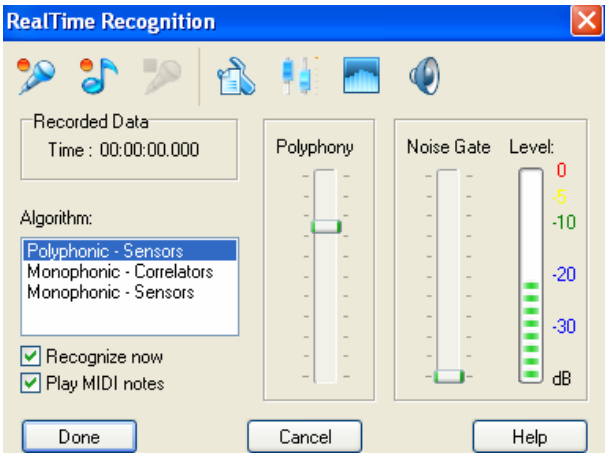


***Fig. 2.2** – Ventana diálogo de Widi*

En el aspecto de polifonías los resultados no son tan positivos, se da una tasa de error mucho superior. Ningún programa de trascripción actual consigue resultados sumamente aceptables para polifonías.

Todo y esto WIDI es posiblemente el sistema de trascripción polifónica más completo hasta la fecha. En la actualidad, ningún programa de trascripción es capaz de separar los instrumentos de una melodía. Pero este, a partir de una fuente de audio polifónica y con varios instrumentos genera un MIDI polifónico. Otra distinción es que WIDI acepta fuentes de sonido no armónicas (ruido, percusiones,...) que no se corresponden con notas, y las omite a la salida de la trascripción. Además, es sensible a la amplitud sonora de las notas.

Digital Ear

Éste software también da buenos resultados para monofonías y permite la trascripción en tiempo real de un forma eficaz y flexible. A continuación mostramos una captura de la ventana de trascripción en tiempo real. Se puede ver como en la pantalla se muestran parámetros como la frecuencia, en nombre de la nota, el tipo de fuente de audio que entra…

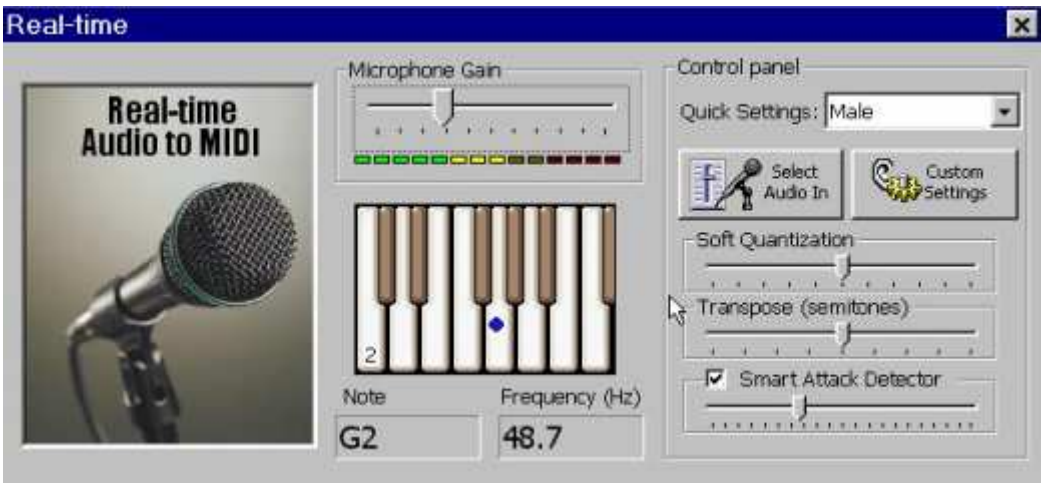


***Fig. 2.3** - Programa Digital Ear*

SOLO Explorer

SOLO Explorer es uno de los mejores sistemas comerciales de trascripción monofónica, y no necesita información previa del instrumento que se ha usado para generar la melodía. Funciona muy bien con voz y es bastante robusto ante el ruido, además de detectar desviaciones leves de la altura (vibratos).

Fig. 2.4 *– Solo Explorer*

2.4 Aplicaciones

Las aplicaciones que permite éste sistema son muy variadas y flexibles de cara al usuario.

Una de las más importantes es la trascripción de música interpretada. Para los músicos se traduce en la posibilidad de no necesitar escribir la música que componen manualmente, sino simplemente tocarla y automáticamente ésta se va generando en formato MIDI o en una partitura. Es semejante a la de tener un teclado MIDI o cualquier otro controlador pero con la diferencia de que lo que se maneja es el instrumento real, no un controlador adaptado.

Otra aplicación es la generación de información MIDI a partir de una interpretación para su posterior sintetización con instrumentos virtuales. Por lo tanto podemos interpretar con una guitarra eléctrica una melodía y después escuchar esta misma sonando con otro instrumento.

Un músico también tendrá la opción de tocar una melodía con su instrumento y a la vez la melodía sonar por varios canales con distintos timbres de instrumentos virtuales. Podemos montarnos una pequeña orquesta que nos acompañe interpretando lo mismo que estamos tocando o variarlo en función del MIDI.

Realmente hay miles de opciones, ya que el formato MIDI permite la variación de muchos parámetros: Transposición de notas MIDI (posibilidad de armonizar una melodía), cuantización de la interpretación, facilidad de retocar los eventos MIDI y sus propiedades…etc.

Las aplicaciones son muy amplias y variadas, dando flexibilidad al usuario para desarrollar su creatividad, su interpretación musical, experimentación con la música…etc.

3. Aspectos Teóricos

3.1. Caracterización de la señal de guitarra eléctrica

3.1.1 Introducción

Como se ha detallado anteriormente, el sistema que vamos a diseñar está orientado para realizar la conversión del audio que genera un instrumento real (guitarra eléctrica) a formato MIDI.

Uno de los puntos de partida más decisivos e importantes es conocer el tipo de señal de entrada que vamos a tener en nuestro sistema y estudiar sus características y parámetros.

Cada instrumento genera un sonido distinto, que podremos identificar por su timbre y por otras características. Por ejemplo, un violín suena de una forma muy diferente a como suena una trompeta o un piano. Desde un punto de vista físico podemos apreciar que suenan diferentes porque están construidos con materiales distintos, por la forma como se tocan, sus dimensiones,…etc. Todas estas distinciones las obviaremos ya que el análisis que realizaremos se centrará en el procesado digital de la señal.

La diferencia de sonido de los distintos instrumentos dentro del análisis de señal viene determinada, entre otros factores, por los armónicos que se producen al ser tocado. Los armónicos son los que generan el timbre característico de una fuente de sonido, en nuestro caso un instrumento. Por lo tanto cada uno genera una señal sonora distinta, que contiene una serie de armónicos característicos con una amplitud determinada, que darán el timbre correspondiente al instrumento.

Esta es la razón por la cual nuestro conversor de audio a MIDI está orientado exclusivamente a guitarras eléctricas. La parte de detección de frecuencia estará diseñada en función de las características de una señal de guitarra eléctrica, asimismo como el registro de las notas, el ancho de banda y los distintos parámetros a configurar.

3.1.2 Registro del instrumento

Para empezar a caracterizar nuestra señal de entrada lo primero que tendremos que ver es cual es el rango de notas o frecuencias que es capaz de generar nuestro instrumento.

Una guitarra eléctrica convencional tiene un total de 6 cuerdas, situadas encima del diapasón del mástil, el cual está divido por trastes en intervalos de un semitono. Observando esto podemos afirmar que la resolución tonal es de un semitono.

***Fig. 3.1** – Diapasón guitarra eléctrica*

Cada una de estas cuerdas, cuando se toca al aire (sin pulsar ningún traste) con una afinación estándar, nos da una nota que produce una determinada frecuencia:

	Cuerda 6	*Cuerda 5*	*Cuerda 4*	*Cuerda 3*	*Cuerda 2*	*Cuerda 1*
Notación latina	Mi2	La2	Re3	Sol3	Si3	Mi4
Notación inglesa	E2	A2	D3	G3	B3	E4
Frecuencia (Hz)	82.41	110	146.83	196	246.94	329.63

Fig. 3.2 *– Tabla notas-frecuencia de cada cuerda*

Viendo la tabla superior podemos ver que la nota más grave que podrá producir una guitarra eléctrica con afinación estándar es un Mi2, correspondiente con la frecuencia de 82.41 Hz. Este será el límite inferior que fijaremos en la detección de frecuencia, ya que no podrán sonar notas más graves.

A la hora de fijar el límite superior la situación varía ligeramente, ya que podemos utilizar los trastes del diapasón para tocar distintas notas más agudas. Las guitarras eléctricas actuales tienen un total de 24 trastes aproximadamente, variando ligeramente en función de la marca de guitarra. Como hemos dicho anteriormente, cada traste es un semitono, por lo que si tenemos un total de 24 trastes se traduce en que podremos tocar 24 semitonos más agudos a partir de la cuerda al aire. Dicho de una forma más simple, podemos recorrer 2 octavas (24 semitonos) a partir de la cuerda al aire.

Por lo tanto, si la cuerda más aguda que tenemos es un Mi4 con una frecuencia de 329.63 Hz, si calculamos 2 octavas a partir de ahí obtenemos un Mi6, con una frecuencia de 1318.52 Hz.

Nuestro rango de notas musicales y de frecuencias por lo tanto será el siguiente:

Fig. 3.3 *– Rango frecuencial guitarra eléctrica*

3.1.3 Armónicos

Éste es uno de los aspectos de más relevancia. Para realizar la conversión de la señal generada por el instrumento a formato MIDI tendremos que extraer una serie de parámetros, entre ellos la frecuencia fundamental, que irá asociada a la nota que está siendo tocada. En esta parte entran en juego los armónicos. Los armónicos que produce la señal sonora de la guitarra eléctrica determinan el timbre del sonido.

Durante el análisis tendremos que tener en cuenta todos los armónicos que puedan aparecer en las notas tocadas, ya que habrá que descartarlos y no confundirlos con la frecuencia fundamental.

Los armónicos que aparecen son múltiplos de la frecuencia fundamental y su amplitud puede variar, incluso a veces siendo superior a la frecuencia fundamental.

Por lo tanto tendrán una frecuencia determinada, siendo asimismo notas. Para hacernos una idea, en la siguiente tabla se toma como referencia la nota Do4 y se analizan los 16 primeros armónicos y su posición relativa en función de la nota fundamental, en éste caso Do4.

Nº de Armónico	Frecuencia	Nota	Intervalo
1º armónico	264 Hz	*Do*$_4$	tono fundamental (el primer *do* a la izquierda del piano)
2º armónico	528 Hz	*Do*$_5$	octava
3º armónico	792 Hz	*Sol*$_5$	quinta
4º armónico	1056 Hz	*Do*$_6$	octava
5º armónico	1320 Hz	*Mi*$_6$	tercera mayor
6º armónico	1584 Hz	*Sol*$_6$	quinta
7º armónico	1848 Hz	*Si*b$_6$	séptima menor (muy desafinada)
8º armónico	2112 Hz	*Do*$_7$	octava
9º armónico	2376 Hz	*Re*$_7$	segunda mayor
10º armónico	2640 Hz	*Mi*$_7$	tercera mayor
11º armónico	2904 Hz	*Fa#*$_7$	cuarta aumentada
12º armónico	3168 Hz	*Sol*$_7$	quinta justa
13º armónico	3432 Hz	*La*$_7$	sexta mayor (muy desafinada)
14º armónico	3696 Hz	*Si*b$_7$	séptima menor
15º armónico	3960 Hz	*Si*$_7$	séptima mayor
16º armónico	4224 Hz	*Do*$_8$	octava

Fig. 3.4 *– Tabla armónicos, frecuencia y nota*

En el caso particular de la señal de guitarra eléctrica, realizando un estudio de los armónicos que se producen al tocar una nota se puede observar que llegamos hasta el orden de aproximadamente unos 10-20 armónicos, variando en función de la nota que está siendo tocada y la cuerda. La figura inferior muestra el análisis espectral de la señal producida al tocar la nota La al aire (cuerda 5) en la guitarra.

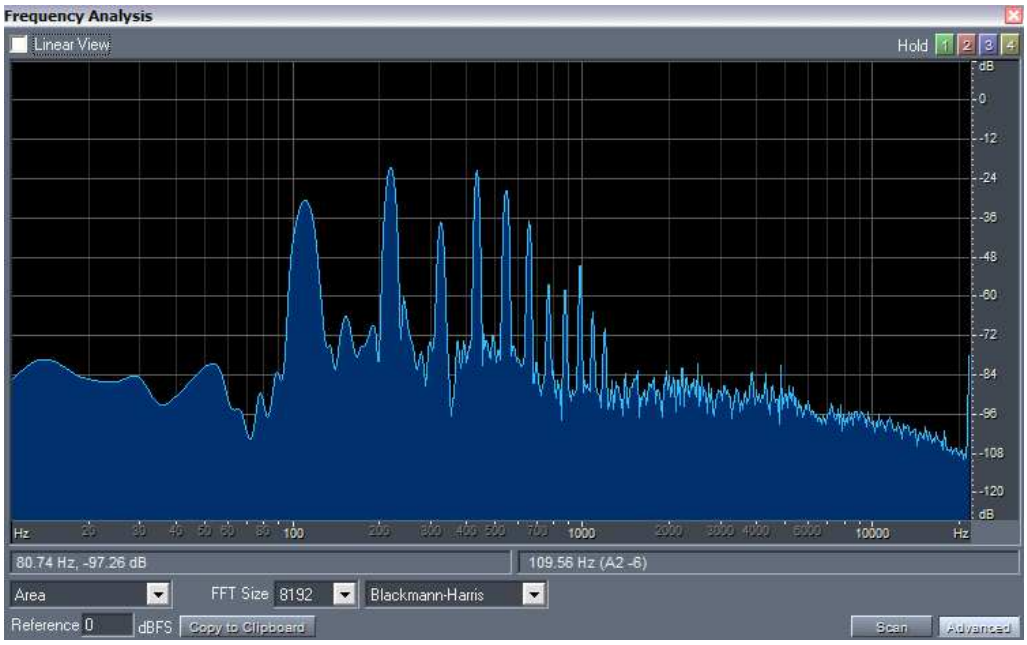


Fig. 3.5 *- Análisis espectral para la nota La al aire*

Podemos apreciar la frecuencia fundamental, que es de 109.56 Hz. Es de vital importancia destacar que la frecuencia fundamental puede que no sea el pico con más energía, por lo tanto una detección del máximo no nos puede asegurar su detección. En este caso podemos ver que el segundo y cuarto armónico tienen más energía. Necesitaremos emplear alguna técnica con la finalidad de eliminar esos armónicos que pueden ser confundidos con frecuencias fundamentales por su alta energía.

Como hemos comentado anteriormente, para notas más agudas aparecen más armónicos que para notas graves. La figura 3.6 muestra el mismo análisis espectral pero esta vez para la nota Si al aire (cuerda 2).

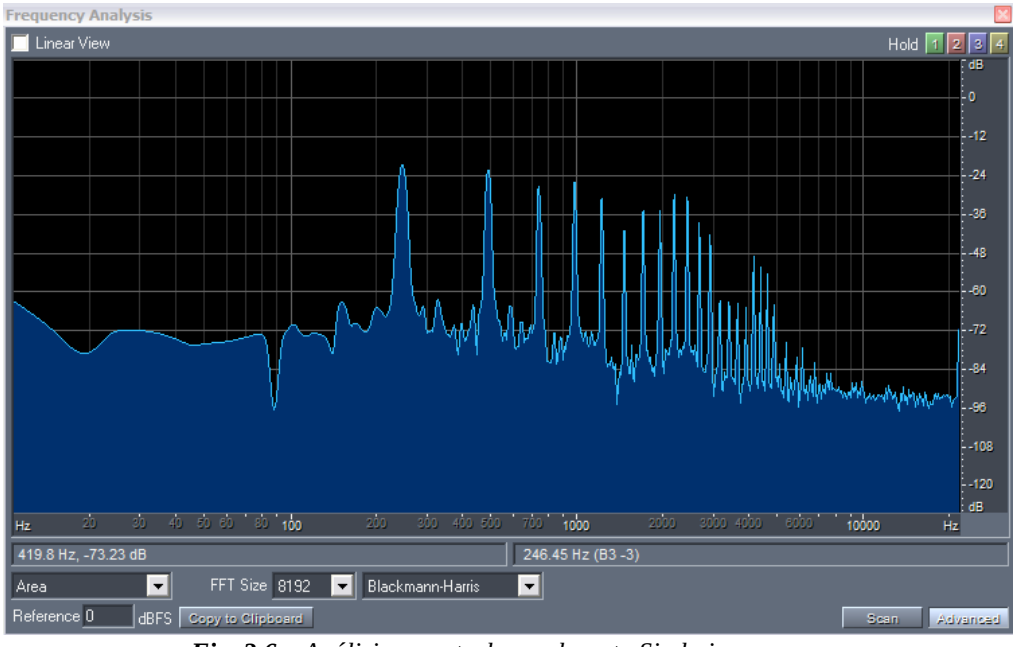


Fig. 3.6 *– Análisis espectral para la nota Si al aire*

En este caso se puede ver la aparición de más armónicos de alta frecuencia, aproximadamente del orden de unos 20. Ahora el máximo pico si que corresponde a la frecuencia fundamental, seguidos de los primeros armónicos que tiene una energía muy semejante.

Otro factor a remarcar es que los armónicos que se producen al tocar una nota van variando en función del tiempo, no se mantienen constantes.

3.1.4 Variación de la amplitud (ADSR)

Una señal sonora generada por un instrumento tiene una variación de amplitud a lo largo del tiempo. Para expresar esta variación se utiliza la función ADSR (Attack Decay Sustain Release). Esta variación de amplitud también se asocia al envolvente acústico, que será diferente para cada instrumento.

En la figura 3.7 podemos ver los distintos parámetros ADSR.

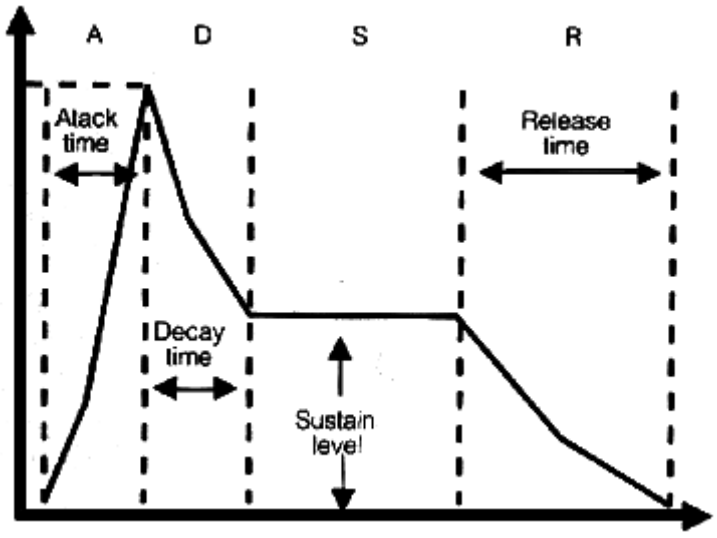


***Fig. 3.7** – Parámetros ADSR*

El **tiempo de ataque** (Attack Time) es el tiempo que tarda en escucharse el sonido después de haber sido tocado el instrumento. Desde que se toca hasta que se alcanza la amplitud máxima.

El **tiempo de decaimiento** (Decay Time) es el tiempo de atenuación de amplitud después de haber alcanzado el máximo hasta que se estabiliza.

El **sostenimiento (**sustain) va asociado al tiempo de duración máximo que suena una nota manteniendo su amplitud una vez ha sido pulsada. Este factor viene determinado en gran parte por la construcción y constitución de la guitarra en nuestro caso. También hay que mencionar que el sostenimiento también depende de las cuerdas. Las graves tienen más mientras que las agudas menos.

Por lo tanto es un parámetro importante, ya que nos determinará gran parte de la duración de la nota que nuestro sistema convertirá a MIDI.

En cambio la duración mínima de una nota tocada con la guitarra ya dependerá de la velocidad del intérprete o del músico. Deberemos establecer unos parámetros de configuración determinados y calcular cual es la resolución temporal de notas que puede detectar nuestro sistema. Esto lo veremos más adelante.

Por último la **relajación** (Release Time) es el tiempo que tarda el sonido en perder toda su amplitud después de pasar por la fase de sostenimiento.

La función ADSR de la guitarra eléctrica se caracteriza porque no tiene fase de sostenimiento. Esto es debido al método físico de tocar el instrumento. La guitarra es un instrumento de cuerda pulsada. Esto provoca que una vez ha sido tocado, las cuerdas vibren. Esta vibración no se mantiene constante, su fuerza va disminuyendo a lo largo de tiempo, provocando que no haya fase de sostenimiento.

En la figura 3.8 se muestra la forma de onda generada al tocar con la guitarra la nota La (cuerda al aire).

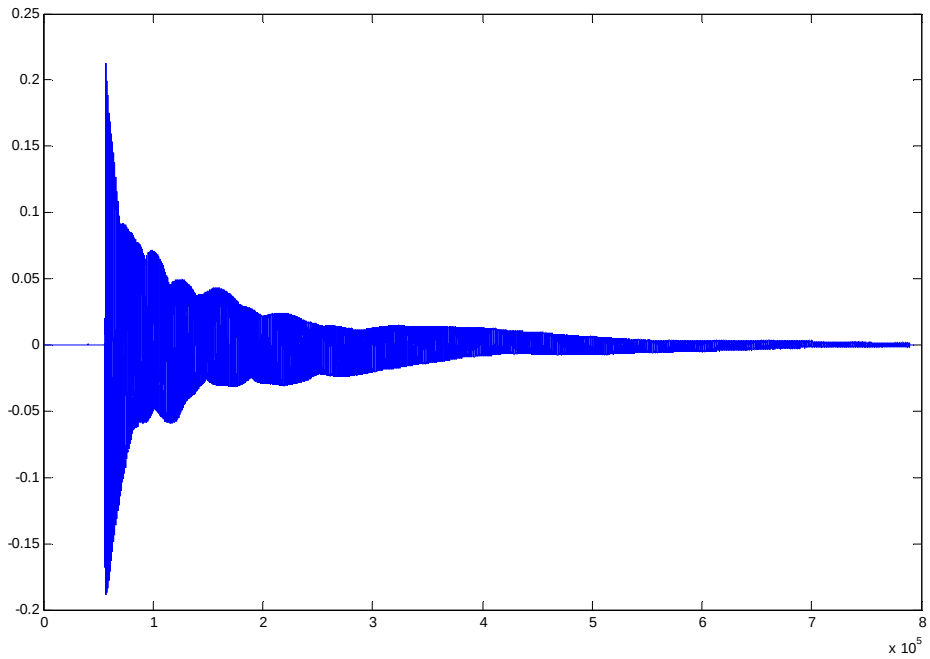


***Fig. 3.8** – Forma de onda de una nota tocada con la guitarra*

3.2 Configuración de los parámetros en el procesado de la señal

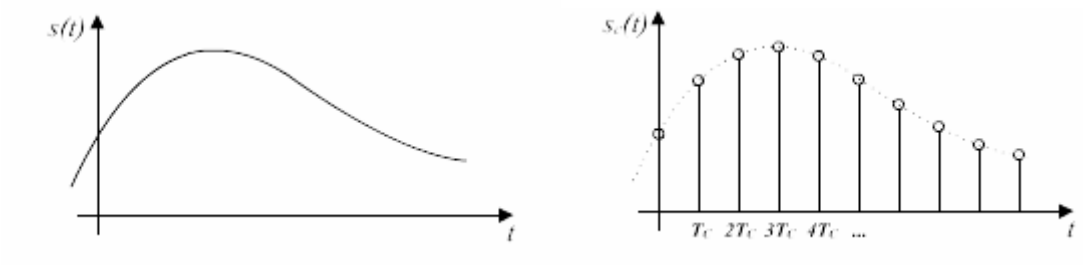
3.2.1 Introducción

A la hora de realizar nuestro procesado de señal necesitaremos configurar una serie de parámetros de forma adecuada para optimizar el funcionamiento de los algoritmos. Para configurarlos correctamente deberemos realizar un estudio previo de la señal de entrada que tendremos en nuestro sistema, ya que el procesado se basa en ella.

3.2.2 Frecuencia de muestreo

En nuestro caso la señal entrante es el audio generado por una guitarra eléctrica que está conectada a una entrada de la tarjeta de sonido del ordenador. La señal es analógica, por lo tanto esta ha de ser convertida a digital mediante un conversor A/D que ya incorpora la tarjeta de sonido. Aquí entra en juego el primer parámetro importante que tendremos que configurar, la frecuencia de muestreo.

La frecuencia de muestreo se define como el número de muestras por unidad de tiempo que se toman de una señal continua para producir una señal discreta. La unidad utilizada es el hercio.



Señal original *Señal muestreada*

***Fig 3.9** – Muestreo de una señal contínua*

A la hora de seleccionar una frecuencia de muestreo tendremos que tener en cuenta diversos factores.

Por una parte se ha de cumplir el **criterio de Nyquist** con la finalidad de realizar una representación adecuada de la forma de onda de la señal que queremos discretizar. El teorema dice que la frecuencia de muestreo debe ser igual o superior al doble de la frecuencia máxima, es decir:

$$Fm \geq 2 * F\max$$

Por encima de ese valor, cuanto mayor sea el número de muestras, más fiel será la conversión analógica digital (A/D), lo que se traduce en una mayor calidad de la señal resultante.

Para aplicaciones de audio se suelen utilizar frecuencias de muestreo de 44100 Hz o superiores. Con esos niveles se obtiene una calidad elevada y fiel a la señal original.

Deberemos configurar la frecuencia de muestreo más adecuada en función de los resultados que necesitamos obtener.

Para empezar tendremos que estudiar el rango frecuencial que produce la guitarra eléctrica para saber cual es la frecuencia mínima de muestreo que podemos usar para poder captar todas las frecuencias producidas por el instrumento.

Como vimos en un capítulo anterior, el rango frecuencial de la guitarra va de los 82.41 a los 1318.52 Hz.

Fig. 3.10 *– Rango frecuencial guitarra*

Podemos observar por lo tanto que la frecuencia máxima es de 1318.52 Hz. Según el criterio de Nyquist la frecuencia mínima de muestreo que podremos usar será de:

$$Fm\min = 2 * 1318.52 = 2637.04 \text{Hz}$$

Para utilizar una estándar tendríamos que seleccionar 8000 Hz, ya que es la más baja posible. De esta forma utilizaríamos el mínimo ancho de banda necesario para captar las todas frecuencias producidas por el instrumento.

Una vez conocida la Fm mínima estudiaremos los resultados obtenidos al configurar distintas Fm.

Por otra parte hay que valorar la repercusión que se produce sobre el bloque de muestras, ya que este va a ser la unidad principal de proceso del audio y depende directamente de la frecuencia de muestreo.

Para frecuencias de muestreo elevadas, como puede ser 44100 Hz (decimos que es una frecuencia elevada porque para este sistema no es necesario el uso de tanto ancho de banda para realizar el análisis frecuencial) podemos ver que las propiedades del bloque de muestras tomado varían ligeramente.

Una Fm alta provocará que el bloque o tamaño de la ventana tome un intervalo de tiempo de señal más pequeño. Es decir, con bloques de igual número de muestras, usar una frecuencia de muestreo más elevada provocara que esas muestras representen un intervalo de tiempo en la señal más corto. En la figura inferior se puede apreciar mejor.

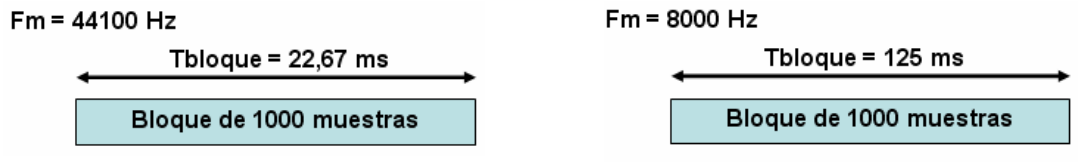


Fig. 3.11 *– Diferencia tiempo de bloque*

El caso de tener un periodo de tiempo de señal muy pequeño puede provocar que este no sea suficiente para realizar un adecuado análisis frecuencial.

El tamaño del bloque de muestras también será clave para decidir el tiempo de bloque. Podemos decir que una alta frecuencia de muestreo requerirá un tamaño de bloque mayor.

El valor de la frecuencia de muestreo también determinará la **carga de proceso** del sistema, ya que contra más muestras adquiramos, más muestras deberemos que procesar.

Otro factor determinante es la **resolución frecuencial y temporal** que obtendremos en función de la frecuencia de muestreo, el tamaño de la ventana o bloque y el número de puntos de la FFT. Esto lo veremos más adelante después de introducir los demás parámetros.

3.2.3 Tamaño del bloque de muestras

El tamaño del bloque hace referencia a la cantidad de muestras que se toman para realizar el procesado de la señal. Es decir, consiste en el tamaño de los bloques que iremos tomando a medida que entran las muestras para ir realizando sobre cada bloque el procesado y la extracción de los parámetros necesarios.

Periodo máximo de la señal

Un factor determinante para establecer el tamaño del bloque será ver cual es el periodo máximo de la señal que se puede dar, para que tengamos suficientes muestras y podamos captarlo adecuadamente. El periodo se define como el intervalo de tiempo necesario para completar un ciclo. Cabe aclarar que con la captación de un solo periodo no será suficiente. Para tener una buena detección necesitaremos aproximadamente unos 4 o 5 como mínimo. El periodo máximo de la señal se dará cuando suene la nota de frecuencia más baja posible, es decir, la más grave. En el caso de la guitarra eléctrica son 82.41 Hz, correspondientes a la nota Mi en la cuerda más grave. Sabiendo que el periodo de una señal es inversamente proporcional a su frecuencia:

$$Periodo = \frac{1}{Frecuencia}$$

$$Periodo\max \Rightarrow \frac{1}{82.41} = 0{,}01213 \rightarrow 12{,}13 milisegundos$$

Como hemos comentado anteriormente, necesitaremos como mínimo 4 o 5, por lo tanto el tiempo de bloque tendrá que cumplir la siguiente ecuación.

$$Tbloque \geq 4T\max$$

Conociendo la frecuencia de muestreo podemos averiguar el tamaño mínimo del bloque que ha de tener para cumplir la ecuación superior.

$$T\max = 12{,}13ms \qquad 4T\max = 48{,}52ms$$

$$Bloque\min = 4T\max * fm$$

Tomando como ejemplo una frecuencia de muestreo de Fm = 8000

$$Bloque\min = 8000 muestras / seg * 48{,}52ms = 388 muestras$$

Tiempo real

Para que el sistema funcione en tiempo real, el tiempo de proceso de un bloque ha de ser inferior al tiempo entre muestras consecutivas. Por lo tanto también deberemos buscar un tamaño adecuado para que el sistema funcione correctamente en tiempo real.

Por último el tamaño del bloque también determinará la resolución frecuencial y temporal en conjunto con la Fm y la resolución de la FFT, como veremos más adelante.

3.2.4 Tamaño de la FFT (NFFT)

Los resultados de la transformada de Fourier principalmente dependerán del número de muestras de la señal (tamaño del bloque) que hemos tomado para realizarla y de la frecuencia de muestreo. Podemos dar a la FFT un tamaño superior que el número de muestras del bloque pero no obtendremos una mejor resolución frecuencial, ya que la información útil que ésta nos proporciona solo nos la darán las muestras adquiridas.

Esto significa que el número de puntos de la FFT siempre tendrá que cumplir lo siguiente:

$$NFFT \geq Long.Bloque$$

Por conveniencia el valor de NFFT será potencia de 2, ya que de esta forma la transformada se realiza de forma más eficiente y con más velocidad.

3.2.5 Resolución frecuencial

Vistos los distintos parámetros anteriores ya podemos entrar a realizar un análisis para determinar la resolución frecuencial del sistema.

La resolución frecuencial depende directamente del valor de la frecuencia de muestreo, del tamaño del bloque de muestras y del número de puntos de la FFT. Por lo tanto deberemos buscar los valores adecuados para cada uno de estos parámetros para la optimización del sistema.

Como es lógico, primero deberemos saber la resolución frecuencial que requerimos. Para ello tendremos que ver que frecuencias es capaz de generar nuestro instrumento y con que resolución. Como vimos en un capítulo anterior, la guitarra eléctrica es capaz de realizar distintas notas con una separación de medio tono, es decir, un semitono. Esto se traduce en que la resolución frecuencial tendrá que ser de medio tono.

Pero también hay que tener en cuenta el comportamiento frecuencial de estos semitonos o notas a lo largo del espectro de frecuencias, ya que trabajan de una forma logarítmica.

Un tono equivale a dos semitonos y una octava esta formada por doce semitonos, los cuales se reparten siguiendo la siguiente regla:

$$F_{n+1}=F_n*2^{(1/12)}$$

Esta relación es obtenida sabiendo que al aumentar una octava una nota queda multiplicada por dos. Las notas entre ellas se dividen en doce partes de forma logarítmica, ya que la recepción auditiva del sonido en el cerebro humano sigue pautas logarítmicas.

Si queremos aumentar un semitono se multiplica esa nota por dos elevado a 1/12, si por el contrario queremos bajarla se dividirá por dos elevado a 1/12. El numerador del exponente se corresponde con el número de semitonos de distancia que queremos calcular.

Esto nos da a entender que la distancia en hercios entre dos semitonos de baja frecuencia no será la misma que la de dos de frecuencia alta, ya que el comportamiento es logarítmico. Para graves la distancia será mucho menor que para agudos. En el ejemplo se muestran distintos intervalos de semitono en frecuencia para distintos registros.

$A2 \Leftrightarrow A\#2 \qquad \Delta f = 116.54 - 110 \Rightarrow 6{,}54 Hz$

$A5 \Leftrightarrow A\#5 \qquad \Delta f = 932{,}33 - 880 \Rightarrow 52{,}33 Hz$

$A8 \Leftrightarrow A\#8 \qquad \Delta f = 7458{,}62 - 7040 \Rightarrow 418{,}62 Hz$

Visto esto se puede deducir que la distancia más pequeña en hercios que encontraremos en nuestro instrumento será la que se dará de la nota más grave posible (E2) a su semitono superior (F2), y será la que definirá la resolución frecuencial de nuestro sistema.

Por lo tanto la resolución frecuencial del sistema:

$E2 \Leftrightarrow F2 \qquad \boxed{\Delta f = 87{,}31 - 82{,}41 \Rightarrow 4{,}9 Hz}$

Calculada la resolución necesaria deberemos adaptar los distintos parámetros para el funcionamiento correcto del sistema.

La primera relación que se establece es que el número de muestras de señal que contenga cada bloque determinará la resolución que tendrá la FFT, necesaria para realizar la detección de frecuencia fundamental. Por lo tanto para empezar partiremos de que NFFT es igual al número de muestras del bloque.

A partir de este punto la resolución frecuencial vendrá determinada por la relación entre la frecuencia de muestreo y el número de puntos de la FFT.

La ecuación que relaciona estos 2 parámetros es la siguiente:

$$\Delta f = \frac{Fm}{NFFT}$$

Podemos apreciar que si fijamos un determinado tamaño de bloque (porque nos pueda interesar trabajar con ese tamaño) y utilizamos frecuencias de muestreo elevadas, perderemos resolución frecuencial. Al hacer la FFT tendremos que representar con un determinado número de puntos NFFT un ancho de banda alto. La separación entre cada punto de la FFT corresponderá a un salto en frecuencia elevado, provocando un pérdida de resolución.
En cambio, con el mismo tamaño de bloque fijado y una frecuencia de muestreo mucho menor podremos ver que ganamos resolución frecuencial, ya que con el mismo número de puntos NFFT que antes ahora tenemos que representar un ancho de banda mucho

menor. Obviamente la separación entre cada punto de la FFT supondrá un salto en frecuencia mucho menor que en caso anterior.

En las siguientes figuras se muestra un ejemplo sobre la explicación teórica anterior. Se toma la FFT de un bloque de 1024 muestras con el mismo número de puntos NFFT. La figura 3.12 muestra la resolución frecuencial para una Fm de 44100 Hz y la figura 3.13 para 8000 Hz.

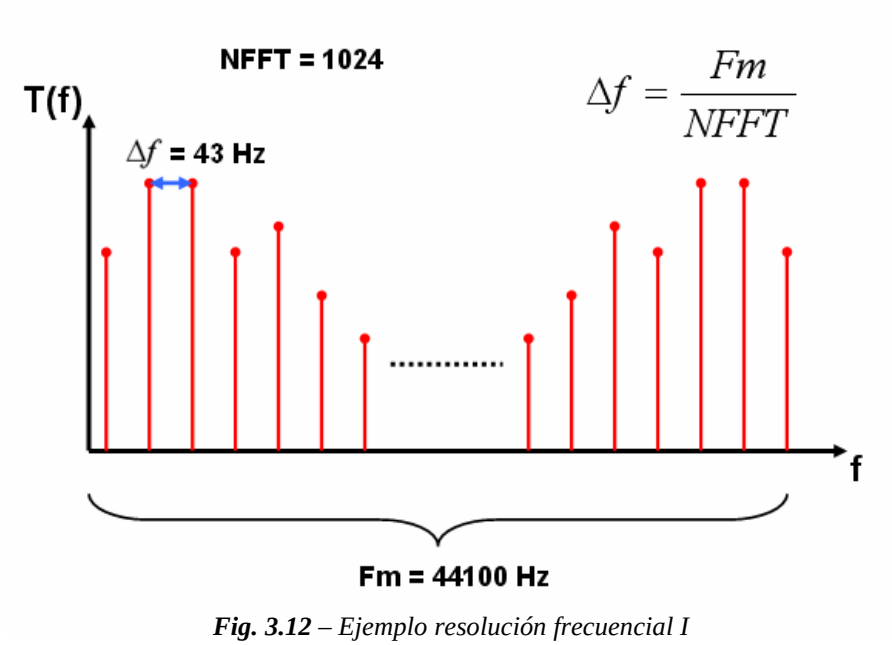


Fig. 3.12 *– Ejemplo resolución frecuencial I*

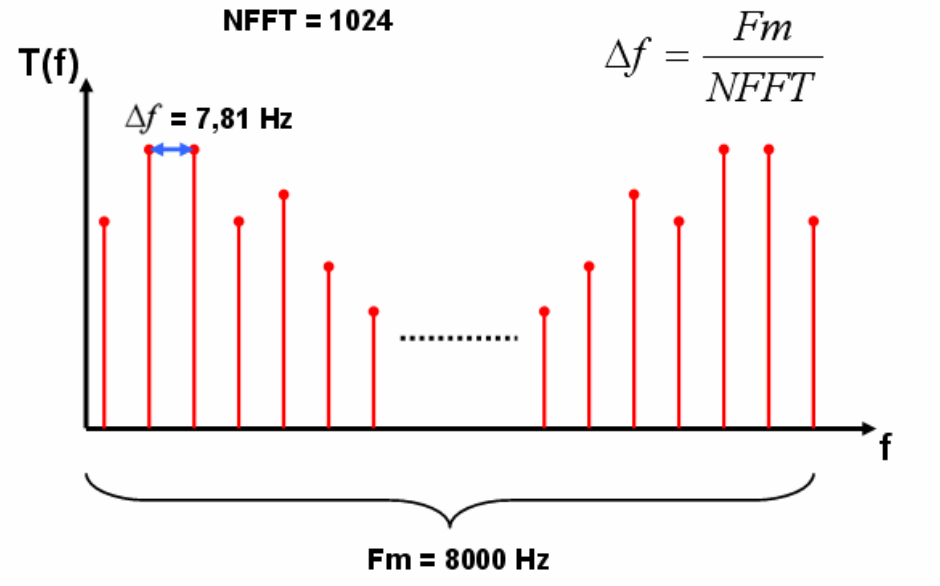


Fig. 3.13 *– Ejemplo resolución frecuencial II*

Podemos comprobar que en el segundo caso la resolución frecuencial es mucho mayor.

Dicho lo anterior podemos establecer un tamaño mínimo de bloque de muestras para una determinada frecuencia de muestreo conociendo la resolución que necesitamos.

$$LongBloque.Min = \frac{Fm}{\Delta f}$$

Por último, para elegir la frecuencia de muestreo y la longitud del bloque adecuados nos faltará valorar la resolución temporal que requerimos. Lo veremos a continuación.

3.2.6 Resolución temporal

La resolución temporal está directamente relacionada con la nota de duración temporal más corta que es capaz de detectar el sistema. Es decir, para nuestra aplicación, cual es la figura rítmica musical más rápida que es capaz de captar el conversor.

El tamaño del bloque y la frecuencia de muestreo son cruciales para determinar este factor, ya que establecen la unidad de análisis. Para calcular la resolución temporal que nos ofrece el bloque deberemos calcular el tiempo de duración de éste. Una vez calculado, obtendremos el valor temporal más pequeño que es capaz de captar. Si por ejemplo se tocan 2 notas rápidas consecutivas iguales con un tiempo inferior al del bloque lo que sucederá es que las 2 caerán dentro del mismo bloque y serán interpretadas como una sola al realizar el procesado posterior.

Para ajustar esta resolución temporal deberemos variar adecuadamente la frecuencia de muestreo y tamaño del bloque con la finalidad de optimizar la detección, siempre respetando que la resolución frecuencial se mantenga correcta como calculamos anteriormente.

Primero debemos establecer cual va a ser la resolución temporal de nuestro sistema. Para ello calcularemos cual es la figura rítmica de duración más corta que detectaremos con un determinado tempo establecido, que tendrá que ser el más alto que permitiremos configurar en el conversor.

El tempo se expresa en bpm (beats per minute) y hace referencia al número de golpes (beats) por minuto. A continuación se asocia una figura musical a ese golpe, de manera que ya sabemos la duración de una figura musical. La duración de las demás figuras musicales se obtiene por equivalencia con la conocida, tomándola como referencia.

A continuación se muestran las equivalencias entre las distintas figuras musicales que existen.

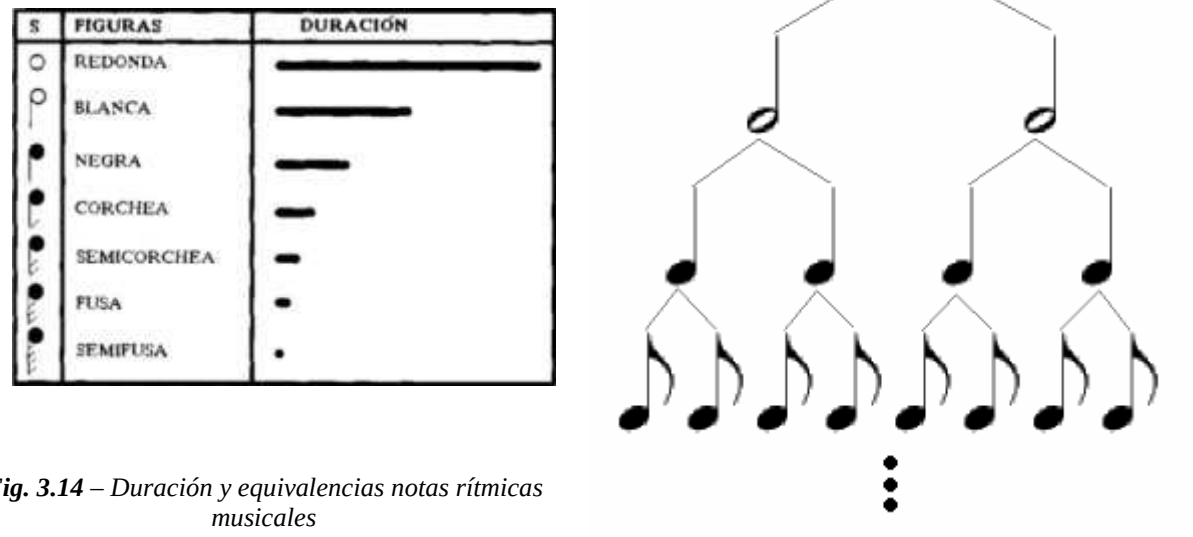


Fig. 3.14 *– Duración y equivalencias notas rítmicas musicales*

Podemos ver que las equivalencias son muy claras. A medida que descendemos un nivel en la tabla la nota vale la mitad que su inmediatamente superior. En la figura superior de la derecha vemos un esquema más claro. Solo llegamos hasta las corcheas, pero el esquema continuaría, con 2 semicorcheas en cada corchea, 2 fusas para cada semicorchea…etc.

Una vez conocido el bpm y su nota asociada podemos extraer el valor temporal de esta y después mediante equivalencias de las demás.

$$Tfigura = \frac{60 segundos}{1\min} * \frac{1}{bpm}$$

Conocido el tiempo de duración de la figura musical más corta que queremos tendremos que adaptar los distintos parámetros con la finalidad de que el tiempo de duración del bloque sea igual o más pequeño que esa figura musical.

$$Tbloque \leq Tfigura \min$$

No hay que olvidar que tenemos una restricción marcada por la resolución frecuencial, calculada anteriormente. A partir de ese valor podemos conocer cual es el número de muestras mínimo por bloque necesarios para que cumpla la restricción frecuencial, que también nos dará el valor temporal más pequeño del bloque, por lo tanto el de mayor resolución temporal.

En conclusión, marcada una resolución frecuencial determinada solo podremos obtener una resolución temporal máxima única, ya que la resolución frecuencial es inversamente proporcional a la resolución temporal. Podemos verlo en la siguiente demostración: (suponemos que el tamaño NFFT es el mismo que el del bloque de muestras)

$$\left.\begin{aligned} \Delta f = \frac{Fm}{Bloque} \Rightarrow Bloque = \frac{Fm}{\Delta f} \\ \Delta t = \frac{Bloque}{Fm} \Rightarrow Bloque = \Delta t * Fm \end{aligned}\right\} \frac{Fm}{\Delta f} = \Delta t * Fm \Rightarrow \frac{1}{\Delta f} = \Delta t$$

Realizamos el cálculo de la resolución temporal máxima que podemos obtener:

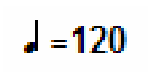
$$\Delta t = \frac{1}{\Delta f} \Rightarrow \Delta t = 0{,}2041 segundos$$

Una vez explicado el concepto de resolución temporal tendremos que determinar su valor.

El tempo máximo que vamos a permitir configurar en el sistema será de 120 bpm asociado a una negra. Es un tempo bastante corriente en la música moderna.

A continuación estableceremos la corchea como la figura rítmica más rápida que vamos a permitir detectar.

Una vez hecho esto tendremos que calcular cual es el tamaño mínimo del bloque que nos podemos permitir utilizar.



$$Tfigura = 60 * \frac{1}{120} = 0{,}5 segundos / negra$$

Figura							
Valor	2 seg	1 seg	500ms	250ms	125ms	62,5ms	31,25ms

***Fig. 3.15** – Tabla duración notas musicales configuradas*

Vista la tabla superior podemos ver que la duración mínima del bloque tendrá que ser de 250ms, correspondiente a una corchea a un bpm de 120 la negra. Esa será la resolución temporal necesaria para el sistema.

$$LongBloque \leq Fm * 250ms$$

3.2.7 Determinación de los parámetros

Después de ver y explicar los distintos aspectos a tener en cuenta a la hora de configurar los parámetros deberemos realizar varias pruebas y ver los resultados que se obtienen, verificando siempre que se cumplan los requisitos establecidos anteriormente.

Hemos establecido las siguientes restricciones:

- ✓ **Resolución frecuencial**
  - Capaz de distinguir intervalos de un semitono
  - $\Delta f \leq 4{,}9 Hz$
- ✓ **Resolución temporal**
  - Capaz de detectar corcheas a 120 bpm la negra.
  - $Tbloque \leq 250 ms$

A partir de esas condiciones fijaremos un valor determinado para la frecuencia de muestreo, la longitud del bloque y el número de puntos de la FFT (NFFT).

Frecuencia de muestreo

Configuraremos una frecuencia de muestreo de 44.100 Hz. La razón por la cual utilizamos una frecuencia de muestreo más elevada de la necesaria es por dar la opción de monitorizar el audio entrante con una calidad buena, para que de esta forma podamos escuchar a la vez lo que estamos tocando con la guitarra y el MIDI que se va generando a partir de ello.

Tamaño del bloque

Para fijar el tamaño del bloque tendremos que tener en cuenta la resolución temporal. Primero calcularemos el tamaño máximo que puede tener:

$$Bloque\max = Fm * \Delta t \Rightarrow 44100 * 0.250 = 11025 muestras$$

Tomaremos como tamaño del bloque el valor de 9216 muestras, ya que es preferible tomar un valor menor al calculado anteriormente para obtener mejores resultados.

De esta forma cumpliremos con la restricción de resolución temporal.

$$N = 9216 muestras$$

También será necesario comprobar que con éste tamaño la resolución frecuencial ofrecida cumple las expectativas fijadas. Lo comprobaremos en el siguiente punto.

Tamaño de la FFT (NFFT)

El valor de NFFT y el tamaño del bloque serán decisivos para fijar la resolución frecuencial del sistema.

$$\Delta f = \frac{Fm}{NFFT}$$

Conocido el valor de la resolución frecuencial y de la frecuencia de muestreo podemos calcular cuantos puntos NFFT necesitamos como mínimo:

$$NFFT\min = \frac{Fm}{\Delta f} \Rightarrow NFFT\min = 9000$$

Asignaremos un valor de NFFT que sea potencia de 2. Por lo tanto:

$$NFFT = 16384 puntos$$

En éste caso estamos dando más puntos a la FFT que muestras contiene el bloque a transformar. La verdadera resolución frecuencial solo vendrá determinada por el número de muestras del bloque.

Esto se traduce en que la resolución frecuencial que realmente tiene nuestro sistema es la siguiente:

$$\Delta f = \frac{Fm}{L.Bloque} \Rightarrow 4{,}78Hz$$

Podemos ver que la restricción frecuencial se cumple correctamente.

3.3 Detección de pitch

3.3.1 Planteamiento

El principal problema en la detección de frecuencia fundamental de notas utilizando métodos espectrales es, como vimos en uno de los capítulos anteriores, la aparición de armónicos. Estos pueden ser confundidos como fundamentales a causa de su elevada energía, que incluso pueden superar a la misma fundamental.

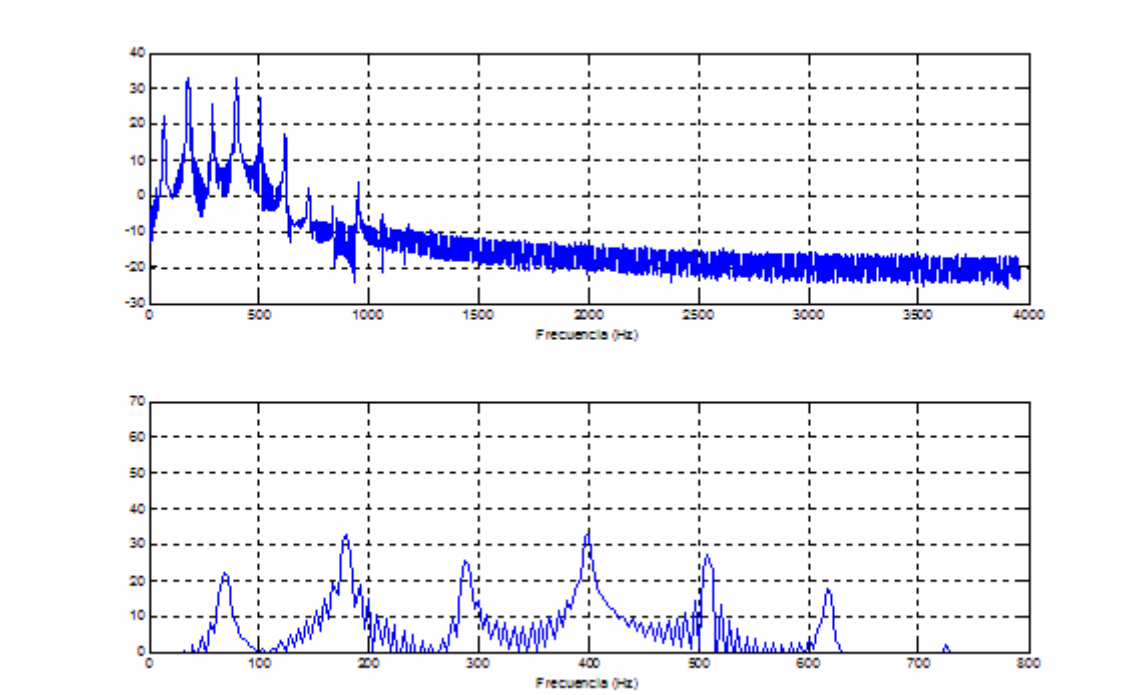


Fig. 3.16 *– Fundamental y armónicos en espectro frecuencial*

Dada esta situación es necesario el empleo de alguna técnica capaz de minimizar la energía de estos armónicos, de forma que el espectro frecuencial que vayamos a tratar tan sólo contenga los picos debidos a las frecuencias fundamentales de las distintas notas tocadas, que serán las que deberemos detectar.

3.3.2 Algoritmo HPS

Para realizar la detección de frecuencia fundamental se ha utilizado el algoritmo HPS (Harmonic Product Spectrum). Esta técnica nos permite minimizar la influencia de los armónicos y detectar de forma más eficaz la fundamental.

Basa su funcionamiento en aprovechar la propiedad de que los armónicos son múltiplos de su fundamental. Esto significa que se encuentran a F*n, donde “n” es un número entero.

Si se comprime el espectro un número entero de veces y se compara con el original podremos ver que picos corresponden a armónicos y cuales a fundamentales.

Para entenderlo mejor, si diezmamos la señal espectral por un factor de 2 y la multiplicamos con el espectro inicial, el valor de la fundamental se sumará con la de su primer armónico, reforzando así su amplitud. Todo nivel de señal que no sea armónico o fundamental quedará multiplicado por un valor pequeño, dejándola a un nivel imperceptible comparado con la fundamental.

Si volvemos a diezmar, esta vez por un factor 3, los 3º armónicos se multiplicarán a la fundamental, aumentando otra vez su nivel.

En la figura inferior se muestra un esquema más práctico sobre ésta técnica:

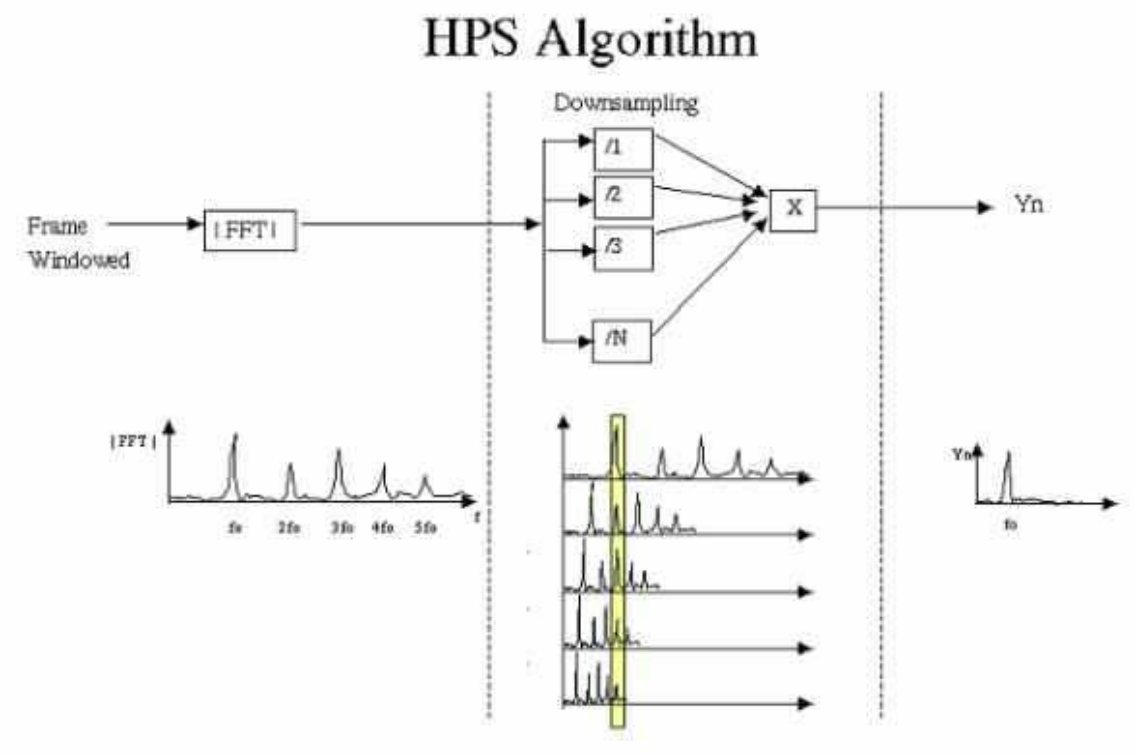


Fig. 3.17 *– Funcionamiento Técnica HPS*

Podemos comprobar que el resultado obtenido es bastante satisfactorio. Para realizar pruebas y ver los resultados se ha utilizado el entorno Matlab. A continuación se muestra la figura 3.18, que contiene la trama temporal a analizar, su espectro original y su respectivo HPS. Se observa que la técnica da buenos resultados.

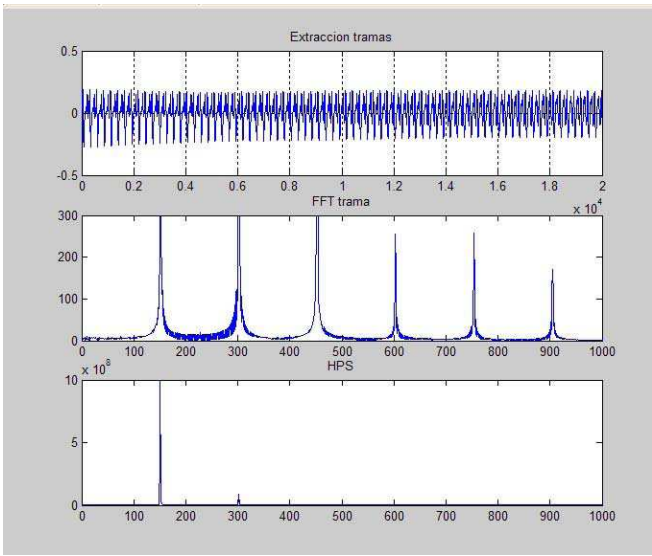


Fig. 3.18 *– Resultados HPS*

3.4 Sistema MIDI

3.4.1 Introducción

MIDI es el acrónimo de Musical Instruments Digital Interface (Interfaz digital para instrumentos musicales). Es un protocolo digital de comunicaciones, surgido del entendimiento entre fabricantes de equipos musicales electrónicos que permitió que estos instrumentos se comunicaran entre ellos y que, por extensión, se comunicaran con los ordenadores. En el MIDI se dan cita conceptos relacionados con la música, la informática y las comunicaciones.

La diferencia entre la información de audio y los datos MIDI se puede comparar con la diferencia que existe entre un CD de música y su correspondiente partitura. La diferencia es que el MIDI trata de partituras que han de ser entendidas por máquinas, y no por seres humanos.

3.4.2 El MIDI en el mundo de la música

Tal como se ideó inicialmente, el MIDI permitía la comunicación entre instrumentos, de forma que desde un único teclado controlador se podían disparar sonidos en otras unidades. Pero la gran revolución llegó con la incorporación de los ordenadores personales.

Por su naturaleza, los ordenadores son especialmente indicados para grabar, almacenar, manipular y reproducir cualquier otro tipo de dato digital, lo que incluye a los datos MIDI. Los músicos profesionales se percataron de todo esto rápidamente y, a pesar de que los ordenadores personales acababan de aparecer y eran una sombra de lo que son hoy en día, el ordenador MIDI no tardó en convertirse en una herramienta imprescindible en muchas áreas de la producción musical y audiovisual.

Por ello, si a la potencia de los actuales ordenadores le añadimos el espectacular abaratamiento sufrido por el hardware musical (tarjetas de sonido, teclados, etc.), no nos será difícil imaginar que actualmente uno puede disponer en su casa de prestaciones que muchos músicos hubiesen envidiado hace tan sólo una década.

Todo y eso, lógicamente el MIDI no puede suplir los conocimientos y las habilidades de un músico a lo largo de todos sus años de estudio y práctica, pero si ha tenido un gran peso en la evolución de los estilos musicales de esta última década, permitiendo nuevos enfoques de creación musical en los cuales las ideas y la imaginación juegan un papel incluso más importante que la propia destreza musical con un instrumento.

3.4.3 Especificación MIDI

Los mensajes se transmiten de forma binaria y en serie, es decir, mediante pulsos (bits) sucesivos. La transmisión se produce de forma asíncrona, es decir, siempre que un dispositivo decida enviar un mensaje. Esta asincronía obliga a que cada byte de mensaje vaya rodeado de un bit de comienzo y un bit de final. Estas transmisiones se realizan a una velocidad de 31.250 bits por segundo, por lo que la velocidad máxima de transmisión es de 3125 bytes/sec.

Puertos

Puerto emisor: **MIDI OUT**. Se encarga de convertir los datos digitales generados por el dispositivo en series de voltajes eléctricos.

Puerto receptor: **MIDI IN**. Realiza el proceso inverso al puerto emisor.

Puede existir un tercer puerto, denominado **MIDI THRU**, que simplemente reenvía la información llegada al MIDI IN del interfaz.

Todos ellos utilizan conectores DIN hembras de cinco pines (de los cuales sólo se utilizan en realidad tres).

Principales dispositivos MIDI

Los dispositivos MIDI se clasifican en tres categorías:

- ✓ **Controladores**: generan los mensajes MIDI. El controlador más típico existente tiene forma de teclado de piano, aunque pueden adoptar otras formas.

- ✓ **Unidades generadoras de sonido**: conocidas como módulos de sonidos. Reciben los mensajes MIDI y los transforman en señales sonoras.

- ✓ **Secuenciadores**: aparatos destinados a grabar, reproducir o editar mensajes MIDI.

3.4.4 Mensajes MIDI

Existen diferentes tipos de mensajes MIDI y cada uno de ellos tiene un tamaño fijo, normalmente de dos o tres bytes.

A continuación citaremos los principales y más importantes:

- ✓ **Nota**: Este mensaje indica la altura tonal y la intensidad de la nota. La intensidad viene determinada por la fuerza o velocidad con la que el músico pulsa la tecla. El mensaje recibe el nombre de Note On. No se incluye la duración de la nota por una razón muy simple: cuando el músico pulsa una tecla, el sintetizador no sabe cuánto va a durar esa nota. Por lo tanto cuando el músico deja de pulsar la tecla el sintetizador vuelve a enviar el mismo mensaje, pero esta vez con intensidad nula.

- ✓ **Cambio de programa** (Program Change): indica que se debe cambiar de instrumento. Contiene un único parámetro, el número del nuevo instrumento deseado.

- ✓ **Variación de la altura** (Pitch Bend): En la mayoría de instrumentos acústicos la afinación de una nota varía ligeramente a lo largo de su duración. Esto no debe considerarse una imperfección, ya que es uno de los matices que enriquecen el sonido. Sin embargo en los instrumentos electrónicos la afinación es estable. Para cubrir esta necesidad los teclados electrónicos disponen de una pequeña rueda giratoria que permite una desafinación controlada.

- ✓ **Cambio de control (Control Change)**: Este mensaje se compone de dos parámetros, siendo el primero el tipo de control o efecto elegido, y el segundo el valor o intensidad de este control. Entre los controles de uso más frecuente podemos citar el volumen (control 7), la posición panorámica (control 10) o la reverberación (control 91). Existen algunos más y otros muchos por definir, lo que lo convierte en uno de los mensajes más versátiles.

3.4.5 Estructura de un mensaje

Tal y como acabamos de ver en el apartado anterior, los mensajes MIDI se componen de dos o tres bytes. Estos bytes se dividen en dos categorías: bytes de status y bytes de datos. Su distinción se realiza a partir del valor de su bit más significativo. En los de status vale 1, mientras que en los bytes de datos está siempre a 0.

Los siete bits libres restantes son los que condicionan que el número de posibles programas en el mensaje de Program Change sea 128 (y no 256), y lo mismo es aplicable a cualquier otro mensaje. Los datos MIDI están siempre comprendidos entre los valores decimales 0 y 127 (binarios 00000000 y 01111111).

Todo mensaje MIDI se compone de un primer byte de status (que determina el tipo del mensaje) y uno o dos bytes restantes de datos (dependiendo del tipo de mensaje). En el byte de status, tan solo tres de los siete bits disponibles (no olvidemos que el más significativo está siempre a 1) son los que determinan el tipo de mensaje. Los cuatro restantes indican el canal al que el mensaje va dirigido, lo que explica porque son dieciséis (2^4) los canales MIDI posibles. En la figura inferior se puede apreciar la estructura binaria de un mensaje.

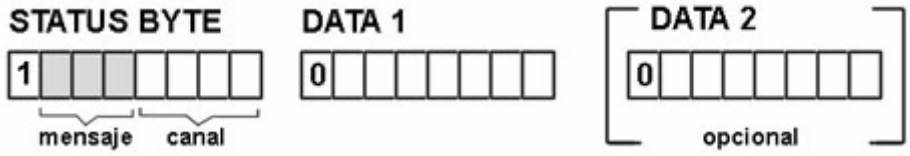


***Fig. 3.19** – Estructura mensaje MIDI*

De lo anterior se deduce que pueden existir ocho (2^3) tipos de mensaje diferentes, que se detallan en la siguiente tabla:

Nombre	Binario	Hex.	Data1	Data2
Note Off	1000 nnnn	8 N	altura	velocidad
Note On	1001 nnnn	9 N	altura	velocidad
Poly. Aftertouch	1010 nnnn	A N	altura	presión
Control Change	1011 nnnn	B N	tipo de control	intensidad
Chan. Aftertouch	1100 nnnn	C N	presión	
Pitch Bend	1101 nnnn	D N	MSByte	LSByte
Program Change	1110 nnnn	E N	programa	
System Message	1111 xxxx	F X		

- ✓ *nnnn son los cuatro bits que determinan el canal al que se aplica el mensaje. N corresponde al carácter hexadecimal de este canal (0-F).*
- ✓ *Todos los bytes de datos tienen una resolución de siete bits, con valores decimales comprendidos entre 0 y 127.*
- ✓ *Cuando en la tabla el segundo byte de datos está en blanco significa que el mensaje utiliza un único byte de datos.*
- ✓ *En el mensaje Pitch Bend, los dos bytes de datos se combinan para formar un único valor con catorce bits de resolución, comprendido entre -8192 y +8191.*

***Fig. 3.20** – Tabla de mensajes MIDI*

4. Tecnologías Utilizadas

Para llevar a cabo el desarrollo del presente proyecto se han utilizado diversas tecnologías relacionadas con el mundo de la ingeniería. A continuación realizaremos una breve descripción de cuales son, sus características principales y cual ha sido su utilidad durante el desarrollo del proyecto.

4.1 Visual C++

La implementación del proyecto la realizaremos mediante Visual C++ y distintas librerías. Visual C++ es una herramienta y un lenguaje de programación, producto de los lenguajes manejados por Microsoft C, C++ y C++/CLI. Está especialmente diseñado para el desarrollo y depuración de código escrito para las API's de Microsoft Windows, DirectX y la tecnología Microsoft .NET Framework, y no está especializado en ningún tipo de aplicación.

***Fig. 4.1** – Microsoft Visual C++*

Es un lenguaje versátil, potente y general. Su éxito en los programadores profesionales le ha llevado a ocupar el primer puesto como herramienta de desarrollo de aplicaciones. Goza de una gran riqueza en cuanto a operadores y expresiones, flexibilidad, concisión y eficiencia.

También hemos tomado esta herramienta debido a que el sistema ha de funcionar en tiempo real, lo que requiere un lenguaje potente y capaz de realizar con éxito todas las operaciones programadas.

En el capítulo de programas y funciones veremos como hemos desarrollado el proyecto utilizando esta herramienta.

4.2 PortAudio

La librería que utilizaremos para tratar los datos de entrada/salida de audio en tiempo real será PortAudio.

PortAudio es una API de audio para tiempo real, la cual trabaja en C a bajo nivel. Pertenece al proyecto PortMusic y una de las características a remarcar es que es totalmente gratuito, de código abierto y multiplataforma.

A continuación mostramos las características principales más destacables de PortAudio:

- ✓ API de audio para tiempo real
- ✓ Mecanismo de interrupciones periódicas para el procesado de audio
- ✓ Código abierto
- ✓ Trabaja en C a bajo nivel
- ✓ Multiplataforma
    - ASIO para Windows y Macintosh
    - BeOS
    - Macintosh Sound Manager para OS 8,9 y Carbon
    - Macintosh Core Audio para OS X
    - Silicon Graphics AL
    - Implementación de OSS bajo UNIX
    - Windows Direct Sound (Direct X)
    - Windows MME

Una de las ventajas de PortAudio es su control eficiente y directo del sonido, sin necesidad de un control hardware complicado.

La versión que utilizaremos será la "v18_1". Para empezar a trabajar con ésta librería primero deberemos realizar unos pasos previos para configurarla.

Añadiremos al proyecto los ficheros "pa_win_wmme.c", "pa_host.h" y "pa_lib.c", necesarios para la compilación. A continuación colocaremos en los directorios de "include files" el archivo "portaudio.h". Para finalizar linkaremos el proyecto con la librería "winmm.lib".

Una vez realizada la configuración previa ya podremos empezar a utilizar PortAudio.

Más adelante veremos más a fondo las utilidades que ofrece esta librería y su funcionamiento a la hora de programar nuestra aplicación.

4.3 PortMidi

PortMidi (Portable Real-Time MIDI library) es una librería multiplataforma destinada al proceso en tiempo real de datos MIDI. También pertenece al proyecto PortMusic como PortAudio, y obviamente es gratuita y de código abierto. Después de realizar varias pruebas en distintas versiones finalmente nos decantamos por utilizar la versión "portmidi20oct06". Por lo tanto trabajaremos con ella en la parte MIDI de nuestro proyecto.

Para empezar a utilizar PortMidi primero deberemos realizar unos pasos previos para conseguir compilar correctamente la aplicación en nuestro entorno Visual C++.

Una vez descargado el paquete PortMidi se han de compilar los proyectos que contiene con la finalidad de obtener las librerías necesarias (portmidi.lib, porttime.lib, pm_dll.lib y pm_dll.dll) que como es lógico linkaremos a nuestro proyecto de Visual C++. También nos será necesario linkar la librería "winmm.lib", que más adelante veremos su utilidad.

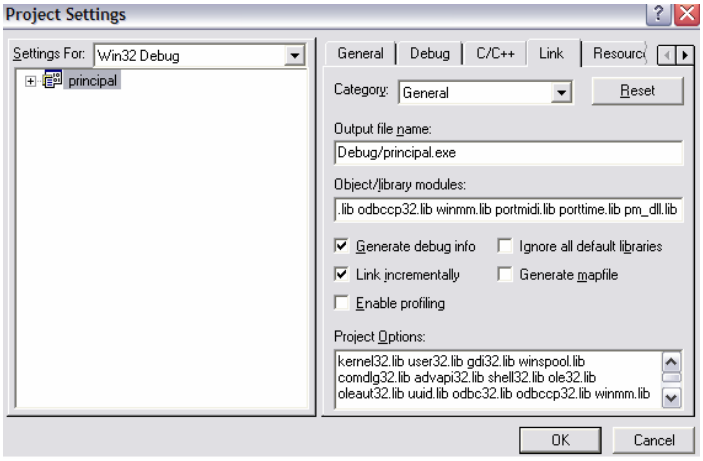


***Fig. 4.2** – Linkaje de librerías al proyecto*

También será necesario colocar en la carpeta "Debug" de nuestro programa la librería dinámica "pm_dll.dll".

A continuación situaremos en los directorios de los "include files" los archivos "porttime.h" y "portmidi.h" para el correcto funcionamiento de la librería.

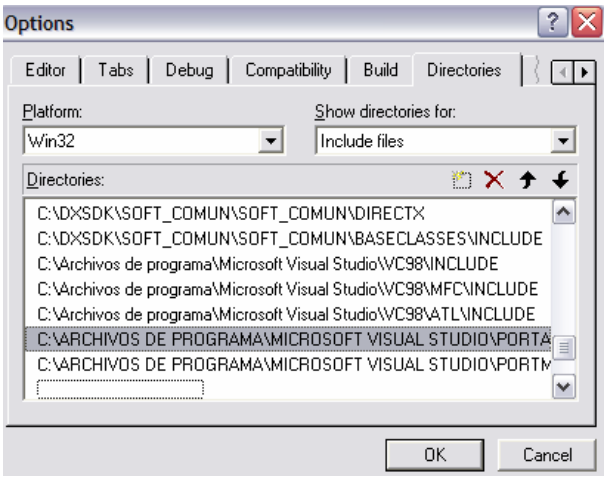


***Fig. 4.3** – Configuración entorno Visual C++I*

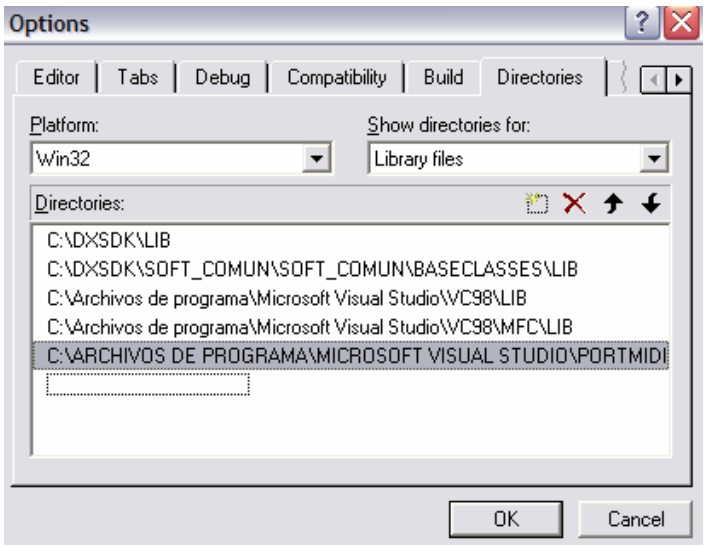


***Fig. 4.4** – Configuración entorno Visual C++ II*

Por último también situaremos en los directorios de las librerías "Library files" las librerías mencionadas anteriormente.

Una vez realizados los pasos anteriores ya podremos empezar a programar con Visual C++ usando la librería PortMidi

4.4 Matlab

Para la realización de pruebas y obtención de resultados en el procesado de señal se ha utilizado como herramienta el software Maltab, de "The MathWorks", debido a que es mucho más cómodo y sencillo a la hora de trabajar.

Integra cálculos, visualización, y programación de un ambiente sencillo, donde los problemas y las soluciones se expresan en una notación matemática familiar y sencilla.

Es un programa de cálculo numérico, orientado a matrices y vectores. Por tanto desde el principio hay que pensar que todo lo que se pretenda hacer con el será mucho más rápido y efectivo si se piensa en términos de matrices y vectores.

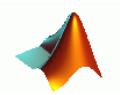


Fig. 4.5 *– Icono Matlab*

Se pueden ampliar sus capacidades con "*Toolboxes"*, algunas de ellas destinadas al procesado digital de señal, adquisición de datos, inteligencia artificial,...etc. En mi caso utilizaré un Toolbox para analizar información MIDI.

También hay que mencionar que el desarrollo de la interfaz gráfica la realizaré mediante la herramienta de Matlab "GUID", como veremos más adelante.

5. Desarrollo programas y funciones

5.1 Estructura

Antes de iniciar la implementación del sistema comentaremos la estructura general que seguiremos en la programación mediante Visual C++.

La primera tarea que realizaremos será procesar el audio en tiempo real con la finalidad de extraer los parámetros necesarios de cada bloque. Para ello emplearemos la librería PortAudio y una serie de funciones para la detección de frecuencia y su correspondiente nota.

Será necesario crear nuevas funciones para adaptar algunos aspectos del funcionamiento en tiempo real.

Una vez extraídos los parámetros de cada bloque crearemos un algoritmo que gestione la información MIDI que nos va llegado. Por último emplearemos la librería PortMidi para generar los correspondientes mensajes MIDI y enviarlos por un puerto de salida MIDI OUT disponible.

5.2 Procesado audio en tiempo real mediante PortAudio

Anteriormente vimos las características principales de la librería PortAudio. A continuación analizaremos su mecanismo de funcionamiento y como utilizar la librería para asumir los objetivos de éste bloque.

El siguiente esquema muestra los pasos que seguiremos en la implementación con PortAudio.

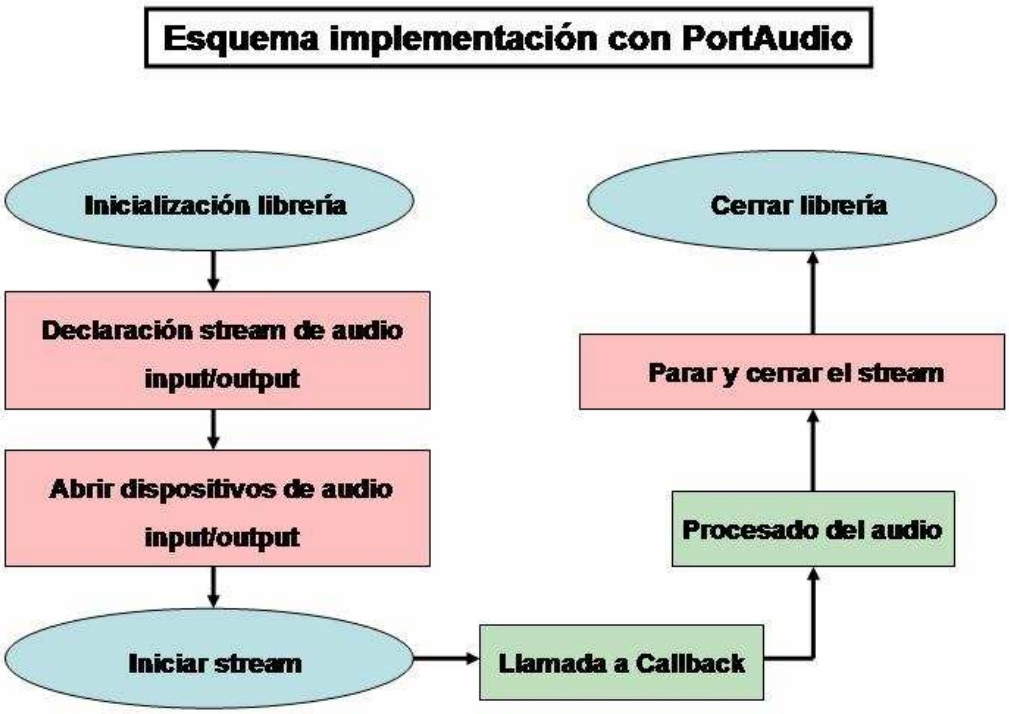


Fig. 5.1 *– Esquema implementación PortAudio*

Una vez visto el esquema analizaremos las funciones más importantes de la librería necesarias y como realizar los pasos adecuadamente. La principal fuente de documentación que utilizaremos será "portaudio.h".

Inicialización de la librería

En "portaudio.h" podemos encontrar la función ***Pa_Initialize( )*** que es la encargada de inicializar la librería, por lo tanto la llamaremos antes de usar la librería.

```
/*
 Pa_Initialize() is the library initialisation function - call this before
 using the library.

*/

PaError Pa_Initialize( void );
```

Mensajes de error

Cuando se produce un error, la función ***"Pa_GetHostError( )"*** se encarga de retornar su código.

```
/*
 Pa_GetHostError() returns a host specific error code.
 This can be called after receiving a PortAudio error code of paHostError.

*/

long Pa_GetHostError( void );
```

Para que éste error pueda ser entendido por el usuario existe otra función encargada de realizar una traducción de código de error a texto. La función es "***Pa_GetErrorText( )"***

```
/*
 Pa_GetErrorText() translates the supplied PortAudio error number
 into a human readable message.

*/

const char *Pa_GetErrorText( PaError errnum );
```

Formato muestras

Existen una serie de formatos para pasar los datos de sonido entre el Callback y el stream. Cada dispositivo tiene un formato "nativo" que puede ser utilizado cuando se requiere eficacia o control óptimo sobre la conversión necesaria.

PortAudio nos ofrece la posibilidad de usar formatos de distinto tamaño y precisión. Formatos de 1, 2, 3 o 4 bytes.

```
typedef unsigned long PaSampleFormat;
#define paFloat32        ((PaSampleFormat) (1<<0)) /*always available*/
#define paInt16          ((PaSampleFormat) (1<<1)) /*always available*/
#define paInt32          ((PaSampleFormat) (1<<2)) /*always available*/
#define paInt24          ((PaSampleFormat) (1<<3))
#define paPackedInt24    ((PaSampleFormat) (1<<4))
#define paInt8           ((PaSampleFormat) (1<<5))
#define paUInt8          ((PaSampleFormat) (1<<6))
#define paCustomFormat   ((PaSampleFormat) (1<<16))
```

En nuestra aplicación utilizaremos como formato para las muestras de audio “paInt16”, ya que es el formato nativo de nuestros dispositivos (se puede comprobar en la figura 5.2) y así podremos utilizar un tipo short de 2 bytes. Por lo tanto realizaremos la declaración correspondiente:

```
#define PA_SAMPLE_TYPE  paInt16
typedef short SAMPLE;
```

Gestión de los dispositivos

Para empezar a implementar nuestro sistema necesitaremos saber los dispositivos de los que disponemos, su información, propiedades y demás características.

Para empezar comprobaremos el rango de dispositivos que tenemos mediante la función “***Pa_CountDevices( )”***.

```
int Pa_CountDevices( void );
```

El rango irá de 0 a Pa_CountDevices( ) – 1.
Para cada dispositivo detectado tendremos una estructura que contendrá información sobre su nombre, el número máximo de canales de entrada y salida, las frecuencias de muestreo soportadas y el formato nativo correspondiente.

```
typedef struct
{
    int structVersion;
    const char *name;
    int maxInputChannels;
    int maxOutputChannels;
    /* Number of discrete rates, or -1 if range supported. */
    int numSampleRates;
    /* Array of supported sample rates, or {min,max} if range supported. */
    const double *sampleRates;
    PaSampleFormat nativeSampleFormats;
}
PaDeviceInfo;
```

Para obtener la información de un determinado dispositivo pasaremos su identificador a la función ***“Pa_GetDeviceInfo( )”***, que nos retornará un puntero a la estructura del dispositivo especificado.

```
const PaDeviceInfo* Pa_GetDeviceInfo( PaDeviceID device );
```

En nuestro caso utilizaremos 2 funciones que seleccionan los dispositivos de entrada y de salida por defecto, de forma que se realiza automáticamente. Estas funciones son:

```
PaDeviceID Pa_GetDefaultInputDeviceID( void );
PaDeviceID Pa_GetDefaultOutputDeviceID( void );
```

El resultado de las funciones lo pasaremos después a Pa_OpenStream( ) para abrir los dispositivos, como veremos más adelante.

En caso de que algún dispositivo no estuviera disponible la función retorna "***paNoDevice***".

A continuación hemos realizado un pequeño programa que no utilizaremos en nuestro proyecto (ya que seleccionaremos los dispositivos por defecto) que muestra por pantalla las características de los dispositivos de audio de nuestro sistema. En la figura inferior se puede ver el aspecto de la consola de comandos al ejecutar el programa:

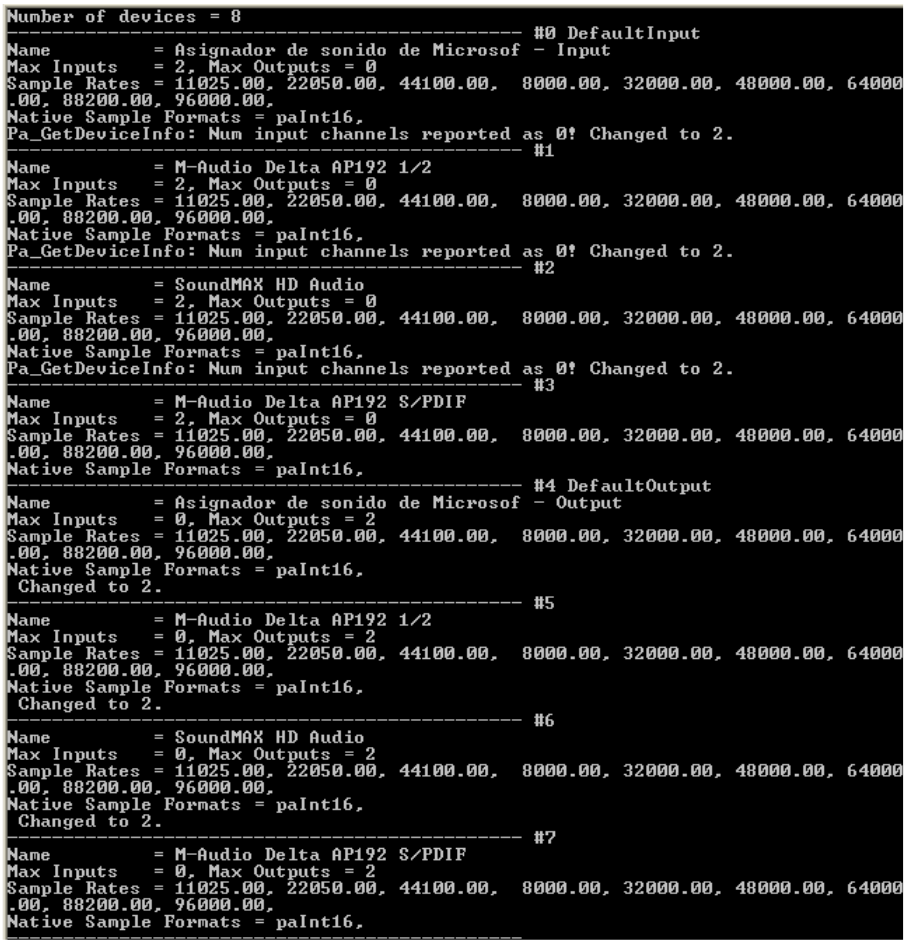


Fig. 5.2 *– Consola de comandos con características de los dispositivos*

Streams

Cualquier programa realizado con PortAudio requiere un stream o corriente de datos para el audio de entrada/salida (un mismo stream puede ser a la vez de entrada y salida).

Cada stream se caracteriza porque tiene una serie de parámetros a configurar como la frecuencia de muestreo, número de canales de entrada y salida, muestras por buffer, dispositivos asociados…etc.

Lo primero que realizaremos es la declaración del stream.

```
PortAudioStream *stream; // DECLARAMOS EL STREAM DE AUDIO
```

Para realizar la creación del stream deseado tendremos que llamar a la función ***"Pa_OpenStream( )"*** y pasarle una serie de parámetros. A continuación podemos ver su estructura.

```
PaError Pa_OpenStream( PortAudioStream** stream,
                       PaDeviceID inputDevice,
                       int numInputChannels,
                       PaSampleFormat inputSampleFormat,
                       void *inputDriverInfo,
                       PaDeviceID outputDevice,
                       int numOutputChannels,
                       PaSampleFormat outputSampleFormat,
                       void *outputDriverInfo,
                       double sampleRate,
                       unsigned long framesPerBuffer,
                       unsigned long numberOfBuffers,
                       PaStreamFlags streamFlags,
                       PortAudioCallback *callback,
                       void *userData );
```

El primer parámetro es el stream que hemos declarado. Los siguientes 4 parámetros hacen referencia al dispositivo input que desearemos abrir. Tenemos el identificador del dispositivo de entrada que queremos seleccionar (que lo habremos detectado como explicamos anteriormente), el número de canales de entrada, su respectivo formato y información sobre su driver (puntero opcional a estructura de datos específica del driver). Los siguientes 4 parámetros son iguales únicamente diferenciando que hacen referencia al dispositivo de output.

Los parámetros que siguen son la frecuencia de muestreo (sampleRate), el tamaño de la ventana de muestras (framesPerBuffer), el número de ventanas (numberOfBuffers), la función callback y datos de intercambio que se pasarán a callback.

Vista la estructura de la función que crea el stream veremos los parámetros configurados:

```
err = Pa_OpenStream(
          &stream,
          Pa_GetDefaultInputDeviceID(),
          1,
          PA_SAMPLE_TYPE,
          NULL,
          Pa_GetDefaultOutputDeviceID(),
          1,
          PA_SAMPLE_TYPE,
          NULL,
          SAMPLE_RATE,
          FRAMES_PER_BUFFER,
          0,
          paClipOff,
          SonidoCallback,
          NULL );
```

Como se puede ver, el primer parámetro pasado es el stream declarado anteriormente. Para seleccionar los dispositivos de entrada y salida hemos utilizado las funciones comentadas anteriormente que seleccionan los dispositivos por defecto. Configuramos entrada y salida de un solo canal mono. El formato de los datos (PA_SAMPLE_TYPE), como explicamos anteriormente, es “PaInt16”. Como no utilizamos información de los drivers usamos NULL. Los demás valores son variables que hemos definido en el programa. (SAMPLE_RATE, FRAMES_PER_BUFFER, …etc).

Para finalizar solo queda iniciar el stream antes de llamar a la función callback. Lo realizaremos mediante ***Pa_StartStream(PortAudioStream*)***.

Hay que mencionar una serie de funciones necesarias que sirven para manipular el stream.

Se puede consultar mediante ***Pa_StreamActive(PortAudioStream*).***

Una vez finalizada la función callback, para terminar el programa deberemos parar y cerrar el stream. Lo realizaremos con las siguientes funciones:

Se para mediante ***Pa_StopStream(PortAudioStream*)***
Pa_AbortStream(PortAudioStream*)

Se cierra mediante ***Pa_CloseStream(PortAudioStream*)***

Función Callback

Esta es la función principal de proceso de nuestro programa. La función callback es llamada una vez se ha creado y abierto el stream de audio.

La función callback se llama desde un proceso que nosotros no controlamos (normalmente desde una API).

Se caracteriza por tener un funcionamiento cíclico, ya que es llamada periódicamente. Se adquieren un determinado número de muestras de audio (marcado por el tamaño del buffer) y se procesan dentro de la función callback. Una vez terminada la función, ésta se vuelve a iniciar de nuevo capturando las siguientes muestras de audio consecutivas, y así sucesivamente. Esto se traduce en que la función callback se ejecutará cada “framesPerBuffer” muestras y nosotros podremos realizar nuestro procesado sobre las muestras de cada buffer.

Esta función recibe una serie de parámetros. A continuación comentaremos los que utilizaremos para la programación del sistema.

```
static int SonidoCallback( void *inputBuffer, void *outputBuffer,
                           unsigned long framesPerBuffer,
                           PaTimestamp outTime, void *userData )
```

inputBuffer es un puntero que apunta hacia las muestras de audio de entrada que se capturan. Por otra parte, ****outputBuffer*** apunta hacia el out del sistema, con la finalidad de poder enviar muestras hacia la salida y escucharlas.

El parámetro ***framesPerBuffer*** goza de bastante importancia ya que nos delimitará cuantas muestras tomamos para realizar el procesado de audio y cada cuantas muestras se ejecutará la función callback.

Lo primero que realizaremos al iniciar la función callback será declarar 2 punteros del tipo establecido (paInt16) para poder acceder a las muestras del buffer de entrada (in) y acceder al buffer de salida (salida de audio, out).

```
SAMPLE *in = (SAMPLE*)inputBuffer;  // DEFINICION DE PUNTERO TIPO SAMPLE QUE CONTIENE LA
                                    // LA DIRECCION DE MEMORIA DE LAS MUESTRAS DE ENTRADA.

SAMPLE *out = (SAMPLE*)outputBuffer; // DEFINICION DE PUNTERO TIPO SAMPLE QUE CONTIENE
                                     // LA DIRECCION DE MEMORIA DE LA SALIDA OUT
```

En este momento tendremos las muestras de audio del buffer de entrada en una array (in) a la cual podremos acceder.

Si deseamos escuchar las muestras que tenemos en esa array (in), deberemos recorrer cada posición y escribirla en su correspondiente posición del buffer de salida. A continuación se muestra el proceso:

```
for (int g = 0; g < framesPerBuffer; g++)

{

    *out++ = *in++;

}
```

Incrementamos la posición de los punteros “framesPerBuffer” veces, de forma que finalmente recorremos y escribimos una a una todas las posiciones de los respectivos buffers.

Una vez hecho empezaremos a realizar el procesado sobre las muestras de entrada de audio. Deberemos recuperar de nuevo la posición inicial a la que apuntaba “*in” (será la primera muestra), ya que después del bucle se ha desplazado. Lo realizaremos de la siguiente forma:

```
in = (SAMPLE *)inputBuffer;
```

Ahora ya podremos pasar la array de muestras “in” a la funciones de detección de pitch y cálculo de las notas.

Cerrar librería

Una vez finalizado el programa deberemos cerrar la librería correctamente mediante la función ***Pa_Terminate( )***.

```
/*
 Pa_Terminate() is the library termination function - call this after
 using the library.

*/

PaError Pa_Terminate( void );
```

5.3 Funciones procesado audio

Hace falta destacar que para adaptar los algoritmos de detección desarrollados previamente con nuestra aplicación en tiempo real se han tenido que crear y modificar algunas funciones. A continuación se explicarán los nuevos cambios y funciones.

Función “Previo”

Esta función ha sido creada con la finalidad de dar ganancia a la señal de entrada del sistema, ya que el previo de guitarra que utilicé para las pruebas daba una señal excesivamente baja y no se escuchaba nada al monitorizarla por la aplicación. Por lo tanto esta función es susceptible a cambios en función del previo utilizado.

```
void previo(short *in)
{
    int gan,g;

    gan = 10; // DAMOS UNA GANANCIA DE 10

    for (g = 0; g < FRAMES_PER_BUFFER; g++)// BUCLE PARA MULTIPLICAR CADA MUESTRAS POR LA GANANCIA
    {
        in[g] = gan * in[g];
    }

}
```

Como se puede apreciar el bucle for multiplica cada muestra del buffer de entrada por una ganancia/atenuación seleccionable en función del previo utilizado.

Función “Variables”

Esta función ha sido creada con la finalidad de adaptar el funcionamiento del algoritmo de detección ante cualquier configuración de los parámetros de frecuencia de muestreo y número de puntos de la FFT. El sistem ha de permitir la entrada de distintos valores de esos parámetros con la finalidad de analizar los resultados obtenidos en el procesado.

El algoritmo de detección previamente desarrollado no estaba orientado para aplicaciones en tiempo real y no permitía seleccionar distintos parámetros, solo estaba configurado y funcionaba para ficheros entrantes de audio con una Fm = 44100 y un valor de NFFT de 32768.

Este algoritmo basa su funcionamiento en dividir el espectro en los rangos de muestras de la FFT que abarcan las frecuencias de cada cuerda. Se define el rango de cada cuerda hasta el 5º traste (a partir de él la nota se puede tocar en la siguiente cuerda más aguda), excepto en la primera cuerda (la más aguda) que tendrá un rango mucho mayor. Todo y esto lo que importa es la nota que se toca y no la cuerda que la toca.

Visto lo anterior, el problema surge al utilizar distintas configuraciones de Fm y NFFT, ya que su variación provocará que la longitud de los vectores de cada cuerda extraídos del espectro sea distinta.

Es obvia la necesidad de desarrollar una función que a partir de los parámetros de entrada deseados calcule cual es la nueva longitud de cada uno de los 6 vectores de las cuerdas.

La función sigue el siguiente esquema:

```
void variables(int Fm, int NFFT, // PARAMETROS DE ENTRADA
               double *offsets, int *lens) // PARAMETROS ENTRADA/SALIDA

{
    // DEFINO LOS RANGOS DE CADA CUERDA EN PTOS DE FFT


    offsets[0] = (NFFT * 79.40 / Fm);
    offsets[1] = (NFFT * 107.00 / Fm);
    offsets[2] = (NFFT * 142.0 / Fm);
    offsets[3] = (NFFT * 190.00 / Fm);
    offsets[4] = (NFFT * 240.00 / Fm);
    offsets[5] = (NFFT * 320.3 / Fm);
    offsets[6] = (NFFT * 1600 / Fm);


    // CALCULAMOS LAS LONGITUDES DE LOS VECTORES DE CADA CUERDA EN MUESTRAS

    lens[0] = int(offsets[1] - offsets[0] - 1);
    lens[1] = int(offsets[2] - offsets[1] - 1);
    lens[2] = int(offsets[3] - offsets[2] - 1);
    lens[3] = int(offsets[4] - offsets[3] - 1);
    lens[4] = int(offsets[5] - offsets[4] - 1);
    lens[5] = int(offsets[6] - offsets[5] - 1);



}
```

Como parámetros de entrada recibimos la Fm y NFFT. Los parámetros de salida corresponden a 2 arrays que contienen los límites que abarcan cada una de las cuerdas y la longitud de cada una de ellas en muestras de la FFT.

A continuación, conociendo las frecuencias analógicas de cada cuerda podemos extraer los puntos NFFT (offsets) correspondientes utilizando la siguiente deducción:

$$k = NFFT * \frac{Frec}{Fm}$$

Hemos utilizado variables tipo “double” para no perder precisión en las operaciones.

Por último solo queda calcular la longitud en muestras de la FFT de cada una de las cuerdas de la guitarra. Conociendo los valores anteriores de los límites (offsets) solo tendremos que ir restando las cuerdas consecutivas para saber cual es su tamaño (lens).

Finalizado esto deberemos enviar estos valores a la función de detección para que se ejecute correctamente con los parámetros que hemos seleccionado.

Modificaciones en “detección”

Se han realizado unos pequeños cambios para adaptar esta función al sistema en tiempo real.

A la función se le han de pasar los 2 nuevos parámetros calculados anteriormente.

```
void deteccion (double *y, //Hps de entrada
                int Fm,    //Frecuencia de muestreo
                int *z,    //Vector de salida con los maximos de cada cuerda
                double *offsets, //Valores de offsets de las cuerdas
                int *lens) // Longitud en muestras FFT de cada cuerda
```

A continuación tendremos que declarar los vectores de cada cuerda. Los declararemos como punteros para posteriormente poder reservar espacio en memoria dinámica.

```
double *s1, *s2, *s3, *s4, *s5, *s6; //PUNTEROS DE CADA UNA DE LAS CUERDAS
```

A continuación reservaremos el espacio necesario en memoria dinámica para cada vector mediante la función “malloc”.

```
s1 = (double *) malloc (lens[0] * sizeof (double));
s2 = (double *) malloc (lens[1] * sizeof (double));
s3 = (double *) malloc (lens[2] * sizeof (double));
s4 = (double *) malloc (lens[3] * sizeof (double));
s5 = (double *) malloc (lens[4] * sizeof (double));
s6 = (double *) malloc (lens[5] * sizeof (double));
```

No podemos realizar la declaración de forma estática ya que las longitudes de las cuerdas son un parámetro de entrada que no está disponible en las declaraciones iniciales.

Por otra parte los valores de threshold han de ser modificados de acuerdo con el nivel de señal de entrada para que haya una buena detección de máximos.

Para finalizar debemos aumentar el rango frecuencial que abarca la cuerda más aguda, ya que ahora nuestro instrumento es una guitarra eléctrica capaz de llegar a registros más elevados. Antes la detección alcanzaba hasta la nota B3 (493,88 Hz). La ampliación llegará hasta F#5 (1479,78 Hz).

5.4 Procesado MIDI en tiempo real mediante PortMidi

Una vez visto el bloque anterior deberemos desarrollar la parte de MIDI en tiempo real. Como hemos comentado anteriormente, utilizaremos la librería PortMidi para ello.

A continuación veremos su funcionamiento y el desarrollo necesario para implementar éste procesado.

Inicialización de la librería

Antes de empezar a utilizar la librería PortMidi será necesario inicializarla. La función encargada de realizarlo es ***"Pm_Initialize( )".***

```
/*
    Pm_Initialize() is the library initialisation function - call this before
    using the library.
*/

PmError Pm_Initialize( void );
```

Gestión de los dispositivos MIDI

Deberemos programar un mecanismo que nos de información acerca de los puertos MIDI disponibles en el sistema y de sus propiedades, con la finalidad de seleccionar el que nos interese.

Cada dispositivo-puerto MIDI (hardware o virtual) tiene un driver asociado. Entre la aplicación y el driver existe una capa intermedia (winmm.dll). Por lo tanto nos será necesario linkar a nuestro proyecto la librería winmm.lib, la cual nos ofrecerá unas funciones básicas MIDI necesarias: preguntar por los puertos disponibles de entrada y de salida, sus propiedades, abrirlos y cerrarlos, recibir y mandar mensajes…

Para conocer los dispositivos disponibles de nuestro sistema utilizaremos la función ***"Pm_CountDevices( )",*** que los numerará de 0 a Pm_CountDevices( ) – 1. Una vez realizado lo anterior deberemos conocer las características de cada uno de los dispositivos reconocidos, ya que deberemos seleccionar uno específico, en nuestro caso un puerto MIDI OUT.

Para cada dispositivo tenemos una estructura que contiene información acerca de sus propiedades principales. Sigue el siguiente esquema:

```
typedef struct {
    int structVersion;
    const char *interf;
    const char *name;
    int input;
    int output;
    int opened;

} PmDeviceInfo;
```

El puntero tipo char "*interf" apunta a una cadena de caracteres que contienen el tipo de API MIDI utilizado (por ejemplo MMSystem o DirectX).
El otro puntero tipo char "*name" apunta, como se puede deducir, a una cadena de caracteres que contienen el nombre del dispositivo MIDI.

Por último mencionar las variables "input" y "output". "input" tendrá un valor de 1 (true) si el puerto es de entrada, y con "output" igual, pero cuando el puerto sea de salida.

Para obtener la información de un determinado dispositivo pasaremos su identificador a la función ***"Pm_GetDeviceInfo( )"***que nos retornará un puntero a la estructura del dispositivo especificado.

Selección del puerto de salida MIDI

Para seleccionar el puerto MIDI adecuado solo listaremos los que son de salida, ya que los de entrada no los utilizaremos. Para cada puerto de salida detectado haremos un print por pantalla de su identificador, el tipo de API que utiliza y de su nombre.

Utilizaremos un bucle for para recorrer los dispositivos MIDI de salida del sistema y hacer el print de las características mencionadas anteriormente:

```
// LISTAMOS LOS DISPOSITIVOS DE SALIDA

for (disp = 0; disp < Pm_CountDevices(); disp++) {
    const PmDeviceInfo *info = Pm_GetDeviceInfo(disp);

        printf("%d: %s, %s", disp, info->interf, info->name);

        if (info->output) printf(" (output)");
        printf("\n");

}
```

A continuación pediremos que se introduzca por teclado el identificador del puerto MIDI OUT deseado y lo guardaremos para posteriormente utilizarlo.

```
// DETERMINAMOS QUE OUTPUT VAMOS A UTILIZAR
disp = get_number("Inserta el dispositivo de salida MIDI deseado :\n ");
```

En la figura inferior podemos ver el aspecto que muestra la ventana de comandos al programar la selección del puerto MIDI OUT como hemos explicado:

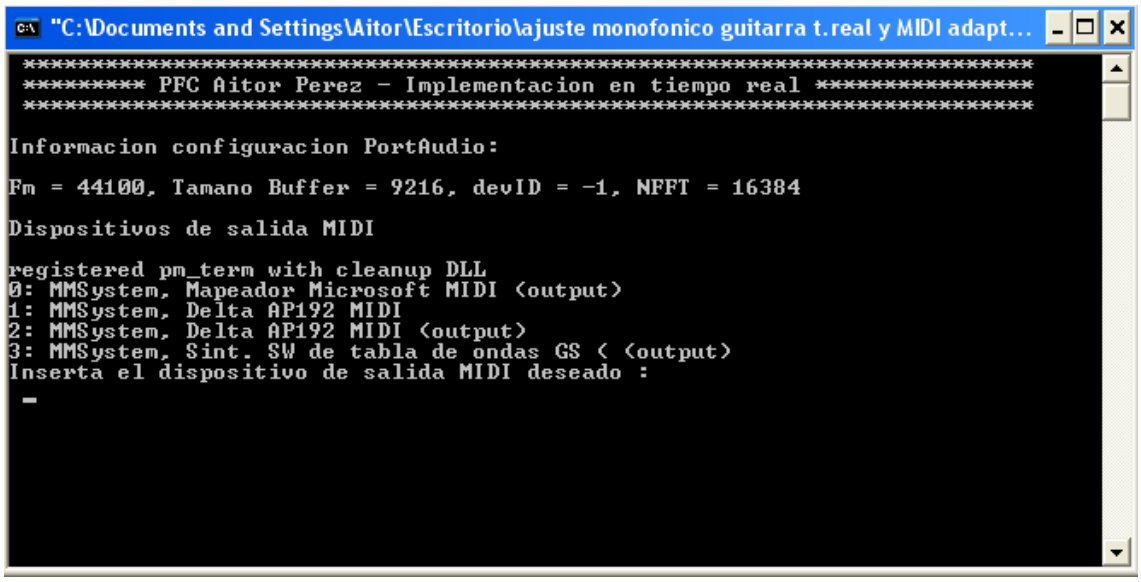


Fig. 5.3 *– Consola de comandos para selección de puerto MIDI OUT*

Podemos apreciar que también se muestran los parámetros de configuración inicial.

Stream y apertura del dispositivo de salida MIDI

Una vez que conocido el dispositivo MIDI de salida que queremos utilizar deberemos abrirlo correctamente. Para ello, antes de nada, deberemos declarar un stream MIDI, que será el que utilizaremos para gestionar la información MIDI que enviaremos al puerto correspondiente.

```
PmStream * midi; // DECLARAMOS EL STREAM PARA EL MIDI
```

Antes también tendremos inicializaremos un timer para tener una referencia temporal.

```
TIME_START; // INICIAMOS EL CONTADOR DE TIEMPO
```

Emplearemos la función ***"Pm_OpenOutput( )"*** , a la cual le deberemos pasaremos una serie de parámetros para abrir correctamente el puerto deseado.

```
PmError Pm_OpenOutput( PortMidiStream** stream,
                PmDeviceID outputDevice,
                void *outputDriverInfo,
                long bufferSize,
                PmTimeProcPtr time_proc,
                void *time_info,
                long latency );
```

El primer parámetro hace referencia al corriente de datos MIDI declarada anteriormente. El siguiente elemento es el identificador del dispositivo de salida MIDI deseado.

"*outputDriverInfo" es un puntero opcional que apunta a una estructura de datos específica del driver.

"bufferSize" especifica el número de eventos MIDI de salida acumulables en el buffer esperando para salir.

El siguiente parámetro sirve para indicar una referencia temporal al sistema (Timestamp). En éste caso es un reloj con una precisión de milisegundos. Será necesario para la escritura de información MIDI.

Escritura de mensajes MIDI

Una vez abierto el dispositivo ya podemos empezar a escribir mensajes MIDI a partir de los parámetros extraídos del audio.

Para generar los mensajes necesitaremos un buffer, el cual tendremos que ir actualizando temporalmente y al cual también le especificaremos el tipo de mensaje deseamos utilizar. El buffer que emplearemos es de tamaño 1 para escribir de forma secuencial, por lo que solo podrá albergar 1 evento MIDI.

A continuación se muestra la metodología para generar mensajes MIDI y escribirlos.

```
buffer[0].timestamp = TIME_PROC(TIME_INFO);
buffer[0].message = Pm_Message(0x90,act,100);

Pm_Write(midi,buffer,1);
```

Por una parte se actualiza el valor temporal del Timestamp del buffer para cada nuevo mensaje. Después, mediante ***"Pm_Message( )"*** escribiremos el byte de status y los 2 bytes siguientes de datos.

Pm_Message(status, data1, data2)

Por último escribiremos la información MIDI a partir del buffer anterior. Es necesario especificar el stream MIDI, el buffer y su respectivo tamaño.

Pm_Write(PortMidiStream *stream, PmEvent *buffer, long length);

Una vez realizado el mensaje se enviará correctamente a través del puerto MIDI OUT que hayamos seleccionado.

Cerrar la librería

Cuando finalicemos el uso de la librería en el programa deberemos cerrarla adecuadamente. Lo haremos al final mediante el uso de la función ***"Pm_Terminate( )".***

```
/*
    Pm_Terminate() is the library termination function - call this after
    using the library.
*/

PmError Pm_Terminate( void );
```

5.5 Algoritmo monofonía

Como comentamos en la presentación del proyecto, el conversor de audio a MIDI en tiempo real que desarrollaremos será monofónico, lo que supondrá que solo podrá detectar y hacer sonar una única nota a la vez a la salida.

Mediante éste algoritmo también solucionaremos posibles errores debidos al problema de detección de armónicos de una fundamental como notas. Como al ser monofónico solo tenemos que detectar una nota, haremos un análisis de la detección que se ha producido para cada bloque y solo nos quedaremos con la nota más grave detectada, descartando las demás (que serán armónicos si solo hemos tocando una nota con el instrumento). Mediante éste proceso podremos eliminar eficientemente los armónicos que puedan aparecer y mejoraremos el funcionamiento del conversor.

El funcionamiento del algoritmo desarrollado se basa en analizar la matriz de notas detectadas y encontrar la más grave. Una vez encontrada ésta se ponen a silencio todas las demás, que obviamente serán más agudas. A continuación se envía la información hacía el algoritmo de escritura MIDI, que veremos a continuación.

5.6 Algoritmo de escritura MIDI

Visto el funcionamiento de la librería PortMidi para gestionar la información MIDI, deberemos ver que tipo de metodología utilizamos para generar los mensajes a partir de los parámetros que iremos extrayendo del audio en tiempo real.

Para cada bloque de audio procesado en la función Callback obtenemos un vector de 6 posiciones, donde cada una de estas contendrá la nota detectada en la cuerda correspondiente, o en caso de que no haya nota un silencio. Como hemos comentado en el punto anterior, sólo tendremos una nota detectada por vector y las demás posiciones de silencio, ya que el conversor es monofónico.

Recordemos que nuestra función Callback se ejecuta cada 9216 muestras a una frecuencia de muestreo de Fm = 44.100 Hz. Esto provoca que nos lleguen unos 4 vectores consecutivos por segundo (que contendrán las notas detectadas) a lo largo de todo el tiempo que estemos utilizando la aplicación.

La metodología que emplearemos en el algoritmo será la de comparar siempre 2 vectores consecutivos de notas con la finalidad de analizar los cambios que se han producido entre ellos, actualizar los eventos MIDI e inmediatamente escribir los nuevos eventos por el puerto de salida en tiempo real.

Para ello necesitaremos declarar un vector auxiliar de 6 posiciones para poder guardar el vector anterior al que se está procesando en el Callback en ese mismo momento. Las posiciones del vector auxiliar las inicializaremos a silencios para la primera vez que se utilice. El valor de silencio lo representaremos mediante un 1.

```
int vect_ant[] = { 1, 1, 1, 1, 1, 1 };
```

La comparación de los 2 vectores será de cada posición con su correspondiente en el otro vector. Se detectarán los cambios de nota y la aparición de silencios. La duración de los eventos viene determinada al implementar las transiciones entre los eventos.

Deberemos analizar las posibles transiciones que se pueden dar entre las notas de los 2 vectores:

Posibilidades transición entre eventos MIDI consecutivos

Evento Anterior	Evento Actual	Acción	Mensajes
Silencio	Nota	Iniciar nota	NoteOn (Nota)
Nota	Silencio	Finalizar nota	NoteOff (Nota)
Nota	Nota''	Cambio de nota	NoteOff (Nota) y NoteOn (Nota'')
Silencio	Silencio	Ninguna	Ninguno
Nota	Nota	Ninguna	Ninguno

Fig. 5.4 *– Tabla posibles transiciones eventos MIDI*

El siguiente diagrama muestra esquemáticamente los posibles cambios que se pueden producir entre los eventos:

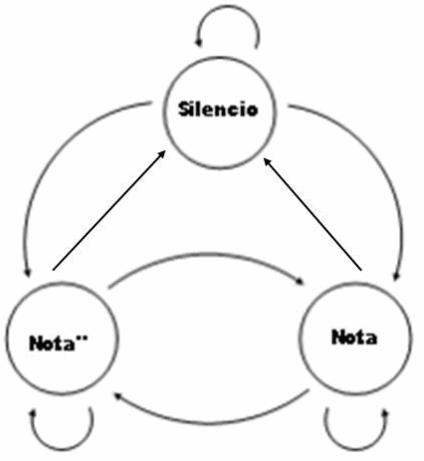


Fig. 5.5 *– Diagrama eventos MIDI*

Vistas las posibles transiciones podemos realizar la implementación de la función.

```
void funcmidi(int *a, int *p)
{

    int i, act, post;

    for (i = 0; i < 6; i++) // RECORREMOS CADA UNA DE LAS CUERDAS
    {

        act = a[i]; // SELECCIONAMOS LAS NOTAS DE LA CUERDA CORRESPONDIENTE
        post = p[i];
```

Mediante las variables "act" y "post" iremos seleccionando las posiciones del vector anterior y el actual para cada una de las cuerdas (utilizando el bucle for).

Después de realizar todas las funciones MIDI para cada posición habrá que actualizar el vector posterior:

```
      // ACTUALIZAMOS EL VECTOR DE NOTAS ANTERIORES
      p[i] = a[i];


  }
```

A continuación veremos la implementación de los distintos casos. Hay que mencionar que siempre es necesaria la actualización del valor del Timestamp. Para ello utilizaremos "**TIME_PROC(TIME_INFO)**", que nos dará el valor temporal actual. El mensaje de NoteOn utilizando el canal 0 (utilizaremos ese canal) en hexadecimal corresponde a 90. Como velocidad fijaremos siempre 100. Hay que aclarar que para realizar los NoteOff utilizaremos el mensaje NoteOn con una velocidad de 0, que es equivalente.

Caso 1

```
if ((post == 1) && (act != 1))
{
     buffer[0].timestamp = TIME_PROC(TIME_INFO);
     buffer[0].message = Pm_Message(0x90, act, 100); //HACEMOS NOTEON

     Pm_Write(midi,buffer,1); // ESCRIBIMOS EL MENSAJE EN MIDI OUT
}
```

Caso 2

```
if ((act == 1) && (post != 1))
{
     buffer[0].timestamp = TIME_PROC(TIME_INFO);
     buffer[0].message = Pm_Message(0x90, post, 0); //HACEMOS NOTEOFF

     Pm_Write(midi,buffer,1); // ESCRIBIMOS EL MENSAJE EN MIDI OUT
}
```

Caso 3

```
if ((act != post) && (post != 1) && (act != 1))
{

     buffer[0].timestamp = TIME_PROC(TIME_INFO);
     buffer[0].message = Pm_Message(0x90, post, 0); //HACEMOS NOTEOFF

     Pm_Write(midi,buffer,1); // ESCRIBIMOS EL MENSAJE EN MIDI OUT

     buffer[0].timestamp = TIME_PROC(TIME_INFO);
     buffer[0].message = Pm_Message(0x90,act,100); // HACEMOS EL NOTEON

     Pm_Write(midi,buffer,1); // ESCRIBIMOS EL MENSAJE EN MIDI OUT

}
```

Los 2 siguientes casos no requieren implementación, ya que si no sucede ningún cambio entre dos eventos consecutivos no hay que realizar nada.

5.6 Desarrollo Interfaz Gráfica

La creación de una interfaz gráfica fue pensada con la finalidad de obtener una visualización más clara y fácil de los resultados durante las pruebas realizadas en la parte de detección del proyecto. Hay que aclarar que esta interfaz no funciona con audio en tiempo real, sino que carga un archivo de audio y realiza el análisis correspondiente.

Para el desarrollo de la interfaz emplearemos el GUIDE de Matlab, un ambiente de desarrollo de interfaz de usuario.

GUIDE es un entorno de programación visual que contiene un conjunto de herramientas para crear interfaces gráficas muy parecidas a las aplicaciones de Windows. Estas herramientas simplifican el proceso de creación y de programación.

Una aplicación consta de dos archivos: *.m* y *.fig*. El fichero *.fig* hace referencia a la consola de edición de la parte gráfica de la aplicación a implementar, es decir, permite diseñar los elementos que formarán la interfaz. En cambio el archivo *.m* es el ejecutable de Matlab donde se determinan las subrutinas que van asociadas a cada elemento creado en el .fig. Permite determinar que pasará al pulsar cualquiera de los botones.

Para empezar a trabajar con GUIDE en Matlab tendremos que ejecutar instrucción "guide" en la ventana de comandos. Aparecerá un cuadro de diálogo en la que seleccionaremos iniciar un nuevo GUI en blanco.

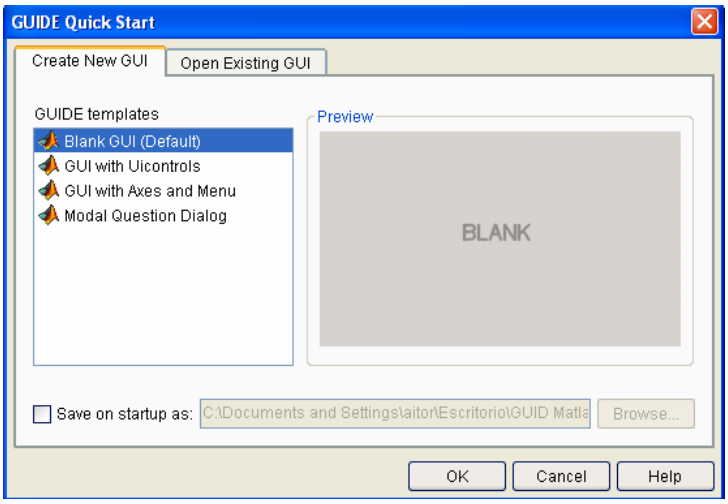


***Fig. 5.6** – Cuadro diálogo GUIDE*

Una vez con la plantilla en blanco empezaremos a colocar los botones de la paleta de componentes (figura 5.7) que necesitemos y utilizaremos las herramientas que se ofrecen para diseñar nuestra pantalla principal en la interfaz. Mostraremos distintas gráficas, opción de configurar parámetros, cargar diferentes archivos de audio, escucharlos y ver información de estos…etc.

Podremos modificar las propiedades de cada uno de los componentes en el "property inspector".

***Fig. 5.7**– Paleta de componentes*

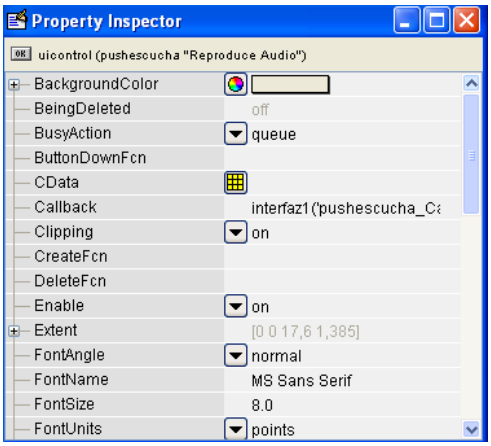


***Fig. 5.8** – Property Inspector*

En nuestro diseño incluiremos los siguientes elementos:

- ✓ Lista muestras de audio
- ✓ Descripción de cada muestra de audio en una caja
- ✓ Botón para seleccionar archivos de audio externos y su descripción
- ✓ Botón para reproducir el audio seleccionado
- ✓ Selección de modo de funcionamiento
- ✓ Botones de información y de ayuda
- ✓ Configuración de parámetros
- ✓ Botones play/stop
- ✓ 3 gráficas sobre procesado de audio
    - o Trama temporal
    - o Transformada de Fourier
    - o HPS

Este es el aspecto gráfico que muestra el editor de la interfaz una vez colocados todos los componentes y herramientas:

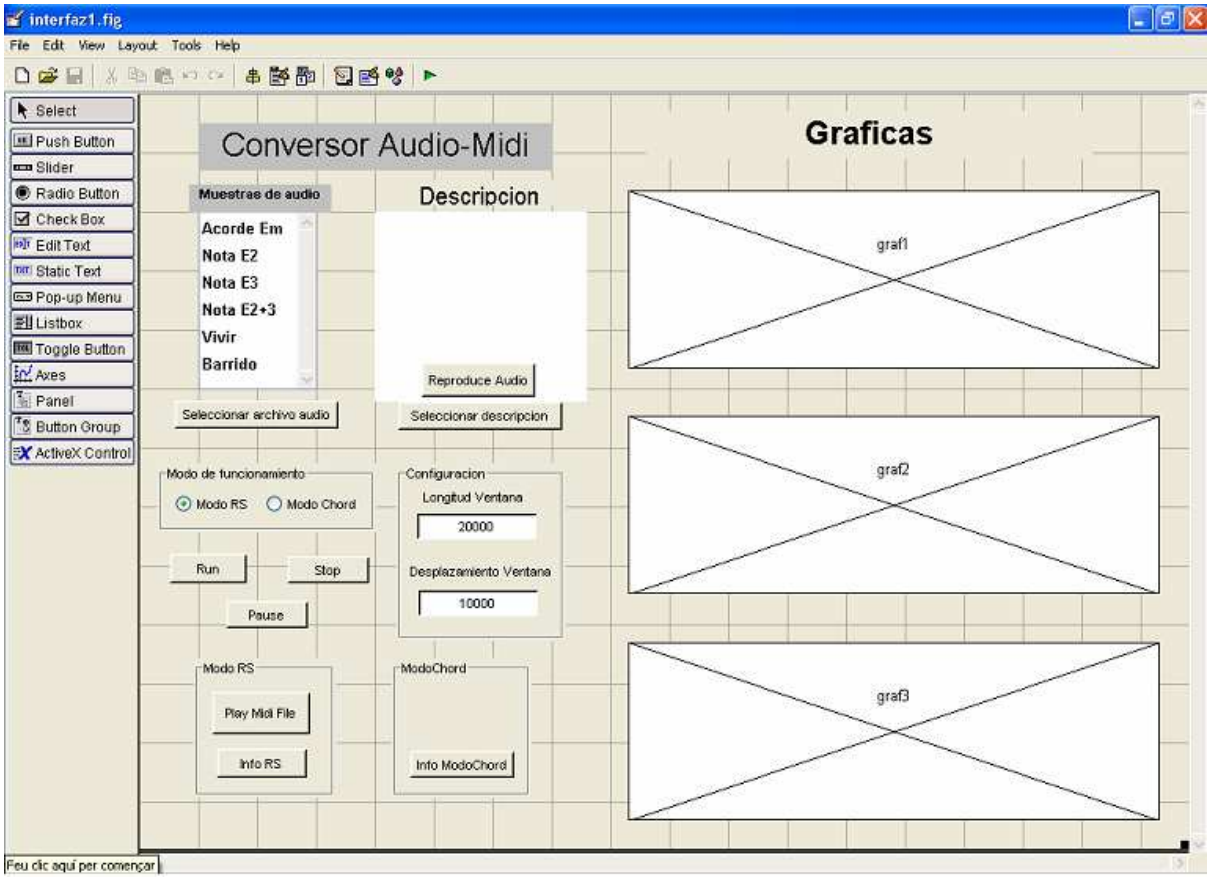


Fig. 5.9 *– Editor GUIDE*

A continuación se ha de realizar la tarea de programar el fichero .m de los distintos botones de la interfaz para que cada elemento cumpla la función correcta.

Una de las opciones más importantes a configurar de cada botón es "*View Callbacks*", la cual, al ejecutarla, abre el archivo .m asociado a nuestro diseño y nos posiciona en la parte del programa que corresponde a la subrutina que se ejecutará cuando se realice una determinada acción sobre el elemento que estamos editando.

La programación del fichero .m de la interfaz se adjunta al proyecto, de forma que ahí se puede ver la programación cada una de las distintas subrutinas.

Finalmente éste es el aspecto que muestra la interfaz gráfica cuando es ejecutada.

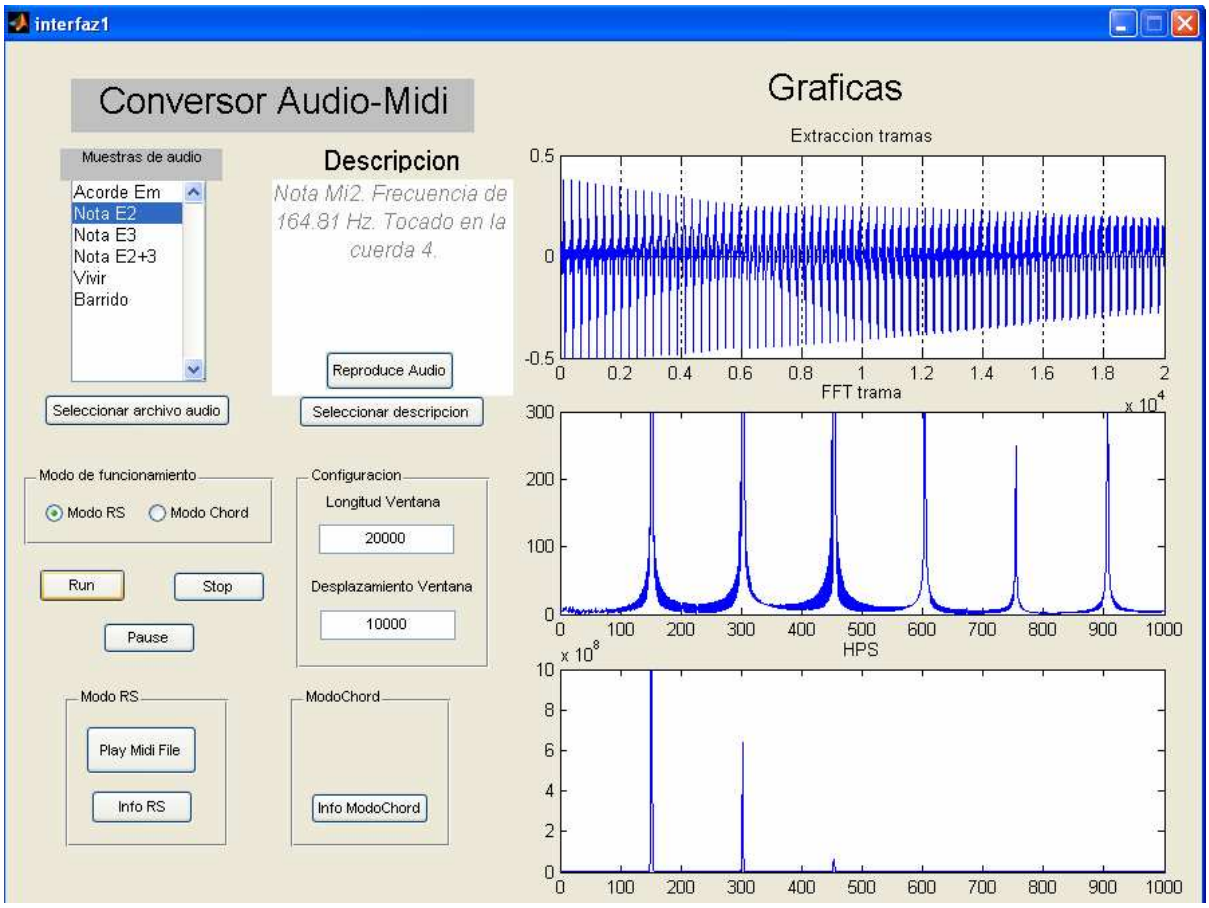


Fig. 5.10 *– Resultado y ejecución de la interfaz gráfica Matlab*

Podemos comprobar que mediante la interfaz gráfica desarrollada en Matlab es mucho más cómoda la realización de pruebas y visualización de resultados.

6. Testeo del sistema implementado y obtención de resultados

Una vez acabado el diseño y la implementación del sistema de conversión de audio a MIDI en tiempo real es la hora de hacer una serie de pruebas para poder realizar los últimos ajustes y comprobar que cumple las expectativas fijadas.

6.1 Configuración física del sistema

Según vimos en un capítulo anterior, el esquema de nuestro sistema sigue una estructura como la de la figura 6.1.

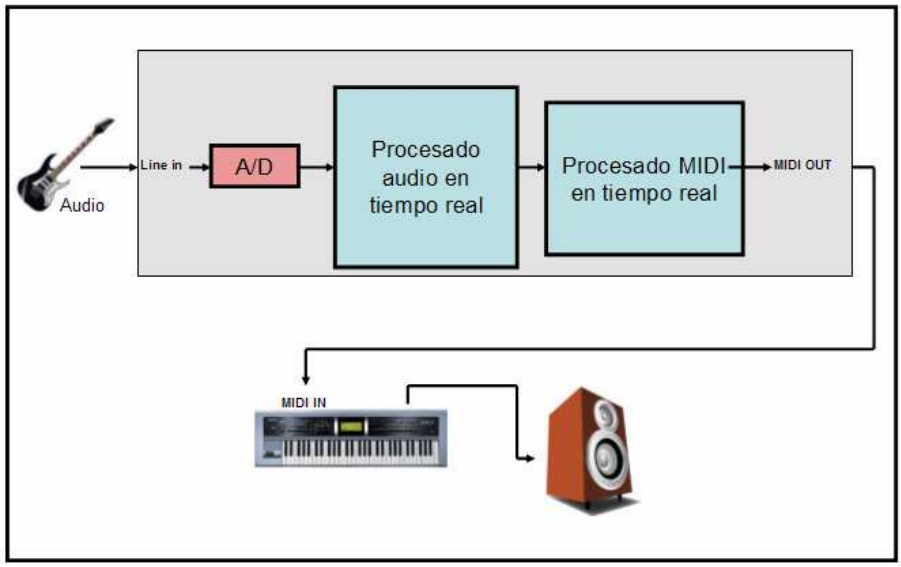


Fig. 6.1 *– Esquema del sistema*

Como podemos ver la guitarra va conectada a la entrada de la tarjeta de sonido de nuestro PC. A continuación nuestra aplicación captura y digitaliza el sonido para procesarlo en tiempo real con la finalidad de extraer una serie de parámetros que luego se envían al bloque que gestiona y genera información MIDI a partir de ellos. Esta información MIDI generada se ha de enviar también en tiempo real por el puerto MIDI OUT de la tarjeta de sonido, con la finalidad de que llegue a un sintetizador externo MIDI que sintetizará la información en audio y la hará sonar.

Para la realización de las pruebas variaremos ligeramente el esquema, todo y que el principio de funcionamiento será exactamente el mismo. Lo único que cambiaremos será que no utilizaremos un sintetizador externo, sino que nos será mucho más fácil emplear software dentro nuestro PC para sintetizar la información MIDI.

Para montar el esquema explicado anteriormente deberemos conectar mediante un cable MIDI tipo DIN de 5 pines la salida de la tarjeta MIDI OUT a la entrada MIDI IN, de tal forma que la salida MIDI del conversor (MIDI OUT) estará entrando de nuevo por el puerto MIDI de entrada y la podremos analizar con el software adecuado.

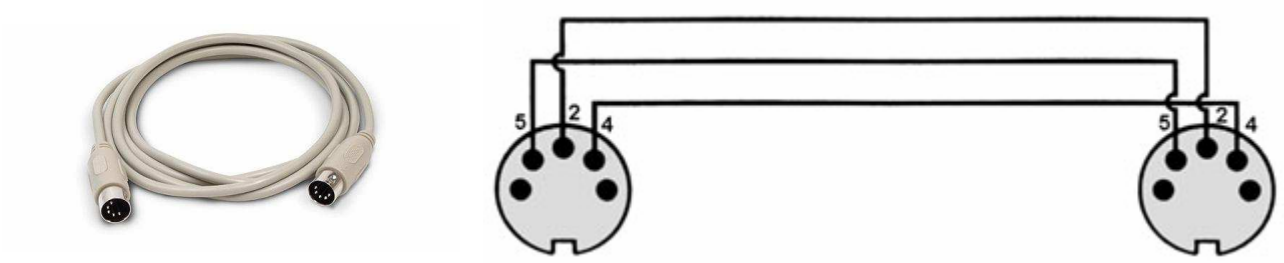


Fig. 6.2 *- Cable MIDI y conexión*

Utilizaremos el software "Cubase SX3" de Steinberg, el cual incluye funciones MIDI muy interesantes y nos permitirá visualizar y procesar la información perfectamente.

Fig. 6.3 *– Software Cubase SX*

Otro elemento clave a la hora de procesar el audio y el MIDI es la tarjeta de sonido. Una tarjeta de sonido común (como las que vienen integradas en los PC's) nos introducirá una latencia elevada, tanto en MIDI como en sonido a la hora trabajar con aplicaciones orientadas al mundo del proceso de audio. Otro hecho a remarcar es que necesitamos una serie de entradas y salidas físicas específicas, en nuestro caso las MIDI. Las tarjetas convencionales no llevan, en cambio las más especializadas sí.

En nuestro hardware disponemos de una buena tarjeta de audio, por lo tanto no tendremos problemas. La tarjeta pertenece a la casa M-Audio, y es el modelo Audiophile 192. En la figura inferior podemos ver una imagen.

Fig. 6.4 *– Tarjeta M-Audio Audiophile 192*

Esta tarjeta nos brinda un gran abanico de frecuencias de muestreo, hasta de 192 KHz, que como es lógico no utilizaremos. Entre otros tiene 2 puertos MIDI, uno de IN y el otro de OUT.

Uniremos estos 2 puertos mediante un cable MIDI. Una vez realizado todo lo anterior solo queda seleccionar en el software Cubase cual va a ser la entrada y la salida de MIDI.

La entrada que seleccionaremos será la del puerto MIDI físico de la tarjeta. Para la salida será diferente, aquí tenemos un gran abanico de posibilidades, ya que entra en juego la sintetización de información.

Como hemos explicado, la sintetización la realizaremos mediante software. Para ello existen una infinidad de programas que emulan distintos instrumentos virtuales. Nosotros hemos seleccionado el software "BandStand", de Native Instruments. Este programa es capaz de sintetizar información MIDI con sonidos de gran cantidad de instrumentos y con una calidad más que aceptable.

Fig. 6.5 *- BandStand*

Por lo tanto la salida MIDI que seleccionaremos en Cubase será "BandStand".

Llegados a éste punto ya estamos listos para conectar la guitarra eléctrica al sistema, empezar a tocar y observar los resultados de la aplicación en tiempo real.

Aspectos a tener en cuenta

- ✓ Usar en la guitarra la pastilla de graves para mejorar la detección y evitar potenciar la energía de los armónicos.

- ✓ La guitarra no suele estar perfectamente afinada y quintada, lo que provoca pequeñas desviaciones frecuenciales en las notas tocadas propiciando algunos errores en la detección.

- ✓ Las notas tocadas con la guitarra sufren una pequeña modulación en frecuencia que influirá directamente en la detección.

6.2 Ajuste del nivel de entrada

Antes de nada es de vital importancia ajustar el nivel de entrada de la señal de audio. Para hacerlo conectaremos la guitarra a la entrada de la tarjeta de sonido y tocaremos las distintas cuerdas comprobando el resultado que se obtiene a la salida del conversor. Pueden ocurrir 2 cosas: que no se detecte nada, o que al tocar una cuerda suenen muchas notas. En el primer caso lo que está pasando es que el nivel de entrada es muy bajo, por lo que desde el panel de control de la tarjeta de audio subiremos en nivel de señal hasta que encontremos el adecuado. En el segundo caso haremos lo contrario, como el nivel de entrada es demasiado alto lo bajaremos hasta encontrar el correcto. Esto es debido a que el algoritmo de detección tiene unos valores de threshold fijados para la detección de los máximos de energía. Una vez hecho esto ya tendremos ajustado correctamente en nivel de entrada de nuestro instrumento. También se podría valorar la opción de desarrollar un acondicionador de la señal de entrada.

6.3 Metodología y pruebas

Para realizar las distintas pruebas ejecutaremos la aplicación diseñada y tocaremos con la guitarra distintas notas musicales. Con el propósito de analizar los resultados obtenidos de la aplicación en tiempo real necesitaremos grabar lo que se va generando mientras tocamos para después conseguir verlo y analizarlo. Para escuchar el MIDI mientras tocamos seleccionaremos un instrumento en Bandstand. En el proyecto se adjuntan las muestras de audio del MIDI sintetizado de las distintas pruebas.

Prueba 1 – Escala de C Mayor

La primera prueba que realizaremos será tocar con la guitarra la escala de Do Mayor en sentido ascendente y posteriormente descendente. En la figura 6.6 se pueden ver las notas que conforman la escala.

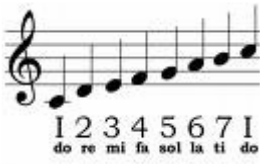


Fig. 6.6 *– Escala de Do mayor*

En las pruebas el tiempo de duración de cada nota será libre, ya que lo que nos interesa es ver si se genera adecuadamente la información MIDI a partir del audio en tiempo real, si se detectan bien las notas y si la información llega correctamente a través de los puertos.

La visualización de los resultados se hará mediante el editor MIDI de Cubase y con la trascripción musical del resultado.

Resultados obtenidos en prueba 1 - Escala de C Mayor

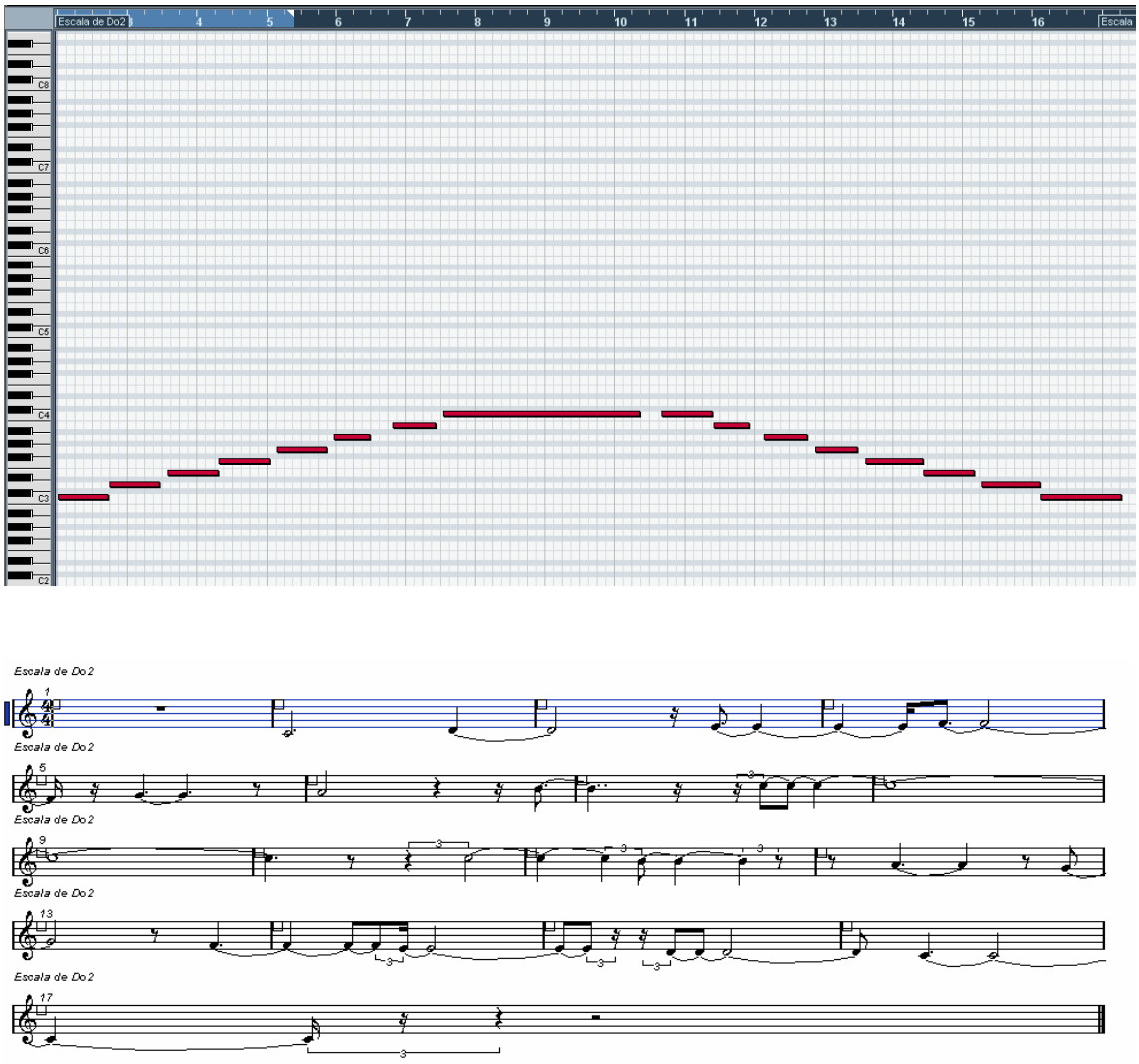


Fig. 6.7 *– Resultados obtenidos prueba 1*

Viendo las imágenes de los resultados, por una parte vemos que la información ha llegado correctamente a través de la aplicación y los puertos. En lo referente a la detección de las notas en tiempo real, se puede ver que los resultados han sido muy buenos, ya que se han detectado todas las notas correctamente. En la trascripción musical podemos ver que las notas detectadas pertenecen a la escala de Do Mayor.

Prueba 2 – Cuerda E cromática

La siguiente prueba la realizaremos tocando todos los semitonos que contiene la cuerda más aguda del instrumento (E). En éste caso el rango de notas que recorreremos será de 2 octavas. Tocaremos todos los semitonos cromáticamente en sentido ascendente.

Resultados obtenidos en prueba 2 – Cuerda E cromática

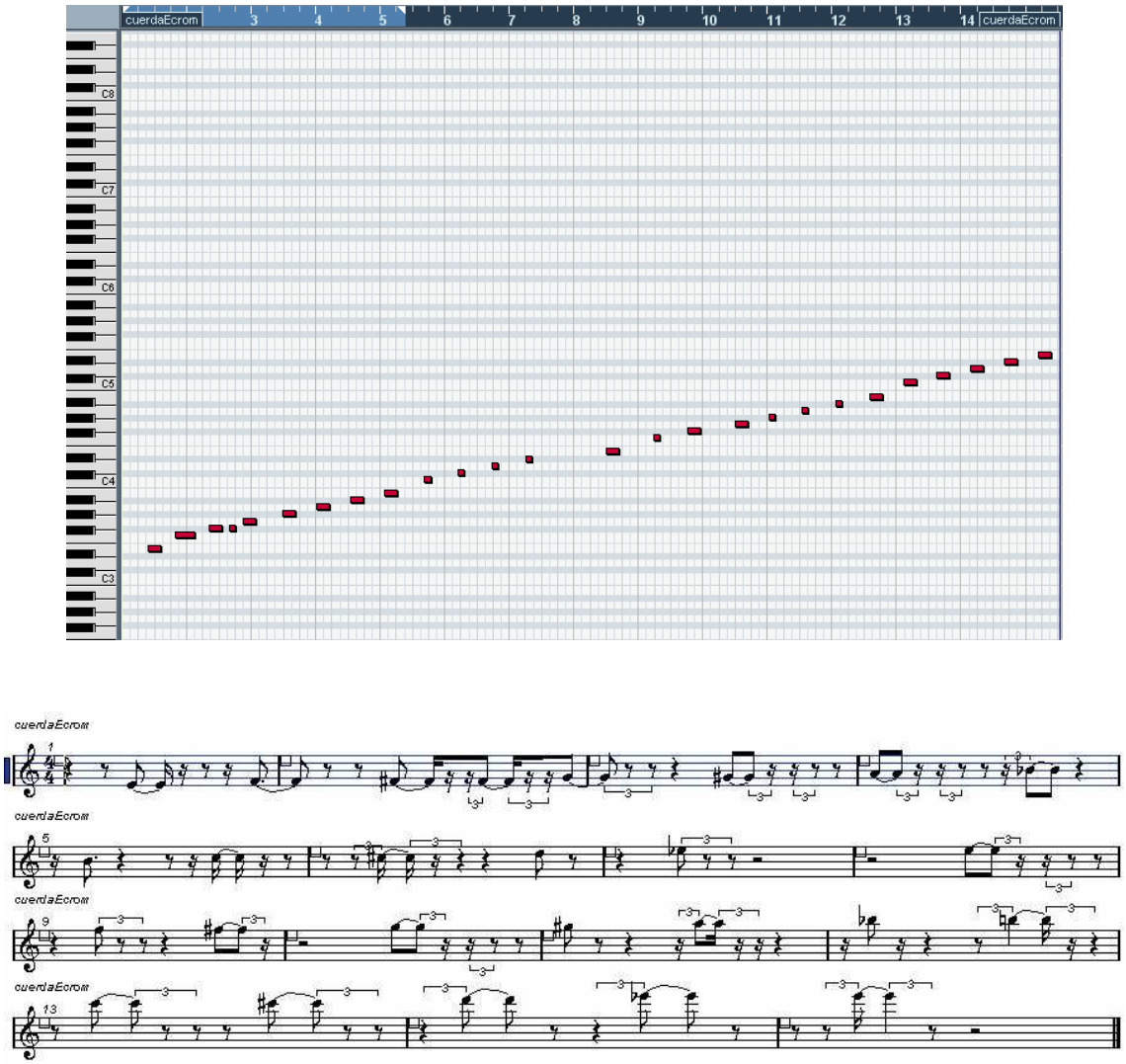


Fig. 6.8 *– Resultados obtenidos prueba 2*

Los resultados nos indican una perfecta detección para las notas de la cuerda más aguda, ya que aparecen todas correctamente. En la trascripción se puede apreciar el movimiento cromático ascendente.

Prueba 3 – Cuerda G cromática

Esta prueba es similar a la anterior, pero en éste caso la cuerda elegida es la tercera (G). Para probar la detección tocaremos 2 veces los 5 primeros trastes (G-G#-A-A#-B).

Resultados obtenidos en prueba 3 – Cuerda G cromática

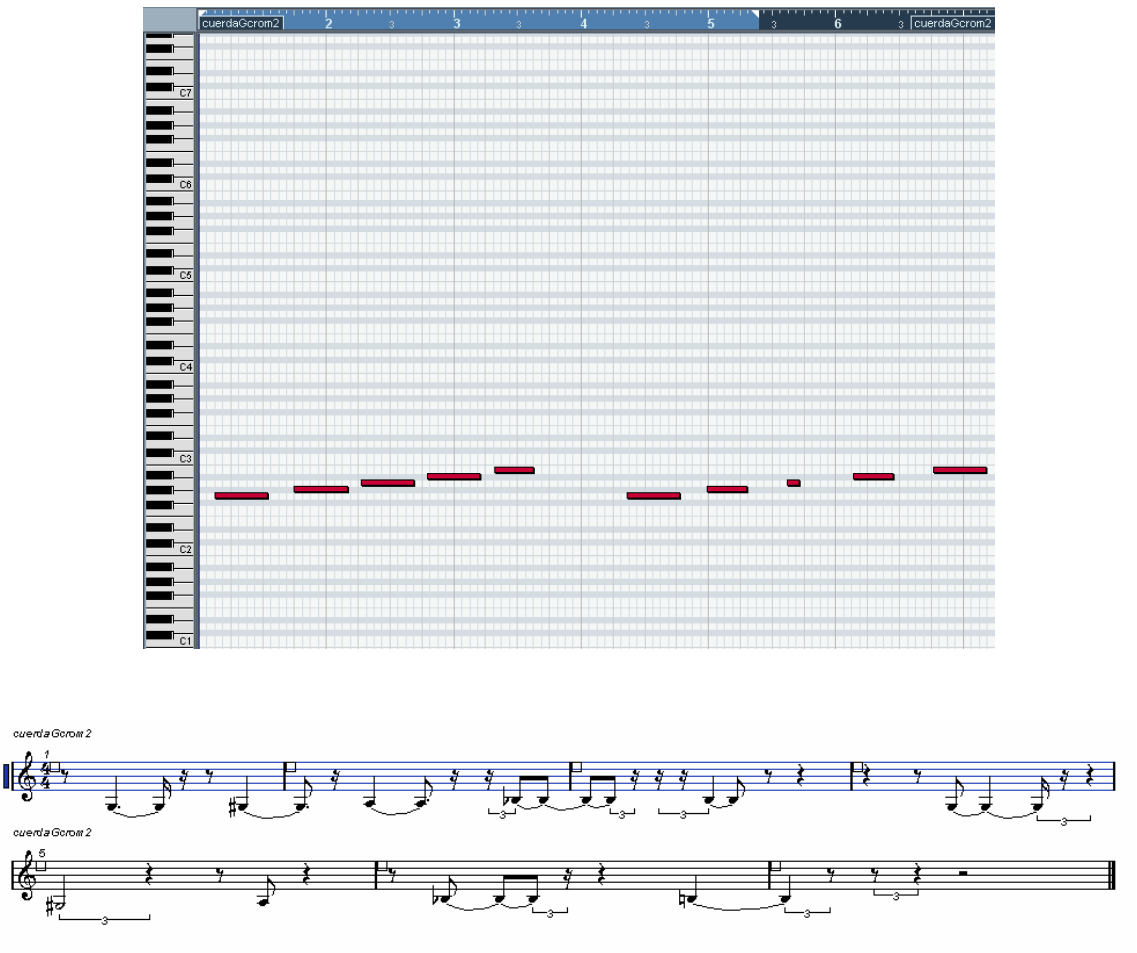


Fig. 6.9 *– Resultados obtenidos prueba 3*

Viendo las 2 figuras podemos ver que la detección ha funcionado correctamente.

Prueba 4 – Notas graves

La siguiente prueba se realiza sobre notas graves de la guitarra, que obviamente están en la cuerda 6 (E). Esta es la zona de más conflicto en la detección ya que al ser notas graves la distancia entre semitonos consecutivos medida en hercios es muy baja. Todo y esto recordemos que nuestro sistema está diseñado para que tenga una resolución frecuencial suficiente capaz de captar el cambio entre la nota más grave y su inmediatamente superior.

Para hacer la prueba tocaremos la nota más grave posible Mi (E) y el Sol (G). Veamos los resultados.

Resultados obtenidos en prueba 4 – Notas graves

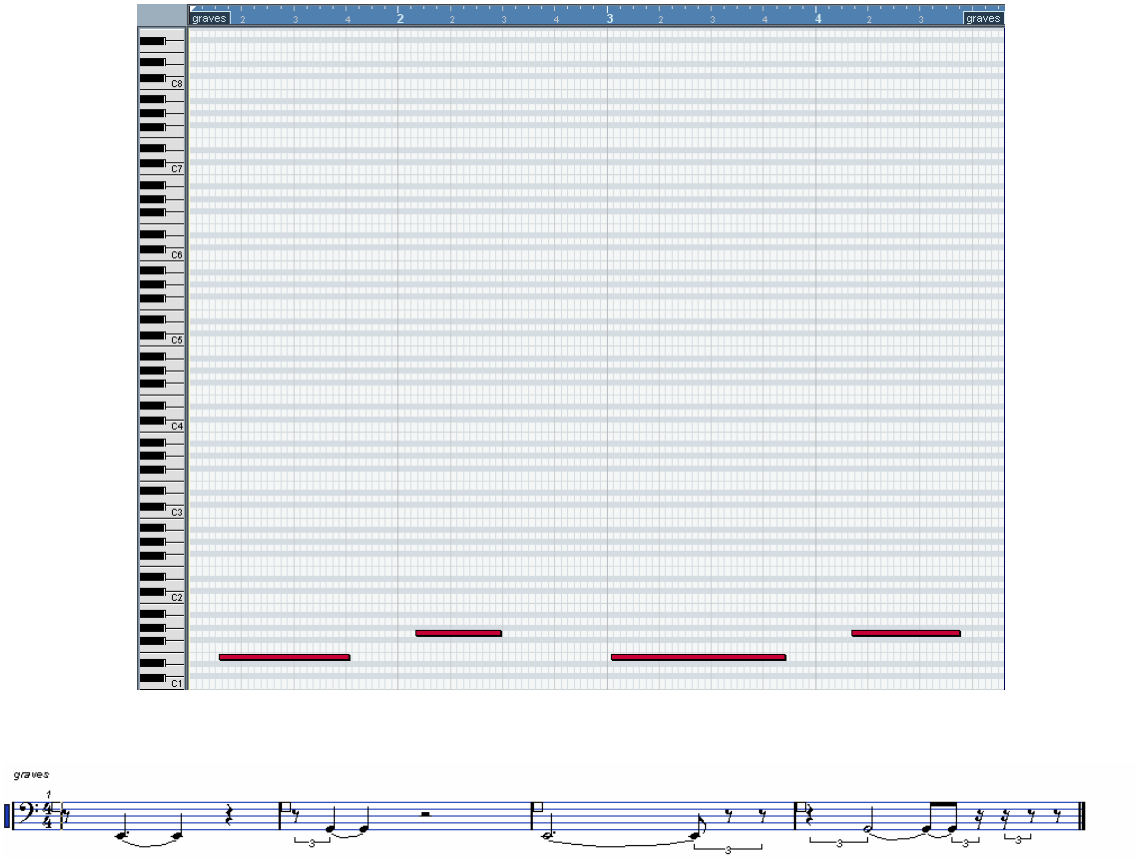


Fig. 6.10 *– Resultados obtenidos prueba 4*

Esta también es la zona más conflictiva de detección porque es donde aparecen los armónicos más enfatizados. Gracias al postprocesado monofónico estos son eliminados correctamente. Podemos ver que los resultados muestran una buena detección de las notas graves.

Prueba 5 – Arpegio de C

La última prueba la realizaremos sobre un arpegio. Un arpegio es la sucesión de unos determinados intervalos. En nuestro caso elegiremos el arpegio de Do Mayor, formado por C – E – G. Lo tocaremos con la guitarra en sentido ascendente y después descendente.

Resultados obtenidos en prueba 5 – Arpegio de C

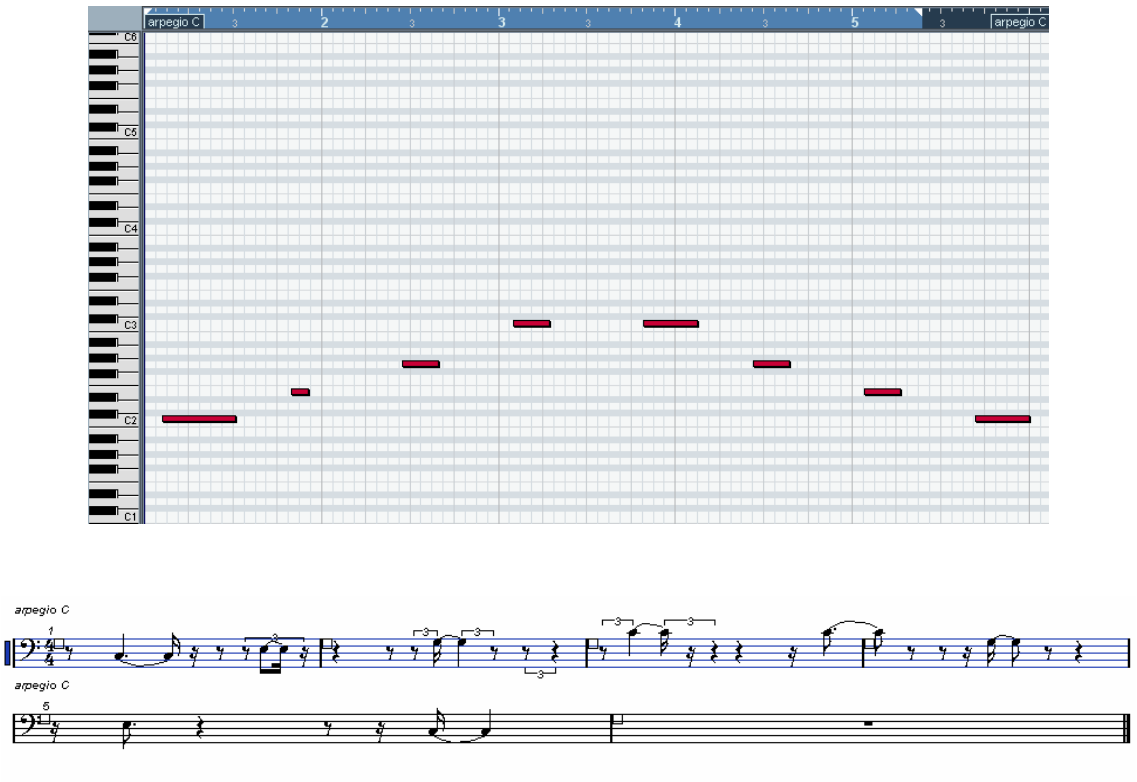


Fig. 6.11 *– Resultados obtenidos prueba 5*

Vemos que las notas del arpegio se han detectado correctamente.

Hay que remarcar que en la detección de duración de notas pueden aparecer problemas debidos a que la guitarra no tiene una fase de sustain marcada, lo que implica que la duración de la nota detectada solo se dé durante la fase de máxima energía, desestimando la cola decadente de la nota tocada.

7. Conclusiones

Como hemos visto en el presente proyecto, hemos desarrollado un conversor de audio a MIDI con funcionamiento en tiempo real monofónico orientado a guitarras eléctricas. Partiendo de una señal entrante de audio generada por una guitarra, a la salida vamos obteniendo la conversión de ésta a formato MIDI, obteniendo así la trascripción de la interpretación instrumental.

Analizando las pruebas y valorando los resultados obtenidos podemos afirmar que estos son bastantes satisfactorios, sobretodo en el enfoque principal, que era implementar la detección de notas en tiempo real y mejorarla. Además éste sistema goza de gran compatibilidad con otros sistemas operativos y componentes hardware. Los objetivos fijados se han conseguido cumplir satisfactoriamente.

Este sistema puede ser muy útil de cara a músicos, ya que les permite utilizar su instrumento convencional como controlador MIDI, sintiéndose totalmente cómodos durante su interpretación a diferencia de cuando utilizaban un controlador convencional. Se abre el campo de la creatividad y la experimentación musical fusionando tecnología y música.

Posibles Mejoras

El sistema implementado deja abierta la posibilidad de realizar nuevas mejoras. Como comentamos en la introducción, el conversor desarrollado es monofónico, por lo que solo soporta la detección de una nota a la vez. Esto es debido a que al tocar varias notas o acordes las técnicas de análisis espectrales se vuelven más ineficientes a causa de la mezcla de todos los armónicos de cada una de las notas. En el mercado actual se han desarrollado unas pastillas especiales para conversión MIDI las cuales detectan la nota tocada en cada una de las 6 cuerdas de forma independiente. Como mejora se podría utilizar éste tipo de pastillas y adaptar el programa desarrollado en éste proyecto para que realizara una detección polifónica.

También se podría implementar la detección de velocidad MIDI o lo que es lo mismo, las dinámicas en la interpretación de guitarra. Posiblemente se podría realizar mediante el estudio y seguimiento de la energía de los armónicos detectados.

Como también hemos comentado en el proyecto, se puede considerar la opción de crear un acondicionador de señal con la finalidad de adaptar cualquier tipo de señal entrante al funcionamiento del conversor.

Por último otra opción sería mejorar la detección de duración de notas. Como comentamos en el proyecto, el sonido producido por una guitarra no tiene fase de sustain. Esto provoca que al detectar la duración de una nota solo cojamos el intervalo de tiempo en el que se supera el máximo de energía establecido (threshold) para esa cuerda, normalmente asociado al golpe con la púa, descartando la fase de decaimiento en la cual la guitarra sigue sonando. Para ello se tendría que estudiar la evolución temporal de la función ADSR de la guitarra y desarrollar un algoritmo que consiga una perfecta detección completa de la nota.

8. Bibliografía y Referencias

- **DSP first a multimedia approach.** James H. McClellan, Ronald W. Schafer, Mark A. Yoder.

- **A Digital signal processing primer with applications to digital audio and computer music.** Ken Steiglitz.

- **Real Time Speech Classification and Pitch Detection**. Jonathan A. Marks. International Digital Corporation. 2000.

- **Time-frequency Analysis of Musical Signals**. William J. Pielemeier, Gregory H. Wakefield & Mary H. Simoni. IEEE.

- **Audio Digital y MIDI.** Sergi Jordà Puig. Madrid Anaya Multimedia cop.1997.

- **Cursos de programación en C/C++**

    - **Curso de programación en C** por Miquel A Garcies
    - **Manual de programación en C++** de servicios informáticos UCM
    - **Tutorial Programación en Visual C++** de la escuela superior de ingenieros de la universidad de Navarra.

- **Documentación PortMusic**

    - http://www.cs.cmu.edu/~music/portmusic

- **Documentación PortAudio**

    - http://www.portaudio.com/

- **González, Pablo Busto. Convertidor de audio a MIDI**

    - http://www.pandreonline.com/p_busto/ (12/12/2006)

- **PFC Alberto Molina Reverte**. Junio 2006.

- **Apuntes de las asignaturas:**

    - Procesado de Voz y Audio (PVA)
    - Algorítmica y Programación

Anexos

Índice de Figuras

Muestras Audio Sintetizado

Con el presente proyecto se adjunta una carpeta que contiene las muestras de audio resultantes de sintetizar las pruebas MIDI realizadas en el proyecto mediante Bandstand.

sint.escalaDo.sitar.wav → Sintetización de la prueba 1 con un sitar

sint.Ecrom.banjo.wav → Sintetización de la prueba 2 con un banjo

sint.Gcrom.violin.wav → Sintetización de la prueba 3 con un violín

sint.graves.bajoacustico.wav → Sintetización de la prueba 4 con un bajo acústico

sint.arpegioC.organo.wav → Sintetización de la prueba 5 con un órgano