Conversor WAV a MIDI de muestras monofónicas

Matías A. Larroque, Lucero G. Fernández, Manuel F. Mollon, Tomas G. Orlando y Ezequiel Vijande

***Abstract*— En el trabajo se implementaron varios métodos de detección de tono, métodos de obtención de tempo y método de obtención de onset-offset de audios (formato .wav) con el fin de procesar cada resultado individual y crear un archivo MIDI a partir del audio original. La conversión es de un solo musical de cualquier instrumento compuesto de solo notas individuales, es decir una muestra monofónica. De los varios métodos implementados se analizó su eficiencia y se presentaron mejoras posibles a realizar en el futuro.**

***Index Terms*—Procesamiento de señales, audio, tono, tempo, separación espectral.**

I. INTRODUCCION

EL esquema elemental en el cual se basa este trabajo se puede ver en la Fig. 1. A lo largo del trabajo se explica en detalle la implementación de los diferentes algoritmos desarrollados para cada bloque, así como su eficiencia. Tras un trabajo de investigación y desarrollo de estos algoritmos, se implementaron en un programa principal los más eficientes con el fin de realizar con la menor cantidad de error la conversión WAV a MIDI.

La primera etapa corresponde a la detección del onset y offset de las notas musicales (Onset/Offset Detection). El onset se determina al inicio de una nota musical y el offset se refiere al fin de esta. Los algoritmos implementados detectan el inicio y el final de las notas con el fin de usar esa información para separar notas individuales que luego son procesadas por el detector de tonos (referenciado como pitch en adelante) y también para procesar el inicio y fin de las notas en el archivo MIDI.

La segunda etapa, en paralelo con la primera, es la detección del tempo en el audio (Tempo Detection). En esta etapa se calcula el tempo del audio medido en bpm (beats per minute) para luego usar esta información para la creación del MIDI.

De la primera etapa se usa la separación de notas para aplicar el algoritmo de detección de pitch como se mencionó anteriormente. Este pitch de cada nota es usado luego para la conversión a MIDI.

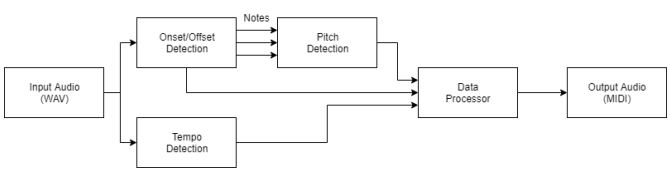


Fig. 1. Diagrama en bloques del programa principal realizado.

II. SEGMENTACIÓN TEMPORAL DE NOTAS MUSICALES

A. *Concepto de Segmentación – Onset – Offset*

La Segmentación temporal en las notas musicales hace referencia a la detección de intervalos temporales en donde se manifiesta una señal de audio con un pitch característico, es decir, una nota musical. La Segmentación conlleva dos eventos los cuales son el "onset" y "offset".

El evento de onset es el cual determina el comienzo de la nota musical. Este puede se manifiesta mediante un cambio abrupto en la energía de la señal de audio, o lo que también se conoce como "attack".

Por otro lado, el offset es el evento que determina la finalización de la nota. El hecho de que la nota musical finalice es análogo a decir que la energía de la señal de audio no es distinguible de la energía del ruido del ambiente sonoro.

Habiendo aclarado el significado de estos conceptos se procederá a continuación a presentar un esquema que se puede utilizar para llevar a cabo la segmentación, para luego abordar la realización de lo propuesto en dicho esquema.

B. *Etapas de la Segmentación*

Si es posible detectar tanto el evento de onset como el de offset, entonces se tendrá el intervalo temporal en el cual se manifiesta la nota musical, ya que se tendrá el comienzo y el final de este. Entonces como primera medida se proceden a hallar estos dos eventos.

En cuanto al evento de offset, este será después del evento de onset. Suponiendo que se tiene el onset, de manera sencilla se puede detectar el offset identificando el momento para el cual la señal de audio es comparable con la señal de ruido (mediante un "threshold" de offset, por ejemplo), comenzando la búsqueda de este instante a partir del evento de onset.

Dicho esto, se puede considerar a la segmentación como principalmente un problema de detección de onset. Esto se abordará mediante el procedimiento empleado por la mayoría de los algoritmos de detección de este tipo de eventos.

En la Fig. 2 se ilustra el diagrama para abordar el problema mencionado, el cual consta de la generación de una señal de detección (también llamada señal de novedad) y la posterior detección de los picos de esta. El instante (temporal) de detección de los picos es también el instante de onset.

Además, para generar la función de novedad mencionada, podría procesarse la entrada original de audio preparándola para generar una función de novedad que podría ser más efectiva ("preprocesamiento") o también puede procesada la

señal de detección de manera que se logre mayor eficiencia en la detección de picos ("postprocesamiento").

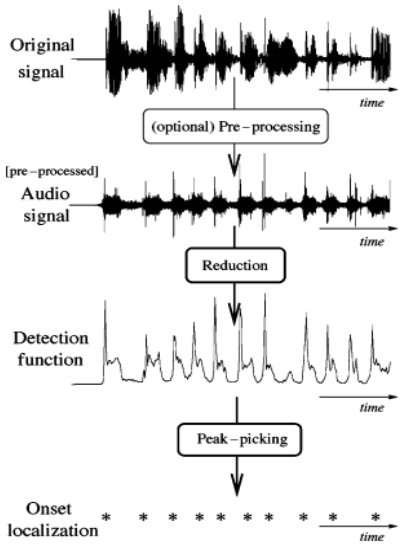


Fig. 2. Flowchart of a standard onset detection algorithm.

Fig. 2. Diagrama simple de algoritmo de detección.

C. *Función de Detección*

El algoritmo de generación de la función de detección es determinante para la detección de offset. Un algoritmo efectivo tanto para señales de audio armónicas e inarmónicas (las señales emitidas por instrumentos de percusión, por ejemplo, son inarmónicas) es introducido en (Masri 1996) construye una función de novedad de alto contenido armónico ("High Frequency Content" o "HFC") sumando valores linealmente ponderados de las magnitudes espectrales de la siguiente manera:

$$D_H(n) = \sum_{k=0}^{N} k|X_k(n)|$$

En la anterior ecuación la función ponderada en módulo es la "Short Time Fourier Transform" (STFT). La función de detección enfatiza las zonas de la señal que presenta alto contenido armónico, cambiando bruscamente en el tiempo.
Un algoritmo tratado en (Bello, Duxbury, Davies y Sandler, 2004) que se concentra más en las variaciones armónicas es el de "diferencia espectral" ("Spectral Diference" o "SD") y obtiene la función de detección de la siguiente manera:

$$D_S(n) = \frac{1}{N}\sum_{k=0}^{N}\sqrt{|\frac{dX_k(n)}{dn}|}$$

En la ecuación, la derivada de la STFT con respecto a "n" representa la diferencia entre bines de la magnitud espectral de las muestras sucesivas.
A esta focalización en la diferencial espectral puede incorporarse un análisis tratado en [1] sobre el cambio de fase de la señal de audio. Combinando estos dos aspectos, de cambio espectral en magnitud y de fase se logra la siguiente función de detección:

$$D_C(n) = \sum_{k=0}^{N} ||X_k(n)^a - X_k(n)||^2$$

$$X_k(n)^a = |X_k(n)|e^{j\emptyset_k^a(n)}$$

$$\emptyset_k^a(n) = map(\frac{\partial^2 \emptyset_k(n)}{\partial n^2})$$

En donde "map" es una función que traduce la fase en un ángulo dentro del intervalo $[-\pi; \pi]$.
Esta última función de detección, según resultados mostrados en [1], funciona exitosamente para señales de audio armónico, pero no funciona para señales de percusión (inarmónicas).

D. *Detección de Picos*

Una vez hallada la función de detección o novedad, se proceden a identificar los picos de esta, teniendo en cuenta los picos lo suficientemente pronunciados y que superen un "threshold" (umbral) fijo. Una opción más eficaz es implementar una función que genere un umbral adaptativo, es decir que tome un valor de umbral para cada muestra de la función de detección, basándose en muestras del entorno para hallar este umbral óptimo. Estos algoritmos se pueden implementar computacionalmente con librerías nativas de algunos lenguajes de programación como lo son "Python" o "JavaScript".

E. *Offset*

Como se ha mencionado en la sección B, la detección de onset puede realizarse de manera relativamente sencilla una vez que se ha detectado el onset, comparando la función original con un umbral que se corresponda con el ruido ambiental y así decidir el instante para el cual se da la nota musical por finalizada (instante de detección de offset). En efecto para obtener los resultados que se mostrarán más adelante, se utilizó un umbral de offset teniendo en cuenta que ante una señal sonora normalmente existe un ruido ambiental que se corresponde en -80dB.
Debido a que podría ocurrir que se aplique el algoritmo de segmentación en una señal de audio que presente una cantidad numerosa de notas (esto es lo que sucede en la mayoría de los casos para el cual este algoritmo es de interés), luego podría existir un nuevo onset antes de la detección del offset correspondiente el anterior onset. Ante este caso, se decidió determinar el offset mediante el nuevo onset., es decir que el evento de offset y el nuevo onset coinciden para este caso. Esto quiere decir que la finalización de la nota la determina el comienzo de la siguiente nota, siempre que esta última se manifieste lo suficientemente rápido tal que no se detecte offset para la nota anterior.

F. *Implementación y Resultados de la Segmentación*

Para realizar la segmentación se ha utilizado el algoritmo de HFC para obtener la función de detección y a esta se la proceso aplicando una decimación de orden 4 para luego realizar una interpolación del mismo orden, logrando así suavizar la función de detección y evitar que se detecten picos que no correspondan a onsets.
Una vez realizado esto, se detectaron los picos y se utilizaron estos mismos para obtener los offsets.

Esta segmentación se ha aplicado a una señal de audio con cuatro notas y se obtuvieron los resultados de la Fig. 3. En dicha figura, se pueden observar los onsets detectados como cruces ‘x’ y los offsets como círculos ‘o’.

III. Detección de Pitch

A. *Definición*

El tono (pitch) de una nota musical se refiere a la frecuencia percibida por el oído al escucharla. Esta misma coincide con la separación entre armónicos, la cual es la misma para todas.

Por lo que, si se computa la FFT, la frecuencia fundamental puede no estar presente, ya que, si los armónicos están espaciados una frecuencia f, esta será la fundamental y la percibida, sin que esta se encuentre presente en la señal. Esto trae problemas a la hora de encontrar la frecuencia fundamental, ya que esta puede no estar presente, y si lo está, no necesariamente es el máximo pico. Ya que hay instrumentos donde los armónicos son más potentes que la fundamental en el sentido de densidad espectral. Por lo que para hacer una detección correcta se necesitan algoritmos más sofisticados. Algunos de estos fueron implementados y son explicados a continuación. Luego hay un análisis de resultados a partir de la implementación de estos en notas individuales de varios instrumentos.

B. *Algoritmo de Autocorrelación*

El algoritmo se basa en la función de autocorrelación de la nota en cuestión. La autocorrelación es definida de la siguiente manera:

$$R_x(m) = \sum_{n=0}^{N-1-m} x(n)x(n+m)$$

Se puede probar que, si la función original es periódica de periodo tau, la autocorrelación también es periódica del mismo periodo. Se puede observar también como la autocorrelación debe dar un máximo en el corrimiento $m = 0$, ya que es la comparación de dos señales idénticas sin desfasaje. Por lo que, si se encuentra el primer máximo de la función tras el origen, se encuentra el periodo de la función original, ya que se cumple:

$$R_x(0) = R_x(\tau)$$

Como la señal no es completamente periódica, encontrar el primer máximo no es sencillo ya que puede haber falsos picos que se aproximan al verdadero. Mas adelante en resultados se puede ver como este algoritmo falla en algunas ocasiones.

Para mejorar este algoritmo, se introduce un preprocesamiento de la señal, con el fin de reducir estos picos falsos y tener mayor planicie fuera de la región del máximo que se requiere. Este preprocesamiento introducido por [2] se puede ver en el diagrama presentado en Fig. 4.

Donde $H[x](n)$ es una transformación lineal dada por:

$$H[x](n) = \begin{cases} 0, & x(n) < Cl \\ 1, & x(n) \geq Cl \end{cases}$$

Donde Cl es un threshold seteado experimentalmente. En el artículo referenciado, se explica que se encontró que el mejor parámetro es tomar el mínimo entre los máximos absolutos del primer y último tercio del audio. Este máximo después es multiplicado por 0.68. Este valor es el utilizado como Cl para

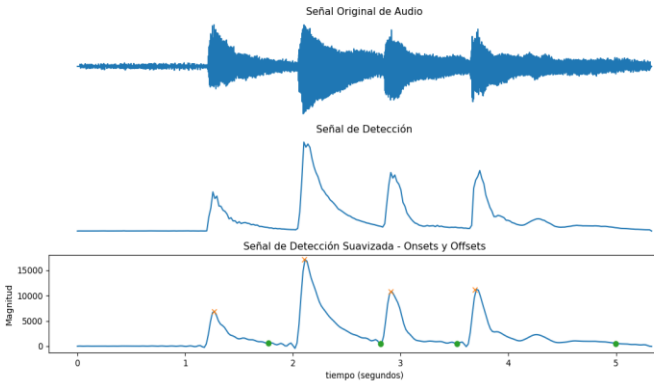


Fig. 3. Magnetization as a function of applied field. Note that “Fig.” is abbreviated. There is a period after the figure number, followed by two spaces. It is good practice to explain the significance of the figure in the caption.

ese audio en cuestión. Luego de aplicar esta transformación y calcular la convolución, vuelve a buscarse el primer máximo, tarea simplificada por estas transformaciones. Como se puede ver en resultados, los picos falsos desaparecen y el algoritmo es mucho menos susceptible a errores de búsqueda del máximo verdadero. Este algoritmo resulto ser muy rápido y eficiente, ya que la convolución fue implementada con FFT. Al igual que todos los algoritmos descriptos en esta sección, falla para muestras de audio polifónicas, es decir, solo funciona para notas individuales y no acordes.

C. *Algoritmo Harmonic Product Spectrum*

Este algoritmo, a diferencia del primero, no busca en el tiempo, sino que en frecuencia a través de la FFT.

El algoritmo se puede apreciar fácilmente en la Fig. 5. Consiste en generar la FFT de la señal original y de otras FFT de la señal downsampleada por un factor desde 2 a n, siendo n el número de armónicos para tener en cuenta. Al hacer el downsample, si el fundamental tiene armónicos, estos quedan

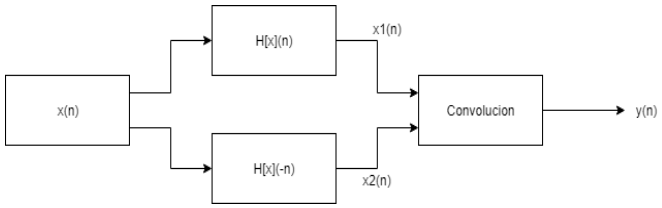


Fig. 4. Diagrama de preprocesamiento de la señal.

alineados en la misma frecuencia. Luego de calcular todas las FFT, estas se multiplican entre sí. En la frecuencia fundamental, se multiplicarán todos los armónicos generando un pico mayor al resto. Esta frecuencia donde el pico es mayor será la frecuencia fundamental, por lo que será el pitch de este. Con este algoritmo en ocasiones se puede producir un error de una octava. Un criterio introducido en [3] es: Si la amplitud del segundo pico por debajo del elegido es de la mitad de la frecuencia del elegido, y la ratio de las amplitudes supera cierto threshold, entonces de elije como frecuencia fundamental del

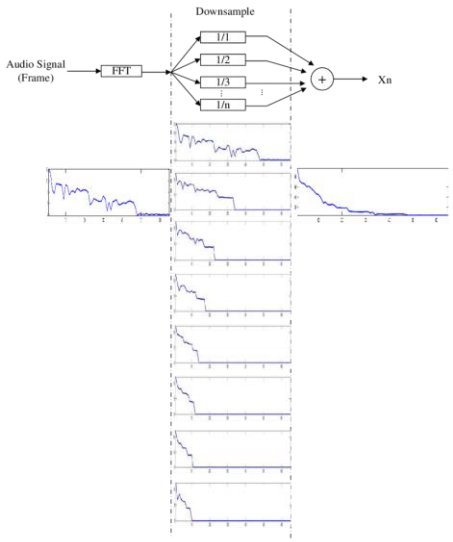


Fig. 5. Funcionamiento del algoritmo HPS

pico anterior al elegido. Según [3], para una implementación con 5 armónicos, el threshold es de 0.2.

El algoritmo resulto ser muy rápido y eficiente para todas las muestras testeadas, pero tuvo la desventaja ante la autocorrelación de necesitar un mayor número de muestras mínimas para estimar el pitch sin error.

D. Algoritmo YIN

El ultimo algoritmo implementado es el algoritmo YIN, propuesto en [4]. El mismo se basa en la ecuación fundamental:

$$x(n) = x(n+\tau)$$

Donde se toma la diferencia, se eleva al cuadrado y se introduce la sumatoria:

$$d_\tau(\tau) = \sum (x(n) - x(n+\tau))^2 = 0$$

Por lo que se computa esta función y se busca el primer cero de esta. La cual corresponde a el periodo de la frecuencia fundamental. La implementación en código de esta función es lenta, por lo que expandiendo la diferencia al cuadrado se puede usar propiedades de la convolución y de FFT para poder hacer una implementación más rápida:

$$d_\tau(\tau) = r_t(0) + r_{t+\tau}(0) - 2r_t(\tau)$$

Esta búsqueda puede ser difícil y poco precisa, por lo que se define una nueva función basada en esta:

$$d'_t(\tau) = \begin{cases} 1, & \tau = 0 \\ \dfrac{d_t(\tau)}{\frac{1}{\tau}\sum_{j=1}^{\tau} d_t(j)}, & \tau \neq 0 \end{cases}$$

Con esta nueva función, el pico correspondiente a la frecuencia fundamental queda mejor definido. Para encontrarlo, se elige el primer mínimo cercano a cero, debajo de un threshold.

E. Implementación en un audio (WAV)

Tras implementar los algoritmos, se grabó una secuencia de varias notas y se corrió el algoritmo separando el audio en ventanas de un largo wLen y un overlap de wOverlap. En cada ventana se aplicó el algoritmo y se guardó el pitch detectado. Luego de grafico esos pitches detectados conjunto con la grabación. Primero se aplicó el algoritmo sin criterio alguno, obteniendo la primera imagen en Fig. 6. Teniendo en cuenta que para el tamaño de la ventana no puede haber un solo pitch aislado, es decir, no puede una nota sonar tan poco tiempo, estas se cambiaban teniendo en cuenta los valores próximos.

También se elegio un threshold por el cual, a menor amplitud, el audio era considerado ruido y no una nota. Tras aplicar estos criterios se obtuvo la segunda imagen en Fig. 6 con el método de autocorrelación preprocesada. Al usar el método de HPS, como el tamaño de ventana era muy chico ara este algoritmo, se obtuvieron varios errores observables en la tercera imagen de la Fig. 6. Estos se arreglaron agrandando el ancho de la ventana, lo que resulta en la cuarta imagen de la Fig. 6. El algoritmo YIN tuvo el mejor desempeño (ultima imagen de Fig. 6), pero a su vez fue el que más tiempo llevo al no implementar la sumatoria con FFT. Si la sumatoria se implementa con FFT, el algoritmo YIN lleva menos tiempo de procesamiento y es el más confiable.

Para mejorar el rendimiento en el programa principal, se aprovechó el uso del onset y offset. Ya que al tener cuando empieza y termina una nota, en vez de usar el algoritmo explicado anteriormente, solo se llama a la función de detección de pitch una vez por nota. Esto ayuda a eliminar errores provocados por outliers, tener menos tiempo de procesamiento al llamarse menos veces la función y permite poder detectar dos notas consecutivas del mismo pitch.

F. Resultados

Se creo un banco de pruebas con once instrumentos diferentes para probar los diferentes algoritmos implementados. Como se puede ver en la Tabla 1, el algoritmo basado en la autocorrelación tuvo casos donde hay un error entre la nota real y calculada. Estos errores se arreglaron al introducir el preprocesamiento anteriormente explicado. Se puede ver esto en las Fig. 8 del apéndice. En la Fig. 9 puede verse el pico máximo en el algoritmo HPS, superior a los otros picos presentes. La función usada para la detección de pitch en

TABLA I
RESULTADOS DE IMPLEMENTACIÓN EN NOTAS (NUMERO MIDI)

Instrumento (Nota)	Real	AC	AC preprocesada	HPS	YIN
Saxo (B3)	59.0	59.0	59.0	59.0	59.0
Guitarra (G3)	55.0	55.0	55.0	55.0	55.0
Trompeta (A4)	69.0	69.0	69.0	69.0	69.0
Violín (A4)	69.0	69.0	69.0	69.0	69.0
Cello (A#2)	46.0	46.0	46.0	46.0	46.0
Flauta (G5)	79.0	79.0	79.0	79.0	79.0
Oboe (E5)	76.0	88.0	76.0	76.0	76.0
Banjo (E6)	88.0	88.0	88.0	88.0	88.0
Clarinete (F5)	77.0	85.0	77.0	77.0	77.0
Tuba (A1)	33.0	64.0	33.0	33.0	33.0
Piano (F#4)	66.0	65.0	66.0		
Acorde Piano (C4-E4-G4)	60.0	51.0	24.0	36.0	150.0

Valores en número MIDI de la nota, determinados redondeando al entero el resultado de la ecuación: $12\log_2\left(\frac{f}{440}\right) - 69$

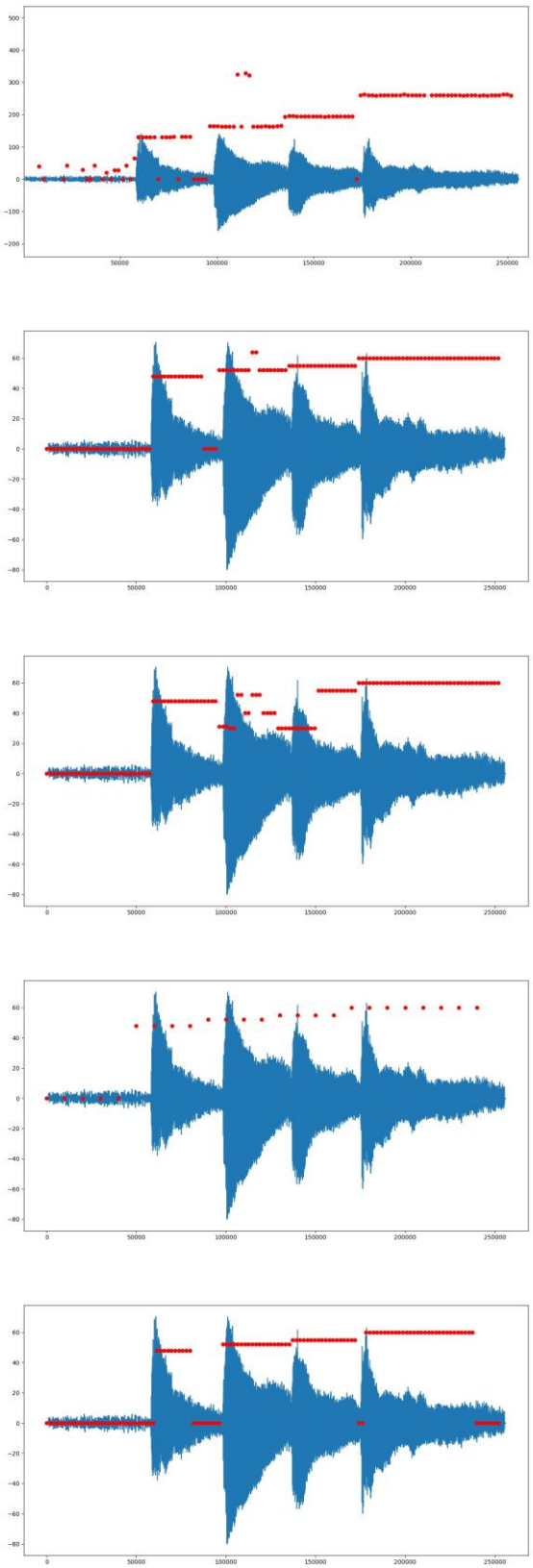


Fig. 6. Implementación en WAV de algoritmos de detección de pitch. Donde la función azul corresponde al audio introducido y los puntos rojos son los tonos detectados en cada bin, medidos en número MIDI.

el algoritmo YIN se ve en la Fig. 10. Ningún algoritmo tuvo éxito en la muestra polifónica, como era de esperarse.

IV. Detección de tempo

A. *Definición de tempo*

El tempo de una canción suele dar una idea del ritmo de una canción. Por lo general, las canciones tienen un pulso subyacente de una frecuencia específica, este pulso se denomina 'beat'. Para un oyente humano el hecho de identificar el beat de una canción es simple, normalmente se acompaña dicho beat con el pie. Sin embargo, es un problema difícil de definir para ser resuelto de manera automática por una computadora. El tempo de una canción puede medirse en BPM(beats per minute) que se define como la cantidad de veces que se repite este pulso subyacente en un intervalo de tiempo de un minuto.

Se estableció como objetivo la implementación de un algoritmo que pueda determinar el BPM de un archivo de música wav dentro de un rango tolerable de error(±4BPM) y ue dicho algoritmo funcione para el rango de $60 \leq BPM \leq 180$.

B. *Algoritmo*

Se implementó un algoritmo de detección de BPM simple basado en los métodos implementados en [5] y [6].

1) *Segmentación*

El algoritmo puede ser implementado en tiempo real por lo que se ejecuta cada vez que se obtienen un numero M de muestras de audio. Para nuestra implementación utilizamos por default M=375.

Estas M muestras de audio se denominará bloque y funciona como uno de los parámetros del algoritmo.

2) *Señal de eventos rítmicos*

Para cada uno de estos bloques de M muestras, se calcula un único valor de potencia que se almacena en memoria. La potencia del bloque se calcula según la siguiente formula:

$$P(n) = \alpha P(n-1) + (1-\alpha).(\frac{1}{M}\sum_{i=0}^{M-1} x(i)^2)^2$$

Donde α es un parámetro definido en el intervalo 0< α< 1. M es el valor de la cantidad de muestras tomadas por bloque, y los elementos dentro de la sumatoria son los cuadrados de las muestras de audio del bloque actual. Para valores más grandes de α se tiene que la señal se ve menos afectada ante nuevos valores de las muestras.

Los valores de potencia obtenidos dan una idea de cómo varia la energía de la señal con el tiempo. Se tienen picos de amplitud muy grandes en los instantes de tiempo en los que hay eventos rítmicos importantes (generalmente cerca de donde ocurre un beat).

El arreglo donde se almacenan los valores de potencia tiene un límite máximo de N valores. Cuándo se excede este máximo se descarta el valor más viejo y se desplazan todos los valores para hacer lugar al valor más reciente.

3) Calculo del espectro

Una vez que se calculó el valor de potencia se procede a calcular el espectro de la señal mediante la aplicación de la FFT (Fast Fourier Transform) a los N valores de potencia. El espectro obtenido tiene una frecuencia de muestreo distinta al de la señal de audio original debido a que se tiene un solo valor de potencia por cada bloque analizado. Por lo que la nueva frecuencia de muestreo obtenida es:

$$f_s = \frac{F_{s\,audio}}{M}$$

Luego se tiene que la resolución del espectro calculado es:

$$\delta_f = \frac{f_s}{N}$$

4) Calculo y estimación del tempo.

Existe la siguiente relación matemática entre la frecuencia y el BPM:

$$BPM = 60.f$$

Debido a que solo nos interesa el rango 60 ≤ BPM ≤ 180. Solo son de interés los bins de frecuencia correspondientes a ese rango de BPM. Los bins correspondientes a los limites se obtienen mediante la relación:

$$bin_{min} = \left\lceil \frac{1Hz}{\delta_f} \right\rceil$$

$$bin_{max} = \left\lceil \frac{3Hz}{\delta_f} \right\rceil$$

Donde se redondea el bin al entero más próximo. Se busca el bin dentro del rango definido para el cual la señal toma su máximo valor y se estima que su BPM asociado es el BPM del bloque.

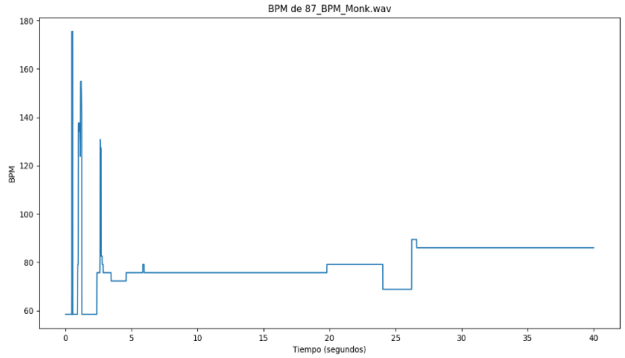


Fig. 10. Grafica del BPM de 'Dinah' de Thelonius Monk *(α=0.85, M=375, N=2048, $F_{s\,audio}$=44.1 KHz)*

C. Implementación y resultados

El algoritmo fue implementado en python utilizando dos canciones distintas y variando los parámetros a fin de ver el impacto en el resultado final. Para el cálculo del promedio $(\frac{1}{M}\sum_{i=0}^{M-1} x(i)^2)$ se utilizó la función 'fftconvolve' del paquete scipy. Para el cálculo del espectro, se utilizó la función 'rfft' del paquete numpy ya que el arreglo con valores de potencia siempre tiene valores reales y 'rfft' es computacionalmente más rápido que 'fft'.

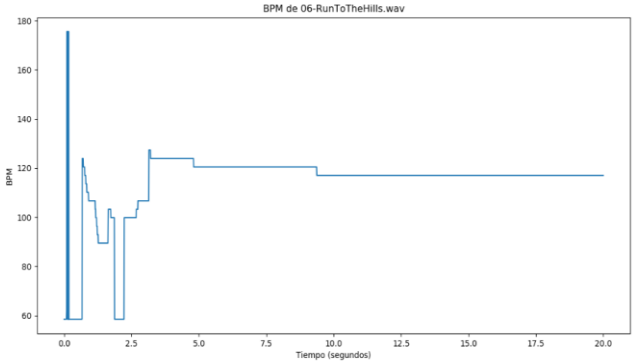


Fig. 11. Grafica de BPM de los primeros 20 segundos de 'Run To The Hills' de Iron Maiden (α=0.85, M=375, N=2048, $F_{s\,audio}$=44.1 KHz)

Como puede verse en 'Fig. 11.' el algoritmo obtiene un valor de BPM constante luego de un transitorio, en este caso los primeros 10 segundos de la canción. El valor de BPM en el que se establece es de 117 BPM lo que coincide con el BPM real de la canción.

El transitorio se debe al cálculo del espectro y también al de la potencia. El cálculo de potencia se basa en los valores calculados previamente, mientras mayor es el parámetro α mayor es la dependencia en los valores previos. Asimismo, el espectro se calcula siempre con N muestras de potencia, en el caso de la imagen para tener N muestras de potencia se tiene que:

$$Muestras\ de\ audio = N * M = 2048 * 375 = 768000$$

$$Tiempo\ requerido = \frac{Muestras\ de\ audio}{F_{s\,audio}} = 17.4\ seg$$

Luego se probó el algoritmo en una canción tocada únicamente por un piano. Esto se debe a que se buscó probar la validez del algoritmo en una canción un poco más compleja que no posea un instrumento de percusión que claramente índice los pulsos subyacentes a la canción. El resultado puede verse en 'Fig. 10.', luego del transitorio el resultado del BPM es de 86, lo cual es una buena estimación ya que se considera que el BPM real de la canción es 87. Se puede ver que para esta canción el transitorio duro el doble que para la primera canción que tenía una batería que indicaba el ritmo.

V. Separación espectral

A. *Objetivo y resumen*

Se tomó como objetivo agregar al programa principal una funcionalidad que permita obtener un archivo de audio únicamente con los instrumentos de percusión y otro con los instrumentos armónicos a partir de un único archivo de audio con una canción.
Dicho objetivo pudo cumplirse empleando el algoritmo obtenido de [7]. El algoritmo consiste en la aplicación de filtros sobre el espectrograma del audio original para crear otros dos espectrogramas a partir de los cuales se generan dos mascaras binarias utilizadas para obtener los audios deseados. La funcionalidad lograda puede utilizarse simplemente indicando el archivo de audio que se quiere separar en parte armónica o parte percusiva. Adicionalmente, cuenta con cuatro parámetros que pueden ser configurados por el usuario para ajustar los resultados obtenidos. Los parámetros con mayor efecto en los resultados fueron el factor de separación β y el tamaño de muestras a tomar para cada Fast Fourier Transform(FFT) del espectrograma.

B. *Sonidos percusivos y armónicos*

Para comenzar a analizar como cumplir nuestro objetivo hace falta primero definir las características de los elementos percusivos de un audio, así como también de los armónicos.

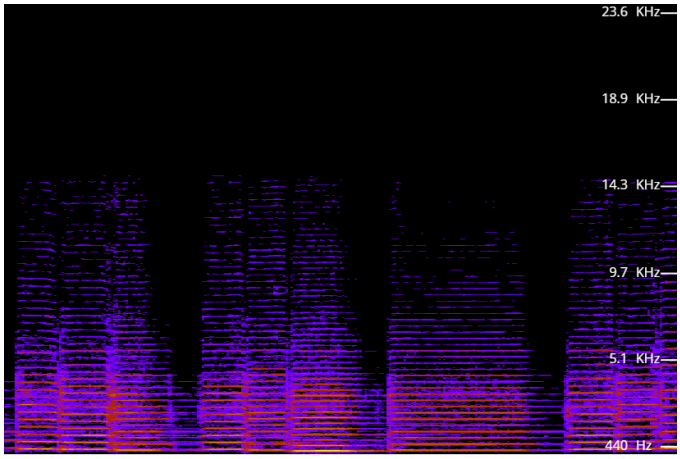


Fig. 12. Espectrograma de un violín

Como puede verse en ‘Fig 12.Fig’ el espectro de un violín en el tiempo forma estructuras horizontales en el tiempo, lo que quiere decir que su espectrograma es continuo en el tiempo pero no en frecuencia.

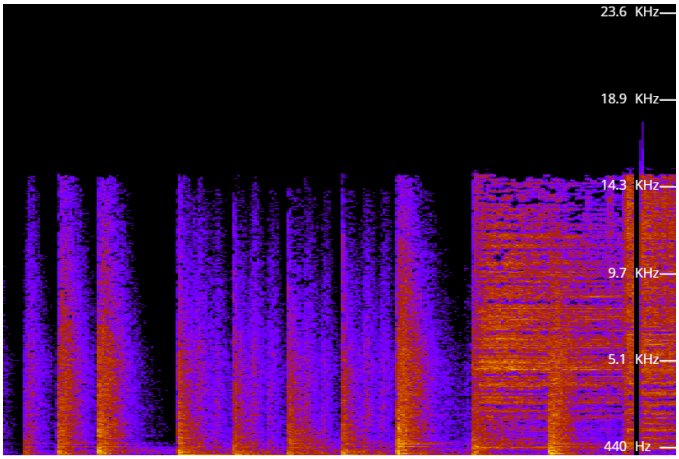
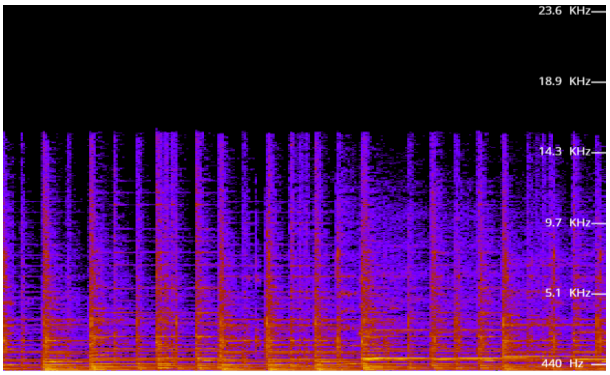


Fig. 13. Espectrograma de un solo de batería

Por otro lado, como se puede ver en ‘Fig. 13.’, un instrumento percusivo como lo es una batería presenta estructuras verticales en su espectrograma, que significa que presenta continuidad en frecuencia, pero no en el tiempo. Esto se debe a que los instrumentos percusivos tienen una duración muy corta en el tiempo lo que significa que se extienden en el dominio de la frecuencia. Esta es la razón por la que los instrumentos percusivos se denominan también instrumentos inarmónicos debido a que como contienen una banda muy amplia de frecuencias no es posible caracterizarlos por las frecuencias o armónicos que poseen.



Espectrograma original del audio

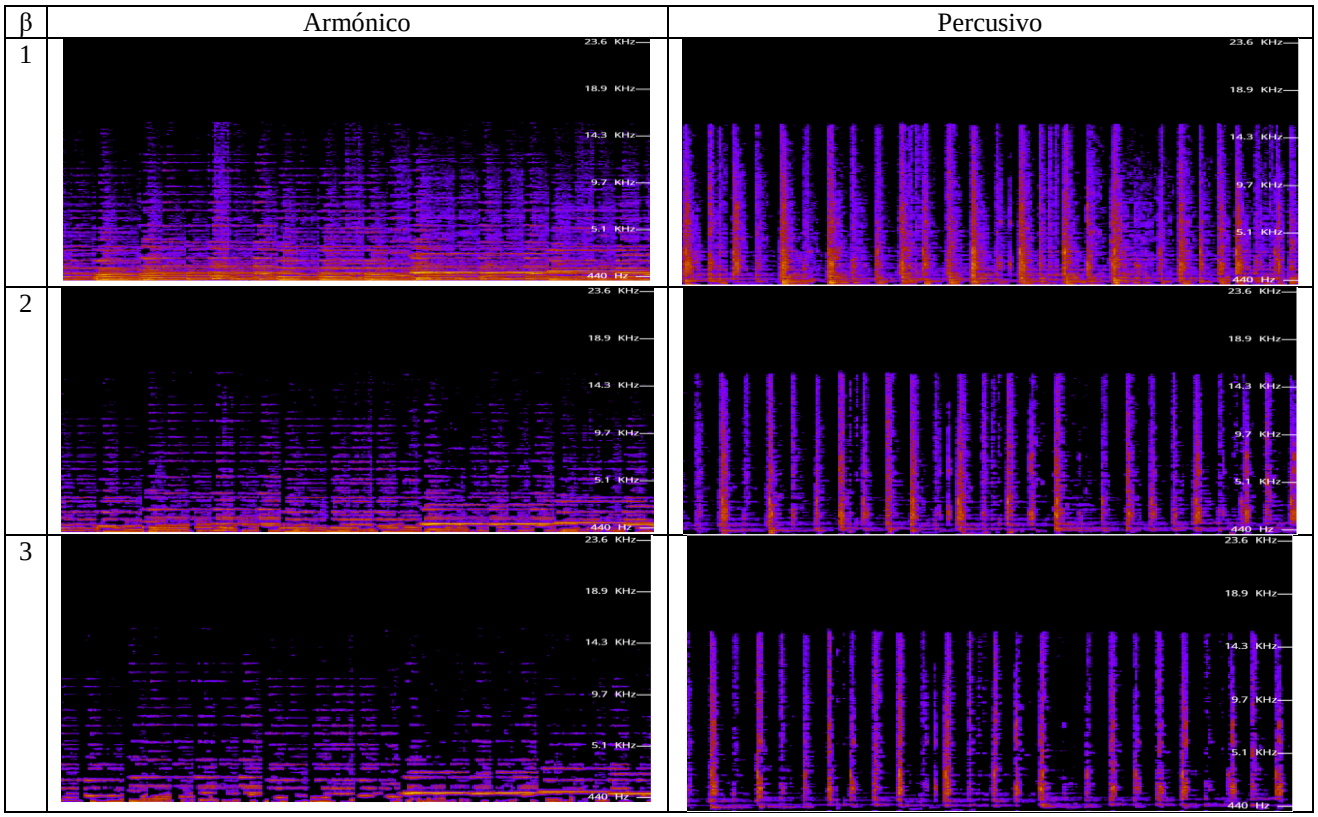


Fig. 14. Espectrogramas obtenidos utilizando N=1024, Lp = La = 15

C. Algoritmo

1) Calculo de espectrograma y filtrado

En primer lugar, se debe obtener el espectrograma del vector con las muestras de la canción. El número de muestras a utilizar para cada FFT de la Short Time Fourier Transform(STFT) es un parámetro importante del algoritmo y se denotara con el símbolo N.

La elección de los demás parámetros de la STFT como el tipo de ventana y el overlap no tienen un efecto apreciable en el resultado final, por lo que simplemente se utiliza la ventana de haming con un overlap del 50%. Una vez realizada la STFT del vector con muestras se toma el modulo del resultado y se obtiene el espectrograma del audio original.

El paso siguiente es la aplicación del filtro armónico y del filtro percusivo al espectrograma calculado previamente. Al aplicar dichos filtros se obtienen dos nuevos espectrogramas. Los filtros utilizados se definen de la siguiente manera:

$$S_P[t, k] = \text{Median}(S[t, k - L_p/2], \dots , S[t, k + L_p/2])$$
$$S_a[t, k] = \text{Median}(S[t - L_a/2, k], \dots , S[t + L_a/2, k])$$

Donde t es el índice para los bloques de tiempo y k es el índice para los bins de frecuencia. La función Median

devuelve la mediana del conjunto de valores recibidos. Las variables L_p y L_a son las longitudes de los filtros y son dos de los cuatro parámetros del algoritmo.
Los valores de L_p y L_a no afectan drásticamente los resultados finales mientras no se tomen valores extremos. En [1] se sugiere tomar L_p equivalente a un ancho de banda de 500Hz y L_a equivalente a 200ms. Dicha selección depende de la frecuencia de muestreo del audio utilizado pero por lo general ronda en valores de L_p y L_a entre 10 y 15.

2) Aplicación de mascaras

Una vez que se tienen los espectrogramas filtrados S_P y S_a, se generan dos mascaras binarias definidas de la siguiente manera:

$$M_a[t,k] = (S_a[t, k] / S_P[t, k]) > \beta$$
$$M_p[t,k] = (S_p[t, k] / S_a[t, k]) \geq \beta$$

El símbolo β es el factor de separación y es el último parámetro del algoritmo de separación. Multiplicando una máscara con la STFT del audio original se obtiene una nueva STFT correspondiente a la parte percusiva o armónica del audio dependiendo de la máscara utilizada.
Finalmente, para obtener los audios deseados solo se debe aplicar la Inverse Short Time Fourier Transform(ISTFT) a cada uno de los resultados obtenidos luego de aplicar las máscaras a la STFT del audio original.

D. Implementación y resultados[1]

El algoritmo explicado previamente se implementó en Python como una funcionalidad más dentro de un programa que realiza varias funciones relacionados con archivos de audio. Se puede observar de “Fig. 14.” los espectrogramas de la canción utilizada, así como de los resultados obtenidos de la separación. Se puede ver que mientras mayor el valor de β se distinguen mejor las estructuras horizontales en el espectrograma armónico, mientras que en el percusivo se distinguen mejor las estructuras verticales. Esto se debe a que el criterio de separación es más estricto por lo que hay una menor fuga de elementos percusivos al espectrograma armónico y viceversa.
Asimismo, a mayor β se puede ver que se pierde una energía notable en el audio resultante, esto tiene sentido ya que al tener una separación más estricta se incrementa la cantidad de elementos que no son ni armónicos ni percusivos y son descartados.
Al tomar valores de N más altos se tiene que aumenta considerablemente la resolución en frecuencia por lo que aumenta la cantidad de detalles en los sonidos resultantes. Particularmente para N=512 los sonidos en el audio percusivo son cortos y secos, mientras que los sonidos en el audio armónico son poco claros y no se reconocen bien los instrumentos.
Al aumentar el valor de N se distinguen cada vez mejor los instrumentos y los sonidos percusivos suenan menos secos (esto se debe a que, aunque se denominan ‘inarmónicos’ si se elimina su contenido de bajas frecuencias cambia el sonido producido). Aumenta la resolución en frecuencia, sin embargo, esto tiene como desventaja que disminuye la resolución en el tiempo. Los sonidos armónicos que se filtran al audio percusivo se escuchan más nítidos y distintivos por lo que parece como si la separación fuera menos exigente.
Se encontró que generalmente el valor que da los mejores resultados es N=1024.

[1] Todos los audios utilizados y generados se encuentran en https://tinyurl.com/ve5phd5

VI. Implementación y resultados

Luego de implementar los diferentes bloques se integraron en un solo programa donde se detectaban las diferentes notas tocadas y su correspondiente tono. En la Fig. 6. se pueden contemplar los resultados para dos breves canciones. Estas pudieron ser exitosamente procesadas, encontrando con precisión el inicio y fin de las notas, así como su tono.

VII. Conclusión y futuras mejoras

En conclusión, se pudo implementar con éxito el algoritmo propuesto obteniendo los datos necesarios para procesar el WAV a formato MIDI. Una mejora al algoritmo propuesto seria expandirlo a muestras polifónicas, es decir, que reconozca acordes y notas individuales, con el fin de poder convertir un mayor rango de audios a MIDI. Otra idea interesante es poder hacer el procesamiento en tiempo real, o lo mas parecido a tiempo real, de manera que el programa pueda usarse para aplicaciones con fines educativas o por ejemplo de afinación de instrumentos.

Con respecto a la detección del bpm se podrían utilizar distintos algoritmos y comparar la rapidez, tasa de éxito de cada uno. Asimismo, la implementación en Python no permite la implementación en tiempo real por lo que, para un proyecto en tiempo real, aunque el algoritmo es válido el código debería ser reescrito en otro lenguaje.

Una última mejora posible al programa para que sea menos propenso a fallas es la utilización de un atenuador de ruido, ara suprimir el ruido indeseado en el audio y así poder interpretar mejor las notas y sus tonos.

APÉNDICE

REFERENCIAS

[1] P. Brossier, J. P. Bello, M. D. Plumbley “Real-time temporal segmentation of note objects in musical signals” Centre of Digital Music, Queen Mary University of London, 2004, pp. 2–3.
[2] R. Lawrence, “On the use of autocorrelation analysis for pitch detection” February 1977.
[3] P. De la Cuadra, “Efficient Pitch Detection Techniques for Interactive Music”.
[4] H. Kawahara, A. de Cheveigne, “YIN, a fundamental frequency estimator for speech and music” Octubre 2001.
[5] Luis Cavo, Siyu Tan, Adam Urga, “Beat Detection Algorithms in Signal Processors ETIN80” 2016.
[6] Jaime Gancedo, Sakif Hossain, Wenpeng Song, “Design and implementation of a Beat Detector algorithm (ETIN80 report)” 2018.
[7] Driedger, M. Muller and S. Disch, *Extending harmonic-percussive separation of audio signals*. Erlangen, Germany, 2014.

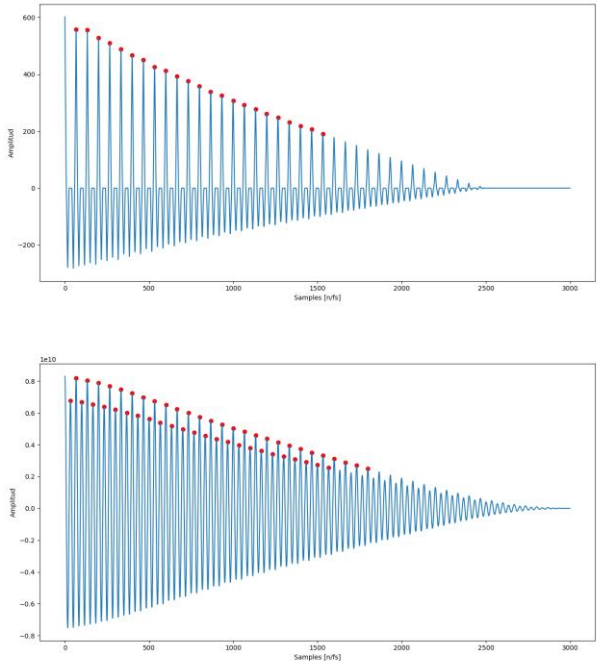


Fig. 7. Funciones de autocorrelación del audio de un oboe. La segunda imagen muestra el error en la detección del pico con la función de autocorrelación. En la primera imagen se preprocesó la señal por lo que se evita el error de detección en el pico.

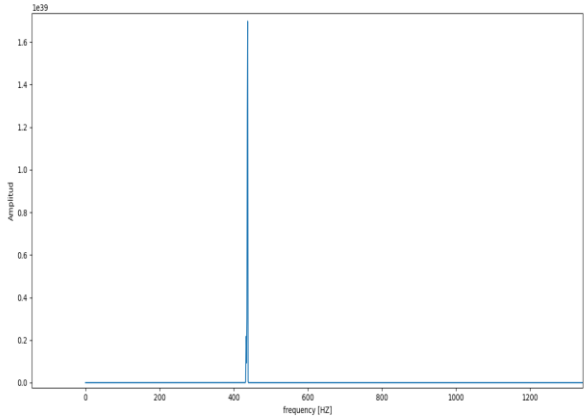


Fig. 8. Resultado de la implementación del algoritmo HPS

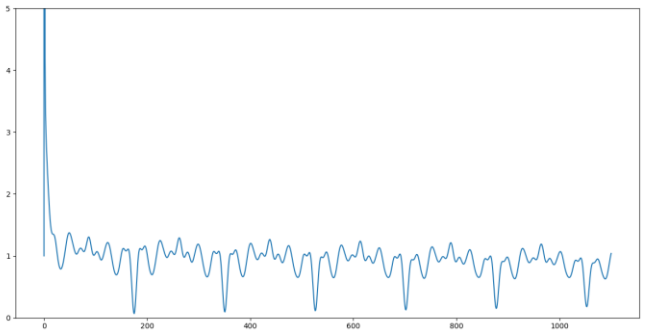


Fig. 9. Funcionamiento del algoritmo YIN