Vocoder de aplicación musical en tiempo real

Matías Bergerman, Lucas Agustín Kammann, Rafael Nicolás Trozzo

I. Introducción

La aplicación de efectos digitales a la voz humana en la música es un tema de mucha actualidad. Una gran cantidad de artistas hacen uso de estas herramientas, siendo el *autotune* la más conocida entre ellas.

Si bien el *autotune* se popularizó durante el siglo XXI, la modificación digital de la voz comenzó en la segunda mitad del siglo XX, y una de las primeras técnicas utilizadas fue la del *vocoder* (nombre derivado de *voice encoder*), la cual se desarrolla en este trabajo. Por ejemplo, *Electric Light Orchestra* lo utilizó en la década de 1970 en su canción *Mr. Blue Sky*. Otro ejemplo son las canciones de los grupos *Kraftwerk* y *Daft Punk*.

La técnica del *vocoder* se basa en el modelado del sistema de producción de voz del ser humano a partir del conocimiento de su anatomía.

El sistema de producción de voz humano se compone de órganos de respiración, fonación y articulación, los cuales pueden ser modelados respectivamente como una fuente de corriente de aire, un modulador que transforma el flujo de aire continuo en pulsos (denominados *pulsos glotales*) o en flujo turbulento (dependiendo de si se tratara de un sonido sonoro o sordo), y por último un filtro. Este filtro representa físicamente cómo la onda generada se propaga a través de los órganos de articulación hasta el exterior. La Figura 1 muestra un diagrama en bloques del modelo de producción de voz.

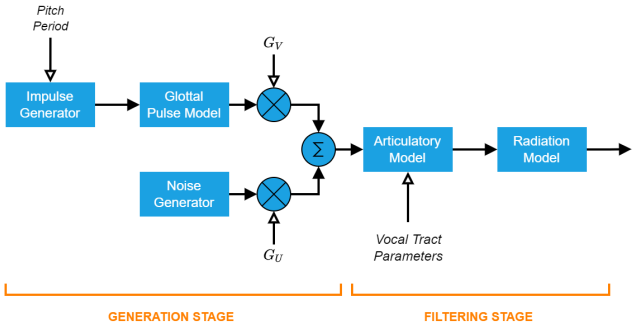

Figura 1. Modelo de producción de voz.

En la técnica del *vocoder*, se buscan los parámetros del filtrado para sintetizar la voz reemplazando al generador por una fuente de sonido arbitraria. El método fue desarrollado para la codificación de voz en telecomunicaciones y empezó a ser utilizado en el ámbito musical.

M. Bergerman, Alumno, Dpto. de Ingeniería Electrónica, ITBA, mbergerman@itba.edu.ar.
L. A. Kammann, Alumno, Dpto. de Ingeniería Electrónica, ITBA, lkammann@itba.edu.ar.
R. N. Trozzo, Alumno, Dpto. de Ingeniería Electrónica, ITBA, rtrozzo@itba.edu.ar.

En este trabajo, se implementa un *vocoder* de tiempo real. Para ello, se usa el algoritmo LPC para estimar en tiempo real el filtro de una voz capturada mediante un micrófono, y se la reproduce reemplazando el generador por una onda sintetizada en base a los comandos de un controlador MIDI[1].

II. Arquitectura del sistema

La Figura 2 muestra un diagrama en bloques con el diseño del sistema de procesamiento para aplicar *vocoding*. El sistema posee dos entradas, la voz humana capturada por un micrófono y comandos MIDI enviados por un controlador a través de una conexión USB. La voz sintetizada es enviada a un micrófono virtual. Las funciones principales del sistema son:

- Detección de voz para habilitar fuente sintética
- Procesamiento en tiempo real
- Aplicación de *vocoding* a la voz humana

II-A. *Esquema de framing*

La voz humana es una señal aleatoria que cambia en el tiempo según lo que la persona está diciendo o pronunciando, es decir, no es una señal estacionaria. Por lo tanto, el filtro del modelo de producción de voz será un sistema variable en el tiempo. No obstante, si se toman segmentos temporales de muy corta duración es posible encontrar estacionariedad local, por ende, se considera que los parámetros del filtro se mantienen fijos en ese intervalo. Así, se actualizan periódicamente los parámetros del filtrado y se asumen invariantes durante un intervalo que asegure estacionariedad local. Es importante aclarar que la Figura 2 representa cómo se realiza el procesamiento asumiendo que se tiene conocimiento a priori de las entradas para todo tiempo, es decir, se considera que el sistema trabaja *offline* (no en tiempo real).

La Figura 3 muestra que en el sistema de tiempo real es necesario tomar los datos de entrada con ventanas de duración ΔT_R cuya extensión deberá ser lo suficientemente larga para realizar el procesamiento de la siguiente ventana de datos sintetizados. De no ser así, la ventana enviada a la salida del sistema sería consumida por el micrófono virtual antes de que el procesamiento genere el siguiente segmento, provocando un tiempo de silencio que introduce artefactos que degradan la experiencia. Por otro lado, hay una relación de compromiso dado que aumentar la duración de los segmentos provoca un aumento en el tiempo de retardo. Si el tiempo de retardo se vuelve notorio para los usuarios, la experiencia se ve degradada.

Por otro lado, estas ventanas se deben fragmentar en tramas de menor duración para conseguir la estacionariedad local.

[1] https://cecm.indiana.edu/361/midi.html

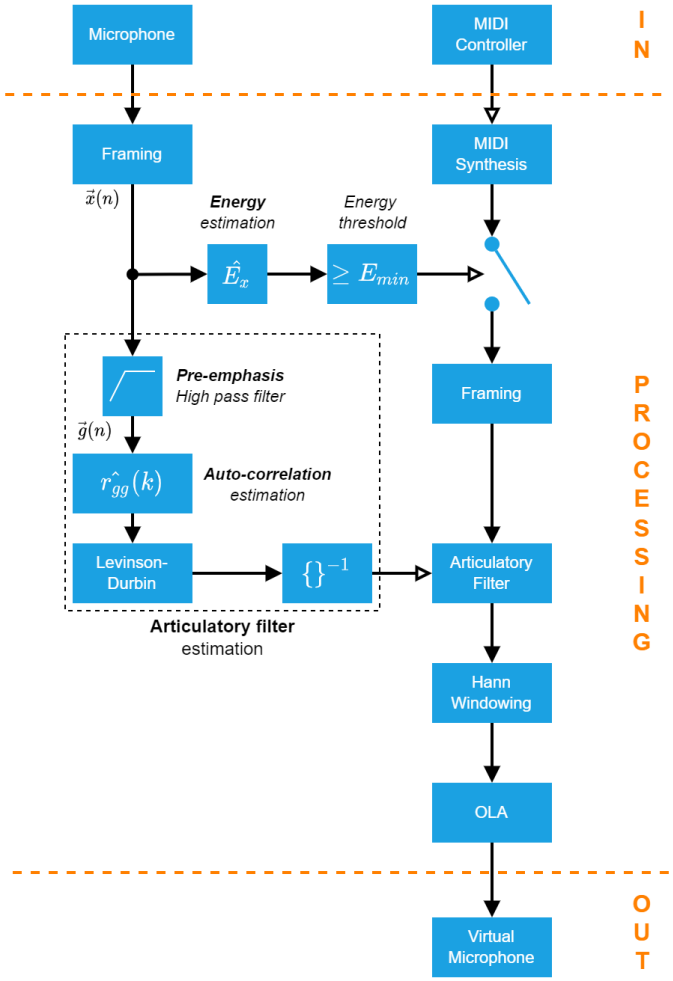

Figura 2. Diagrama en bloques del sistema.

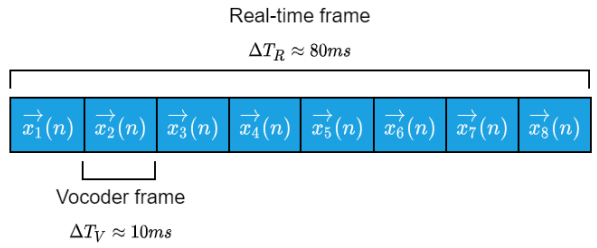

Figura 3. Esquema de *framing* en tiempo real.

Para eso, se suelen tomar segmentos de entre 10 ms y 20 ms. El procesamiento se aplica sobre estas tramas.

En el sistema desarrollado se utilizan tramas de duración $\Delta T_R \approx 80ms$. Se elige este valor buscando que el tiempo de retardo sea mínimo, pero que al mismo tiempo no se presenten artefactos notorios para el usuario. Luego, para garantizar la estacionariedad, la trama se fragmenta en segmentos de duración $\Delta T_V \approx 10ms$.

II-B. Síntesis MIDI

La Figura 4 muestra el esquema de *buffering* empleado para sintetizar el sonido a partir de los comandos MIDI en tiempo real. Para sintetizar, es necesario utilizar el método *overlap-add* empleando ventanas de Hann con una superposición del 50 % de los segmentos, de esta forma se cumple la condición de reconstrucción perfecta y la señal obtenida no presenta artefactos [2]. En cada ciclo de procesamiento es necesario generar el segmento de datos que será consumido en el siguiente ciclo. Para esto, se propone utilizar un *buffer* con espacio en memoria para tres segmentos: el que está listo para ser consumido durante el ciclo de procesamiento, el que está siendo preparado para el siguiente ciclo y un segmento que se empieza a preparar para dos ciclos en el futuro. Esto quiere decir que los segmentos pasan por dos etapas antes de estar listos para ser consumidos.

En cada ciclo se sintetizan dos tramas a las cuales se les aplica una ventana de Hann y se suman al *buffer*. Para la síntesis de cada trama se genera una combinación lineal de ondas cuadradas de amplitud constante, cada una ellas con una frecuencia correspondiente a las notas que están siendo reproducidas según los comandos MIDI. La síntesis mediante ondas cuadradas podría ser mejorada utilizando un algoritmo basado en síntesis BLIT (*Bandlimited Impulse Train*), puesto que tomar muestras de una onda cuadrada introduce alias en la señal resultante [3].

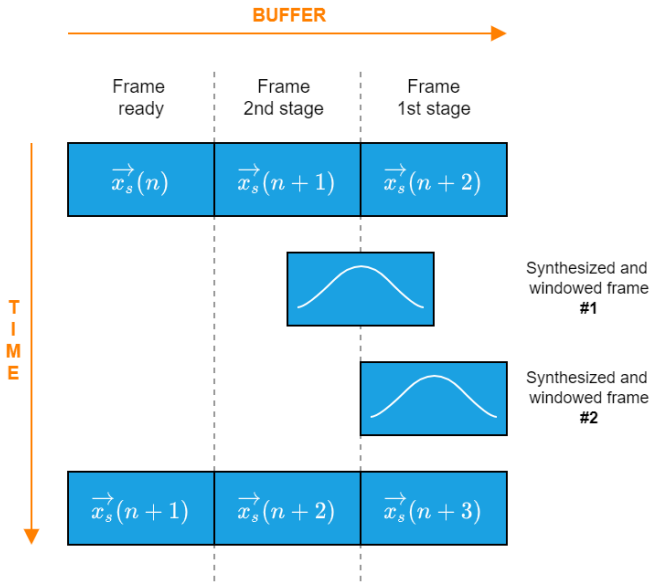

Figura 4. Esquema de *buffering* para la síntesis de sonido a partir de comandos MIDI en tiempo real.

II-C. Detección de presencia de voz

Se utiliza la Ecuación 1 para estimar la energía de la señal de voz en el segmento bajo análisis y se compara con un umbral configurable por el usuario. Cuando la energía no supera el umbral, se establece que no hay presencia de voz y se desactiva la síntesis, permitiendo que haya silencio.

$$\hat{E}_x = ||\vec{x}(n)||^2 \tag{1}$$

II-D. *Vocoder*

Para realizar el *vocoding* mediante LPC, sobre la señal de voz primero se aplica un filtro pasa-altos de preénfasis con el fin de compensar la respuesta pasa-bajos de los pulsos glotales. Para el preénfasis se utiliza el filtro de primer orden auto-regresivo (IIR) de la Ecuación 2, con un parámetro $\alpha = 0{,}97$. Seguidamente, se estima la autocorrelación de la señal para luego aplicar el algoritmo de Levinson-Durbin y obtener los coeficientes del filtro de error. El filtro obtenido debe ser invertido para que pueda ser aplicado sobre una nueva señal generadora. El sistema digital opera con una frecuencia de muestreo $f_s = 48kHz$.

$$y(n) = y(n) - \alpha \cdot y(n-1) \tag{2}$$

Cuando la frecuencia de muestreo es $f_s = 16kHz$, suele emplearse un filtro de orden $M = 12$ por la cantidad de polos que se consideran en el modelo de producción de voz. En las pruebas realizadas con el sistema, se encontró que trabajando con una frecuencia de muestreo de $f_s = 48kHz$, subir el orden mejora la inteligibilidad de la voz y se escogió un orden $M = 36$. Esto puede deberse a que por aumentar la frecuencia de muestreo, la resolución espectral de la estimación del filtro con $M = 12$ disminuye, provocando un mayor error en la obtención del filtro articulatorio.

Para sintetizar la señal de salida se emplea el método *overlap-add* utilizando ventanas de Hann con superposición del 50 % de los segmentos. Esto quiere decir que en cada iteración es necesario procesar dos segmentos porque avanzan a la mitad del tiempo. Entonces, hay que guardar en memoria los segmentos del ciclo anterior, y además, la salida saldrá con un tiempo de retardo.

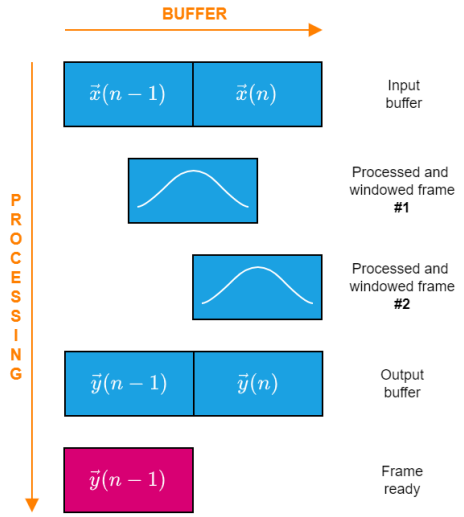

Figura 5. Esquema de *buffering* para el procesamiento del *vocoder*.

En la Figura 5 se muestra el esquema de *buffering* para la aplicación del *vocoding*. La entrada y la salida cuentan cada una con un *buffer* que permite almacenar en memoria dos segmentos. En el ciclo n, se tienen los datos de entrada de los ciclos ciclos n y $n-1$. Se aplica el procesamiento del *vocoder* sobre la superposición de las tramas y sobre la trama nueva. Al finalizar el ciclo, se obtiene como resultado la trama correspondiente al ciclo $n-1$. Por eso es que hay un retardo entre la entrada y la salida.

II-E. *Procesamiento de audio en tiempo real*

Para el manejo de la entrada y salida de audio se utilizó la biblioteca PyAudio[2] en forma asincrónica. Se configuran *callbacks* para cada trama que debe ser procesada. Al tratarse de procesamiento en tiempo real, se debe evitar cualquier tipo de inconsistencia en la sincronización del sistema, ya que su efecto en el audio final puede ser notorio.

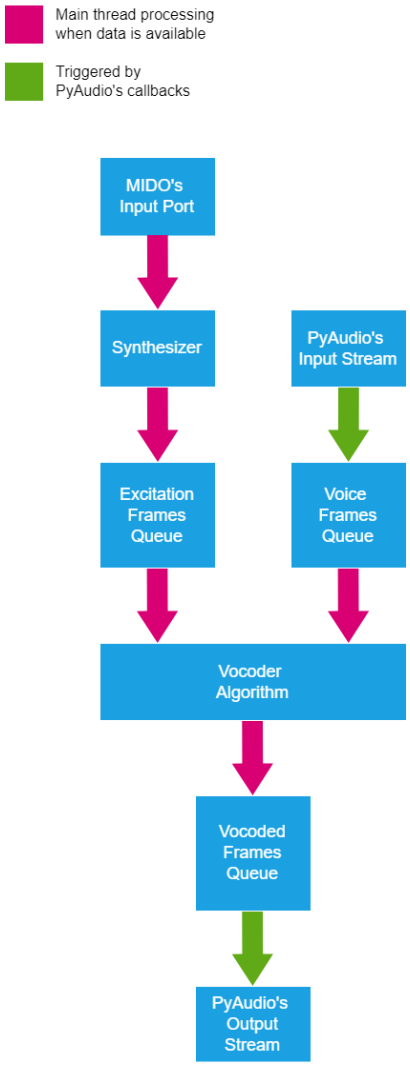

Figura 6. Diagrama del procesamiento de audio en tiempo real.

La Figura 6 muestra un esquema de cómo se implementa la sincronización. Por un lado, un hilo de procesamiento principal espera eventos del controlador MIDI con el cual se establece una comunicación con la biblioteca *mido*[3], y con ellos se configura el sintetizador. Cuando se agotan los datos, se sintetizan nuevos y se cargan en una cola. Siempre que las colas para la excitación y la voz poseen datos disponibles, se

[2] https://pypi.org/project/PyAudio/
[3] https://pypi.org/project/mido/

procesan y se sintetizan segmentos que se cargan en la cola de salida. La entrada de audio del micrófono carga datos en la cola de forma asincrónica a través del *callback* de PyAudio, y la salida consume datos de la cola de la misma forma.

II-F. Interfaz gráfica

La aplicación cuenta con una interfaz gráfica desarrollada con *tkinter*[4]. Esta interfaz es ejecutada en un hilo separado del resto del sistema para permitir la ejecución concurrente del procesamiento de audio. La interfaz permite seleccionar las fuentes de audio y de MIDI que vaya a utilizar el procesamiento. Además, permite ajustar el volumen de salida y el umbral de sensibilidad en la voz. Se puede observar una captura de la misma en la Figura 7.

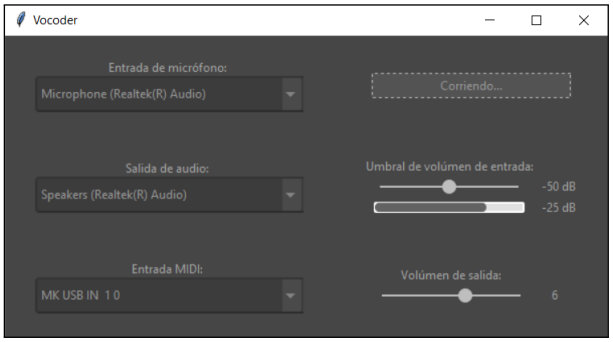

Figura 7. Interfaz gráfica implementada.

III. Resultados

Dada la naturaleza del proyecto, los resultados del trabajo son mayormente perceptuales. Aquí se puede escuchar una demostración de los resultados: https://www.youtube.com/watch?v=WvF8JJHSznA.

Se considera un resultado satisfactorio cuando no se perciben artefactos o *glitches* en el sonido de salida. Para lograrlo, se tuvieron que variar los tamaños de las ventanas de datos utilizadas.

En el caso de los *frames* de entrada y salida de audio, se tuvo que elegir un tamaño lo suficientemente grande como para evitar que el *overhead* del manejo de entradas y salidas limitara demasiado el tiempo de procesamiento. El mínimo tamaño que dió resultados positivos fue de 80 ms. Luego, para el tamaño de la ventana del *vocoder*, se obtuvieron resultados satisfactorios para *frames* de 10ms.

El retardo de procesamiento depende principalmente del tamaño de los *buffers* de entrada y salida de audio. Cada trama de datos de entrada tiene un retardo de procesamiento de aproximadamente 80 ms. Por la sincronización del sistema, podría ocurrir que los datos no se consuman hasta un ciclo posterior, lo que provocaría que haya un retardo máximo de 160 ms.

[4]https://docs.python.org/es/3/library/tkinter.html

IV. Conclusión

Se observó que la inteligibilidad de la voz cambia en función de la excitación utilizada en el *vocoder*. Cuando la forma de onda es cuadrada, a medida que se agregan más notas en simultáneo, aumenta la inteligibilidad. Esto puede deberse a que las ondas cuadradas tienen un espectro discreto con contenido espectral en frecuencias particulares, por ende, al agregar más tonos en simultáneo se aumenta la cantidad de frecuencias con energía. Esto provoca que aumente el contenido espectral de la señal resultante. En otras palabras, se muestrea el filtro articulatorio en una mayor cantidad de frecuencias.

En relación a esta última observación, es importante notar un punto a criticar del sistema desarrollado. El sistema genera las ondas de excitación con amplitud constante. Esto quiere decir que no hay un control de la energía de la señal, lo cual presenta múltiples desventajas. Si no cambia la cantidad de notas que están siendo enviadas por el controlador MIDI, la sonoridad permanece invariante a pesar de que el usuario hable más fuerte o más bajo. Modificar la cantidad de notas modifica además la energía de la señal, provocando que esos dos parámetros no sean independientes. Por último, sin un control que asegure una energía establecida según algún criterio, el uso de múltiple notas tiende a saturar la señal y provocar distorsiones. Una forma de corregir estos problemas sería utilizar la estimación de la energía de la voz, con un filtrado para evitar cambios abruptos en la intensidad, que permita controlar la amplitud de las ondas de excitación para que el *vocoder* produzca una voz sintética que refleje la sonoridad de la voz capturada.

Si bien el sistema no ajusta la energía de la señal sintética, sí detecta la presencia de voz para garantizar que existan silencios cuando el usuario no está hablando. Un problema con la implementación actual es que al detectar la voz por energía, no se tiene en cuenta que los fonemas por naturaleza no poseen necesariamente la misma energía. Por ejemplo, en el caso de un sonido sordo, podría ocurrir que no supere el umbral de sensibilidad que se ajustó para un sonido sonoro. Una forma de corregir esto, sería tener un detector adicional para sonidos sordos, a partir de la cantidad de cruces por cero.

Referencias

[1] Hyung-Suk Kim, *Linear predictive coding is all-Pole Resonance Modeling*. [Online]. Disponible: https://ccrma.stanford.edu/~hskim08/lpc/. [Accedido: 23-Mayo-2022].

[2] J. Smith, *Spectral audio signal processing*. Stanford: W3K, 2011. [Online]. Disponible: https://ccrma.stanford.edu/~jos/sasp/Overlap_Add_Decomposition.html. [Accedido: 23-Mayo-2022].

[3] T. Stilson, J. Smith, *Alias-Free Digital Synthesis of Classic Analog Waveforms*. Disponible: https://ccrma.stanford.edu/files/papers/stanm99.pdf#page=48. [Accedido: 23-Mayo-2022].