Informática Musical

Procesamiento de audio digital. PyAudio

En los siguientes ejercicios utilizaremos Python 3 con las librerías pyAudio y numPy, y la clase *KBHit* que se proporciona. Puede utilizarse alguno de los entornos de programación para Python instalados en los laboratorios si se tiene experiencia previa o bien un editor como Notepad++ y trabajar desde línea de comandos (consola de Anaconda). Para familiarizase, comenzar ejecutando el reproductor de archivos *.wav* con arrays de numPy que se ha visto en clase.

Algunos ejercicios están marcados como **(Entregable)**. Dichos ejercicios pueden resolverse y subirse al campus como práctica de laboratorio evaluable. Los alumnos, en grupos de 2, deben seleccionar alguno(s) de ellos, desarrollarlos y subirlos al Campus para su posterior evaluación.

1. Modificar el oscilador sinusoidal visto en clase para poder variar su frecuencia en tiempo de ejecución con las teclas `'F'` (subir) y `'f'` (bajar).

2. Implementar un oscilador como el anterior, pero de onda cuadrada. Hacer otro de diente de sierra y otro triangular.

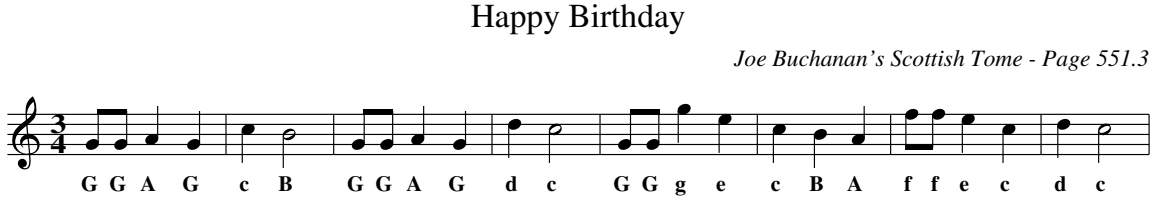
3. Implementar un *modulador amplitud*, i.e., un efecto que tome como entrada un array de muestras y las multiplique por una señal sinusoidal de frecuencia dada. Aplicarlo sobre alguno de los osciladores de los ejercicios anteriores.

4. Implementar un reproductor de *wav* que normalice el volumen de la salida en función de los picos (máximos y mínimos) de los CHUNKS previamente procesados. El parámetro multiplicador debe modificase *suavemente* para no generar cambios abruptos de dinámica.

5. Implementar una versión con *callback* del grabador de archivos *.wav* visto en clase.

6. Investigar la representación de muestras estéreo y hacer un efecto *modulador de balance* que utilice un oscilador sinusoidal asociado al balance de la pista (con -1 se envía toda la señal a la pista izquierda; con 1 a la derecha).

7. **(Entregable)** Recordemos la partitura de la hoja anterior:



Podemos hacer una representación de esta partitura mediante una lista (Python) de pares *(nota,duración)*. Las notas se representan con las letras `A B ...G`; para subir una octava se utilizan las minúsculas `a b ...g`[1]. La duración se representa con un número que indica las *unidades de tiempo* (la unidad que se puede fijar en el programa). De este modo, la partitura anterior quedaría:

$$[(G,0.5),(G,0.5),(A,1),(G,1),(c,1),(B,2),\ldots]$$

Recordemos la tabla de frecuencias (para una octava):

C	D	Ee	F	G	A	B	c	d . . . a
523,251	587,33	659,255	698,456	783,991	880	987,767	. . .	. . .

Puede extenderse la implementación para leer de archivo la partitura en notación ABC `https://es.wikipedia.org/wiki/Notacinn\_musical\_Abc` y reproducirla.

[1] Para la siguiente octava por arriba podrían utilizarse los apostrofes (`a'`), para la siguiente doble apostrofe (`a''`), etc. Para bajar octava se podrían utilizar las comas (`A, A,,`)

8. **(Entregable)** Implementar un piano simple utilizando el sample proporcionado *piano.wav*. Utilizaremos KBHit para mapear las teclas *zxc...* a una octava del piano y *qwe...* a la octava superior. Una forma sencilla de alterar el pitch de la nota es modificar la frecuencia de muestreo en el stream de salida, haciendo los cálculos pertinetes.

9. **(Entregable vía campus como práctica de laboratorio)** Implementar una clase Python para hacer un sencillo efecto de Delay (línea de retardo). Este efecto recibe una señal de entrada y devuelve la misma señal, pero retardada en el tiempo una cantidad prefijada de tiempo. Tanto la entrada como la salida, serán chunks de audio e internamente deberá gestionarse un buffer de datos.

10. **(Entregable)** Implementar un *idiotizador*[2]: abrir un stream de entrada y otro de salida, y reenviar la señal de entrada a la salida, con retardo (configurable) de tiempo.

11. Implementar una función para *remuestrear* un audio dado con un frame rate a otro especificado, sin alterar el pitch (investigar formas de interpolación para hacerlo).

[2] Véase `https://www.youtube.com/watch?v=zhUDQGWM8kM`