Informática Musical

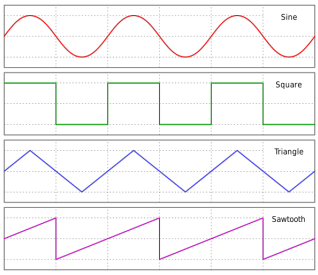
Procesamiento de audio digital. PyAudio

En los siguientes ejercicios utilizaremos Python 3 con las librerías estándar de PyAudio, SciPy y NumPy, así como la clase *KBHit* que se proporciona. Para empezar y comprobar el funcionamiento de PyAudio puede ejecutarse el reproductor de archivos *.wav* con arrays de numPy que se ha visto en clase.

Los ejercicios marcados con **(Entregable)** pueden resolverse como práctica de laboratorio evaluable. Los alumnos, en grupos de 2, deben seleccionar alguno(s) de ellos, desarrollarlos y subirlos al Campus.

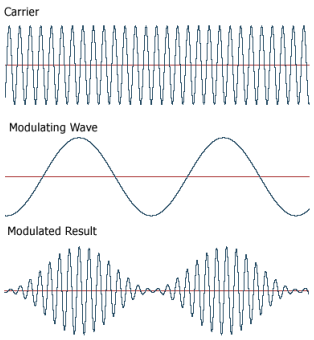
1. Modificar el oscilador sinusoidal visto en clase (con chunks) para poder variar su frecuencia en tiempo de ejecución con las teclas `'F'` (subir) y `'f'` (bajar).

2. Implementar osciladores similares al anterior, pero de onda cuadrada, diente de sierra y triangular.



   Estas formas pueden generarse de manera manual o utilizar predefinidas de SciPy[1]

3. Implementar un *modulador amplitud*: dada una señal de entrada sube y baja el volumen cíclicamente. Este efecto se consigue multiplicando la señal de entrada por un oscilador sinusoidal (de la frecuencia deseada) que oscile en el intervalo $[a, b]$ con $0 \leq a \leq b \leq 1$.



   Probarlo utilizando alguno de los osciladores anteriores. Implementar también un *modulador en anillo* (ring modulation), similar pero con $a = b \in [0, 1]$.

4. Implementar un reproductor de *wav* que *normalice* el volumen de la salida, procesando por chunks. Para cada chunk calculará el valor máximo de las muestras en valor absoluto. Con ese coeficiente es fácil calcular el máximo valor por el que puede incrementar el volumen sin saturar la señal (todas las muestras deben quedar en el rango $[-1, 1]$).

   Recordemos que los archivos *wav* pueden codificar las muestras en distintos formatos (int o float con distinto número de bits, y rango de valores). Puede ser buena idea comenzar convirtiendo el array a formato `np.float32` (véase documentación de NumPy).

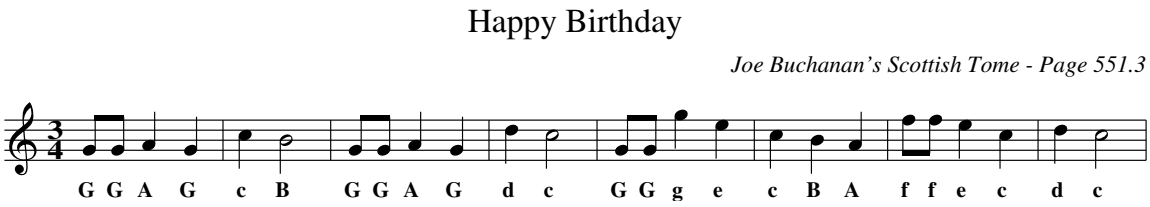
   Como mejora, el coeficiente multiplicador puede modificase *suavemente* entre chunks para no generar cambios abruptos de dinámica en el sonido de salida.

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

[1] Véanse las funciones `scipy.signal.square/sawtooth`

6. Investigar la representación de muestras estéreo en NumPy y hacer un efecto *modulador de balance*: utiliza un oscilador sinusoidal para modular el balance de la pistas (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`[2]. 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	. . .	. . .

   Recordemos también que dada la frecuencia de una nota, la de la octava superior se obtiene duplicando dicha frecuencia. La de la octava inferior, dividiendo entre 2. Para reproducir la partitura puede utilizarse el oscilador *osc(nota,volumen,duración)* visto en clase.

   Puede extenderse la implementación para leer de archivo la partitura en notación ABC (Wikipedia, entrada "Notación musical ABC").

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. A partir de muestra de una nota, por ejemplo un C, el resto de notas pueden obtenerse variando la velocidad de reproducción de esa muestra de acuerdo a estas proporciones:

Nota	C	D	E	F	G	A	B	c	d . . . a
Frec.	1.0	1.12	1.19	1.33	1.5	1.59	1.78	2.0	. . .

   Es decir, para obtener un D (re) podemos reproducir a 1.12 de velocidad. Esto altera el pitch de la muestra en la proporción justa para tener el D. Una forma sencilla de alterar el pitch de la nota es modificar la frecuencia de muestreo en el stream de salida, haciendo los cálculos pertinentes.

   Para hacerlo de forma más sofisticada observemos que dado un array de *NumPy*, si generamos otro que contenga solo los samples en posición par (de la mitad de tamaño), obtenemos un sonido una octava por arriba. Y al contrario, si generamos un array del doble de tamaño añadiendo muestras intermedias (como media de la anterior y la superior de una dada) obtenemos una octava menos. Esta idea de interpolación puede generalizarse para obtener cualquier nota, utitilizando la tabla de pitch dados.

9. **(Entregable)** 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.

   Para probar el funcionamiento, reproducir simultáneamente 2 veces la nota de piano del ejercicio anterior, una sin retardo y otra retardada una fracción de segundo (debe sonar un eco).

[2] 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,,`)

10. **(Entregable)** Implementar un *idiotizador*[3]: 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. Esta función servirá para cambiar la frecuencia de muestreo (sample rate). Investigar formas de interpolación para hacerlo.

12. Implementar un oscilador *chirp(frecIni,frecFin,dur)* que genere una señal sinusoidal de duración *dur*, que comience con una frecuencia *frecIni* y la incremente gradualmente (de manera lineal) hasta alcanzar *frecFin*. Este tipo de osciladores es conocido y utilizado en distintas aplicaciones (`https://en.wikipedia.org/wiki/Chirp`). Para implementarlo se puede partir del oscilador básico visto en clase y modificarlo para variar el parámetro de frecuencia en el argumento de la función *sin*.

    A continuación mejorar la funcionalidad para que, utilizando la misma muestra, pueda producir la señal con la frecuencia requerida.

13. **(Entregable)** Utilizar el filtro IIR visto en clase para implementar un filtro paso banda (BP) con frecuencia de corte y ancho de banda configurables. Utilizar dos filtros LP y HP en secuencia, calculando los valores de α según se ha explicado en clase.

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