Universidad Politécnica de Madrid
Grado en Matemáticas e Informática
Procesado Digital de la Señal

Procesado Digital de Audio

Sebastián Kay Conde Lorenzo (12454206M)
Francisco Manuel López López (26926830V)

Índice

Tarea: Generador de tonos

Realizar un generador de tonos (“Barrido_de_frecuencia.py”) que realice un barrido en frecuencia desde los 100 Hz hasta los 20 kHz para testear su rango auditivo ¿Cuál es la frecuencia más alta que puede oír?

Para esta tarea se ha usado el script **Generador_de_tonos.py**, que genera un tono (una señal sinusoidal pura con su frecuencia, duración y volumen parametrizados) y lo reproduce teniendo en cuenta una cierta frecuencia de muestreo (también parametrizada).

Luego, se ha creado una función que encapsule esta funcionalidad y se ha definido una frecuencia inicial y una frecuencia final de 100 y 20000 Hz respectivamente. Se ha definido también la duración de cada tono –que será de 1 segundo– y se ha definido la frecuencia de muestreo, además del número de Hz en el que aumenta cada tono, que será de 100 Hz en 100 Hz.

Junto con la función y estos parámetros, se arma un bucle *while* que vaya aumentando los Hz en la cantidad definida (de 100 en 100 en este caso) hasta que se haya superado la frecuencia final, y que vaya reproduciendo cada tono a la par que se imprime la frecuencia de dicho tono.

Hecho esto, se obtiene que la máxima frecuencia que personalmente puedo detectar es de unos 15000 Hz. Un poco por debajo de lo normal para mi edad.

Tarea: Espectro en frecuencia de la voz

Usando los ejemplos "Grabar_y_guardar_audio.py", graba tu voz en un fichero "voz_para_fft.wav". Analiza su espectro en frecuencias creando un fichero llamado "Transformada_de_Fourier_de_voz.py" (use el ejemplo "Cargar_y_reproducir_audio.py" para cargar el fichero). Analice el resultado del espectro y relaciónelo con el rango de frecuencias de la voz.

El script **Grabar_y_guardar_audio.py** incluye una función llamada **grabar_audio**, que, dado un índice de dispositivo (a los cuales se puede acceder con **sd.query_devices**) y una frecuencia de muestreo, graba audio desde un dispositivo hasta que se presione la barra espaciadora.

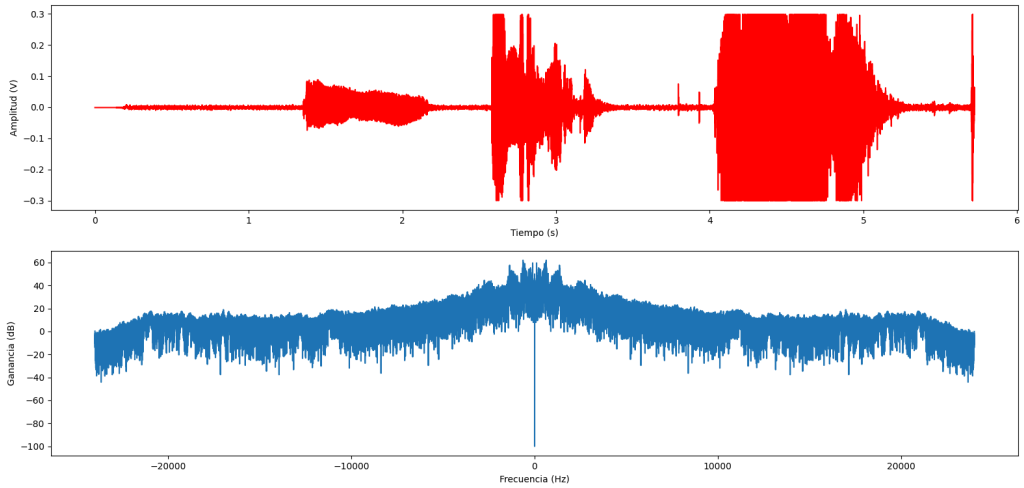
Por otro lado, el script **Cargar_y_reproducir_audio.py** se limita a abrir y reproducir un fichero wav con el módulo para tratar ficheros con dicha extensión de *soundpy*.

Luego, en el script **Transformada_de_Fourier.py** se puede apreciar cómo obtener la representación frecuencial de tres señales sinusoidales puras, siendo sus frecuencias, duración y volumen parámetros del programa. El paso al dominio de la frecuencia se hace mediante la *Fast Fourier Transform* (fft), por lo que también es un parámetro el número de muestras a tomar de dicha transformada. Al final del script se incluye también una sección de visualización donde se dibuja la señal en el dominio del tiempo y en el dominio de la frecuencia (en este último se utiliza una escala logarítmica para poder expresar la imágen de la transformada en decibelios).

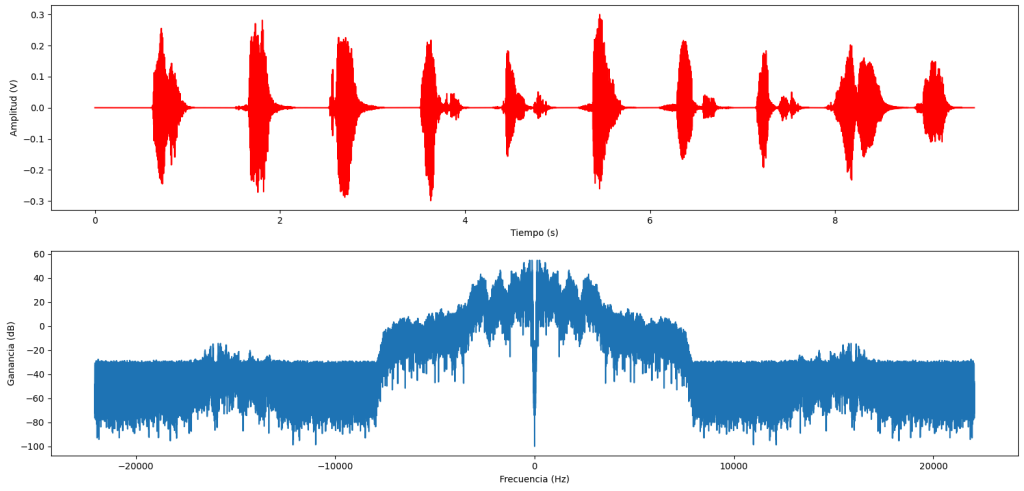
Entonces, usando estos tres scripts mencionados anteriormente se puede concebir el script solicitado: **Transformada_de_Fourier_de_voz.py**. Lo primero que se hace en este script es leer la señal *voz_para_fft.wav* (que fue grabada con anterioridad usando el script **Grabar_y_guardar_audio.py**), y fijar los parámetros a usar durante el resto de la ejecución. Luego, se obtiene la representación de la señal en el espectro de la frecuencia, utilizando la escala logarítmica que permita expresarla en decibelios. Finalmente, se da la opción a reproducir el audio cargado

(actualmente está comentado) y se dibuja la señal tanto en el dominio del tiempo (rojo) como en el dominio de la frecuencia (azul).

A continuación se adjunta el resultado de procesar con el script un audio personal llamado *voz_para_fft.wav* y el audio *hombre.wav*



voz_para_fft.wav



hombre.wav

Por último, se puede apreciar en ambas figuras que las componentes frecuenciales de las dos señales son mayores entre los 40 Hz y los 10 kHz, que es el rango de frecuencias de la voz humana. Incluso podemos apreciar que la cota superior no es de 10 kHz, sino de entre unos 4 u 8 kHz, lo cual se debe a limitaciones del micrófono.

Tarea: Filtros FIR

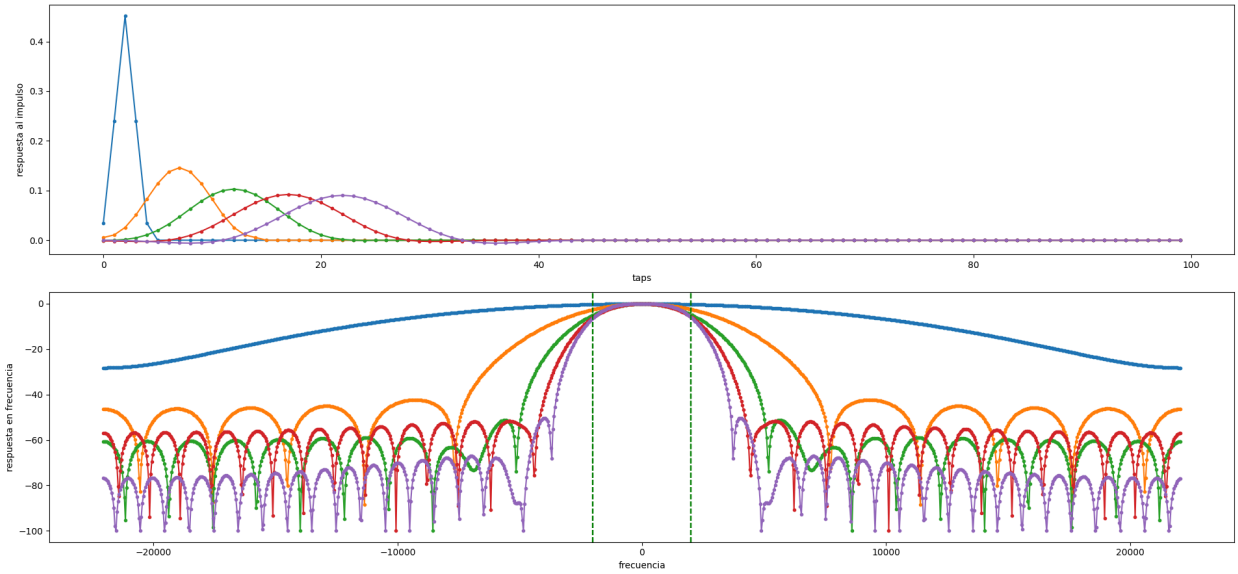
Analizar el fichero “filtro_FIR.py”, el cual genera múltiples filtros FIR con diferente número de etapas y calcula su respuesta al impulso en el dominio de la frecuencia. Generar un segundo fichero “filtro_FIR_con_audio.py” que cargue un fichero de audio “hombre.wav” o “ruido_blanco.wav”, aplique los distintos filtros generados (usar la función signal.lfilter), reproduzca el nuevo audio filtrado (2 segundos en torno al punto de mayor intensidad) y muestre en el gráfico el espectro en frecuencia del audio filtrado.

Comencemos analizando el fichero mencionado: **filtro_FIR.py** genera filtros FIR (Finite Impulse Response) variando su orden (que se entiende como su *tap* - 1. Es decir, el *tap* se puede entender como el número de coeficientes del filtro) y analiza su respuesta en el dominio del tiempo y la frecuencia. En las líneas 23-33, se definen los parámetros del filtro, como las frecuencias de corte (low_cut_off y high_cut_off), el rango de taps y el comportamiento del filtro (pasa altas, pasa bajas, suprime banda o pasa banda, en función a la combinación de las dos variables booleanas *use_low_cut_off* y *use_high_cut_off*). El filtro se aplica a la señal por ventanas de análisis, la ventana a usar está fija en una ventana *hamming* (aunque podría cambiarse o añadirse como parámetro adicional). Adicionalmente, de la línea 10 a la línea 20 se define una función para poder dibujar con *matplotlib* sobre una misma gráfica cada cierto tiempo (por defecto, cada 3 segundos) y, además, se indica la frecuencia de muestreo a usar.

Posteriormente, calculamos – por cada *tap* – los coeficientes del filtro con la función *signal.firwin* de la librería *scipy,* en función de los parámetros definidos previamente: Es decir, dependiendo de las frecuencias de corte y del tipo de filtro. A continuación, en las líneas 45-48, se calcula la respuesta en frecuencia (en escala logarítmica) del filtro haciendo uso de la FFT y se centra el origen con la función *np.fft.fftshift*. Luego, de la línea 50 a la 52 se obtiene la respuesta al impulso unitario del filtro, empleando la función *signal.lfilter.* Este tipo de filtros se pueden caracterizar por su respuesta a dicho impulso.

Finalmente, se *plotea* la respuesta al impulso (en tiempo vs amplitud) y la respuesta en frecuencia (en frecuencia vs decibelios).

A continuación se adjuntan 5 iteraciones del script utilizando un filtro pasa bajas, con frecuencia de corte de 2000Hz y con sus *taps* variando de 5 a 256 en incrementos de 10 en 10:



Haciendo uso de este fichero, construimos el solicitado por la tarea: **filtro_FIR_con_audio.py,** siguiendo las especificaciones del enunciado. Este procesa y visualiza una señal de audio almacenada en un archivo con extensión WAV utilizando diferentes filtros FIR. Se visualiza la propia señal en el dominio del tiempo y de la frecuencia, la respuesta al impulso del filtro y el su espectro.

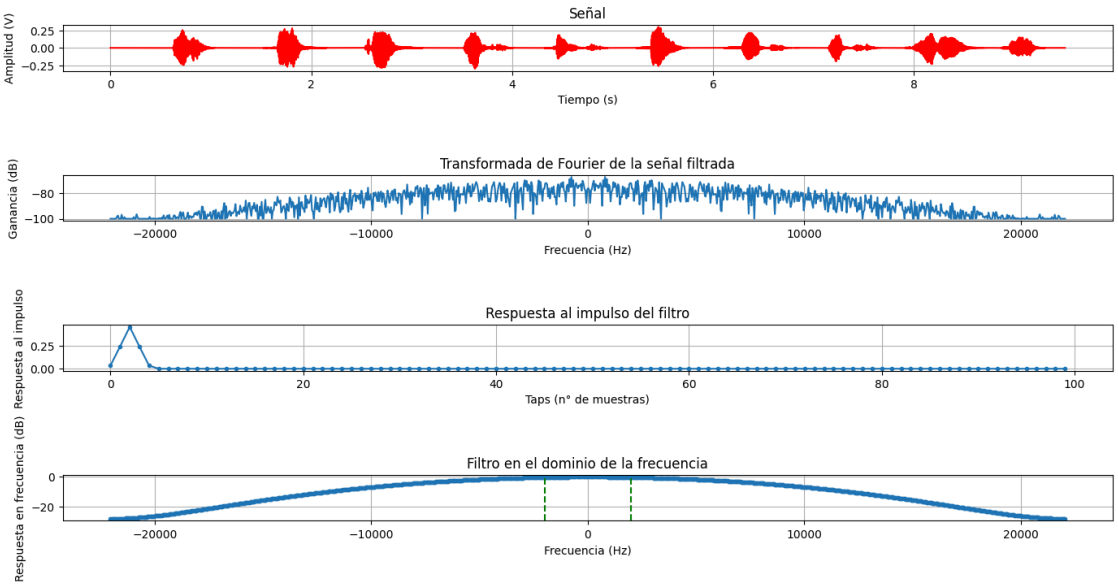
Entonces, comenzamos cargando la señal, la cual se normaliza y se escala para ajustar su amplitud. Se configura el tipo de filtro: pasa bajas, altas, pasa banda o suprime banda dependiendo de los parámetros *use_low_cut_off* y *use_high_cut_off*. Se indican también parámetros como el rango de *taps*, el incremento entre dichos *taps*, las frecuencias de corte y el tipo de ventana que se va a usar para aplicar el filtro (en este caso particular, está puesto a *hamming*, pero esta vez es un parámetro, como ya se indicó).

Se itera entonces sobre distintos *taps,* empezando desde el mínimo y hasta alcanzar el máximo, siguiendo el incremento indicado. Por cada *tap* se calculan los coeficientes del filtro según los parámetros indicados, y se obtienen:

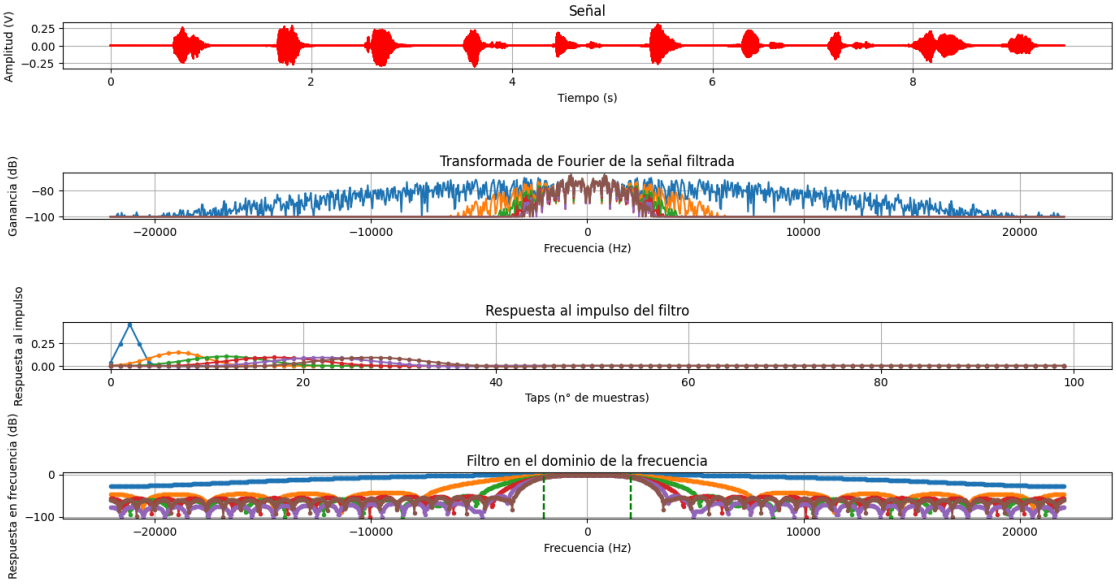
- La respuesta en frecuencia del filtro.
- La señal filtrada (utilizando la función *signal.lfilter*).
- El espectro de la señal filtrada.
- La respuesta al impulso del filtro.

Finalmente, se *plotea* la señal en el dominio del tiempo y todos los resultados enumerados anteriormente. Como nota adicional, es posible ir escuchando la señal filtrada si se asigna como verdadero el valor de la variable *play_audio.*

A continuación se muestra la salida correspondiente a una iteración del código donde se carga la señal *hombre.wav*, y se utiliza un filtro pasa bajas configurado para atenuar frecuencias superiores a 2000 Hz, empleando una ventana de tipo *hamming*. El filtro opera con los *taps* (orden del filtro + 1) de entre 5 y 256, incrementándose en pasos de 10. En este caso, la configuración mantiene activada la opción de tener su banda de paso en las bajas frecuencias (*use_low_cut_off = True*) y su banda de rechazo en las altas frecuencia (*use_high_cut_off = False*). La frecuencia de corte a usar será entonces la indicada por *low_cutoff*, que es de 2000Hz, como ya se indicó.



Asimismo, tras cinco iteraciones se obtiene el siguiente resultado:



Tarea: Filtros IIR

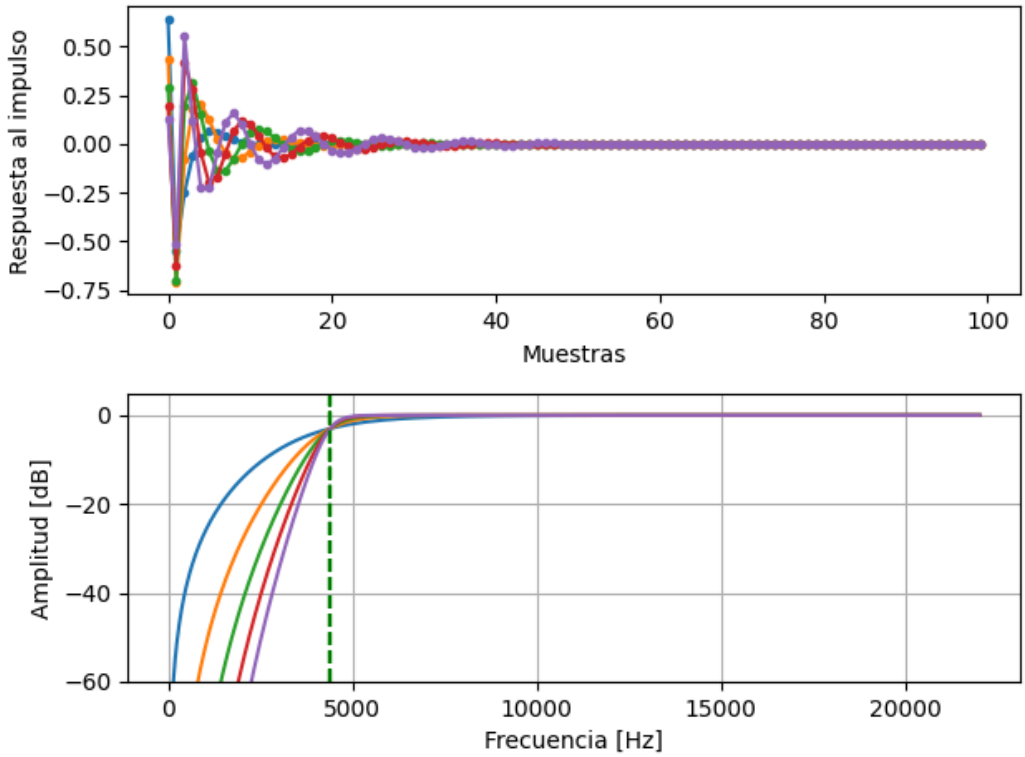
Analizar el fichero “filtro_IIR.py”, el cual genera múltiples filtros IIR con diferente número de etapas y calcula su respuesta al impulso en el dominio de la frecuencia. Generar un segundo fichero “filtro_IIR_con_audio.py” que cargue un fichero de audio “hombre.wav” o “ruido_blanco.wav”, aplique los distintos filtros generados (usar la función signal.lfilter), reproduzca el nuevo audio filtrado (2 segundos en torno al punto de mayor intensidad) y muestre en el gráfico el espectro en frecuencia del audio filtrado.

Filtro_IIR.py genera filtros IIR (*Infinite Impulse Response*) variando su orden y analiza sus respuestas tanto en el dominio del tiempo como en el de la frecuencia. En las líneas 19-34 se especifican los parámetros de configuración, como la frecuencia de muestreo (44100 Hz en este caso), el rango de orden del filtro (mínimo, máximo e incremento), las frecuencias de corte (frecuencia_corte_baja y frecuencia_corte_alta) y el uso de cortes definidos por las variables booleanas *usar_corte_bajo* y *usar_corte_alto*. Según estas combinaciones, el filtro puede comportarse como pasa bajas, pasa altas, pasa banda o elimina banda. Además, se define el tipo de filtro entre opciones las opciones Butterworth, Chebyshev de tipo 1 o 2, elíptico o Bessel, y para los de tipo Chebyshev y elíptico, se configuran los parámetros de ondulación (rp) y atenuación (rs) (*ripple* en la *passband* y en la *stopband* respectivamente).

En el bloque principal, se utiliza un bucle para generar filtros con órdenes que varían dentro del rango definido. Dependiendo del tipo de filtro y los parámetros de corte, se calculan los coeficientes del filtro con la función *signal.iirfilte*. Para cada filtro, se obtiene la respuesta en frecuencia utilizando *signal.freqz*, y se convierte a escala logarítmica en decibelios. Asimismo, se genera la respuesta al impulso unitario mediante la función *signal.lfilter*. Finalmente, el script dibuja dos gráficos por cada orden de filtro: uno de la respuesta al impulso (amplitud frente a muestras) y otro de la respuesta en frecuencia (amplitud en dB frente a frecuencia).

A continuación se adjuntan 5 iteraciones del script utilizando un filtro Butterworth pasa altas, con una frecuencia de muestreo de 44100 Hz y una

frecuencia de corte de 4400 Hz. El orden del filtro varía de 2 a 10 en incrementos de 2, lo que permite observar cómo cambia la respuesta del filtro tanto en el dominio del tiempo (respuesta al impulso) como en el dominio de la frecuencia (respuesta en amplitud en decibelios).



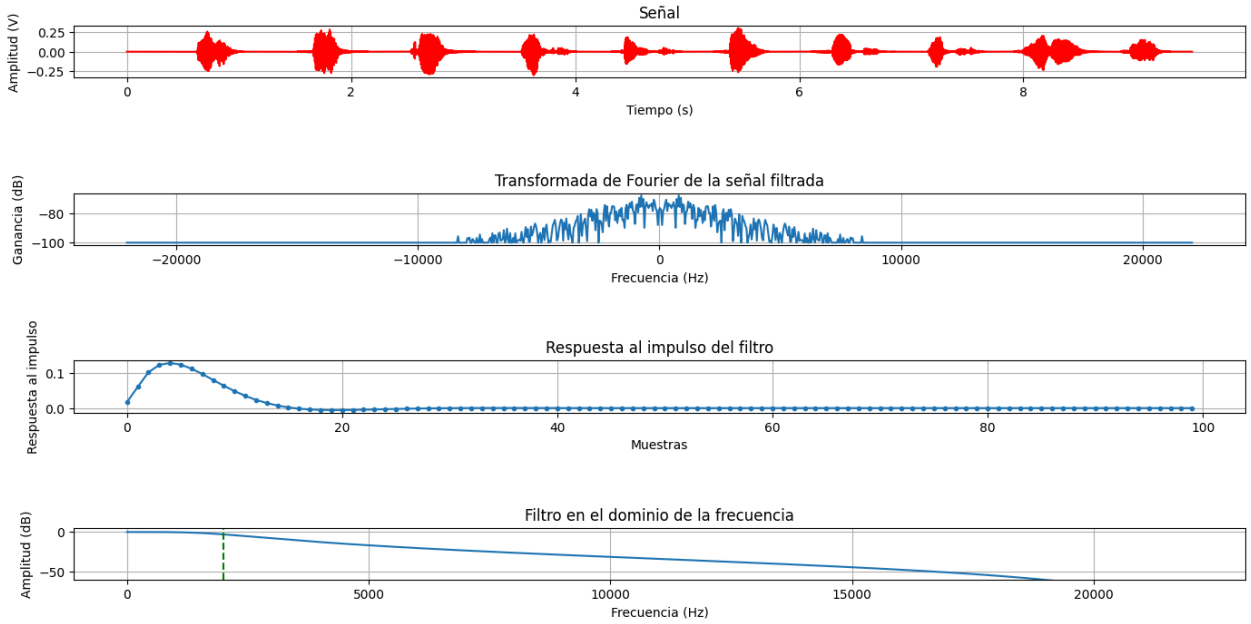
Haciendo uso dicho archivo, construimos **filtro_IIR_con_audio.py** que aplica diferentes filtros IIR (*Infinite Impulse Response*) a una señal de audio que viene dada en formato WAV, siguiendo las especificaciones del enunciado. Este código procesa y visualiza la señal en el dominio del tiempo y su espectro una vez filtrada, mostrando también la respuesta al impulso del filtro y su visualización en el dominio de frecuencia.

Comenzamos entonces cargando la señal de audio desde el archivo indicado, normalizándola y escalándola. A continuación, configuramos el tipo de filtro a utilizar: pasa bajas, pasa altas, pasa banda o suprime banda, dependiendo de los valores asignados a las variables *use_low_cut_off* y *use_high_cut_off*. Además, se establecen otros parámetros como las frecuencias de corte, el rango del orden del filtro, el incremento entre órdenes, y el tipo de filtro IIR, con opciones como Butterworth, Chebyshev, elíptico, o Bessel. Para los filtros de tipo Chebyshev y elípticos, se puede definir también el *ripple* en la banda de paso o la banda de atenuación (*passband* y *stopband* respectivamente)

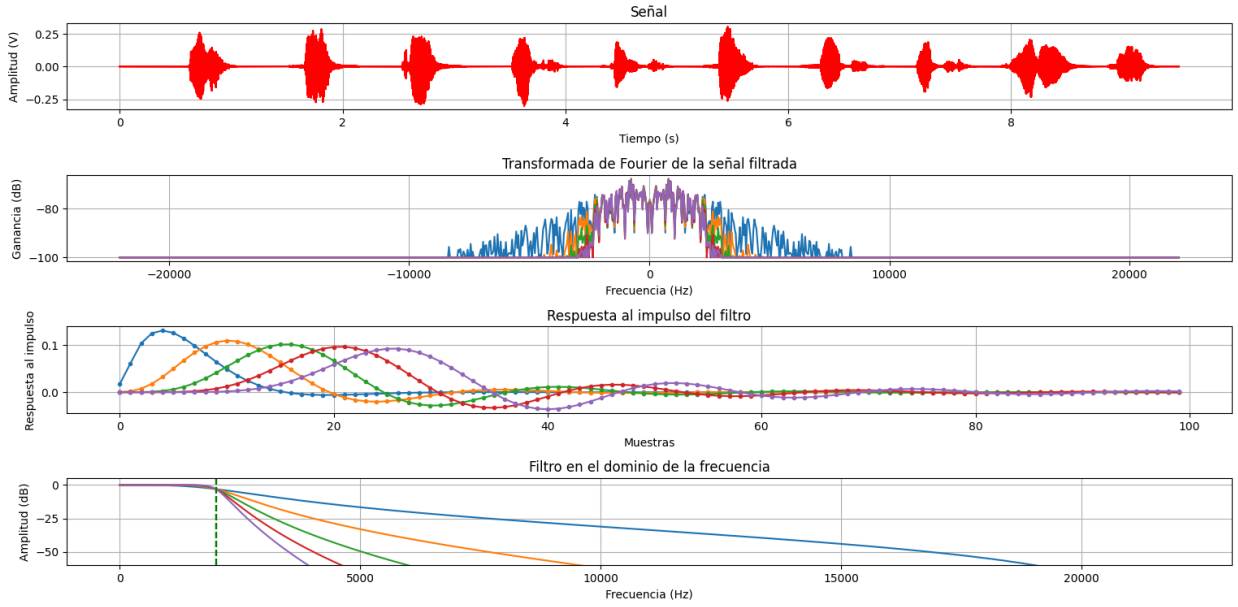
Luego, se itera sobre diferentes órdenes de filtro, comenzando desde el orden mínimo configurado y avanzando hasta el máximo, siguiendo el incremento definido. Para cada orden del filtro, se realiza lo siguiente: Se calculan los coeficientes del filtro utilizando la función *signal.iirfilter*, en función de las frecuencias de corte y el tipo de filtro seleccionado. Se obtiene la respuesta en frecuencia del filtro en escala logarítmica (para poder considerar su amplitud en decibelios) y se filtra la señal de audio usando la función *signal.lfilter*. Seguidamente, calculamos el espectro de la señal filtrada haciendo uso de la FFT, y se genera la respuesta al impulso del filtro. Finalmente, se grafica la señal original en el dominio del tiempo, el espectro de la señal filtrada, la respuesta al impulso del filtro y la respuesta en frecuencia del filtro en decibelios. Si la variable *play_audio* está fija a verdadero, se reproduce la señal filtrada después de cada iteración.

A continuación, se presenta la salida de una iteración del programa que utiliza un filtro pasa bajas Butterworth configurado para atenuar frecuencias superiores 2000 Hz. En este caso, el programa se ejecuta con un rango de órdenes de filtro de entre 2 y 10, avanzando en incrementos de 2. Se encuentra activada la

opción para tener una banda de paso en las bajas frecuencias (*use_low_cut_off* = True) y se desactiva la banda de paso en las altas frecuencias (*use_high_cut_off* = False); es decir, estamos considerando un filtro pasa bajas (como ya se había mencionado previamente).



Tras cinco iteraciones del código, obtenemos el siguiente *plot*:



Tarea: Procesado de la voz

Realizar un pequeño sistema de detección de propiedades de la voz ("procesado_de_la_voz.py") que abarque los siguientes conceptos:

- **Detección de actividad de voz: Este módulo deberá, dada una señal de voz (mujer.wav, hombre.wav o sintetico.wav), detectar las zonas de silencio y de actividad de voz. Para ello, habrá que definir un umbral de energía a partir del cual se determine que estamos ante presencia de señal. La energía se calculará como la suma de las muestras al cuadrado dentro de una ventana.**
  - **Longitud de la ventana: 2048 muestras.**
  - **Solapamiento entre ventanas: 75% (nos desplazamos 512 muestras).**

- **Detección de sonoridad: Para cada una de las ventanas calculadas se evaluará la tasa de cruces por cero. Se establecerá el umbral a partir del cual se puede determinar que un sonido es sordo o sonoro. Con este criterio, más el de detección de señal, se propondrá un segmentador de señal que establezca las zonas sordas y sonoras.**

- **Detección de frecuencia fundamental f0: Aprovechando el enventanado de la etapa anterior, se realizará un detector de frecuencia fundamental basado en cepstrum. Los valores de frecuencia se darán sólo si el detector de actividad de voz decide que existe presencia de señal y esta se ha clasificado en sorda o sonora. Los posibles valores para la frecuencia fundamental estarán comprendidos entre 75 y 400 Hz.**

El script a entregar está estructurado en tres partes, una con las funciones a utilizar (donde se corresponderá una por cada uno de los puntos mencionados en la tarea), otra con los parámetros y la carga de los datos, y por último una con las gráficas. Sabiendo que el script está estructurado de esta manera, empezaremos comentando primero la sección de parámetros y carga de datos, para luego cubrir las funciones y por último las gráficas.

Entonces, en la sección de parámetros y carga de datos (que se desenvuelve entre la línea 213 y la línea 265) obtenemos la señal a utilizar de un fichero con extensión wav y la frecuencia a la que se muestreó la misma. Adicionalmente se estandariza la señal (se sustrae su media y se divide entre su máximo). Luego, se define el umbral de energía a utilizar – qué es y para que se usa se explicará luego –, la longitud (en número de muestras) de la ventana de análisis que usará para aventanar la señal y poder procesarla, y el porcentaje (entre 0 y 1) de solapamiento entre dichas ventanas. Con estos dos últimos datos obtenemos el desplazamiento (en número de muestras) de las ventanas y el número de ventanas a usar. Después crearemos un vector de tiempos equiespaciados por el periodo de muestreo para poder dibujar la señal en el dominio del tiempo, y un vector de "tiempos de ventanas", donde cada punto indicará el instante en el que acaba cada ventana; nos servirá también para dibujar los análisis hechos con ventanas. Finalmente, almacenamos en variables los valores de retorno de las funciones que detallaremos a continuación.

Cabe destacar que otros parámetros, como el umbral mínimo para detectar cruces por cero, el umbral para clasificar una señal como sonido sonoro o sordo, y el rango de frecuencias en las que buscar la frecuencia fundamental se pasan como argumentos con valor por defecto en sus funciones correspondientes. De esta forma nos ahorramos el tener que definirlos explícitamente en esta sección.

Ahora detallaremos la sección de funciones a utilizar, repasándolas punto a punto:

Detección de actividad de voz:

La firma de la función que cubre este punto es la siguiente:

```
def deteccion_act_voz(senal           : np.ndarray,
                      umbral_energia  : float,
                      long_ventana    : int,
                      despl_ventana   : float) -> tuple[np.ndarray]:
```

Esta función empieza calculando el número de ventanas que se van a utilizar para analizar la señal. Luego, inicializa dos listas vacías: Una de números flotantes y otra de booleanos. En la primera lista se almacenará la energía de la señal por ventana y en la segunda si ha habido activación de la voz o no por ventana.

Luego, se itera por el número de ventanas que hay. En cada iteración, se obtiene la señal en la ventana correspondiente y se calcula su energía como la integral de la señal al cuadrado; dicha energía se añade al vector de energías.

Seguidamente, se analiza si la energía de la señal en la ventana supera el **umbral de energía**. Este fue el umbral que se mencionó antes y está parametrizado (en este caso la función lo toma como argumento). Si dicho umbral se supera, se considera que ha habido activación de la voz, y si no, se considera que ha habido silencio. Si ha habido activación se añade *True* a la lista de booleanos, en otro caso se añade *False*.

Luego de iterar se normalizan las energías entre 0 y 1, y se devuelve una tupla de tres elementos donde el primero es el vector de activaciones de la voz normalizadas, el segundo es el vector booleano de activaciones de la voz y el tercero es el "umbral normalizado" (simplemente el umbral adaptado para las energías normalizadas). Este "umbral normalizado" se devuelve para poder dibujarlo luego.

Detección de sonoridad:

La firma de la función asociada a este punto es la siguiente:

```
def deteccion_sonoridad(senal           : np.ndarray,
                        long_ventana    : int,
                        despl_ventana   : float,
                        umbral_energia  : float = 0.01,
                        umbral_min      : float = 0.001,
                        umbral_sordo    : float = 0.06) ->
tuple[np.ndarray]:
```

Esta función se ha obtenido de las funciones básicas, y se corresponde a la definida en el fichero **sonido_sonoro_vs_sordo.py**.

Al igual que antes, esta función comienza calculando el número de ventanas a usar, para luego inicializar tres vectores: Uno de números enteros donde se almacenarán los cruces por cero de la señal por ventana, y otros dos de valores binarios (0 o 1) que almacenan si la señal se clasifica como un sonido sonoro o sordo por ventana respectivamente. Todos estos vectores se inicializan a 0.

Una vez hecho esto, procedemos a obtener el vector booleano de activaciones de la voz que se mencionó en la función anterior (para ello usamos dicha función).

Luego, iteramos por cada una de las ventanas y obtenemos la señal en cada una de ellas. Por cada señal enventanada se estudia si ha habido activación de la voz y, si la ha habido, se empieza calculando el número de cruces por cero que ha tenido la función enventanda. Para ello, se empieza obteniendo un vector de enteros que sólo pueden ser -1, 0 o 1 y representan el signo de cada muestra de la señal en la ventana. Luego, se obtiene la diferencia *signo[i+1] - signo[i]* para cada valor del índice "*i*" en el vector de signos, y seguidamente se obtiene su valor absoluto, dejándonos con un vector de valores binarios (0 o 1). Es entonces claro que ocurrirá un cruce por cero si y sólo si el valor absoluto de la diferencia de signos es mayor a 0; por tanto obtenemos los índices de estos elementos.

Una vez obtenidos estos índices, comprobamos si el valor de la función en ellos supera un **umbral mínimo**. Si no lo supera, se considera entonces que ha sido una pequeña oscilación de la señal y no se considera como un cruce por cero significativo. Rescatamos entonces sólo los índices de los cruces por cero significativos.

Finalmente, obtenemos el *ratio* del número de cruces significativos (la longitud del vector de índices anterior) por el número de muestras de la señal enventanada. Dicho valor se añade al primer vector mencionado.

Luego, armados con este número comprobamos si es mayor a 0 (es decir, si ha habido algún cruce por cero significativo en la ventana) y, si lo ha habido, se estudia si el *ratio* calculado supera el **umbral sordo**. Si lo supera, se clasifica la señal como un sonido sordo (los sonidos sordos presentan muchos más cruces por cero) y si no como un sonido sonoro. Si no ha habido activación de la voz todos estos cálculos mencionados se obvian y simplemente se añade un cero al vector de cruces por cero.

Finalmente se devuelve una tupla de tres elementos correspondientes a los tres vectores inicializados a cero al comienzo de la función.

Detección de frecuencia fundamental:

Para este punto hemos usado dos funciones, la firma de la primera es:

```
def calcular_f0_cepstrum(senal : np.ndarray,
                         freq_muestreo : float,
                         min_frequency : float,
                         max_frequency : float) -> float:
```

Esta función hace uso del código definido en el script **frecuencia_fundamental.py**, y su funcionamiento consiste en lo siguiente: Primero, se define una ventana de análisis (en concreto, una ventana de hamming), y se ventanea la señal (se hace el producto de la señal por la ventana) para mejorar su comportamiento en los extremos.

Seguidamente, se calcula el espectro de la señal ventaneada mediante el uso de la FFT. Posteriormente, se calcula el logaritmo del valor absoluto del espectro (más una cierta cantidad infinitesimal *epsilon* para evitar problemas con el logaritmo de 0). Con la inversa del logaritmo del espectro obtenemos el **cepstrum**, que es otro espacio de representación de la señal útil para hallar propiedades de la misma.

Una vez encontrado el cepstrum, hallamos el rango de **quefrecuencias** (frecuencias en el cepstrum) en el que buscar el pico cepstral. Dicho pico nos

revelará cuál es la frecuencia fundamental. Como se comentaba anteriormente, la quefrecuencia mínima y máxima se calculan como el cociente entre la frecuencia de muestreo de la señal y la máxima ó mínima frecuencia respectivamente donde queremos hallar la frecuencia fundamental de dicha señal.

Una vez hallados estos valores, obtenemos la quefrecuencia en la que se encuentra el pico cepstral en el cepstrum acotado a este intervalo. El cociente de la frecuencia de muestreo por ese valor será la frecuencia fundamental de la señal.

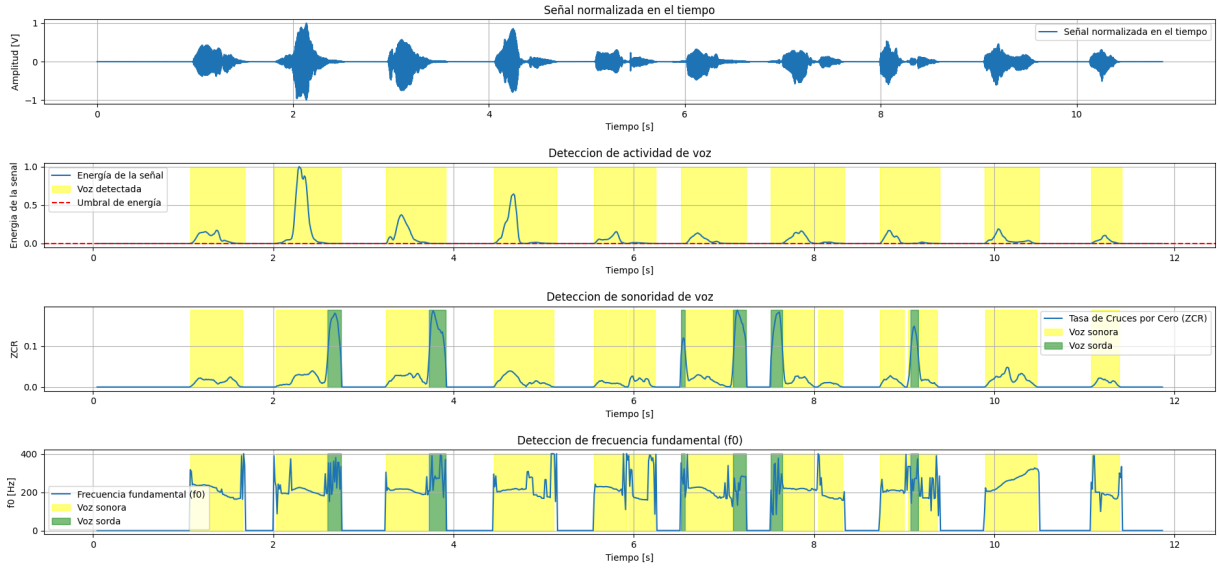
Luego, la firma de la segunda función usada es:

```
def calcular_freq_fundamental_por_ventana(senal           : np.ndarray,
                                          freq_muestreo   : float,
                                          long_ventana    : float,
                                          despl_ventana   : float,
                                          umbral_energia  : float = 0.01,
                                          min_frequency   : float = 75,
                                          max_frequency   : float = 400) ->
np.ndarray:
```

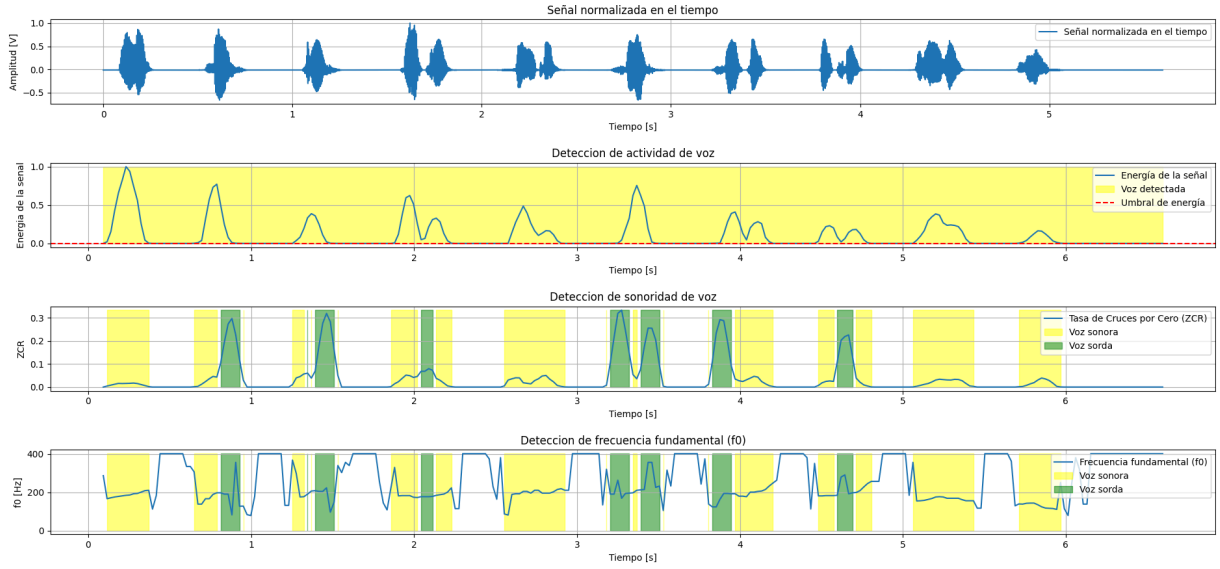
Que, de manera resumida, itera por las ventanas posibles, ventanea la señal por cada ventana y, si ha habido activación de la voz en la ventana, obtiene la frecuencia fundamental de la señal en dicha ventana y la añade a un vector. Si no ha habido activación de la voz, simplemente se añade un cero al vector mencionado previamente. La función entonces retorna este vector, que contendrá la frecuencia fundamental de la señal por ventana en las ventanas donde haya habido activación de la voz, y 0 en el resto de ventanas.

Finalmente, una vez cubiertas las funciones y, en consecuencia, todos los puntos de la tarea, podemos pasar a la sección de gráficas. En esta sección graficaremos la señal en el dominio del tiempo, su energía, su tasa de cruces por cero por ventana y su frecuencia fundamental por ventana. En las tres últimas gráficas incluiremos también una marca para especificar cuando ha habido detección de la voz, y – cuando corresponda – si la señal se clasifica como un sonido sonoro o sordo. Además, también incluiremos en la gráfica de la energía de la señal una línea horizontal a la altura del umbral de energía usado.

Las gráficas obtenidas entonces para el audio *mujer.wav* son:



Y para el audio *sintetico.wav:*



Tarea: Asistente de voz

Descripción:

El asistente de voz para operaciones matemáticas elementales se concibió como una clase que tiene la capacidad de **recoger audio** de un usuario mediante el dispositivo de grabación de la máquina donde se ejecute, **transcribir** ese audio obtenido de la manera más fidedigna posible, **detectar y resolver** la operación matemática indicada (siempre que sea posible, indicando el caso en el que no lo sea), y **sintetizar** con una voz computacional la respuesta generada.

Para la función de recoger audio se ha utilizado la función *grabar_audio* definida en el fichero **Grabar_y_guardar_audio.py**, aplicando una ligera modificación para esperar a que el usuario presione la tecla "*Enter*" para empezar la grabación, y presione la tecla "*espacio*" para terminarla. Esta función devuelve el audio grabado en un array de numpy.

Luego, para la transcripción del audio se utilizó la función *transcribir_voz*, definida en el fichero **Transcribir_voz_remotamente.py**. Esta función utiliza un modelo de *Deep Learning* (en concreto, un modelo reconocedor de voz de *Google*) para transcribir un audio dado como un array de numpy. La adaptación que se le hizo a esta función fue mínima, parametrizando el lenguaje a detectar y la frecuencia de muestreo del audio pasado como argumento. Cabe destacar que para poder llamar a esta función es necesaria una conexión a internet, pues el modelo está alojado en un servidor remoto al cual se le hace una petición.

Para la detección y resolución de operaciones matemáticas se utilizó sin ninguna modificación la función *resolver_operacion* definida en el fichero **Resolver_pregunta_de_matematicas.py**. Esta función toma como argumento una cadena con una pregunta matemática (por ejemplo: "*¿Cuál es el resultado de dividir 124 entre 3894?*"), y, haciendo uso de expresiones regulares, se extraen los datos clave de la pregunta, como los dos números que intervienen y la propia operación. Una vez se han extraído estos datos clave, se resuelve la operación y se genera un string de respuesta.

Finalmente, para la síntesis de la respuesta del asistente se tomó inspiración de la función *sintetizar_voz* del fichero **Sintetizar_voz.py**. No se utilizó exactamente esta función porque tuvimos dificultades con la librería, que es *pyttsx3*. Para la función implementada usamos una librería llamada*gtts*, que también permite la síntesis de voz usando un modelo local (no es necesaria la conexión a internet). Luego, con *gtts* se genera la síntesis de la voz en formato *mp3* y se guarda temporalmente en el directorio de trabajo. Luego, se transforma la voz generada a un array de numpy, se reproduce con *sounddevice* y finalmente se elimina empleando utilidades del sistema operativo.

En el final del fichero **Asistente_de_voz.py** (que es donde se ha implementado la clase) se incluye un pequeño script que permite ejecutar al asistente. En este primero se da una bienvenida explicando el uso básico del asistente. Luego se da la pauta al usuario para decir su pregunta, seguidamente se transcribe la voz y se estudia si es una pregunta o si el usuario desea terminar la ejecución del asistente (esto último se hace consultando si el usuario ha incluido la palabra “salir” en la frase dicha); una vez transcrita la voz, se obtiene la respuesta del asistente a la pregunta (si es que se ha logrado detectar una pregunta) y por último se sintetiza dicha respuesta. Este proceso se repite en bucle hasta que el usuario indique la terminación del programa (en tal caso se enseñará un mensaje de agradecimiento) o bien hasta que se haga una terminación manual.

Instrucciones de uso:

Para ejecutar el código del asistente de voz es necesario modificar de manera acorde la construcción del asistente en la línea 155: Empezando por escoger una frecuencia de muestreo adecuada (usualmente es 44100 Hz, pero para sistemas operativos Unix como *Pop_OS!*, detectamos que es necesaria una frecuencia de muestreo de 48000 Hz por cuestiones de las librerías que sostienen el código y los *drivers* del sistema operativo). Luego, habrá que escoger el idioma en el que se van a esperar las preguntas, y en el que se va a sintetizar la voz (esto se indica como un string. Por ejemplo: "*es-ES", "en-US"*, entre otros…), y el género de la voz sintética (actualmente la librería sólo soporta voces femeninas, por lo que es irrelevante la elección). Por último, habrá que escoger el índice del dispositivo de grabación de audio que se va a usar (este índice se puede hallar tal y como indica el comentario en la línea 26: `"* device_index : Número que representa al dispositivo de grabación de audio. Para ver los indices, ejecutar print(sd.query_devices()) y buscar 2 in 0 out"`).

Finalmente, este código se debe ejecutar como *super usuario (sudo)* en sistemas Unix. No lo hemos probado en sistemas Windows o MacOS, pero sospechamos que en estos será necesario ejecutar como administrador.

Como nota final, están comentadas las líneas de la 97 a la 110. Esto porque queríamos centrarnos en verificar que los ficheros *.mp3* se guardaban adecuadamente, y porque en la máquina con el sistema *Pop_OS!* la reproducción de audio no se hacía correctamente . El código sigue funcionando, sólo que no se escuchará la respuesta del asistente directamente, sino que para escucharla será necesario abrir el fichero *mp3* generado (este fichero se llama *.temp_sint.mp3*). Las líneas mencionadas pueden descomentarse, o se puede escuchar la pista de audio generada como ya se ha comentado.

Códigos solicitados y ficheros de audio

Todo el código implementado para realizar los ejercicios de esta memoria, junto con el código de apoyo suministrado al cual se hace referencia, puede ser encontrado en el siguiente repositorio de GitHub. En particular, el código se encuentra en la carpeta *src*, la presentación hecha en el curso en la carpeta *presentación*, y esta memoria en formato pdf y el link al documento de Google Docs donde se creó en *docs*.