Universidad Técnica Federico Santa María

DEPARTAMENTO DE INGENIERIA ELECTRÓNICA

ELO 314 - Laboratorio de procesamiento Digital de Senales

Señales de Audio en MatLab

Estudiante	**ROL**
Rodrigo Graves	201621009-1
Ricardo Mardones	201621036-9

Paralelo: **1**

Profesor

Gonzalo Carrasco

Ayudante

Jaime Guzman

Fecha : 15 de mayo de 2021

Índice

Índice de figuras

Índice de cuadros

1. Archivos de Audio y Visualización

1.1. Extracción y Guardado de Segmentos de Audio

Para esta sección se utiliza el script de MATLAB `rp1.m`, el cual se adjunta en la entrega.

Se carga el archivo *besh.wav* como vector en MATLAB y se extraen las porciones de la vocal /ϵ/ y la fricativa /sh/. En la figura 1 se muestra el gráfico de la señal *besh.wav* y extracción de las secciones de audio de interés.

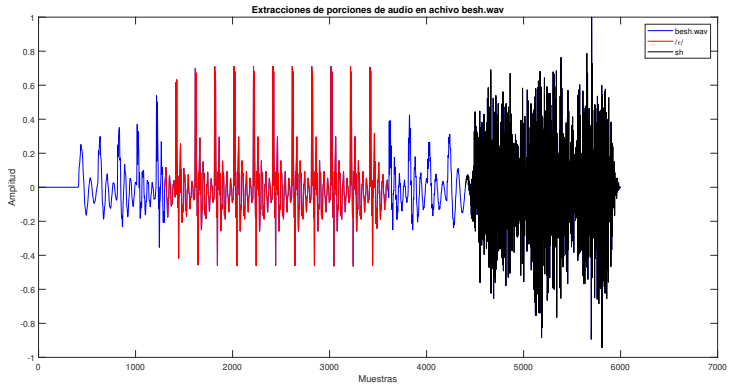

Figura 1: Audio *besh.wav* y extracción de la vocal /ϵ/ y fricativa /sh/.

Se guarda además el segmento de audio correspondiente a la vocal /ϵ/ como *Lab2p1_vocal.wav* y se adjunta a la entrega.

1.2. Función en MATLAB para la selección gráfica de un segmento de Audio

Se escribe la función en MATLAB `select_wav` con la cual se selecciona con el mouse un segmento de una señal, se copia dicho segmento en un vector y se guarda en un archivo de audio. La función descrita se muestra a continuación:

```
function select_wav(input_name, output_name)
    [signal, Fs] = audioread(input_name);
    figure;plot(signal)
    xlabel('Muestras'); ylabel('Amplitud');
    title('Seleccione 2 instantes para guardar audio')
    [x,~] = ginput(2);
    close
    wav = signal(round(x(1)):round(x(2)));
    audiowrite(output_name,wav,Fs)
end
```

Se utiliza la función para extraer el segmento de la vocal /ϵ/, quedando guardado en el archivo *lab2p1_segmento vocal*. Se adjunta dicho archivo a la entrega.

2. Efectos de Audio

2.1. Implementación de Overdrive en MATLAB

Se implementa en MATLAB un overdrive simple dado por la siguiente expresón:

$$x' = G_i x \tag{1}$$

$$y' = \begin{cases} \beta x' + \text{sign}(x')(1-\beta)\alpha & \text{si } x' \geq \alpha \\ x' & \text{si } x' < \alpha \end{cases} \tag{2}$$

$$y = G_o y' \tag{3}$$

La función implementada se muestra a continuación:

```
function  y = overdrive(file_name, alpha, beta, Gi,G0)

    [signal, Fs] = audioread(file_name);

    %separar canales
    ch1 = signal(:,1);
    ch2 = signal(:,2);

    %x2 para cada canal
    x2_ch1 = Gi*ch1;
    x2_ch2 = Gi*ch2;

    %y2 para cada canal lleno de ceros inicialmente
    y2_ch1= zeros(length(signal),1);
    y2_ch2= zeros(length(signal),1);

    %y de salida
    y = [y2_ch1 , y2_ch2];

    for i  = 1:length(signal)
        i

        %canal 1
        if abs(x2_ch1(i)) >= alpha
            y2_ch1(i) = beta*x2_ch1(i)
                        +  sign(x2_ch1(i))*(1-beta)*alpha;
        else
            y2_ch1(i) = x2_ch1(i);
        end
        %canal 2
        if abs(x2_ch2(i)) >= alpha
            y2_ch2(i) = beta*x2_ch2(i)
                        +  sign(x2_ch2(i))*(1-beta)*alpha;
        else
            y2_ch2(i) = x2_ch2(i);
        end
    end
    y = [y2_ch1 , y2_ch2];
end
```

Posteriormente se aplica dicha función al archivo de audio `gtr_jazz.wav` obteniendo el audio `gtr-ov_a02b005Gi1Go1.wav` el cual utiliza $\alpha = 0{,}2, \beta = 0{,}05$ y $G_i = G_o = 1$. Se adjunta el audio resultante a la entrega.

Con respecto a la percepción auditiva, se pecibe una clara distorsión en el sonido. Lo anterior se le atribuye a que la función overdrive es no lineal por lo tanto cambia el contenido en frecuencias de la señal.

Se pide aplicar el efecto a una señal de audio adecuada para una comparación visual de lo que hace el efecto sobre la señal. Se considera `gtr_jazz.wav` un buen audio para lo anterior, pues debido a sus claros cambios de intensidad en el sonido se pasa de zona lineal a no lineal con frecuencia.

En la figura 2 se muestra el audio *gtr-.wav* original y luego de aplicar overdrive. Se nota claramente que la distorsión cumple su cometido, añadiendo cambios bruscos en la señal, lo cual puede interpretarse como que se agregaron componentes de alta frecuencia.

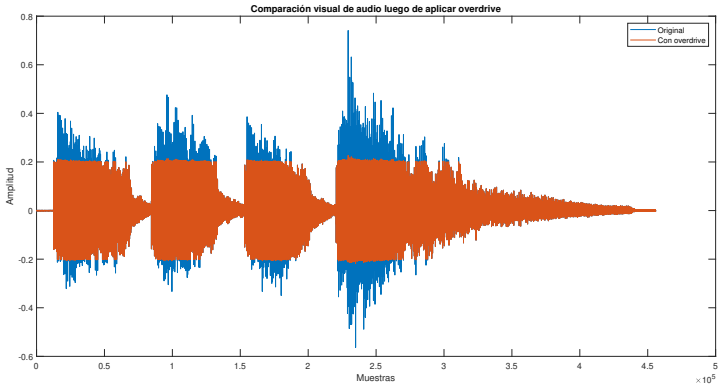

Figura 2: Audio *gtr-.wav* original y luego de aplicar overdrive.

Posteriormente se analiza que ocurre al variar los parámetros de la función, tomando $\alpha = 0{,}1$ y $G_i = 3$. La relación salida/entra de este overdrive y el original se muestra en la figura 3.

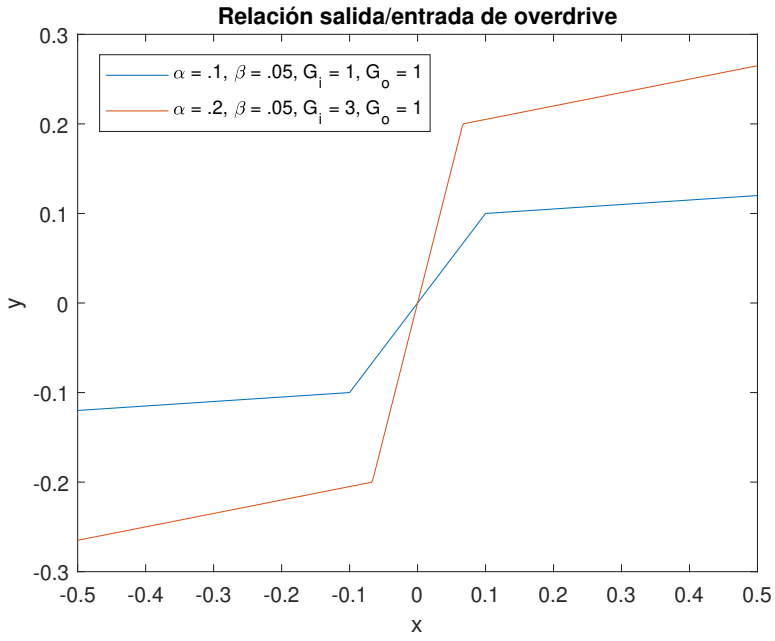

Figura 3: Relación Salida/Entrada de overdrive con distintos parámetros.

A partir de la figura 3 se aprecia que el overdrive con los nuevos parámetros genera una señal con menos energía que el anterior, pues cada punto azul es menor en magnitud a su punto rojo correspondiente para algún valor x de entrada. Lo anterior podría corregirse variando G_o. Por otro lado vemos que con la nueva combinación de parámetros el cambio de pendiente es aparentemente menos abrupto, por lo que se esperaría una percepción auditiva menor en el overdrive. Además, la zona lineal aumenta, lo cual es otra razón para que la percepción auditiva de la distorsión producto al overdrive sea menor.

Los parámetros del modelo de overdrive utilizado pueden interpretarse de la siguiente forma:

- G_i: Ganancia de entrada. la cual sirve para balancear que la señal x' "se mueva" por los intervalos de no linealidad. Si G_i es muy bajo no habrá distorsión.
- α: Umbral de no linealidad. Si α es muy alto no hay distorsión pues x estaría en zona lineal. De la misma forma si α es muy pequeño no se aprecia distorsión.
- β: Relacionado a que tan abrupto es el cambio de pendiente en el módulo no lineal (ecuación de y'). Afecta a la percepción de la distorsión.
- : G_o: Ganancia de salida. Está relacionada con el "volumen" de la señal de salida. No genera distorsión.

2.2. Implementación de un Delay Multi-Tap

Una implementación de un delay multitap corresponde a la siguiente expresión

$$y[n] = \sum_{k=1}^{N} b(k)x[n - k \cdot M] \tag{4}$$

Donde:

- x: Señal de entrada
- N: Número de etapas de retardo.
- M: Número de muestras equivalentes a la longitud de cada retardo. Dado T un retardo por etapa deseado en s y una T_s periodo de muestreo, M puede obtenerse como:
$$M = \frac{T}{T_s}$$
- $b(k)$: Ganancia de la k-ésima etapa.

En primer lugar se solicita aplicar un delay multi-tap de 4 etapas (N) de longitud 125 ms (T) y ganancia constante $b(k) = 0{,}35$ al archivo *gtr-jazz.wav*.

Auditivamente se nota que el efecto de repetición de la señal a modo de "eco", sin embargo, el hecho de que $b(k)$ se mantenga constante hace que el sonido sea antinatural. El archivo de audio obtenido corresponde a *gtr-jazz_delay-multitap.wav*, el cual se adjunta en la entrega

La función creada para implementar el delay multi-tap en MATLAB se muestra a continuación:

```
function [y,Fs] = delay_multitap(audiofile,outputname, N, M, b)

    [x,Fs] = audioread(audiofile);

    x_r = x(:,1); x_l = x(:,2); %Separacion por canales
    L = N*M+1; %Largo necesario para implementar buffer

    buff = zeros(L,1);        %Inicializacion buffer
    y_r = zeros(length(x),1);

    for n = 1:length(x)
        buff_start = mod(n-1,L)+1;   %Dir de inicio de buffer circ
        buff(buff_start) = x_r(n);   %Actualizacion de buffer

        for k = 1:N
            y_r(n)= y_r(n)+b(k)*buff(mod(L+buff_start-k*M-1,L)+1);
            %Aplicacion de filtro
        end
    end

    buff = zeros(L,1);        %Inicializacion buffer
    y_l = zeros(length(x),1);

    for n = 1:length(x)
        buff_start = mod(n-1,L)+1;   %Dir de inicio de buffer circ
        buff(buff_start) = x_l(n); %Actualizacion de buffer

        for k = 1:N
            y_l(n)= y_l(n)+b(k)*buff(mod(L+buff_start-k*M-1,L)+1);
            %Aplicacion de filtro
        end
    end

    y = [y_l y_r];
    audiowrite(outputname,y,Fs);
end
```

Se decide que *gtr-jazz.wav* es una buena señal para mostrar el efecto del delay muli-tap debido a los claros ecos escuchados en la señal de audio resultante. El gráfico resultante se muestra en la figura 4.

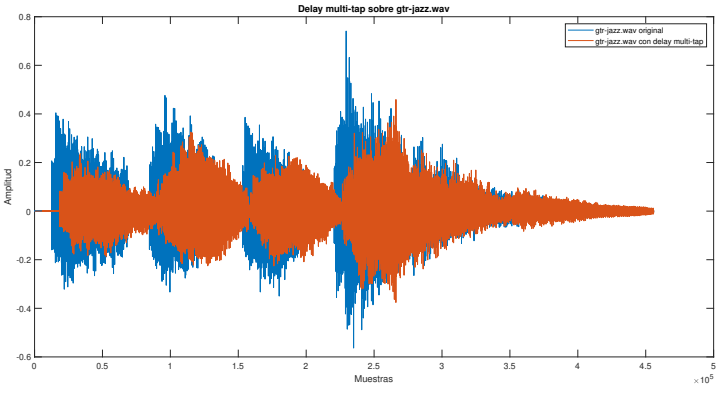

Figura 4: Audio *gtr-jazz.wav* original y luego de aplicar delay multi-tap.

Curiosamente no se logra apreciar el eco de forma visual, pero si el efecto de filtro pasabajos que tiene un filtro de media móvil de ganancias constantes. Lo

anterior se le atribuye a que el tiempo de retardo es muy bajo para la duración de los "segmentos" de la señal (arpegios).

Posteriormente se cambia $N = 10$, $T = 250$ ms y $b(k) = 0{,}35^k$ y se vuelve a aplicar el efecto a *gtr-jazz.wav*, obteniendo *gtr-jazz_delay-multitap2.wav*, el cual se adjunta en la entrega. Auditivamente se siente un decaimiento mucho más natural del eco. Una comparación gráfica se muestra en la figura 5 , siendo notoria una mayor atenuación en la señal que con ganancias constantes.

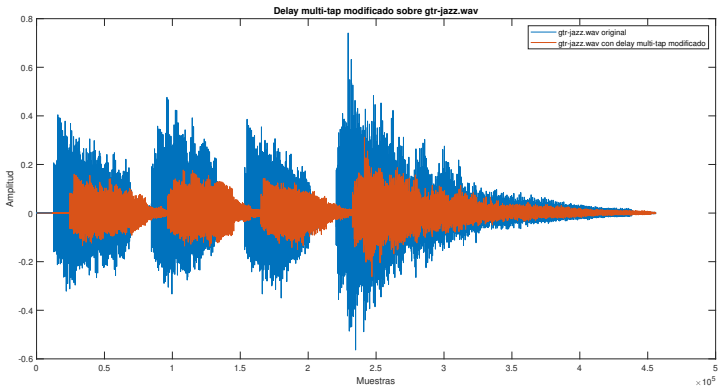

Figura 5: Comparación entre audio *gtr-jazz.wav* original y luego de aplicar delay multi-tap con ganancia $b(k) = 0{,}35^k$.

Finalmente se modificó la frecuencia de muestreo en el audio *gtr-jazz.wav* a 16 kHz usando el comando `resample`. El audio a la nueva frecuencia de muestreo se guardo en el archivo *gtr-jazz_resample-16.wav* el cual se adjunta con la entrega. Auditivamente no se escucha mayor diferencia con el audio original. Los parámetros elegidos para lograr el mismo efecto anterior corresponden a:

- N: Se mantiene en 10, pues las etapas no cambian.
- M: Al cambiar la frecuencia de muestreo este cambia. Como la frecuencia de muestro original (48 kHz), es el triple de la nueva (16 kHz), ahora M corresponde a un tercio de las muestras anteriores.
- $b(k)$: Se mantiene pues no se cambia la ganancia por etapa.

El audio obtenido al aplicar los nuevos parámetros de delay sobre la señal a la nueva tasa de muestre se guarda en el archivo *gtr-jazz_resample-16_delay.wav*, el cual se adjunta en la entrega. No se perciben diferencias auditivas notorias con el audio a 48 kHz.

Con respecto a las ventajas y desventajas de aplicar resample se tiene:

- Ventajas: Menor tiempo de cálculo de la salida del filtro. En aplicaciones que tengan algún requisito de tiempo máximo para procesamiento puede ser útil.
- Desventajas: El cambio de frecuencia de muestreo puede provocar perdida de contenido espectral de interés.

3. Buffers Lineales y Circulares en Lenguaje C

3.1. Buffer lineal

1. Para lograr una linea de retardo se implementa un buffer lineal en lenguaje C, integrado a MATLAB mediante el uso de *s-function*.

   El siguiente código muestra la función en lenguaje C que implementa el buffer.

```
1
2  #define N (32000)
3  double retardo_lineal(double input) {
4      // Buffer lineal
5      static double buffer[N];
6
7      //Inicializacion del buffer
8      static int flag = 1;
9
10     if (flag == 1) {
11         for (int i = 0; i < N; i++) {
12             buffer[i] = 0;
13         }
14         flag = 0;
15     }
16
17     double output = buffer[N - 1];
18
19     for (int idx = N - 1 ; idx >= 1; idx--) {
20         buffer[idx] = buffer[idx -1];
21     }
22     buffer[0] = input;
23     return output;
24 }
```

   Donde la entrada a la función es una muestra actual n de la señal que se desea retardar, dato que se guarda en el buffer de N elementos, mientras que la salida retornada por la función corresponde a la primera posición del buffer, es decir el dato guardado n - N.

   Para verificar el correcto funcionamiento de este retardo lineal implementado se escoge el archivo de audio *sonidos_de_voz_16_8.wav* ya que por la manera en la que está distribuida la información en el tiempo en este archivo (viendo la forma de onda a graficar los datos del arhcivo) un retardo en un mensaje hablado es significativamente notorio. En este punto es necesario comentar la relación que existe entre el **tiempo de retardo** y la frecuencia de muestreo **Fs** asociado a las señales a trabajar, el inverso de esta frecuencia corresponde al periodo de muestreo , que al ser multiplicado por la cantidad de muestras que se desean almacenar en el buffer se obtiene el tiempo total de retardo provocado en la señal.

   La frecuencia de muestreo de la señal escogida es de 8 kHz, por lo que con un N de 32000 muestras se debiera obtener un retardo de 4 s como se muestra a continuación

$$T_{retardo} = \frac{1}{Fs} \cdot N = \frac{1}{8\ kHz} \cdot 32000 = 4\ s$$

Se crean los bloques necesarios para hacer uso de la *s-Function* creada, importar el archivo de audio y un scope para poder observar y comparar las señales obteniendo de esta forma las gráficas presentes en la figura 6

Cabe destacar que si bien la guía propone un retardo de $N = 100$, debido a la frecuencia de muestreo que posee el archivo de audio esta cifra provocaría un retardo despreciable por lo que se optó por un valor de $N = 32000$, tal como se mencionó antes.

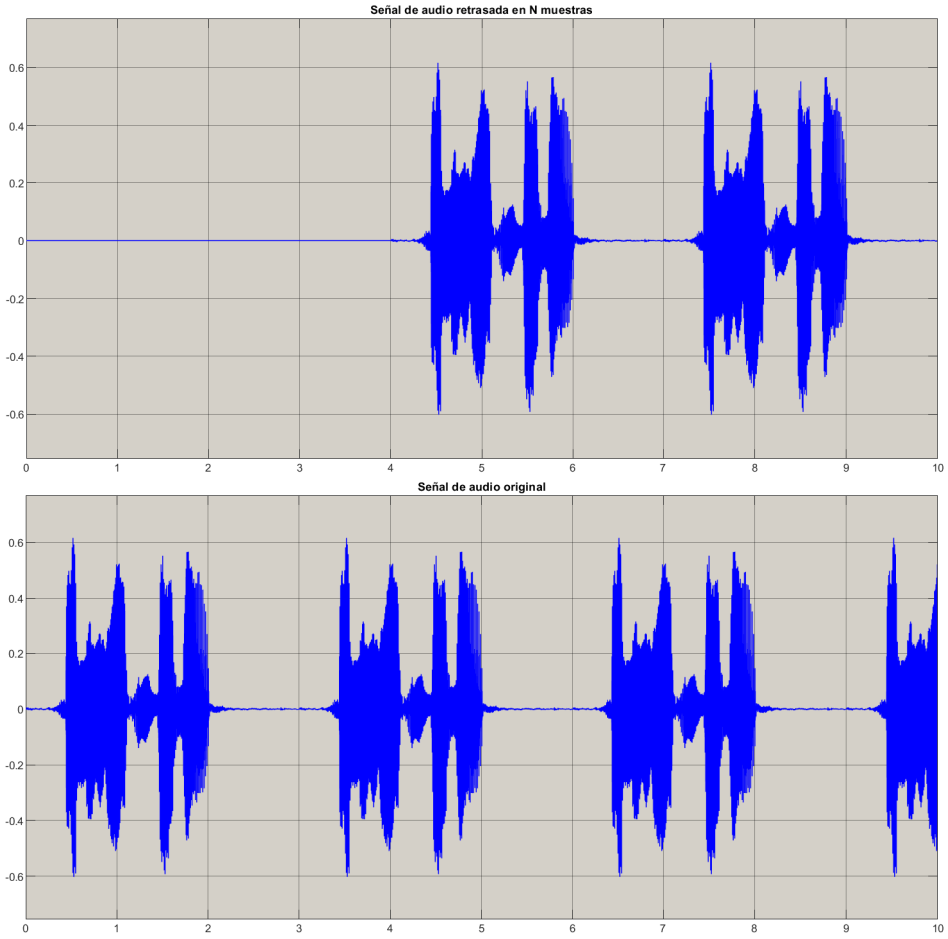

Figura 6: Gráfica del resultado de aplicar el retardo lineal implementado a una señal de audio.

En la figura anterior se puede ver que tal como se esperaba el buffer lineal implementado provoca un retardo de 4 s respecto de la señal de audio original desplazándola en 32000 muestras hacia la derecha.

2. De la misma manera que en el apartado anterior, se implementa una linea de retardo usando código en lenguaje C pero esta vez haciendo uso de un **buffer circular** en tiempo real, el código de implementación es el siguiente

```
1
2       #define n (32000)
3       double retardo_circular(double input) {
4
5
6
7       static double buffer[n]; //Crea buffer tama o n
8
9       //inicialica el "puntero" idx en la ultima posicion
10      static  int idx = n-1;
11
12      //Bloque para inicializar el buffer con ceros
13      static int flag = 1;
14
15      if (flag == 1)
16      {
17          for (int i = 0; i < n; i++)
18          {
19              buffer[i] = 0;
20          }
21          flag = 0;
22      }
23      //Fin inicializacion del Buffer
24
25      double output = buffer[idx]; //Lee dato del Buffer
26      buffer[idx] = input;  //Actualiza el dato en posicion idx
27      idx = ((idx+1)%n); //Actualiza posicion idx
28
29
30      return (output);
31      }
```

Al graficar la señal original y la que sufrió el retardo debido usando buffer circular son practicamente idénticas a las gráficas obtenidas con el buffer de tipo lineal ya que ambas implementaciones cumplen el objetivo que es provocar un retardo.

3. Para comparar la eficiencia de ambos métodos para retardo implementados, usando el modo *PROFILE* de simulink se estudian los tiempos de ejecución de la simulación que requieren ambas *s-Function* (la asociada a buffer llineal y a buffer circular respectivamente). De este modo se obtuvieron los datos que se presentan en los cuadros 1 y 2

N	Scope	From multimedia file	s-Function	Simulacion	Total
20	0,042	0,091	0,019	0,563	0,612
200	0,044	0,091	0,024	0,488	0,647
2000	0,045	0,099	0,079	0,482	0,705
20000	0,055	0,113	0,617	0,433	1,2185

Cuadro 1: Tiempos de ejecución en segundos de la simulación haciendo uso de buffer lineal.

N	Scope	From multimedia file	s-Function	Simulacion	Total
20	0,120	0,0860	0,0170	0,175	0,398
200	0,107	0,0860	0,0170	0,478	0,688
2000	0,126	0,0880	0,0180	0,475	0,707
20000	0,100	0,0800	0,0170	0,178	0,375

Cuadro 2: Tiempos de ejecución en segundos de la simulación haciendo uso de buffer circular.

De las tablas resumen anteriores se puede ver cómo varía la eficiencia en ejecución de los buffers lineal y circular implementados dependiendo del valor de N correspondiente al número de muestras que se deben almacenar en el buffer.

Cuando N tiene un valor de 20 muestras ambos buffers tienen un rendimiento similar cercano a los 20 ms.

Al pasar N por los valores siguientes de 200, 2000 y 20000 el tiempo que requiere la simulación para ejecutar el bloque del buffer circular se mantiene relativamente constante en el mismo valor anterior cercano a los 20 ms, mientras que el buffer lineal aumenta el tiempo requerido en una proporción similar al incremento en el valor de N haciendo notar que para implementaciones que requieran pocas muestras se pueden utilizar ambos tipos de buffer, lineal y circular indistintamente, pero que si el contexto requiere almacenar grandes cantidades de datos en el buffer, el de tipo circular es claramente superior al momento de cumplir con esa labor. Esto se debe a que en esencia el buffer lineal requiere mover todos los datos dentro de las posiciones del arreglo en el que esta implementado, recorriendo dicho arreglo mediante un ciclo iterativo (`for`, por ejemplo), por lo que si este buffer tiene un tamaño considerable el ciclo iterativo encargado de recorrerlo para mover los datos de las posiciones tendría que realizar más iteraciones para mover todos los datos provocando un evidente desmedro en la eficiencia de este tipo de buffer.

4. Al cambiar el tipo de variable con las que trabaja la linea de retardo basada en buffer lineal de tipo `double` a `int16`, considerando un valor N de 20000 muestras y medir el tiempo de ejecución con el modo PROFILE de simulink, se obtuvo un tiempo para la *s-Function* de **0.635 ms**, mayor al obtenido con el mismo buffer, mismo valor de N y tipo de datos `double` que fue de **0.617 ms** (ver cuadro 1). Si bien la diferencia no es significativa correspondiendo ambas cifras al mismo es importante notar que efectivamente existe y se debe a que se está trabajando con una arquitectura diseñada y optimizada para trabajar con datos de 64 bits, al forzar a que el procesador trabaje con otro tipo de datos, este debe implementar nuevas instrucciones que realicen la conversión de datos desde el tipo de dato por defecto al indicado, en este caso `int16`.

   Se concluye entonces que el tiempo de ejecución efectivamente depende de la arquitectura que se esté usando, siendo óptimo el intentar trabajar con tipos de datos que coincidan con el numero de bits para los que fue diseñado el dispositivo que vaya a procesarlos, por ejemplo un *arduino UNO* presentaría mayor eficiencia usando datos de 8 bits así como una tarjeta

MSP430 optimizaría sus funciones trabajando con datos de 16 bits pues no tendrían que agregar instrucciones dedicadas a la conversión de datos.

5. Utilizando el modelo Simulink bypass del material del laboratorio se aprovecha el retardo de procesamiento de la señal del micrófono al pasar por Simulink y retornar a los audífonos. Con esta simulación se analiza como afecta este retardo a la capacidad de habla.

   Al desarrollar este ejercicio, intentando leer o hablar fluidamente, esta simple acción se complica bastante al escuchar la señal de voz retardada a través de los audífonos, esto se puede deber a que si se hace la analogía con un sistema de control en lazo cerrado, la señal de salida que se mide por los audífonos que posee retardo, al acoplarse con la señal de entrada (voz hablada) la suma de estas señales provoca un error significativo a la entrada del controlador (en este caso el cerebro), impidiendo elaborar acciones acertadas, que en el contexto sería continuar la lectura o el habla normal.

4. Principio de Procesar un Frame de Datos

1. Se implementa en lenguaje C una *s-Function* que recibe como entrada una señal digital con tasa de muestro de 16 kps, cuya salida actualiza cada 20 ms el valor RMS de los últimos 20 ms transcurridos de la señal.

   Como se desea calcular RMS para frames de 20 ms de una señal de frecuencia de muestreo de 16 kps, con las conclusiones obtenidas en el apartado 1, se puede determinar cual es el valor de N necesario en el código para poder cumplir con las especificaciones dadas. En este caso

$$Ts = \frac{1}{Fs} = \frac{1}{16\ kps} = 62{,}5\ \mu s$$

   Luego si se busca $T = 20\ ms$, entonces

$$N = \frac{T}{Ts} = \frac{20\ ms}{62{,}5\ \mu s} = 320$$

   El código en donde se implementa la *s-Function mencionada* es el siguiente

```
1  //Ts = 1/16kps = 62 us -> N = 20ms/62us = 320
2      #define N 320
3      double rms_20ms(double input) {
4
5      static double buffer = 0;
6      static unsigned int cont = 0;
7      static double output = 0;
8
9  //eleva la muestra al cuadrado y la guarda en la sumatoria
10     buffer = buffer + pow(input,2);
11
12 //reinicia el buffer y entrega el rms acumulado
13     if (cont >= N)
14         {
15         buffer = buffer/N;
16         output = sqrt(buffer);
17         buffer = 0;
18         cont = 0;
19         }
20     else
21         {
22             cont ++;
23         }
24
25
26     return output;
27 }
```

Al agregar el código asociado a la *s-Function* implementada para esta tarea, procede a simular su funcionamiento utilizando como entrada el archivo de audio *musica_16_16.wav* ya que cumple con el requerimiento de frecuencia de muestro de 16 kps, obteniendo de esta forma la gráfica que se muestra en la figura 7

Las figuras 8 y 9 intentan evidenciar como cambia la envolvente dependiendo del tamaño del frame que se procesa, siendo en el primer caso de una duración de 10 ms y 40 ms en el segundo.

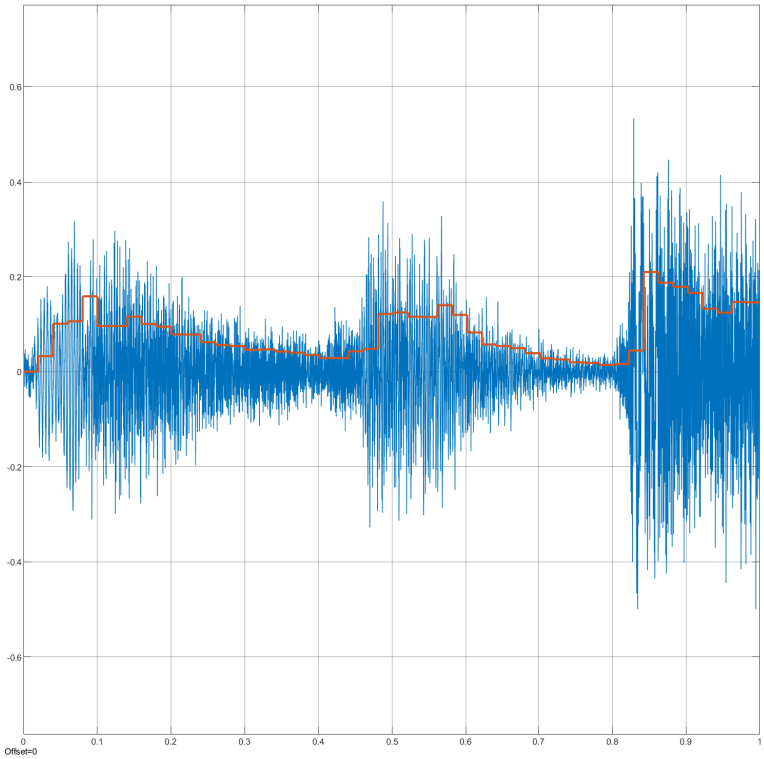

Figura 7: Resultado de calculo del valor RMS para frames de 20 *ms* al archivo de audio *musica_16_16.wav*.

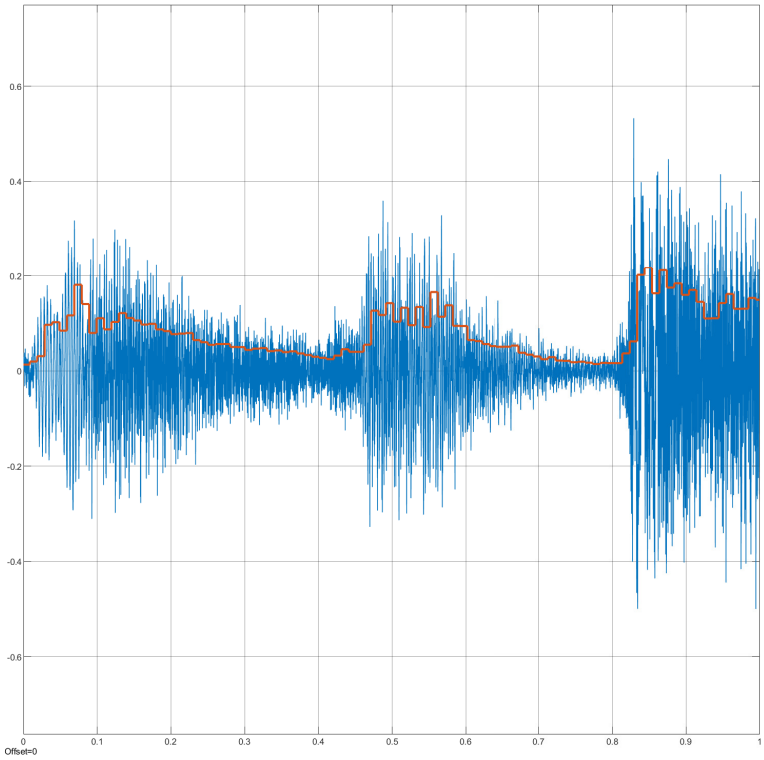

Figura 8: Resultado de calculo del valor RMS para frames de 10 *ms* al archivo de audio *musica_16_16.wav*.

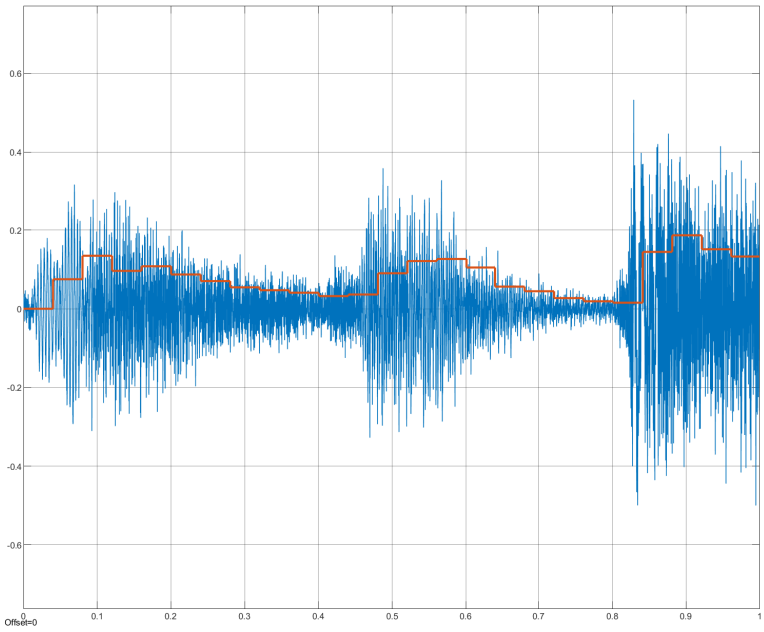

Figura 9: Resultado de calculo del valor RMS para frames de 40 *ms* al archivo de audio *musica_16_16.wav*.

5. Resampling: Interpolación y Decimación

1. Se implementa en MATLAB la función `y = interpola(x,P)`, que interpola una señal realizando los pasos de *upsampling* y filtrado. Donde x es la señal de entrada con una frecuencia de muestreo Fs y P es el factor de *upsampling*, es decir el numero de muestras ceros que se agregan a la señal original entre sus muestras (Técnicamente se agregan P-1 ceros).

```
function y = interpola(x,P)
    y = [];
    N = length(x);

    %Upsampling
    for i = 1:N
        y = horzcat(y, x(i), zeros(1,P-1));
    end

    %Filtrado
    B = fir1(40, 1/P);

    %Correccion de Magnitud
    y = P*filter(B,1,y);

end
```

Para visualizar el efecto de la interpolación implementada se realizan pruebas con una señal aleatoria de 10 muestras y utilizando un factor P de 7, obteniendo de esta forma las gráficas presentes en la figura 10. Cabe mencionar que por el efecto de el filtro aplicado se genera un retardo de grupo en la señal resultante, este efecto se muestra en la primera gráfica de la figura, mientras que en la segunda gráfica presenta este mismo resultado pero con el resultado de la interpolación eliminando las muestras correspondientes a este retardo de grupo.

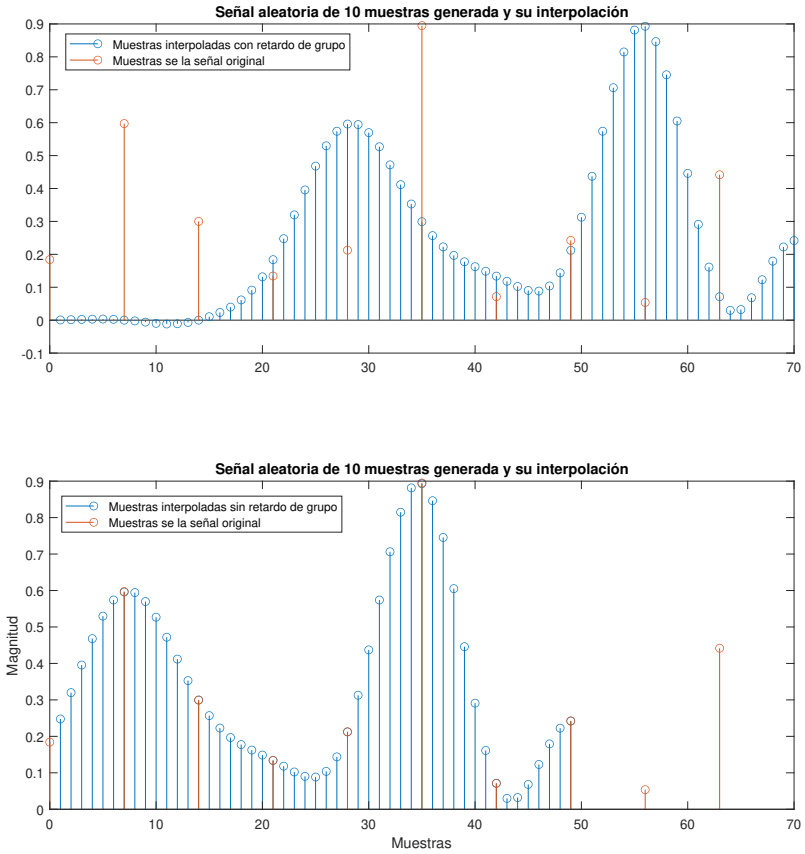

Figura 10: Gráfica de señal aleatoria de 10 muestras junto a su interpolación obtenida.

Para analizar como va variando el espectro de una señal interpolada se grafica el espectro obtenido en cada paso del proceso de interpolación, esto es, la señal original, la señal luego del *upsampling* y la señal interpolada para el archivo aliasing_test_16_16.mat. Estas gráficas se muestran en la figura 11, donde se puede ver que el espectro luego del *upsampling* se contrajo apareciendo la información del espectro a un tercio de la frecuencia a la que aparecía en el espectro de la señal original además de presentar aliases periódicos tal como se esperaba según la teoría. Finalmente luego de aplicar el filtro se obtiene el espectro resultante de la señal interpolada.

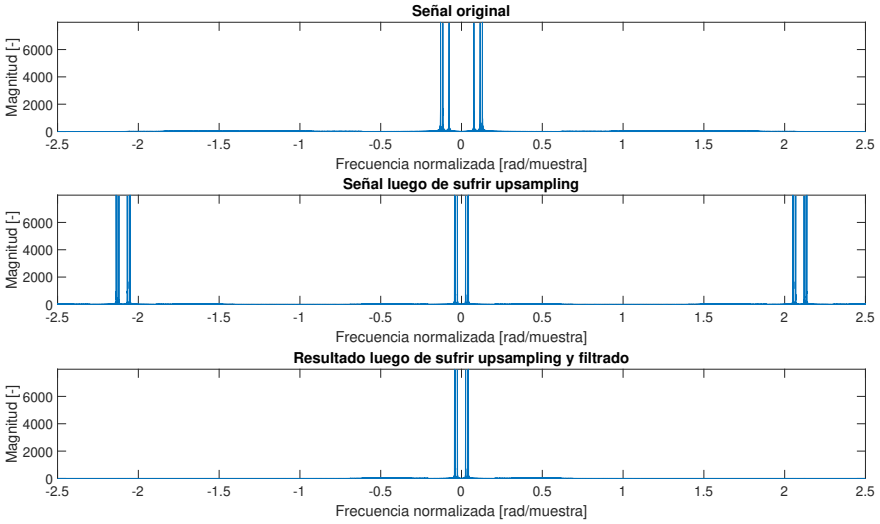

Figura 11: Espectro de los diferentes pasos del proceso de interpolación para el archivo aliasing_test_16_16.mat

Posteriormente se genera una señal Delta de Kronecker en un vector de muestras donde dicho delta se encuentra en la muestra numero 20, para poder observar el efecto de la interpolación tanto en muestras anteriores como posteriores al impulso. Esta señal generada se utilizará como entrada al proceso de interpolación que se ha implementado, los resultados obtenidos se muestran en la figura 12

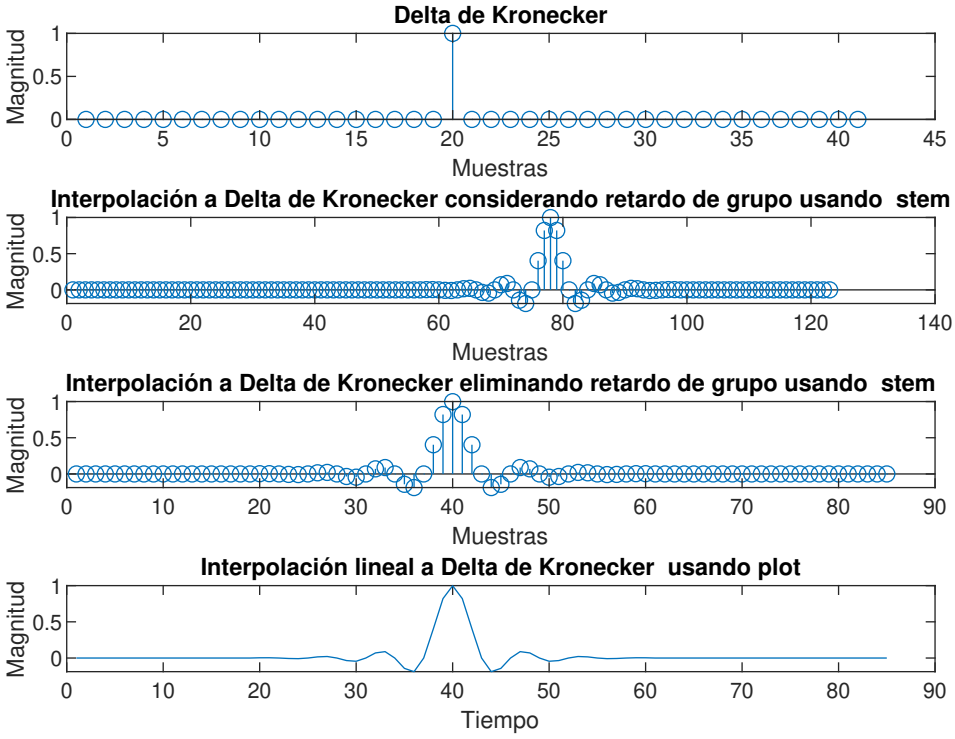

Figura 12: Interpolación a un Delta de Kronecker.

En la figura anterior se puede ver que la respuesta Y a un impulso de Kronecker se asemeja a una señal *sinc* con el inherente retardo de grupo asociado al filtro (segundo gráfico),ignorando dicho retardo, el resultado hace sentido, ya que el espectro en frecuencia de un impulso corresponde a una constante que al pasar por un filtro FIR de alto orden (en este caso orden 40), entrega una señal *rect* en frecuencia, al llevar este resultado de vuelta al dominio del temporal se obtiene una teóricamente una señal *sinc*. Se concluye entonces que la respuesta a impulso de el filtron FIR de orden 40 corresponde a una sinc.

2. Se implementa en MATLAB la función `y = decima(x,P)`, que aplica decimación de factor P a la señal de entrada x mediante filtrado y downsampling. La implementación se muestra a continuación:

```
function Y = decima(X,Q)
    %Filtrado
    B = fir1(40, 1/Q);
    y = filter(B,1,X);

    %Downsampling
    N = round(length(X)/Q);
    Y = zeros(N,1);
    for i = 1:N
        Y(i)= y((i-1)*Q+1);
    end

    %Correccion de Magnitud
    Y = Q*Y;
end
```

Posteriormente se utiliza la función para procesar la señal `aliasing_test` para $Q = 4$ y se grafica la magnitud de la FFT de la señal original y de la señal luego de la decimación. Los gráficos resultantes se muestran en la figura 13. Se observa el correcto funcionamiento de la función de decimación, expandiéndose en un factor de Q el ancho del espectro y manteniéndose las amplitudes.

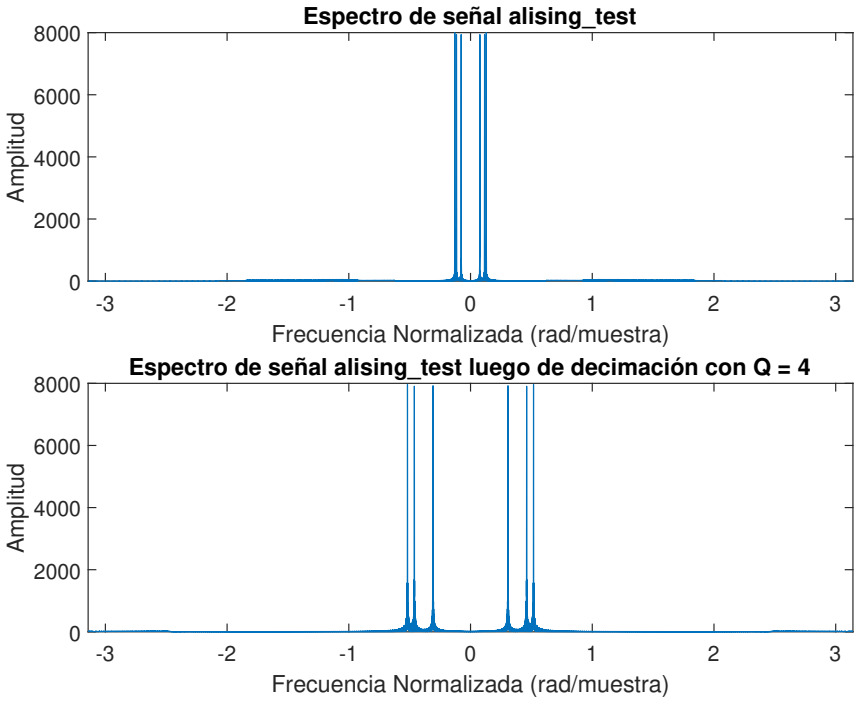

Figura 13: Espectro de señal `aliasing_test` original y luego de aplicar decimación.

3. Se utilizan las funciones de decimación e interpolación para hacer un remuestreo de la señal `aliasing_test` desde 16 $ksps$ a 12 $ksps$.

   Para encontrar el factor de decimación se busca llegar a una frecuencia de muestreo que permita llegar a 12 $ksps$ usando interpolación. Lo anterior corresponde a encontrar el máximo común divisor entre las frecuencias, por lo tanto el factor de decimación Q está dado por:

$$Q = \frac{16k}{\text{M.C.D}(16, 12)} = \frac{16}{4} = 4 \text{ ksps}$$

   Luego el factor P es el necesario para llevar de 4 ksps a 12 ksps, por lo tanto $P = 3$.

   Luego se grafica la magnitud de la FFT de la señal original y la señal remuestreada, lo cual se muestra en la figura 14. Se observa lo esperado en la "expansión" del espectro por disminuir la frecuencia de muestreo.

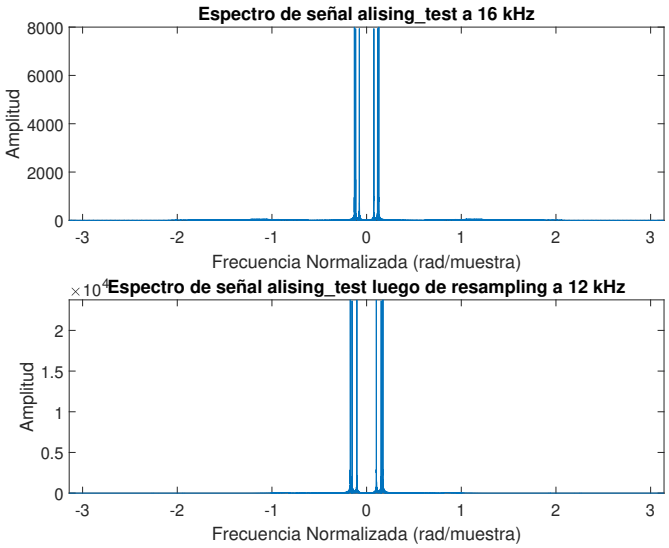

Figura 14: Espectro de señal `aliasing_test` y señal resampleada.

La señal resultante se guarda en el archivo *aliasing_test_16_12.wav* y se adjunta a la entrega.