INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

Teoría de Comunicaciones y Señales

"Simulación de Circuito RC"

Resumen

Implementación de un circuito RC en configuración filtro pasa bajas, haciendo uso de la convolución discreta y la respuesta al impulso del circuito RC serie para señales provenientes de archivos WAV en lenguaje C.

Por:

Romero Gamarra Joel Mauricio

Profesor:
GUTIÉRREZ ALDANA EDUARDO

Septiembre 2017

Índice

Contenido

Introducción:

En el siguiente programa, se realizó el diseño de un filtro pasa – bajas en lenguaje C[1], haciendo uso del teorema de convolución, sin embargo al realizarse en una computadora con tiempo memoria finita, se hizo un pequeño cambio al teorema para migrarlo al tiempo discreto, ya que por definición una integral es una suma, por lo tanto el único cambio fue una sumatoria de 0 al 20 (memoria usada para el filtro FIR), que es un sistema de fácil implementación y diseño pero que consume recursos de memoria y procesamiento mayor, sin embargo son sistemas **estables**,[1] que en este caso se utilizarán 20 flotantes.

Al decir que un sistema es estable, nos referimos a que la respuesta del sistema a un impulso será 0 cuando el tiempo tiende a ∞, si, por ejemplo, la respuesta del sistema al impulso tiende a un valor finito distinto de 0, se dice que es **críticamente estable**, y si tiende a un valor infinito, quiere decir que el sistema es **inestable**. En la Figura 1, podemos apreciar en el diagrama de polos y ceros la respuesta homogénea de un sistema a partir de sus polos y cómo se comporta el sistema en cada polo (estable o inestable).

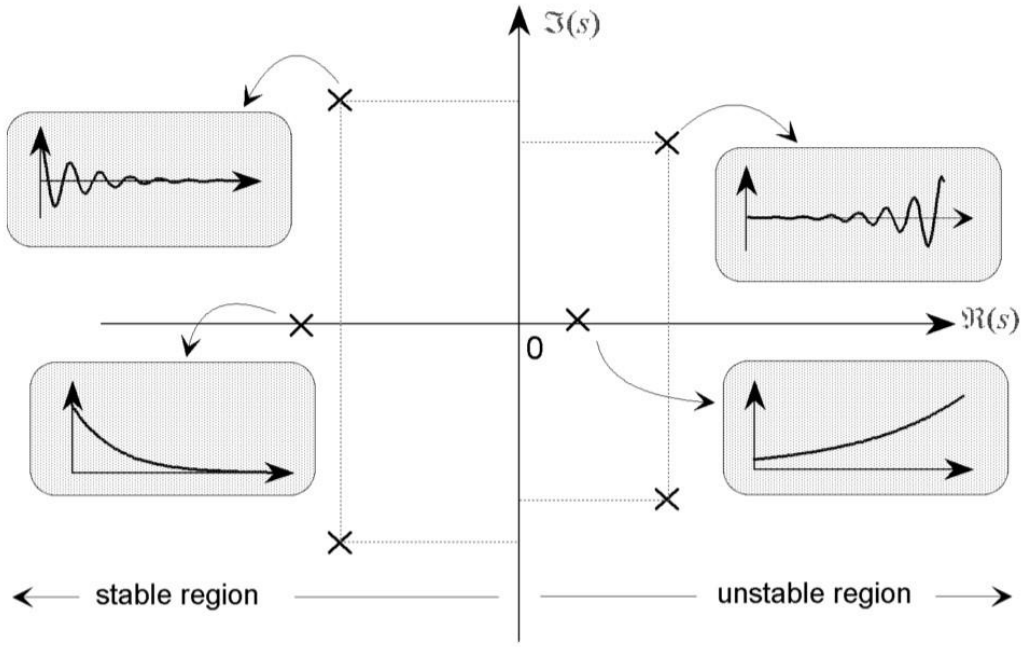


Figura 1. Respuesta homogénea al sistema en la gráfica de polos y ceros.

Después de observar la Figura 1, sabemos que, para ser un sistema estable, los polos de la función de transferencia deben estar ubicados en el semiplano izquierdo del diagrama de polos y ceros[2], este tipo de sistemas es el filtro pasa – bajas que diseñamos en esta práctica.

Análisis Teórico:

Esta sección necesita proveer un entendimiento del teórico, matemático y conceptual del contexto, antecedentes y justificación del trabajo.

Se pueden incluir diagramas, fórmulas, algoritmos, etc.

Comenzando con el análisis a priori del algoritmo a utilizar, necesitamos tener un mejor entendimiento de lo que es un filtro pasa – bajas y cómo funciona. Tomemos como ejemplo un circuito RC de primer orden, que es un circuito electrónico formado por una resistencia y un capacitor. Una de las características más importantes de este tipo de circuitos, es que este tipo de sistema es **lineal** (Cumple con propiedades de superposición) e **invariante en el tiempo** (la salida es la misma sin importar el instante de tiempo en el que se le aplique la entrada). En la Figura 2, podemos observar la configuración física de un circuito RC en una configuración pasa – bajas.[3]

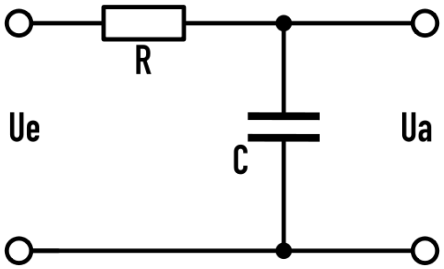


Figura 2. Configuración Pasa Bajas de un filtro RC de primer orden.

Lo que caracteriza a un filtro pasa bajas es que permite el paso de las frecuencias menores a la frecuencia de corte. Para altas frecuencias, la reactancia es baja logrando con esto que las señales sean atenuadas. En cambio, a bajas frecuencias (por debajo de la frecuencia de corte) la reactancia capacitiva es grande, lo que causa que estas frecuencias no se vean afectadas o son afectadas muy poco por el filtro.[4]

La frecuencia de corte es aquella donde la amplitud de la señal entrante cae hasta un 70.7 % de su valor máximo, esto ocurre cuando XC = R, a las frecuencias debajo de la frecuencia de corte se les conoce como **Banda de paso**, a las frecuencias por encima de la frecuencia de corte se les llama **Banda de atenuación**.[4]

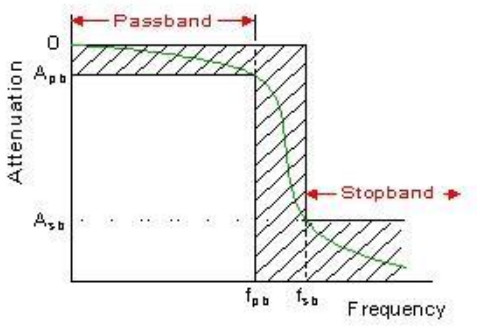


Figura 3. Comportamiento de un filtro pasa bajas.

En la Figura 3, se puede ver el comportamiento que tiene un filtro pasa bajas graficando la frecuencia, la zona donde comienza "*stopband*", es donde las frecuencias superaron a la frecuencia de corte, por lo tanto, en ese momento baja a 0 (filtro ideal), sin embargo, un filtro real va atenuando poco a poco las frecuencias hasta llegar a 0.

Posteriormente, ya que tenemos un entendimiento general del funcionamiento de un circuito RC en configuración de un filtro pasa bajas, procedemos a dar una explicación acerca de la convolución.

Como se mencionó en la introducción, la convolución en tiempo continuo se define como sigue:

$$x(t) * h(t) = \int_{-\infty}^{\infty} x(\tau)h(t - \tau)d\tau$$

Donde:

- x(t): Señal de entrada.
- h(t): Respuesta del sistema al impulso.
- τ: Desplazamiento en el tiempo.

Sin embargo, al diseñar un filtro pasa bajas en tiempo discreto para una señal digital (en este caso un archivo WAV), debemos tener en cuenta que poseemos una memoria y un tiempo de cómputo finito, sabemos que, una integral es una suma de todos los números desde -∞ hasta ∞, eso represente un número infinito de números, inclusive, si hacemos la integral de 0 a 1 (suponiendo que solo tuviéramos valores en las señales en ese intervalo), entre ambos valores hay una infinidad de números, por lo tanto haremos un cambio para hacer una suma tomando valores discretos (-∞, …, -2, -1, 0, 1, 2, …, ∞) como sigue[1]:

$$x(n) * h(n) = \sum_{\tau = -\infty}^{\infty} x(\tau)h(n - \tau)$$

Ya que tomamos en cuenta únicamente valores discretos para realizar nuestra convolución, debemos notar que la suma está definida desde -∞ hasta ∞, por lo que es prácticamente imposible realizarla en una computadora. De modo que, en esta parte es donde entra en juego la cantidad de memoria utilizada, en esta práctica se realizó con un impulso con 20 valores, así que, finalmente nuestra ecuación a usar en lenguaje C, será la siguiente.

$$x(n) * h(n) = \sum_{\tau = 0}^{20} x(\tau)h(n - \tau)$$

Ya que tenemos la fórmula que usaremos para calcular la convolución, necesitamos 2 cosas: Nuestra señal de entrada, y la respuesta al impulso que usaremos, sabemos que la respuesta al impulso de un sistema se define como la respuesta de un sistema a δ(t), que se define así:

$$\int_{-\infty}^{\infty} \delta(t)dt = 1$$

Para obtener la respuesta al impulso del sistema, primero debemos obtener la función de transferencia de nuestro circuito RC en configuración pasa – bajas, ver Figura 2. Para resolver el circuito, definimos a la función de transferencia como sigue:

$$H(s) = \frac{Vsal\ (s)}{Vent\ (s)}$$

Al observar la ecuación anterior, podemos ver que debemos obtener el voltaje de salida y el voltaje de entrada en el circuito RC. Al ver la Figura 2, vemos que el voltaje de salida se mide en el capacitor, por lo tanto, la impedancia compleja en el capacitor se define como $\frac{1}{sC}$, que colocaremos en el numerador de la ecuación anterior en la función de transferencia. Además, para el voltaje de entrada se observa que se encuentran el capacitor y la resistencia en serie, sabiendo que para calcular el voltaje en ambos elementos se deben sumar, lo realizamos, conociendo la impedancia en el capacitor y sumamos la resistencia, que se define como $\frac{1}{sC}$ + R, que colocaremos como el denominador de la función de transferencia y la nueva ecuación nos queda de la siguiente forma:

$$H(s) = \frac{\frac{1}{sC}}{\frac{1}{sC} + R} = \frac{1}{1 + sRC}$$

Ya que tenemos calculada la función de transferencia para el circuito RC en configuración Pasa – Bajas, calculamos la transformada inversa de Laplace para pasar la función de transferencia al dominio del tiempo y obtener h(t) de la siguiente forma:

$$\mathcal{L}\{\mathrm{H(s)}\} = h(t) = e^{-\frac{2\tau\pi f_C}{44{,}100}}$$

Al tener la respuesta al impulso deseada, vemos que existe τ, y recordemos que, en nuestra convolución discreta, τ va a tener valores de 0 a 20, para poder calcular nuestro impulso que usaremos como h(n) para realizar la convolución con la señal de entrada.

Software (librarías, paquetes, herramientas):

- GoldWave versión 4.26
- Frhed 1.6.0
- Dev C
- Sublime Text 3
- Bizagi Modeler

Procedimiento:

Para explicar el procedimiento utilizado, retomaremos la idea que este filtro pasa – bajas estará programado en lenguaje C, por lo tanto, debemos obtener el algoritmo para realizar la convolución, sabiendo que, por definición, la señal de entrada debe voltearse y multiplicar sus valores por la respuesta al impulso, sumando estos valores y escribiéndolos en una señal nueva. Por lo tanto, utilizaremos 2 arreglos de tipo short (ya conocemos la estructura de un archivo WAV[5]). Sin

embargo, como la señal generada con GoldWave es una señal que entra a un sistema (en este caso la computadora), ya entra "invertida", por lo que únicamente se guardarán los elementos en el arreglo y se irá multiplicando por el impulso obtenido. A continuación, en la Figura 4, podemos observar el diagrama de flujo que nos muestra los pasos a seguir para realizar el filtrado a la señal de entrada.

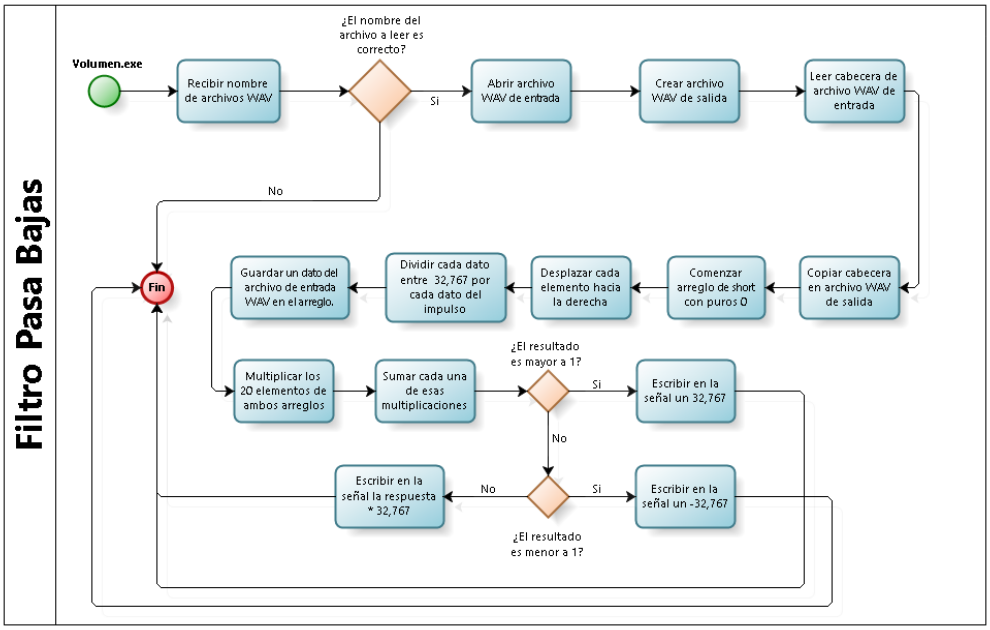


Figura 4. Diagrama de flujo del proceso general del filtro pasa bajas en lenguaje C.

Ahora que conocemos el proceso general (Ver Figura 4), podemos comenzar a codificar. Ya sabemos toda la estructura de la cabecera y los datos en un archivo WAV (revisar **[5]**), así que procederemos a explicar únicamente el algoritmo para realizar la convolución en lenguaje C[1].

El algoritmo es muy sencillo, como lo indica el diagrama primero debemos crear un arreglo de tipo short de tamaño 20 (que es la memoria utilizada), al que llamaremos entrada en donde guardaremos los datos de la señal generada con GoldWave. Debemos tener otro arreglo de tipo float y tamaño 20, sin embargo, lo llamaremos impulso y en él, guardaremos la respuesta del sistema al impulso unitario, que está descrito en el análisis teórico de este reporte, estos valores serán fijos durante toda la ejecución del programa.

```
float * generaImpulso ()
{
    float * impulso = (float *) malloc (sizeof (float) * TAM_ARREGLO);
    for (i = 0; i < TAM_ARREGLO; i ++)
    {
        impulso [i] = (exp ((-2 * PI * FREC_CORTE * i) / 44100));
    }
    return impulso;
}
```

Figura 5. Función que genera el impulso a partir de las fórmulas obtenidas en el análisis teórico de la solución.

Posteriormente, utilizaremos otro flotante auxiliar al que llamaremos max, en el que guardaremos la suma de multiplicar cada elemento del arreglo impulso por 32,767 (valor máximo de un short, y valor máximo que puede tener la señal generada en GoldWave), para obtener nuestro peor caso. Ahora, teniendo ambos arreglos, procedemos a hacer un ciclo de 0 a SubChunkSize2[5] para leer los datos de la señal de entrada, cada dato lo vamos a dividir entre max para redimensionar nuestra entrada y tener valores entre -1 y 1 únicamente. Procedemos a recorrer nuestros elementos del arreglo 'entrada' hacia la derecha y posteriormente guardar el dato dividido en la primera posición del arreglo.

```
for (i = 0; i < cab.SubChunk2Size; i ++, j++)
{
    lectura = fread(&muestra, sizeof (short), 1, archivoEntrada);
    for (k = (TAM_ARREGLO - 1); k >= 0; k --)
        entrada [k] = entrada [k - 1];
    entrada [0] = (muestra / max);
    muestra = convolucion (entrada, impulso);
    escritura = fwrite(&muestra, sizeof (short), lectura, archivoSalida);
}
```

Desplazamiento de elementos

Figura 6. Ciclo donde se realiza el desplazamiento, almacenamiento y escritura de la señal filtrada.

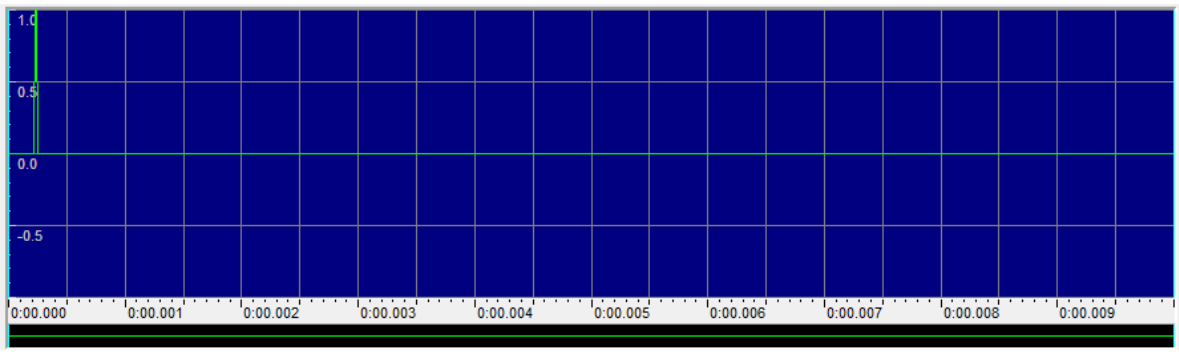
Ya que lo almacenamos, debemos hacer un ciclo de 0 a 20, en el que vamos a multiplicar los 2 arreglos (impulso y entrada) y vamos a ir sumando cada resultado.

```
float convolucion (float * entrada, float * impulso)
{
    float respuesta = 0;

    //Realizamos la convolución
    for (i = 0; i < TAM_ARREGLO; i ++)
        if (entrada [i] != 0)
            respuesta += (entrada [i] * impulso [i]);

    if (respuesta > 1)
        return (1 * 32767);
    if (respuesta < -1)
        return (-1 * 32767);
    else
        return (respuesta * 32767);
}
```

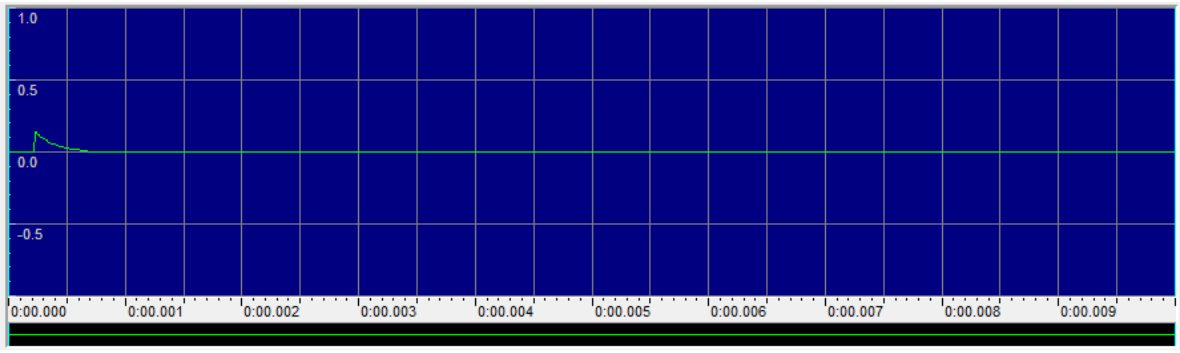
Figura 7. Algoritmo que realiza la convolución tomando en cuenta los valores posibles.

Después de esto, **existen 3 posibles valores** que obtendremos de esa suma, **valores menores a -1**, por lo que escribiremos un -32,767 (valor mínimo que puede tener un short), **valores mayores a 1**, por lo que escribiremos un 32, 767 (valor máximo que puede tener un short), y **valores entre -1 y 1**, por lo que escribiremos el valor obtenido multiplicado por 32,767. Todo este algoritmo, se realizará en un ciclo hasta que leamos todos los datos, posteriormente, tendremos la señal filtrada. En la sección de resultados se muestran los ejemplos de entrada y las señales obtenidas al realizar el filtrado con el programa.

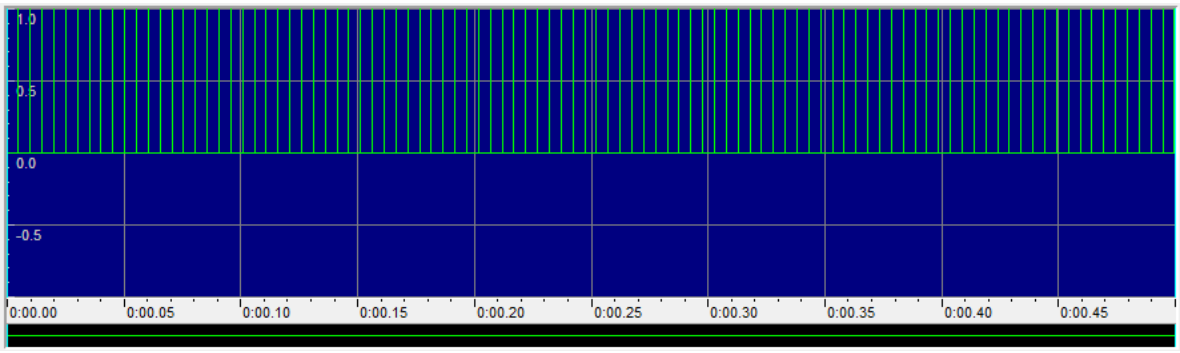
Resultados



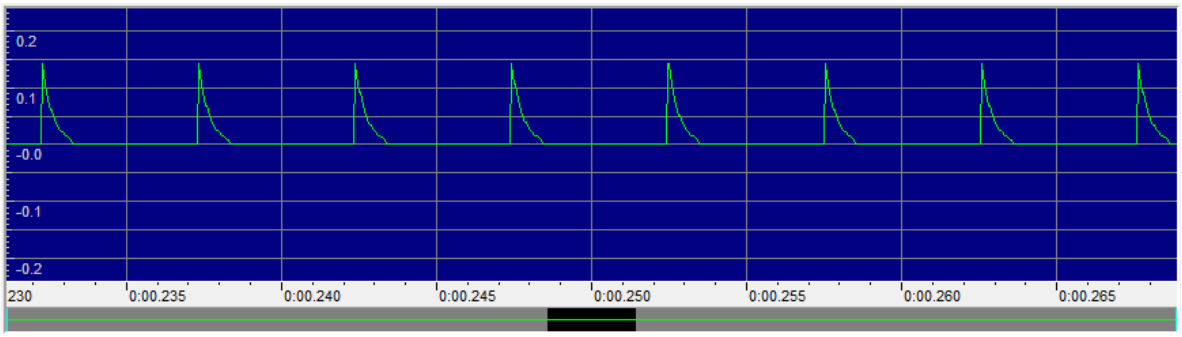
Señal de Entrada 1. Delta de Dirac



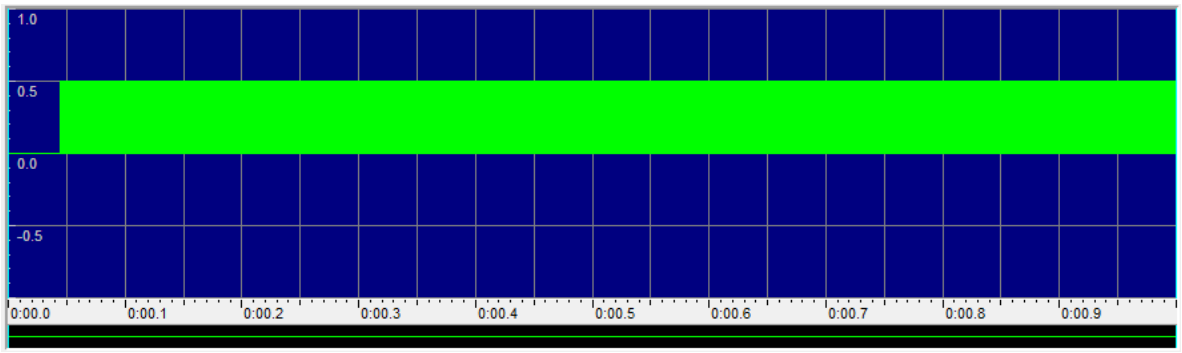
Señal Filtrada 1. Delta de Dirac



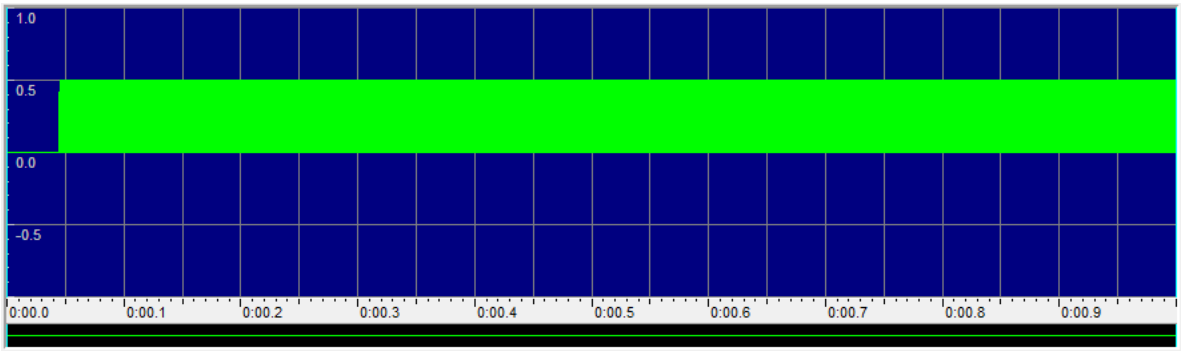
Señal de Entrada 2. Tren de Impulsos



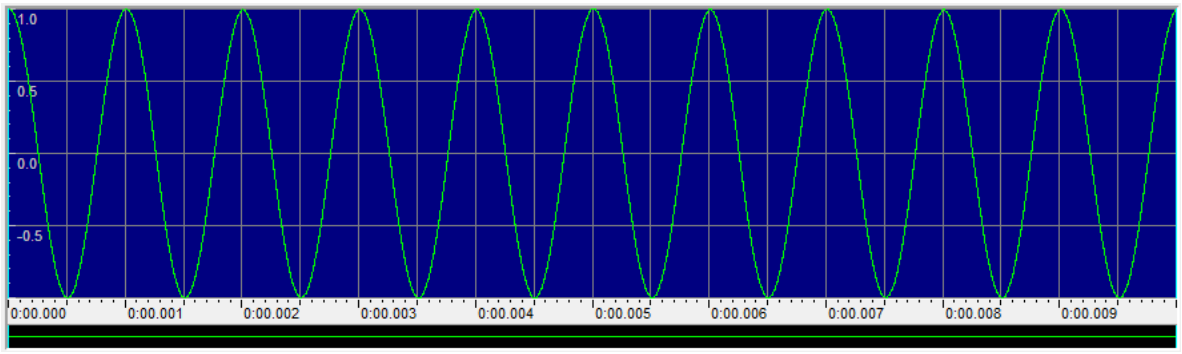
Señal Filtrada 2. Tren de Impulsos (Ampliado)



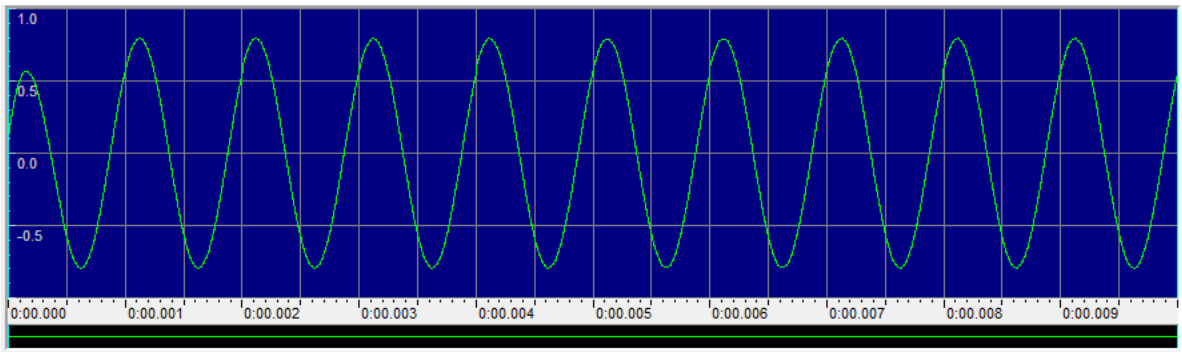
Señal de Entrada 3. Función Heaviside



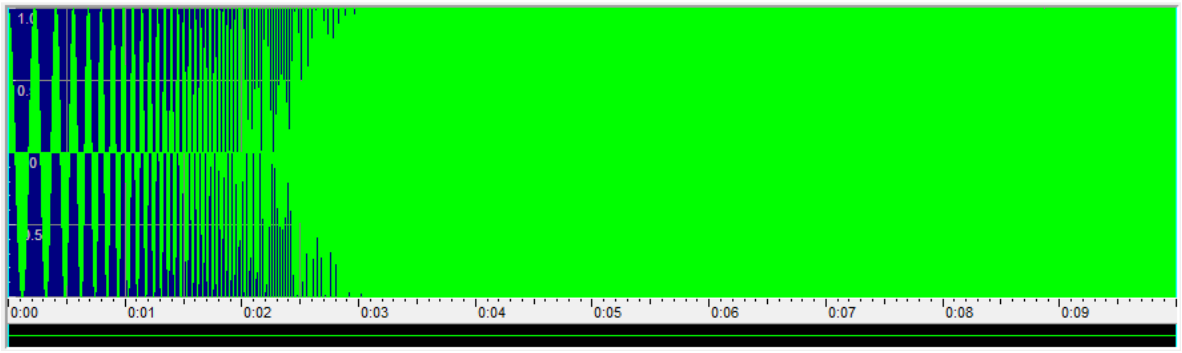
Señal Filtrada 3. Función Heaviside



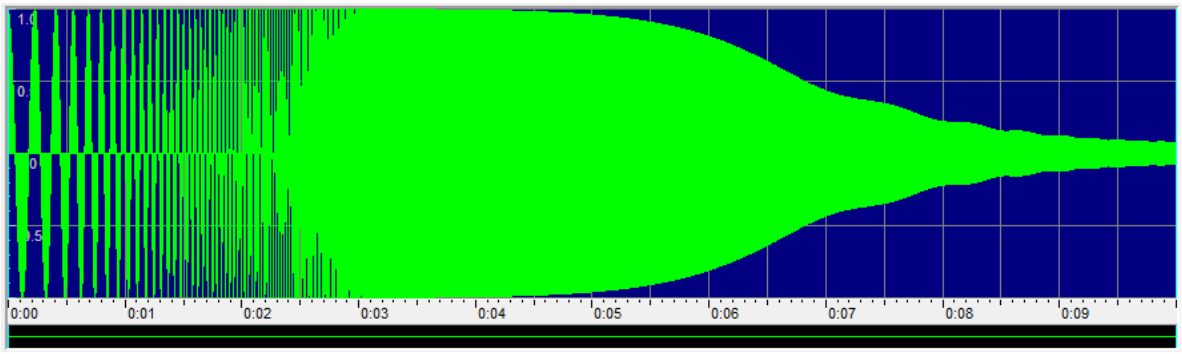
Señal de Entrada 4. Coseno



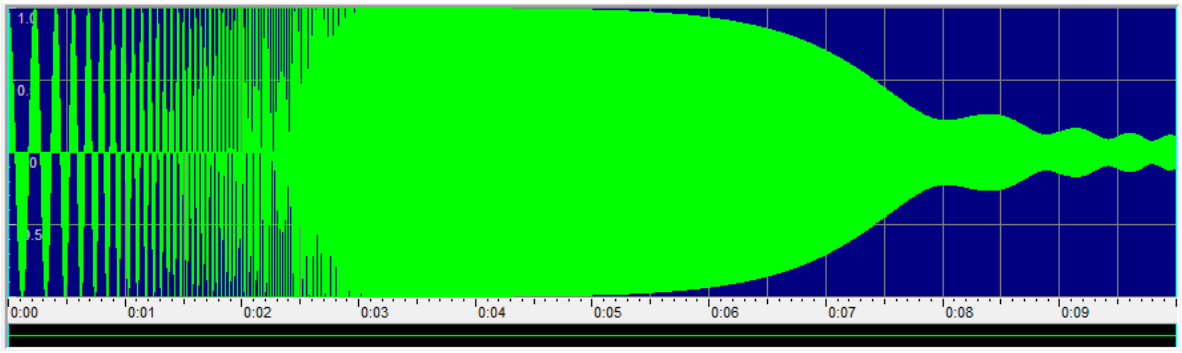
Señal Filtrada 4. Coseno



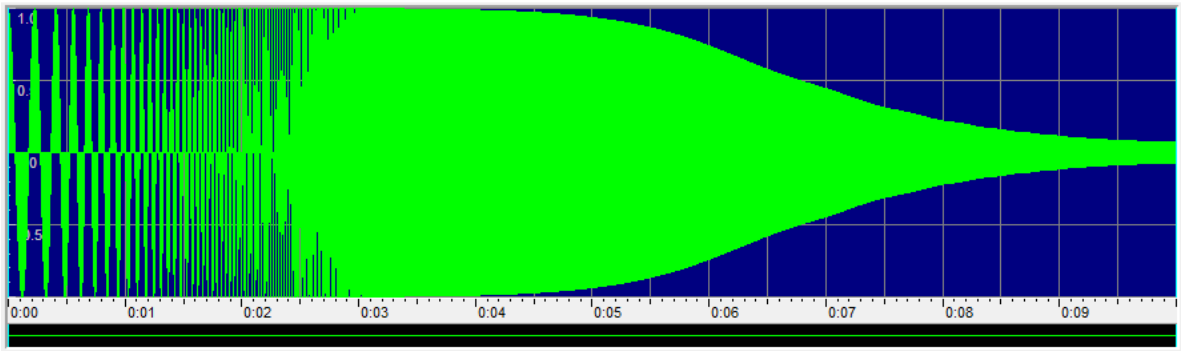
Señal de Entrada 5. Coseno Inestable



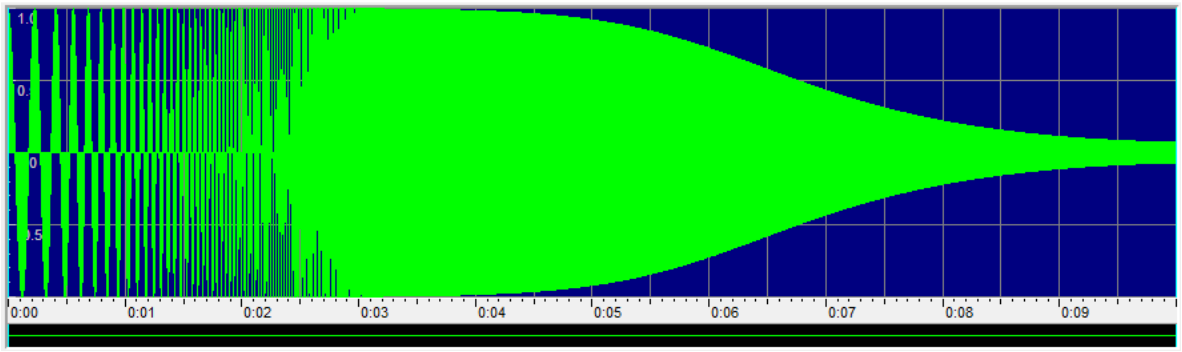
Señal Filtrada 5. Coseno Inestable (Memoria utilizada = 20)



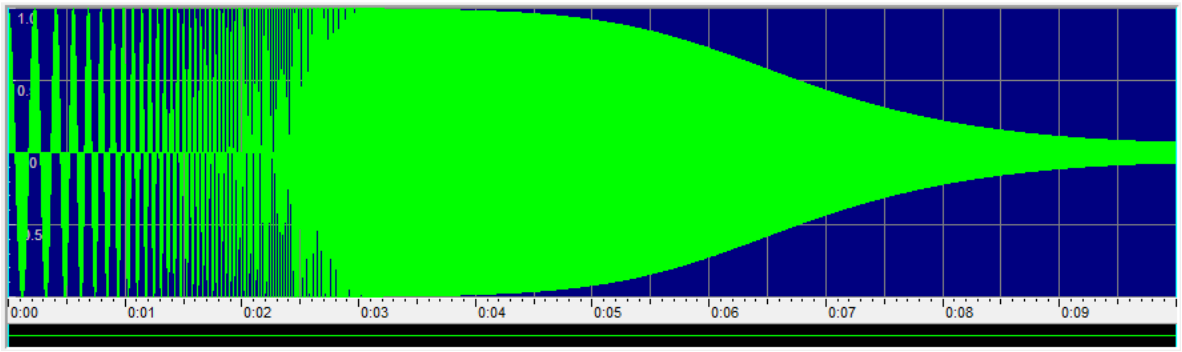
Señal Filtrada 6. Coseno Inestable (Memoria utilizada = 10)



Señal Filtrada 7. Coseno Inestable (Memoria utilizada = 30)



Señal Filtrada 8. Coseno Inestable (Memoria utilizada = 100)



Señal Filtrada 9. Coseno Inestable (Memoria utilizada = 1000)

Discusión:

La sección de discusión tiene 2 objetivos principales:

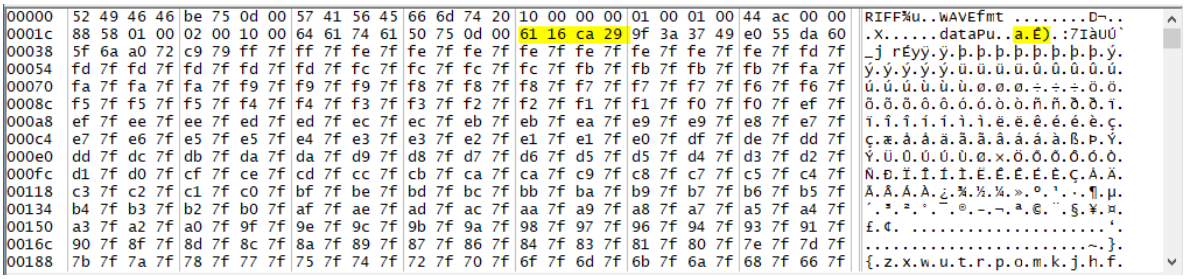
- Interpretar y explicar los resultados del estudio.
- Explorar la importancia del estudio, encontrando, calificando y explorando la importancia teórica de los resultados.

La discusión es también un espacio en el reporte donde cualquier calificación o reservación que se tiene sobre la investigación debe ser mencionada.
En los resultados obtenidos, se puede observar una notable diferencia entre la señal de salida que se introduce al sistema y la señal de salida al pasar por el filtro pasa bajas. En algunas señales, como la función Heaviside o el Coseno, no es tan notorio el cambio, sin embargo, en las demás, es bastante distinta la señal, dejando pasar únicamente las frecuencias menores a la frecuencia de corte (para esta práctica, la frecuencia de corte es de 1 KHz), y atenuando las frecuencias mayores a ésta.

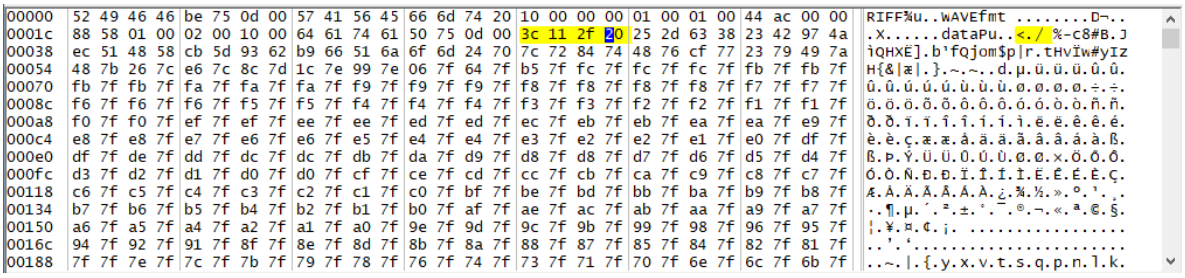
Es importante resaltar que podemos comprobar que el comportamiento mostrado en la Figura 3 es correcto, además, aseveramos que la gráfica real difiere de la gráfica ideal de un filtro pasa bajas, debido a que existen distintos factores que hacen imposible el eliminar inmediatamente las frecuencias (es decir, mandarlas a 0), pero, lo que sí podemos hacer, es irlas atenuando paulatinamente.

En las señales filtradas del coseno inestable (recordemos que una señal inestable es una señal que crece indefinidamente, es decir que los polos que la describen se encuentran gráficamente en el semiplano derecho del diagrama de polos y ceros), es interesante notar que mientras menor sea la memoria utilizada para realizar la convolución, la señal filtrada tiene un poco de ruido, ya que no tiene un filtrado tan correcto como lo esperaríamos.

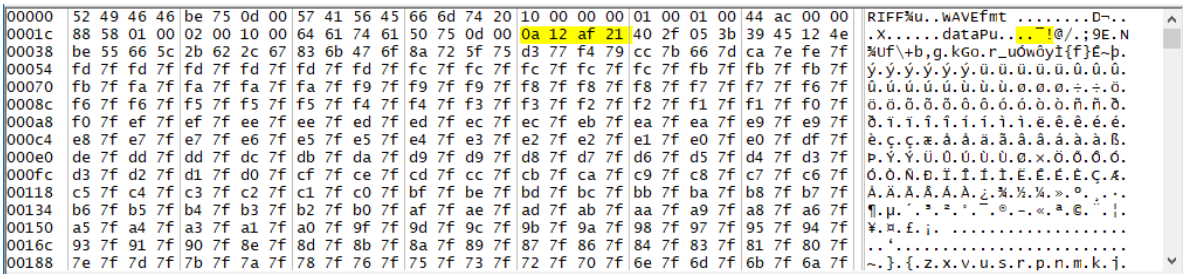
Además, debido a esto, podríamos esperar que mientras se utilice más memoria, la señal filtrada obtenida sería más limpia, y es algo cierto, sin embargo, existe un límite, como podemos ver si utilizamos 30 números flotantes, comparada con 10 o 20, la diferencia es muy notoria, pero si la aumentamos a 100, ya no hay diferencia, o al menos no gráficamente, y cuando aumentamos a 1,000, se puede ver que efectivamente no existe un cambio significativo en la señal de salida. Por lo cual, estaríamos consumiendo demasiado tiempo de cómputo y recursos de la computadora, realizando operaciones innecesarias para una respuesta idéntica o muy parecida.
Para poder corroborar esto, no podemos seguir viendo únicamente con el software GoldWave, así que nos auxiliaremos con Frhed que es un software que nos permite visualizar estas señales en formato binario o hexadecimal.



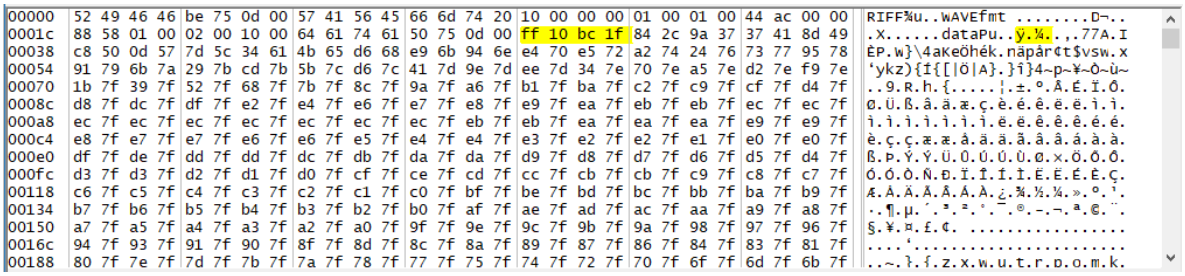
Hexadecimal 1. Coseno Inestable Filtrado (Memoria utilizada = 10)



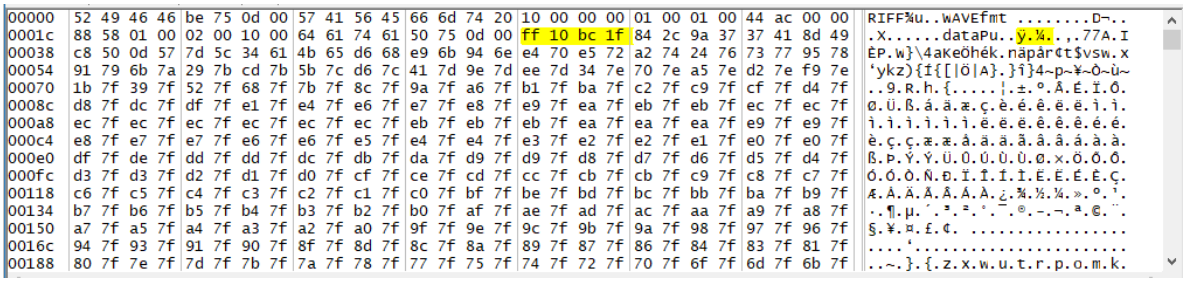
Hexadecimal 2. Coseno Inestable Filtrado (Memoria utilizada = 30)



Hexadecimal 3. Coseno Inestable Filtrado (Memoria utilizada = 20)



Hexadecimal 4. Coseno Inestable Filtrado (Memoria utilizada = 100)



Hexadecimal 5. Coseno Inestable Filtrado (Memoria utilizada = 1000)

Al ver los archivos abiertos en Frhed, se puede notar claramente en la parte resaltada en amarillo que los archivos no son los mismos, a excepción de los Hexadecimales 4 y 5, que son señales idénticas.

Así que nuestra aseveración era correcta, las señales no son idénticas, sin embargo, es algo complicado saber en que difieren, ya que hay que observarlo en un editor hexadecimal, por lo tanto, aumentar la memoria es aumentar exponencialmente las operaciones que realiza el CPU, por un beneficio que a simple vista no es muy notorio, así que, personalmente, con una memoria de 30 es suficiente y no se realizan tantas operaciones, ya que son 3 ciclos anidados:

- El ciclo para ir recorriendo todos los datos (aproximadamente 88,200 iteraciones)
- El ciclo para desplazar los elementos hacia la derecha (se realiza 20 veces por cada iteración del ciclo que recorre los datos, así que se realiza 1,764,000 veces en total).
- El ciclo que ejecuta el algoritmo de la convolución discreta que al igual que el anterior, se ejecuta 1,764,000 veces.

Podemos notar que son demasiadas operaciones, y eso es para 20 unidades de memoria, si lo incrementamos a 30, las operaciones del segundo y tercer ciclo se realizarían 2,646,000 veces. Con una memoria de 100, se realizan 8,820,000 veces, y si la memoria es de 1,000, las operaciones se realizan 88, 200, 000 veces.

Si consideramos que una computadora puede procesar 1,000,000,000 de operaciones por segundo, para una memoria de 20, la ejecución del puro algoritmo (sin considerar copiar la cabecera, generar el impulso, etc.), se tardaría aproximadamente 0.00361 segundos. Para una memoria de 30, la ejecución tardaría 0.00538 segundos, para memoria de 100, la ejecución tardaría aproximadamente 0.01772 segundos y para de 1000, la ejecución tardaría aproximadamente 0.1764 segundos.

Como se puede apreciar, los tiempos de ejecución no parecen demasiado, pero poniéndolo en perspectiva, el tamaño más grande que se probó con estas señales fue de 1 segundo, si al filtro le

metemos un archivo WAV que dure 3 minutos (aproximadamente lo que dura una canción), el tiempo de ejecución aumentaría a 31.7678 segundos con una memoria utilizada de 1,000, que es un tiempo de cómputo demasiado grande. Sin embargo, para 20, el tiempo de ejecución aumentaría a 0.6509 segundos, que es un tiempo bastante aceptable, inclusive para 30, el tiempo de ejecución estimado es de 0.9684 segundos, que sigue siendo un tiempo bastante bueno.

Por lo tanto, la cantidad de memoria ideal se encuentra entre 20 y 30, para obtener una mejor relación de costo – beneficio (tomando en cuenta tiempo computacional y calidad de la señal de salida).

Conclusiones:

El diseño de filtros digitales es algo interesante, sin mencionar que la era de las computadoras está en pleno auge, debido a que están en todos lados. Además, a diario escuchamos música, ya sea en nuestro automóvil, en el radio, etc. Por lo que la manipulación de este tipo de archivos resulta interesante para poder experimentar y pensar distintas aplicaciones con ellos.

En el caso del filtro pasa bajas, como vimos en la señal del coseno inestable, al pasarla por el filtro, se volvió una señal estable, eliminando la peligrosidad de una señal que crece sin parar.

Una aplicación de este tipo de filtros, puede ser un ecualizador musical, que modifica el volumen del contenido en frecuencias de la señal de entrada, por lo que, podemos eliminar o añadir decibeles en determinadas frecuencias (en este caso, eliminar decibeles por encima de la frecuencia de corte), para compensar posibles errores en la grabación de un audio profesional.

Realizar la convolución en lenguaje C, es un algoritmo demasiado costoso computacionalmente hablando, además de que no puede realizarse en el tiempo continuo debido a las limitaciones que tiene cualquier computadora ya que contamos con una memoria y tiempo de procesamiento finitos, por lo tanto los resultados obtenidos son buenos, pero no excelentes, las aplicaciones que tiene, no solo en el procesamiento de señales, son bastante amplias e interesantes, algunas de ellas es el procesamiento de imágenes, específicamente el filtrado en las mismas, en acústica, ya que un eco es la convolución del sonido original con una función que represente los objetos variados que lo reflejan.

Referencias:

[1] Clavijo Mendoza Juan Ricardo, 'Diseño y Simulación de Sistemas Microcontrolados en lenguaje C'. Colombia: 2011, pp. 190 – 193.

[2] Department of Mechanical Engineering, 'Understanding Poles and Zeros'. [Online]. Disponible en: http://web.mit.edu/2.14/www/Handouts/PoleZero.pdf. [Accedido: 11 – septiembre – 2017].

[3] Wikipedia, 'Circuito RC'. [Online]. Disponible en: https://es.wikipedia.org/wiki/Circuito_RC. [Accedido: 11 – septiembre – 2017].

[4] Electrónica Unicrom, 'Filtro RC Paso Bajo'. [Online]. Disponible en: https://unicrom.com/filtro-rc-paso-bajo/. [Accedido: 11 – septiembre – 2017].

[5] Romero Gamarra Joel Mauricio, 'Bajar Volumen a un Archivo WAV'. [Online]. Disponible en: https://github.com/JoelRomero97/Teoria-de-Comunicaciones-y-Senales/blob/master/Prácticas/Bajar%20Volumen/Reporte.docx.

Código

Filtro.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "Cabecera.c"
#define TAM_ARREGLO 20

int main(int argc, char const *argv[])
{
	FILE * entrada, * salida;
	cabecera cab;
	int i, k;
	short muestra;
	float * impulso = (float *) malloc (sizeof (float) * 20);
	float * signal = (float *) malloc (sizeof (float) * 20);
	char * archivo_salida = (char *) malloc (sizeof (char));
	char * archivo_entrada = (char *) malloc (sizeof (char));
	system ("cls");
	if (argc < 3)
	{
		printf("Error, faltan argumentos.\n");
		printf ("Ejemplo: '%s Archivo1.wav Salida.wav'\n\n", argv [0]);
		exit (0);
	}else
	{
		archivo_entrada = (char *) argv [1];
		archivo_salida = (char *) argv [2];
	}

	//Abrimos los archivos en modo binario
	entrada = abre_archivo (archivo_entrada, archivo_salida,1);
	salida = abre_archivo (archivo_entrada, archivo_salida, 2);

	//Leemos e imprimimos la cabecera del archivo wav
	copiar_cabecera (entrada, salida, &cab);
	imprimir_cabecera (&cab);

	//Generamos la respuesta al impulso
	impulso = generaImpulso ();
	float max = 0;
	for (i = 0; i < TAM_ARREGLO; i ++)
		max += (impulso [i] * 32767);

	//Llenamos el arreglo de entrada con puros ceros
	memset (signal, 0, TAM_ARREGLO);

	//Escribimos el resto de los datos realizando la convoluci□n
	for (i = 0; i < (cab.SubChunk2Size / 2); i ++)
	{
		fread (&muestra, sizeof (short), 1, entrada);
		for (k = (TAM_ARREGLO - 1); k >= 0; k --)
			signal [k] = signal [k - 1];
		signal [0] = (muestra / max);
					//Insertamos los datos en el arreglo
		muestra = convolucion (signal, impulso);
```

```
                    fwrite (&muestra, sizeof (short), 1, salida);
                    //Escribimos los datos nuevos en el archivo
          }
          printf ("\n\n");
          fclose (entrada);
          fclose (salida);
          printf ("Archivo '%s' filtrado correctamente guardado en '%s'.\n",
archivo_entrada, archivo_salida);
          return 0;
}
```

Cabecera.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "Cabecera.h"
#define PI 3.14159265
#define FREC_CORTE 1000
#define TAM_ARREGLO 20

int i;                                          //Variable global para manejar ciclos

FILE * abre_archivo (char * entrada, char * salida, int tipo)
{
          FILE * pt1, * pt2;
          pt1 = fopen (entrada, "rb");
          if (pt1 == NULL)
          {
                    printf("Error al abrir archivo '%s'.\n", entrada);
                    exit(0);
          }

          //Abrimos el archivo a escribir en modo binario
          pt2 = fopen (salida, "wb");
          if (pt2 == NULL)
          {
                    printf("Error al crear el archivo '%s'.\n", salida);
                    exit(1);
          }
          if (tipo == 1)
          {
                    printf("Archivo '%s' abierto correctamente.\n", entrada);
                    return pt1;
          }else
          {
                    printf("Archivo '%s' creado correctamente.\n", salida);
                    return pt2;
          }
}

void copiar_cabecera (FILE * entrada, FILE * salida, cabecera * cab)
{
          rewind (entrada);
          rewind (salida);

          //ChunkID
          fread (cab -> ChunkID, sizeof (char), 4, entrada);
          fwrite (cab -> ChunkID, sizeof (char), 4, salida);

          //ChunkSize
          fread (&cab -> ChunkSize, sizeof (int), 1, entrada);
          fwrite (&cab -> ChunkSize, sizeof (int), 1, salida);

          //Formato "Fmt"
          fread (cab -> Format, sizeof (char), 4, entrada);
          fwrite (cab -> Format, sizeof (char), 4, salida);
```

```
        //SubChunk1ID Formato de datos "fmt"
        fread (cab -> SubChunk1ID, sizeof (char), 4, entrada);
        fwrite (cab -> SubChunk1ID, sizeof (char), 4, salida);

        //SubChunk1Size
        fread (&cab -> SubChunk1Size, sizeof (int), 1, entrada);
        fwrite (&cab -> SubChunk1Size, sizeof (int), 1, salida);

        // Formato de audio
        fread (&cab -> AudioFormat, sizeof (short), 1, entrada);
        fwrite (&cab -> AudioFormat, sizeof (short), 1, salida);

        //Canales
        fread (&cab -> NumChannels, sizeof (short), 1, entrada);
        fwrite (&cab -> NumChannels, sizeof (short), 1, salida);

        //SampleRate
        fread (&cab -> SampleRate, sizeof (int), 1, entrada);
        fwrite (&cab -> SampleRate, sizeof (int), 1, salida);

        //ByteRate
        fread (&cab -> ByteRate, sizeof (int), 1, entrada);
        fwrite (&cab -> ByteRate, sizeof (int), 1, salida);

        //Block Align
        fread (&cab -> BlockAlign, sizeof (short), 1, entrada);
        fwrite (&cab -> BlockAlign, sizeof (short), 1, salida);

        //Bits per Sample
        fread (&cab -> BitsPerSample, sizeof (short), 1, entrada);
        fwrite (&cab -> BitsPerSample, sizeof (short), 1, salida);

        //SubChunk2ID
        fread (cab -> SubChunk2ID, sizeof (char), 4, entrada);
        fwrite (cab -> SubChunk2ID, sizeof (char), 4, salida);

        //SubChunk2Size
        fread (&cab -> SubChunk2Size, sizeof (int), 1, entrada);
        fwrite (&cab -> SubChunk2Size, sizeof (int), 1, salida);
}

void imprimir_cabecera (cabecera * cab)
{
        char * formatoArchivo = (char *) malloc (sizeof (char));
        printf("\n\n\n(1-4) Chunk ID: %s\n",cab -> ChunkID);
        printf("(5-8) ChunkSize:  %u\n",cab -> ChunkSize);
        printf("(9-12) Format: %s\n",cab -> Format);
        printf("(13-16) SubChunk 1 ID: %s\n",cab -> SubChunk1ID);
        printf("(17-20) SubChunk 1 Size: %u\n",cab -> SubChunk1Size);
        if (cab -> AudioFormat == 1)
                        strcpy(formatoArchivo,"PCM");
                printf("(21-22) Audio Format: %u,%s\n",cab ->
AudioFormat,formatoArchivo);
        if (cab -> NumChannels == 1)
                        strcpy(formatoArchivo,"Mono");
                else
                        strcpy(formatoArchivo,"Stereo");
                printf("(23-24) Number of Channels: %u, Tipo: %s\n",cab ->
NumChannels,formatoArchivo);
        printf("(25-28) Sample Rate: %u\n",cab -> SampleRate);
        printf("(29-32) Byte Rate: %u BitRate: %u\n",cab -> ByteRate,cab -> ByteRate*8);
        printf("(33-34) Block Align: %u\n",cab -> BlockAlign);
        printf("(35-36) Bits Per Sample: %u\n",cab -> BitsPerSample);
        printf("(37-40) SubChunk 2 ID: %s\n",cab -> SubChunk2ID);
        printf("(41-44) SubChunk 2 Size: %u\n",cab -> SubChunk2Size);
}

float * generaImpulso ()
{
        float * impulso = (float *) malloc (sizeof (float) * TAM_ARREGLO);
```

```
	for (i = 0; i < TAM_ARREGLO; i ++)
	{
		impulso [i] = (exp ((-2 * PI * FREC_CORTE * i) / 44100));
	//F□rmula de filtro RC con fc = 1,000 Hz
	}
	return impulso;
}

float convolucion (float * entrada, float * impulso)
{
	float respuesta = 0;

	//Realizamos la convoluci□n
	for (i = 0; i < TAM_ARREGLO; i ++)
		if (entrada [i] != 0)
			respuesta += (entrada [i] * impulso [i]);

	//Como entran valores entre -1 y 1, comprobamos la respuesta, para saber que
enviamos (aun as□ todos los multiplicamos por 32767 que es
	//el valor m□ximo que puede tomar un short)
	if (respuesta > 1)
		return (1 * 32767);
	if (respuesta < -1)
		return (-1 * 32767);
	else
		return (respuesta * 32767);
}
```

Cabecera.h

```
typedef struct CABECERA
{
	char ChunkID[4];				//Contiene las 'RIFF'
	int ChunkSize;					//Contiene el
tama□o total sin contar este y el segmento anterior (8 bytes)
	char Format[4];					//Contiene
'WAVE'

	//Aqui comienza el primer subchunk 'fmt'
	char SubChunk1ID[4];			//Contiene 'fmt'
	int SubChunk1Size;				//Contiene el tama□o del
resto de el primer subchunk
	short AudioFormat;				//Formato de audio, es es
distinto de 1, es forma de compresi□n
	short NumChannels;				//Numero de canales, mono
= 1, estereo = 2, etc.
	int SampleRate;					//8000, 44100,
etc.
	int ByteRate;					//(SampleRate *
Numero canales * Bits per Sample) / 8
	short BlockAlign;				//(Numero canales * Bits
per Sample) / 8
	short BitsPerSample;			//8 bits, 16 bits, etc.

	//Aqui comienza el segundo subchunk 'data'
	char SubChunk2ID[4];			//Contiene 'data'
	int SubChunk2Size;				//Numero de bytes en los
datos, es decir, bytes despues de este segmento
}cabecera;

FILE * abre_archivo (char * entrada, char * salida, int tipo);
void copiar_cabecera (FILE * archivoEntrada, FILE * archivoSalida, cabecera * cab);
void imprimir_cabecera (cabecera * cab);
float * generaImpulso ();
float convolucion (float * entrada, float * impulso);
```