INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

Teoría de Comunicaciones y Señales

"Transformada Discreta de Fourier"

Resumen

Simulación de un circuito RC que actúa como filtro en la frecuencia haciendo uso de la Transformada Discreta de Fourier para pasar al dominio de la frecuencia y posteriormente realizar una multiplicación que filtre la señal original.

Por:

Joel Mauricio Romero Gamarra

Profesor:

EDUARDO GUTIÉRREZ ALDANA

Diciembre 2017

Índice

Contenido

Introducción:

La Transformada Discreta de Fourier (en lo subsecuente, TDF), sirve para hacer el análisis en frecuencia de una señal, es decir, nos permite representar el espectro de una señal de forma discreta tomando N muestras y genera N coeficientes. [1]

La TDF se define de la siguiente manera:

$$X\,[k] = \sum_{n=0}^{N-1} x\,[n] \cdot e^{-\frac{2k\pi nj}{N}}, \qquad k = 0, 1, 2, \dots, N-1$$

Y la TDF Inversa como sigue:

$$x\,[n] = \frac{1}{N} \cdot \sum_{k=0}^{N-1} X\,[k] \cdot e^{\frac{2k\pi nj}{N}}, \qquad n = 0, 1, 2, \dots, N-1$$

Es importante notar que los N coeficientes de la TDF (en el mundo discreto) son en realidad muestras de la Transformada de Fourier (en el mundo continuo), con un espaciado (obviamente en frecuencia) de $\Delta f = \frac{1}{N}$ y cubren de forma completa un periodo de la Transformada de Fourier. [1]

La TDF tiene la propiedad de ser periódica, es decir, que X [k + N] = X [k], $0 \leq k \leq N$.

Esto quiere decir, que, aunque la TDF siempre (o casi siempre) se calcula en el intervalo de 0 a N – 1, se puede calcular para cualquier valor de k ya que es periódica con periodo N. [1]

Otra de las propiedades con las que cuenta es la propiedad de linealidad, es decir que, si tenemos 2 señales de distinta duración, al sumarlas esa señal resultante tendrá la duración del mayor entre la señal 1 y la señal 2. Además, la TDF de esa señal se define como:

$$x_3\,[n] = x_1\,[n] + x_2\,[n] \qquad \text{y} \qquad X_3\,[k] = X_1\,[k] + X_2\,[k]$$

Como podemos observar en las ecuaciones de arriba, el cálculo para 1 solo de los coeficientes de la TDF es un algoritmo muy costoso, ya que por cada coeficiente que queremos calcular debemos recorrer las N muestras que tengamos, y, aunque la computadora procese todo muy rápido, si llegamos a tener una señal (suponiendo), que tenga una frecuencia de muestre de 44, 100 Hz y una duración de 5 segundos, tendríamos en total 220, 500 muestras.

Además, en la fórmula tenemos una exponencial, que representa un número complejo, sin embargo, la computadora no conoce de números complejos y hay que separarlo en la parte real y la parte imaginaria, es decir, convertir el número complejo de la forma de Euler a una suma de senos y cosenos.

$$e^{-\frac{2k\pi nj}{N}} = \cos\left(\frac{2k\pi n}{N}\right) - \sin\left(\frac{2k\pi n}{N}\right)$$

Para ver el cálculo de los coeficientes de una manera un poco más simple, se utilizó una prueba de escritorio en Excel para señales con una frecuencia de muestre de 2,000 Hz y una duración de 0.016 segundos, es decir, 32 muestras, una de las señales a las que se les aplicó la TDF es:

$$\cos\big(2 \cdot \pi \cdot f \cdot t \cdot (62.5)\big), \qquad f = 1, 2, 3, \dots, 31$$

Al tener 32 muestras, debemos saber que frecuencia le corresponde a cada muestra, para esto, dividimos la frecuencia que tenemos (2, 000 Hz) entre el número de muestras, y nos queda 62.5, sin embargo, como estamos variando un 62.5 en la frecuencia del coseno, quiere decir que cuando la f sea 1, un pico (de los 2 que muestra el resultado) debe estar en la primera muestra, cuando f sea 2 entonces uno de esos picos debe estar en la segunda muestra y así sucesivamente. A continuación, se muestran algunos ejemplos de la prueba de escritorio realizada para la señal anterior.

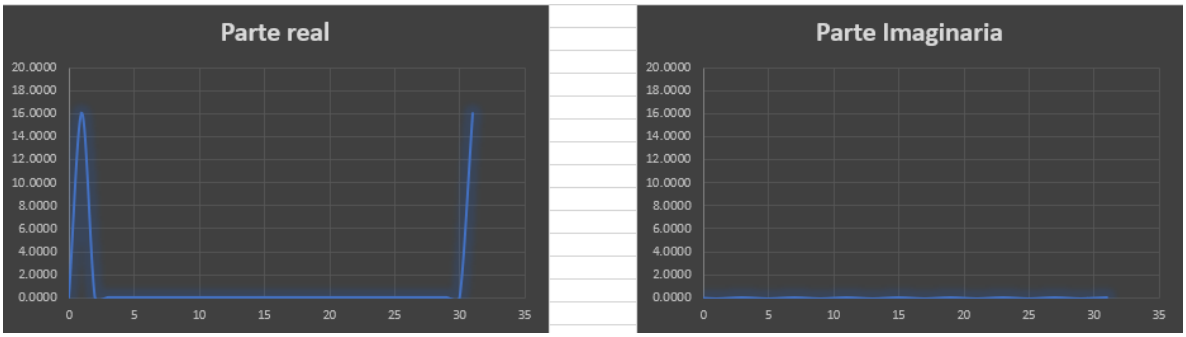

Figura 1. TDF con f = 1

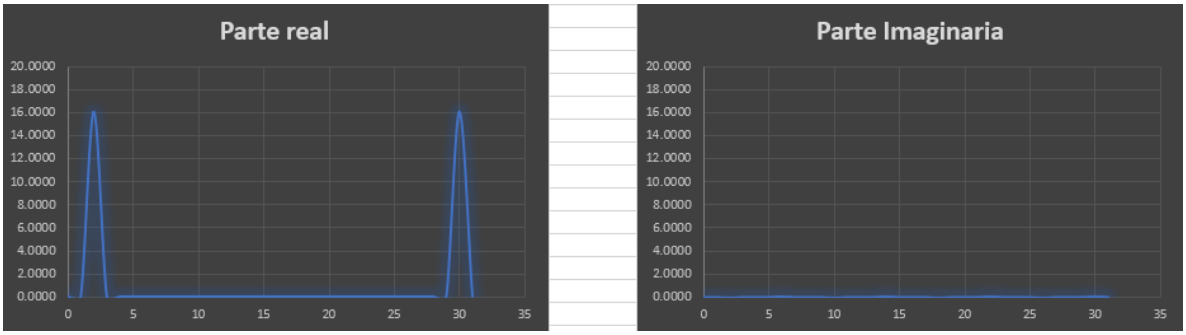

Figura 2. TDF con f = 2

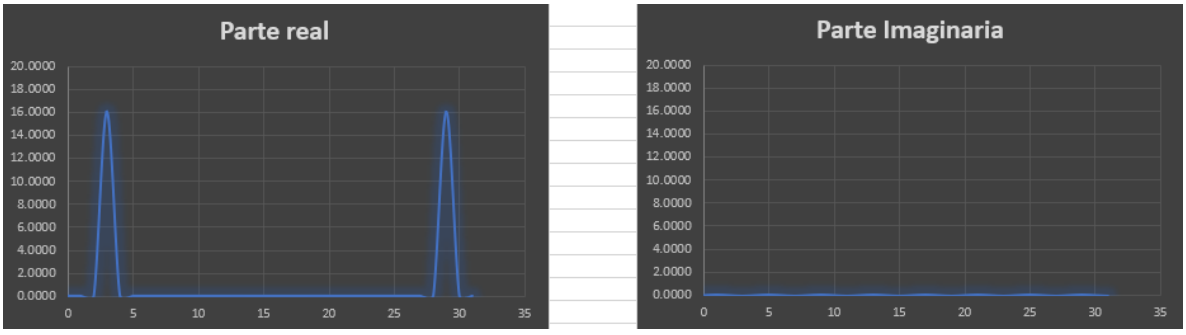

Figura 3. TDF con f = 3

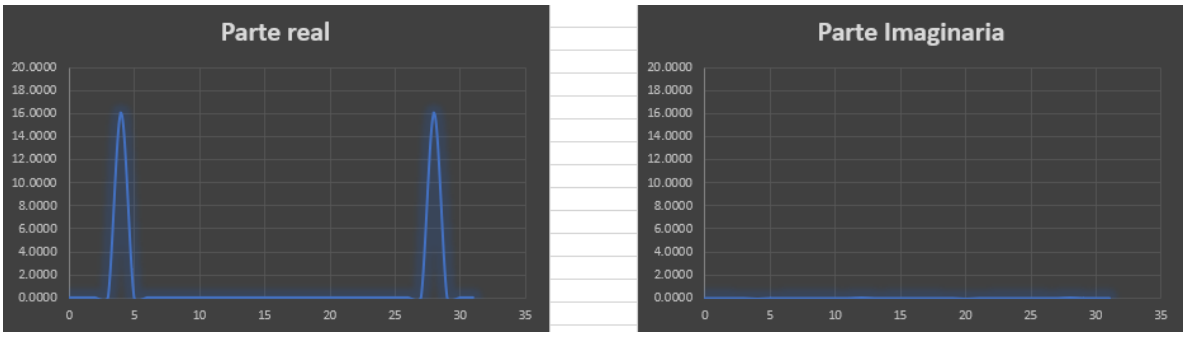

Figura 4. TDF con f = 4

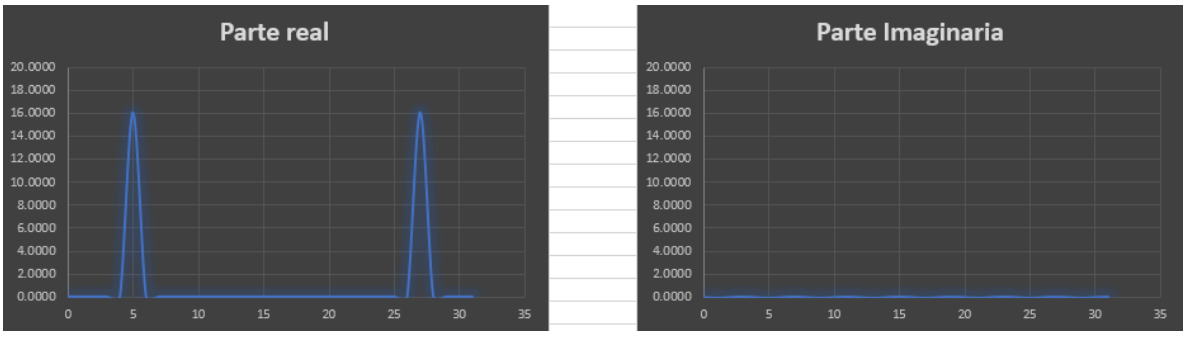

Figura 5. TDF con f = 5

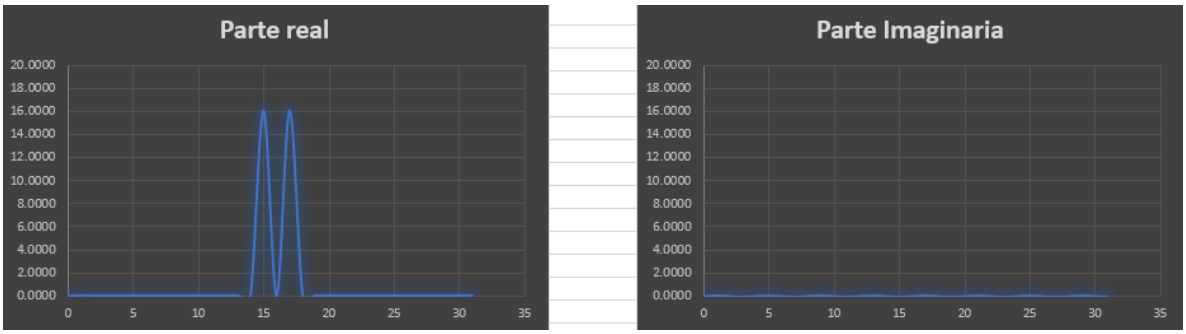

Figura 6. TDF con f = 15

Como se puede apreciar, los picos se van acercando, en la Figura 6 podemos apreciar una frecuencia de 15 * 62.5 = 937.5. Está a 1 muestra más de llegar a la mitad de la frecuencia de muestreo, también es interesante apreciar que los picos cuentan con una magnitud de aproximadamente 16, como sabemos que cada muestra tiene una frecuencia de 62.5, cuando lleguemos a la muestra número 16, la frecuencia será 1,000 Hz (la mitad de la frecuencia de muestreo), esto quiere decir que dejarán de haber 2 picos y se convertirán en 1, como muestra la Figura 7.

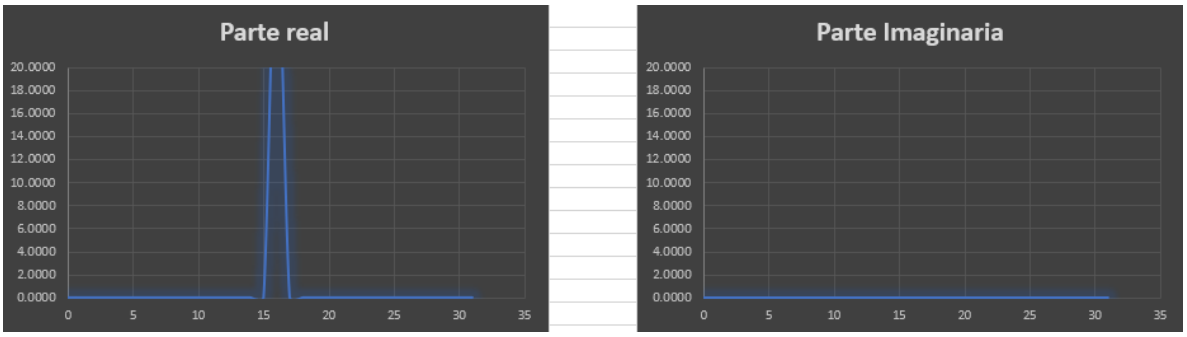

Figura 7. TDF con f = 16

Como se puede ver, el pico sale de la gráfica debido a la escala, ya que en vez de magnitud 16 como los otros, este pico tiene una magnitud de 32 (el doble), y conforme seguimos aumentando la frecuencia de la señal (variando el valor de f), es decir, ahora continuamos de 17 en adelante, los picos vuelven a separarse y se puede observar algo interesante, y esto es, que, las muestras de la TDF de 1 < k < (N/2) son el complejo conjugado de las muestras (N/2) < k < N – 1.

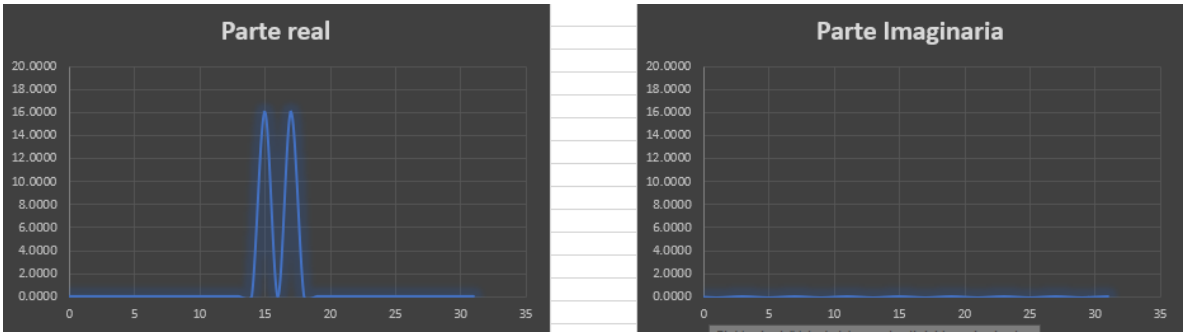

Figura 8. TDF con f = 17

Así que podemos ver que la TDF con f = 15 es igual a la TDF con f = 17, esto es llamado el efecto alias, y por ello, la frecuencia de muestreo debe ser, por lo menos, el doble de la frecuencia de la señal, a esto se le conoce como el teorema del muestreo y sirve para evitar el efecto alias. [2]

Podemos separar en 2 partes debido a las propiedades de los números complejos, sabemos que:

$$X\,[k] = \sum_{n=0}^{N-1} x\,[n] \cdot e^{-\frac{2k\pi nj}{N}} \quad \text{y} \quad e^{-\frac{2k\pi nj}{N}} = \cos(\frac{2k\pi n}{N}) - \sin(\frac{2k\pi n}{N})$$

Así que:

$$X\,[k] = \sum_{n=0}^{N-1} x\,[n] \cdot \cos(\frac{2k\pi n}{N}) - \sin(\frac{2k\pi n}{N})$$

Por lo tanto:

$$Re\,\{X\,[k]\} = \sum_{n=0}^{N-1} x\,[n] \cdot \cos(\frac{2k\pi n}{N}) \quad \text{y} \quad Im\,\{X\,[k]\} = -\sum_{n=0}^{N-1} x\,[n] \cdot \sin(\frac{2k\pi n}{N})$$

En la siguiente sección, se explica el algoritmo utilizado para la programación de la TDF en lenguaje C y un análisis de complejidad por bloques para encontrar una cota que describa al algoritmo y ver el tiempo que tarda aproximadamente para un numero de muestras N (tamaño del problema). [3]

Análisis Teórico:

Como ya vimos en la introducción, la TDF se separa como una suma de cosenos para la parte real y una suma de senos para la parte imaginaria, el código para resolverlo se muestra a continuación:

```
for (k = 0; k < muestras; k ++)
{
          //Se resetean los valores de la parte real e imaginaria en 0
          parte_real = 0;
          parte_imaginaria = 0;
          for (n = 0; n < muestras; n ++)
          {
                    //Se calcula el angulo definido
                    angulo = ((2 * PI * k * n) / muestras);

                    //Se calculan las partes real e imaginaria de la TDF
                    parte_real += (signal [n] * (cos (angulo)));
                    parte_imaginaria += (signal [n] * sin (angulo));
          }
          //Se vuelven a dividir los coeficientes de la TDF para tener valores entre 0 y 1
          parte_real = (parte_real / muestras);
          parte_imaginaria = (parte_imaginaria / muestras);

          //Volvemos a dimensionar los valores de la se�al a escribir
          real [k] = (parte_real * max);
          imaginario [k] = (parte_imaginaria * -1 * max);
}
```

Para analizar el algoritmo, debemos comenzar por la parte más interna, es decir a las líneas que se encuentran dentro del for más interno, y observamos que son 3 líneas estáticas, así que les asignamos un costo de 1 a cada una, por lo tanto, el costo de las 3 líneas hasta el momento es O (1), debido a que al ser una secuencia de instrucciones debemos quedarnos con la que cueste más, pero todas cuestan lo mismo, es un orden constante y prácticamente no cuesta nada.

Ahora, nos vamos al ciclo y vemos que recorre todas las muestras (es decir, N), así que el orden del ciclo es O (n), y cuando hay un ciclo debemos multiplicar el orden de las instrucciones de adentro por las del ciclo, y nos queda $O(1 \cdot n) = O(n)$.

Ahora, observamos que arriba del ciclo y abajo, hay orden constante de complejidad, y al ser una secuencia de instrucciones nos quedamos con la mayor, y $O(n) > O(1)$, por lo tanto la complejidad de las instrucciones dentro del for más externo es O (n), pero al estar dentro de otro ciclo debemos hacer la multiplicación (y el for externo también recorre todas las muestras (es decir, N)), así que, tenemos: $O(n \cdot n) = O(n^2)$.

El orden del algoritmo de la TDF es $O(n^2)$, que no es tan grave como un O (n!) pero es bastante tardado, a continuación se muestra una tabla con diferente número de muestras y un aproximado de cuanto se tardaría el algoritmo.

Número de muestras	Número de instrucciones
32	1, 024
128	16, 384
1, 024	1, 048, 576
2, 048	4, 194, 304
8, 192	67, 108, 864
44, 100	1, 944, 810, 000

Figura 9. Tabla de complejidad del algoritmo TDF

Como podemos ver, es demasiado tardado, ya que imaginando una frecuencia de muestreo de 44,100 y una duración de 1 segundo, realiza poco más de mil millones de operaciones.

Para poder observar mejor lo que tarda el algoritmo, se tomó el tiempo dentro del código de lo que tarda el algoritmo para archivos con distinto número de muestras y se graficó para ver el comportamiento de un orden de complejidad cuadrado, la gráfica del comportamiento de la TDF se muestra en la Figura 10.

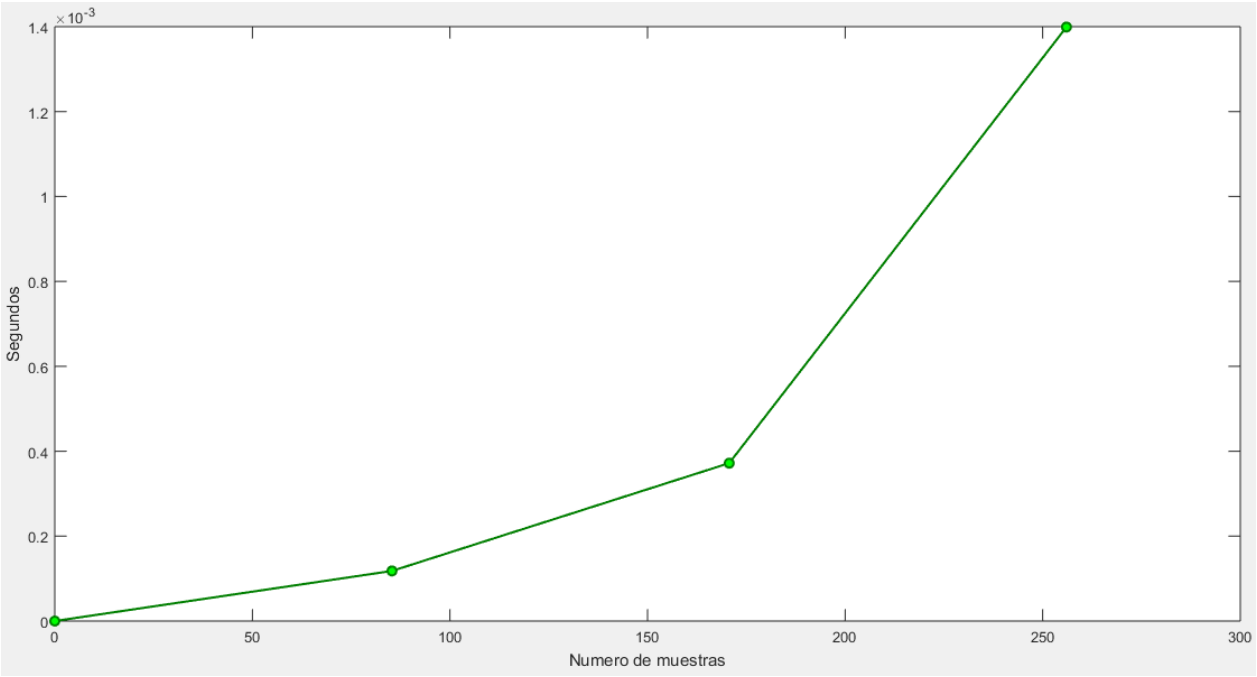

Figura 10. Tiempo de ejecución hasta 128 muestras

Ahora, procederemos a aumentar las muestras desde 0 hasta 1024 para observar el comportamiento de la gráfica en una perspectiva de muestras más grande.

Estas gráficas se muestran a continuación en las Figuras 11 y 12.

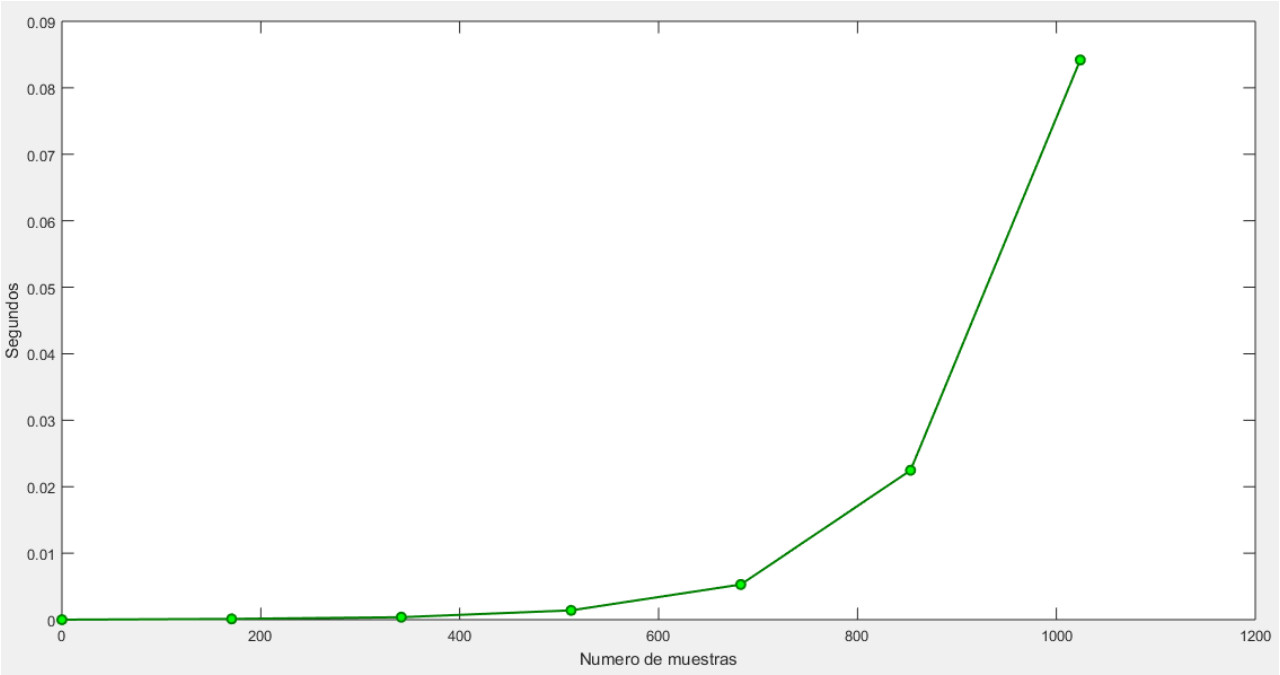

Figura 11. Tiempo de ejecución hasta 1, 024 muestras

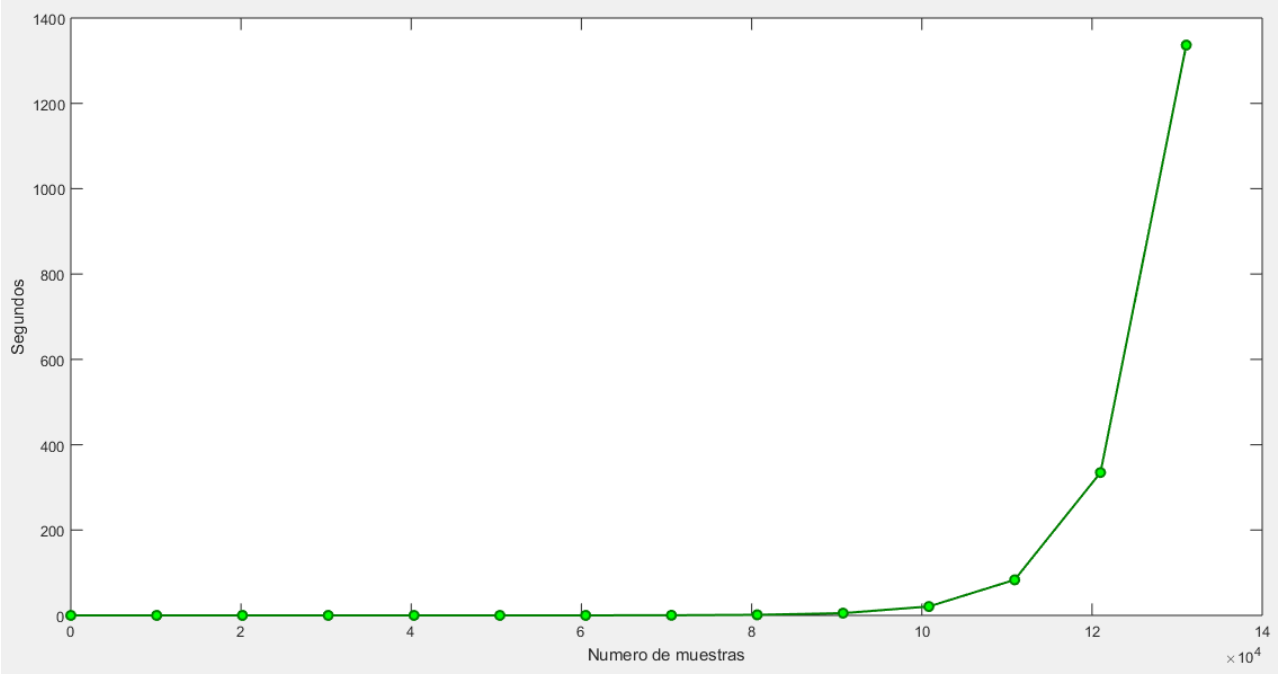

Figura 12. Tiempo de ejecución hasta 131, 072 muestras

Para este programa, se tendrán 3 distintas opciones:

- Mostrar la señal original en el canal izquierdo y la magnitud en el canal derecho.
- Mostrar la parte real en el canal izquierdo y la parte imaginaria en el canal derecho.
- Mostrar la magnitud en el canal izquierdo y la fase en el canal derecho.

Como ya se había explicado en la práctica anterior, podemos ver a un archivo estéreo como un número complejo, así que las fórmulas para calcular tanto la fase como la magnitud las fórmulas son las siguientes:

$$\|z\| = \sqrt{a^2 + b^2}$$

Únicamente debemos tomar las partes real e imaginaria, elevarlas al cuadrado, sumar, y posteriormente calcular la raíz cuadrada, a continuación, se muestra como calcular la fase de la TDF.

$$Arg\ (z) = \arctan\ (\frac{b}{a})$$

Se observa que existe una división, y sabemos que existe una indeterminación en la misma si el denominador es igual a 0, así que, podemos definir a la fase del número complejo como una función por partes separando cada caso distinto que se nos puede presentar, la función por partes que se programará se muestra a continuación:

$$Arg\ (z) = \begin{cases} \arctan\left(\frac{b}{a}\right), & si\ a > 0 \\ \arctan\left(\frac{b}{a}\right) + \pi, & si\ b \geq 0, a < 0 \\ \arctan\left(\frac{b}{a}\right) - \pi, & si\ b < 0, a < 0 \\ \frac{\pi}{2}, & si\ b > 0, a = 0 \\ -\frac{\pi}{2}, & si\ b < 0, a = 0 \\ 0, & si\ b = 0, a = 0 \end{cases}$$

Un dato interesante que es importante saber, es que la señal de entrada (x [n) es una señal mono (con 1 solo canal), sin embargo, la señal resultante de la TDF nos da una señal con una parte real y una parte imaginaria, esto quiere decir que debemos modificar la cabecera para a partir de un archivo mono, generar un archivo estéreo con 2 canales y poder tomar el canal izquierdo y el canal derecho como se define en cada una de las opciones.

Software (librarías, paquetes, herramientas):

- GoldWave versión 4.26 [4]
- Microsoft Excel [5]
- MATLAB R2016a [6]
- Sublime Text 3 [7]

Procedimiento:

Lo primero que debemos hacer es generar un archivo wav con 2 canales para convertirlo en un archivo estéreo y poder meter en cada uno de los canales la opción correspondiente que seleccione el usuario.

En cuanto a la ejecución del programa, algunas pruebas puntuales son presentadas en la sección de resultados, como probar las 3 distintas opciones, y posteriormente usar la opción 2 y regresar a la señal original aplicando la TDF inversa, sin embargo, para hacer la graficación para el orden de complejidad (presentado en la sección de análisis teórico) donde se analizó el algoritmo de la TDF, se utilizó un Script en Linux.

Se realizaron las pruebas para una señal coseno con distinto número de pruebas a una frecuencia de muestreo de 2, 000 Hz variando únicamente la duración (es decir, duplicar la duración, comenzando por 0.016 hasta 65.535 segundos) y utilizando únicamente la opción 2 (ya que calcular la magnitud y fase consume un poco más de operaciones, y solo estamos analizando el algoritmo de la transformada en sí).

SCRIPT UTILIZADO:

```
#!/bin/bash

echo 'TIEMPOS CON LA TRANSFORMADA DISCRETA DE FOURIER' >> Tiempos.txt

cd /home/joel/Escritorio/Se◆ales/Transformada

gcc TDF.c -o TDF -lm

./TDF -2 Coseno32.wav TDF_Coseno32.wav >> Tiempos.txt
./TDF -2 Coseno64.wav TDF_Coseno64.wav >> Tiempos.txt
./TDF -2 Coseno128.wav TDF_Coseno128.wav >> Tiempos.txt
./TDF -2 Coseno256.wav TDF_Coseno256.wav >> Tiempos.txt
./TDF -2 Coseno512.wav TDF_Coseno512.wav >> Tiempos.txt
./TDF -2 Coseno1024.wav TDF_Coseno1024.wav >> Tiempos.txt
./TDF -2 Coseno2048.wav TDF_Coseno2048.wav >> Tiempos.txt
./TDF -2 Coseno4096.wav TDF_Coseno4096.wav >> Tiempos.txt
./TDF -2 Coseno8192.wav TDF_Coseno8192.wav >> Tiempos.txt
./TDF -2 Coseno16384.wav TDF_Coseno16384.wav >> Tiempos.txt
./TDF -2 Coseno32768.wav TDF_Coseno32768.wav >> Tiempos.txt
./TDF -2 Coseno65536.wav TDF_Coseno65536.wav >> Tiempos.txt
./TDF -2 Coseno131072.wav TDF_Coseno131072.wav >> Tiempos.txt
```

Como se puede ver, el script realizado es bastante simple, y se puede ver que los tiempos fueron escritos en un archivo y posteriormente tomar esos tiempos de ejecución en segundos en el archivo y hacer la graficación.

FUNCIÓN PARA MEDIR EL TIEMPO DE EJECUCIÓN: [8]

```
void uswtime(double *usertime, double *walltime)
{
          double mega = 1.0e-6;
          struct rusage buffer;
          struct timeval tp;
          struct timezone tzp;
          getrusage(RUSAGE_SELF, &buffer);
          gettimeofday(&tp, &tzp);
          *usertime = (double) buffer.ru_utime.tv_sec +1.0e-6 * buffer.ru_utime.tv_usec;
          *walltime = (double) tp.tv_sec + 1.0e-6 * tp.tv_usec;
}
```

Resultados

Las primeras pruebas para mostrar las 3 distintas opciones de la práctica se realizarán con funciones sinusoidales, todas a una frecuencia de muestreo de 2, 000 Hz y una duración de 0.016 segundos. En la Figura 13 se muestra la opción número 1 para la siguiente función:

$$\sin\left(2 \cdot \pi \cdot f \cdot t \cdot (62.5)\right), \qquad con\ f = 5$$

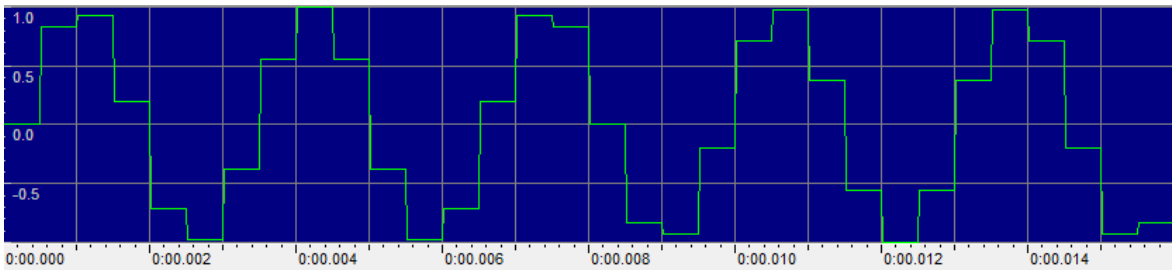

Figura 13. Señal original

A continuación, se ejecuta la TDF en la opción 1, que muestra la señal original en el canal izquierdo y la magnitud en el canal derecho.

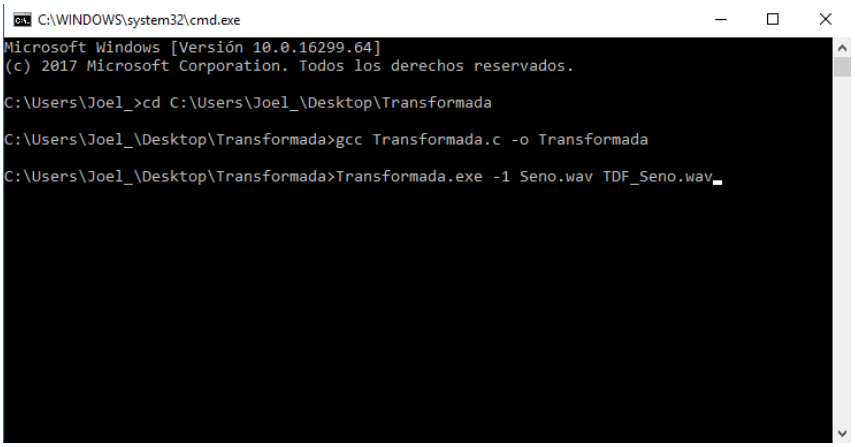

Figura 14. Compilación y ejecución con opción 1

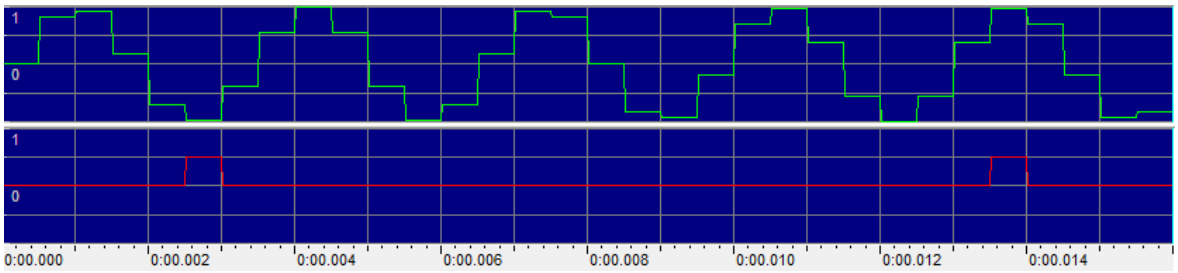

Figura 15. Señal original y Magnitud TDF

A continuación, se muestra la ejecución de la TDF para la misma señal en la opción 2.

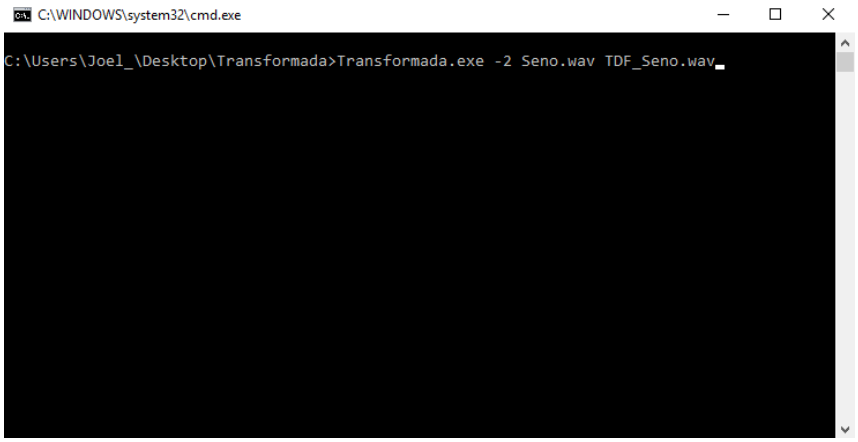

Figura 16. Ejecución con opción 2

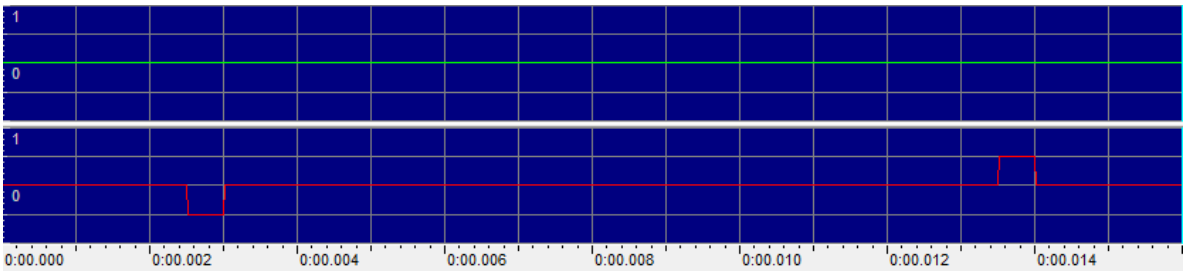

Figura 17. Parte real y parte imaginaria TDF

A continuación, se muestra la ejecución de la TDF para la misma señal en la opción 3.

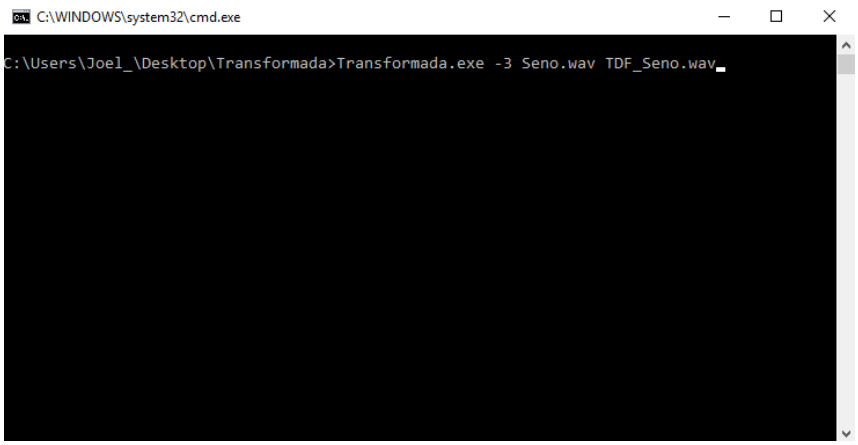

Figura 18. Ejecución con opción 3

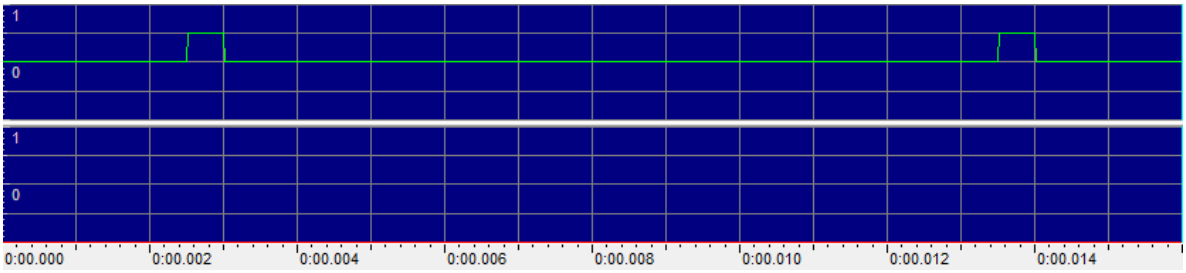

Figura 19. Magnitud y fase TDF

Ahora, generamos un tren de impulsos para poder apreciar mejor la respuesta cuando se filtre la señal en la frecuencia (como en la práctica de convolución).

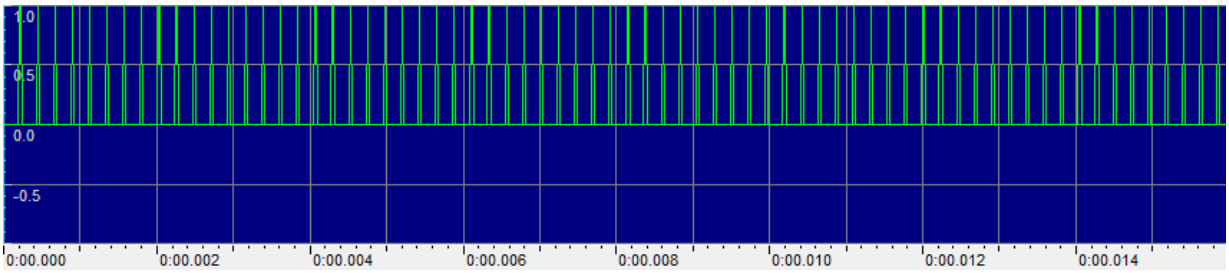

Figura 20. Señal original

A continuación, se le aplica la TDF en la opción 2 para mostrar la parte real y la parte imaginaria y posteriormente filtrar la señal con una multiplicación en frecuencia.

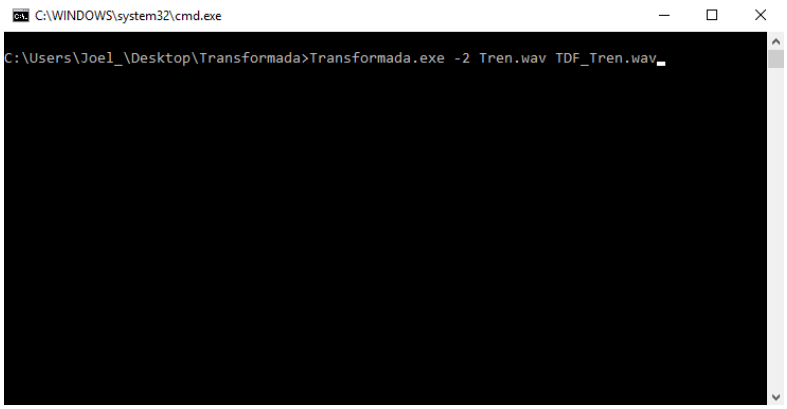

Figura 21. Ejecución con la opción 2

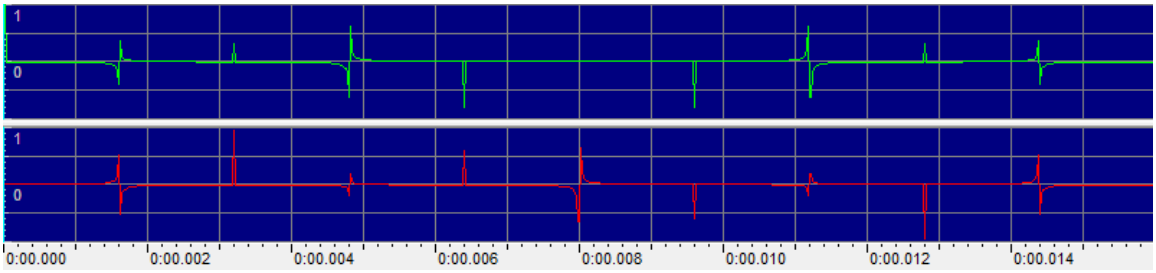

Figura 22. TDF Tren de Impulsos

Ahora, si multiplicamos la respuesta al impulso con el tren de impulsos obtenemos la siguiente señal:

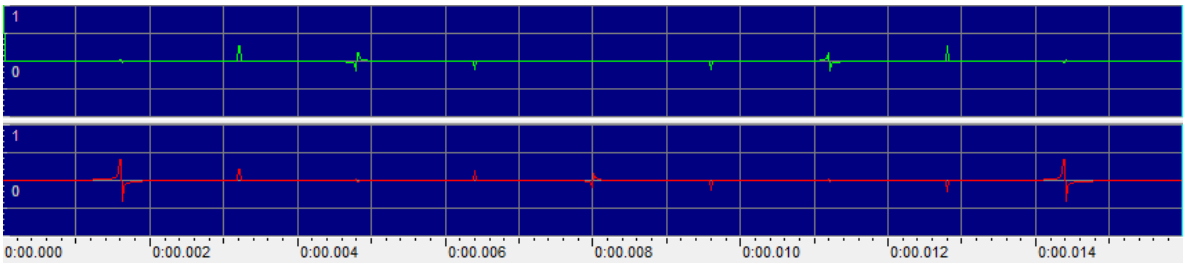

Figura 23. Multiplicación en frecuencia

Y si ahora aplicamos la TDF inversa a la señal obtenida, nos queda la siguiente señal:

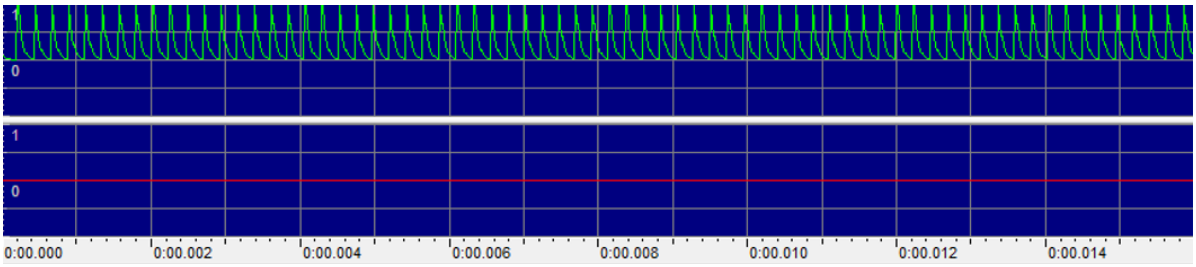

Figura 24. Tren de impulsos filtrado

Para hacer la comparación, observamos lo que pasó en la práctica 2 cuando hicimos la convolución para filtrar un tren de impulsos, y debería ser lo mismo que obtuvimos en la parte real de la TDF.

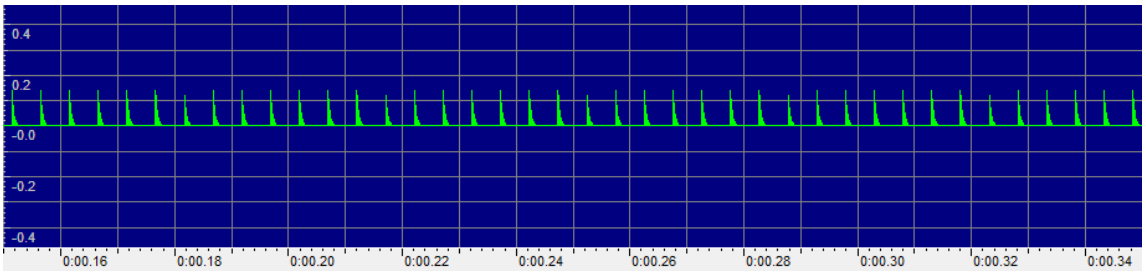

Figura 25. Filtrado de tren de impulsos con convolución

Discusión:

La Transformada Discreta de Fourier tiene distintas aplicaciones, sin embargo, al igual que la convolución, es un algoritmo muy costoso, y por ejemplo si quisiéramos filtrar una señal directo desde la frecuencia con una TDF y luego una multiplicación, es más fácil que hacer una convolución. En cuanto a la velocidad, habría que analizar bien ambos, ya que si la convolución se implementa como se hizo (Como un filtro FIR, es decir, Finito), entonces conviene más hacer una TDF y luego una multiplicación, debido a que para la convolución primero hay que llenar todo con 0, luego recorrer todos los elementos, posteriormente multiplicar todos los elementos y sumarlos para obtener un elemento, y todo esto en un ciclo que va de 1 a N donde N son las muestras que forman a la señal.

El tiempo de ejecución y el uso de memoria excesivo es algo que tiene este algoritmo, ya que hay que tener un arreglo con la parte imaginaria, uno con la parte real, y en cuanto a operaciones, sabemos que realizar operaciones a nivel de bit es muy rápido, sin embargo involucrar funciones como coseno, seno o tangente seguramente tardan muchísimo en ejecutarse y obviamente, hacen más lento el programa.

Conclusiones:

Algo interesante es que para cambiar el programa de hacer la TDF a la TDF inversa es únicamente cambiar un signo para la parte imaginaria, y al final dividir entre el número de muestras ambas respuestas, por lo tanto es demasiado simple, prácticamente no hay que hacer muchos cambios.

El algoritmo de la FFT es mucho más rápido realizando menos operaciones que la TDF, sin embargo se analizará con detenimiento en la siguiente práctica, es bastante útil debido a que como ya vimos en la tabla en la sección de análisis teórico y en las gráficas con los tiempos que se tardan para distintos tipos de muestras que para un número de muestras considerable es muy tardado, por ejemplo, en la vida real las frecuencias a las que escuchamos la música está muestreada a 44, 100 Hz, es decir, 44, 100 muestras por segundo, ahora, imaginando que tuviéramos toda una canción a la que queremos aplicarle un filtro (como un ecualizador o algo así por el estilo), tomemos una duración de 3 minutos, es decir, 180 segundos. Tendríamos un total de 7, 938, 000 muestras que es casi 7 veces el número mayor de muestras que utilizamos para graficar.

Otra aplicación, sería reconocer una canción por medio del análisis de frecuencia, como por ejemplo la famosa aplicación Shazam que escucha una canción por un periodo de aproximadamente 10 segundos, detectar las frecuencias de cada muestra para reconocer una canción debe ser un trabajo muy extenso que involucra una transformada de Fourier para analizar las frecuencias y posteriormente "matchear" con las frecuencias de alguna canción ya conocida debe ser un trabajo que se realice en menos de 3 segundos.

Es importante saber cómo optimizar los algoritmos ya conocidos que tienen aplicaciones en la vida real, como por ejemplo, la convolución para filtrar una señal es un algoritmo muy tardado que podemos optimizar haciendo uso de la transformada Z en cuanto a tiempos de ejecución y memoria utilizada ya que necesitamos de únicamente 4 valores, y en este caso para la FFT (Fast Fourier Transform) se utiliza un concepto de diezmado en el tiempo y un arreglo de mariposa para evitar repetir operaciones previamente realizadas. [1]

Referencias

[1] A. A. Colomer, V. Naranjo Ornedo y J. Prades Nebot, Tratamiento Digital de la Señal Teoría y Aplicaciones, México: Limusa, 2009.

[2] G. Carman, «Efecto Aliasing,» Julio 2014. [En línea]. Available: http://grupocarman.com/blog/efecto-aliasing/. [Último acceso: Diciembre 2017].

[3] E. A. F. Martínez, «Análisis de Algoritmos No Recursivos,» Octubre 2017. [En línea]. Available: http://www.eafranco.com/docencia/analisisdealgoritmos/files/05/Tema05.pdf. [Último acceso: Diciembre 2017].

[4] G. W. Inc, «GoldWave Version 4.26 Download,» [En línea]. Available: https://www.goldwave.com/release426.php.

[5] Microsoft, «Office 365,» [En línea]. Available: https://products.office.com/es-mx/products?tab=O-Home.

[6] MathWorks, «R2016a,» [En línea]. Available: https://www.mathworks.com/products/new_products/release2016a.html.

[7] S. Text, «Sublime Text 3 Download,» [En línea]. Available: https://www.sublimetext.com/3.

[8] E. A. F. Martínez, «Pruebas a Posteriori,» Octubre 2017. [En línea]. Available: http://www.eafranco.com/docencia/analisisdealgoritmos/files/practicas/01/Practica01.pdf.

Código

Cabecera.h

```
typedef struct CABECERA
{
          char ChunkID[4];              //Contiene 'RIFF'
          int ChunkSize;                //Contiene el tama□o total del archivo - 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 primer subchunk
          short AudioFormat;            //Formato de audio
          short NumChannels;            //Numero de canales
          int SampleRate;               //Frecuencia de muestreo
          int ByteRate;                 //(SampleRate * Numero canales * Bits per Sample) / 8
          short BlockAlign;             //Bytes por muestra
          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
}cabecera;

void imprimir_cabecera (cabecera * cab);
void opcion_uno (FILE * salida, short * signal, cabecera * cab);
void opcion_dos (FILE * salida, short * signal, cabecera * cab);
void opcion_tres (FILE * salida, short * signal, cabecera * cab);
void uswtime (double * usertime, double * walltime);
void calculaTiempo (double utime0, double wtime0, double utime1, double wtime1, int n, int
opcion);
```

Cabecera.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/resource.h>
#include <sys/time.h>
#include "Cabecera.h"
#define PI 3.14159265

double utime0, wtime0, utime1, wtime1;

void imprimir_cabecera (cabecera * cab)
{
	char * formatoArchivo = (char *) malloc (sizeof (char));
	printf ("(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);
}

void opcion_uno (FILE * salida, short * signal, cabecera * cab)
{
	int i, k, n;
	float max = 32767, muestras = (cab -> SubChunk2Size / cab -> BlockAlign);
	double real, imaginario, angulo;
	short parte_real, parte_imaginaria, original, magnitud;
	fseek (salida, 44, SEEK_SET);
	uswtime (&utime0, &wtime0);
	for (k = 0; k < muestras; k ++)
	{
		real = 0;
		imaginario = 0;
		original = signal [k];
		for (n = 0; n < muestras; n ++)
		{
			angulo = ((2 * PI * k * n) / muestras);
			real += (signal [n] * (cos (angulo)));
			imaginario -= (signal [n] * sin (angulo));
		}
		parte_real = (real / muestras);
		parte_imaginaria = (imaginario / muestras);
		magnitud = (short) sqrt (pow (parte_real, 2) + pow (parte_imaginaria,
2));
		fwrite (&original, sizeof (short), 1, salida);
		fwrite (&magnitud, sizeof (short), 1, salida);
	}
	uswtime (&utime1, &wtime1);
	calculaTiempo (utime0, wtime0, utime1, wtime1, muestras, 1);
}
```

```
void opcion_dos (FILE * salida, short * signal, cabecera * cab)
{
          int i, k, n;
          float max = 32767, muestras = (cab -> SubChunk2Size / cab -> BlockAlign);
          double real, imaginario, angulo;
          short parte_real, parte_imaginaria;
          fseek (salida, 44, SEEK_SET);
          uswtime (&utime0, &wtime0);
          for (k = 0; k < muestras; k ++)
          {
                    real = 0;
                    imaginario = 0;
                    for (n = 0; n < muestras; n ++)
                    {
                              angulo = ((2 * PI * k * n) / muestras);
                              real += (signal [n] * (cos (angulo)));
                              imaginario -= (signal [n] * sin (angulo));
                    }
                    parte_real = (real / muestras);
                    parte_imaginaria = (imaginario / muestras);
                    fwrite (&parte_real, sizeof (short), 1, salida);
                    fwrite (&parte_imaginaria, sizeof (short), 1, salida);
          }
          uswtime (&utime1, &wtime1);
          calculaTiempo (utime0, wtime0, utime1, wtime1, muestras, 2);
}

void opcion_tres (FILE * salida, short * signal, cabecera * cab)
{
          int i, k, n;
          float max = 32767, muestras = (cab -> SubChunk2Size / cab -> BlockAlign);
          double real, imaginario, angulo;
          short parte_real, parte_imaginaria, fase, magnitud;
          fseek (salida, 44, SEEK_SET);
          uswtime (&utime0, &wtime0);
          for (k = 0; k < muestras; k ++)
          {
                    real = 0;
                    imaginario = 0;
                    for (n = 0; n < muestras; n ++)
                    {
                              angulo = ((2 * PI * k * n) / muestras);
                              real += (signal [n] * (cos (angulo)));
                              imaginario -= (signal [n] * sin (angulo));
                    }
                    parte_real = (real / muestras);
                    parte_imaginaria = (imaginario / muestras);
                    magnitud = (short) sqrt (pow (parte_real, 2) + pow (parte_imaginaria,
2));
                    if (magnitud == 0)
                              fase = 0;
                    else
                    {
                              if ((parte_real == 0) && (parte_imaginaria < 0))
                                        fase = (short) (- PI / 2);
                              else if ((parte_real == 0) && (parte_imaginaria > 0))
                                        fase = (short) (PI / 2);
                              else if ((parte_real < 0) && (parte_imaginaria >= 0))
                                        fase = (short) (atan (parte_imaginaria / parte_real)
+ PI);
                              else if ((parte_real < 0) && (parte_imaginaria < 0))
                                        fase = (short) (atan (parte_imaginaria / parte_real)
- PI);
                    }
                    fwrite (&magnitud, sizeof (short), 1, salida);
                    fwrite (&fase, sizeof (short), 1, salida);
          }
          uswtime (&utime1, &wtime1);
          calculaTiempo (utime0, wtime0, utime1, wtime1, muestras, 3);
}
```

```
void uswtime (double * usertime, double * walltime)
{
          double mega = 1.0e-6;
          struct rusage buffer;
          struct timeval tp;
          struct timezone tzp;
          getrusage (RUSAGE_SELF, &buffer);
          gettimeofday (&tp, &tzp);
          *usertime = (double) buffer.ru_utime.tv_sec +1.0e-6 * buffer.ru_utime.tv_usec;
          *walltime = (double) tp.tv_sec + 1.0e-6 * tp.tv_usec;
}

void calculaTiempo (double utime0, double wtime0, double utime1, double wtime1, int n, int
opcion)
{
          char * option = (char *) malloc (sizeof (char));
          switch (opcion)
          {
                    case 1:
                              strcpy (option, "Opcion 1");
                              break;
                    case 2:
                              strcpy (option, "Opcion 2");
                              break;
                    case 3:
                              strcpy (option, "Opcion 3");
                              break;
          }
          printf("Tiempo con la %s para un archivo de %d muestras:\n\n", option, n);
          printf("Tiempo total:  %.10f s\n",  wtime1 - wtime0);
          printf("Tiempo de procesamiento en CPU: %.10f s\n",  utime1 - utime0);
          printf
("_____
_____");
}
```

TDF.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "Cabecera.h"

int main(int argc, char const *argv[])
{
	FILE * entrada, * salida;
	cabecera cab;
	int opcion_seleccionada, i, numero_muestras;
	char * archivo_salida = (char *) malloc (sizeof (char));
	char * archivo_entrada = (char *) malloc (sizeof (char));
	if (argc < 4)
	{
		printf("Error, faltan argumentos.\n");
		printf ("Ejemplo: '%s -1 Archivo1.wav Salida.wav'\n\n", argv [0]);
		exit (0);
	}else
	{
		opcion_seleccionada = atoi (argv [1]);
		archivo_entrada = (char *) argv [2];
		archivo_salida = (char *) argv [3];
	}

	//Abrimos los archivos en modo binario
	entrada = fopen (archivo_entrada, "rb");
	if (entrada == NULL)
	{
		printf ("Error al abrir el archivo: '%s'", archivo_entrada);
		exit (0);
	}
	salida = fopen (archivo_salida, "wb");
	if (salida == NULL)
	{
		printf ("Error al crear el archivo: '%s'", archivo_salida);
		exit (0);
	}

	//Leer la cabecera del archivo de entrada
	fread (&cab, 44, 1, entrada);

	numero_muestras = (cab.SubChunk2Size / cab.BlockAlign);

	//Modificar algunos par?metros de la cabecera
	cab.ChunkSize = (cab.ChunkSize * 2);
	cab.NumChannels = (short) 2;
	cab.ByteRate = (cab.ByteRate * 2);
	cab.BlockAlign = (short) 4;
	cab.BitsPerSample = (short) 16;
	cab.SubChunk2Size = (cab.SubChunk2Size * 2);

	//Copiar la cabecera modificada al archivo de salida
	fwrite (&cab, 44, 1, salida);

	//Imprimir los valores de la cabecera
	imprimir_cabecera (&cab);

	short * signal = (short *) malloc (sizeof (short) * numero_muestras);
	printf ("\n\n");
	for (i = 0; i < numero_muestras; i ++)
		fread (&signal [i], sizeof (short), 1, entrada);

	if (opcion_seleccionada == -1)
		opcion_uno (salida, signal, &cab);
	else if (opcion_seleccionada == -2)
		opcion_dos (salida, signal, &cab);
	else if (opcion_seleccionada == -3)
```

```
                opcion_tres (salida, signal, &cab);
        else
        {
                printf ("\nOpcion invalida");
                exit (0);
        }
        fclose (entrada);
        fclose (salida);
        return 0;
}
```

Cabecera.h (Inversa)

```
typedef struct CABECERA
{
        char ChunkID[4];            //Contiene 'RIFF'
        int ChunkSize;                      //Contiene el tama□o total del archivo - 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 primer subchunk
        short AudioFormat;          //Formato de audio
        short NumChannels;          //Numero de canales
        int SampleRate;                     //Frecuencia de muestreo
        int ByteRate;                       //(SampleRate * Numero canales * Bits per
Sample) / 8
        short BlockAlign;           //Bytes por muestra
        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
}cabecera;

void imprimir_cabecera (cabecera * cab);
void transformada_inversa (FILE * salida, short * re, short * im, cabecera * cab);
void uswtime (double * usertime, double * walltime);
void calculaTiempo (double utime0, double wtime0, double utime1, double wtime1, int n);
```

Cabecera.c (Inversa)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/resource.h>
#include <sys/time.h>
#include "Cabecera.h"
#define PI 3.14159265

double utime0, wtime0, utime1, wtime1;

void imprimir_cabecera (cabecera * cab)
{
        char * formatoArchivo = (char *) malloc (sizeof (char));
        printf ("\n\n");
        printf ("(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);
}

void transformada_inversa (FILE * salida, short * re, short * im, cabecera * cab)
{
          int k, n, numero_muestras;
          short parte_real, parte_imaginaria;
          double real, imaginario, angulo;
          numero_muestras = (cab -> SubChunk2Size / cab -> BlockAlign);
          fseek (salida, 44, SEEK_SET);
          uswtime (&utime0, &wtime0);
          for (n = 0; n < numero_muestras; n ++)
          {
                    real = 0;
                    imaginario = 0;
                    for (k = 0; k < numero_muestras; k ++)
                    {
                              angulo = ((2 * PI * k * n) / numero_muestras);
                              real += ((re [k] * cos (angulo)) - (im [k] * sin (angulo)));
                              imaginario += ((im [k] * cos (angulo)) + (re [k] * sin (angulo)));
                    }
                    parte_real = (real / numero_muestras);
                    parte_real = (parte_real * numero_muestras);
                    parte_imaginaria = (imaginario / numero_muestras);
                    parte_imaginaria = (parte_imaginaria * numero_muestras);
                    fwrite (&parte_real, sizeof (short), 1, salida);
                    fwrite (&parte_imaginaria, sizeof (short), 1, salida);
          }
          uswtime (&utime1, &wtime1);
          calculaTiempo (utime0, wtime0, utime1, wtime1, numero_muestras);
}

void uswtime (double * usertime, double * walltime)
{
          double mega = 1.0e-6;
          struct rusage buffer;
          struct timeval tp;
          struct timezone tzp;
          getrusage (RUSAGE_SELF, &buffer);
          gettimeofday (&tp, &tzp);
          *usertime = (double) buffer.ru_utime.tv_sec +1.0e-6 * buffer.ru_utime.tv_usec;
          *walltime = (double) tp.tv_sec + 1.0e-6 * tp.tv_usec;
}

void calculaTiempo (double utime0, double wtime0, double utime1, double wtime1, int n)
{
          printf("Tiempo con la Transformada Inversa de Fourier para un archivo de %d
muestras:\n\n", n);
          printf("Tiempo total:  %.10f s\n",  wtime1 - wtime0);
          printf("Tiempo de procesamiento en CPU: %.10f s\n",  utime1 - utime0);
```

```
        printf
("_____
_____");
}
```

ITDF.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "Cabecera.h"

int main(int argc, char const *argv[])
{
        FILE * entrada, * salida;
        cabecera cab;
        int i, numero_muestras;
        char * archivo_salida = (char *) malloc (sizeof (char));
        char * archivo_entrada = (char *) malloc (sizeof (char));
        if (argc < 3)
        {
                printf("Error, faltan argumentos.\n");
                printf ("Ejemplo: '%s Entrada.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 = fopen (archivo_entrada, "rb");
        if (entrada == NULL)
        {
                printf ("Error al abrir el archivo: '%s'", archivo_entrada);
                exit (0);
        }
        salida = fopen (archivo_salida, "wb");
        if (salida == NULL)
        {
                printf ("Error al crear el archivo: '%s'", archivo_salida);
                exit (0);
        }

        //Copiar la cabecera del archivo de entrada al de salida
        fread (&cab, 44, 1, entrada);
        fwrite (&cab, 44, 1, salida);

        //Imprimir los valores de la cabecera
        imprimir_cabecera (&cab);

        numero_muestras = (cab.SubChunk2Size / cab.BlockAlign);

        short * re = (short *) malloc (sizeof (short) * numero_muestras);
        short * im = (short *) malloc (sizeof (short) * numero_muestras);

        for (i = 0; i < numero_muestras; i ++)
        {
                fread (&re [i], sizeof (short), 1, entrada);
                fread (&im [i], sizeof (short), 1, entrada);
        }
        transformada_inversa (salida, re, im, &cab);
        fclose (entrada);
        fclose (salida);
        return 0;
}
```